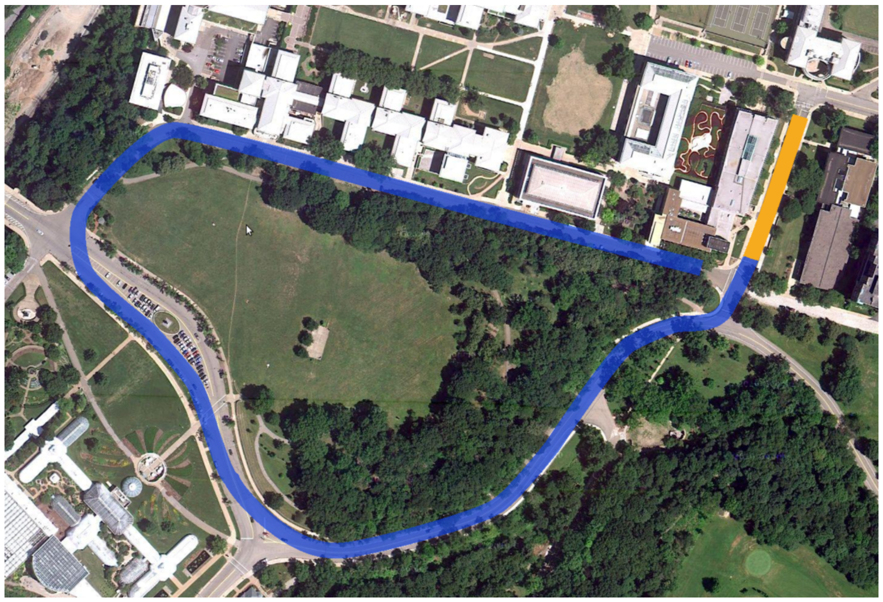
Figure 4: Buggy track[3]

4 P1: Problems

Exercise 1. Model Linearization. As mentioned in class, model linearization is always the first step for non-linear control. During this assignment, you will approximate the given model with a linear model.

Since the longitudinal term $\dot{x}$ is non-linear in the lateral dynamics, we can simplify the controller by controlling the lateral and longitudinal states separately. You are required to write the system dynamics in linear forms as $\dot{s_1} = A_1 s_1 + B_1 u$ and $\dot{s_2} = A_2 s_2 + B_2 u$ in terms of the following given input and states:

$$u = \begin{bmatrix} \delta \\ F \end{bmatrix}, s_1 = \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix}, s_2 = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Hints:

- When linearizing lateral dynamics, you can assume longitudinal states to be constant, so they can be included in the linear model. Vice versa for the longitudinal dynamics.
- For longitudinal dynamics, if you have some terms that cannot be combined into A_2 and B_2, you can write them as disturbances and put it as an extra term.

Solution:

If you linearize your system with reasonable assumptions, you will get full points.

Under the small angle assumption, we have $\cos(\delta) = 1$. Then, by clustering the variables with respect to the state variables, we have the following for s_1:

$$\ddot{y} = \dot{y}\left(\frac{-4C_\alpha}{m\dot{x}}\right) + \dot{\psi}\left(-\dot{x} + \frac{2C_\alpha(l_r - l_f)}{m\dot{x}}\right) + \delta\left(\frac{2C_\alpha}{m}\right)$$
$$\ddot{\psi} = \dot{y}\left(\frac{2(l_r - l_f)C_\alpha}{I_z\dot{x}}\right) + \dot{\psi}\left(-\frac{2\left(l_f^2 + l_r^2\right)C_\alpha}{I_z\dot{x}}\right) + \delta\left(\frac{2l_fC_\alpha}{I_z}\right)$$

For s_2:

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

Write them in matrix form:

$$\frac{d}{dt}s_1 = \frac{d}{dt}\begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-4C_\alpha}{m\dot{x}} & 0 & -\dot{x} + \frac{2C_\alpha(l_r - l_f)}{m\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2(l_r - l_f)C_\alpha}{I_z\dot{x}} & 0 & -\frac{2\left(l_f^2 + l_r^2\right)C_\alpha}{I_z\dot{x}} \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2l_fC_\alpha}{I_z} & 0 \end{bmatrix} \begin{bmatrix} \delta \\ F \end{bmatrix}$$

$$\frac{d}{dt}s_2 = \frac{d}{dt}\begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \begin{bmatrix} \delta \\ F \end{bmatrix} + \begin{bmatrix} 0 \\ \dot{\psi}\dot{y} - fg \end{bmatrix}$$

Exercise 2. Controller Synthesis in Simulation. The driver functions that control the car take the desired steering angle δ and a throttle input - ranging from 0 to 1 - which is derived from the desired longitudinal force F.

For this question, you have to design a PID longitudinal controller and a PID lateral controller for the vehicle. A PID is an error-based controller that requires tuning proportional, integral, and derivative gains. As a PID allows us to minimize the error between a set point and process variable without deeper knowledge of the system model, we will not need our result from Exercise 1 (though it will be useful in future project parts).

Design the two controllers in `your_controller.py`. You can make use of Webots' built-in code editor, or use your own.

Check the performance of your controller by running the Webots simulation. You can press the play button in the top menu to start the simulation in real-time, the fast-forward button to run the simulation as quickly as possible, and the triple fast-forward to run the simulation without rendering (any of these options is acceptable, and the faster options may be better for quick tests). If you complete the track, the scripts will generate a performance plot via `matplotlib`. This plot contains a visualization of the car's trajectory, and also shows the variation of states with respect to time.

Submit `your_controller.py` and the final completion plot as described on the title page. Your controller is **required** to achieve the following performance criteria to receive full points:

1. Time to complete the loop = 400 s
2. Maximum deviation from the reference trajectory = 10.0 m
3. Average deviation from the reference trajectory = 5 m

Some hints that may be useful:

- Using a PID controller requires storing variables between method calls. Python allows this through use of the `self` preface (in other words, making them class member variables). Think about which variables you use in your controller that you will need to store, and be sure to add them to `CustomController`'s `__init__` method so they are initialized.
- If you are tuning your lateral controller for a point on the trajectory that is the closest to the vehicle, the controller may struggle on sudden turns. Think about how to add some mechanism that "looks ahead" of your current position before calculating a control command.
- Functions in `util.py` might be useful. For example, `closestNode` returns the absolute value of the cross-track error, which is the distance from the vehicle's center of gravity to the nearest waypoint on the trajectory. In addition, the function also returns the

index of the closest waypoint to the vehicle. Also you may need to use `wrapToPi` when you deal with angles.

- If your car does not perform well, it may just be that it requires tuning. You may already have some experience tuning PID controllers, but if not, viewing online resources is recommended. See this Wiki link [4] for more background in the process of manual tuning.

Solution:

```
# Fill the respective function to implement the PID controller

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

# Custom Controller Class
class CustomController(BaseController):

    def __init__(self, trajectory):

        super().__init__(trajectory)

        # Initialize necessary variables
        self.integralPsiError = 0
        self.previousPsiError = 0
        self.previousXdotError = 0

    def update(self, timestep):

        trajectory = self.trajectory

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


        # ---------------|Lateral Controller|-------------------------
        # Find the closest node to the vehicle
        _, node = closestNode(X, Y, trajectory)

        # Choose a node that is ahead of our current node based on index
        forwardIndex = 50
```

```
# Two distinct ways to calculate the desired heading angle:
# 1. Find the angle between a node ahead and the car's current
↪  position
# 2. Find the angle between two nodes - one ahead, and one closest
# The first method has better overall performance, as the second
↪  method
# can read zero error when the car is not actually on the
↪  trajectory

# We use a try-except so we don't attempt to grab an index that is
↪  out of scope
# 1st method
try:
    psiDesired = np.arctan2(trajectory[node+forwardIndex,1]-Y, \
                            trajectory[node+forwardIndex,0]-X)
except:
    psiDesired = np.arctan2(trajectory[-1,1]-Y, \
                           trajectory[-1,0]-X)

# 2nd method
# try:
    # psiDesired =
    ↪  np.arctan2(trajectory[node+forwardIndex,1]-trajectory[node,1],
    ↪  \
            # trajectory[node+forwardIndex,0]-trajectory[node,0])
# except:
    # psiDesired = np.arctan2(trajectory[-1,1]-trajectory[node,1],
    ↪  \
            # trajectory[-1,0]-trajectory[node,0])
# PID gains
kp = 1
ki = 0.005
kd = 0.001

# Calculate difference between desired and actual heading angle
psiError = wrapToPi(psiDesired-psi)

self.integralPsiError += psiError
derivativePsiError = psiError - self.previousPsiError
delta = kp*psiError + ki*self.integralPsiError*delT +
↪  kd*derivativePsiError/delT
delta = wrapToPi(delta)

# ---------------|Longitudinal
↪  Controller|-------------------------
```

```
# PID gains
kp = 200
ki = 10
kd = 30

# Reference value for PID to tune to
desiredVelocity = 6

xdotError = (desiredVelocity - xdot)
self.integralXdotError += xdotError
derivativeXdotError = xdotError - self.previousXdotError
self.previousXdotError = xdotError

F = kp*xdotError + ki*self.integralXdotError*delT +
↪  kd*derivativeXdotError/delT

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

5 Reference

1. Rajamani Rajesh. Vehicle Dynamics and Control. Springer Science & Business Media, 2011.

2. Kong Jason, et al. “Kinematic and dynamic vehicle models for autonomous driving control design.” Intelligent Vehicles Symposium, 2015.

3. cmubuggy.org, https://cmubuggy.org/reference/File:Course_hill1.png

4. “PID Controller - Manual Tuning.” *Wikipedia*, Wikimedia Foundation, August 30th, 2020. https://en.wikipedia.org/wiki/PID_controller#Manual_tuning