ROB 311
BallBot Final Report

By
Max Rucker, Ben Chen
Team 8

December 11th, 2023

Summary

This report details the process we went through to create our Ballbot for ROB 311, along with quantitative and qualitative analysis of the performance of our Ballbot during the final competition. For our Ballbot, we used Solidworks to 3D model our components to build our Ballbot. After manufacturing and assembling our Ballbot, we implemented control algorithms to control our Ballbot and do a series of tasks. A depth analysis of the performance will be introduced in the following.

Design

To create our robot assembly design, we used Solidworks to 3D model our acrylic plates, motor mounts, pico board holder, and leash attachment. We modeled three acrylic plates that when stacked together can hold each attachment we need for our robot. Each of these plates was laser-cut in a CO2 laser cutter. On the bottom plate, our modeled motor mounts would be attached around the edge each 120 degrees apart from each other. Along with that, our pico board holder was attached to the bottom plate with some rubber dampeners to lessen the vibrations felt by our inertial measurement unit. For our motor mounts, we created a design that holds a Pololu 37D motor. We determined that these motors are the most efficient for our torque and velocity requirements we found from defining torque and speed requirements based on the Ballbot's kinematics. We also connected to each motor two Omni wheels connected with a few spaces. These Omni wheels allow our Ballbot to move in any direction and remove the friction created by regular wheels moving perpendicular to the set angle.
Our second plate held our Raspberry Pi and our battery. On the top of the plate, we attached our battery with velcro so it could easily be attached and removed when we needed to charge it. Lastly, on the top plate, we attached our 3D-printed leash mount. This piece allowed us to attach a leash to our Ballbot during testing to ensure when it fell it could be caught without taking any damage.

Data Processing

The main components involved in our data processing were our Raspberry Pi and our Pico board. These two pieces of hardware were responsible for collecting and processing the data that comes from our motors and internal measurement unit. Our motors were fitted with encoders that tracked the rotations of each wheel. This allows us to extract important information such as wheel velocity and eventually ball velocity. Our IMU, the MPU-6050, gives us information about the acceleration of our axes which also allows us to extract information about the lean angle, angular velocity, and such for our Ballbot. With these two measurement devices, we can gather this information through the Pico board.

Both the motors and IMU are connected to the Pico board. The Pico board handles all I^2C communication which is used to collect data from the IMU and the motors. The Pico also handled sending any data back to the motors in terms of how much power to drive the motor with. By using I^2C communication, it allows us to efficiently have multiple communication channels to interface with. This information was then sent to the Raspberry Pi through a serial peripheral interface.

The Raspberry Pi handles all the computation and code to calculate how we should be controlling the motors based on the IMU and motor encoder values from the pico. The Raspberry Pi takes in all this data and transforms it into readable data that we can use to calculate the torques required to control our Ballbot and allows us to change how we want to control it through the code we write. The Raspberry Pi acts as the brains of the system and then sends back data to the Pico to control the motors to keep our Ballbot balanced.

Control Scheme

Balance PID Controller

Our balance controller was the main driver that kept our Ballbot balanced. Our balance controller was a simple PID controller that utilized the measured lean angles from the IMU as input and calculated torques for each motor to keep our Ballbot balanced at a desired lean angle of zero degrees in the X and Y directions. For our balance controller, we saturated the output torques of the PID controller to 0.6 to make sure that the steering controller could apply torque but keep the balance controller dominant in the control loop.

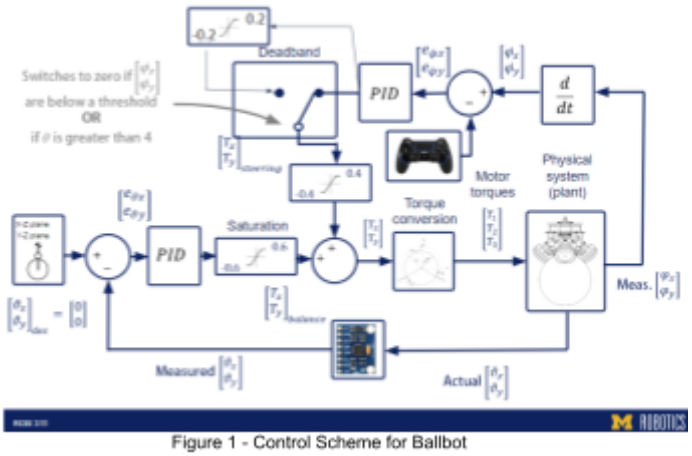

Figure 1 - Control Scheme for Ballbot

Here are the PID values for our Balance Controller: $K_p = 8.5$, $K_I = 0.01$, $K_D = 0$
We also saturated the KI value at 3 to limit the integral windup we often faced when trying to tune the controller. We found that these PID values were the most optimal for keeping our balance controller strong enough yet smooth.

Steering PID Controller

Our steering controller was the second addition to our control loop. This controller is also a PID controller that utilizes the angular velocity of the ball to control the velocity at which the ball moves. The desired velocity was determined by how much the joystick on the controller was moved. We had our Joystick have a range of -1.2 to 1.2 radians per second to set the desired velocity. Our actual angular velocity came from the encoders on the motors which were then transformed to wheel speed, which then were used to calculate the ball velocity.

Here are the PID values for our Steering Controller: $K_p = 0.15$, $K_I = 0.085$, $K_D = 0$
With our steering controller, we made sure to saturate the output at 0.4 to ensure that the steering controller never overpowered the balance controller. Along with this, we implemented a deadband for our steering controller. This deadband was controlled by the lean angle and the ball velocity. If the lean angle was greater than 0.4 or if the ball angular velocity was less than 0.5 radians per second, then we saturate the steering controller at 0.2. This was done to make

sure that the steering controller is lessened when we go past a lean angle threshold and make sure the balance controller is in full effect to keep the Ballbot balanced. Along with this, when the Ballbot is not moving we don't want the steering controller to apply too much torque when the Ballbot is not moving too much. We decided to allow the steering controller to apply some torque when under the 0.5 radians per second threshold to try and eliminate drift in the Ballbot. Overall this steering controller mostly helped us eliminate drift in the Ballbot and allowed us to control the movement with the use of a controller.

Balance Performance

Normal Balance

After implementing the balance controllers, we tested on how long our Ballbot can balance on the basketball. r The balance controller was tested first. Without the help from the steering controller, the Ballbot can stay on the ball for 8 mins then fall off. During the process, the Ballbot drifted a lot . This is inherently a problem with the balance controller, as even though the controller tries to keep the Ballbot balanced, if a disturbance occurs and creates velocity, the balance controller is not able to create torque to counteract this velocity. Some of this disturbance may be caused by the IMU data not being perfect with some velocity being created from our balance controller trying to match the lean angle. A possible solution would be to tune our lowpass filter a bit better to cut out more frequencies, but this comes at a cost of processing speed and such.

After testing the balancing controller solely, we combined it with the steering controller. The overall performance, in terms of balancing time, was the same, which balanced for around 9 minutes. However, the drifting was a lot better. When the Ballbot moves in one direction with a certain velocity, the steering controller will apply more torque in that direction and make the Ballbot upright again. Therefore, the drifting we had for the balance controller can be adjusted by the steering controller. During the testing, a little drifting still existed, which might be due to the reasons that the saturation values we set for the balance controller (0.6 torque) and steering controller (0.4 torque) might not have been tuned enough. The steering controller might either need to put more or less effort, so tuning the saturation value for the steering controller might be helpful. Also, the drift might be caused by the drifting velocity never reaching the deadband we set. Therefore the torques applied by the steering controllers under the deadband might not be enough to fix the drifting.

Hexapod Balance

Besides the non-disturbance test, we also tested on a Hexapod to get a deeper analysis of the balance performance. With some disturbances applied to the system, the system becomes less stable and drifts more; however, the Ballbot can still sit on the ball for 2 minutes on the hexapod. All these behaviors are understandable. Since we gave the system disturbances, which make the Ballbot harder to balance. For this test, we utilized the controller to steer the Ballbot on the hexapod pad to ensure it would not drift off of the small balance area.

Step Response

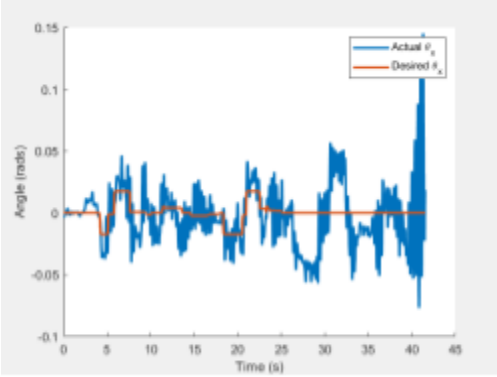

Figure 3 - Step Responses for Lean Angle x direction

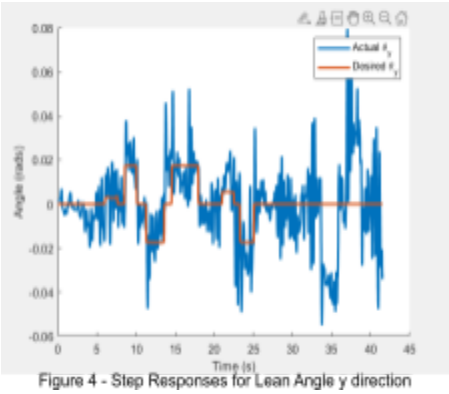

Figure 4 - Step Responses for Lean Angle y direction

From the step responses, we can see that our balance controller follows the desired angle quite well when given a step response. The main issue we see in this graph is the many spikes in the actual lean angle. This comes from noise in the IMU readings. One thing that we could improve is the lowpass filter we use to filter the lean angle to get a smoother graph which may help our balance controller. The only downside of refining the filter may cause a higher delay in the signal. Overall, our balance controller is sufficient to control balancing given disturbances. This is also seen in our hexapod testing from before.

Steering Performance

Step Response:

From our step responses, we can see that our steering control does fairly well when given inputs. One thing that is noticed is that the controller takes a bit to reach the desired value. This may be due to the K_p value being too small. The reason we decided not to increase this K_p value was because we would notice oscillations start to occur when we were at a steady state. We decided to rely on the K_i value since it would help eliminate drifting when trying to balance. This is because a slight drift would cause the integral term to build up and help eliminate the drift in the system.

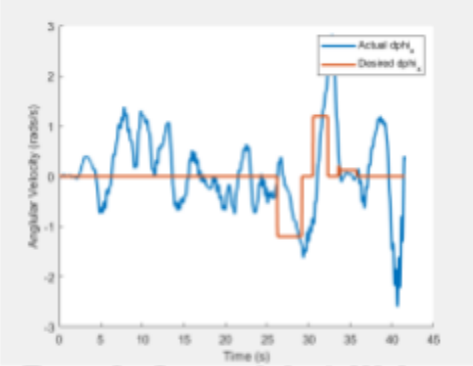

Figure 5 - Step Responses for Angular Velocity x direction

4’ x 4’ Square Challenge:

To plot the 2D top-down view of the Ballbot navigating through a 4’ * 4’ square. We recorded the phi_x and phi_y while the Ballbot was driving square. Then we multiply the ball radius to get the ball displacement of the Ballbot in the x and y directions.

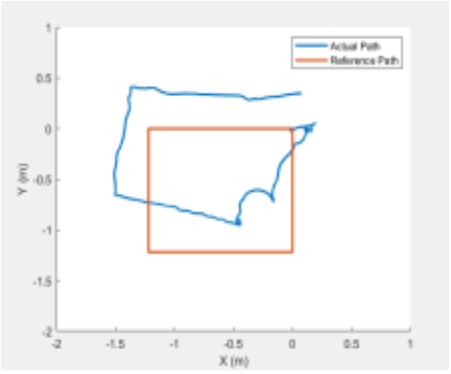

Figure 7 - 4’ x 4’ Driving Square Actual Path vs Reference Path

As shown in Figure 7, the red line is the Reference Path, and the blue line is the Actual Path we get from the

encoder reading. The blue line is approximately a square, but it drifts a lot in the angles. The whole graph rotates in certain angles. This might be due to the reason that when we drive the Ballbot to go around the square, the chassis spins a little (caused by the balance and steering controller). The encoders read data from the local frame of the Ballbot. When the chassis spins a little, the phi_x and phi_y data will be recorded in the robot's frame, not in the world frame. This might be the reason why our Ballbot is not going vertically at the beginning of the drive square, even though it is on the line in reality. Along with this, we used the PS4 controller to make the Ballbot drive, which is pretty unstable and hard to control. Therefore some round curves are drawn in the figure.

Maximum Angular Velocity

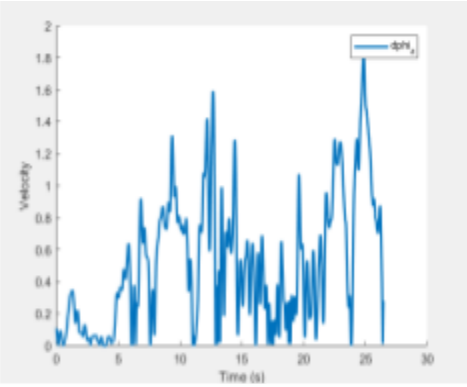

Figure 8 - Maximum Angular Velocity on z-axis

Getting the maximum angular velocity that the Ballbot can have under balancing is straightforward. We extracted the dphi_z data from the encoders and plot this data. The reason why we choose to use dphi_z is that when the Ballbot is spinning, the ball itself rarely spins. Therefore the dphi_z describes the angular velocity of the chassis in the same magnitude but in the opposite direction. To make the graph clearer, we take the absolute value of the angular velocity, since the direction the Ballbot spins doesn't affect the magnitude. In Figure 8, some spikes like the one at around 13 seconds and the one at 25 seconds were ignored. These are some outliers that were recorded when the Ballbot became too unbalanced. Generally, the maximum angular velocity of the Ballbot is around 1.4 rad/sec. The reason this is our maximum is that the motors can't apply enough torque to balance the Ballbot when they are already spinning to apply torque along the z-axis. Essentially, the balance and steering controller get overwhelmed by the torque used to spin the ball. This makes it extremely difficult for the Ballbot to balance.

APPENDIX:

Solidworks Render

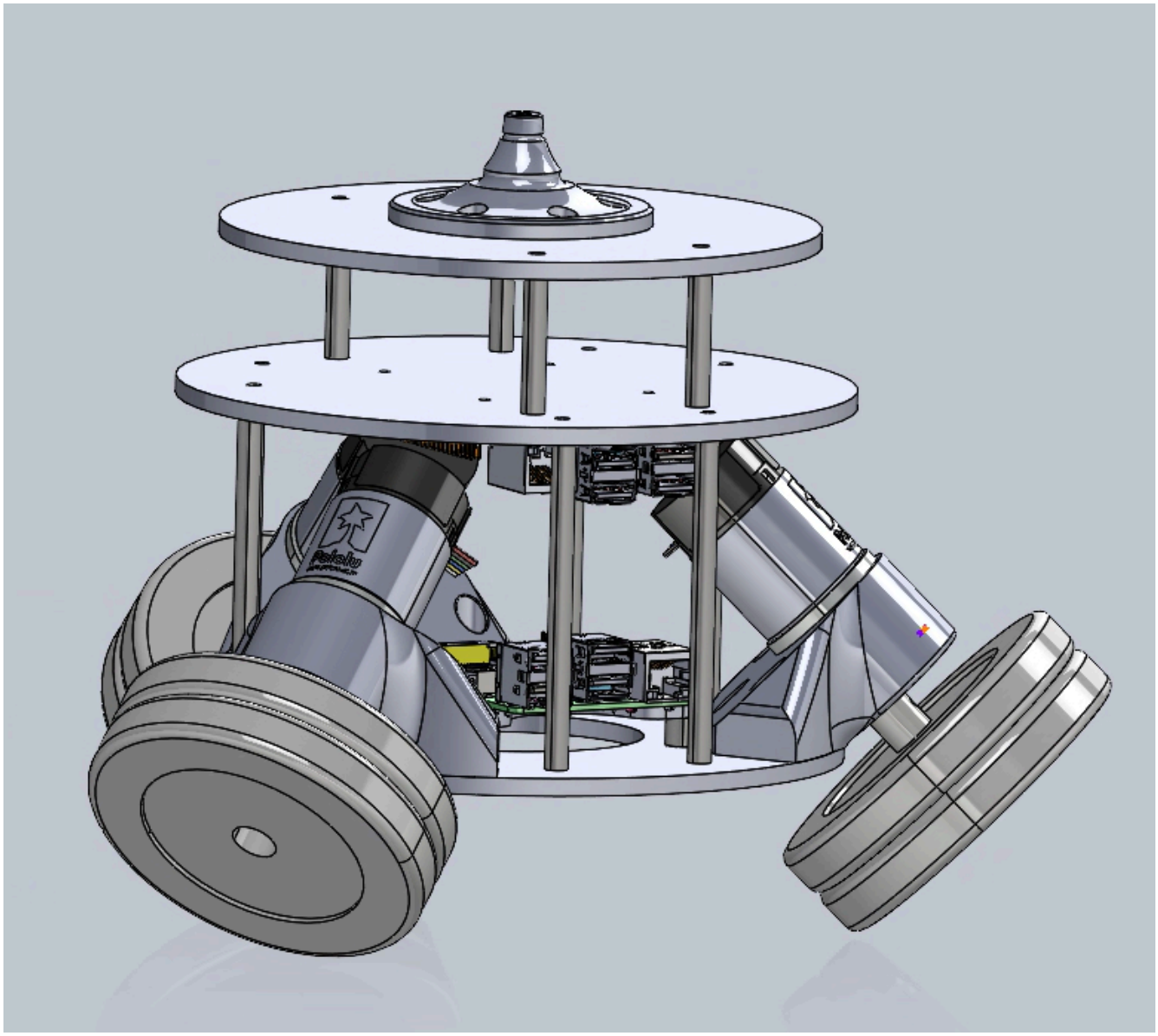

Figure 9 - Solidworks Render of Ballbot

Code

```
"""
ROB 311 - Ball-bot steering boilerplate code

Authors: Senthur Ayyappan, Japmanjeet Singh Gill, and Elliott Rouse
Neurobionics Lab / Locomotor Control Lab
"""

import sys
from threading import Thread
import time
import numpy as np

from MBot.Messages.message_defs import mo_states_dtype, mo_cmds_dtype,
mo_pid_params_dtype
from MBot.SerialProtocol.protocol import SerialProtocol

from collections import deque
from DataLogger import dataLogger

from loop import SoftRealtimeLoop
from ps4 import BBController

from constants import *
from simple_pid import PID

from transformations import transform_w2b, compute_motor_torques

import FIR as fir

# Initializing lowpass filters for the ball-velocity estimates.
# Please note that these filters can be modified to filter the IMU data,
# ball velocity, or ball position.
lowpass_filter_dphi_x = fir.FIR()
lowpass_filter_dphi_x.lowpass(N, Fn)

lowpass_filter_dphi_y = fir.FIR()
lowpass_filter_dphi_y.lowpass(N, Fn)

def register_topics(ser_dev:SerialProtocol):
    # Mo :: Commands, States
    ser_dev.serializer_dict[101] = [lambda bytes: np.frombuffer(bytes, dtype=mo_cmds_dtype),
lambda data: data.tobytes()]
```

```
    ser_dev.serializer_dict[121] = [lambda bytes: np.frombuffer(bytes, dtype=mo_states_dtype),
lambda data: data.tobytes()]

def wma_filter(wma_window):
    return np.sum(WMA_NORM * wma_window)

if __name__ == "__main__":

    trial_num = int(input('Trial Number? '))
    filename = 'ROB311_Test_%i' % trial_num
    dl = dataLogger(filename + '.txt')

    t_start = 0.0

    ser_dev = SerialProtocol()
    register_topics(ser_dev)

    # Initializing a thread for reading data from the Pico
    serial_read_thread = Thread(target = SerialProtocol.read_loop, args=(ser_dev,),
daemon=True)
    serial_read_thread.start()

    # Local data structures for storing the data obtained from the Pico
    commands = np.zeros(1, dtype=mo_cmds_dtype)[0]
    states = np.zeros(1, dtype=mo_states_dtype)[0]

    # Local variables for the controller
    psi = np.zeros((3, 1))
    psi_offset = np.zeros((3, 1))

    dpsi = np.zeros((3, 1))

    phi = np.zeros((3, 1))
    dphi = np.zeros((3, 1))
    prev_phi = phi
    theta_x = 0.0
    theta_y = 0.0

    dphi_x = 0.0
    dphi_y = 0.0
    dphi_z = 0.0

    # deque is a data structure that automatically pops the previous data based on its max length.
    theta_x_window = deque(maxlen=WMA_WINDOW_SIZE) # A sliding window of values
```

```
theta_y_window = deque(maxlen=WMA_WINDOW_SIZE) # A sliding window of values

for _ in range(WMA_WINDOW_SIZE):
    theta_x_window.append(0.0)
    theta_y_window.append(0.0)

# Net Tx, Ty, and Tz initialization
Tx = 0.0
Ty = 0.0
Tz = 0.0

# Steering controller torques: Tx_steer, Ty_steer, and Tz_steer initialization
Tx_steer = 0.0
Ty_steer = 0.0
Tz_steer = 0.0

# Stability controller torques: Tx_bal, Ty_bal, and Tz_bal initialization
Tx_bal = 0.0
Ty_bal = 0.0
Tz_bal = 0.0

# T1, T2, and T3
T1 = 0.0
T2 = 0.0
T3 = 0.0

error_x_sum_bal = 0
error_x_bal = 0
error_y_sum_bal = 0
error_y_bal = 0

commands['start'] = 1.0
zeroed = False

# Time for communication between the RPi and Pico to be established
time.sleep(1.0)
ser_dev.send_topic_data(101, commands)

# Set points for the stability controller (along x|roll and y|pitch).
theta_roll_sp = 0.0
theta_pitch_sp = 0.0

# Set points for the steering controller (along x|roll and y|pitch).
dphi_roll_sp = 0.0
```

```
    dphi_pitch_sp = 0.0

    dphi_roll_pid_components = np.array([0.0, 0.0, 0.0])
    dphi_pitch_pid_components = np.array([0.0, 0.0, 0.0])
    dphi_roll_pid = PID(DPHI_KP, DPHI_KI, DPHI_KD, dphi_roll_sp)
    dphi_pitch_pid = PID(DPHI_KP, DPHI_KI, DPHI_KD, dphi_pitch_sp)
    dphi_roll_pid.output_limits = (-MAX_VEL_DUTY, MAX_VEL_DUTY)
    dphi_pitch_pid.output_limits = (-MAX_VEL_DUTY, MAX_VEL_DUTY)

    print('Starting the controller!')
    i = 0

    # This thread runs in parallel to the main controller loop and listens for any PS4 input
    # from the user. The PS4 controller is used to update setpoints the steering controller and to
tune
    # the controller gains on the fly. Please refer to the ps4.py file for more details.
    bb_controller = BBController(interface="/dev/input/js0", connecting_using_ds4drv=False)
    bb_controller_thread = Thread(target=bb_controller.listen, args=(10,))
    bb_controller_thread.start()

    # Entering our main control loop, which is set to run at 200 Hz.
    # Feel free to experiment with the frequency of the controller loop by
    # changing the value of "FREQ" variable in constants.py.
    for t in SoftRealtimeLoop(dt=DT, report=True):

        # Reading data from the Pico, if it isn't available, then skip this iteration.
        # This is true when the Pico is connected and is collecting the IMU offsets.
        try:
            states = ser_dev.get_cur_topic_data(121)[0]

        except KeyError as e:
            # Calibrates for 10 seconds
            print("<< CALIBRATING :: {:.2f} >>".format(t))
            continue

        # Extracting the Motor Encoder values from the Pico's data
        psi[0] = states['psi_1']
        psi[1] = states['psi_2']
        psi[2] = states['psi_3']

        dpsi[0] = states['dpsi_1']
        dpsi[1] = states['dpsi_2']
        dpsi[2] = states['dpsi_3']
```

```
# A sliding window of "WMA_WINDOW_SIZE" values for the WMA filter
# on the IMU values. Please edit the constants.py file to change the
# WMA_WINDOW_SIZE and the WMA_WEIGHTS.
theta_x_window.append(states['theta_roll'])
theta_y_window.append(states['theta_pitch'])

# Applying a WMA filter on the IMU values
theta_x = wma_filter(theta_x_window)
theta_y = wma_filter(theta_y_window)

# A ten second wait to place the bot on top of the ball--this is to
# reset the encoder values so that at i=0, the ball-bot's position is (0, 0)
if t > 11.0 and t < 21.0:
    print("<< PLACE THE BOT ON TOP OF THE BALL :: {:.2f} >>".format(t))
elif t > 21.0:
    if not zeroed:
        psi_offset = psi
        zeroed = True

psi = psi - psi_offset

# Transforming wheel attributes (position and velocity) to ball attributes.
phi[0], phi[1], phi[2] = transform_w2b(psi[0], psi[1], psi[2])
dphi[0], dphi[1], dphi[2] = transform_w2b(dpsi[0], dpsi[1], dpsi[2])

# Lowpass filtering the ball-velocity estimates
dphi_x = lowpass_filter_dphi_x.filter(dphi[0][0])
dphi_y = lowpass_filter_dphi_y.filter(dphi[1][0])
dphi_z = lowpass_filter_dphi_y.filter(dphi[1][0])

# Few conditional statements to start the time counter only
# after the ball-bot is placed on top of the ball.
if zeroed:
    if i == 0:
        t_start = time.time()

    i = i + 1
    t_now = time.time() - t_start

# PID - Balance Controller (along x|roll and y|pitch).
# Your PID implementation goes here
# Controller error terms

theta_roll_sp = bb_controller.theta_x_sp
```

```
        theta_pitch_sp = bb_controller.theta_y_sp

        error_x_bal = theta_roll_sp - theta_x
        error_y_bal = theta_pitch_sp - theta_y
        error_x_sum_bal = error_x_bal + error_x_sum_bal
        error_y_sum_bal = error_y_bal + error_y_sum_bal

        if (error_x_sum_bal > 3):
            error_x_sum_bal = 3
        if (error_x_sum_bal < -3):
            error_x_sum_bal = -3
        if (error_y_sum_bal > 3):
            error_y_sum_bal = 3
        if (error_y_sum_bal < -3):
            error_y_sum_bal = -3

        # if(np.abs(theta_x) < np.deg2rad(2)):
        #     error_x_sum_bal = 0
        # if(np.abs(theta_y) < np.deg2rad(2)):
        #     error_y_sum_bal = 0



        Tx_bal = (PITCH_THETA_KP * error_x_bal) + (PITCH_THETA_KI * error_x_sum_bal)
        Ty_bal = (ROLL_THETA_KP * error_y_bal) + (ROLL_THETA_KI * error_y_sum_bal)

        Tz_bal = 0


        ################
       ## Start the steering controller if there is a change in the
       ## ball-velocity (dphi) setpoint using the PS4 controller.
        if np.abs(bb_controller.dphi_y_sp) > DPHI_DEADBAND or np.abs(bb_controller.dphi_x_sp)
> DPHI_DEADBAND:
            dphi_pitch_pid.setpoint = bb_controller.dphi_y_sp
            dphi_roll_pid.setpoint = bb_controller.dphi_x_sp

            Tx_steer = dphi_roll_pid(dphi_x)
            Ty_steer = dphi_pitch_pid(dphi_y)

        # Also start the steering controller if the ball-velocity is greater than
        # DPHI_DEADBAND (0.5 rad/sec) to prevent the ball-bot from drifting around
        elif np.abs(dphi_x) > DPHI_DEADBAND or np.abs(dphi_y) > DPHI_DEADBAND:
            dphi_roll_pid.setpoint = 0.0
```

```
    dphi_pitch_pid.setpoint = 0.0

    Tx_steer = dphi_roll_pid(dphi_x)
    Ty_steer = dphi_pitch_pid(dphi_y)

else:
    dphi_roll_pid.reset()
    dphi_pitch_pid.reset()
    Tx_steer = 0.0
    Ty_steer = 0.0

# Max Lean angle (Theta) constraint: If theta is greater than the maximum lean angle
# (4 degrees), then turn off the steering controller.



# Summation of planar torques
# Stability controller + Steering controller

# if np.abs(Tx) > MAX_PLANAR_DUTY:
#     Tx_bal = np.sign(Tx) * MAX_PLANAR_DUTY

# if np.abs(Ty) > MAX_PLANAR_DUTY:
#     Ty_bal = np.sign(Ty) * MAX_PLANAR_DUTY

# if np.abs(Tx) > MAX_PLANAR_DUTY:
#     Tx_steer = np.sign(Tx) * MAX_PLANAR_DUTY

# if np.abs(Ty) > MAX_PLANAR_DUTY:
#     Ty_steer = np.sign(Ty) * MAX_PLANAR_DUTY

# if(Tx_steer != 0):
#     if(Tx_steer > 0.3):
#         Tx_steer = 0.3
#     elif(Tx_steer < -0.3):
#         Tx_steer = -0.3
#     if(Tx_bal + Tx_steer > 0.75):
#         Tx_bal = 0.75 - Tx_steer
#     elif(Tx_bal + Tx_steer < -0.75):
#         Tx_bal = -0.75 - Tx_steer
if np.abs(theta_x) > MAX_THETA or np.abs(theta_y) > MAX_THETA:
    dphi_roll_pid.setpoint = 0.0
    dphi_pitch_pid.setpoint = 0.0
```

```
            Tx_steer = dphi_roll_pid(dphi_x)
            Ty_steer = dphi_pitch_pid(dphi_y)
            # Tx_steer = Tx_steer - (np.sign(Tx_steer)*0.03)
            # Ty_steer = Ty_steer - (np.sign(Tx_steer)*0.03)
            if np.abs(Tx_bal) > MAX_PLANAR_DUTY:
                Tx_bal = np.sign(Tx_bal) * MAX_PLANAR_DUTY
            if np.abs(Ty_bal) > MAX_PLANAR_DUTY:
                Ty_bal = np.sign(Ty_bal) * MAX_PLANAR_DUTY

            if np.abs(Tx_steer) > 0.2:
                Tx_steer = np.sign(Tx_steer) * 0.2
            if np.abs(Ty_bal) > 0.2:
                Ty_steer = np.sign(Ty_steer) * 0.2

        else:
            if np.abs(Tx_bal) > MAX_STA_DUTY:
                Tx_bal = np.sign(Tx_bal) * MAX_STA_DUTY
            if np.abs(Ty_bal) > MAX_STA_DUTY:
                Ty_bal = np.sign(Ty_bal) * MAX_STA_DUTY



        Tz_steer = bb_controller.Tz

        Tx = Tx_bal + Tx_steer
        Ty = Ty_bal + Ty_steer
        Tz = Tz_bal + Tz_steer

            # -----------------------------------------------------
        # Saturating the planar torques
        # This keeps the system having the correct torque balance across the wheels in the face of
saturation of any motor during the conversion from planar torques to M1-M3
        if np.abs(Tx) > MAX_PLANAR_DUTY:
            Tx = np.sign(Tx) * MAX_PLANAR_DUTY

        if np.abs(Ty) > MAX_PLANAR_DUTY:
            Ty = np.sign(Ty) * MAX_PLANAR_DUTY

        # Conversion of planar torques to motor torques
        T1, T2, T3 = compute_motor_torques(Tx, Ty, Tz)

        # Sending motor torque commands to the pico
        commands['motor_1_duty'] = T1
        commands['motor_2_duty'] = T2
```

```
        commands['motor_3_duty'] = T3

        ser_dev.send_topic_data(101, commands)

        if zeroed:
            print(" << Iteration no: {}, DPHI X: {:.2f}, DPHI Y: {:.2f} THETA X: {:.2f}, THETA Y:
{:.2f}>>".format(i, dphi[0][0], dphi[1][0], theta_x, theta_y))
            # print("Iteration no. {}, ERROR X: {:.2f}, ERROR Y: {:.2f}, ERRORSUM X: {:.2f},
ERRORSUM Y: {:.2f}".format(i, error_x_bal, error_y_bal, error_x_sum_bal, error_y_sum_bal))
            #print("Iteration no. {}, ERROR X STEER: {:.2f}, ERROR Y STEER: {:.2f}, ERRORSUM
X STEER: {:.2f}, ERRORSUM Y STEER: {:.2f}".format(i, error_x_steer, error_y_steer,
error_x_sum_steer, error_y_sum_steer))
            print("Iteration no. {}, Tx: {:.2f}, Ty: {:.2f}, Tx Steer: {:.2f}, Ty Steer: {:.2f}, Tx bal: {:.2f}, Ty
bal: {:.2f}".format(i, Tx, Ty, Tx_steer, Ty_steer, Tx_bal, Ty_bal))
            print("Iteration no. {}, Controller x: {:.2f}, Controller y: {:.2f}".format(i,
bb_controller.dphi_x_sp, bb_controller.dphi_y_sp))
            # Construct the data matrix for saving - you can add more variables by replicating the
format below
            data = [i] + [t_now] + \
                [states['theta_roll']] + [states['theta_pitch']] + \
                    [Tx] + [Ty] + [Tz] + \
                        [T1] + [T2] + [T3] + \
                            [Tx_steer] + [Ty_steer] + [dphi_x] + [dphi_y] + [dphi_z] + \
                                [phi[0][0]] + [phi[1][0]] + [phi[2][0]] + \
                                    [dphi_roll_pid.setpoint] + [dphi_pitch_pid.setpoint] + \
                                        [theta_roll_sp] + [theta_pitch_sp]


            dl.appendData(data)

    print("Resetting Motor commands.")
    time.sleep(0.25)
    commands['motor_1_duty'] = 0.0
    commands['motor_2_duty'] = 0.0
    commands['motor_3_duty'] = 0.0
    time.sleep(0.25)
    commands['start'] = 0.0
    time.sleep(0.25)
    ser_dev.send_topic_data(101, commands)
    time.sleep(0.25)

    dl.writeOut()
```

Some constant definitions are in the attached file and submitted through canvas