Project: Part 2

24-677 Special Topics: Modern Control - Theory and Design

Ryan Wu (ID: weihuanw)

Due: Nov 9, 2023, 11:59 pm.

P2: Problems

Exercise 1.1

Given equation:

$$\frac{d}{dt}\begin{bmatrix} e_1 \\ \dot{e}_1 \\ e_2 \\ \dot{e}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{4C_\alpha}{m\dot{x}} & \frac{4C_\alpha}{m} & -\frac{2C_\alpha(l_f-l_r)}{m\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{2C_\alpha(l_f-l_r)}{I_z\dot{x}} & \frac{2C_\alpha(l_f-l_r)}{I_z} & -\frac{2C_\alpha(l_f^2+l_r^2)}{I_z\dot{x}} \end{bmatrix}\begin{bmatrix} e_1 \\ \dot{e}_1 \\ e_2 \\ \dot{e}_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2C_\alpha l_f}{I_z} & 0 \end{bmatrix}\begin{bmatrix} \delta \\ F \end{bmatrix}$$

Equations for evaluation:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{8000}{1888.6\dot{x}} & 0 & -\dot{x} - \frac{6400}{1888.6\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{6400}{25854\dot{x}} & 0 & -\frac{173384}{25854\dot{x}} \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ \frac{40000}{1888.6} \\ 0 \\ \frac{62000}{25854} \end{bmatrix}$$

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

$$\text{Controllability matrix } P = \begin{bmatrix} B & AB & A^2B & A^3B \end{bmatrix}$$

$$\text{Observability matrix } Q = \begin{bmatrix} C \\ CA \\ CA^2 \\ CA^3 \end{bmatrix}$$

We use rank(P), rank(Q) and the size of matrix A (n=4) to determine the controllability and observability of the given system at the following longitudinal velocities: 2 m/s, 5 m/s and 8 m/s and given variable values. The process was done with Python and the terminal outputs are shown below.

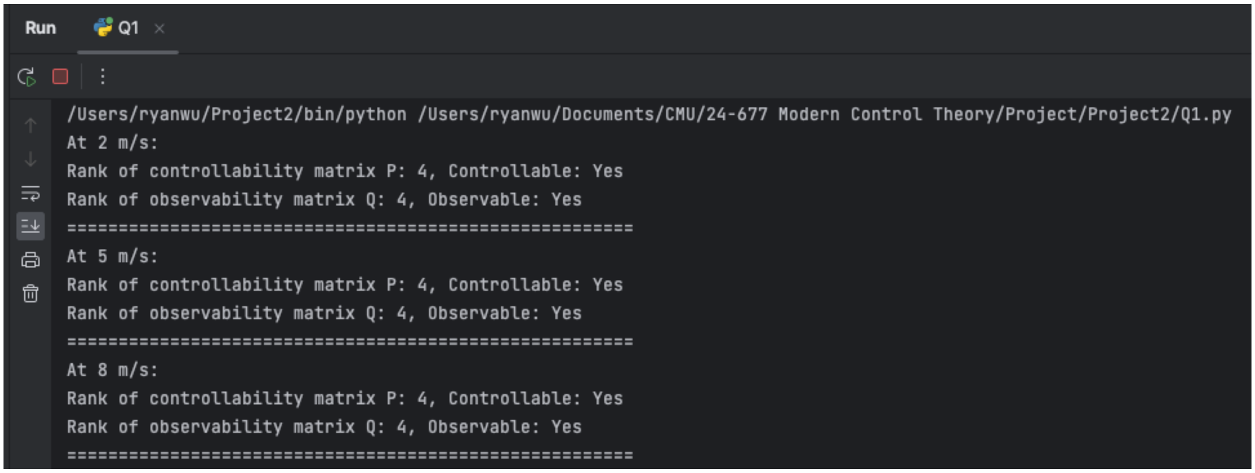


Figure 1. The controllability and observability at each given velocity values.

Exercise 1.2

(a)

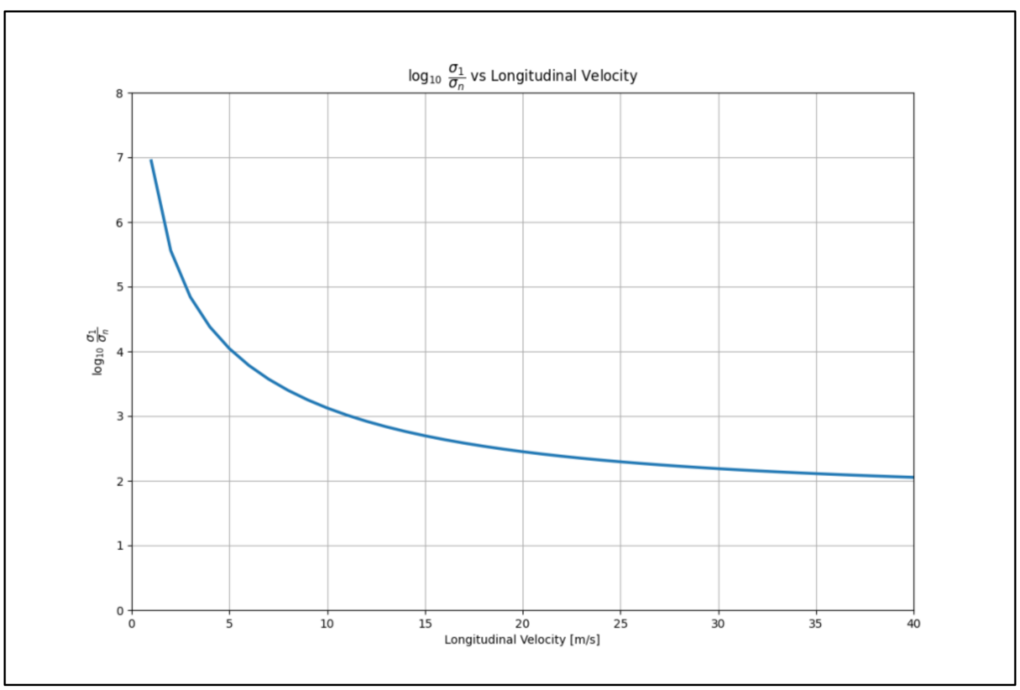


Figure 2. The plot for $log_{10}\frac{\sigma_1}{\sigma_n}$ *versus the desired longitudinal velocity range.*

(b)

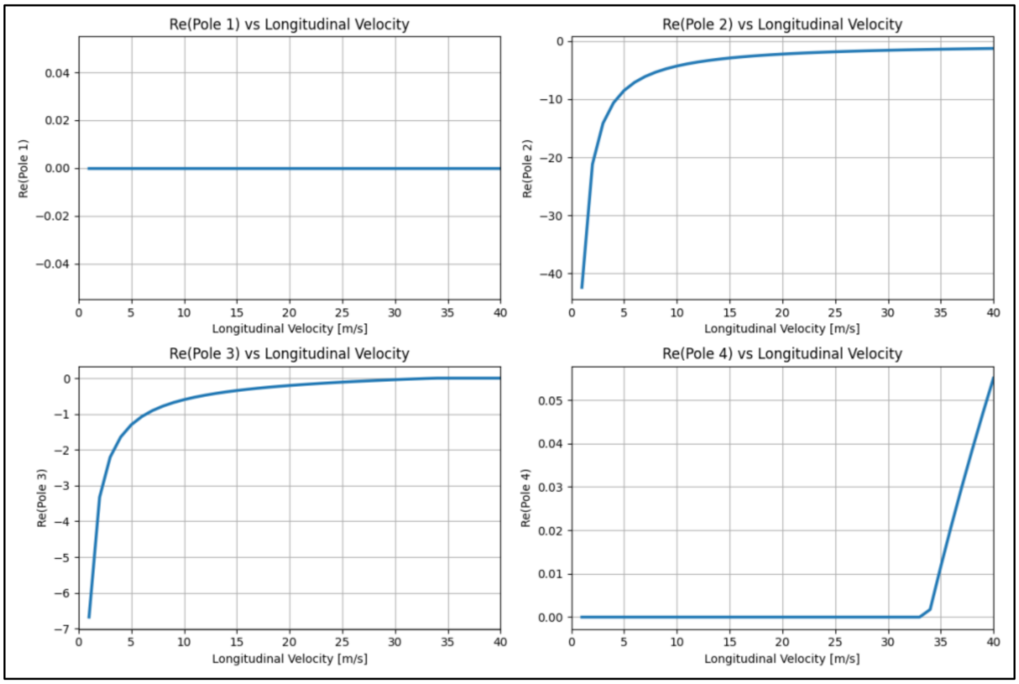


Figure 3. The plots for $Re(p_i)$ versus the desired longitudinal velocity range.

From Figure 2, we can observe that the value of the logarithm of the greatest singular value divided by the smallest converges to a smaller value, which means the system is less likely to be defected and becoming more and more controllable. However, from Figure 3, we can observe as the velocity increased, the poles became less and less negative, which means the overall system is becoming less and less stable.

Exercise 2

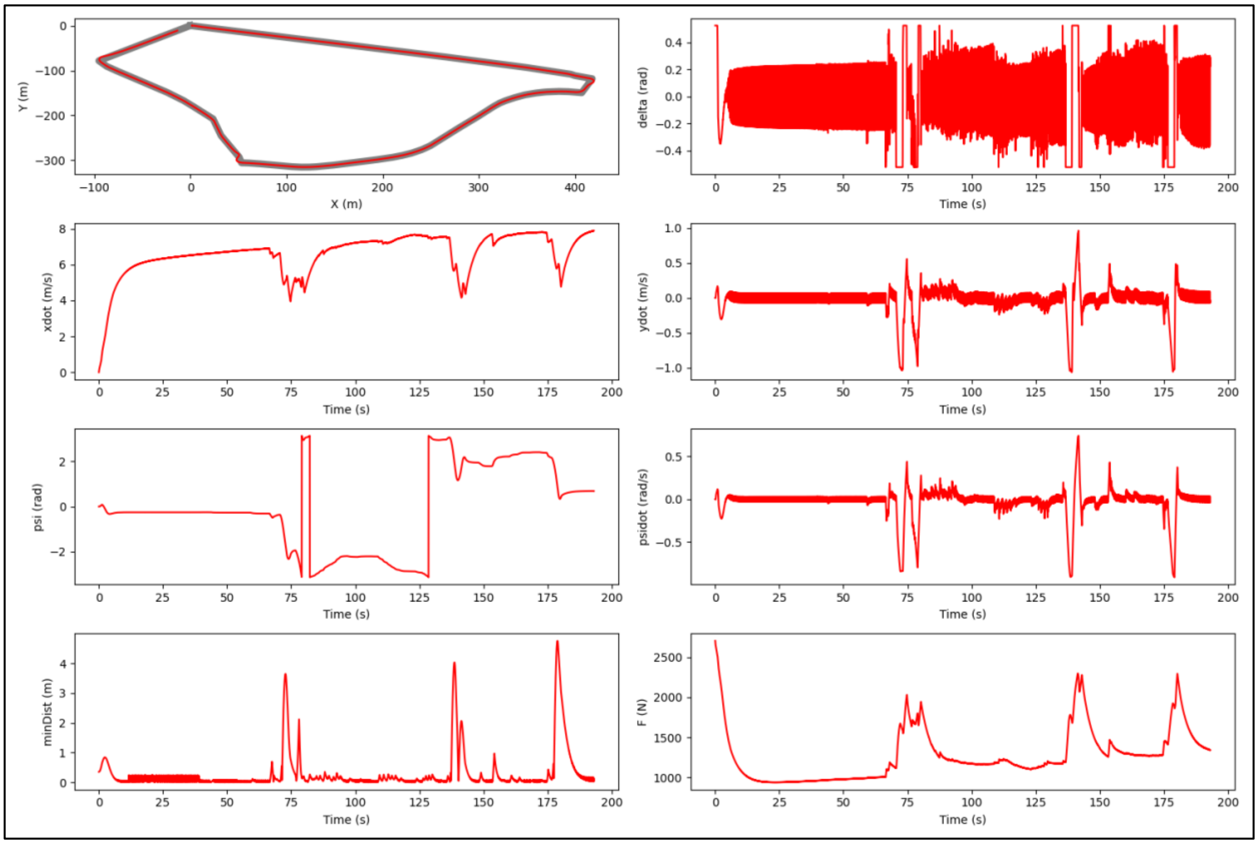
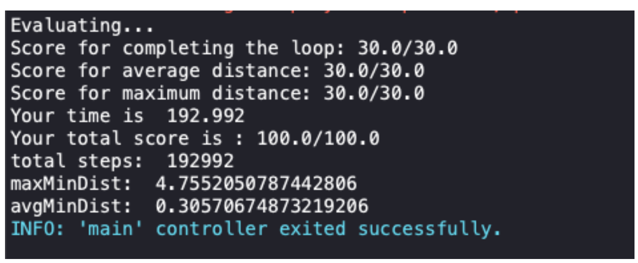


Figure 4. The final completion plot using poles: [-4, -1, -3, -2].



```
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  192.992
Your total score is : 100.0/100.0
total steps:  192992
maxMinDist:  4.7552050787442806
avgMinDist:  0.30570674873219206
INFO: 'main' controller exited successfully.
```

Figure 5. The final completion score using poles: [-4, -1, -3, -2].

```python
# Author: Ryan Wu (ID: weihuanw)
# Carnegie Mellon University
# 24-677 Special Topics: Modern Control - Theory and Design
# Project: Part 2 Exercise 1
# Description: determine controllability and observability of the given system and generate plots
# Due: 11/09/2023 11:59 PM

# import the required libraries
import numpy as np
import matplotlib.pyplot as plt
import control as ctrl

# declaring given variables
Ca = 20000  # Newton
m = 1888.6  # kg
Iz = 25854  # kgm^2
lr = 1.39  # m
lf = 1.55  # m

# given longitudinal velocities for analysis [m/s]
velocities = [2, 5, 8]

# --  Exercise 1.1: check controllability (P) and observability (Q) with velocities [2, 5, 8] m/s -- #

# iterate through each velocity
for velocity in velocities:
    xdot = velocity
    # define the state-space matrices
    A = np.array([[0, 1, 0, 0], [0, -4 * Ca / (m * xdot), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m * xdot)], [0, 0, 0, 1], [0, (-2 * Ca * (lf - lr)) / (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)
```

```python
    D = 0

    # create the state-space model
    sys = ctrl.StateSpace(A, B, C, D)
    # print(sys) # for debugging

    # calculate the rank of the controllability and observability matrices
    P = np.linalg.matrix_rank(ctrl.ctrb(sys.A, sys.B))
    Q = np.linalg.matrix_rank(ctrl.obsv(sys.A, sys.C))
    # check the rank of the controllability and observability matrices
    controllable = P == sys.A.shape[0]
    observable = Q == sys.A.shape[0]

    # print and show the results
    print(f"At {velocity} m/s:")
    print(f"Rank of controllability matrix P: {P}, Controllable: {'Yes' if controllable else 'No'}")
    print(f"Rank of observability matrix Q: {Q}, Observable: {'Yes' if observable else 'No'}")
    print("=" * 55)

# -- Exercise 1.2: plot the log(sigma) vs velocity & real parts vs velocity -- #

# initialize sigma1_values abd real_parts
sigma1_values = []
real_parts = []

# iterate through each velocity
for velocity in range(1, 41):
    xdot = velocity
    # define the state-space matrices
    A = np.array([[0, 1, 0, 0],
                  [0, -4 * Ca / (m * xdot), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m * xdot)],
                  [0, 0, 0, 1],
                  [0, (-2 * Ca * (lf - lr)) / (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 = 0

    # create the state-space model
    sys = ctrl.StateSpace(A, B, C, D)

    # calculate the logarithm of the greatest singular value over the least singular value
    singular_values = np.linalg.svd(ctrl.ctrb(sys.A, sys.B), compute_uv=False)
    sigma1_values.append(np.log10(np.max(singular_values) / np.min(singular_values)))

    # calculate the poles (real parts)
    poles = np.linalg.eigvals(sys.A)
    real_parts.append([np.real(p) for p in poles])

# plot log(sigma) vs velocity
plt.figure(figsize=(12, 8))
plt.grid(True)
plt.plot(range(1, 41)[:len(sigma1_values)], sigma1_values, linewidth=2.5)
plt.title("$\log_{10}$ $\dfrac{\sigma_1}{\sigma_n}$ vs Longitudinal Velocity")
plt.xlabel("Longitudinal Velocity [m/s]")
plt.xlim(0, 40)
plt.ylabel("$\log_{10}$ $\dfrac{\sigma_1}{\sigma_n}$")
plt.ylim(0, 7 + 1)
plt.savefig("log(sigma) vs velocity.png")
plt.show()

# plot real parts vs velocity
plt.figure(figsize=(12, 8))
for i in range(4):
```

```
    plt.subplot(2, 2, i+1)
    plt.plot(range(1, 41), [p[i] for p in real_parts], linewidth=2.5)
    plt.grid(True)
    plt.title(f'Re(Pole {i+1}) vs Longitudinal Velocity')
    plt.xlabel('Longitudinal Velocity [m/s]')
    plt.xlim(0, 40)
    plt.ylabel(f'Re(Pole {i + 1})')

plt.tight_layout()
plt.savefig("real parts vs velocity.png")
plt.show()
```

```
/Users/ryanwu/Project2/bin/python /Users/ryanwu/Documents/CMU/24-677 Modern Control Theory/Project/Project2/Q1.py
At 2 m/s:
Rank of controllability matrix P: 4, Controllable: Yes
Rank of observability matrix Q: 4, Observable: Yes
==========================================================
At 5 m/s:
Rank of controllability matrix P: 4, Controllable: Yes
Rank of observability matrix Q: 4, Observable: Yes
==========================================================
At 8 m/s:
Rank of controllability matrix P: 4, Controllable: Yes
Rank of observability matrix Q: 4, Observable: Yes
==========================================================

Process finished with exit code 0

```

```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 closestNode, wrapToPi
from scipy.signal import place_poles

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

    def __init__(self, trajectory, look_ahead_distance=50):

        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

        # Add additional member variables according to your need here.
        self.look_ahead_distance = look_ahead_distance
        self.previous_psi = 0
        self.velocity_start = 30
        self.velocity_integral_error = 0
        self.velocity_previous_step_error = 0

    def update(self, timestep):

        trajectory = self.trajectory
        lr = self.lr
```

```python
        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)

        #  Set the look-ahead distance
        look_ahead_distance = 100
        _, closest_index = closestNode(X,Y, trajectory)

        if look_ahead_distance + closest_index >= 8203:
            look_ahead_distance = 0

        # Calculate the look-ahead distance
        closest_index = np.argmin(np.sqrt((trajectory[:, 0] - X) ** 2 + (trajectory[:, 1] - Y) ** 2))
        look_ahead_distance = min(self.look_ahead_distance, len(trajectory) - closest_index - 1)
        # look_ahead_X, look_ahead_Y = trajectory[closest_index + look_ahead_distance]

        # Calculate the desired heading angle
        X_desired = trajectory[closest_index + look_ahead_distance][0]
        Y_desired = trajectory[closest_index + look_ahead_distance][1]
        psi_desired = np.arctan2(Y_desired - Y, X_desired - X)

        # Design your controllers in the spaces below.
        # Remember, your controllers will need to
```

```
use the states
        # to calculate control inputs (F, delta).

        # ---------------|Lateral Controller |-------------------------

        # Please design your lateral controller below.
        # state space model for lateral control
        A = np.array([[0, 1, 0, 0], [0, -4 * Ca / (m * xdot), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m * xdot)], [0, 0, 0, 1], [0, (-2 * Ca * (lf - lr)) / (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]])

        # desired poles
        P = np.array([-4, -1, -3, -2])

        # calculate the gain matrix K using pole placement
        K = place_poles(A, B, P).gain_matrix

        # calculate lateral control error vector E
        e1 = 0
        e2 = wrapToPi(psi - psi_desired)
        e1dot = ydot + xdot * e2
        e2dot = psidot
        E = np.array([e1, e2, e1dot, e2dot])

        # control delta using the gain matrix K and error vector E
        delta = -np.dot(K, E)[0]
        delta = np.clip(delta, -np.pi/6, np.pi/6)

        # update the previous psi
        self.previous_psi = psi

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

```python

        # Please design your longitudinal controller below.

        # declaring PID variables
        Kp_velocity = 90
        Ki_velocity = 1
        Kd_velocity = 0.005

        # velocity error calculation
        velocity = np.sqrt(xdot ** 2 + ydot ** 2) * 3.6
        velocity_error = self.velocity_start - velocity
        self.velocity_integral_error += velocity_error * delT
        velocity_derivative_error = (velocity_error - self.velocity_previous_step_error) / delT

        # F with PID feedback control
        F = (velocity_error * Kp_velocity) + (self.velocity_integral_error * Ki_velocity) + (velocity_derivative_error * Kd_velocity)

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

```