Project: Part 2

24-677 Special Topics: Linear Control Systems

Prof. D. Zhao

Due: Nov 10, 2020, 11:59 pm. Submit within the deadline.

- Your online version and its timestamp will be used for assessment.
- We will use Gradescope to grade. The link is on the panel of CANVAS. If you are confused about the tool, post your questions on Campuswire.
- Submit **your_controller.py** to Gradescope under **Programming-P2** and your solutions in **.pdf** format to **Project-P2**. Insert the performance plot image in the **.pdf**. We will test **your_controller.py** and manually check all answers.
- We will make extensive use of Webots, an open-source robotics simulation software, for this project. Webots is available here for Windows, Mac, and Linux.
- For Python usage with Webots, please see the Webots page on Python. Note that you may have to reinstall libraries like `numpy, matplotlib, scipy,` etc. for the environment you use Webots in.
- Please familiarize yourself with Webots documentation, specifically their User Guide and their Webots for Automobiles section, if you encounter difficulties in setup or use. It will help to have a good understanding of the underlying tools that will be used in this assignment. To that end, completing at least Tutorial 1 in the user guide is highly recommended.
- If you have issues with Webots that are beyond the scope of the documentation (e.g. the software runs too slow, crashes, or has other odd behavior), please let the TAs know via Campuswire. We will do our best to help.
- We advise you to start with the assignment early. All the submissions are to be done before the respective deadlines of each assignment. For information about the late days and scale of your Final Grade, refer to the Syllabus in Canvas.

1 Introduction

In this part of the project, you will complete the following two assignments:

1. Check the controllability and stabilizability of the linearized system
2. Design a lateral full-state feedback controller

[Remember to submit the write-up, plots, and codes on Gradescope.]

2 Model

The error-based linearized state-space for the lateral dynamics is as follows.
e_1 is the distance to the center of gravity of the vehicle from the reference trajectory.
e_2 is the orientation error of the vehicle with respect to the reference trajectory.

$$\frac{d}{dt}\begin{bmatrix} e_1 \\ \dot{e_1} \\ e_2 \\ \dot{e_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & \frac{4C_\alpha}{m} & -\frac{2C_\alpha(l_f-l_r)}{m\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2C_\alpha(l_f-l_r)}{I_z\dot{x}} & \frac{2C_\alpha(l_f-l_r)}{I_z} & -\frac{2C_\alpha(l_f^2+l_r^2)}{I_z\dot{x}} \end{bmatrix}\begin{bmatrix} e_1 \\ \dot{e_1} \\ e_2 \\ \dot{e_2} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix}\begin{bmatrix} \delta \\ F \end{bmatrix} + \begin{bmatrix} o \\ -\frac{2C_\alpha(l_f-l_r)}{m\dot{x}} - \dot{x} \\ 0 \\ -\frac{2C_\alpha(l_f^2+l_r^2)}{I_z\dot{x}} \end{bmatrix}\dot{\psi}_{des}$$

In lateral vehicle dynamics, $\dot{\psi}_{des}$ is a time-varying disturbance in the state space equation. Its value is proportional to the longitudinal speed when the radius of the road is constant. When deriving the error-based state space model for controller design, $\dot{\psi}_{des}$ can be safely assumed to be zero.

$$\frac{d}{dt}\begin{bmatrix} e_1 \\ \dot{e_1} \\ e_2 \\ \dot{e_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & \frac{4C_\alpha}{m} & -\frac{2C_\alpha(l_f-l_r)}{m\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2C_\alpha(l_f-l_r)}{I_z\dot{x}} & \frac{2C_\alpha(l_f-l_r)}{I_z} & -\frac{2C_\alpha(l_f^2+l_r^2)}{I_z\dot{x}} \end{bmatrix}\begin{bmatrix} e_1 \\ \dot{e_1} \\ e_2 \\ \dot{e_2} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix}\begin{bmatrix} \delta \\ F \end{bmatrix}$$

For the longitudinal control:

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

Assuming $\dot{\psi} = 0$:

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

3 P2: Problems [Due Nov. 10, 2020]

Exercise 1. Considering the linearized, error-based state space system for the vehicle in the Model section above:

1. Check the controllability and observability of the system at the following longitudinal velocities: 2 m/s, 5 m/s and 8 m/s.

2. For longitudinal velocities v from 1 m/s to 40 m/s, plot the following:

   (a) $log_{10}(\frac{\sigma_1}{\sigma_n})$ versus v (m/s), where σ_i is the ith singular value of the controllability matrix P ($i = 1, 2, ..., n$). (In other words, what is the logarithm of the greatest singular value divided by the smallest?)

   (b) $Re(p_i)$ versus v (m/s), where Re is real part and p_i is the ith pole of the continuous state space system. [Use 4 subplots, one for each of the 4 poles]

   What conclusions can you draw about the overall controllability and stability of the system in observing these two plots?

[Submit you answers in the **.pdf** file and also submit the Python script. The Python script should be named **Q1.py**]

Solution

1.
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal

# 1.1 - Check controllability and observability

# Vehicle dynamics parameters
lr = 1.39
lf = 1.55
Ca = 20000
Iz = 25854
m = 1888.6
g = 9.81
f = 1

V = [2,5,8]

for idx, val in enumerate(V):

    xdot = val
```

```
    # State-space equation
    A =
    ↪  np.array([[0,1,0,0],[0,-4*Ca/(m*xdot),4*Ca/m,2*Ca*(lr-lf)/(m*xdot)]
    ↪  \
                  ↪  ,[0,0,0,1],[0,(2*Ca)*(lr-lf)/(Iz*xdot),(2*Ca)*(lf-lr)/Iz,
                  ↪  \
                  (-2*Ca)*(lf**2 + lr**2)/(Iz*xdot)]])
    B = np.array([[0],[2*Ca/m],[0],[2*Ca*lf/Iz]])
    # phidot_des term is ignored for B
    C = np.identity(4)
    D = np.zeros((4,1))

    # Check controllability by manually building P
    P1 = B
    P2 = np.dot(A,B)
    P3 = np.dot(np.linalg.matrix_power(A,2),B)
    P4 = np.dot(np.linalg.matrix_power(A,3),B)

    # Check observability by manually building Q
    Q1 = C
    Q2 = np.dot(C,A)
    Q3 = np.dot(C,np.linalg.matrix_power(A,2))
    Q4 = np.dot(C,np.linalg.matrix_power(A,3))

    P = np.concatenate((P1,P2,P3,P4),axis=1)
    Q = np.vstack((Q1,Q2,Q3,Q4))

    # Determine the rank of both P and Q
    print('P for {} m/s has rank
    ↪  {}'.format(xdot,np.linalg.matrix_rank(P)))
    print('Q for {} m/s has rank
    ↪  {}'.format(xdot,np.linalg.matrix_rank(Q)))

print("Thus the system is controllable and observable for every value
↪  of Vx tested")


# 1.2 - Graphs of singular value ratios and poles

V = np.linspace(1, 40, 40)
logsigma_arr = []
pole1 = []
pole2 = []
pole3 = []
```

```
pole4 = []

for idx, val in enumerate(V):

    xdot = val

    # State-space equation
    A =
    ↪  np.array([[0,1,0,0],[0,-4*Ca/(m*xdot),4*Ca/m,2*Ca*(lr-lf)/(m*xdot)]
    ↪  \

                  ↪  ,[0,0,0,1],[0,(2*Ca)*(lr-lf)/(Iz*xdot),(2*Ca)*(lf-lr)/Iz,
                  ↪  \
                  (-2*Ca)*(lf**2 + lr**2)/(Iz*xdot)]])
    B = np.array([[0],[2*Ca/m],[0],[2*Ca*lf/Iz]])
    C = np.identity(4)
    D = np.zeros((4,1))

    # Manually build P
    P1 = B
    P2 = np.dot(A,B)
    P3 = np.dot(np.linalg.matrix_power(A,2),B)
    P4 = np.dot(np.linalg.matrix_power(A,3),B)
    P = np.concatenate((P1,P2,P3,P4),axis=1)

    # Get first and last singular values of P
    [u,sigma,v] = np.linalg.svd(P)
    sigma_1 = sigma[0]
    sigma_n = sigma[-1]

    # Determine logarithm of their ratio
    logsigma = np.log10(sigma_1/sigma_n)
    logsigma_arr = np.append(logsigma_arr,logsigma)

    # Get eigenvalues of A, which are the poles of the system
    [lam,v] = np.linalg.eig(A)
    # Only get the real components of each pole for plotting
    realroots = lam.real
    pole1 = np.append(pole1,realroots[0])
    pole2 = np.append(pole2,realroots[1])
    pole3 = np.append(pole3,realroots[2])
    pole4 = np.append(pole4,realroots[3])

plt.title('log$_{10}(\sigma_1/\sigma_n)$ vs. V')
plt.xlabel('V (m/s)')
```

```python
plt.ylabel('log$_{10}(\sigma_1/\sigma_n)$')
plt.plot(V, logsigma_arr)


fig = plt.figure()

plt.subplot(221)
plt.title('Re($p_1$)')
plt.plot(V, pole1)

plt.subplot(222)
plt.title('Re($p_2$)')
plt.plot(V, pole2)

plt.subplot(223)
plt.title('Re($p_3$)')
plt.plot(V, pole3)

plt.subplot(224)
plt.title('Re($p_4$)')
plt.plot(V, pole4)

fig.tight_layout()
plt.show()
```

```
P for 2 m/s has rank 4
Q for 2 m/s has rank 4
P for 5 m/s has rank 4
Q for 5 m/s has rank 4
P for 8 m/s has rank 4
Q for 8 m/s has rank 4
Thus the system is controllable and observable for every value of Vx tested
```

Figure 1: The system is both controllable and observable at the values tested.

2. (a) The ratio of singular values of the controllability matrix reflects the defectiveness of the system. The defectiveness of the system is inversely proportional to the ratio, i.e. the smaller the ratio, the less the system is likely to be defective. Therefore, the system is comparatively more controllable in the lateral direction at higher longitudinal velocities. This fits with the intuition that it is easier to make steering changes in a car when the car is traveling at a faster speed. If the car is traveling at a very low speed, it is much more difficult to control the car in the lateral direction.

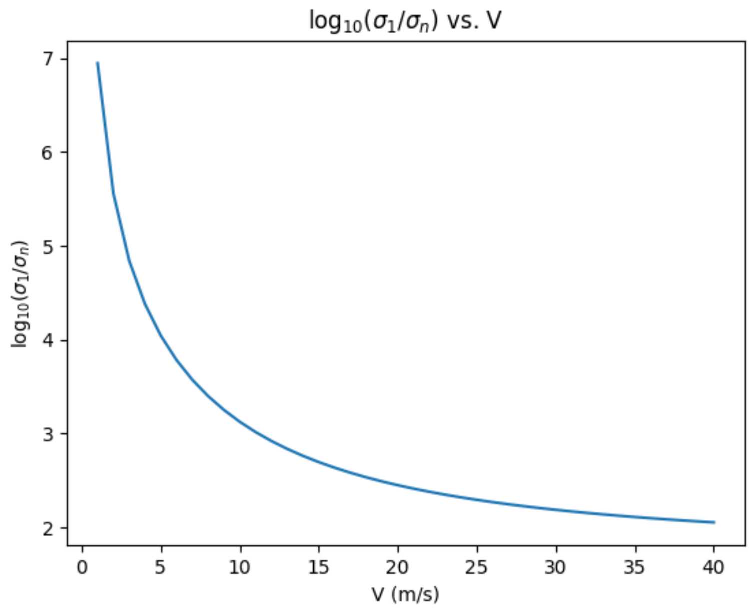

Figure 2: $log_{10}(\frac{\sigma_1}{\sigma_n})$ versus $\dot{x}$ (m/s)

(b) The system is a second-order system with two poles and two zeros. With an increase in the longitudinal velocity, the conjugate pole pairs move closer to the imaginary axis, indicating that the system tends to be less stable as the velocity increases. Notably, one of the poles goes above the imaginary axis at a value of $\dot{x} \approx 34$ m/s - in other words, the system becomes unstable at this velocity.

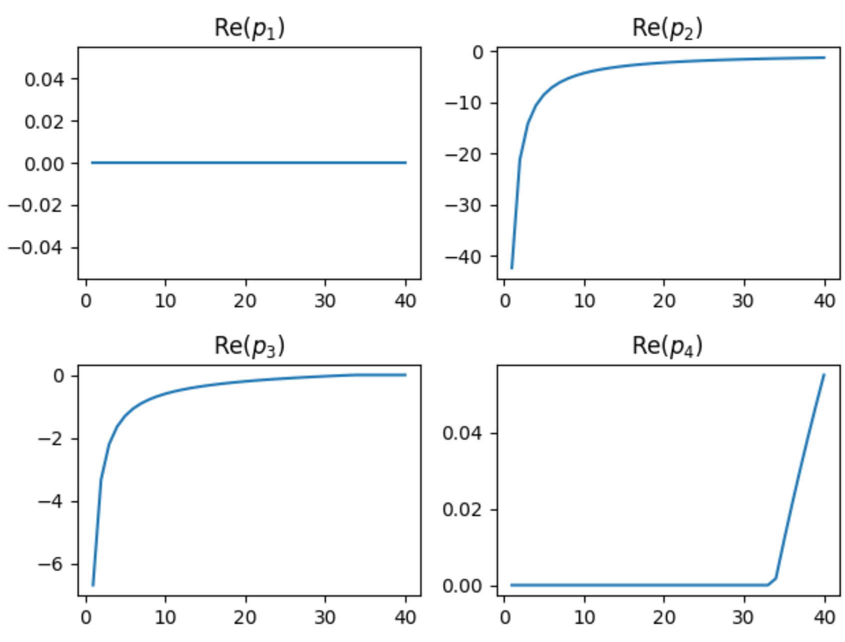

Figure 3: $Re(p_i)$ versus $\dot{x}$ (m/s)

In conclusion, as the longitudinal velocity increases, it is easier to make a change in the lateral state (controllability) though there is a higher risk of the system becoming unstable.

When $l_r C_a < l_f C_a$, as the longitudinal velocity increases, less control input is needed to steer the car. This is true until the car reaches a velocity (denoted as the critical velocity), where even with zero steering angle, the car steers in a direction. For our system, this critical velocity appears to be at $\dot{x} \approx 34$ m/s. If the velocity continues to increase, a left steering input is required to turn to the right. This phenomenon is called oversteering. If you would like to learn more about this behavior, please visit: https://www.youtube.com/watch?v=K4yaXb8nOW4

Exercise 2. For the lateral control of the vehicle, design a state feedback controller using pole placement. Tune the poles of the closed loop system such that it can achieve the performance criteria mentioned below.

You can reuse your longitudinal PID controller from part 1 of this project, or even improve upon it. However, it may require retuning based on observed performance.

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 = 350 s
2. Maximum deviation from the reference trajectory = 9.0 m
3. Average deviation from the reference trajectory = 4.5 m

Some hints that may be useful:

- The `signal` subpackage within scipy is required for this part. Please investigate which functions you will need to use. The main goal is to calculate a gain matrix K such that $-Kx = u$, where x is the states and u is the control input.
- It is somewhat difficult to tune pole-placement controllers. Learning optimal control in the next submodule will fortunately make this task much easier. Remember - poles should be negative if the system is stable. Poles can also be complex, where an imaginary number is denoted with `j`, e.g. `-3+1j`. If you use a complex pole, you must also include its complex conjugate.
- The controller itself can be continuous or discrete - it is your choice whether to discretize the system or not.

Solution

```python
# Fill in the respective functions to implement the full-state feedback
↪  controller

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

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

    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|-------------------------
        # Use the results of linearization to create a state-space model
        A =
        ↪  np.array([[0,1,0,0],[0,-4*Ca/(m*xdot),4*Ca/m,2*Ca*(lr-lf)/(m*xdot)],
            [0,0,0,1], [0,(2*Ca)*(lr-lf)/(Iz*xdot), (2*Ca)*(lf-lr)/Iz, \
            (-2*Ca)*(lf**2 + lr**2)/(Iz*xdot)]])
        B = np.array([[0],[2*Ca/m],[0],[2*Ca*lf/Iz]])
```

```
C = np.eye(4)
D = np.zeros((4,1))

# Choose pole locations (many valid pole locations are possible)
poles = np.array([-5,-4,-3,-1])

# Place poles and compute the gain matrix
fsf = signal.place_poles(A, B, poles)
K = fsf.gain_matrix

# 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 = 100

# Determine desired heading angle and e1 using two nodes - one
↪  ahead, and one closest
# We use a try-except so we don't attempt to grab an index that is
↪  out of scope

# To define our error-based states, we use definitions from
↪  documentation.
# Please see page 34 of Rajamani Rajesh's book "Vehicle Dynamics
↪  and Control", which
# is available online through the CMU library, for more
↪  information.
# It is important to note that numerical derivatives of e1 and e2
↪  will also work well.
try:
    psiDesired =
    ↪  np.arctan2(trajectory[node+forwardIndex,1]-trajectory[node,1],
    ↪  trajectory[node+forwardIndex,0]-trajectory[node,0])
    e1 =  (Y - trajectory[node+forwardIndex,1])*np.cos(psiDesired) -
    ↪  (X - trajectory[node+forwardIndex,0])*np.sin(psiDesired)

except:
    psiDesired = np.arctan2(trajectory[-1,1]-trajectory[node,1],
    ↪  trajectory[-1,0]-trajectory[node,0])
    e1 =  (Y - trajectory[-1,1])*np.cos(psiDesired) - (X -
    ↪  trajectory[-1,0])*np.sin(psiDesired)

e1dot = ydot + xdot*wrapToPi(psi - psiDesired)
e2 = wrapToPi(psi - psiDesired)
```

```
e2dot = psidot # This definition would be psidot - psidotDesired if
↪  calculated from curvature

# Assemble error-based states into array
states = np.array([e1,e1dot,e2,e2dot])

# Calculate delta via u = -Kx
delta = float(-K @ states)

# ---------------|Longitudinal
↪  Controller|-------------------------
# PID gains
kp = 200
ki = 10
kd = 30

# Reference value for PID to tune to
desiredVelocity = 8

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
```

4 Appendix

(Already covered in P1)

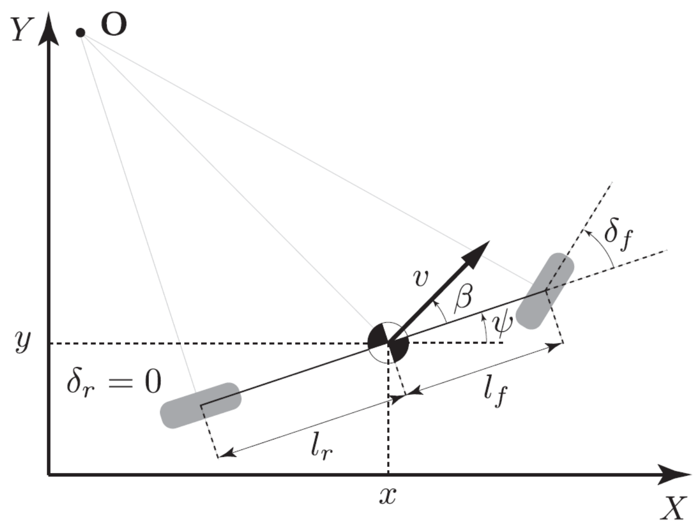

Figure 4: Bicycle model[2]

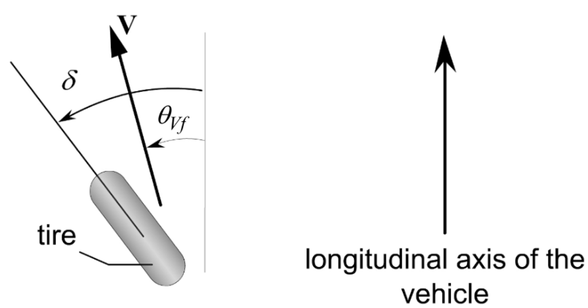

Figure 5: Tire slip-angle[2]

We will make use of a bicycle model for the vehicle, which is a popular model in the study of vehicle dynamics. Shown in Figure 4, the car is modeled as a two-wheel vehicle with two degrees of freedom, described separately in longitudinal and lateral dynamics. The model parameters are defined in Table 2.

4.1 Lateral dynamics

Ignoring road bank angle and applying Newton's second law of motion along the y-axis:

$$ma_y = F_{yf} \cos \delta_f + F_{yr}$$

where $a_y = \left(\frac{d^2y}{dt^2}\right)_{inertial}$ is the inertial acceleration of the vehicle at the center of geometry in the direction of the y axis, F_{yf} and F_{yr} are the lateral tire forces of the front and rear wheels, respectively, and δ_f is the front wheel angle, which will be denoted as δ later. Two terms contribute to a_y: the acceleration $\ddot{y}$, which is due to motion along the y-axis, and the centripetal acceleration. Hence:

$$a_y = \ddot{y} + \dot{\psi}\dot{x}$$

Combining the two equations, the equation for the lateral translational motion of the vehicle is obtained as:

$$\ddot{y} = -\dot{\psi}\dot{x} + \frac{1}{m}(F_{yf}\cos\delta + F_{yr})$$

Moment balance about the axis yields the equation for the yaw dynamics as

$$\ddot{\psi}I_z = l_f F_{yf} - l_r F_{yr}$$

The next step is to model the lateral tire forces F_{yf} and F_{yr}. Experimental results show that the lateral tire force of a tire is proportional to the "slip-angle" for small slip-angles when vehicle's speed is large enough - i.e. when $\dot{x} \geq 0.5$ m/s. The slip angle of a tire is defined as the angle between the orientation of the tire and the orientation of the velocity vector of the vehicle. The slip angle of the front and rear wheel is

$$\begin{aligned}
\alpha_f &= \delta - \theta_{Vf} \\
\alpha_r &= -\theta_{Vr}
\end{aligned}$$

where θ_{Vp} is the angle between the velocity vector and the longitudinal axis of the vehicle, for $p \in \{f, r\}$. A linear approximation of the tire forces are given by

$$\begin{aligned}
F_{yf} &= 2C_\alpha \left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) \\
F_{yr} &= 2C_\alpha \left(-\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right)
\end{aligned}$$

where C_α is called the cornering stiffness of the tires. If $\dot{x} < 0.5$ m/s, we just set F_{yf} and F_{yr} both to zeros.

4.2 Longitudinal dynamics

Similarly, a force balance along the vehicle longitudinal axis yields:

$$\begin{aligned}
\ddot{x} &= \dot{\psi}\dot{y} + a_x \\
ma_x &= F - F_f \\
F_f &= fmg
\end{aligned}$$

where F is the total tire force along the x-axis, and F_f is the force due to rolling resistance at the tires, and f is the friction coefficient.

4.3 Global coordinates

In the global frame we have:

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

4.4 System equation

Gathering all of the equations, if $\dot{x} \geq 0.5$ m/s, we have:

$$\begin{aligned}
\ddot{y} &= -\dot{\psi}\dot{x} + \frac{2C_\alpha}{m}(\cos\delta\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) - \frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}) \\
\ddot{x} &= \dot{\psi}\dot{y} + \frac{1}{m}(F - fmg) \\
\ddot{\psi} &= \frac{2l_f C_\alpha}{I_z}\left(\delta - \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) - \frac{2l_r C_\alpha}{I_z}\left(-\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right) \\
\dot{X} &= \dot{x}\cos\psi - \dot{y}\sin\psi \\
\dot{Y} &= \dot{x}\sin\psi + \dot{y}\cos\psi
\end{aligned}$$

otherwise, since the lateral tire forces are zeros, we only consider the longitudinal model.

4.5 Measurements

The observable states are:

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

4.6 Physical constraints

The system satisfies the constraints that:

$$|\delta| \leqslant \tfrac{\pi}{6}\ rad$$
$$F \geqslant 0 \text{ and } F \leqslant 15736\ N$$
$$\dot{x} \geqslant 10^{-5}\ m/s$$

Table 1: Model parameters.

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
ψ, $\dot{\psi}$	Body yaw angle, angular speed	rad, rad/s	State
δ or δ_f	Front wheel angle	rad	State
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 intertia	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

4.7 Simulation

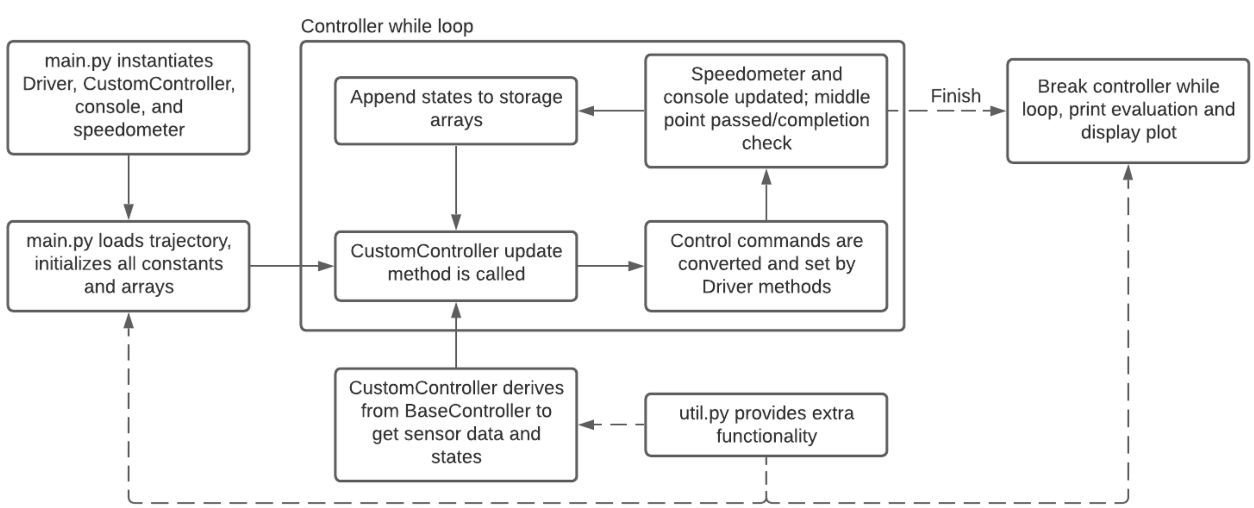

Figure 6: Simulation code flow

Several files are provided to you within the `controllers/main` folder. The `main.py` script initializes and instantiates necessary objects, and also contains the controller loop. This loop runs once each simulation timestep. `main.py` calls `your_controller.py`'s `update` method on each loop to get new control commands (the desired steering angle, δ, and longitudinal force, F). The longitudinal force is converted to a throttle input, and then both control commands are set by Webots internal functions. The additional script `util.py` contains functions to help you design and execute the controller. The full codeflow is pictured in Figure 6.

Please design your controller in the `your_controller.py` file provided for the project part you're working on. Specifically, you should be writing code in the `update` method. Please **do not** attempt to change code in other functions or files, as we will only grade the relevant `your_controller.py` for the programming portion. However, you are free to add to the `CustomController` class's `__init__` method (which is executed once when the `CustomController` object is instantiated).

4.8 BaseController Background

The `CustomController` class within each `your_controller.py` file derives from the `BaseController` class in the `base_controller.py` file. The vehicle itself is equipped with a Webots-generated GPS, gyroscope, and compass that have no noise or error. These sensors are started in the `BaseController` class, and are used to derive the various states of the vehicle. An explanation on the derivation of each can be found in the table below.

Table 2: State Derivation.

Name	Explanation
(X, Y)	From GPS readings
$(\dot{x}, \dot{y})$	From the derivative of GPS readings
ψ	From the compass readings
$\dot{\psi}$	From the gyroscope readings

4.9 Trajectory Data

The trajectory is given in `buggyTrace.csv`. It contains the coordinates of the trajectory as (x, y). The satellite map of the track is shown in Figure 7.

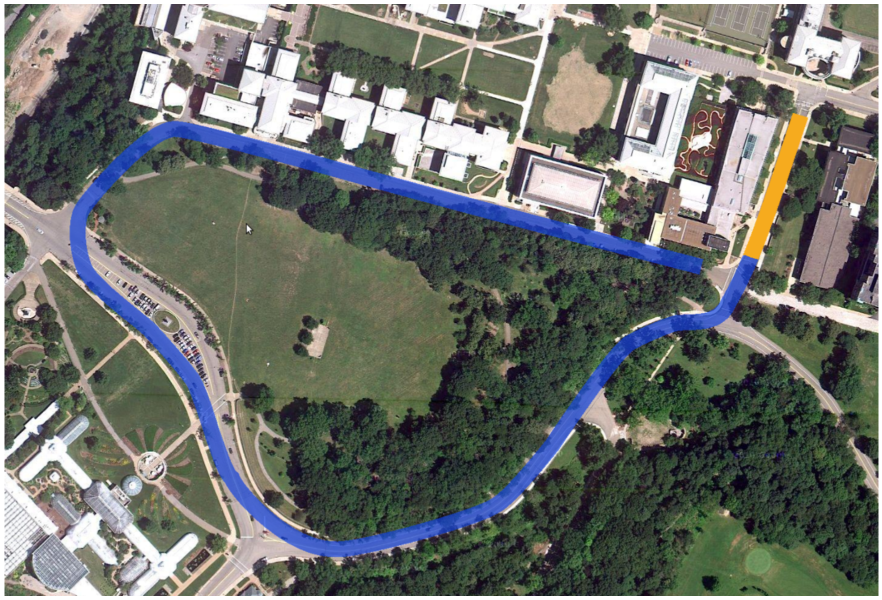

Figure 7: Buggy track[3]

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