State Equations:

$$\ddot{y} = -\dot{\psi}\dot{x} + \frac{2C_\alpha}{m}\left(\cos\delta\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) - \frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right)$$

$$\ddot{x} = \dot{\psi}\dot{y} + \frac{1}{m}(F - fg)$$

$$\ddot{\psi} = \frac{2l_fC_\alpha}{I_z}\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) - \frac{2l_rC_\alpha}{I_z}\left(-\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right)$$

$$\dot{X} = \dot{x}\cos\psi - \dot{y}\sin\psi$$

$$\dot{Y} = \dot{x}\sin\psi + \dot{y}\cos\psi$$

Measurements:

$$y = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ X \\ Y \\ \psi \end{bmatrix}$$

Model Parameters:

$$\delta \leq \frac{\pi}{6}\ rad$$

$$F \geq 0\ and\ F \leq 15367\ N$$

$$x \geq \dot{10^{-5}}\ m/s$$

Name	Description	Unit	Value
$(\dot{x}, \dot{y})$	Vehicle's velocity along the direction of vehicle frame	m/s	State
(X, Y)	Vehicle's coordinates in the world frame	m	State
$\psi, \dot{\psi}$	Body yaw angle, angular speed	$rad, rad/s$	State
δ, δ_f	Front wheel angle	rad	Input
F	Total input force	N	Input
m	Vehicle mass	kg	1888.6
l_r	Length from rear tire to the center of mass	m	1.39
l_f	Length from front tire to the center of mass	m	1.55
C_α	Cornering stiffness of each tire	N	20000
I_z	Yaw inertia	$kg\ m^2$	25854

F_{pq}	Tire Force, $p \in \{x, y\}$, $q \in \{f, r\}$	N	Depends on input force
f	Rolling resistance coefficient	N/A	0.019
$delT$	Simulation timestep	sec	0.032

I. Linearization

Lateral Linearization

$$u = \begin{bmatrix} \delta \\ F \end{bmatrix}, s_1 = \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix}$$

Assume $x, \dot{x}$ are constant in the lateral system.

$$Let\ \dot{x} = v_x$$

Non-linearized System:

$$\dot{s_1} = \begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = \begin{bmatrix} \dot{y} \\ -\dot{\psi} v_x + \frac{2C_\alpha}{m}\left(\cos\delta\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{v_x}\right) - \frac{\dot{y} - l_r\dot{\psi}}{v_x}\right) \\ \dot{\psi} \\ \frac{2l_fC_\alpha}{I_z}\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{v_x}\right) - \frac{2l_rC_\alpha}{I_z}\left(-\frac{\dot{y} - l_r\dot{\psi}}{v_x}\right) \end{bmatrix} = f(s_1, u)$$

$A_1 = \nabla f_{s_1}, B_1 = \nabla f_u$

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{2C_\alpha}{mv_x}(\cos\delta + 1) & 0 & -v_x + \frac{2C_\alpha}{mv_x}(-l_f\cos\delta + l_r) \\ 0 & 0 & 1 & 0 \\ 0 & \frac{2C_\alpha}{I_zv_x}(l_r - l_f) & 0 & \frac{2C_\alpha}{I_zv_x}(-l_f^2 - l_r^2) \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m}\left(-\sin\delta\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{v_x}\right) + \cos\delta\right) & 0 \\ 0 & 0 \\ \frac{2l_fC_\alpha}{I_z} & 0 \end{bmatrix}$$

Apply small angle Approximations → $\cos(\delta) \approx 1, \sin(\delta) \approx 0$

$$A_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{mv_x} & 0 & -v_x - \frac{2C_\alpha}{mv_x}(l_f - l_r) \\ 0 & 0 & 1 & 0 \\ 0 & \frac{2C_\alpha}{I_z v_x}(l_r - l_f) & 0 & -\frac{2C_\alpha}{I_z v_x}(l_f^2 + l_r^2) \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2l_f C_\alpha}{I_z} & 0 \end{bmatrix}$$

Longitudinal Linearization

$$u = \begin{bmatrix} \delta \\ F \end{bmatrix}, s_2 = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Assume $y, \dot{y}, \psi, \dot{\psi}$, are constant in the longitudinal linearization

Non-linearized System:

$$\dot{s_2} = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{\psi}\dot{y} + \frac{1}{m}(F - fg) \end{bmatrix}$$

$A_2 = \nabla f_{s_2}, B_1 = \nabla f_2$

$$A_2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$B_2 = \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix}$$

Dynamics of rolling accommodate for using a disturbance term

$$d(t) = \begin{bmatrix} 0 \\ \dot{\psi}\dot{y} + \frac{fg}{m} \end{bmatrix}$$

PID Results:

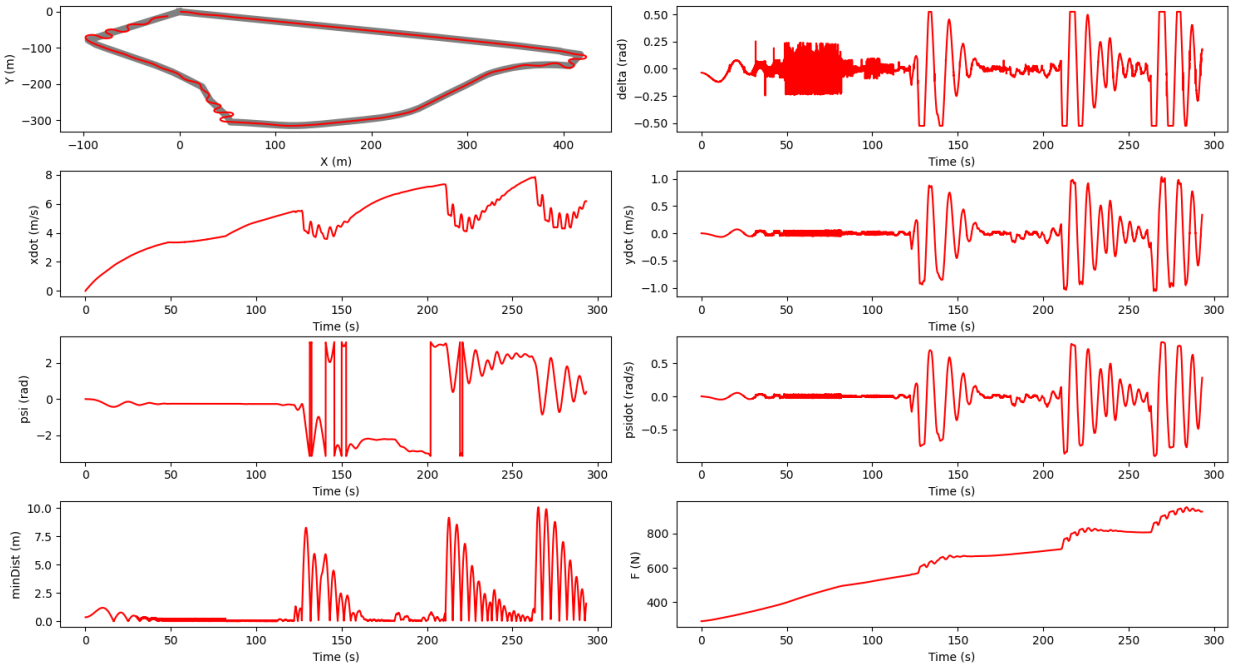

Fig.1 Simulation Plot

Fig.2 Simulation Result

Twiddle Algorithm for PID tuning

```
import numpy as np
from util import getTrajectory, closestNode, clamp

default_dt = 0.032

class PID:
    def __init__(self, Kp, Ki, Kd, dt=default_dt):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.dt = dt
        self.integral = 0
        self.prev_error = 0

    def reset(self):
        self.integral = 0
        self.prev_error = 0

    def update(self, setpoint, measurement):
        error = setpoint - measurement
        self.integral += error * self.dt
        derivative = (error - self.prev_error) / self.dt
        output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        self.prev_error = error
        return output

# Load trajectory using util.py
trajectory = getTrajectory('buggyTrace.csv')

# Simulate lateral controller (returns total squared cross-track error)
def lateral_error(pid_gains, trajectory, dt=default_dt):
    Kp, Ki, Kd = pid_gains
    pid = PID(Kp, Ki, Kd, dt)
    X, Y = trajectory[0, 0], trajectory[0, 1]  # start at first waypoint
    total_error = 0
    for i in range(1, len(trajectory)):
        cross_track_error, idx = closestNode(X, Y, trajectory)
        control = pid.update(0, cross_track_error)  # setpoint is 0
        # Simulate: move towards next waypoint, apply steering
        direction = trajectory[i] - np.array([X, Y])
        speed = 11 #
        max_steer = np.radians(30)
        control = clamp(control, -max_steer, max_steer)
```

```
        X += direction[0] * dt + control * dt
        Y += direction[1] * dt + control * dt
        total_error += cross_track_error ** 2
    return total_error

# Simulate longitudinal controller (returns total squared speed error)
def longitudinal_error(pid_gains, trajectory, desired_speed=11.0, dt=default_dt):
    Kp, Ki, Kd = pid_gains
    pid = PID(Kp, Ki, Kd, dt)
    speed = 0.0
    total_error = 0
    for i in range(1, len(trajectory)):
        speed_error = desired_speed - speed
        control = pid.update(desired_speed, speed)
        # Simulate: update speed
        speed += control * dt
        speed = clamp(speed, 0, 50)  # clamp to reasonable speed range
        total_error += speed_error ** 2
    return total_error

# Twiddle algorithm for PID tuning
def twiddle(error_func, trajectory, tol=0.01, initial_p=[0.00, 0.0, 0.0],
initial_dp=[0.01, 0.01, 0.01]):
    p = initial_p.copy()
    dp = initial_dp.copy()
    best_err = error_func(p, trajectory)
    it = 0
    prev_err = None
    while sum(dp) > tol:
        for i in range(len(p)):
            p[i] += dp[i]
            err = error_func(p, trajectory)
            if err < best_err:
                best_err = err
                dp[i] *= 1.1
            else:
                p[i] -= 2 * dp[i]
                err = error_func(p, trajectory)
                if err < best_err:
                    best_err = err
                    dp[i] *= 1.05
                else:
                    p[i] += dp[i]
                    dp[i] *= 0.95
        it += 1
```

```
        print(f"Iteration {it}, PID: {p}, Error: {best_err}")
        # Stop if error does not change
        if prev_err is not None and abs(best_err - prev_err) < 1e-8:
            print("Error did not change, stopping Twiddle.")
            break
        prev_err = best_err
    return p

if __name__ == "__main__":
    print("Tuning lateral PID...")
    best_pid_lateral = twiddle(lateral_error, trajectory)
    print("Best lateral PID gains:", best_pid_lateral)
    # dp's set to 0.1 - Best lateral PID gains: [0.1, 0.0, 0.0]
    # dp s set to 0.01 - Best lateral PID gains: [0.11435888100000004, 0.0,
0.0178966525]
    print("Tuning longitudinal PID...")
    best_pid_longitudinal = twiddle(longitudinal_error, trajectory)
    print("Best longitudinal PID gains:", best_pid_longitudinal)
    # dp's set to 0.1- Best longitudinal PID gains: [29.91268053287073,
1.3640345468017792, 0.02870992997411423]
    # dp's set to 0.01 -Best longitudinal PID gains: [27.580149049219923,
0.4822354299071474, 0.0734521331616762]
```

Tuned PID Controller:

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

class PID:
    def __init__(self, Kp, Ki, Kd, dt):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.dt = dt
        self.integral = 0
        self.prev_error = 0

    def reset(self):
        self.integral = 0
        self.prev_error = 0

    def update(self, setpoint, measurement):
        error = setpoint - measurement
        self.integral += error * self.dt
        derivative = (error - self.prev_error) / self.dt
        output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        self.prev_error = error
        return output

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

    def __init__(self, trajectory):

        super().__init__(trajectory)

        # Define constants
        self.lr = 1.39
        self.lf = 1.55
        self.Ca = 20000
        self.Iz = 25854
        self.m = 1888.6
        self.g = 9.81
```

```
        # Iter 1: 0.1, 0.0, 0.0
        # Iter 2: 0.11435888100000004, 0.0, 0.0178966525
        # Iter 3: reducing Kp a bit to reduce oscillations
        self.lateral_pid = PID(Kp=0.1, Ki=0.0, Kd=0.0178966525, dt=0.032)

        # Iter 1: 29.91, 1.36, 0.03
        # Iter 2: 27.580149049219923, 0.4822354299071474, 0.0734521331616762
        self.longitudinal_pid = PID(27.580149049219923, 0.4822354299071474,
0.0734521331616762, 0.032)

    def update(self, timestep):

        trajectory = self.trajectory

        lr = self.lr
        lf = self.lf
        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)

        # ---------------|Lateral Controller|-------------------------
        # closest node
        cross_track_error, idx = closestNode(X, Y, trajectory)
        closest_node = trajectory[idx]
        next_node = trajectory[(idx + 1) % len(trajectory)]  # safe wrap-around

        # Path tangent vector
        t_dx = next_node[0] - closest_node[0]
        t_dy = next_node[1] - closest_node[1]

        # Vector from path to car
        v_dx = X - closest_node[0]
        v_dy = Y - closest_node[1]

        # Signed cross-track error
        signed_cte = np.sign(t_dx * v_dy - t_dy * v_dx) * cross_track_error

        # PID control
        delta = self.lateral_pid.update(0, signed_cte)
        delta = clamp(delta, -np.pi/6, np.pi/6)
```

```
        # ---------------|Longitudinal Controller|-------------------------
        # Many of the attempt changes were made by adjusting speed
        desired_speed = 10.5
        current_speed = np.sqrt(xdot**2 + ydot**2)
        F = self.longitudinal_pid.update(desired_speed, current_speed)
        F = clamp(F, 0, 15736)

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

Linearization

October 30, 2025

0.1 Linearization

[1]:
```
import sympy as sp
from IPython.display import display

# Enable LaTeX pretty printing
sp.init_printing(use_latex=True)

# Define symbols for lateral and longitudinal dynamics with LaTeX names
y = sp.symbols('y')
y_dot = sp.symbols(r'\dot{y}')
psi = sp.symbols(r'\psi')
psi_dot = sp.symbols(r'\dot{\psi}')
x = sp.symbols('x')
x_dot = sp.symbols(r'\dot{x}')  # longitudinal velocity with LaTeX dot

delta, F = sp.symbols(r'\delta F')  # inputs

# Vehicle parameters
m, Iz, lf, lr, Ca, fm, g = sp.symbols('m Iz l_f l_r C_alpha f g')  # FIXED: use␣
 ↪C_alpha, not C_\alpha

# Replace vx with x_dot (longitudinal velocity is the second state)
vx = x_dot

# --- Lateral nonlinear dynamics with vx = x_dot ---
y_ddot = -psi_dot * vx + (2*Ca/m) * (sp.cos(delta)*(delta - (y_dot +␣
 ↪lf*psi_dot)/vx) - (y_dot - lr*psi_dot)/vx)
psi_ddot = (2*lf*Ca/Iz) * (delta - (y_dot + lf*psi_dot)/vx) - (2*lr*Ca/Iz) *␣
 ↪(-(y_dot - lr*psi_dot)/vx)

# Lateral state and input vectors
s1 = sp.Matrix([y, y_dot, psi, psi_dot])
u = sp.Matrix([delta, F])

# Lateral nonlinear state derivative
f1 = sp.Matrix([y_dot, y_ddot, psi_dot, psi_ddot])
```

```
# Jacobians lateral
A1 = f1.jacobian(s1)
B1 = f1.jacobian(u)

# Small angle approx cos(delta) ~ 1
A1_lin = A1.subs(sp.cos(delta), 1)
B1_lin = B1.subs(sp.cos(delta), 1)

print("A1 (lateral system matrix):")
display(A1_lin)

print("\nB1 (lateral input matrix):")
display(B1_lin)

# --- Longitudinal nonlinear dynamics ---
# Assume lateral states y, y_dot, psi, psi_dot are constant during longitudinal␣
 ↪linearization
# Longitudinal states and inputs
s2 = sp.Matrix([x, x_dot])

# Define disturbance term from lateral coupling and rolling resistance
disturbance = psi_dot * y_dot - (fm * g) / m

# Longitudinal nonlinear state derivative
f2 = sp.Matrix([
    x_dot,
    disturbance + (1/m) * (F)  # no dependence on delta for x_ddot here
])

# Jacobians longitudinal
A2 = f2.jacobian(s2)
B2 = f2.jacobian(u)  # u = [delta, F], delta no effect on x_ddot

print("\nA2 (longitudinal system matrix):")
display(A2)

print("\nB2 (longitudinal input matrix):")
display(B2)

print("\nDisturbance term:")
display(disturbance)
```

```
A1 (lateral system matrix):
```

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{\dot{x}m} & 0 & \frac{2C_\alpha\left(-\frac{l_f}{\dot{x}}+\frac{l_r}{\dot{x}}\right)}{m}-\dot{x} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2C_\alpha l_f}{Iz\dot{x}}+\frac{2C_\alpha l_r}{Iz\dot{x}} & 0 & -\frac{2C_\alpha l_f^2}{Iz\dot{x}}-\frac{2C_\alpha l_r^2}{Iz\dot{x}} \end{bmatrix}$$

B1 (lateral input matrix):

$$\begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha\left(-\left(\delta-\frac{\dot{\psi}l_f+\dot{y}}{\dot{x}}\right)\sin(\delta)+1\right)}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{Iz} & 0 \end{bmatrix}$$

A2 (longitudinal system matrix):

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

B2 (longitudinal input matrix):

$$\begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix}$$

Disturbance term:

$$\dot{\psi}\dot{y}-\frac{fg}{m}$$