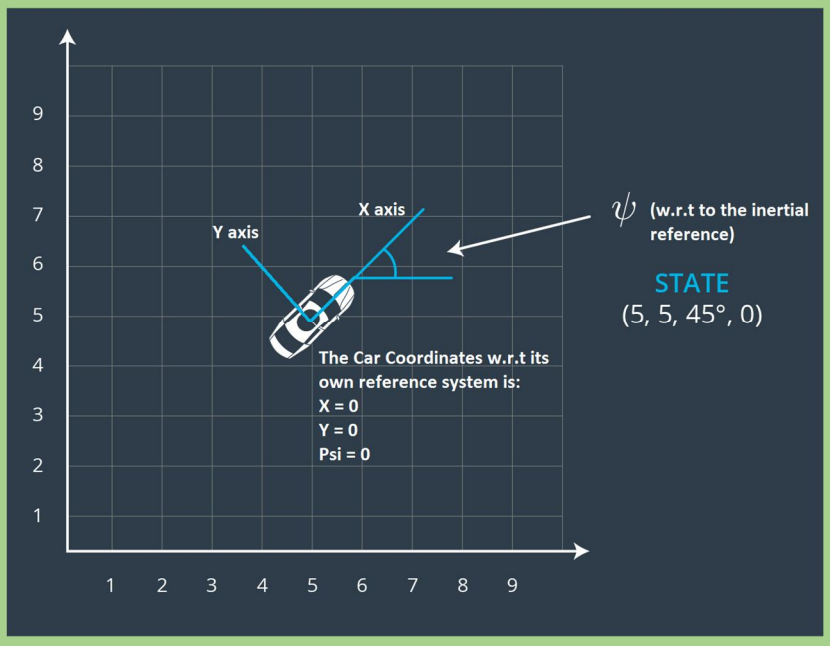
Self-Driving Car ND

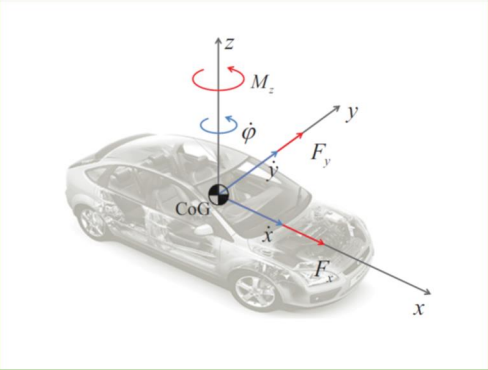
Model Predictive Control

CRITERIA	MEETS SPECIFICATIONS
The Model	Student describes their model in detail. This includes the state, actuators and update equations.

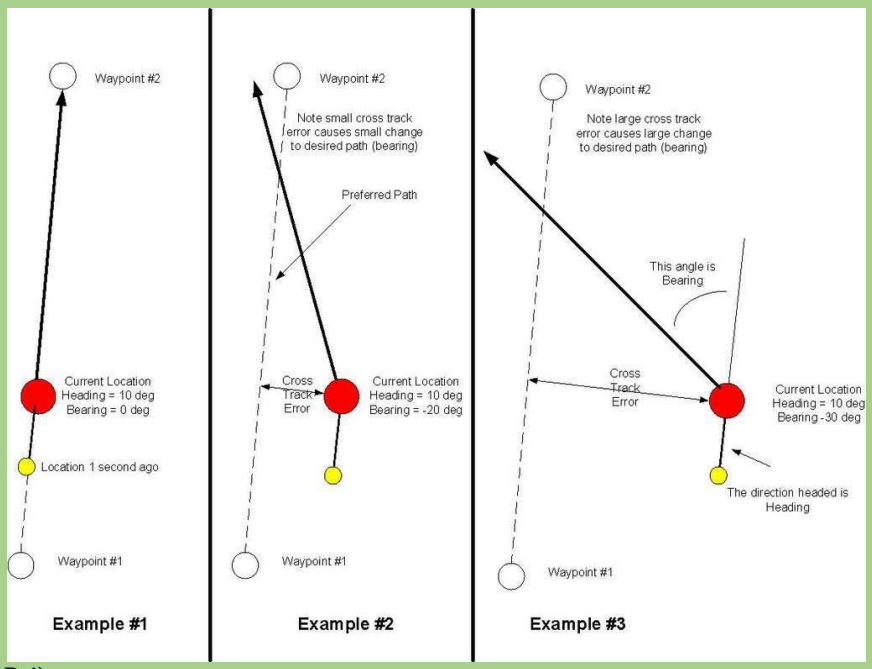
The state of the car is represented by an 8 elements vector:

1. X coordinate (w.r.t any reference systems we are using. In our case and since the waypoints and the MPC trajectory points needed to be on Car system coordinates, we are using this reference system. Also, it turns out that from the Carôs ref system, the car is always at X=Y=Psi=0). The X coordinate axis is pointing out from the car longitudinal axis.
2. Y coordinate (pointing left from the longitudinal axis or the car. See below





3. Psi Coordinate (Heading or Yaw): Angle between the X axis and ñheadingò of the car. The heading is always represented by the line that goes across the longitudinal axis of the car pointing forward.
4. V: The magnitude of the velocity vector (speed).
5. CTE: The Cross Track Error. See below for graphical definition



6. Heading Error (ePsi)
7. Steering command (actuator output): This is the ñAbsoluteò steering angle that the car would be commanded to be at. This command will vary from -1 (right turns) to +1(left turns) which will be converted into degrees between -25 to +25. This is the mechanical and actuator limits.
8. Acceleration command (actuator output): this value also varies from +1 (full throttle) to ï 1(full brake). Any positive value will accelerate the car while a negative value will ñbrakeò the car.

Below are the Model Update equations at time t+1 based on a Bicycle model

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

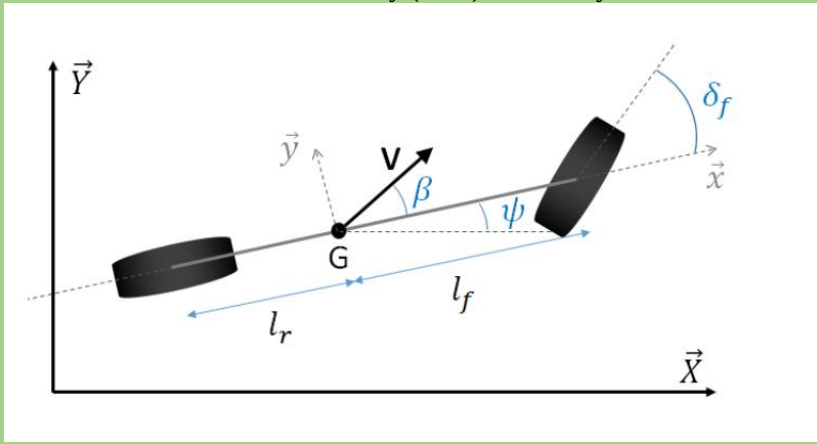
$$v_{t+1} = v_t + a_t * dt$$

$$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$$

$$e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt)$$

Where:

- dt is the discrete time step.
- x, y, psi, v, cte and epsi as described above.
- Lf is the length from the front to the Center Of Gravity (CoG) in our bicycle model. See below



- ŭt is the previous (at time t) steering angle (look above)
- All subscripts ñtò refer to time step t (also referenced as ñpreviousò).
- All subscripts ñt+1ò refer to time step t +1.

Time step Length and Elapsed Duration (N & dt)	Student discusses the reasoning behind the chosen *N* (time step length) and *dt* (elapsed duration between time steps) values. Additionally the student details the previous values tried.

This section could take a whole thesis on its own. So for simplicity Iđll hit some relevant point.

- ñNò represent the ñnumberò of time steps that we will use to form our ñtimeò prediction horizon. In fact N is usually called the ñPrediction Horizonò since the time step ñdtò is usually fixed by the processor limitations and therefore N truly defines how far in time we will predict.

- There is plenty of literature on how to choose the shorter horizons while guarantying stability (ESTIMATES ON THE PREDICTION HORIZON LENGTH IN MODEL PREDICTIVE CONTROL* by K. WORTHMANNÀ, Shorter horizons for model predictive control by S. Emre Tuna, Michael J. Messina, and Andrew R. Teel, etcé) but these studies are outside the scope of this assignment. The truth of the matter is that there are a few formal methods to choose N to be small enough for computational efficiency while large enough to guarantee stability and performance.
- In an oversimplified way, we can say that if N is too large the computational penalty is bigger and also we will be trying to predict too far ahead when in reality it is not necessary since we will apply only the very first step of the prediction. Besides, by the time we get to the end on this far predicted horizon other uncertainties and disturbances will be present and we will have to predict again anyway.
- If N is too small, the dynamics of our system and the latencies might make us apply the first prediction step when is already too late.
- Letôs use an example: imagine the car is traveling a 100mph (160.934 m/s) if we have a latency of 100ms and, letôs say our time step dt is 80ms (meaning that we can guarantee that every 80 ms all our calculations will be done and ready to be sent to the actuators) then, by the time the command reaches the actuators, the car has travelled for 0.18secs at 100mph, that is 8.04 meters!! If our prediction horizon is large enough to see a curve letôs say 1sec in front of us, we can plan a head of time (fitting the curve and applying the model dynamics) to predict whatôs the most suitable command to minimize our cost function in this curve. ☺
- In truth, the Time Prediction Horizon is what it matters and therefore the combined selection of dt and N is what determines the successful output of our MPC. If our dt is fixed by our ROS or by how fast we can process all our code, then we are all set to just decide what value of N is the one that allows us to have a stable and successful result.
- In a formal development, I should have recorded and plotted all the relevant responses (cte, e-psi, and speed) with various N values and compare their different performances. In reality I did that but just used a visual inspection of the drive performance to choose my N.
- My final combination was dt = 0.08 s and N = 15. (1.2 sec Prediction Horizon)
- dt was chosen to be as realistic as possible in a complex and large piece of software. Usually 10-50ms cycle should be enough but I went for 80ms to be more conservative.
- With smaller N values, the MPC does not behave efficiently on the curves or when travelling at high speeds (>90mph)
- With more time, a formal approach capturing responses and plotting the results for different values of N would have been ideal.

Polynomial Fitting and MPC Preprocessing	A polynomial is fitted to waypoints. If the student preprocesses waypoints, the vehicle state, and/or actuators prior to the MPC procedure it is described.

- Since the telemetry coming from the simulator (specifically the waypoints ï ptsx, ptsy) are on MAP reference system but the rest of the visualization (mpc_x_, mpc_y_, next_x, next_y) are in Car Reference System, it seemed only reasonable to convert them all to the same system.
  The easiest way (only one conversion) is to put ptsx and ptsy on CAR Reference system. Also it turns out that this conversion is very convenient since the car position (x, y) and its heading (psi) in a CAR-Reference system are always 0. This fact simplifies some computation.
  This transformation can be found in line 143. Especial attention needs to be paid to the angle convention and the CAR-reference axis to make sure the transformation does not lead you to a problem that might be hard to detect.

- I also converted the velocity magnitude (speed) coming from the simulator from miles-per-hour to m/s. The reason why I did that is because it did NOT make sense to use a constant ñreference speedò and I used my fitted curve (fitted to the waypoints) to calculate a Radius Of Curvature (RoC) and finally to calculate the

Reference (target) speed based on some studies on max accelerations and comfortable driving. For more details, please read next section.

- I also played with the fitting polynomial ñdegreeò.
    - A one degree fitting (straight line) doesnôt look appropriate since the MPC will be missing a proper prediction while in a curve. I tried it just for the sake of a formal scientific method and, as expected, the results are not desirable.
    - A second order polynomial (parabola) is relatively good, but on the straight sections of the track, the fit is really poor and the car tends to follow it, generating a not desirable behavior.
    - A third degree polynomial fits relatively well on the straight sections of the track while doing a great job (if the prediction horizon is not too large) on the curved sections.
    - A higher degree polynomial would add more computational penalty while not improving the curve fitting within a reasonable perdition horizon and therefore the 3rd degree was the desired pick.

Model Predictive Control with Latency	The student implements Model Predictive Control that handles a 100 millisecond latency. Student provides details on how they deal with latency.

The highlights of my MPC implementation are:

- To investigate how to improve the driving while encouraging high speed, I decided to include different **COST elements** on the cost function that would capture what we wanted to accomplish. Here are all the cost elements. Some of them survived the test and others perish since they did not add any visible or relevant improvement on the car performance.
    - **CTE** : This one is obvious, but still worth some discussion. The Cross Track Error (CTE) might feel like the perfect Cost element in our Cost function. We would like to penalize any commands that provoke the CTE to increase. But letôs think about how to quantify this penalty. In a real world situation, where we are ñracingò on a track, how ñbadò it is to be far from the center of the lane? The answer is that it is important, since the CTE is the only reference we have in terms of the car position on the road, but at the same time, when we are racing we really donôt care about the CTE as far as we are not coming out of the track. Think about the F1 drivers trying to make a straight line while taking a curve instead of always trying to be on the center. Therefore, to capture this knowledge and facts, the weight associated to the penalty cost for the CTE will be there, but it will actually be small relative to other factors that we will discuss below.
    - **Psi error (yaw Error):** Donôt confuse this term with ñsteering errorò. Take a look above to the bicycle model and see the difference between the heading of the car and the steering angle. This term is, in a similar way as the CTE, a very important factor to determine if the CTE will increase or decrease in next time step. Itôs an addition to understand better the location and future state of the car. As we did before, letôs try to quantify how ñbadò it is to have a heading error. In a perfect ñLine followingò scenario, we would like the car to follow the curve smoothly but in a race scenario we really donôt care much about that. In fact a big penalty on this error will make the car swerve heavily and unsafely just to try to stay heading in the ñperfectò direction. We donôt want any swerving and we can predict that the weight associated with this cost element will be even smaller than the CTE.
    - **Velocity Error:** As we saw on the lectures, the MPC (as well as any other controller) will try to minimize the cost function and sometimes that could mean that the car could stop in a perfect position on center of the road. To avoid that, one of the approaches we saw on the lectures is to give the MPC a reference/target speed. In my case, and as I mentioned before, we, as humans, donôt have a ñconstantò target speed. We adjust our speed depending on the road profile (curves), condition of the road/tires, visibility, etc. In this case, we tried to calculate the value for a comfortable speed during a curve using a paper that identifies the threshold value of comfort for lateral accelerations on a vehicle as being 1.8 m/s2, with medium comfort and discomfort levels of 3.6 m/s2 and 5 m/s2, respectively "W. J. Cheng, Study on the evaluation method of highway alignment comfortableness [M.S. thesis], Hebei University of Technology, Tianjin, China, 2007." The process is very simple. The radial acceleration equation (also very easy to derive) is:

$$a = \frac{v^2}{R}$$

So, having defined a threshold for a comfortable radial acceleration and knowing the radius of curvature (RoC) of the curve, I can easily proceed to calculate the desirable speed of the vehicle while taking the turn..in real time and avoid a non-realistic constant speed reference.
In this case, I really wanted the car to try to maintain this max speed target so I can ñwinò the race and therefore my penalty weight for not achieving this speed is very high.

- **Steering use:** As it was wisely suggested on the lectures, in certain applications we might not want to abuse the actuators that might lead to fatigue, saturation or simple excessive wear and tear on them. In this case I tried first to get the highest performance and later tune this weight factor. I realized that if the MPC is finely tuned, you might actually be doing the proper steering and not over using them. We will see later that limiting and highly penalizing the differential steering (having big difference between 2 consecutive steering) will actually ñkill 2 birds with one shotò.
- **Accelerator/Brake use:** In a similar fashion as above, I tried first to achieve max performance and then later capture the Accel/brake output commands and avoid overuse.
- **CTE-SPEED:** This cost element was a heritage from the PID project. I wanted to summarize the minimization of the CTE while encouraging high speed. I implemented this element by taking the square of the CTE /Speed. Higher speeds at the same CTE would produce a better cost, encouraging in this way the faster driving. This elements proved to be not convenient for this development while on the PID was almost the Holy Grail. With MPC and its capacity of tackle each cost element separately and therefore avoiding coupling issues, this one did not produce an effective result.
- **Delta Steering:** This cost element represents the penalty associated with ñbigò changes in steering angles between two consecutive time steps. In some sense this is, conceptually, a low pass filter. On the PID that was represented by our Derivative term. We do not want to swerve heavily and constantly, especially driving at high speeds, and therefore we want to hardly penalize this behavior.
- **Delta Accel/Brake:** This cost element represents the penalty associated with ñbigò changes in acceleration and braking between two consecutive time steps. In a real world situation, the car would not be able to instantly go from max acceleration to No acceleration at all. The dynamics of the motor would not allow this behavior. But in a real world situation, a race driver, in fact would actually accelerate and step on the brake if needed to create the force to help him/her maintain a centripetal acceleration while attempting to slow down. In my case and since I really was focusing on the performance, I left this cost element with a ñ0ò weight.
- **Acceleration Error:** This last element was implemented trying to investigate higher order derivatives of the position. Since I didnôt want to have a constant speed reference, I wanted to investigate the 2nd derivative (acceleration) and also a 3rd and 4th derivatives (jerk and snap). Iôve done previous work on quad-copters and planning minimum jerk ad snap trajectories. In this case and after tuning the main parameters I realized that this cost function was not necessary at this point.

The final weights are (lines 40 ï 48 on MPC.cpp):

```
const double W_cte       = 0.75;
const double W_yaw_err   = 0.03;
const double W_vel_err   = 999999999;
const double W_steer_use = 0;
const double W_accel_use = 0;
const double W_cteSpeed  = 0;
const double W_dSteer    = 9999;
const double W_dAccel    = 0;
const double W_acc_err   = 0;
```

As you can see, only 4 cost elements are necessary for an impressive fast drive. (Video Attached)

- **Latency:** The approach I used to deal with a ñknownò (100 ms) latency was very simple. Knowing that my MPC commands would get to the actuators with a 100ms delay I just decided to use the known state of the car at time t, predict its new state (assuming no disturbances and noise) at t+100ms and send that new (predicted) state to the MPC. Basically, what Iôm doing is shifting all my predictions 100ms. Please take a look to the graph below for reference. This was implemented in lines 168 ï 183 on main.cpp

```
const double latency = 0.1; // 100 ms

// Let's predict the state. Remember that px, py and psi wrt car are all 0.
const double pred_px = v * latency;
const double pred_py = 0;
const double pred_psi = -v * steering * latency / Lf;
const double pred_v = v + throttle * latency;
const double pred_cte = cte + v * sin(epsi) * latency;
const double pred_epsi = epsi + pred_psi;

//state << CarRef_px, CarRef_py, CarRef_psi, v, cte, epsi;
state << pred_px, pred_py, pred_psi, pred_v, pred_cte, pred_epsi;
```