Project: Part 3

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

Prof. D. Zhao

Due: Nov 16, 2021, 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 **P3-code** and your solutions in **.pdf** format to **P3-writeup**. 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 project, you will complete the following goals:

1. Design an lateral optimal controller
2. Implement A* path planning algorithm

[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 P3: Problems

Exercise 1. [50pts] All the code related to Exercise 1 is under P3_student/P3-LQR/. For the lateral control of the vehicle, design a discrete-time infinite horizon LQR controller.

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 and report the Your controller is **required** to achieve the following performance criteria to receive full points:

1. Time to complete the loop = 250 s
2. Maximum deviation from the reference trajectory = 7.0 m
3. Average deviation from the reference trajectory = 3.5 m

Some hints that may be useful:

- Make sure to discretize your continuous state-space system with the provided timestep (`delT`) prior to solving the ARE.
- Using LQR requires manually creating two matrices Q and R. Q works to penalize state variables, while R penalizes control input. Try to think about what form your Q and R matrices should take for good performance.
  - For Q, large values will heavily restrict changes to the respective states, while small values will allow the states to easily change.
  - Similarly, in R, large values will heavily restrict control input, while small values will allow the control input to vary widely.
  - One idea for tuning is to set the relevant indices of Q and R to

    $$\frac{1}{(\textbf{max value of the corresponding state/input})^2}$$

    in order to normalize the value. Make sure to experiment outside of this guideline to determine the best performance.

– There is a relationship between Q and R, though it is subtle. For example, if you increase weights in Q, you are more heavily penalizing changes in the states, which will require more control input. This would imply that you should decrease weights in R to see an effect. Due to this, it may be helpful to keep either Q or R fixed while varying the other during tuning.

[10% Bonus]: Complete the loop within 130 s. The maximum deviation and the average deviation should be within in the allowable performance criteria mentioned above.

Solution:

```
# Fill in the respective functions to implement the LQR optimal 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))

# Discretize the system and extract Ad and Bd
sysc = signal.StateSpace(A, B, C, D)
sysd = sysc.to_discrete(delT)
Ad = sysd.A
Bd = sysd.B

# Come up with reasonable values for Q and R (state and control
↪  weights)
Q = np.array([[1,0,0,0],[0,0.1,0,0],[0,0,0.1,0],[0,0,0,0.01]]) #
↪  Weight important states more (e1)
R = 50 # Don't want to oversteer on tight turns, plus smaller R is
↪  less stable

# 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 = 150

# 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
try:
    psiDesired =
    ↪  np.arctan2(trajectory[node+forwardIndex,1]-trajectory[node,1],
    ↪  \

                                ↪  trajectory[node+forwardIndex,0]-trajectory[node,
    e1 =  (Y - trajectory[node+forwardIndex,1])*np.cos(psiDesired) -
    ↪  \
        (X - trajectory[node+forwardIndex,0])*np.sin(psiDesired)
```

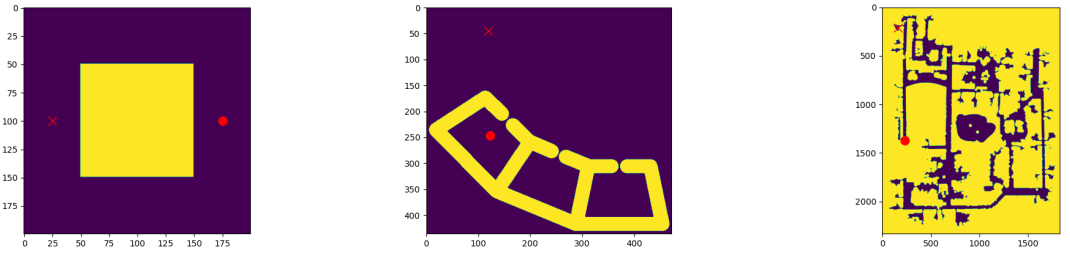
```
except:
    # The index -1 represents the last element in that array (the
    ↪  end of the course)
    psiDesired = np.arctan2(trajectory[-1,1]-trajectory[node,1], \
                            trajectory[-1,0]-trajectory[node,0])
    e1 =  (Y - trajectory[-1,1])*np.cos(psiDesired) - \
        (X - trajectory[-1,0])*np.sin(psiDesired)

e1dot = ydot + xdot*wrapToPi(psi - psiDesired)
e2 = wrapToPi(psi - psiDesired)
e2dot = psidot

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

# Solve discrete ARE and compute gain matrix
S = np.matrix(linalg.solve_discrete_are(Ad, Bd, Q, R))
K = np.matrix(linalg.inv(Bd.T @ S @ Bd + R) @ (Bd.T @ S @ Ad))
#[l,v] = linalg.eig(Ad - Bd @ K)

# Calculate delta via u = -Kx
delta = wrapToPi((-K @ states)[0, 0])

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

# Reference value for PID to tune to
desiredVelocity = 12

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
```

Exercise 2. [50pts] **A* Planning**

All the code related to Exercise 1 is under P3_student/P3-AStar/. In this exercise, we will implement A* path planning algorithm. Now there is another vehicle driving on the track. If the ego vehicle follows the original trajectory (which aligns with the track), it is likely to crash into the other vehicle. In order to perform the overtaking, we need to re-plan the trajectory. In this problem, we will implement A* path planning algorithm.

To simplify the problem, we initialize the other vehicle on the straight section of the track. You will overtake it on the straight section as well. We assume the other vehicle will drive in a straight line so we do not need to worry about trajectory prediction too much. As you get close to the other car, our controller will use A* algorithm to re-plan the path. The predicted trajectory of another vehicle is treated as obstacle in the cost map. In addition, we do not want to drive too far away from the track therefore we treat the area outside a certain distance from the center of the track as obstacle. In your implementation, you only need to find a path given a static and discretized map.

You only need to complete the *plan* function in P3_student/P3-AStar/AStar-scripts/Astar_script.py. To test the algorithm you implemented, simply run Astar_script.py file, which will generate 3 figures of path planning results using the function you implemented. The maps are shown in Figure.1. Yellow region represents the obstacle. Red Crosses represent the start and red dots represent the goal. If you think it plans reasonable paths, you can copy the code from your Astar_script.py to P3_student/P3-AStar/controllers/main/Astar.py and then open Webots world file P3_student/P3-AStar/worlds/Project3.wbt. and try it with the Webots simulator and the controller you implemented in Exercise 1.



(a) (b) (c)

Figure 1: A* Testing cases

In your submission, 1. Save and attach the figures generated by runing Astar.py in your submission and 2. report the time of completion. As long as the generated figures show the successful path planning and the car completes the racing without crashing, you will receive full marks for this problem.

Here is the pseudocode for A* path planning algorithm (Source: https://en.wikipedia.org/wiki/A*_search_algorithm).

```
function calculate_path(node):
    path_ind.append(node.pose)
    while current.parent is not empty:
        current := current.parent
        path_ind.append(current.pose)
    return path_ind
```

```
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from
↪  node n.
function A_Star(start, goal, h)
    // Initialize start node and goal node class

    // Calculate h_value and f_value of start node
    // For node n, g(node) is the cost of the cheapest path from start to n
    ↪  currently known
    // f(node) = g(node) + h(node), which represents ourcurrent best guess
    ↪  as to
    // how short a path from start to finish can be if it goes through n


    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather
    ↪  than a hash-set.
    openSet := {start}

    // For node n, closedSet[n] is the node immediately preceding it on the
    ↪  cheapest path from start
    // to n currently known.
    closedSet := an empty map


    while openSet is not empty
        // Current is the node in open_list that has the lowest f value
        // This operation can occur in O(1) time if openSet is a min-heap or
        ↪  a priority queue
        current := the node in openSet having the lowest fScore[] value
        openSet.Remove(current)

        // Append current node into the closed_list

        if current = goal
            return reconstruct_path(closedSet, current)

        for each neighbor of current (can be computed by calling the
        ↪  get_successor method):
            // d(current,neighbor) is the weight of the edge from current to
            ↪  neighbor
            // g(successor) = g(current) + d(current, successor)
            // h(successor) can be computed by calling the heuristic method
            // f(successor) = g(successor) + h(successor)
            // Don't forget to push the successor into open_list

    // Open set is empty but goal was never reached
    return failure
```

Solution:

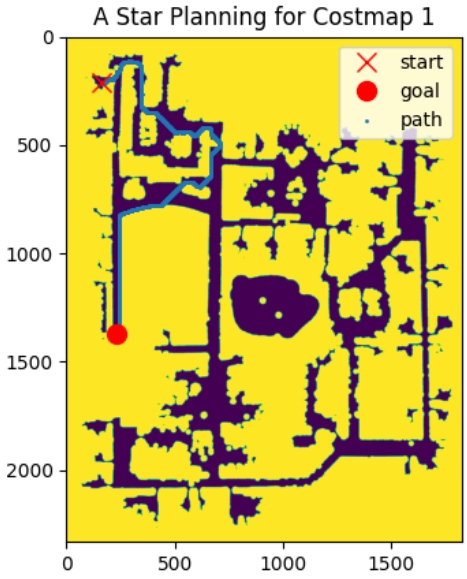


Figure 2: Astar path for map 1

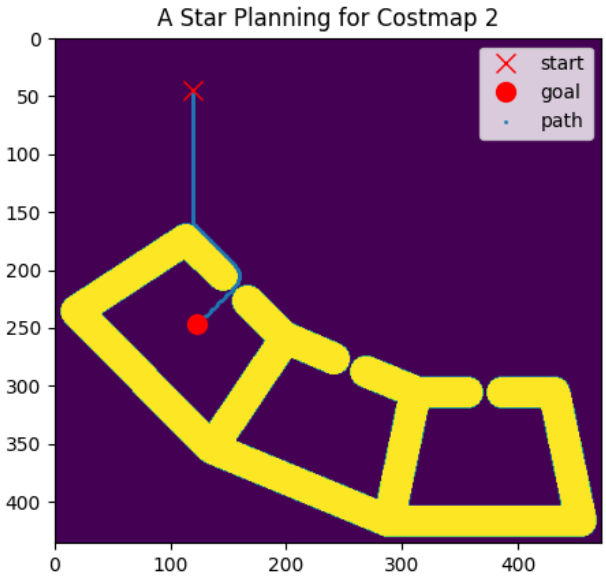


Figure 3: Astar path for map 2

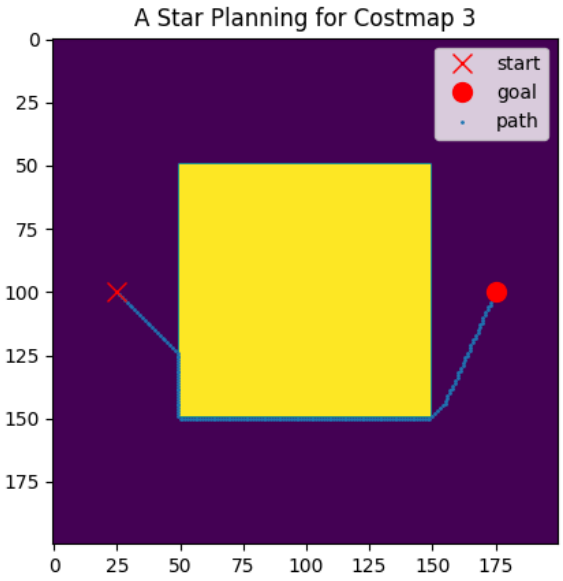


Figure 4: Astar path for map 3

```
from os import close
import numpy as np
from heapq import heappop, heappush
import matplotlib.pyplot as plt

class Node(object):
    def __init__(self, pose):
        self.pose = np.array(pose)
        self.x = pose[0]
        self.y = pose[1]
        self.g_value = 0
        self.h_value = 0
        self.f_value = 0
        self.parent = None

    def __lt__(self, other):
        return self.f_value < other.f_value

    def __eq__(self, other):
        return (self.pose == other.pose).all()

class AStar(object):
    def __init__(self, map_path):
        self.map_path = map_path
        self.map = self.load_map(self.map_path).astype(int)
```

```
        print(self.map)
        self.resolution = 0.05
        self.y_dim = self.map.shape[0]
        self.x_dim =self.map.shape[1]
        print(f'map size ({self.x_dim}, {self.y_dim})')

    def load_map(self, path):
        return np.load(path)

    def reset_map(self):
        self.map = self.load_map(self.map_path)

    def heuristic(self, current, goal):
        """
        Euclidean distance
        """
        return np.hypot(current.x - goal.x, current.y - goal.y)

    def get_successor(self, node):
        successor_list = []
        x,y = node.pose
        pose_list = [[x+1, y+1], [x, y+1], [x-1, y+1], [x-1, y],
                     [x-1, y-1], [x, y-1], [x+1, y-1], [x+1, y]]

        for pose_ in pose_list:
            x_, y_ = pose_
            if 0 <= x_ < self.y_dim and 0 <= y_ < self.x_dim and
            ↪  self.map[x_, y_] == 0:
                self.map[x_, y_] = -1
                successor_list.append(Node(pose_))

        return successor_list

    def calculate_path(self, node):
        path_ind = []
        path_ind.append(node.pose.tolist())
        current = node
        while current.parent:
            current = current.parent
            path_ind.append(current.pose.tolist())
        path_ind.reverse()
        print(f'path length {len(path_ind)}')
        path = list(path_ind)

        return path
```

```
def plan(self, start_ind, goal_ind):
    """
    @param start_ind : [x, y] represents coordinates in webots world
    @param goal_ind : [x, y] represents coordinates in webots world
    @return path : a list with shape (n, 2) containing n path point
    """

    # initialize start node and goal node
    start_node = Node(start_ind)
    goal_node = Node(goal_ind)
    start_node.h_value = self.heuristic(start_node, goal_node)
    start_node.f_value = start_node.g_value + start_node.h_value

    self.reset_map()

    open_list = []
    closed_list = np.array([])
    heappush(open_list, start_node)

    while len(open_list):
        current = heappop(open_list)
        closed_list = np.append(closed_list, current)
        self.map[current.x, current.y] = -1

        if current == goal_node:
            print('reach goal')
            return self.calculate_path(current)

        for successor in self.get_successor(current):
            successor.parent = current
            successor.g_value = current.g_value + np.hypot(successor.x -
            ↪  current.x, successor.y - current.y)
            successor.h_value = self.heuristic(successor, goal_node)
            successor.f_value = successor.g_value + successor.h_value

            heappush(open_list, successor)

    print('path not found')
    return None

def run(self, cost_map, start_ind, goal_ind):
    if cost_map[start_ind[0], start_ind[1]] == 0 and
    ↪  cost_map[goal_ind[0], goal_ind[1]] == 0:
        return self.plan(start_ind, goal_ind)
```

```
else:
    print('already occupied')
```

4 Appendix

(Already covered in P1)

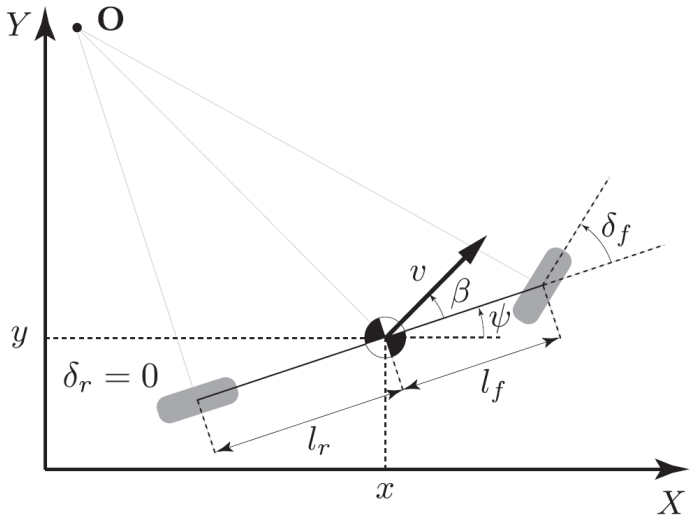


Figure 5: Bicycle model[2]

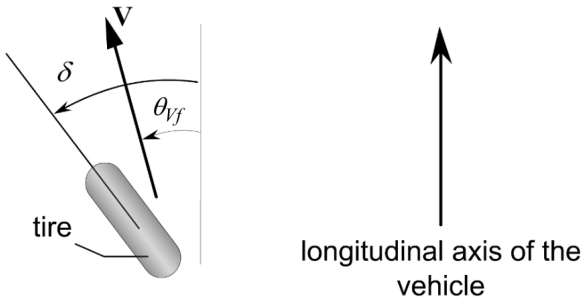


Figure 6: Tire slip-angle[2]

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 5, 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

$$\alpha_f = \delta - \theta_{Vf}$$
$$\alpha_r = -\theta_{Vr}$$

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

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

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:

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

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:

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

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	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 intertia	kg mˆ2	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

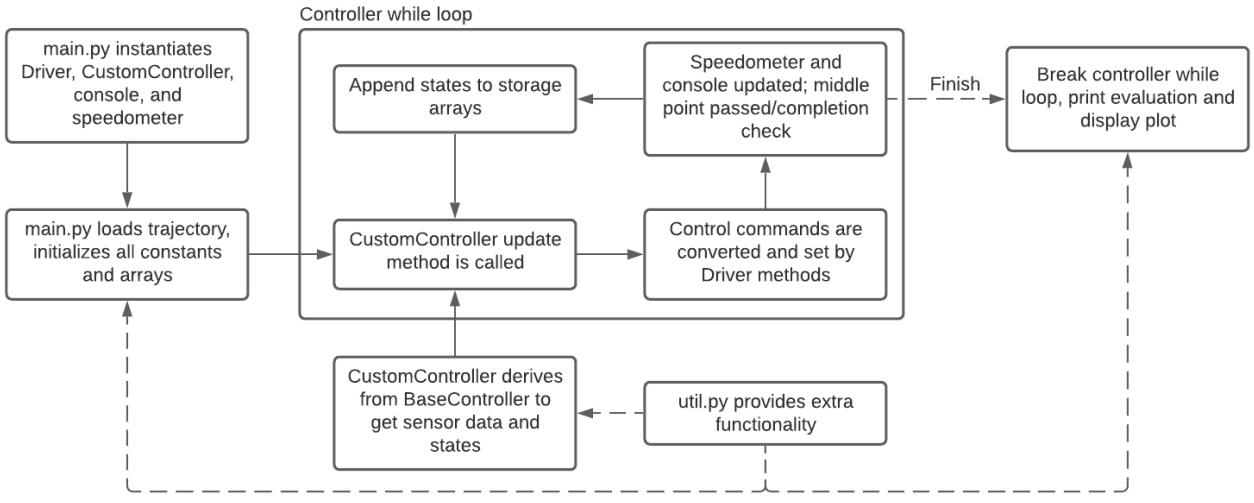


Figure 7: Simulation code flow

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 7.

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 8.

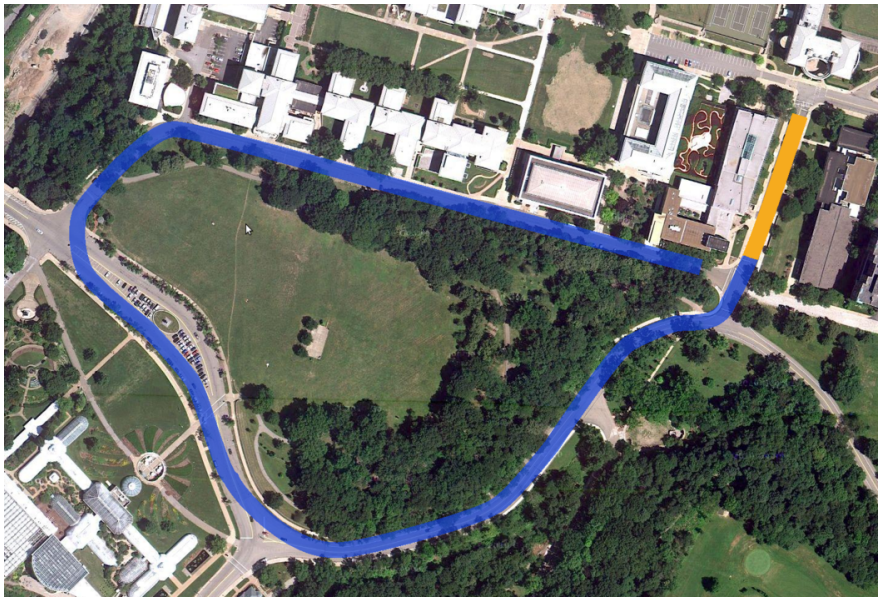


Figure 8: Buggy track[3]

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