Ball-Bot Control Design and Competition Performance

Jason Hernandez Lopez • Alexandru Otlacan

ROB 311: How to Build Robots (and Make Them Move) — University of Michigan —
April 2026

Abstract. We built a three-wheel ball-bot that balances on a basketball using IMU feedback and encoder data. The final controller was a PD balance controller with filtering, PWM limits, motor deadband compensation, and rescue gain scaling. The robot balanced for about 1 minute and 40 seconds in our best test, but drift made smooth driving unreliable. In the competition, we scored 25.33 points by focusing on balance first and then using a forward-reset strategy for the driving events.

1. Introduction

The goal of this project was to build a ball-bot that could balance on a basketball and drive simultaneously. Our robot used three motorized omniwheels to push on the ball from different directions. Because the robot is unstable, even a small lean error can quickly turn into a fall if the motors do not react fast enough.

Our final system used IMU lean angles for balance and encoder data for wheel and ball motion estimates. The controller calculated efforts and converted them into three motor commands, clipped the commands for safety, and sent PWM values to the motors. We measured performance using balance time, IMU and encoder plots, and controller-effort plots.

The main result was that the robot could balance near upright, but smooth driving was still unreliable. The biggest issue was drift. The robot sometimes moved away from center even when the IMU looked close to upright. This report focuses on the controller design, how we tuned it, what the data showed, and what we would fix next.

2. Methodology

2.1 System and Data Flow

The Python controller ran at 100 Hz. The robot sent feedback to Python through LCM. The feedback included IMU angles, encoder ticks, encoder tick changes, encoder timing, and voltage values. Python used this data to calculate the controller output, then sent PWM commands back to the robot through LCM. At startup, the code saved the initial IMU angles and encoder values so later measurements were relative to the starting position. The full controller architecture diagram is included in Appendix A.

The IMU was the main sensor for balance because it directly measured body lean. The encoders were used to estimate wheel motion, ball motion, and the path of the robot. This split mattered because the IMU was better for fast balance correction, while the encoders were more useful for odometry and checking how the robot moved over time.

2.2 Balance and Steering Control Loops

The balance loop tried to keep the robot upright. The desired lean angle was zero, with small command offsets added when we wanted the robot to move. The controller compared the desired lean to the measured IMU lean and created T_x and T_y efforts. These efforts were then converted into three motor commands.

The final controller was written like a PID controller, but the integral gain was zero. This made the final balance controller act like a PD controller. The proportional term corrected lean, and the derivative term slowed down fast lean changes. We decided against an I value doe the accumulation of error and through testing we did not see much success with adding any I term. The final gains were

$$K_P = 8.25, \qquad K_I = 0.00, \qquad K_D = 0.08. \tag{1}$$

The balance equations were

$$e_x = \theta_{x,ref} - \theta_x, \qquad e_y = \theta_{y,ref} - \theta_y, \tag{2}$$

$$T_x = K_{P,eff} e_x + K_I \int e_x \, dt + K_{D,eff} \frac{de_x}{dt}, \quad T_y = -\left(K_{P,eff} e_y + K_I \int e_y \, dt + K_{D,eff} \frac{de_y}{dt} \right). \tag{3}$$

The negative sign on T_y matched the coordinate direction of our robot.

The steering loop was simpler than the balance loop. During testing, the PS4 controller created small setpoint offsets in the x and y directions. On competition day, the controllers did not connect, so we used keyboard commands to move forward. These steering commands did not fully close the loop on position. Instead, they changed the commanded direction while the balance loop kept trying to hold the robot upright. The encoder-based path was then used to evaluate how well the robot moved.

2.3 Controller Logic, Code Features, and Tuning

We added several logic features to make the robot safer and easier to tune. The IMU angle used a 0.1° deadzone and a low-pass filter with $\alpha = 0.85$ to reduce small noise near upright. Motor commands were clipped to ± 0.60 PWM, and small nonzero commands were raised to 0.12 PWM to overcome static friction of the motor. Rescue gain scaling increased K_P and K_D when the robot leaned farther from upright, starting at 0.9° and reaching full scale near 5.0°. The code also used startup IMU zeroing, LED lean feedback, PS4/keyboard command input, and CSV logging. These features made testing easier because we could re-zero the robot, see lean magnitude visually, command motion, and review the data after each run.

Table 1: Key final controller settings.

Setting	Purpose	Final value
K_P	Correct lean error	8.25
K_I	Integral correction	0.00
K_D	Damping	0.08
IMU LPF α	Reduce sensor noise	0.85
PWM limit	Limit motor command	0.60
Motor deadband	Overcome stiction	0.12
Rescue scale	Boost gains at larger lean	2.1 max

We tuned K_P first. If K_P was too low, the robot reacted too slowly and fell. If K_P was too high, the robot pushed too hard and overshot. We then tuned K_D to reduce fast lean changes. We kept $K_I = 0$ because integral action did not help the robot stay upright and made recovery less predictable. The hardest tuning problem was drift. We tried stricter zeroing, filter changes, deadzone changes, gain changes, small offsets, and weight shifts. These helped, but they did not fully remove the drift.

3. Results and Discussion

3.1 Balance Performance

Figure 1 shows the final x-axis balance response from a 20-second test. This figure matches the required balance-controller format: lean angle on top, individual P, I, and D efforts in the middle, and total controller effort on the bottom. The first large correction happens at startup. After that, θ_x stays mostly near upright, with small oscillations that grow and shrink over time.

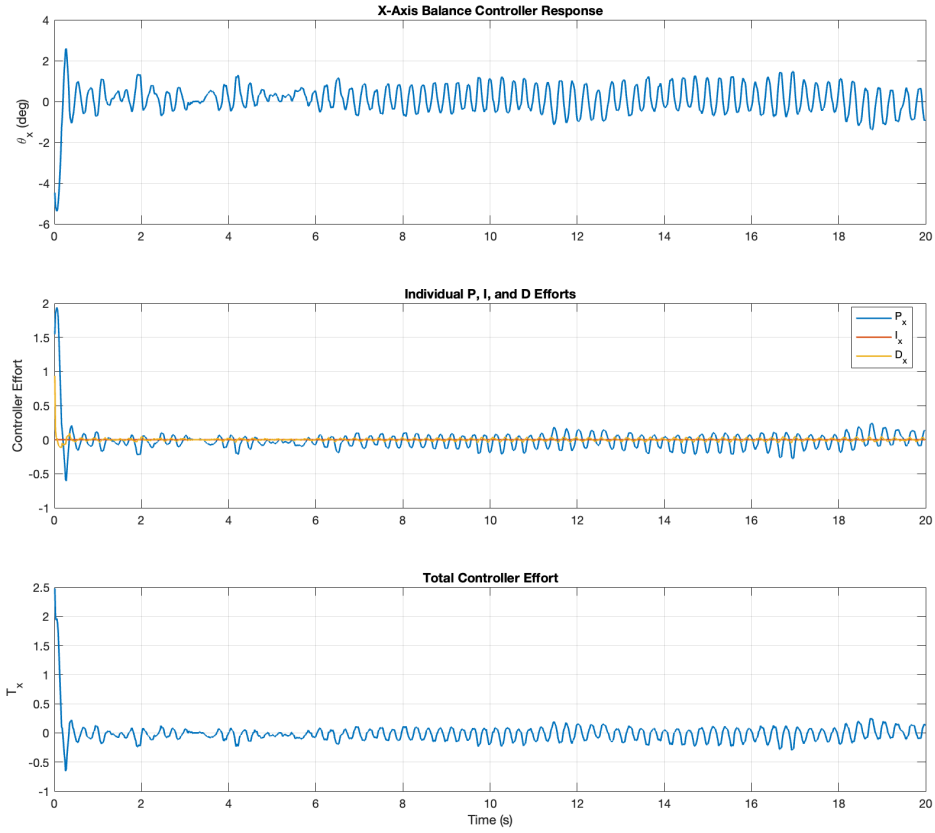

Figure 1: Final x-axis balance-controller response over a 20-second test.

The plot shows that P_x did most of the work after startup. D_x was most important during fast changes, especially during the initial correction and later oscillations. I_x stayed near zero because $K_I = 0$, which confirms that our final controller was effectively PD. The total effort stayed bounded after the first transient, so the robot remained stable.

Our best unsupported balance time was about 1 minute and 40 seconds, and this was during the competition. This showed that the IMU feedback, PD controller, rescue scaling, and motor conversion worked near upright. The weakness was that the stable region was narrow. Once drift moved the robot too far from center, the controller usually could not recover fast enough.

3.2 IMU and Encoder Spin-Speed Comparison

Figure 2 compares IMU yaw with encoder-derived yaw from the ground spin-speed test. We measured the IMU yaw rate directly from the robot body. We calculated the encoder yaw estimate by converting encoder ticks to wheel motion and then using the kinematic conversion to estimate ballbot rotation. The plotted points were then fit with lines to compare the trends. The robot was

placed flat on the ground and spun at several different torque commands while we logged IMU and encoder data.

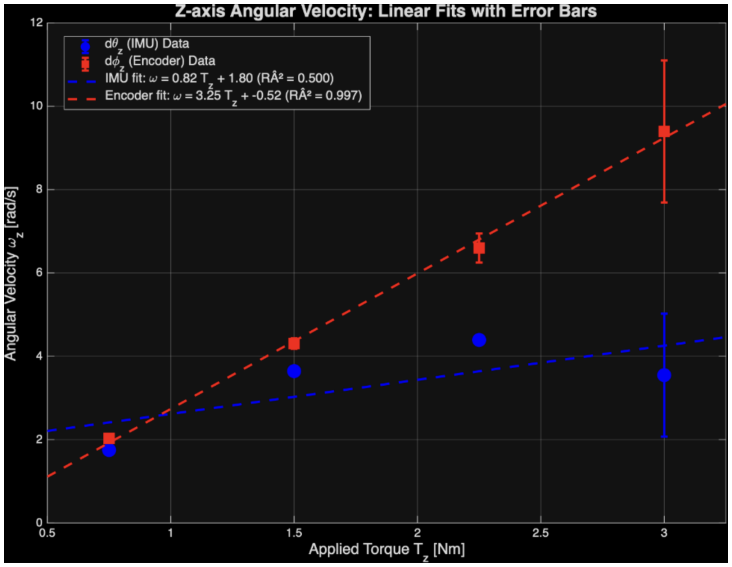

Figure 2: IMU yaw compared with encoder-derived yaw.

The IMU and encoder trends did not align perfectly. This makes sense because the IMU measures body rotation directly, while the encoder estimate depends on wheel contact, rolling assumptions, timing, and the kinematic model. Slip between the omniwheels and the ball also changes the encoder estimate. This helped explain why balance worked better than driving: the robot could look upright from the IMU, while the ball motion still drifted in a way the controller did not fully correct.

3.3 Steering, Square Path, and Competition

Figure 3 shows the encoder-derived path for the square task. The goal was to navigate a square path of at least 4 ft by 4 ft and compare the encoder-converted path to the reference square. Our plotted path shows that the robot performed decently, but corners were rounded and sides were not straight. Several factors contributed to this. First, the turns were controlled by IMU yaw threshold alone with no position correction, so any drift during a turn carried directly into the next side. Additionally, wheel slip between the omniwheels and the basketball means the encoder odometry does not perfectly reflect true ball displacement. The path also shows a noticeable leftward bias, which likely came from a small asymmetry in wheel contact or an unlevel starting position. We also had to periodically support the ballbot during its motion to ensure it remained balanced.

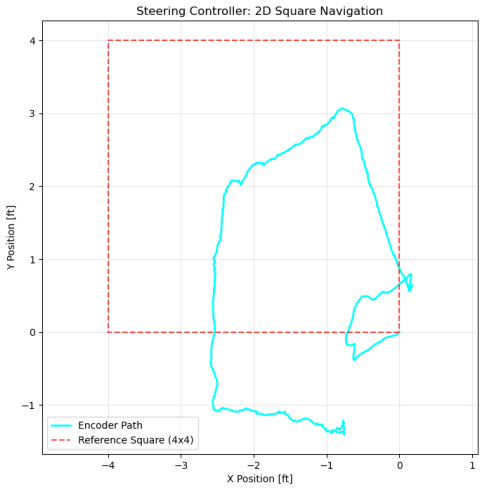

Figure 3: Encoder-derived square-path odometry.

Because smooth driving was not reliable, our competition strategy was to move forward, reset after falls, and continue. This was not ideal, but it allowed us to score in every event. Table 2 shows the final competition scores.

Table 2: Competition score summary for Team A05.

Event	Score	Summary
Balance	6.00	Balanced, but drift limited consistency
Platform	8.33	Made partial progress and earned points
Square	6.00	Used forward-reset strategy around the square
Race	5.00	Advanced despite repeated falls
Total	25.33	Scored in every event

The competition performance showed the same pattern during testing. The robot could balance near upright, but it became much less reliable when it had to move. Given more time, the biggest improvement would be to reduce drift first, then add better closed-loop steering. Better wheel-ball contact, cleaner IMU zeroing, and longer logs that compare IMU and encoder motion during balance would make our next steps much clearer.

4. Conclusion and Future Work

We built a three-wheel ball-bot that could balance on a basketball using IMU feedback, encoder data, and a PD-style controller. The best balance run lasted about 1 minute and 40 seconds, which showed that the core controller worked. The main limitation was drift. Drift made driving unreliable and caused repeated falls during the competition tasks.

The next step should be to fix drift before improving/working on driving. We would keep logging the individual P, I, and D terms, compare IMU and encoder motion during longer balance tests, tune the rescue behavior more carefully, and improve wheel-ball contact. Once the robot stays centered more reliably, smoother closed-loop driving should be much easier. We were also thinking of adding more weight near the center of the robot to attain a better COM.

A. System Reference Figures and Tables

Figure 4 shows the full controller flow used in the final code.

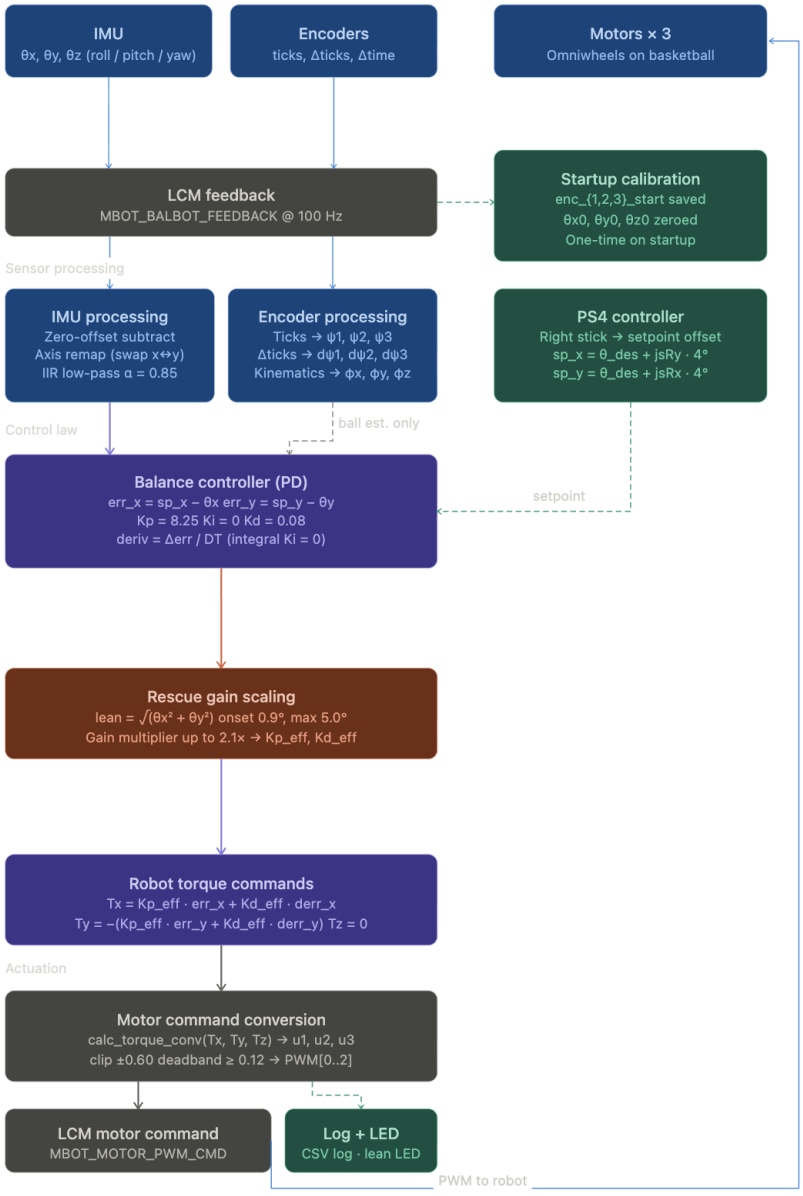

Figure 4: Final controller architecture. IMU feedback drives balance, encoder data estimates ball motion, and Python converts the controller output into three motor PWM commands.

Table 3: Main system components.

Component	Purpose	How we used it
IMU	Lean sensing	Measured ϑ_x, ϑ_y, and ϑ_z
Encoders	Wheel motion	Estimated ball motion and odometry
Python code	Controller	Ran balance loop at 100 Hz
LCM	Communication	Sent feedback and PWM messages
LED	Status display	Showed startup and lean magnitude
Computer input	Competition commands	Used keyboard inputs when controllers did not connect

B. Annotated Ball-Bot Image

Figure 5 shows the final ball-bot layout.

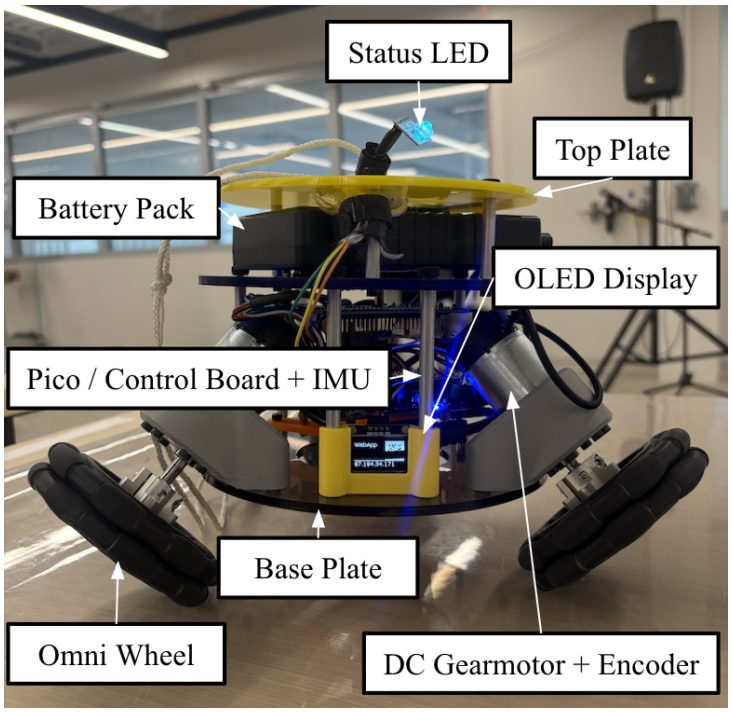

Figure 5: Final ball-bot assembly with the main mechanical and electrical subsystems shown: frame/top plate, electronics stack, battery, motors, and omniwheels.

C. Final Python Code

The final Python file is included below.

```
"""
ROB 311 - WN2026
Author: Prof. Greg Formosa
University of Michigan

WARNING: be sure motors are secured and in a safe orientation to spin before starting!

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!
Will store data as a "ballbot_control_<#>.txt" tab-delimited file that can be parsed in Matlab.

Press Ctrl+C to stop script at any time.

PWM_MAX should be [0.05,0.99] and PWM_PERIOD > 1.5 seconds.

LCM data type for ROB 311 ball-bots:
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 DataLogger import dataLogger
from ps4_controller_api import PS4InputHandler
from blinkm_i2c import stop_script, set_color_now, fade_to_color, get_current_color

# Constants for the control loop - should not change!
FREQ = 100  # Frequency of control loop [Hz]
DT = 1 / FREQ  # Time step for each iteration [sec]
PWM_MAX = 0.6  # Max motor signal for accel/decel of motor test (keep between [0.05,0.99])
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 = 0.120 # Radius of basketball [m]

# 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}

# === LCM methods for communication with Pico (no changes needed) ===
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)

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

# === Student-defined methods for control ===
def calc_enc2rad(ticks):
    # TODO [LAB-08]: Calculate the angle (radians) of the wheel given the motor encoder ticks
    # done
    rad = 2 * np.pi * ticks / (64*70)
    return rad

def calc_torque_conv(Tx,Ty,Tz):
    # TODO [LAB-07]: Calculate desired motor torques T1,T2,T3 from given Tx,Ty,Tz
    # Done
    T1 = 1/3 * (Tz - ((2*Ty)/(np.pi/4)))
    T2 = 1/3 * (Tz + ((1/(np.pi/4))*(-np.sqrt(3)*Tx + Ty)))
    T3 = 1/3 * (Tz + ((1/(np.pi/4))*(np.sqrt(3)*Tx + Ty)))
    return T1, T2, T3

def calc_kinematic_conv(psi1,psi2,psi3):
    # TODO [LAB-08]: Calculate ball angular position from encoder odometry
    # done
    phix = np.sqrt(2/3) * (R_W/R_K) * (psi2 - psi3)
    phiy = np.sqrt(2)/3 * (R_W/R_K) * (-2*psi1 + psi2 + psi3)
    phiz = np.sqrt(2)/3 * (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 func_deadzone(x,limit):
    # A function to create a deadzone, to zero values below a threshold [-limit,limit]
    if abs(x) < limit:
        x = 0
    return x

def func_motor_deadband(u, min_pwm=0.12):
    if abs(u) < 1e-4:
        return 0.0
    return np.sign(u) * max(abs(u), min_pwm)

IMU_DEADZONE = np.radians(0.1)
LPF_ALPHA = 0.85
def calc_iir_lpf(x_new, y_prev, alpha):
    return alpha * x_new + (1 - alpha) * y_prev

# === Main loop ===
def main():
    # === Data Logging Initialization (no changes needed) ===
    # Prompt user for trial number and create a data logger
    trial_num = int(input("Test Number? "))
    filename = f"pid_copy_{trial_num}.txt"
    dl = dataLogger(filename)

    # === Controller Initialization (no changes needed) ===
    # 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()
    controller_status = 0 # Comment this line if you want to disable controller input and just run open-loop
    print("PS4 Controller is active...")

    # === LCM Messaging Initialization (no changes needed) ===
    # 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...")

    try:
        # === Main Control Loop ===
        command = mbot_motor_pwm_t()
        print("Starting steering control loop...")
        # Startup: blink red 5 times over 5 seconds, zero IMU after first blink (within first second)
        print("Startup: blinking red 5x, zeroing IMU after first blink...")
        for _blink in range(5):
            set_color_now(255, 0, 0)
            time.sleep(0.5)
            set_color_now(0, 0, 0)
            time.sleep(0.5)
            if _blink == 0:
                enc_1_start = msg.enc_ticks[0]
                enc_2_start = msg.enc_ticks[1]
                enc_3_start = msg.enc_ticks[2]
                theta_x_0 = msg.imu_angles_rpy[0]
                theta_y_0 = msg.imu_angles_rpy[1]
                theta_z_0 = msg.imu_angles_rpy[2]
                print("IMU zeroed.")
        stop_script()
        data = ["i t_now Tx Ty Tz u1 u2 u3 theta_x theta_y theta_z psi_1 psi_2 psi_3 dpsi_1 dpsi_2 dpsi_3 phi_x phi_y phi_z dphi_x dphi_y dphi_z"]
        dl.appendData(data)
        i = 0
        t_start = time.time()
        t_now = 0
        u1 = 0
        u2 = 0
        u3 = 0
        desired_theta = 0.0
        Kp = 8.25
        Ki = 0.00
        Kd = 0.08

        prev_error_x, prev_error_y = 0.0, 0.0
        integral_x, integral_y = 0.0, 0.0

        theta_x_filt = 0.0
        theta_y_filt = 0.0
        theta_z_filt = 0.0
        error_x_filt = 0.0
        error_y_filt = 0.0
        ERROR_LPF_ALPHA = 0.9
        SETPOINT_MAX_RAD = np.radians(4.0)
        RESCUE_START_DEG = 0.9
        RESCUE_END_DEG   = 5.0
        RESCUE_MAX_GAIN  = 2.1

        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()
                # TODO [IF DESIRED]: choose which BT buttons you want data from, see "test_BT.py" script from Lab-05 for help
                trigger_R2 = bt_signals["trigger_R2"]
                trigger_L2 = bt_signals["trigger_L2"]
                js_R_y = bt_signals["js_R_y"]
                js_R_x = bt_signals["js_R_x"]


                # Pull sensor data
                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

                theta_x_temp = theta_x_raw
                theta_x_raw = -theta_y_raw
                theta_y_raw = theta_x_temp

                theta_x_filt = calc_iir_lpf(func_deadzone(theta_x_raw, IMU_DEADZONE), theta_x_filt, LPF_ALPHA)
                theta_y_filt = calc_iir_lpf(func_deadzone(theta_y_raw, IMU_DEADZONE), theta_y_filt, LPF_ALPHA)
                theta_z_filt = calc_iir_lpf(func_deadzone(theta_z_raw, IMU_DEADZONE), theta_z_filt, LPF_ALPHA)

                theta_x = theta_x_filt
                theta_y = theta_y_filt
                theta_z = theta_z_filt


                lean_magnitude = np.sqrt(theta_x**2 + theta_y**2)
                max_lean = np.radians(15)
                brightness = int(func_clip(lean_magnitude / max_lean, 0, 1) * 255)
                set_color_now(brightness, 0, 255 - brightness)

                #comment out as needed

                enc_1 = msg.enc_ticks[0] - enc_1_start
                enc_2 = msg.enc_ticks[1] - enc_2_start
                enc_3 = msg.enc_ticks[2] - enc_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

                # Calculate motor angles from encoder ticks
                # TODO [LAB-08]: Call method to calculate motor angles & speeds from measured encoder values [LAB-07]: Implement placeholders from Notion
                # Done
                psi_1 = calc_enc2rad(enc_1)
                psi_2 = calc_enc2rad(enc_2)
                psi_3 = calc_enc2rad(enc_3)
                enc_dt_sec = enc_dt / 1e6
                dpsi_1 = calc_enc2rad(enc_dtick_1) / enc_dt_sec if enc_dt_sec > 0 else 0
                dpsi_2 = calc_enc2rad(enc_dtick_2) / enc_dt_sec if enc_dt_sec > 0 else 0
                dpsi_3 = calc_enc2rad(enc_dtick_3) / enc_dt_sec if enc_dt_sec > 0 else 0

                # Calculate ball's roll and translation through kinematic conversions of wheel data
                phi_x, phi_y, phi_z = calc_kinematic_conv(psi_1, psi_2, psi_3)
                dphi_x, dphi_y, dphi_z = calc_kinematic_conv(dpsi_1, dpsi_2, dpsi_3)

                # Set x-y-z bot commands
                # TODO [LAB-07 & LAB-08]: Choose how Tx,Ty,Tz are set
                # Done

```

```
                #automatic control

                        # print("Test complete (12s reached). Stopping motors...")
                        # command.utime = int(time.time() * 1e6)
                        # command.pwm[:] = [0.0, 0.0, 0.0] # remember to set the pwm c
                        # lc.publish("MBOT_MOTOR_PWM_CMD", command.encode())
                        # break

                if controller_status == 0:
                    setpoint_x = desired_theta + js_R_y * SETPOINT_MAX_RAD
                    setpoint_y = desired_theta + js_R_x * SETPOINT_MAX_RAD

                    error_x = setpoint_x - theta_x
                    error_y = setpoint_y - theta_y

                    derivative_x = (error_x - prev_error_x) / DT
                    derivative_y = (error_y - prev_error_y) / DT

                    error_x_filt = calc_iir_lpf(error_x, error_x_filt, ERROR_LPF_ALPHA)
                    error_y_filt = calc_iir_lpf(error_y, error_y_filt, ERROR_LPF_ALPHA)
                    integral_x += error_x_filt * DT
                    integral_y += error_y_filt * DT

                    lean_mag = np.sqrt(theta_x**2 + theta_y**2)
                    rescue_t = func_clip(
                        (lean_mag - np.radians(RESCUE_START_DEG)) /
                        (np.radians(RESCUE_END_DEG) - np.radians(RESCUE_START_DEG)),
                        0.0, 1.0)
                    rescue_scale = 1.0 + rescue_t * (RESCUE_MAX_GAIN - 1.0)
                    Kp_eff = Kp * rescue_scale
                    Kd_eff = Kd * rescue_scale

                    Tx = Kp_eff * error_x + Ki * integral_x + Kd_eff * derivative_x
                    Ty = -(Kp_eff * error_y + Ki * integral_y + Kd_eff * derivative_y)
                    Tz = 0

                    prev_error_x = error_x
                    prev_error_y = error_y


                if controller_status == 1: #joystick code
                    Tx = js_R_y
                    Ty = js_R_x
                    Tz = trigger_R2 - trigger_L2







                # Calculate motor effort/commands from desired Tx,Ty,Tz motion
                # TODO [LAB-07]: Call method to calculate motor commands (u1,u2,u3) from axis torques
                ↪  (Tx,Ty,Tz)
                # Done
                u1, u2, u3 = calc_torque_conv(Tx,Ty,Tz)


                # Send individual motor commands
                cmd_utime = int(time.time() * 1e6)
                command.utime = cmd_utime
                u1 = func_motor_deadband(func_clip(u1,-PWM_MAX,PWM_MAX))
                u2 = func_motor_deadband(func_clip(u2,-PWM_MAX,PWM_MAX))
                u3 = func_motor_deadband(func_clip(u3,-PWM_MAX,PWM_MAX))
                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
                # TODO [IF DESIRED]: Update variables to match data header names for logging
```

```
                data = [i, t_now, Tx, Ty, Tz, u1, u2, u3, theta_x, theta_y, theta_z, psi_1, psi_2, psi_3,
                ↪  dpsi_1, dpsi_2, dpsi_3, phi_x, phi_y, phi_z, dphi_x, dphi_y, dphi_z]
                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"Time: {t_now:.3f}s | Tx: {Tx:.2f}, Ty: {Ty:.2f}, Tz: {Tz:.2f} | "
                #     f"u1: {u1:.2f}, u2: {u2:.2f}, u3: {u3:.2f} | "
                #     f"Theta X: {theta_x:.2f}, Theta Y: {theta_y:.2f}, Theta Z: {theta_z:.2f} | "
                #     f"Psi 1: {psi_1:.1f}, Psi 2: {psi_2:.1f}, Psi 3: {psi_3:.1f} | "
                #     f"dPsi 1: {dpsi_1:.2f}, dPsi 2: {dpsi_2:.2f}, dPsi 3: {dpsi_3:.2f} | "
                # )

            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()
        listening = False
        print("Stopping LCM listener...")
        listener_thread.join(timeout=1)
        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()
```