Sensor-Integrated Gimbal Stabilization with Dual-Joystick Control

Farris M. Hamid, *Mechanical Engineering*
University of Missouri, Kansas City

Ali Fakhar, *Mechanical Engineering*
University of Missouri, Kansas City

ABSTRACT

To address the challenge of stabilizing motion-based systems while allowing manual control, a dual-joystick gimbal setup is developed. The system enables users to adjust roll and pitch angles smoothly while maintaining overall stability, which is essential for tasks like camera tracking, robotic movement, and remote-controlled platforms.

By integrating two joysticks, the design supports intuitive multi-axis control, giving users direct input over orientation without sacrificing system balance. The concept of joystick control is based on research involving motion-matching controllers used in drone systems. These studies showed that traditional joystick setups can be difficult to operate due to poor alignment between user input and system response. By improving the coordination between joystick signals and gimbal motion, the system aims to offer a more intuitive and responsive control experience [2].

To enhance stabilization, an Inertial Measurement Unit (IMU) was implemented with Kalman Filtering to best correct for sensor noise from the IMU. The filter was implemented in MATLAB and compared with the Complementary Filter method. Filter constants were tuned to reduce sensor noise and improve angle estimation, resulting in smoother and more reliable motion tracking.

Key Indexes: *First-Person View (FPV), MPU6050, Arduino, Joystick*

I. INTRODUCTION

A. Background

THE objective of this project, as outlined in the *Abstract* section, is to design a compact, low-cost, and real-time self-stabilizing gimbal system that integrates IMU with the extra functionality of a manual joystick control input using Arduino IDE.

In recent years, motion-controlled systems have become an essential part of robotics and mechatronics, enabling more natural and precise ways to control devices such as drones, robotic arms, and camera gimbals [1]. Traditional joystick-based control methods, while effective, often require the user to mentally translate two-dimensional joystick motion into three-dimensional system movement [2]. This can reduce intuitiveness and increase operator workload, particularly for users with limited experience in multi-axis control [1]. Developing systems that align physical user movement with the corresponding mechanical response has therefore become a growing area of research in intuitive control interfaces.

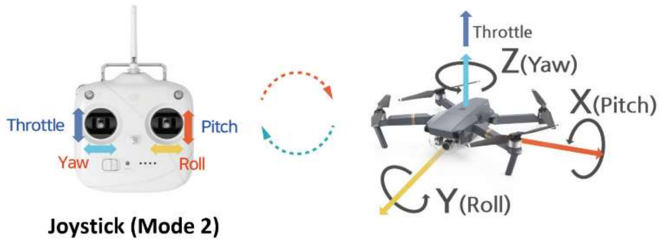


Fig. 1. Typical dual-joystick control layout for drone or gimbal systems. [2]

Recent studies explored how motion-matching controllers improve usability by linking the user's hand orientation directly to the device's motion [2]. Their research compared conventional joystick controllers with a motion-matching prototype and found that users achieved higher success rates, shorter task times, and fewer collisions [2]. These findings highlight that mapping the physical motion of the operator to the controlled system reduces reference-frame misalignment and enhances user intuition.

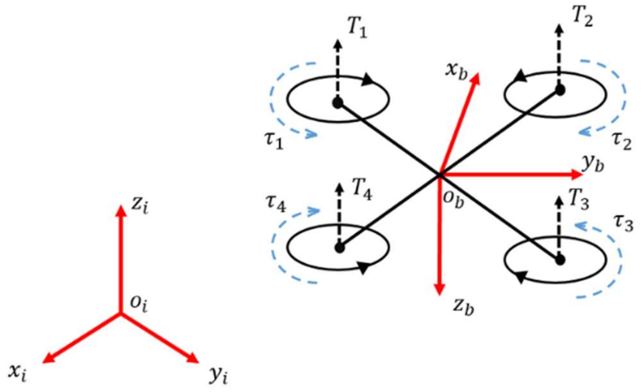


Fig. 2. Illustration of reference-frame misalignment [3]

One key problem identified in the study is the reference-frame misalignment, such as the illustration in Fig. 2., where the user's intended motion and the device's actual motion differ due to perspective changes [2]. For example, when a drone rotates, the pilot must mentally adjust directions to align the joystick inputs with the drone's new orientation [3]. This cognitive effort often leads to control errors [3]. By contrast, a motion-matching controller directly couples the controller's orientation to the drone's orientation, allowing for more natural, body-based control inputs [1].

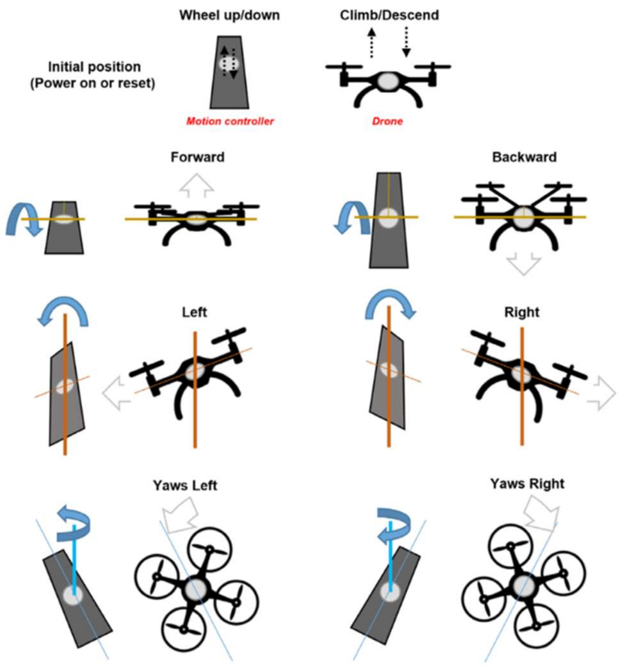


Fig. 3. Motion-matching control concept. [2]

Inspired by this concept, the project aims to develop a gimbal control system that integrates two joysticks for added functionality and precision. The gimbal provides stable three-axis motion for pitch, roll, and yaw control, while the joysticks offer additional inputs for speed adjustment and directional fine-tuning. This hybrid design allows the user to experience both intuitive motion-matching control and accurate manual input, and therefore, can be a flexible prototype for various mechatronic applications like a camera stabilizer on drones, unmanned vehicle control, etc.

During the experimental tests, the control system is designed using Arduino Internal Development Environment (IDE). The signal readings were taken and postprocessed in MATLAB and the results found that the hand motion sensing is generally more stable and has smoother motion transitions compared to controlling the object using the two joysticks. The percent error when controlling the motion tilts using the IMU hand motion sensing has an error range from ~9% to ~20%. Whereas, percent error when controlling the motion tilts using the joystick has an error range from ~10% up to ~99% due to calibration errors arising due to sensor noise. Therefore, the experiment proves why using the hand motion sensing is generally more effective when being integrated with the joystick controller.

B. Design Challenges and Approach

Key challenges include real-time orientation estimation, servo response, mode switching logic, and mechanical alignment. Therefore, the integration of the X-Y joystick control and compromising the Gimbal stabilization adds complexity to both hardware and software aspects of the final design. Another factor is the physical placement of motors, joysticks, and center of mass of all the components must be balanced for minimal disturbances due to the misalignment behaviors.

C. Theories on IMU and Kalman Filtering

The inertial measurement unit (IMU) is a key part of many navigation systems [4]. It provides motion data that does not rely on external signals, which makes it useful when satellite signals are weak or lost. The MPU6050 is a 6-axis inertial measurement unit (IMU) that combines a 3-axis accelerometer and a 3-axis gyroscope. It detects linear acceleration and angular velocity and usually passes through a complimentary filter to get accurate readings for measuring orientations. The complementary filter is an effective algorithm used to estimate orientation by combining accelerometer and gyroscope data. The accelerometer provides long-term stability by measuring tilt angles through trigonometric relationships. The gyroscope measures angular velocity, which can be integrated over time to track short-term changes in orientation, but it loses accuracy due to drift. The complementary filter blends these two signals, weighting the gyroscope for short-term accuracy and the accelerometer for long-term correction. This ensures that the estimated pitch and roll angles remain both stable and responsive. Mathematically, the complementary filter can be expressed as:

$$\theta_{comp}(t) = \alpha * \left(\theta_{comp}(t-1) + \omega\Delta t\right) + (1-\alpha)\theta_{acc}(t)$$

In the gimbal system code, this behavior is implemented as:

```
compAngleX = (ALPHA * (compAngleX + gyroRate * dt)) +
((1.0 - ALPHA) * accXangle);

compAngleY = (ALPHA * (compAngleY + gyroRate * dt)) +
((1.0 - ALPHA) * accYangle);
```

On the other hand, the Kalman Filter is a recursive estimation algorithm that provides an optimal estimate of a system's state in the presence of noise. In the context of gimbal stabilization, the filter is used to estimate roll and pitch angles by combining gyroscope and accelerometer measurements. The gyroscope provides short-term accuracy but suffers from drift, while the accelerometer provides long-term stability but is sensitive to noise. The Kalman Filter fuses these two sources to achieve a reliable orientation estimate.

The prediction step uses the gyroscope rate to estimate the new angle. The state update equation is expressed as:

$$\hat{x} = \hat{x}_{k-1} + \Delta t * (\omega - b_{k-1})$$

Where $\hat{x}$ is the predicted angle, $\hat{x}_{k-1}$is the previous estimate, Δt is the sampling interval, ω is the gyroscope rate, and b_{k-1} is the estimated bias. In the MATLAB code, this is implemented as:

```
angleX = angleX + dt * (rateX - biasX);
angleY = angleY + dt * (rateY - biasY);
```

The error covariance matrix is also updated to reflect the uncertainty in the prediction:

$$P = P_{k-1} + Q$$

where Q represents the process noise covariance. This is written in MATLAB as:

```
P_X = P_X + dt * ([Q_angle -P_X(1,2); -
P_X(2,1) Q_bias]);

P_Y = P_Y + dt * ([Q_angle -P_Y(1,2); -
P_Y(2,1) Q_bias]);
```

When a new sensor measurement (accelerometer angle) is available, the filter computes the difference between the measurement and the prediction:

$$y = z_k - x_k$$

In MATALB code, it is computed as:

```
y = allData(k,4) - angleX;   % for roll
y = allData(k,5) - angleY;   % for pitch
```

The Kalman Gain determines how much weight to give the measurement versus the prediction, where R is the measurement noise covariance:

$$K = \frac{P_{k-1}}{P_{k-1} + R}$$

In MATALB code, it is computed as:

```
S = P_X(1,1) + R_measure;
K = P_X(:,1) / S;
```

The predicted state is corrected using the Kalman Gain and the innovation:

$$\hat{x}_k = \hat{x}_{k-1} * Ky$$

In MATALB code, it is computed as:

```
angleX = angleX + K(1) * y;
biasX  = biasX  + K(2) * y;
```

Finally, the error covariance is updated to reflect the improved estimate:

$$P_k = (1 - K) * P_{k-1}$$

In MATALB code, it is computed as:

```
P_X = P_X - K * P_X(1,:);
```

Overall, the MATLAB postprocess follows the Kalman Filter framework: (1) predict the angle from gyroscope data, (2) compute the accelerometer measurements, (3) calculate Kalman Gain to (4) update the angle estimates. This approach demonstrates how sensor fusion can be effectively applied in motion control systems.

II. Materials and Methods

A. Components

The gimbal prototype is constructed using a combination of electronic components and a custom mechanical frame. At its core, the system employs the **ATmega328P microcontroller**, implemented through an Arduino Nano board, which serves as the primary control unit. Motion sensing and orientation feedback are provided by the **MPU6050 inertial measurement unit (IMU)**, enabling precise detection of angular movement.

Actuation is achieved through three **MG996R servo motors**, which deliver the necessary torque and responsiveness for stabilizing the gimbal structure. To enhance user interaction, a **KY-023 analog joystick** is incorporated, offering additional manual control functionality. Power regulation is managed by a **buck converter**, which ensures a stable 5V supply to the electronic components.

Finally, the mechanical structure of the prototype is realized through a **3D-printed gimbal frame**, designed using computer-aided design (CAD) software such as SolidWorks, as illustrated in Fig. 2. This frame provides both lightweight construction and structural support for the integrated components.

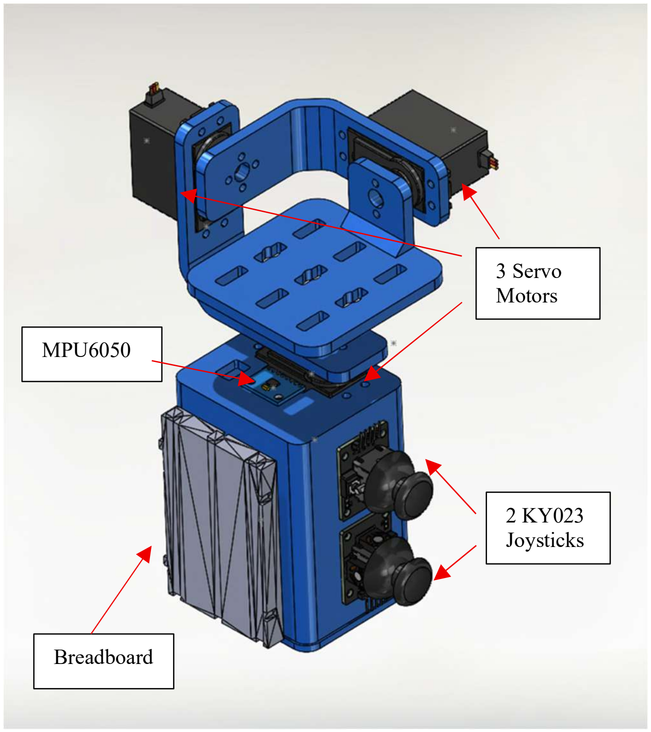


Fig. 2. Computer Aided Design of a Gimbal Prototype

B. Methodology

Fig. 3. presents a methodology mind map outlining two control strategies for a gimbal stabilization prototype. Method 1 uses an IMU sensor to detect motion, which is processed by a control algorithm and sent to a motor driver to adjust the gimbal position automatically. Method 2 relies on a joystick to manually guide the gimbal, but also incorporates IMU feedback to support motion tracking and the use of the IMU feedback is helpful for determining the angle trajectories for the gimbal. In both methods, the control algorithm plays a central role in interpreting input signals and driving the motors to maintain or adjust orientation.

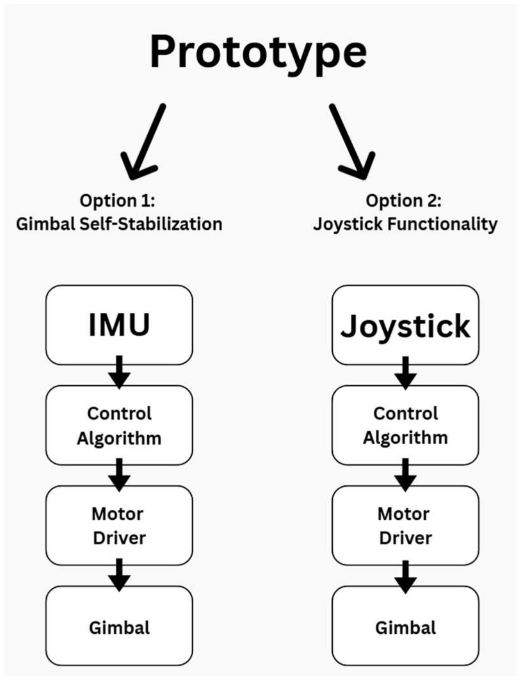


Fig. 3. Methodology Mind map

C. Communication Protocol and Pin Mapping

The system uses an Arduino Nano to read IMU data and compute pitch and roll angles. These angles are mapped to servo positions for stabilization. Furthermore, the stabilization relies on real-time sensor feedback, where the IMU comes in handy. The MPU6050 IMU communicates with the microcontroller via the I^2C protocol, that allows data exchange using only two wires via (SDA and SCL). This compact interface is ideal for embedded systems with limited input/output resources and lets the microcontroller to continuously poll the IMU for accelerometer and gyroscope data.

On the other hand, the KY-023 joystick module operates by translating physical displacement into analog voltage signals, which are then interpreted by the microcontroller using the ADC protocol. The joystick contains two potentiometers, one for X-axis (VRx) and one for Y-axis (VRy). As the user moves the joystick, the voltage across each potentiometer changes proportionally. These analog voltages are transmitted to the microcontroller's ADC pins, where they are sampled and converted into digital values. For instance, on an Arduino Nano, the ADC converts a voltage range of 0–5V into a 10-bit digital value ranging from 0 to 1023, which would allow the system to quantify joystick position and map it to motor commands or directional control logic.

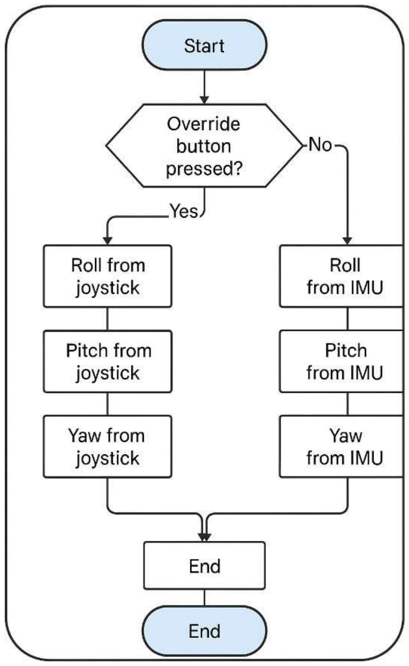


Fig. 4. Logic Flowchart for controlling both functionalities.

The MPU-6050 and KY-023 joystick links to Arduino Nano using I^{2}C and analog/digital signaling respectively. The detailed wiring configuration is shown in Table II, II, and IV in the *Appendix section* of the report.

D. Data Flow

The workflow, as depicted in Fig. 4, ensures a systematic approach to design and testing the Gimbal. Each stage listed is evaluated through experimental procedures in order to ensure that the final prototype meets the target specifications for stability, responsiveness, and user control. Moreover, the user commands from the joystick project analog that are sampled by the microcontroller's ADC and converted into digital signals, while the IMU streams accelerometer and gyroscope data to the microcontroller over I²C for real-time orientation estimation. The microcontroller executes control logic and passes through the filters to compute corrective targets and generates PWM motor commands for the gimbal actuators on each axis. The Joysticks adds an extra functionality for the Gimbal's platform stabilization by sharing each axis per motion, such as Roll and Pitch in one joystick, while the second inputs a Yaw command for the Servo Motor Actuation Commands.

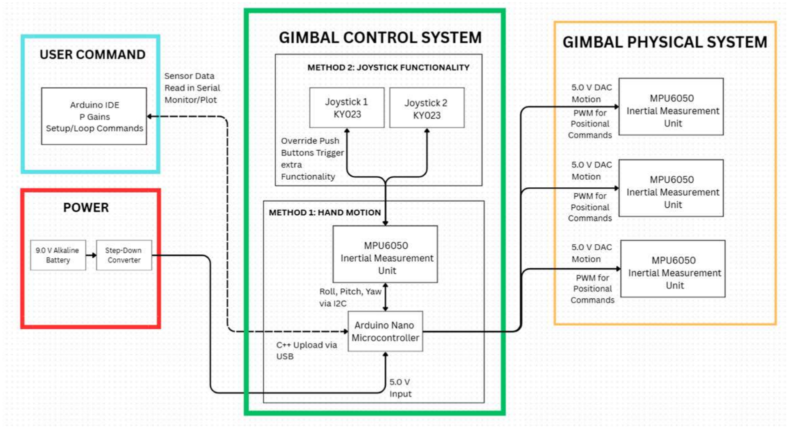

USER COMMAND
Arduino IDE
P Gains
Setup/Loop Commands
Sensor Data Read in Serial Monitor/Plot
POWER
9.0 V Alkaline Battery
Step-Down Converter
GIMBAL CONTROL SYSTEM
METHOD 2: JOYSTICK FUNCTIONALITY
Joystick 1 KY023
Joystick 2 KY023
Override Push Buttons Trigger extra Functionality
METHOD 1: HAND MOTION
MPU6050 Inertial Measurement Unit
Roll, Pitch, Yaw via I2C
Arduino Nano Microcontroller
C++ Upload via USB
5.0 V Input
GIMBAL PHYSICAL SYSTEM
5.0 V DAC Motion
PWM for Positional Commands
MPU6050 Inertial Measurement Unit
5.0 V DAC Motion
PWM for Positional Commands
MPU6050 Inertial Measurement Unit
5.0 V DAC Motion
PWM for Positional Commands
MPU6050 Inertial Measurement Unit

Fig. 4. Data Flow Structure for the DIY Gimbal Project

III. Experiment

A. Final Prototype

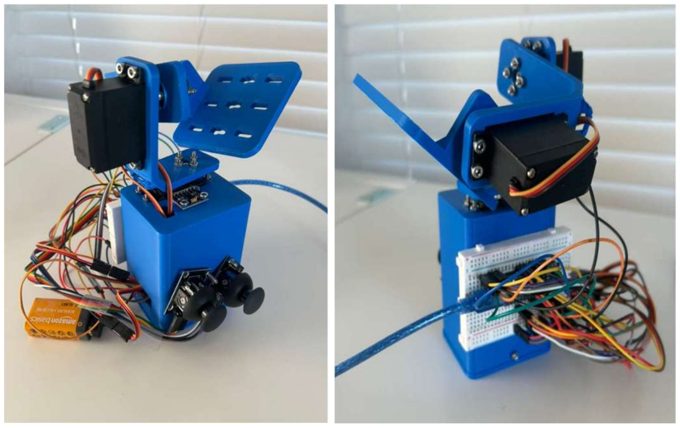


Fig. 5. Final Design of Gimbal Prototype

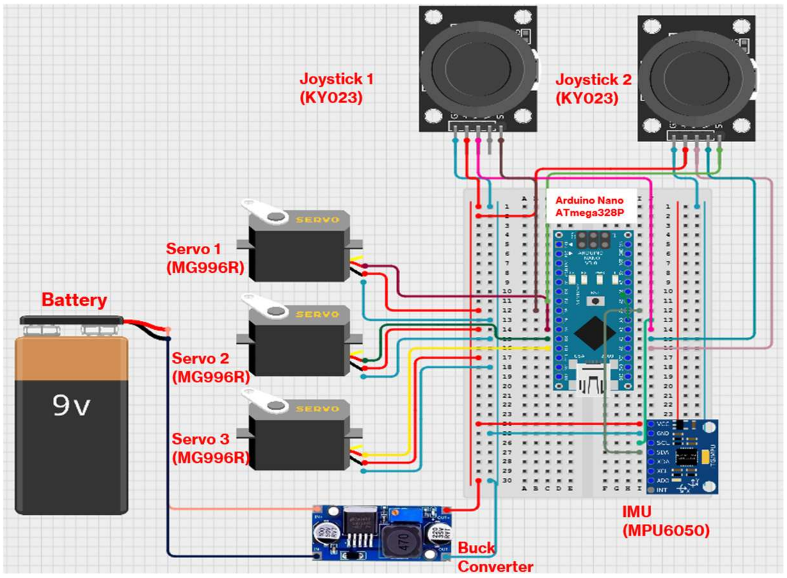


Fig. 6. Gimbal and Joystick Circuit Diagram

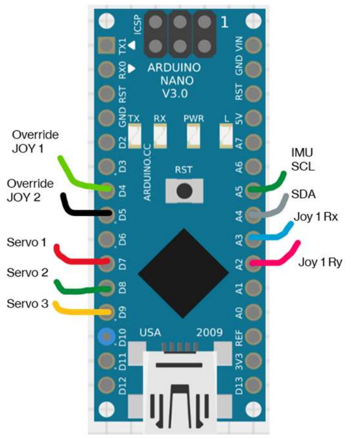


Fig. 7. Arduino Nano Microcontroller Circuit Diagram

The circuit design for the DIY gimbal system incorporates two distinct control functions: autonomous stabilization via IMU feedback and manual control via XY joystick modules.

The circuit diagrams, **Figs. 6 and 7**, illustrate a compact Arduino-based control system integrating multiple input and output components for a gimbal stabilization project. At the center of the layout is an **ATmega328P** mounted on a breadboard, serving as the main microcontroller. It interfaces with sensors and actuators through digital and analog pins.

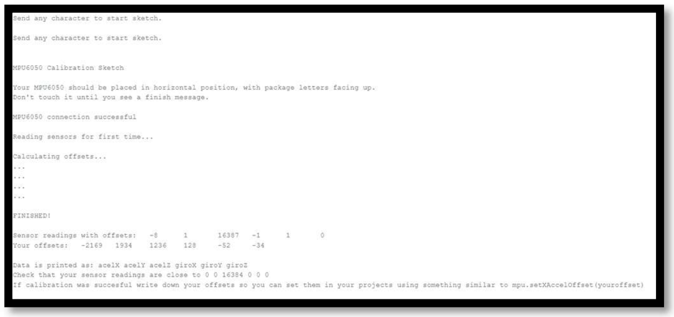
To the left and right of the Nano, **two joystick modules** are connected via multiple wires. These joysticks provide analog input signals for manual control of roll, pitch, and yaw. Each joystick has three axes (X, Y, and button), and their outputs are routed to analog input pins on the Nano.

Three **MG996R servo motors** are positioned around the board, each connected with three wires: signal (to digital PWM pins), power (5V), and ground. These servos control the mechanical movement of the gimbal system along the roll, pitch, and yaw axes.

Power is supplied by a **9V battery**, which is connected to the breadboard through a voltage regulator, that regulates the supplied voltage to be converted to about **5V**, providing adequate energy to both the Arduino and the servos. Proper grounding is maintained across all components to ensure stable operation.

Finally, an **MPU-6050 module** is wired to the Nano using the I²C protocol, with **SCL and SDA lines** connected to dedicated pins. This IMU provides real-time accelerometer and gyroscope data for orientation estimation, essential for closed-loop stabilization.

B. Evaluations from MATLAB



```
Send any character to start sketch.

Send any character to start sketch.


MPU6050 Calibration Sketch

Your MPU6050 should be placed in horizontal position, with package letters facing up.
Don't touch it until you see a finish message.

MPU6050 connection successful

Reading sensors for first time...

Calculating offsets...
...
...
...
...

FINISHED!

Sensor readings with offsets:   -8      1       16387   -1      1       0
Your offsets:   -2169   1934    1236    128     -52     -34

Data is printed as: acelX acelY acelZ giroX giroY giroZ
Check that your sensor readings are close to 0 0 16384 0 0 0
If calibration was succesful write down your offsets so you can set them in your projects using something similar to mpu.setXAccelOffset(youroffset)
```

Fig. 8. Calibrated Offset values for the Gimbal observations

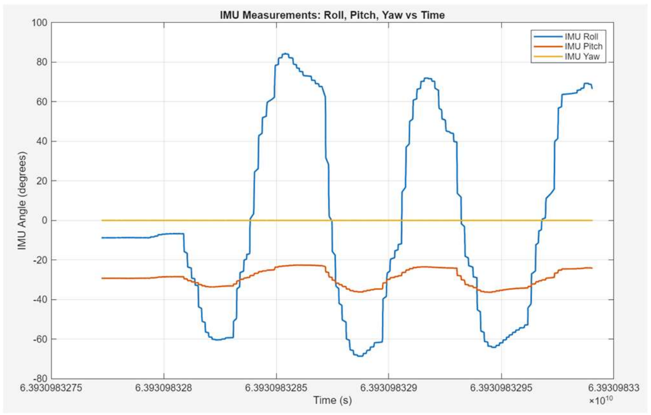


Fig. 9. Gimbal Method: IMU angle vs Time for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

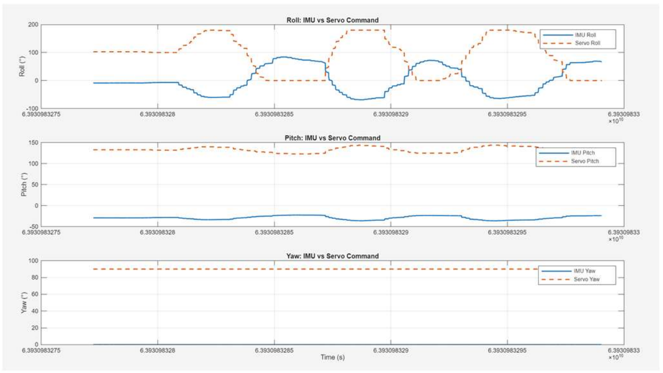


Fig. 10. Gimbal Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

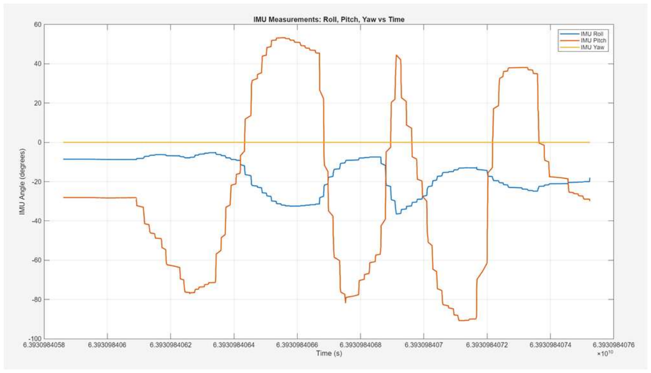


Fig. 11. Gimbal Method: IMU angle vs Time for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

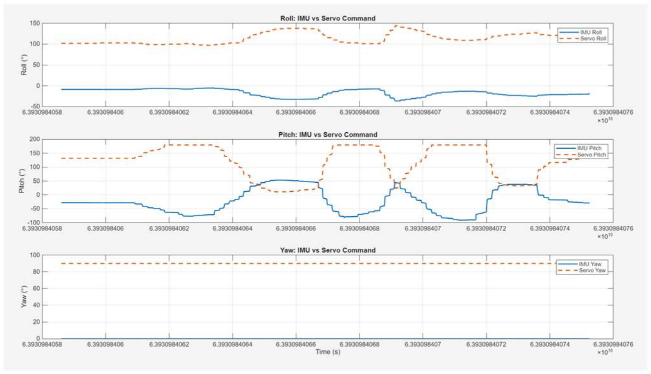


Fig. 12. Gimbal Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

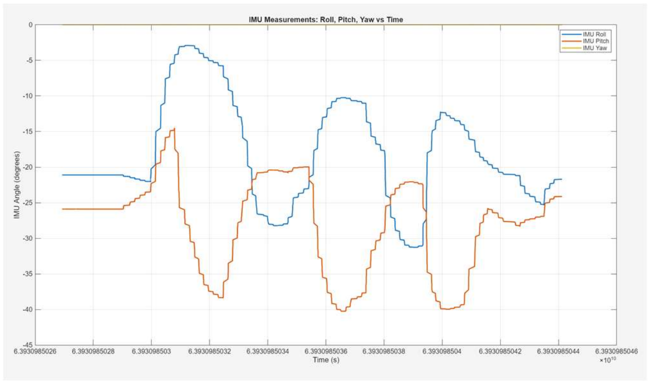


Fig. 13. Gimbal Method: IMU angle vs Time for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

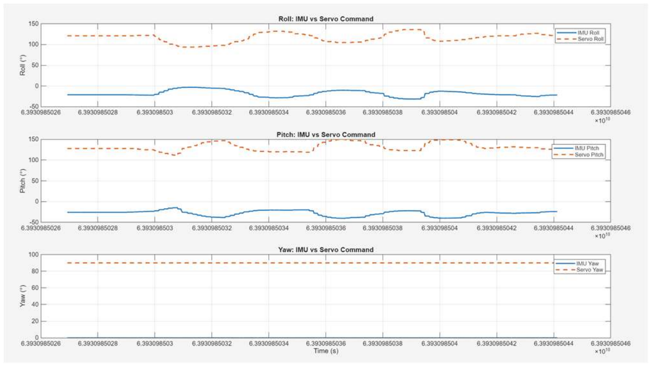
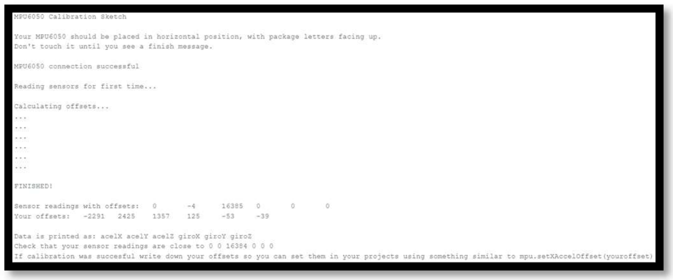


Fig. 14. Gimbal Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.



```
MPU6050 Calibration Sketch

Your MPU6050 should be placed in horizontal position, with package letters facing up.
Don't touch it until you see a finish message.

MPU6050 connection successful

Reading sensors for first time...

Calculating offsets...
...
...
...
...
...
...

FINISHED!

Sensor readings with offsets:   0       -4      16385   0       0       0
Your offsets:   -2291   2425    1357    125     -53     -39

Data is printed as: acelX acelY acelZ giroX giroY giroZ
Check that your sensor readings are close to 0 0 16384 0 0 0
If calibration was succesful write down your offsets so you can set them in your projects using something similar to mpu.setXAccelOffset(youroffset)
```

Fig. 15. Calibrated Offset values for the Gimbal observations

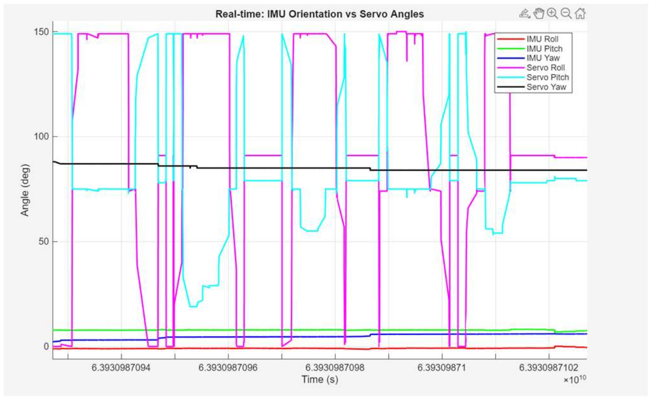


Fig. 16. Joystick Method: IMU vs Joystick Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

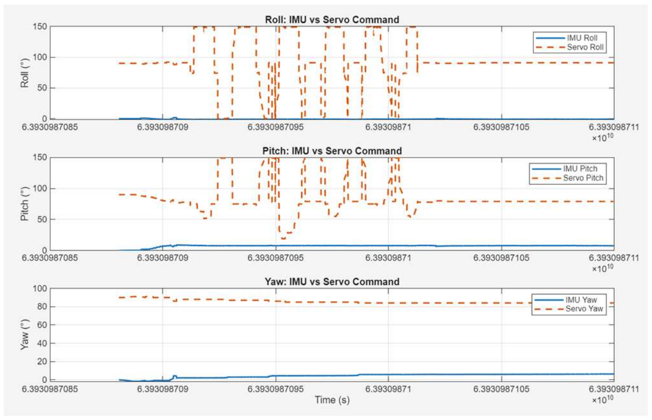


Fig. 17. Joystick Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

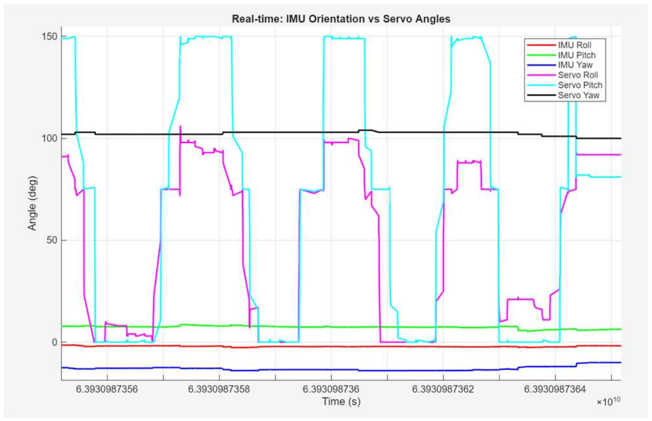


Fig. 18. Gimbal IMU vs Joystick Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

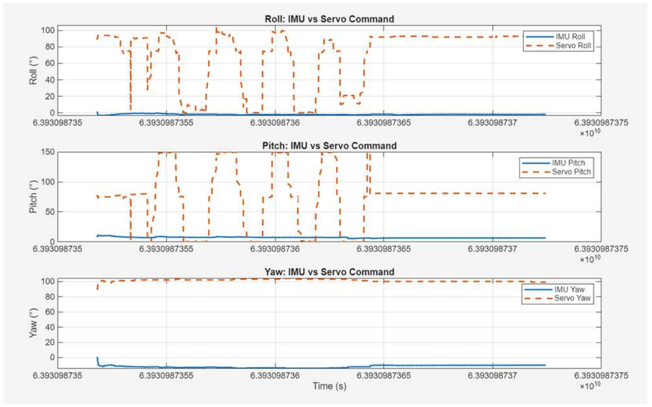


Fig. 19. Joystick Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

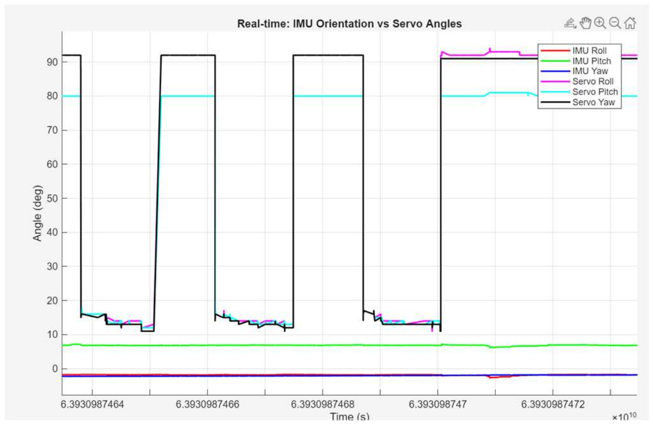


Fig. 20. Gimbal IMU vs Joystick Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

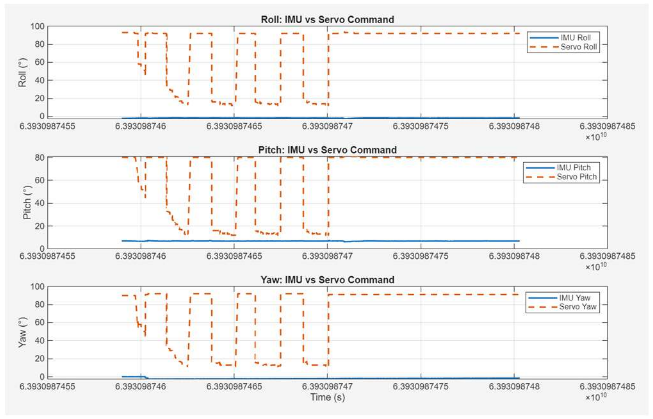


Fig. 21. Joystick Method: IMU vs Servo Command for the Roll/Pitch/Yaw Motion tracking for linear true rotation using Kalman Filter.

C. Evaluations from MATLAB

The following experiments examine the test trials for gimbal response and evaluate how accurately the tilt angles are actuated based on either hand motion detected by the IMU or joystick commands.

Therefore, the experiments are divided into two cases: IMU tilt and joystick command, each tested over three trials.

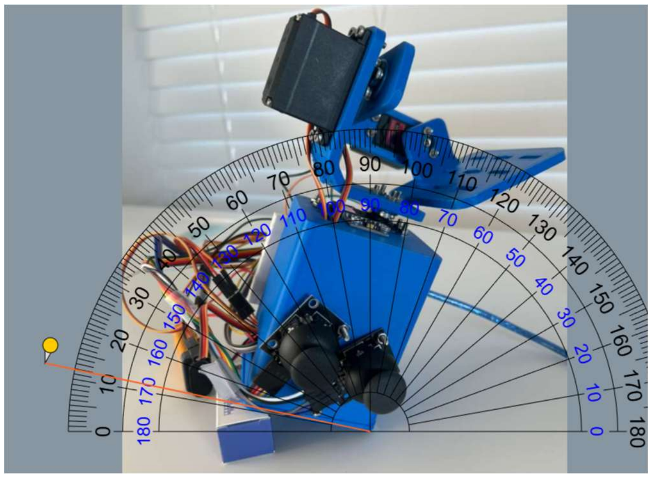


Fig. 22. Case 1: IMU Ground truth of about 12°

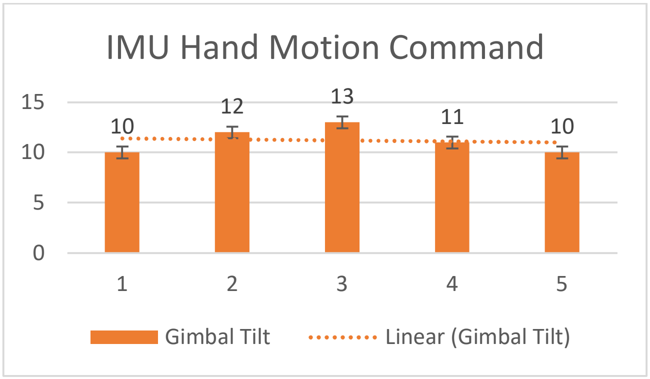
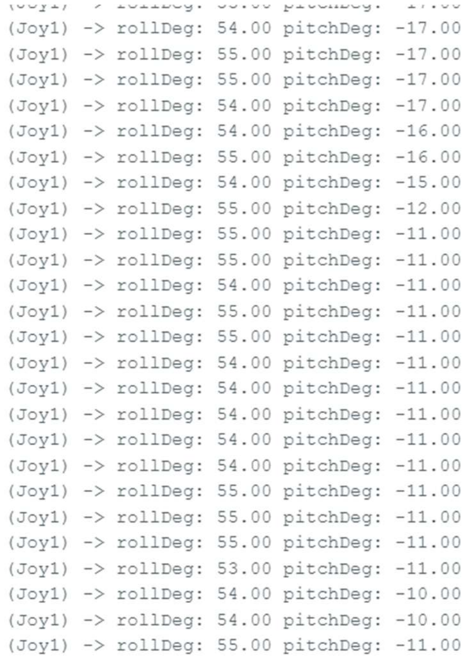


Fig. 23. Case 1 Test Trials of Gimbal Response (solid) and truth value (dashed).

The graph shows how much the gimbal tilts in response to hand movements detected by an IMU sensor. Each orange bar represents each tilt test, and the height of the bar shows how much the gimbal tilted during that moment. The error bars on top of each bar show how much variation or uncertainty there was in the tilt. Overall, the tilt values range from 10 to 13 degrees, and the dotted line shows a general upward trend, meaning the tilt tends to increase slightly over time or across tests. The values of the test trials suggest that the tilts responses of the gimbal relative to the hand motion are close to the ground truth value of 12°. This suggests that the gimbal is responding consistently to hand motion commands, with some variation.

Additionally, the Joystick Command Case goes over the tilt of the experiment as well. The experiment goes over by converting the joystick movement into readable angles from radians to degrees. The image below shows the high voltage as a 53-55° tilt command to the Gimbal.


```
(Joy1) -> rollDeg: 54.00 pitchDeg: -17.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -17.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -17.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -17.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -16.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -16.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -15.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -12.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 53.00 pitchDeg: -11.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -10.00
(Joy1) -> rollDeg: 54.00 pitchDeg: -10.00
(Joy1) -> rollDeg: 55.00 pitchDeg: -11.00
```
Fig. 24. Joystick Roll Degree Command tilt of 53-55°

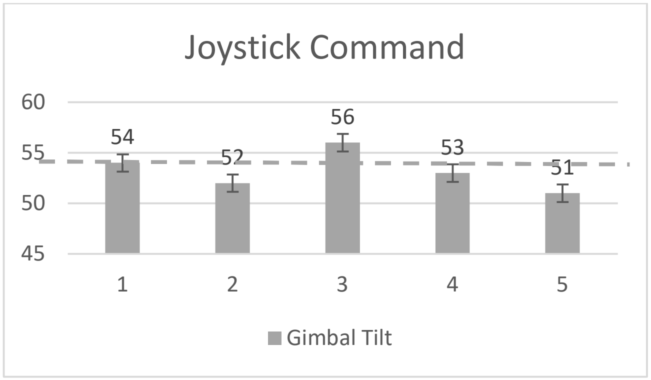


Fig. 25. Case 2 Test Trials of Gimbal Response (solid) and truth value (dashed).

This graph shows how the gimbal tilt responds to different joystick command levels. Each green bar represents the tilt test angle for a tilt command of 53-55°. The tilt values range from 51 to 56 degrees, with small error bars showing slight variation in each measurement. The dotted trend line slopes gently downward, suggesting that as the joystick command increases, the gimbal tilt slightly decreases. Overall, the tilt remains consistent, which may indicate stable control behavior across different joystick inputs.

D. Pitch Angle Experiments

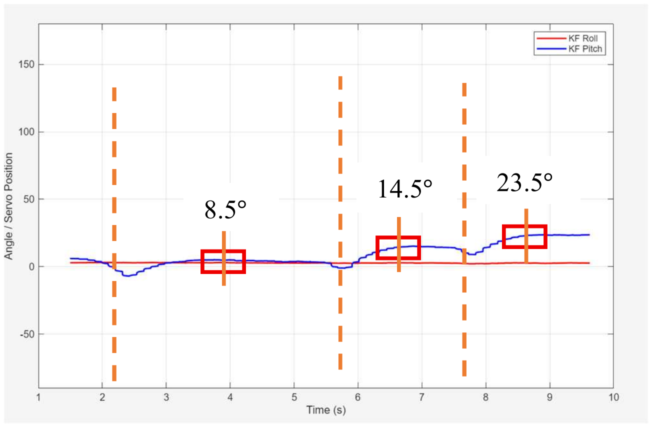


Fig. 26. Pitch and Roll Measured Results Postprocessing from IMU Hand Motion.

TABLE I: Percentage Error for MPU6050 Readings for Pitch Axis

Actual Rotation (Degree)	Measured Reading (Degree)	Percent Error
11	8.5 ±3	22.73 ±0.72%
16.5	14.5 ±3	9.37 ±0.81%
25	23.5 ±3	6% ±0.88%

This tilt experiment allows for one axis rotations, along the Pitch axis, for three trials and the actual tilts and measured tilts are tabulated in Table I. By analyzing the increasing Pitch rotation using the hand motion, the percent error between the measured and actual rotations deviate less significantly, with the confidence level about ±5, therefore, this experiment suggests that the angle displacements are accurate when taking into account designing the motion controller when integrating the hand motion sensing control for irregular angle tilts for the object.

Case 2: Joystick Command

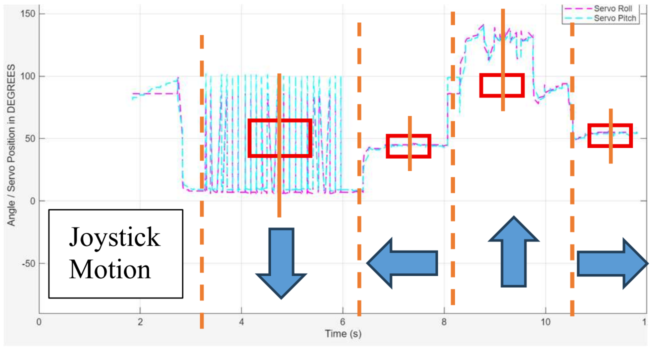


Fig. 27. Pitch and Roll Measured Results Postprocessing from IMU Joystick Command.

TABLE II: Percentage Error for MPU6050 Readings for Pitch Axis

Joystick Direction	Actual Rotation (Degree)	Measured Reading (Degree)	Percent Error
Down	0	~50 ±40	~99%
Left	50	~45 ±10	10%
Up	100	~100 ±10	10%
Right	50	~60 ±10	20%

This tilt experiment allows for axis rotations, along the Pitch and Roll axes, for four trials and the actual tilts and measured tilts are tabulated in Table II. The reasoning behind experimenting between the Pitch and Roll axes for using the joystick is to analyze the behavior of Joystick transitioning moving in all four directions and the smoothness of the measured readings. Therefore, by analyzing the increasing Pitch and Roll rotations using the joystick commands, the percent error between the measured and actual rotations deviate significantly, with the confidence level ranging from ±10 up to ±40, therefore, this experiment suggests that the angle displacements are less accurate when taking into account designing the motion controller.

IV. Discussion and Recommendations

The gimbal prototype displays the two control motion tracking methods as illustrated in the experimental readings. There was good reading using both hand rotation and joystick control, and the IMU measurements aligning well with servo commands. However, a major limitation observed during hand rotation plots is the lack of reading yaw values from the IMU. Unlike roll and pitch, which are derived from accelerometer and gyroscope fusion, yaw estimation depends heavily on magnetometer data, which the MPU6050 lacks. As a result, yaw tracking becomes unstable or unreadable during free rotation, making it difficult to validate true orientation in the horizontal plane. This hardware constraint limits the system's ability to perform full 3D orientation tracking, especially in environments with dynamic or nonlinear yaw motion. For future improvements, the system should consider upgrading to a 9-axis IMU that includes magnetometer support for accurate yaw estimation. In the joystick method plots, the IMU and servo angles are more synchronized, especially for pitch and yaw, suggesting improved control precision. As illustrated in the graphs, the Proportional Gain values allowed for the IMU angle reading to change almost instantly from LOW to HIGH, due to already defined Proportional Gain values in the code, which makes sense for the high sensitivity readings when using the Joystick for reading all the three motion axis.

Furthermore, the experiments demonstrated that the gimbal responded reliably to both IMU hand motion inputs and joystick commands, with tilt angles tracking closely to the intended values. However, during the live demo a noticeable issue arose with the servo motor performance, where the pitch axis did not actuate as smoothly as expected. This irregularity is likely linked to a gear ratio conversion mismatch, which may have caused the servo to deliver less torque or inaccurate angular displacement under load. Addressing this mechanical limitation will be important in future iterations, as proper gear ratio alignment is critical for ensuring consistent and precise gimbal control across all axes.

V. References

[1] K. W. Williams, "A Summary of Unmanned Aircraft Accident/Incident Data: Human Factors Implications," Federal Aviation Administration, Oklahoma City, 2004.

[2] H. Kim and W. Chang, "Intuitive Drone Control using Motion Matching between a Controller and a Drone," Archives of Design Research, vol. 35, no. 1, pp. 93–113, 2022. doi:10.15187/adr.2022.02.35.1.93

[3] M. Menebo Madebo, "Neuro-fuzzy-based adaptive sliding mode control of quadrotor UAV in the presence of matched and unmatched uncertainties," IEEE Access, vol. 12, pp. 117745–117760, 2024. doi:10.1109/access.2024.3447474

[4] E. Mounier, M. Karaim, M. Korenberg, and A. Noureldin, "Multi-IMU System for Robust Inertial Navigation: Kalman Filters and Differential Evolution-Based Fault Detection and Isolation," *IEEE Sensors Journal*, vol. 25, no. 6, pp. 9998–10014, Mar. 2025, doi: 10.1109/JSEN.2025.3536806.

VI. APPENDIX

TABLE II. Summary of design challenges and approaches for the DIY gimbal system.

Challenge	Description	Approach
Real-time orientation estimation	Maintaining stable pitch and roll orientation despite vibrations or external motion.	Implement gains to tune the responses for smooth stabilization
Sensor Response	MPU6050 accelerometer/gyroscope data exhibit bias and noise over time.	The IMU will be calibrated by calculating its offsets and then removing the bias in the readings. Additionally, Kalman filter will be used to ensure accurate readings to avoid gyro bias over time.
Integration of Dual Joysticks	Two joysticks must operate simultaneously through ADC channels.	An iterative design process will be used to integrate both joysticks into one microcontroller, allowing them to interact effectively with the Gimbal system for manual control.
Mode Switching Logic	Switching between automatic stabilization and manual override requires careful control to prevent unstable movement.	The mode-switching function will be implemented in the Arduino code.
Sensor and ADC Communication	The system involves simultaneous I²C communication for the IMU.	The Analog-to-Digital communication architecture will be optimized to prevent data overlapping, using polling and timing control within the main program loop.
Component Placement and Wiring Constraints	Component layout can cause signal interference and imbalance in Gimbal Structure.	The components will be positioned to ensure short wiring paths and stability between the IMU and servo control signals.

Table III. Communication Protocol Between the MPU-6050 Sensor and The Arduino Nano

MP6050	**Arduino Nano Pin**	**Function**
VCC	5V	Power
GND	GND	Ground
SCL	A5	I^{2}C Clock
SDA	A4	I^{2}C Data

The table shows the wiring connections between the MPU6050 sensor and the Arduino Nano. The sensor's VCC pin is connected to the 5V pin on the Arduino to supply power, while the GND pin is connected to ground. For data communication, the SCL (clock) pin is connected to analog pin A5, and the SDA (data) pin is connected to analog pin A4. These connections enable I²C communication, allowing the Arduino to receive motion data from the sensor for orientation tracking.

Table IV. Communication Protocol Between the KY-023 Joystick Module and the Arduino Nano (Joystick 1)

Joystick Pin	**Arduino Nano Pin**	**Function**
GND	GND	Ground connection
+5 (VCC)	5V	Power Supply
VRx	A0	Roll Movement
VRy	A1	Pitch Movement
SW	D2	Digital Signal Press- down switch (Manual mode switching)

Tables III shows how joystick 1 is connected to the Arduino Nano for controlling the Roll and Pitch motion axes. The joystick receive power through the 5V and GND pins. The first joystick uses analog pins A0 and A1 to control roll and pitch, while its push-button switch is connected to digital pin D2 for manual mode switching.

Table V. Communication Protocol Between the KY-023 Joystick Module and the Arduino Nano (Joystick 2)

Joystick Pin	**Arduino Nano Pin**	**Function**
GND	GND	Ground connection
+5 (VCC)	5V	Power Supply
VRx	A2	Yaw Movement
VRy	-	-
SW	D2	Digital Signal Press- down switch (Manual mode switching)

Tables IV shows how joystick 2 is connected to the Arduino Nano for controlling the Yaw motion axis. The joystick receive power through the 5V and GND pins. The second joystick connects its yaw control to analog pin A2, and also uses digital pin D2 for its switch. These connections allow the Arduino to read directional input and button presses from both joysticks for multi-axis gimbal control.

MATLAB Code for IMU angle vs Time for Roll/Pitch/Yaw

```
time = allData(:,1);
imu_roll  = allData(:,2);
imu_pitch = allData(:,3);
imu_yaw   = allData(:,4);

figure;
plot(time, imu_roll,  'LineWidth', 1.5); hold
on;
plot(time, imu_pitch, 'LineWidth', 1.5);
plot(time, imu_yaw,   'LineWidth', 1.5);
hold off;

xlabel('Time (s)');
ylabel('IMU Angle (degrees)');
title('IMU Measurements: Roll, Pitch, Yaw vs
Time');
legend('IMU Roll','IMU Pitch','IMU Yaw');
grid on;
```

MATLAB Code for IMU vs Servo Command for Roll/Pitch/Yaw

```
time = allData(:,1);

figure;

subplot(3,1,1);
plot(time, allData(:,2), 'LineWidth', 1.5);
hold on;
plot(time, allData(:,5), '--', 'LineWidth',
1.5);
ylabel('Roll (°)');
title('Roll: IMU vs Servo Command');
legend('IMU Roll','Servo Roll');
grid on;

subplot(3,1,2);
plot(time, allData(:,3), 'LineWidth', 1.5);
hold on;
plot(time, allData(:,6), '--', 'LineWidth',
1.5);
ylabel('Pitch (°)');
title('Pitch: IMU vs Servo Command');
legend('IMU Pitch','Servo Pitch');
grid on;

subplot(3,1,3);
plot(time, allData(:,4), 'LineWidth', 1.5);
hold on;
plot(time, allData(:,7), '--', 'LineWidth',
1.5);
ylabel('Yaw (°)');
xlabel('Time (s)');
title('Yaw: IMU vs Servo Command');
legend('IMU Yaw','Servo Yaw');
grid on;
```

MATLAB Code for IMU/Joystick Angle Measurements

```
%% Real-time Joystick vs IMU Plot
clearvars; close all; clc;

% --- USER SETTINGS ---
Port = 'COM3';            % Your COM port
BaudRate = 115200;        % Serial baud rate
TimeWindow = 10;          % seconds of data to
show on x-axis
YLimAngles = [-90 90];    % KF Roll/Pitch
angles limits
YLimServo  = [0 180];     % Servo limits

% --- Serial setup ---
s = serialport(Port, BaudRate);
flush(s);  % clear the buffer

% --- Initialize data storage ---
tData = [];
KF_RollData = [];
KF_PitchData = [];
ServoRollData = [];
ServoPitchData = [];

% --- Create figure ---
figure('Name','Joystick vs
IMU','NumberTitle','off');
hKF_Roll   = plot(nan, nan, 'r', 'LineWidth',
1.5); hold on;
hKF_Pitch  = plot(nan, nan, 'b', 'LineWidth',
1.5);
hServoRoll = plot(nan, nan, 'm--',
'LineWidth', 1.2);
hServoPitch= plot(nan, nan, 'c--',
'LineWidth', 1.2);
xlabel('Time (s)');
ylabel('Angle / Servo Position in DEGREES');
legend('KF Roll','KF Pitch','Servo
Roll','Servo Pitch');
grid on;
ylim([min(YLimAngles(1), YLimServo(1)),
max(YLimAngles(2), YLimServo(2))]);

startTime = datetime('now');

%% --- Real-time loop ---
while ishandle(hKF_Roll)
```

```
    % Read line from serial
    line = readline(s);

    % Try parsing numbers
    try
        numbers = sscanf(line, ...
            'KF Roll: %f KF Pitch: %f
GyroYaw: %f | ServoRoll: %f ServoPitch: %f
ServoYaw: %f');

        if numel(numbers) == 6
            tNow = seconds(datetime('now') -
startTime);

            % Append data
            tData(end+1) = tNow;
            KF_RollData(end+1)    =
numbers(1);
            KF_PitchData(end+1)   =
numbers(2);
            ServoRollData(end+1)  =
numbers(4);
            ServoPitchData(end+1) =
numbers(5);

            % Keep only data within
TimeWindow
            idx = tData >= (tNow -
TimeWindow);
            tData = tData(idx);
            KF_RollData = KF_RollData(idx);
            KF_PitchData = KF_PitchData(idx);
            ServoRollData =
ServoRollData(idx);
            ServoPitchData =
ServoPitchData(idx);

            % Update plots
            %set(hKF_Roll, 'XData', tData,
'YData', KF_RollData);
            %set(hKF_Pitch, 'XData', tData,
'YData', KF_PitchData);
            set(hServoRoll, 'XData', tData,
'YData', ServoRollData);
            set(hServoPitch, 'XData', tData,
'YData', ServoPitchData);

            drawnow limitrate
        end
    catch
        warning('Failed to parse line: %s',
line);
    end
end

% --- Cleanup ---
clear s;
```

ARDUINO IDE CODE

```
   #include <Wire.h>
   #include <MPU6050.h>
   #include <Servo.h>
   #include "Kalman.h"

   // --- IMU and Kalman Filter ---
   MPU6050 mpu;
   Kalman kalmanX;   // Roll filter
   Kalman kalmanY;   // Pitch filter

   Servo servoRoll;
   Servo servoPitch;
   Servo servoYaw;

   unsigned long timerMicros;
   double kalAngleX, kalAngleY;
   double gyroXangle, gyroYangle, gyroZangle;
   double compAngleX, compAngleY;
   double yawAngle = 0;

   #define ROLL_GAIN   1.5
   #define PITCH_GAIN  1.5
   #define YAW_GAIN    1.5
   #define   ALPHA              0.98      //
complementary filter blend factor

   // Joystick 1: pitch/roll
   const int joy1RollPin  = A0;
   const int joy1PitchPin = A1;
   const  int  btnOverride1  =  4;    //  active
LOW

   // Joystick 2: roll/pitch/yaw
   const int joy2RollPin  = A6;
   const int joy2PitchPin = A7;
   const  int  joy2YawPin    =  A2;  //  NEW:  yaw
axis
   const  int  btnOverride2  =  5;    //  active
LOW

   static  inline  double  rad2deg(double  r)
{ return r * (180.0 / PI); }

   void setup() {
     Serial.begin(115200);
     Wire.begin();

     // Initialize MPU
     mpu.initialize();
```

```
      if (!mpu.testConnection()) {
           Serial.println("MPU6050 connection failed");
        while (1);
      }
      Serial.println("MPU6050 Connected");

      // Calibration offsets
      mpu.setXAccelOffset(-2169);
      mpu.setYAccelOffset(1934);
      mpu.setZAccelOffset(1236);
      mpu.setXGyroOffset(128);
      mpu.setYGyroOffset(-52);
      mpu.setZGyroOffset(-34);

      // Initial sensor read
      int16_t ax, ay, az, gx, gy, gz;
      mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

      double accXangle = rad2deg(atan2((double)ay, (double)az));
      double accYangle = rad2deg(atan2(-(double)ax, sqrt((double)ay * ay + (double)az * az)));

      kalmanX.setAngle(accXangle);
      kalmanY.setAngle(accYangle);

      gyroXangle = accXangle;
      gyroYangle = accYangle;

      compAngleX = accXangle;
      compAngleY = accYangle;

      // Attach servos
      servoRoll.attach(9);
      servoPitch.attach(8);
      servoYaw.attach(7);

      // Joystick buttons
      pinMode(btnOverride1, INPUT_PULLUP);
      pinMode(btnOverride2, INPUT_PULLUP);

      timerMicros = micros();
    }

    void loop() {
      // --- Sensor read ---
      int16_t ax, ay, az, gx, gy, gz;
      mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

      unsigned long now = micros();
      double dt = (double)(now - timerMicros) / 1e6;
      timerMicros = now;
      if (dt <= 0) dt = 1e-3;

      double gyroXrate = (double)gx / 131.0;
      double gyroYrate = (double)gy / 131.0;
      double gyroZrate = (double)gz / 131.0;

      double accXangle = rad2deg(atan2((double)ay, (double)az));
      double accYangle = rad2deg(atan2((double)ax, sqrt((double)ay * ay + (double)az * az)));

      // Integrate gyro
      gyroXangle += gyroXrate * dt;
      gyroYangle += gyroYrate * dt;
      gyroZangle += gyroZrate * dt;

      // Complementary filter
      compAngleX = ALPHA * (compAngleX + gyroXrate * dt) + (1.0 - ALPHA) * accXangle;
      compAngleY = ALPHA * (compAngleY + gyroYrate * dt) + (1.0 - ALPHA) * accYangle;

      yawAngle += gyroZrate * dt;

      if (yawAngle > 180) yawAngle -= 360;
      if (yawAngle < -180) yawAngle += 360;

      // Slow drift damping
      yawAngle *= 0.9993;

      // Kalman fusion
      kalAngleX = kalmanX.getAngle(accXangle, gyroXrate, dt);
      kalAngleY = kalmanY.getAngle(accYangle, gyroYrate, dt);

      // --- Override logic ---
      bool override1 = (digitalRead(btnOverride1) == LOW);
      bool override2 = (digitalRead(btnOverride2) == LOW);

      int servoRollAngle;
      int servoPitchAngle;
    int servoYawAngle;

    if (override1 || override2) {
          int rollRaw = override2 ?
analogRead(joy2RollPin) :
analogRead(joy1RollPin);
          int pitchRaw = override2 ?
analogRead(joy2PitchPin) :
analogRead(joy1PitchPin);

        servoRollAngle = map(rollRaw, 0,
1023, 0, 180);
        servoPitchAngle = map(pitchRaw, 0,
1023, 0, 180);

      if (override2) {
        int yawRaw = analogRead(joy2YawPin);
        servoYawAngle = map(yawRaw, 0, 1023,
0, 180);
      } else {
            servoYawAngle = map((long)(-
gyroZangle * YAW_GAIN), -180, 180, 0, 180);
      }

      Serial.print("Manual Override ");
        Serial.print(override2 ? "(Joy2)" :
"(Joy1)");
          Serial.print(" -> rollRaw: ");
Serial.print(rollRaw);
            Serial.print(" pitchRaw: ");
Serial.print(pitchRaw);
      if (override2) {
              Serial.print(" yawRaw: ");
Serial.print(analogRead(joy2YawPin));
      }
    } else {
      // Automatic stabilization
          servoRollAngle = map((long)(-
kalAngleX * ROLL_GAIN), -90, 90, 0, 180);
          servoPitchAngle = map((long)(-
kalAngleY * PITCH_GAIN), -90, 90, 0, 180);
          servoYawAngle = map((long)(-
gyroZangle * YAW_GAIN), -180, 180, 0, 180);
    }

    // --- Drive servos ---
              servoRollAngle =
constrain(servoRollAngle, 0, 180);
              servoPitchAngle =
constrain(servoPitchAngle, 0, 180);
            servoYawAngle =
constrain(servoYawAngle, 0, 180);

     servoRoll.write(servoRollAngle);
     servoPitch.write(servoPitchAngle);
     servoYaw.write(servoYawAngle);

     // --- Debug ---
             Serial.print("\nKF Roll:
"); Serial.print(kalAngleX, 3);
             Serial.print("KF Pitch:
"); Serial.print(kalAngleY, 3);
             Serial.print(" GyroYaw:
"); Serial.print(gyroZangle, 3);
           Serial.print(" | ServoRoll:
"); Serial.print(servoRollAngle);
            Serial.print(" ServoPitch:
"); Serial.print(servoPitchAngle);
             Serial.print(" ServoYaw:
"); Serial.print(servoYawAngle);

     delay(10); // ~100 Hz
   }
```