24677 — P1

Connor W. Colombo (cwcolomb)

November 3rd 2020

1 Project 1

1.1 Model Linearization

1.1.1 Nonlinear Vehicle Dynamics

The system dynamics are given by the following system of equations:

$$\begin{cases} \ddot{x} = \dot{\psi}\dot{y} + \frac{F}{m} - fg \\ \ddot{y} = \begin{cases} -\dot{\psi}\dot{x} & \dot{x} < 0.5 \mathrm{m}/\mathrm{s} \\ -\dot{\psi}\dot{x} + \frac{2C_\alpha}{m}\left(\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right)\cos(\delta) - \frac{\dot{y}-l_r\dot{\psi}}{\dot{x}}\right) & \dot{x} \geq 0.5 \mathrm{m}/\mathrm{s} \end{cases} \\ \ddot{\psi} = \begin{cases} 0 & \dot{x} < 0.5 \mathrm{m}/\mathrm{s} \\ \frac{2C_\alpha l_f}{I_z}\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right) + \frac{2C_\alpha l_r}{I_z}\left(\frac{\dot{y}-l_r\dot{\psi}}{\dot{x}}\right) & \dot{x} \geq 0.5 \mathrm{m}/\mathrm{s} \end{cases} \\ \dot{X} = \dot{x}\cos(\psi) - \dot{y}\sin(\psi) \\ \dot{Y} = \dot{x}\sin(\psi) + \dot{y}\cos(\psi) \end{cases} \tag{1}$$

subject to the following constraints:

$$\begin{cases} |\delta| \leq \frac{\pi}{6}\mathrm{rad} \\ 0\mathrm{N} \leq F \leq 15,736\mathrm{N} \\ \dot{x} \geq 10^{-5} \mathrm{m}/\mathrm{s} \end{cases} \tag{2}$$

and the following system parameters:

$$\begin{cases} m = & 1,888.6\mathrm{kg} \\ l_r = & 1.39\mathrm{m} \\ l_f = & 1.55\mathrm{m} \\ C_\alpha = & 20,000\mathrm{N} \\ I_z = & 25,854\mathrm{kgm}^2 \\ f = & 0.019 \\ \Delta t = & 0.032\mathrm{s} \end{cases} \tag{3}$$

1.1.2 Linearization of Vehicle Dynamics

Let the linearized system be separated into two LTI state spaces for the lateral states, $\mathbf{s}_1$, and the longitudinal states, $\mathbf{s}_2$, both subject to the control inputs $\mathbf{u}$ as defined below.

$$\dot{\mathbf{s}}_1 = \mathbf{A}_1\mathbf{s}_1 + \mathbf{B}_1\mathbf{u} \quad | \quad \mathbf{s}_1 = \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} \tag{4}$$

$$\dot{\mathbf{s}}_2 = \mathbf{A}_2\mathbf{s}_2 + \mathbf{B}_2\mathbf{u} \quad | \quad \mathbf{s}_2 = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \ \mathbf{u} = \begin{bmatrix} \delta \\ F \end{bmatrix} \tag{5}$$

Accordingly, based on the vehicle dynamics defined above in Equation (1), the nonlinear dynamics for each of the lateral and longitudinal subspaces can be defined as shown below:

$$\dot{\mathbf{s}}_1 = \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{cases} \begin{bmatrix} \dot{y} \\ -\dot{\psi}\dot{x} \\ \dot{\psi} \\ 0 \end{bmatrix}, & \dot{x} < 0.5^{\mathrm{m}}/_{\mathrm{s}} \\ \begin{bmatrix} \dot{y} \\ -\dot{\psi}\dot{x} + \frac{2C_\alpha}{m}\left(\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right)\cos(\delta) - \frac{\dot{y}-l_r\dot{\psi}}{\dot{x}}\right) \\ \dot{\psi} \\ \frac{2C_\alpha l_f}{I_z}\left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right) + \frac{2C_\alpha l_r}{I_z}\left(\frac{\dot{y}-l_r\dot{\psi}}{\dot{x}}\right) \end{bmatrix} & \dot{x} \geq 0.5^{\mathrm{m}}/_{\mathrm{s}} \end{cases} \tag{6}$$

$$\dot{\mathbf{s}}_2 = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{\psi}\dot{y} + \frac{F}{m} - fg \end{bmatrix} \tag{7}$$

Lateral System

From the form of the complete nonlinear lateral dynamics presented in Equation (4), it's clear that the low-speed ($\dot{x} < 0.5^{\mathrm{m}}/_{\mathrm{s}}$) lateral dynamics are already essentially linear and, thus, the LTI state space form would be expressed as:

$$\dot{\mathbf{s}}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \mathbf{0u} \quad | \quad \dot{x} < 0.5^{\mathrm{m}}/_{\mathrm{s}} \tag{8}$$

Since the low-speed lateral dynamics leave all states with no connection to the inputs and, thus, are inherently uncontrollable, they won't be considered any further. Rather, only the high-speed ($\dot{x} \geq 0.5^{\mathrm{m}}/_{\mathrm{s}}$) condition will be linearized to determine the LTI state space for the lateral dynamics.

$$\text{let: } \dot{\mathbf{s}}_1 = f_1(\mathbf{s}_1, \mathbf{u})$$

To linearize this system, the equilibrium condition must first be found as follows:

$$\text{let: } f_1(\bar{\mathbf{s}}_1, \bar{\mathbf{u}}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{\bar{y}} \\ -\dot{\bar{\psi}}\dot{\bar{x}} + \frac{2C_\alpha}{m}\left(\left(\bar{\delta} - \frac{\dot{\bar{y}}+l_f\dot{\bar{\psi}}}{\dot{\bar{x}}}\right)\cos(\bar{\delta}) - \frac{\dot{\bar{y}}-l_r\dot{\bar{\psi}}}{\dot{\bar{x}}}\right) \\ \dot{\bar{\psi}} \\ \frac{2C_\alpha l_f}{I_z}\left(\bar{\delta} - \frac{\dot{\bar{y}}+l_f\dot{\bar{\psi}}}{\dot{\bar{x}}}\right) + \frac{2C_\alpha l_r}{I_z}\left(\frac{\dot{\bar{y}}-l_r\dot{\bar{\psi}}}{\dot{\bar{x}}}\right) \end{bmatrix}$$

$$\rightarrow \bar{\mathbf{s}}_1 = \begin{bmatrix} \bar{y} \\ \dot{\bar{y}} \\ \bar{\psi} \\ \dot{\bar{\psi}} \end{bmatrix} = \begin{bmatrix} \bar{y} \\ 0 \\ \bar{\psi} \\ 0 \end{bmatrix}, \quad \bar{\mathbf{u}} = \begin{bmatrix} \bar{\delta} \\ \bar{F} \end{bmatrix} = \begin{bmatrix} 0 \\ \bar{F} \end{bmatrix}$$

The system can then be linearized using the Jacobians of the nonlinear system evaluated at the equilibrium:

$$\dot{\delta}_{\mathbf{s}_1} = \left[\frac{\partial f_1}{\partial \mathbf{s}_1}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} \delta_{\mathbf{s}_1} + \left[\frac{\partial f_1}{\partial \mathbf{u}}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} \delta_{\mathbf{u}} \quad | \quad \delta_{\mathbf{s}_1} = \mathbf{s}_1 - \bar{\mathbf{s}}_1 = \begin{bmatrix} y - \bar{y} \\ \dot{y} \\ \psi - \bar{\psi} \\ \dot{\psi} \end{bmatrix}, \quad \delta_{\mathbf{u}} = \mathbf{u} - \bar{\mathbf{u}} = \begin{bmatrix} \delta \\ F - \bar{F} \end{bmatrix}$$

$$\because \left[\frac{\partial f_1}{\partial \mathbf{s}_1}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} = \begin{bmatrix} \frac{\partial f_{1_1}}{\partial y} & \frac{\partial f_{1_1}}{\partial \dot{y}} & \frac{\partial f_{1_1}}{\partial \psi} & \frac{\partial f_{1_1}}{\partial \dot{\psi}} \\ \frac{\partial f_{1_2}}{\partial y} & \frac{\partial f_{1_2}}{\partial \dot{y}} & \frac{\partial f_{1_2}}{\partial \psi} & \frac{\partial f_{1_2}}{\partial \dot{\psi}} \\ \frac{\partial f_{1_3}}{\partial y} & \frac{\partial f_{1_3}}{\partial \dot{y}} & \frac{\partial f_{1_3}}{\partial \psi} & \frac{\partial f_{1_3}}{\partial \dot{\psi}} \\ \frac{\partial f_{1_4}}{\partial y} & \frac{\partial f_{1_4}}{\partial \dot{y}} & \frac{\partial f_{1_4}}{\partial \psi} & \frac{\partial f_{1_4}}{\partial \dot{\psi}} \end{bmatrix}_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{2C_\alpha}{m\dot{x}}(1+\cos(\delta)) & 0 & \frac{2C_\alpha}{m\dot{x}}(l_r - l_f\cos(\delta)) - \dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2C_\alpha}{I_z\dot{x}}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z\dot{x}}(l_r^2 + l_f^2) \end{bmatrix}_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})}$$

$$\left[\frac{\partial f_1}{\partial \mathbf{s}_1}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & 0 & \frac{2C_\alpha}{m\dot{x}}(l_r - l_f) - \dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2C_\alpha}{I_z\dot{x}}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z\dot{x}}(l_r^2 + l_f^2) \end{bmatrix},$$

$$\because \left[\frac{\partial f_1}{\partial \mathbf{u}}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} = \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m}\left(\cos(\delta) - \left(\delta - \frac{\dot{y}+l_f\dot{\psi}}{\dot{x}}\right)\sin(\delta)\right) & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix}_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})}$$

$$\left[\frac{\partial f_1}{\partial \mathbf{u}}\right]_{(\bar{\mathbf{s}}_1,\,\bar{\mathbf{u}})} = \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix}$$

$$\therefore \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & 0 & \frac{2C_\alpha}{m\dot{x}}(l_r - l_f) - \dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2C_\alpha}{I_z\dot{x}}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z\dot{x}}(l_r^2 + l_f^2) \end{bmatrix} \begin{bmatrix} y - \bar{y} \\ \dot{y} \\ \psi - \bar{\psi} \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix} \begin{bmatrix} \delta \\ F - \bar{F} \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & 0 & \frac{2C_\alpha}{m\dot{x}}(l_r - l_f) - \dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2C_\alpha}{I_z\dot{x}}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z\dot{x}}(l_r^2 + l_f^2) \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix} \begin{bmatrix} \delta \\ F \end{bmatrix} \tag{9}$$

Based on the simplified linearized system presented above in Equation (9), the LTI state space form of the linearized lateral dynamics can be determined to take the form shown below.

$$\dot{\mathbf{s}}_1 = \mathbf{A}_1\mathbf{s}_1 + \mathbf{B}_1\mathbf{u} \quad | \quad \mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & 0 & \frac{2C_\alpha}{m\dot{x}}(l_r - l_f) - \dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2C_\alpha}{I_z\dot{x}}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z\dot{x}}(l_r^2 + l_f^2) \end{bmatrix}, \quad \mathbf{B}_1 = \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix} \tag{10}$$

Longitudinal System

The nonlinear longitudinal dynamics presented in Equation (7) can decomposed into a primary system $f_2(\mathbf{s}_2, \mathbf{u})$ and a collection of disturbance terms D so that the dynamics take the form of $\dot{\mathbf{s}}_2 = f_2(\mathbf{s}_2, \mathbf{u}) + D$ as shown below.

$$\dot{\mathbf{s}}_2 = f_2(\mathbf{s}_2, \mathbf{u}) + D \quad | \quad f_2(\mathbf{s}_2, \mathbf{u}) = \begin{bmatrix} \dot{x} \\ \frac{F}{m} \end{bmatrix}, \quad D = \begin{bmatrix} 0 \\ \dot{\psi}\dot{y} - fg \end{bmatrix} \tag{11}$$

Since the primary undisturbed system $f_2(\mathbf{s}_2, \mathbf{u})$ is itself LTI, the linearized longitudinal system, in the form of Equation (5), can be found to be simply be:

$$\dot{\mathbf{s}}_2 = \mathbf{A}_2\mathbf{s}_2 + \mathbf{B}_2\mathbf{u} \quad | \quad \mathbf{A}_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad \mathbf{B}_2 = \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \tag{12}$$

1.2 Controller Synthesis in Simulation

The following figures show the controller performance subject to the following parameters.

$$K_{u\text{long}} = 163500,\ T_{u\text{long}} = 0.064s \tag{13}$$

$$K_{u\text{lat}} = 0.366,\ T_{u\text{lat}} = 6s \tag{14}$$

$$K_p = 0.6K_u,\ K_i = 1.2\frac{K_u}{T_u},\ K_d = \frac{3}{40}K_uT_u \tag{15}$$

$$T_{\text{target}} = 300s \tag{16}$$

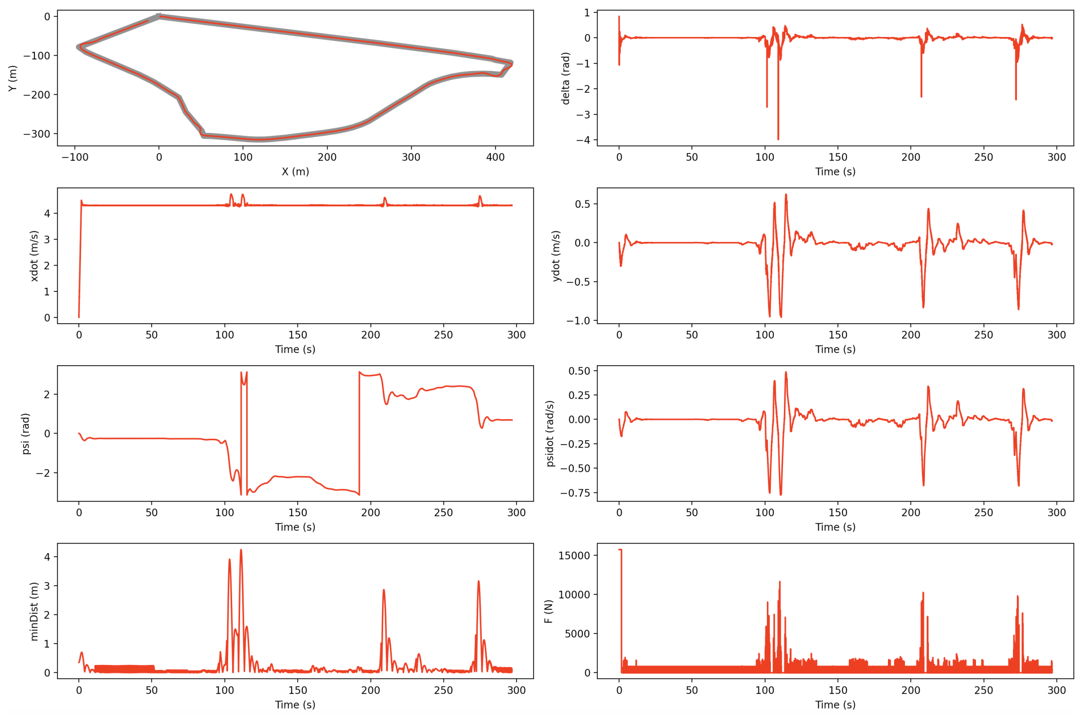

Figure 1: Performance Plot

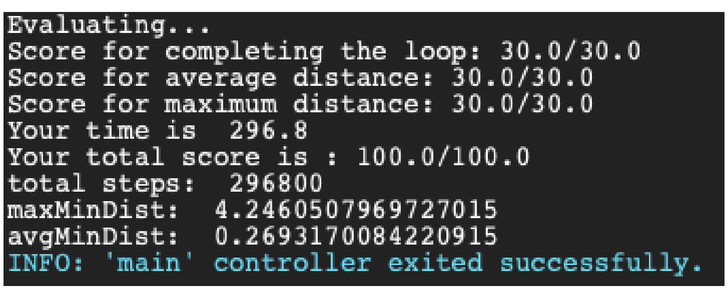
```
Evaluating...
Score for completing the loop: 30.0/30.0
Score for average distance: 30.0/30.0
Score for maximum distance: 30.0/30.0
Your time is  296.8
Your total score is : 100.0/100.0
total steps:  296800
maxMinDist:  4.2460507969727015
avgMinDist:  0.2693170084220915
INFO: 'main' controller exited successfully.
```

Figure 2: Performance Data

Code

```python
# Fill in the respective functions to implement the controller

# Import libraries
import numpy as np
from base_controller import BaseController
from scipy import signal, linalg
from util import *

```

```python
# CustomController class (inherits from BaseController)
class CustomController(BaseController):

    def __init__(self, trajectory):

        super().__init__(trajectory)

        # Define constants
        # These can be ignored in P1
        self.lr = 1.39
        self.lf = 1.55
        self.Ca = 20000
        self.Iz = 25854
        self.m = 1888.6
        self.g = 9.81

        self.time = 0 # [s] Current time into trajectory execution
        self.target_time = 300 # [s] Target time to complete loop.
        self.track_length = 1290.385282704354; # [m] Total path length of the track

        ####
        # Control Parameters:
        ####
        # Target Correction Time [s] (used to calibrate PID constants for each control loop)
        self.kt = 1 # note: this is a qualitative trimming factor
        # Core PID Constants (kp, ki, kd), adjusted (trimmed) for each controller via kt.
        #L_long = (12768 + 0.6*(13088-12768))/1000 # time for long controller to reach 10% SS with step input
        #T_long = (85824 + 0.4*(86720-85824))/1000 # time for long controller to reach 90% SS with step input
        Ku_long = 163500 # Lowest long gain causing rapid oscillation in speed under step input
        Tu_long = (0.096/2 + 0.032/2) # Period of that oscillation
        Ku_lat = 0.60*0.61
        Tu_lat = 6
        self.k_pid_longitudinal = np.asarray([0.60*Ku_long, 1.2*Ku_long/Tu_long, 3*Ku_long*Tu_long/40]).reshape((1,3)) #np.asarray([163500,0,0]).reshape((1,3))#np.asarray([1.2*T_long/L_long, 1.2*T_long/L_long/(2.0*L_long), 1.2*T_long/L_long*0.5*L_long]).reshape((1,3))#np.asarray([0.60*Ku, 1.2*Ku/Tu, 3*Ku*Tu/40]).reshape((1,3))
        self.k_pid_lateral = np.asarray([0.60*Ku_lat, 1.2*Ku_lat/Tu_lat, 3*Ku_lat*Tu_lat/40]).reshape((1,3))#np.asarray([500,0,0]).reshape((1,3))#np.asarray([1,0,0]).reshape((1,3))#

        self.look_ahead_multiple = 2.5 # How many car lengths to look ahead and average over when determining desired heading angle
        self.look_ahead_indices = int(self.look_ahead_multiple * 24.0) # Corresponding number of indices (note: car length is approx. equiv. to path length over 24 points)

        ### Time of Last Zero Crossing for Each Signal (for determining Ziegler-Nichols Tu):
        self.last_zero_crossing = np.zeros((3,1))
        self.oscillation_period = np.zeros((3,1))

        ####
        # Initialize Signals:
        ####

        # Initialize Error Signals as an Errors Matrix, E:
        # Rows are: (x,y,psi) = (alongtrack, crosstrack, heading),
        # Cols are: Position, Integral, Derivative
        self.E = np.zeros((3,3))

    def update(self, timestep):

        trajectory = self.trajectory

        lr = self.lr
        lf = self.lf
```

```python
        Ca = self.Ca
        Iz = self.Iz
        m = self.m
        g = self.g

        # Fetch the states from the BaseController method
        delT, X, Y, xdot, ydot, psi, psidot = super().getStates(timestep)

        self.time += delT # update total time into trajectory execution

        # Design your controllers in the spaces below.
        # Remember, your controllers will need to use the states
        # to calculate control inputs (F, delta).

        # Compute rotation matrix from world frame to local (vehicle) frame:
        rotation_mat = np.array([[np.cos(psi), np.sin(psi)], [-np.sin(psi), np.cos(psi)]])

        ####
        # Compute Pose to Path Error:
        # This is what keeps the vehicle on track.
        ####
        ### Compute Vector from Vehicle COM to Nearest Waypoint in Local (Vehicle) Frame:
        # Get nearest waypoint to vehicle COM:
        _, nearest_waypoint_idx = closestNode(X, Y, trajectory)
        nearest_waypoint = trajectory[nearest_waypoint_idx]
        # Compute pose to path (waypoint) vector in world frame:
        p2p_world = nearest_waypoint - np.asarray([X,Y])
        # Compute pose to path (waypoint) vector in local (vehicle) frame:
        p2p_local = rotation_mat @ p2p_world

        ### Compute Pose to Path Heading Angle Error:
        # Points that comes before and 2.5 car lengths later waypoint:
        nearest_waypoint_prev = trajectory[(nearest_waypoint_idx-1) % trajectory.shape[0]]
        nearest_waypoint_fwd = trajectory[(nearest_waypoint_idx+self.look_ahead_indices) % trajectory.shape[0]] # look roughly 2.5 car lengths ahead
        # Desired Trajectory Position Delta around Current Waypoint:
        nearest_waypoint_delta = nearest_waypoint_fwd - nearest_waypoint_prev
        # Compute Desired Heading:
        desired_heading = np.arctan2(nearest_waypoint_delta[1], nearest_waypoint_delta[0])
        # Compute current heading of the vehicle (direction it's currently driving, not necessarily pointing direction if there's sideslip):
        Xdot, Ydot = np.linalg.inv(rotation_mat) @ np.asarray([xdot,ydot])
        current_heading = np.arctan2(Ydot,Xdot)
        # Compute Heading Error:
        p2p_heading_error = desired_heading - current_heading
        p2p_heading_error = np.arctan2(np.sin(p2p_heading_error), np.cos(p2p_heading_error)) # Remap to atan2 space

        ####
        # Compute Alongtrack Error as longitudinal speed difference:
        ####
        speed_diff = self.track_length / self.target_time - xdot

        """
        # Second oldest alongtrack error:
        ###
        # Compute path length from the nearest waypoint to the desired waypoint based on desired completion time:
        # This is what propels the vehicle forward.
        ###
        ### Get The Waypoint Vehicle Should be at given the time into execution:
        target_waypoint_idx = clamp(np.round(self.time / self.target_time * trajectory.shape[0]), 0,trajectory.shape[0]-1)
        ### Compute the path distance between the waypoints:
        idx_1, idx_2 = tuple(map(int, (nearest_waypoint_idx, target_waypoint_idx)))
```

```
        path_sign = 1.0
        if idx_1 > idx_2:
            # Ensure proper ordering. Swap order and sign if necessary.
            path_sign = -1.0
            idx_1, idx_2 = idx_2, idx_1
        section = trajectory[idx_1:(idx_2+1)] # Section of trajectory between idx_1 and idx_2, inclusive
        point_distances = np.sqrt(np.sum(np.diff(section, axis=0)**2, axis=1)) # Distances between each point
        path_distance = path_sign * np.sum(point_distances)
        """
        """
        ### Old alongtrack error (alongtrack distance from current position to target position):
        target_waypoint = trajectory[int(target_waypoint_idx)]
        # Compute pose to target waypoint vector in world frame:
        p2target_world = target_waypoint - np.asarray([X,Y])
        # Compute pose to target waypoint vector in local (vehicle) frame:
        p2target_local = rotation_mat @ p2target_world
        path_distance = p2target_local[0]
        """

        ####
        # Update Error Signals in an Errors Matrix, E:
        ####
        # Rows are: (x,y,psi) = (alongtrack, crosstrack, heading),
        # Cols are: Position, Integral, Derivative
        E = self.E
        E_prev = E.copy()

        ### Update Proportional Signals:
        # Crosstrack error (speed difference from required speed)
        E[0,0] = speed_diff
        # Alongtrack error (from current position to nearest waypoint in trajectory):
        E[1,0] = p2p_local[1]
        # Heading error (from current heading to heading implictly specified by trajectory at nearest waypoint):
        E[2,0] = p2p_heading_error

        ### Update Integral Signals:
        E[:,1] = E[:,1] + E[:,0] * delT # mult by timestep in case not uniform (not if Webots Fast changes dt)

        ### Update Derivative Signals:
        E[:,2] = (E[:,0] - E_prev[:,0]) / delT # divide by timestep in case not uniform (not if Webots Fast changes dt)

        ####
        # Condition Error Signals:
        ####
        ### Prevent Integral Windup:
        # If error has crossed zero, zero it:
        zero_crossing = (np.sign(E[:,0]) * np.sign(E_prev[:,0])) <= 0.0
        E[zero_crossing,1] = 0.0
        # If derivative is high (still driving towards equilibrium), reduce integral contribution:
        high_deriv_error= E[:,2]*delT > np.asarray([
            0.5,
            (lf+lr)/4, # High Y err deriv means > 1/4 car length
            2*np.pi/6/4 # High TH err deriv means > 1/4 steering range
        ])
        E[high_deriv_error,1] -= E[high_deriv_error,0] * delT * 9.0/10.0 # only contribute 1/10th of what you would have to
        ↪  the integral

        ### Find and Print Oscillation Period (for determining Ziegler-Nichols Tu):
        if np.count_nonzero(zero_crossing):
            self.oscillation_period[zero_crossing] = self.time - self.last_zero_crossing[zero_crossing]
            self.last_zero_crossing[zero_crossing] = self.time
            #print(self.oscillation_period.T)
```

```python

        ####
        # Compute PID Correction Factors (allow the same tuned constants to
        # drive each controller = fewer constants to tune overall).
        ####
        kt = self.kt
        kx = 1/kt;
        kth = 1/kt;
        V = np.abs(np.linalg.norm([xdot, ydot]));
        if V < 0.05: # Prevent from getting too large
            ky = 2 / 0.05 / (kt**2);
        else:
            ky = 2 / V / (kt**2); # Lessen impact of crosstrack error as velocity increases (to minimize wild swings)

        ####
        # Create a PID Constants Matrix, K:
        # Each row contains the (kp, ki, and kd) for the error terms
        # corresponding to that row (ex, ey, or eth).
        ####
        K = np.asarray([self.k_pid_longitudinal*kx, self.k_pid_lateral*ky, self.k_pid_lateral*kth]).squeeze()

        ####
        # Compute Control Signals to Minimize Each Type of Error
        # (rows: x=alongtrack, y=crosstrack, th=heading)
        ####
        C = np.sum(E*K, axis=1)
        # Extract Independent Control Signals for Minimizing Alongtrack, Crosstrack, and Heading Errors:
        (Cx, Cy, Cth) = C;

        #print((np.round(E[0,0]), self.track_length / self.target_time, xdot, np.round(Cx), np.round(Cy), np.round(Cth)))
        #print((np.round(1000*self.time), np.round(speed_diff)))

        # ---------------|Lateral Controller|-------------------------
        # Note: Lateral controller consists of two PID subcontrollers working togther to minimize heading error and crosstrack error.
        # Using a heading control component allows for improved handling of turns (since its trying to point the car in the right direction)
        delta = Cy + Cth; # Allow crosstrack and heading errors to drive the steering angle
        # Note: This approach is effectively equivalent to having one lateral pid controller which controls (ey/kt + eth/V/kt**2) but is just a cleaner representation

        # ---------------|Longitudinal Controller|-------------------------
        F = np.linalg.norm([clamp(Cx,0,15737), Cy]); # Allow crosstrack and alongtrack errors to drive the throttle
        # Note: This approach is effectively equivalent to having one longitudinal pid controller which controls norm[ey/kt, eth/V/kt**2] but is just a cleaner representation
        # Todo: Slow down based on Cth? (slower when steering high to increase control authority)

        # Return all states and calculated control inputs (F, delta)
        return X, Y, xdot, ydot, psi, psidot, F, delta
```