ROB311: Ballbot Final Report

Authors: James Oosterhouse, Caitlin Roberts, and Wyatt Wrubel

Team A10

December 16, 2025

Introduction

This report describes our Ballbot system created for the University of Michigan Robotics class, ROB311: How to Build Robots and Make Them Move. Our robot is based off of the MBot ecosystem and is optimized to balance and steer while on top of a standard basketball. We designed mechanical aspects of our robot in Onshape and we used PID controllers to create the control structure. Our robot used sensor input from an on-board IMU as well as motor encoders. Our goal was to create a Ballbot that can compete and win our in-class competition, which required long-term balancing and steering algorithms to be effective and stable. We were proud that our ballbot took 3rd place in our in-class competition and was able to consistently balance despite its constraints.

Methodology

Device Communication & Information Transfer

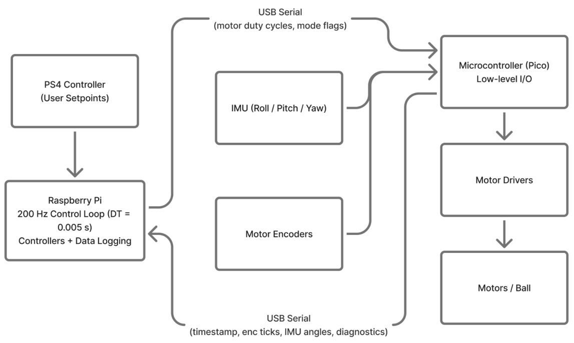
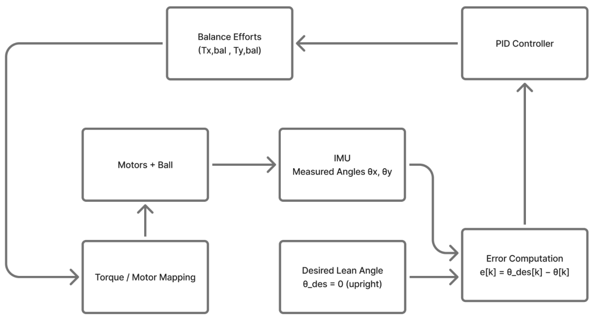


Figure 1. Ballbot System Architecture

Our Ballbot used a two computer control stack consisting of a Pico microcontroller that handles all of the low-level processing, and a Raspberry Pi for high-level control. This separation of boards allows for sensor data and motor actuation to be handled much faster from the computationally heavy control logic. The Pico interfaces directly with our onboard sensors, including the IMU and the motor encoders. During each cycle, the Pico receives data from them and sends them back to the Raspberry Pi. The Pico also receives commands from the Raspberry Pi, and sends those signals right to the motor drivers. The Raspberry Pi runs the main control code at 200 Hz. It takes all of the data from the Pico, evaluates it against the control algorithms, and generates the proper motor outputs. Both directions of this communication happen through USB serial. When needed, our PS4 controller also sends feedback to the Raspberry Pi through bluetooth. This information is then used in our control algorithms.

Controllers & Actuation



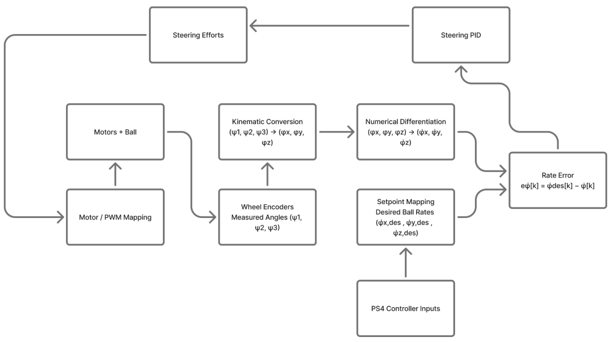

Steering Efforts
Steering PID
Motors + Ball
Kinematic Conversion
(ψ1, ψ2, ψ3) → (φx, φy, φz)
Numerical Differentiation
(φx, φy, φz) → (φ̇x, φ̇y, φ̇z)
Rate Error
eφ̇[k] = φ̇des[k] − φ̇[k]
Motor / PWM Mapping
Wheel Encoders
Measured Angles (ψ1, ψ2, ψ3)
Setpoint Mapping
Desired Ball Rates
(φ̇x,des , φ̇y,des , φ̇z,des)
PS4 Controller Inputs

Figure 2. Ballbot Balance Control Loop

Figure 3. Ballbot Steering Control Loop

Our Ballbot uses two coupled feedback control loops, one for balancing and one for steering. Both controls use the control architecture articulated in the above section. The Balance Controller is the primary loop that is always active, whereas the steering control modifies the balance behaved to produce controlled motion at a desired lean angle. The balance controller works by stabilizing the robot's lean angles in the X and Y axes. The onboard IMU measures these angles, and the balance controller computes the error between robot position and desired angle. Our tuned PID values as discussed in the next section allowed us to send commands back to the motors on our robot, and maintain our balance. Our steering controller allows both translational and rotational motion while maintaining our robot's balance. Our desired steering location is determined from user joystick input on a PS4 controller, and converted to a desired angle. This steering PID loop now controls our robots desire to lean in that direction, with smoothing in place to both lean to and from that angle. The balance and steering control outputs are summed together to form the total control effort for our ballbot, outputting torque commands to our motors.

PID Tuning

For our final PID values, we spent many hours meticulously honing them in until we found the optimal functionality for our own bot. Our final values were: $K_p = 13$, $K_i = 12$, $K_d = 0.1$. We started by tuning just K_p with no other gain involved. Initially adjusting the value in large amounts proved beneficial because once we saw our bot oscillating on top of the ball, we knew we'd gone too high. From there we were able to reduce the value at smaller increments until we landed on 13. With just K_p, our bot was balancing considerably better than the system without any error correction, but still needed some work.

Thus, we decided to introduce K_d next. To our surprise, we consistently found that K_d was making a negative impact on balance performance, so we wanted to minimize it to a point where our bot wasn't too jumpy but could still respond to quick changes in movement. This led to us settling at a value of 0.1, since anything above that was too reactive and anything below didn't assist as much as possible. Even with both of these gains our bot still couldn't hold a single position for very long, and the longer it balanced the worse it performed.

This led us to introducing a K_i term to counteract accumulated error in the system. We started off with incremental small values (increase by 0.02 with $K_i < 1$); however, this made little to no positive impact on the balancing, so we decided to increase K_i by whole numbers to see what happens. This made a massive improvement to our system and seemed to be the thing we were missing all along. In search of finding the best K_i, we pushed the value to 50, which proved to be much too far. We then decreased to 30, to 20, to the teens, then we settled at 12. Before K_i was involved, our bot would sloppily balance for about 15 seconds, then lean one way and never recover. However, the introduction of K_i solved both of these issues. The motors now actuate on a very small scale to keep the ball in virtually one spot on the floor and even if the bot does lean, it can recover and remount on top of the ball.

Our Best Practices for Control

We attempted to implement several best practices to increase the accuracy of our control algorithm but ultimately ended up using only two of them: dead-band of 0.3° around 0° on the IMU and low-pass filter of controller steering commands. The dead-band essentially removes small amounts of noise when the bot is perfectly mounted on top of the ball. This worked very well because it allows for less accumulation of error and less unnecessary small motor actuation, which we noticed during testing. The low-pass filter acts as a mode of smoothing the steering commands that come from the joystick. A human is inherently bad at making precise, continuous movements of their thumb on a joystick, resulting in jerky inputs which are devastating to the adjustment of the ballbot's reference angle. This filter ensures that quick movements of the joystick won't throw the bot off of the ball, but instead correlate to a smooth change in reference angle, enabling effective steering.

We also attempted to dead-band the joystick and limit integral gain windup. Though these are good ideas in theory, we ended up having issues with both. Dead-banding the joystick in a small amount should've just removed any physical stick drift in the controller when at rest, but it ended up rendering our steering algorithm useless. In other words, motion of the joystick wouldn't correlate to any physical steering of the ballbot. Additionally, our idea for limiting integral gain windup came from the fact that K_i increased significantly whenever the bot and ball would start to "run away," always resulting in the bot falling off. We thought that if we set a limit to the max K_i then it would help the runaway situation, but it just made it even worse. This does make sense because if K_i is limited when leaning, then the bot's control efforts are less than they could be, resulting in inefficient lean recovery.

Results

IMU vs Encoder Spin Speeds

In order to collect data comparing IMU measurements to encoder calculations of spin speed ($d\vartheta_z$), we controlled the ball bot with 4 different duty cycles. Each cycle was 3 seconds, with Tz = [0.75, 1.5, 2.25, 3] for each of the four cycles, respectively. We controlled the torque output for three seconds to be able to trim noisy data during acceleration between duty cycles.

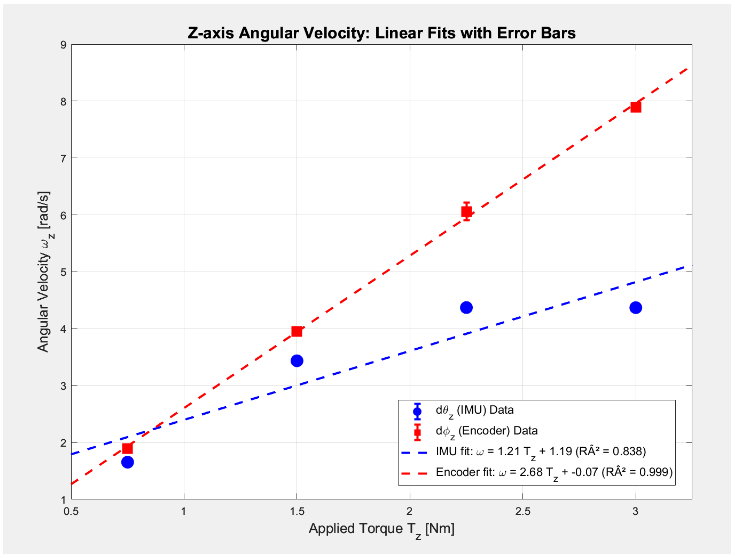


Figure 4. Ballbot IMU Measurements vs Encoder Calculations

While varying Tz for each cycle, we recorded IMU angles (ϑ_x, ϑ_y, ϑ_z) and encoder angular velocities ($d\varphi_x$, $d\varphi_y$, $d\varphi_z$) from the ballbot datalogger. The orientation angles (ϑ_x, ϑ_y, ϑ_z) are obtained from the ballbot's IMU measurements. Each angle is relative to the initial reference value to express the ballbot's orientation relative to the starting pose. To reduce the effect of sensor noise and prevent small, unnecessary corrections, we apply a deadzone filter to each angle. The deadzone filter clips any angle less than the deadzone (0.3°) to zero, while values that exceed this threshold are unchanged. This filtering step improves the stability of control and allows for smoother angle error correction when balancing and steering the ball-bot.

The ballbot encoder data measures the position of each wheel on the ballbot. The raw encoder counts are converted into wheel rotation angles (ψ_1, ψ_2 ,ψ_3) in radians. Wheel angular velocities ($d\psi_1$, $d\psi_2$, $d\psi_3$) are computed by differentiating psi123 with respect to time. We then map the wheel angular velocity to the ball's motion with a kinematic conversion model. The ball's angular velocities ($d\varphi_x$, $d\varphi_y$, $d\varphi_z$) as seen in Figure 4 capture the rotational motion of the ball.

Once we obtained the IMU and encoded angular velocities at four different Tz duty cycles we used Matlab to present Figure 4, a comparison of the Z-axis angular velocity measured by the IMU and estimated from the wheel encoders. For each Tz value, we extracted a smaller time window without ballbot accelerations, and calculated the mean and standard deviations for $d\vartheta_z$ and $d\varphi_z$.

As seen in Figure 4, the angular velocity estimated from the wheel encoders is consistently higher than the angular velocity measured by the IMU. This difference can be explained by factors related to sensing, assumptions when modeling, and interactions between the ball and the wheel. First, the encoder-based angular velocity is mapped using a kinematic model that assumes ideal rolling conditions without slip. In reality, slip occurs, which causes the wheels to rotate slightly faster than the ball itself, leading to higher values of $d\varphi_z$. Additionally, the deadzone filter that we apply to IMU angles reduces sensitivity to small rotations that are close to zero. Although this is helpful for balance control, it may contribute to a lower average angular velocity from the IMU relative to the encoder's estimate.

Balance Control

To develop the balance controller, we tuned the PID controller based on the ballbot's observed behaviors at different proportional (K_p), integral (K_i), and derivative (K_d) values. The proportional term responds to the current error, the integral term reduces steady state error, and the derivative term dampens oscillations by reacting to small changes in error. Figure 5 shows the ballbot's lean angle, individual PID control gains, and resulting torque output along the x-axis for 15 seconds of balancing. For our controller, the proportional term contributes the most to the torque output and is the most dominant component of our balance control.

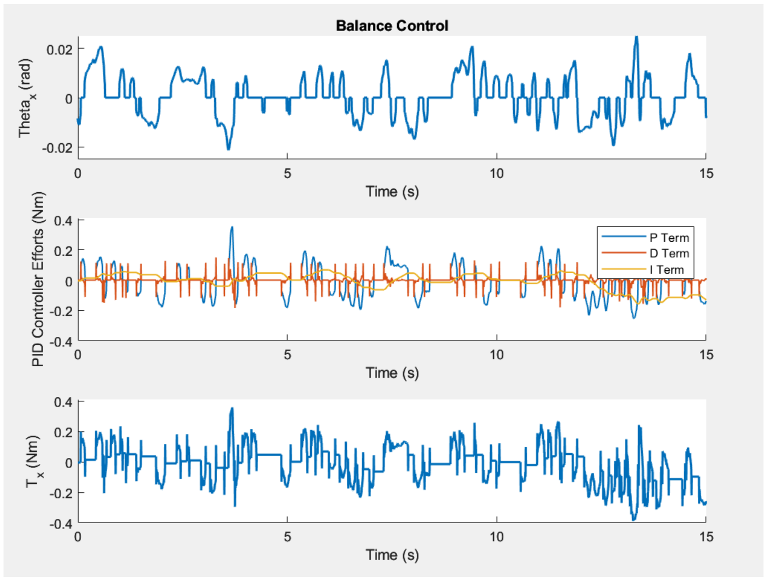


Figure 5. Ball Angle, PID Control Efforts, and Total Torque for Balance

When comparing subplots two and three in Figure 5, we see that the shape of the torque plot closely matches the shape of the proportional control term of the PID controller. This indicates that the balance control's response is primarily driven by the current lean angle error. In contrast, the integral and derivative terms have smaller magnitudes overall. We theorize that this is because the integral and derivative gains work to remove different types of error, past and future error. The integral and derivative terms serve mainly to correct for residual error and oscillations, rather than drive the overall balance control effort.

Steering Control

When comparing our ball's odometry to actual position, we used our steering control to drive the ballbot in a square approximately 2m x 2m. In order to determine the robot's odometry, we recorded the ball's angular position (φ_x and φ_y) during the steering control. φ_x and φ_y are calculated based on a kinematic conversion from the wheel rotation (ψ_1, ψ_2 ,ψ_3). The ballbot's wheel rotation angles are determined based on encoder readings on each of the wheels. Using the equations (1) $x_k = -r_k * \varphi_y$ and (2) $y_k = r_k * \varphi_x$, with $r_k = 0.121$m (the radius of the ball) we are able to calculate the ballbot's odometry. Figure 6 plots the ballbot's odometry when driving in a square.

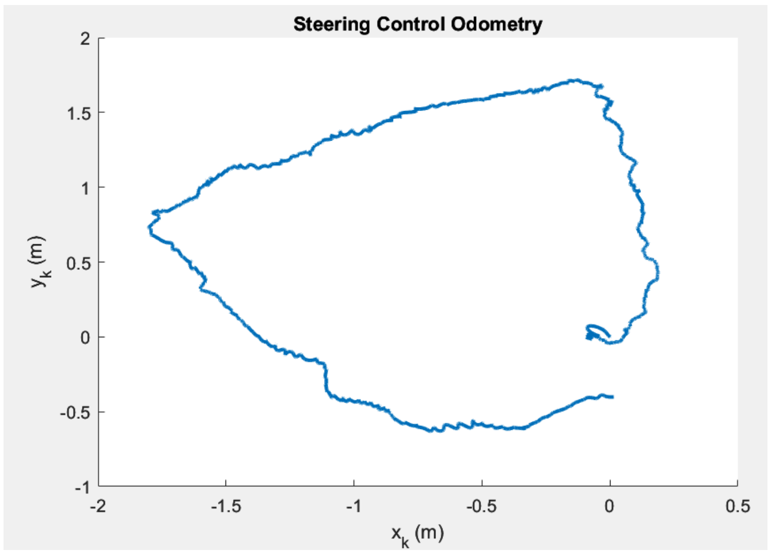


Figure 6. Ball Odometry Driving a 2m x 2m Square

As seen in the figure, the ball's odometry shows a lot of drift from the actual position relative to the start. This discrepancy is due to accumulated odometry error over time. The odometry is computed entirely from wheel encoder measurements. Any small error in the encoder readings or kinematic conversions from wheel rotations to ball angles accumulates overtime. Minor inaccuracies can cause drift, resulting in a lopsided or skewed square, like seen in Figure 6.

Second, the wheels can slip or have uneven contact on the surface of the ball. When steering the ballbot, especially at square corners, slip is likely to occur on the ball, causing the measured wheel rotation to differ from the actual ball motion. This leads to incorrect $\varphi_x$and φ_y values and therefore incorrect position estimates. Over the course of the square path, these errors accumulate and produce a shape other than a perfect square. Finally, we did not implement a position PID controller, so there was no mechanism in our ballbot to correct for odometry errors. Therefore, the ballbot's odometry estimate will differ from the true position of the ball.

Conclusion & Future Work

In this project, we designed, built, and tested a Ballbot that balances and steers on a standard basketball using an IMU, motor encoders, and two PID loops for balancing and steering. We quantified performance through logged experiments including IMU vs encoder yaw-rate comparisons, steady-state balancing plots, and a square driving test for odometry. The results showed that encoder-based yaw-rate estimates were consistently higher than IMU measurements, which we attribute mainly to slip and non-ideal rolling that violate kinematic assumptions, and our balancing data showed that the proportional term dominated the control effort while the integral and derivative terms mainly reduced residual bias and oscillations. The square driving test demonstrated significant odometry drift over time due to accumulated encoder and modeling error and the lack of a position-level correction loop.

With more time, we would improve robustness by reducing slip sensitivity through better wheel-ball contact, adding stronger filtering and saturation best practices in the control pipeline, and implementing mechanical changes to the robot to overall increase its effectiveness. Some of these mechanical changes may include increased robot Center of Mass or adding balancing fingers as seen in the ETH Zurich paper. Overall, this project was a fantastic learning experience and we are very satisfied with the robot we developed.

Appendix A - Ballbot Figures and Terms

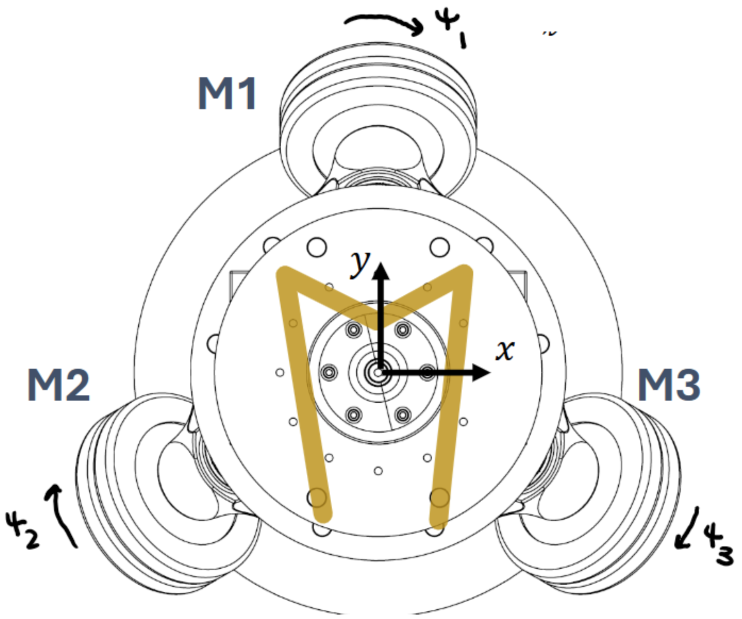


Figure A. Wheel Rotation Angles, Top View

Wheel Rotation Angles (ψ_1, ψ_2 ,ψ_3) as calculated from motor encoder ticks. ψ_1, ψ_2 ,ψ_3 are used to calculate the ball's angular position (φ_x, $\varphi_{y,}$ φ_z) in the x-y-z plane. In turn, we can calculate the changes in these terms by taking the derivative of each.

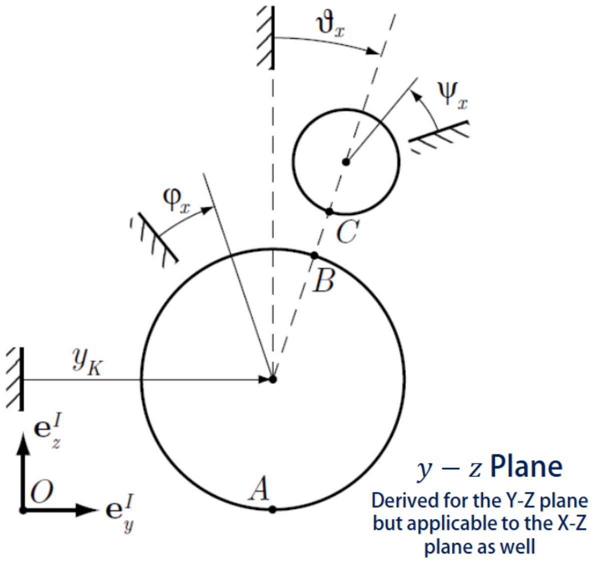


Figure B. Position of the Ball, Side View

ϑ_x as seen in the figure, represents the position of the ball relative to its starting position ($\vartheta_x = 0$). As discussed in the balance control portion of the report, the angle of the ballbot is used to drive balance control.

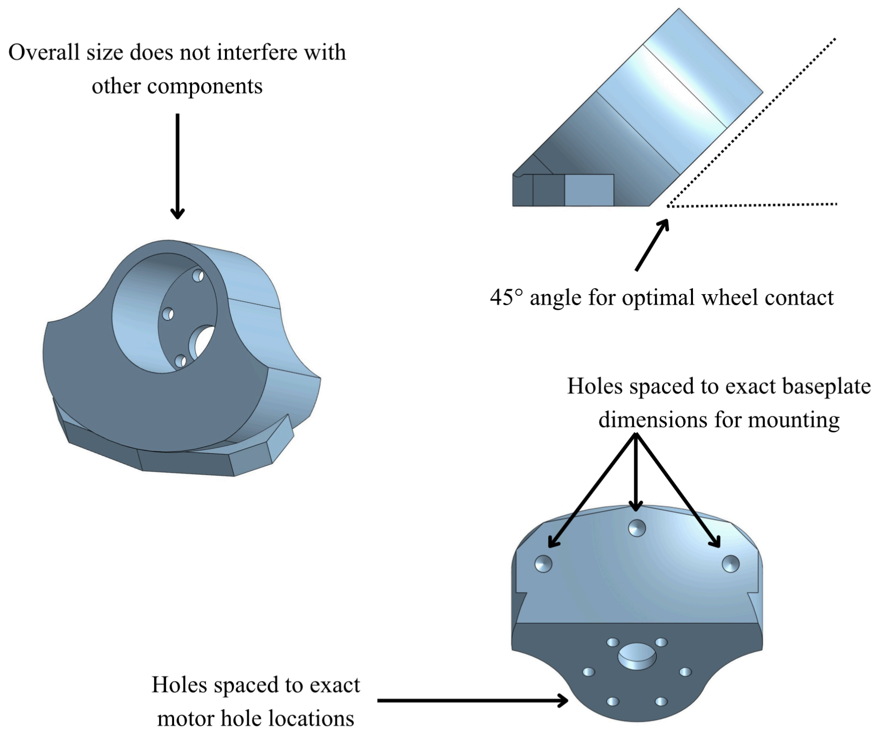

Overall size does not interfere with other components
45° angle for optimal wheel contact
Holes spaced to exact baseplate dimensions for mounting
Holes spaced to exact motor hole locations

Figure C. CAD of Motor Mounts, Various Views

Though we were given creative liberty in designing our motor mounts, there were still requirements we had to meet to ensure proper ball bot functionality. The mounts must: not interfere with internal bot components, ensure 45° wheel contact with the ball, mount to the laser cut holes on the baseplate, and securely house our motors. Satisfaction of these requirements can be found in Figure C.

Appendix B - Additional Figures

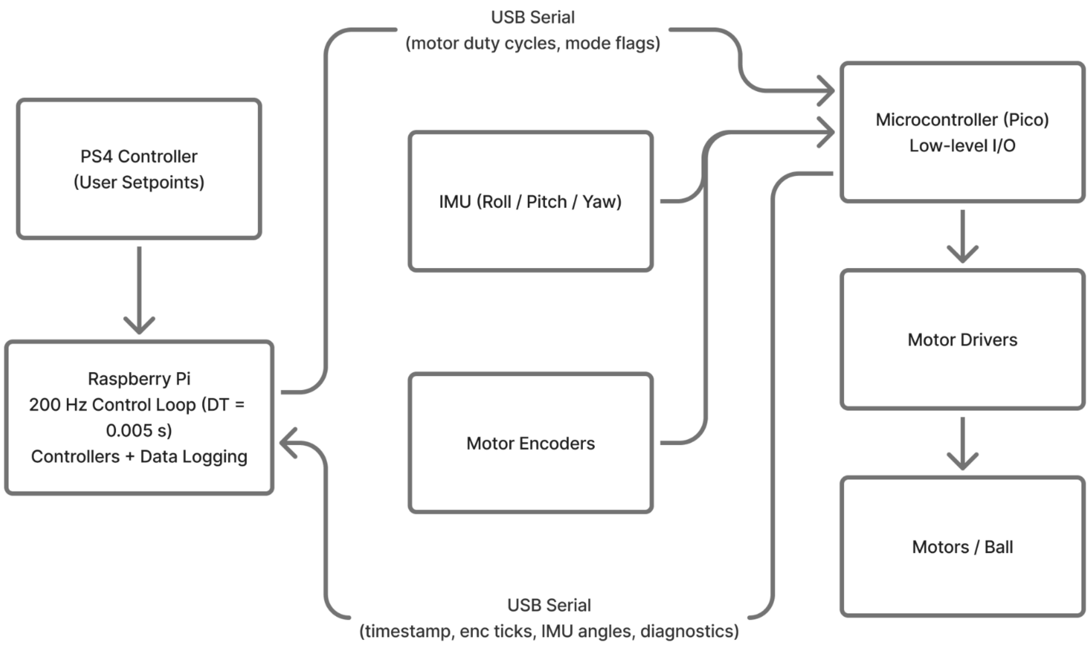


Figure 1. Ballbot System Architecture (Enlarged)

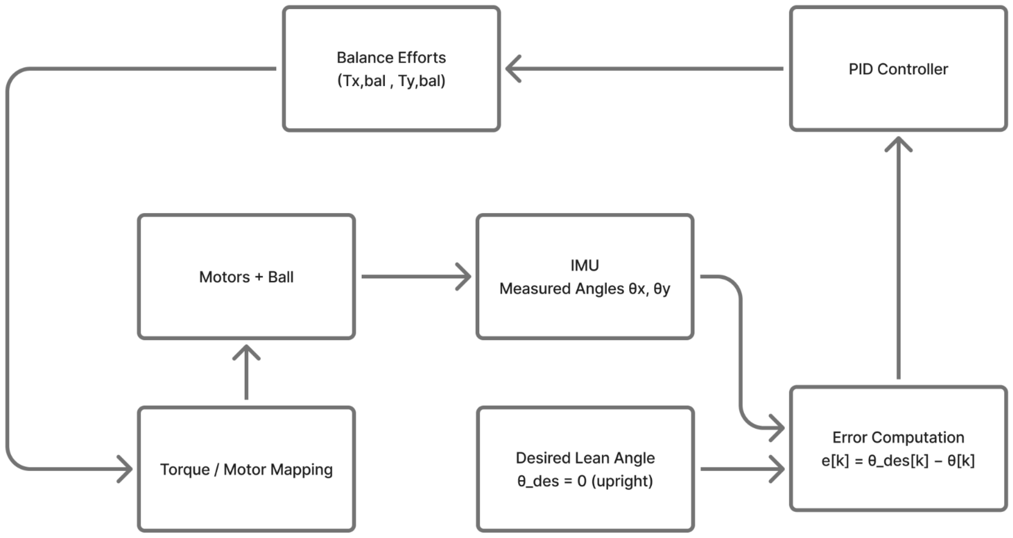
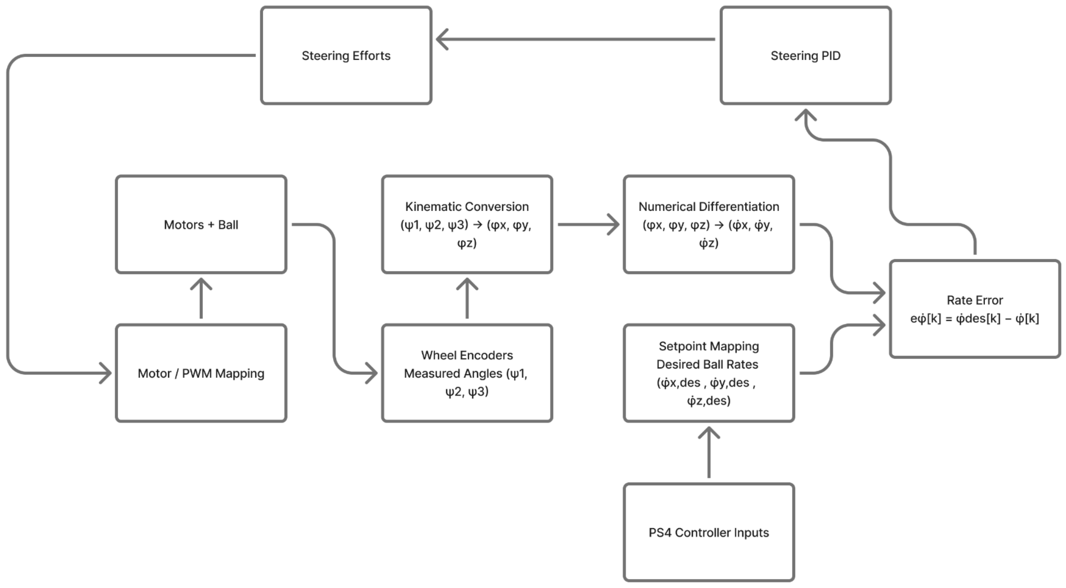


Figure 2. Ballbot Balance Control Loop (Enlarged)



Steering Efforts
Steering PID
Motors + Ball
Kinematic Conversion (ψ1, ψ2, ψ3) → (φx, φy, φz)
Numerical Differentiation (φx, φy, φz) → (φ̇x, φ̇y, φ̇z)
Rate Error eφ̇[k] = φ̇des[k] − φ̇[k]
Motor / PWM Mapping
Wheel Encoders Measured Angles (ψ1, ψ2, ψ3)
Setpoint Mapping Desired Ball Rates (φ̇x,des , φ̇y,des , φ̇z,des)
PS4 Controller Inputs

Figure 3. Ballbot Steering Control Loop (Enlarged)

Ballbot Team A10 (*left to right*): Wyatt Wrubel, James Oosterhouse, and Caitlin Roberts

Appendix C - Python Code

File 1: PID_Controller.py

```
"""
General framework for ball-bot control for students to update as desired.
You may wish to make multiple versions of this file to run your ball-bot in
different modes!

struct mbot_balbot_feedback_t
{
    int64_t utime;
    int32_t enc_ticks[3];       // absolute postional ticks
    int32_t enc_delta_ticks[3]; // number of ticks since last step
    int32_t enc_delta_time;     // [usec]
    float imu_angles_rpy[3];    // [radian]
    float volts[4];             // volts
}

"""

import time
import lcm
import threading
import numpy as np
from mbot_lcm_msgs.mbot_motor_pwm_t import mbot_motor_pwm_t
from mbot_lcm_msgs.mbot_balbot_feedback_t import mbot_balbot_feedback_t
from DataLogger3 import dataLogger
from ps4_controller_api import PS4InputHandler

# Constants for the control loop
FREQ = 200  # Frequency of control loop [Hz]
DT = 1 / FREQ  # Time step for each iteration [sec]
PWM_MAX = 0.98  # Max motor signal for full accel/decel of motor test (keep
between 0 and 1)
N_GEARBOX = 70 # Motor gearbox ratio
N_ENC = 64 # Ticks per revolution of encoder
R_W = 0.048 # Radii of omni-wheels [m]
R_K = 0.121 # Radius of basketball [m]
IMU_DEADZONE = np.radians(.3)  # 0.5 degrees in radians ≈ 0.00873 rad

# Global flags to control the listening thread & msg data
listening = False
msg = mbot_balbot_feedback_t()
last_time = 0
last_seen = {"MBOT_BALBOT_FEEDBACK": 0}

def feedback_handler(channel, data):
    """Callback function to handle received mbot_balbot_feedback_t messages"""
    global msg
    global last_seen
    global last_time
    last_time = time.time()
    last_seen[channel] = time.time()
    msg = mbot_balbot_feedback_t.decode(data)


#Filtering
def apply_deadzone(x, deadzone):
    """
    Apply a deadzone filter to sensor readings.
    Values within ±deadzone are set to zero to filter out noise.
    This prevents constant small corrections due to sensor noise.
    """
    if abs(x) < deadzone:
        return 0.0
    return x


def lcm_listener(lc):
    """Function to continuously listen for LCM messages in a separate thread"""
    global listening
    while listening:
        try:
            lc.handle_timeout(100)  # 100ms timeout
            if time.time() - last_time > 2.0:
                print("LCM Publisher seems inactive...")
            elif time.time() - last_seen["MBOT_BALBOT_FEEDBACK"] > 2.0:
                print("LCM MBOT_BALBOT_FEEDBACK node seems inactive...")
        except Exception as e:
            print(f"LCM listening error: {e}")
            break

# Motor encoder ticks to wheel angle (radians)
def calc_enc2rad(ticks):
    rad = (2 * np.pi * ticks) / (N_ENC * N_GEARBOX)
    return rad

# Calculate motor torques T1, T2, T3 from Tx, Ty, Tz
def calc_torque_conv(Tx,Ty,Tz):
    u1 = (1.0/3.0)*(Tz - 2*np.sqrt(2)*Ty)
    u2 = (1.0/3.0)*(Tz + np.sqrt(2)*(-1.0*np.sqrt(3)*Tx + Ty))
    u3 = (1.0/3.0)*(Tz + np.sqrt(2)*(np.sqrt(3)*Tx + Ty))

    return u1, u2, u3

# Calculate ball angular position from encoder odometry
def calc_kinematic_conv(psi1,psi2,psi3):
    phix = np.sqrt(2.0/3.0) * (R_W/R_K) * (psi2 - psi3)
    phiy = np.sqrt(2)/3.0 * (R_W/R_K) * (-2 * psi1 + psi2 + psi3)
    phiz = np.sqrt(2)/3.0 * (R_W/R_K) * (psi1 + psi2 + psi3)

    return phix, phiy, phiz

def func_clip(x,lim_lo,lim_hi):
    # A function to clip values that exceed a threshold [lim_lo,lim_hi]
    if x > lim_hi:
        x = lim_hi
    elif x < lim_lo:
        x = lim_lo
    return x


def main():
    # === Data Logging Initialization ===
    # Prompt user for trial number and create a data logger
    trial_num = int(input("Test Number? "))
    filename = f"PID_control_{trial_num}.txt"
    dl = dataLogger(filename)

    # === LCM Messaging Initialization ===
    # Initialize the serial communication protocol
    global listening
    global msg
    lc = lcm.LCM("udpm://239.255.76.67:7667?ttl=0")
    subscription = lc.subscribe("MBOT_BALBOT_FEEDBACK", feedback_handler)
    # Start a separate thread for reading LCM data
    listening = True
    listener_thread = threading.Thread(target=lcm_listener, args=(lc,),
daemon=True)
    listener_thread.start()
    print("Started continuous LCM listener...")

    #print("Waiting for first IMU message...")
    #if not wait_for_feedback():
    #   print("[WARN] No IMU feedback.")

    # theta_x_0, theta_y_0, theta_z_0 = calibrate_imu(duration=5)

    enc_pos_1_start = msg.enc_ticks[0]
    enc_pos_2_start = msg.enc_ticks[1]
    enc_pos_3_start = msg.enc_ticks[2]

    # === Controller Initialization ===
    # Create an instance of the PS4 controller handler
    controller =
PS4InputHandler(interface="/dev/input/js0",connecting_using_ds4drv=False)
    # Start a separate thread to listen for controller inputs
    controller_thread = threading.Thread(target=controller.listen, args=(10,))
    controller_thread.daemon = True  # Ensures the thread stops with the main
program
    controller_thread.start()
    print("PS4 Controller is active...")

    try:
        command = mbot_motor_pwm_t()
        # === Main Control Loop ===
        print("Starting steering control loop...")
        time.sleep(0.5)

        # Store variable names as header to data logged, for easier parsing in
Matlab
        # TODO [IF DESIRED]: Update data header variables names to match actual
data logged (at end of loop)
        data = ["i t_now phi_x phi_y"]
        dl.appendData(data)

        i = 0  # Iteration counter
        t_start = time.time()
        t_now = 0

        enc_pos_1_start = msg.enc_ticks[0]
        enc_pos_2_start = msg.enc_ticks[1]
        enc_pos_3_start = msg.enc_ticks[2]

        # Initialize Torque Commands
        u1 = 0
        u2 = 0
        u3 = 0

        # Starting IMU Orientation
        theta_x_0 = msg.imu_angles_rpy[0]
        theta_y_0 = msg.imu_angles_rpy[1]
        theta_z_0 = msg.imu_angles_rpy[2]

        desired_theta = 0.0 # upright

        Kp = 13 # Proportional gain
        # Between 0-15
        Ki = 12  # Integral gain
        Kd = .1  # Derivative gain


        # Starting error and integral terms
        prev_error_x, prev_error_y = 0.0, 0.0
        integral_x, integral_y = 0.0, 0.0
        motor_on = 0
        error_x = 0.0
        error_y = 0.0
        desired_theta_x = 0
        desired_theta_y = 0
        prev_dpad_up = 0
        prev_dpad_down = 0
        prev_dpad_right = 0
        prev_dpad_left = 0

        while True:
            time.sleep(DT)
            t_now = time.time() - t_start  # Elapsed time
            i += 1

            try:
                # retreive dictionary of button press signals from handler
                bt_signals = controller.get_signals()

                shoulder_L1 = bt_signals["shoulder_L1"]

                if shoulder_L1 == 1:  # Rising edge detection
                   motor_on += 1

                # PID Tuning - DPad
                # Previous button states for edge detection
                prev_dpad_up = 0
                prev_dpad_down = 0
                prev_dpad_right = 0
                prev_dpad_left = 0
                if (motor_on >= 1):#Calculate Theta Error
                    error_x = desired_theta_x - theta_x
                    integral_x += error_x
                    integral_y += error_y
                # Debouncing
                last_gain_change_time = 0
                GAIN_CHANGE_COOLDOWN = 0.12  # 120ms between changes

                # Gain selection index: 0=Kp, 1=Ki, 2=Kd
                gain_sel = 0

                # Increment amounts for each gain type
                GAIN_INC = {0: 0.1, 1: 0.01, 2: 0.01}
                #           Kp↑    Ki↑     Kd↑

                # Read D-pad button states (every loop iteration at 200 Hz)
                dpad_up = bt_signals["dir_U"]      # 0 or 1
                dpad_down = bt_signals["dir_D"]    # 0 or 1
                dpad_right = bt_signals["dir_R"]   # 0 or 1
                dpad_left = bt_signals["dir_L"]    # 0 or 1`

                # LEFT button: cycle backward through gains
                if dpad_left == 1 and prev_dpad_left == 0:  # Rising edge
detection
                    gain_sel = (gain_sel - 1) % 3  # P←I←D←P (wraps around)
                    selection_changed = True
                    last_gain_change_time = t_now

                # RIGHT button: cycle forward through gains
                elif dpad_right == 1 and prev_dpad_right == 0:
                    gain_sel = (gain_sel + 1) % 3  # P→I→D→P (wraps around)
                    selection_changed = True
                    last_gain_change_time = t_now

                # UP button: INCREASE selected gain
                elif dpad_up == 1 and prev_dpad_up == 0:
                    inc = GAIN_INC.get(gain_sel, 0.01)  # Get increment size
                    if gain_sel == 0:     # Kp selected
                        Kp += inc  # Increase by 0.1
                        #Kp += inc
                    elif gain_sel == 1:   # Ki selected
                        Ki += inc  # Increase by 0.01
                        #Ki += inc
                    elif gain_sel == 2:   # Kd selected
                        Kd += inc  # Increase by 0.01
                        #Kd += inc
                    gain_changed = True
                    last_gain_change_time = t_now

                # DOWN button: DECREASE selected gain
                elif dpad_down == 1 and prev_dpad_down == 0:
                    inc = GAIN_INC.get(gain_sel, 0.01)  # Get increment size
                    if gain_sel == 0:     # Kp selected
                        Kp -= inc  # Increase by 0.1
                        #Kp -= inc
                    elif gain_sel == 1:   # Ki selected
                        Ki -= inc  # Increase by 0.01
                        #Ki -= inc
                    elif gain_sel == 2:   # Kd selected
                        Kd -= inc  # Increase by 0.01
                        #Kd -= inc
                    gain_changed = True
                    last_gain_change_time = t_now

                # parse out individual buttons you want data from
                js_R_x = bt_signals["js_R_x"]   # steering bot (XY) with js_R
                js_R_y = bt_signals["js_R_y"]

                trigger_L2 = bt_signals["trigger_L2"]   # spinning bot (Z) with
L2/R2 triggers
                trigger_R2 = bt_signals["trigger_R2"]

                # Raw IMU Orientation
                theta_x_raw = msg.imu_angles_rpy[0] - theta_x_0
                theta_y_raw = msg.imu_angles_rpy[1] - theta_y_0
                theta_z_raw = msg.imu_angles_rpy[2] - theta_z_0

                # Filtered IMU Data
                theta_x = apply_deadzone(theta_x_raw, IMU_DEADZONE)
                theta_y = apply_deadzone(theta_y_raw, IMU_DEADZONE)
                theta_z = apply_deadzone(theta_z_raw, IMU_DEADZONE)

                #Calculate Theta Error
                error_x = desired_theta - theta_x
                print("error_x: ", error_x)
                print("\n")
                error_y = desired_theta - theta_y
                print("error_x: ", error_x)
                print("\n")

                integral_x += error_x
                integral_y += error_y

                # Encoder ticks 1, 2, and 3
                enc_pos_1 = msg.enc_ticks[0] - enc_pos_1_start
                enc_pos_2 = msg.enc_ticks[1] - enc_pos_2_start
                enc_pos_3 = msg.enc_ticks[2] - enc_pos_3_start

                # Change in ticks
                enc_dtick_1 = msg.enc_delta_ticks[0]
                enc_dtick_2 = msg.enc_delta_ticks[1]
                enc_dtick_3 = msg.enc_delta_ticks[2]

                #Change in time (microseconds)
                enc_dt = msg.enc_delta_time

                # Calculate motor angles from encoder ticks
                # Wheel Rotation Angles (input)
                psi_1 = calc_enc2rad(enc_pos_1)
                psi_2 = calc_enc2rad(enc_pos_2)
                psi_3 = calc_enc2rad(enc_pos_3)
                print('Psi1: ', psi_1, 'Psi2: ', psi_2, 'Psi3: ', psi_3)

                # Wheel Angular Velocities
                dpsi_1 = calc_enc2rad(1e6*(enc_dtick_1/enc_dt))
                dpsi_2 = calc_enc2rad(1e6*(enc_dtick_2/enc_dt))
                dpsi_3 = calc_enc2rad(1e6*(enc_dtick_3/enc_dt))
                print('dPsi1: ', dpsi_1, 'dPsi2: ', dpsi_2, 'dPsi3: ', dpsi_3)

                # Calculate ball's roll and translation through kinematic
conversions of wheel data
                # Ball Angular Position
                phi_x, phi_y, phi_z = calc_kinematic_conv(psi_1, psi_2, psi_3)

                # Ball Angular Velocity
                dphi_x, dphi_y, dphi_z =
calc_kinematic_conv(dpsi_1,dpsi_2,dpsi_3)

                # Calculate the desired x,y,z torque commands
                # LOW PASS FILTER ON THE D TERM
                Ty = Kp * error_x + Kd*(error_x - prev_error_x)/DT +
Ki*integral_x*DT

                Kp_x = Kp*error_y
                Kd_x = Kd*(error_y - prev_error_y)/DT
                Ki_x = Ki*integral_y*DT
                Tx = Kp_x + Kd_x + Ki_x
                #Tx = Kp * error_y + Kd*(error_y - prev_error_y)/DT +
Ki*integral_y*DT
```

```
                Tz = 0

                # Calculate motor effort/commands from desired Tx,Ty,Tz motion
                u1, u2, u3 = calc_torque_conv(Tx,Ty,Tz)
                u1 = func_clip(u1,-PWM_MAX,PWM_MAX)

                # Set the Previous Error
                prev_error_x = error_x
                prev_error_y = error_y

                # Send individual motor commands
                u1 = func_clip(u1,-PWM_MAX,PWM_MAX)
                u2 = func_clip(u2,-PWM_MAX,PWM_MAX)
                u3 = func_clip(u3,-PWM_MAX,PWM_MAX)
                cmd_utime = int(time.time() * 1e6)
                command.utime = cmd_utime
                if (motor_on >= 1):
                    command.pwm[0] = -u1
                    command.pwm[1] = -u2
                    command.pwm[2] = -u3
                lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

                # Store data in data logger
                data = [i, t_now, phi_x, phi_y]
                dl.appendData(data)

                # Print out data in terminal
                # TODO: [IF DESIRED]: Update for what info you want to see in
terminal (note: this is only printed data, not logged!)
                print(
                    f"theta_x: {theta_x} | theta_y: {theta_y} | theta_z:
{theta_z} |"
                )

                # Emergency Stop with Triangle Button
                emergency_stop = bt_signals["but_tri"]
                if emergency_stop == 1:
                    # Immediately stop all motors
                    command.pwm = [0.0, 0.0, 0.0]
                    command.pwm[0] = 0.0
                    command.pwm[1] = 0.0
                    command.pwm[2] = 0.0

                    lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

                    print("\n" + "!"*80)
                    print("!!! EMERGENCY STOP ACTIVATED !!!")
                    print("!!! Triggered by: TOUCHPAD PRESS !!!")
                    print("!"*80)
                    print("\nAll motors stopped. Exiting control loop
safely...")
                    print("System halted.\n")
                    # Exit the control loop
                    break  # Exits the while True loop

                # Reset the IMU using Shoulder_R1
                shoulder_R1 = bt_signals["shoulder_R1"]
                if shoulder_R1 == 1:
                    theta_x_0 = msg.imu_angles_rpy[0]  # Capture NEW reference
pitch
                    theta_y_0 = msg.imu_angles_rpy[1]  # Capture NEW reference
pitch
                    theta_z_0 = msg.imu_angles_rpy[2]  # Capture NEW reference
pitch


            except KeyError:
                print("Waiting for sensor data...")

    except KeyboardInterrupt:
        print("\nKeyboard interrupt received. Stopping motors...")
        # Emergency stop
        command = mbot_motor_pwm_t()
        command.utime = int(time.time() * 1e6)
        command.pwm[0] = 0.0
        command.pwm[1] = 0.0
        command.pwm[2] = 0.0
        lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

    finally:
        # Save/log data
        print(f"Saving data as {filename}...")
        dl.writeOut()  # Write logged data to the file
        # Stop the listener thread
        listening = False
        print("Stopping LCM listener...")
        listener_thread.join(timeout=1)  # Wait up to 1 second for thread to
finish
        # Stop Bluetooth thread
        controller_thread.join(timeout=1)  # Wait up to 1 second for thread to
finish
        controller.on_options_press()
        # Stop motors
        print("Shutting down motors...\n")
        command = mbot_motor_pwm_t()
        command.utime = int(time.time() * 1e6)
        command.pwm[0] = 0.0
        command.pwm[1] = 0.0
        command.pwm[2] = 0.0
        lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

if __name__ == "__main__":
    main()
```

```
# Extra IMU callibration stuff
#def wait_for_feedback(timeout=3.0):
 #   t0 = time.time()
  #  while time.time() - t0 < timeout:
        # got at least one message?
   #     if hasattr(msg, "imu_angles_rpy") and len(msg.imu_angles_rpy) == 3:
    #        return True
     #   time.sleep(0.01)
    #return False

#def calibrate_imu(duration=1.5):
 #   print(f"[CAL] Hold robot still ~{duration}s to calibrate IMU zero...")
  #  xs, ys, zs = [], [], []
   # t0 = time.time()
    #while time.time() - t0 < duration:
      #  xs.append(msg.imu_angles_rpy[0])
       # ys.append(msg.imu_angles_rpy[1])
        #zs.append(msg.imu_angles_rpy[2])
        #time.sleep(0.005)  # ~200 Hz
    #x0 = float(np.mean(xs))
    #y0 = float(np.mean(ys))
    #z0 = float(np.mean(zs))
    #print(f"[CAL] Offsets -> theta_x_0={x0:.4f}, theta_y_0={y0:.4f}, 
theta_z_0={z0:.4f}")
    #return x0, y0, z0
```

File 2: Steering.py

```
"""
General framework for ball-bot control for students to update as desired.
You may wish to make multiple versions of this file to run your ball-bot in 
different modes!

struct mbot_balbot_feedback_t
{
    int64_t utime;
    int32_t enc_ticks[3];       // absolute postional ticks
    int32_t enc_delta_ticks[3]; // number of ticks since last step
    int32_t enc_delta_time;     // [usec]
    float imu_angles_rpy[3];    // [radian]
    float volts[4];             // volts
}

"""

import time
import lcm
import threading
import numpy as np
from mbot_lcm_msgs.mbot_motor_pwm_t import mbot_motor_pwm_t
from mbot_lcm_msgs.mbot_balbot_feedback_t import mbot_balbot_feedback_t
from DataLogger3 import dataLogger
from ps4_controller_api import PS4InputHandler

# Constants for the control loop
FREQ = 200  # Frequency of control loop [Hz]
DT = 1 / FREQ  # Time step for each iteration [sec]
PWM_MAX = 0.98  # Max motor signal for full accel/decel of motor test (keep
between 0 and 1)
N_GEARBOX = 70 # Motor gearbox ratio
N_ENC = 64 # Ticks per revolution of encoder
R_W = 0.048 # Radii of omni-wheels [m]
R_K = 0.121 # Radius of basketball [m]
IMU_DEADZONE = np.radians(.3)  # 0.5 degrees in radians ≈ 0.00873 rad

# Steering constants
THETA_MAX = np.radians(3)
ALPHA_THETA = 0.005
Tz_scale = 0.4

# Global flags to control the listening thread & msg data
listening = False
msg = mbot_balbot_feedback_t()
last_time = 0
last_seen = {"MBOT_BALBOT_FEEDBACK": 0}

def feedback_handler(channel, data):
    """Callback function to handle received mbot_balbot_feedback_t messages"""
    global msg
    global last_seen
    global last_time
    last_time = time.time()
    last_seen[channel] = time.time()
    msg = mbot_balbot_feedback_t.decode(data)


#Filtering
def apply_deadzone(x, deadzone):
    """
    Apply a deadzone filter to sensor readings.
    Values within ±deadzone are set to zero to filter out noise.
    This prevents constant small corrections due to sensor noise.
    """
    if abs(x) < deadzone:
        return 0.0
    return x


def lcm_listener(lc):
    """Function to continuously listen for LCM messages in a separate thread"""
    global listening
    while listening:
        try:
            lc.handle_timeout(100)  # 100ms timeout
            if time.time() - last_time > 2.0:
                print("LCM Publisher seems inactive...")
            elif time.time() - last_seen["MBOT_BALBOT_FEEDBACK"] > 2.0:
                print("LCM MBOT_BALBOT_FEEDBACK node seems inactive...")
        except Exception as e:
            print(f"LCM listening error: {e}")
            break

# Motor encoder ticks to wheel angle (radians)
def calc_enc2rad(ticks):
    rad = (2 * np.pi * ticks) / (N_ENC * N_GEARBOX)
    return rad

# Calculate motor torques T1, T2, T3 from Tx, Ty, Tz
def calc_torque_conv(Tx,Ty,Tz):
    u1 = (1.0/3.0)*(Tz - 2*np.sqrt(2)*Ty)
    u2 = (1.0/3.0)*(Tz + np.sqrt(2)*(-1.0*np.sqrt(3)*Tx + Ty))
    u3 = (1.0/3.0)*(Tz + np.sqrt(2)*(np.sqrt(3)*Tx + Ty))

    return u1, u2, u3

# Calculate ball angular position from encoder odometry
def calc_kinematic_conv(psi1,psi2,psi3):
    phix = np.sqrt(2.0/3.0) * (R_W/R_K) * (psi2 - psi3)
    phiy = np.sqrt(2)/3.0 * (R_W/R_K) * (-2 * psi1 + psi2 + psi3)
    phiz = np.sqrt(2)/3.0 * (R_W/R_K) * (psi1 + psi2 + psi3)

    return phix, phiy, phiz

def func_clip(x,lim_lo,lim_hi):
    # A function to clip values that exceed a threshold [lim_lo,lim_hi]
    if x > lim_hi:
        x = lim_hi
    elif x < lim_lo:
        x = lim_lo
    return x

def main():
    # === Data Logging Initialization ===
    trial_num = int(input("Test Number? "))
    filename = f"steering_control_{trial_num}.txt"
    dl = dataLogger(filename)

    # === LCM Initialization ===
    global listening
    global msg
    lc = lcm.LCM("udpm://239.255.76.67:7667?ttl=0")
    subscription = lc.subscribe("MBOT_BALBOT_FEEDBACK", feedback_handler)

    listening = True
    listener_thread = threading.Thread(target=lcm_listener, args=(lc,),
daemon=True)
    listener_thread.start()
    print("Started continuous LCM listener...")

    # Starting encoder reference
    enc_pos_1_start = msg.enc_ticks[0]
    enc_pos_2_start = msg.enc_ticks[1]
    enc_pos_3_start = msg.enc_ticks[2]

    # === Controller Initialization ===
    controller = PS4InputHandler(interface="/dev/input/js0",
                                 connecting_using_ds4drv=False)
    controller_thread = threading.Thread(target=controller.listen, args=(10,))
    controller_thread.daemon = True
    controller_thread.start()
    print("PS4 Controller is active...")

    try:
        command = mbot_motor_pwm_t()

        # === Main Control Loop ===
        print("Starting balance + lean-steering control loop...")
        time.sleep(0.5)

        # Data header
        data_header = [
            "i t_now Tx_total Ty_total Tz_total "
            "psi_1 psi_2 psi_3 dpsi_1 dpsi_2 dpsi_3 "
            "phi_x phi_y phi_z dphi_x dphi_y dphi_z "
            "theta_x theta_y theta_z"
        ]
        dl.appendData(data_header)

        i = 0
        t_start = time.time()
        t_now = 0.0

        enc_pos_1_start = msg.enc_ticks[0]
        enc_pos_2_start = msg.enc_ticks[1]
        enc_pos_3_start = msg.enc_ticks[2]

        # Initialize motor commands
        u1 = u2 = u3 = 0.0

        # Starting IMU orientation (zero reference)
        theta_x_0 = msg.imu_angles_rpy[0]
        theta_y_0 = msg.imu_angles_rpy[1]
        theta_z_0 = msg.imu_angles_rpy[2]

        # Desired lean for balance loop
        theta_d_x = 0.0
```

```
theta_d_y = 0.0

# Balance PID gains and error
Kp_theta = 13.0
Ki_theta = 12.0
Kd_theta = 0.12

err_theta_x_prev = 0.0
err_theta_y_prev = 0.0
int_theta_x = 0.0
int_theta_y = 0.0

# Previous ball angles
phi_x_prev = 0.0
phi_y_prev = 0.0
phi_z_prev = 0.0

# Motor enable flag
motor_on = 0

# D-pad gain tuning state
prev_dpad_up = 0
prev_dpad_down = 0
prev_dpad_right = 0
prev_dpad_left = 0
gain_sel = 0            # 0 = Kp_theta, 1 = Ki_theta, 2 = Kd_theta
last_gain_change_time = 0.0
GAIN_CHANGE_COOLDOWN = 0.12
GAIN_INC = {0: 0.1, 1: 0.01, 2: 0.01}

while True:
    time.sleep(DT)
    t_now = time.time() - t_start
    i += 1

    try:
        # Read sensors
        bt_signals = controller.get_signals()

        # Motor enable
        shoulder_L1 = bt_signals["shoulder_L1"]
        if shoulder_L1 == 1:
            motor_on += 1

        # Joystick and triggers
        js_R_x = bt_signals["js_R_x"]
        js_R_y = bt_signals["js_R_y"]
        trigger_L2 = bt_signals["trigger_L2"]
        trigger_R2 = bt_signals["trigger_R2"]

        # Raw IMU orientation relative to starting pose
        theta_x_raw = msg.imu_angles_rpy[0] - theta_x_0
```

```
theta_y_raw = msg.imu_angles_rpy[1] - theta_y_0
theta_z_raw = msg.imu_angles_rpy[2] - theta_z_0

# Deadzone filtering
theta_x = apply_deadzone(theta_x_raw, IMU_DEADZONE)
theta_y = apply_deadzone(theta_y_raw, IMU_DEADZONE)
theta_z = apply_deadzone(theta_z_raw, IMU_DEADZONE)

# Encoders
enc_pos_1 = msg.enc_ticks[0] - enc_pos_1_start
enc_pos_2 = msg.enc_ticks[1] - enc_pos_2_start
enc_pos_3 = msg.enc_ticks[2] - enc_pos_3_start

enc_dtick_1 = msg.enc_delta_ticks[0]
enc_dtick_2 = msg.enc_delta_ticks[1]
enc_dtick_3 = msg.enc_delta_ticks[2]
enc_dt = msg.enc_delta_time  # [usec]

# Wheel angles
psi_1 = calc_enc2rad(enc_pos_1)
psi_2 = calc_enc2rad(enc_pos_2)
psi_3 = calc_enc2rad(enc_pos_3)

# Wheel angular velocities
if enc_dt > 0:
    dpsi_1 = calc_enc2rad(1e6 * (enc_dtick_1 / enc_dt))
    dpsi_2 = calc_enc2rad(1e6 * (enc_dtick_2 / enc_dt))
    dpsi_3 = calc_enc2rad(1e6 * (enc_dtick_3 / enc_dt))
else:
    dpsi_1 = dpsi_2 = dpsi_3 = 0.0

# Kinematic conversion: wheel angles -> ball angles
phi_x, phi_y, phi_z = calc_kinematic_conv(psi_1, psi_2, psi_3)

# Ball angles -> angular velocities
dphi_x = (phi_x - phi_x_prev) / DT
dphi_y = (phi_y - phi_y_prev) / DT
dphi_z = (phi_z - phi_z_prev) / DT

phi_x_prev = phi_x
phi_y_prev = phi_y
phi_z_prev = phi_z

# Joystick -> lean reference for steering
# Stick right  -> lean right (theta_d_x)
# Stick forward -> lean forward (theta_d_y)
theta_cmd_x = js_R_x * THETA_MAX
theta_cmd_y = -js_R_y * THETA_MAX
# Filter/smooth movement of setpoint
theta_d_x += ALPHA_THETA * (theta_cmd_x - theta_d_x)
theta_d_y += ALPHA_THETA * (theta_cmd_y - theta_d_y)
```

```
# Balance loop (around lean setpoint)
err_theta_x = theta_d_x - theta_x
err_theta_y = theta_d_y - theta_y

int_theta_x += err_theta_x * DT
int_theta_y += err_theta_y * DT

d_err_theta_x = (err_theta_x - err_theta_x_prev) / DT
d_err_theta_y = (err_theta_y - err_theta_y_prev) / DT

# Map lean error to torques
Ty_balance = (Kp_theta * err_theta_x +
              Ki_theta * int_theta_x +
              Kd_theta * d_err_theta_x)

Tx_balance = (Kp_theta * err_theta_y +
              Ki_theta * int_theta_y +
              Kd_theta * d_err_theta_y)

err_theta_x_prev = err_theta_x
err_theta_y_prev = err_theta_y


# Z rotation/steering
Tx_steering = 0.0
Ty_steering = 0.0
Tz_steering = (trigger_R2 - trigger_L2) * Tz_scale

# D-pad gain tuning
dpad_up    = bt_signals["dir_U"]
dpad_down  = bt_signals["dir_D"]
dpad_right = bt_signals["dir_R"]
dpad_left  = bt_signals["dir_L"]

if t_now - last_gain_change_time > GAIN_CHANGE_COOLDOWN:
    if dpad_left == 1 and prev_dpad_left == 0:
        gain_sel = (gain_sel - 1) % 3
        last_gain_change_time = t_now

    elif dpad_right == 1 and prev_dpad_right == 0:
        gain_sel = (gain_sel + 1) % 3
        last_gain_change_time = t_now

    elif dpad_up == 1 and prev_dpad_up == 0:
        inc = GAIN_INC.get(gain_sel, 0.01)
        if gain_sel == 0:
            Kp_theta += inc
        elif gain_sel == 1:
            Ki_theta += inc
        elif gain_sel == 2:
            Kd_theta += inc
        last_gain_change_time = t_now
```

```
                elif dpad_down == 1 and prev_dpad_down == 0:
                    inc = GAIN_INC.get(gain_sel, 0.01)
                    if gain_sel == 0:
                        Kp_theta -= inc
                    elif gain_sel == 1:
                        Ki_theta -= inc
                    elif gain_sel == 2:
                        Kd_theta -= inc
                    last_gain_change_time = t_now

            prev_dpad_up = dpad_up
            prev_dpad_down = dpad_down
            prev_dpad_right = dpad_right
            prev_dpad_left = dpad_left


            # Combine outputs
            Tx_total = Tx_balance + Tx_steering
            Ty_total = Ty_balance + Ty_steering
            Tz_total = Tz_steering

            # Tx,Ty,Tz -> u1,u2,u3
            u1, u2, u3 = calc_torque_conv(Tx_total, Ty_total, Tz_total)

            # Clip to PWM limits
            u1 = func_clip(u1, -PWM_MAX, PWM_MAX)
            u2 = func_clip(u2, -PWM_MAX, PWM_MAX)
            u3 = func_clip(u3, -PWM_MAX, PWM_MAX)

            # Send individual motor commands
            cmd_utime = int(time.time() * 1e6)
            command.utime = cmd_utime
            if (motor_on >= 1):
                command.pwm[0] = -u1
                command.pwm[1] = -u2
                command.pwm[2] = -u3
            lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

            # Store data in data logger
            data = [i, t_now, phi_x, phi_y]
            dl.appendData(data)

            # Print out data in terminal
            # TODO: [IF DESIRED]: Update for what info you want to see in
terminal (note: this is only printed data, not logged!)
            print(
                f"theta_x: {theta_x} | theta_y: {theta_y} | theta_z:
{theta_z} |"
            )

            # Emergency stop
```

```
                emergency_stop = bt_signals["but_tri"]
                if emergency_stop == 1:
                    # Immediately stop all motors
                    command.pwm = [0.0, 0.0, 0.0]
                    command.pwm[0] = 0.0
                    command.pwm[1] = 0.0
                    command.pwm[2] = 0.0

                    lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

                    print("\n" + "!"*80)
                    print("!!! EMERGENCY STOP ACTIVATED !!!")
                    print("!!! Triggered by: TOUCHPAD PRESS !!!")
                    print("!"*80)
                    print("\nAll motors stopped. Exiting control loop
safely...")
                    print("System halted.\n")

                    # Exit the control loop
                    break  # Exits the while True loop

                # Zero the IMU
                shoulder_R1 = bt_signals["shoulder_R1"]
                if shoulder_R1 == 1:
                    theta_x_0 = msg.imu_angles_rpy[0]
                    theta_y_0 = msg.imu_angles_rpy[1]
                    theta_z_0 = msg.imu_angles_rpy[2]


            except KeyError:
                print("Waiting for sensor data...")

    except KeyboardInterrupt:
        print("\nKeyboard interrupt received. Stopping motors...")
        # Emergency stop
        command = mbot_motor_pwm_t()
        command.utime = int(time.time() * 1e6)
        command.pwm[0] = 0.0
        command.pwm[1] = 0.0
        command.pwm[2] = 0.0
        lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

    finally:
        # Save/log data
        print(f"Saving data as {filename}...")
        dl.writeOut()  # Write logged data to the file
        # Stop the listener thread
        listening = False
        print("Stopping LCM listener...")
        listener_thread.join(timeout=1)  # Wait up to 1 second for thread to
finish
        # Stop Bluetooth thread
        controller_thread.join(timeout=1)  # Wait up to 1 second for thread to
finish
        controller.on_options_press()
        # Stop motors
        print("Shutting down motors...\n")
        command = mbot_motor_pwm_t()
        command.utime = int(time.time() * 1e6)
        command.pwm[0] = 0.0
        command.pwm[1] = 0.0
        command.pwm[2] = 0.0
        lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())

if __name__ == "__main__":
    main()
```