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.

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.

II. 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. Traditional joystick-based control methods, while effective, often require the user to mentally translate two-dimensional joystick motion into three-dimensional system movement. 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.

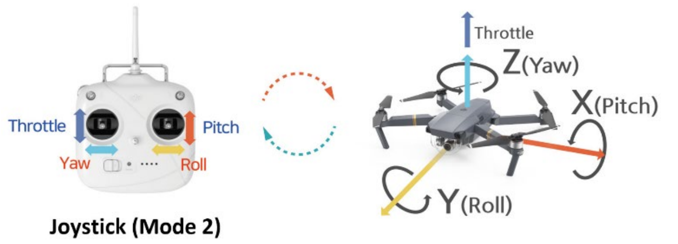


Fig. 1. Typical dual-joystick control layout for drone or gimbal systems. [2]

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, especially in first-person-view (FPV) operation. These findings highlight that mapping the physical motion of the operator to the controlled system reduces reference-frame misalignment and enhances user intuition.

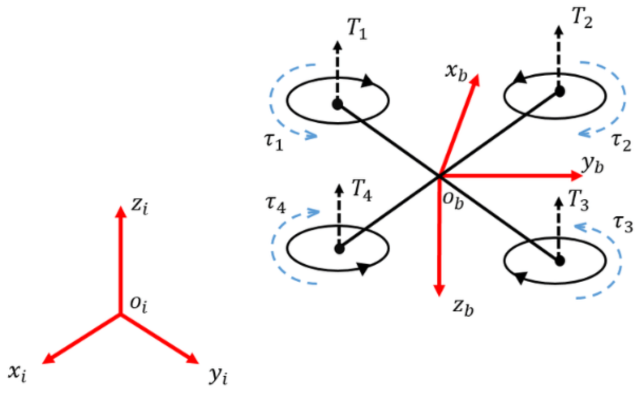


Fig. 2. Illustration of reference-frame misalignment [3]

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. This cognitive effort often leads to control errors. 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.

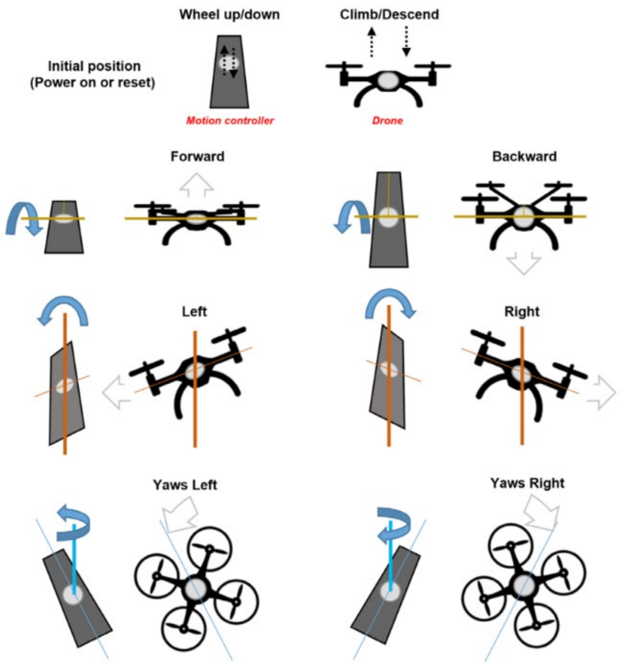


Fig. 3. Motion-matching control concept. [2]

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.

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, joystick, and center of mass of all the components must be balanced for minimal disturbances due to the misalignment behaviors. Overall, the key design challenges and strategies are summarized in Table 1.

TABLE I. 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.

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.

III. Materials and Methods

A. Components

- ATmega328P – Arduino Nano microcontroller
- MPU6050 IMU – sensor module
- MG996R servo motors – 3 of them
- KY-023 analog joystick – Extra functionality
- Buck Converter – 5V Power Regulator
- 3D-printed Gimbal frame (CAD shown in Fig. 2)

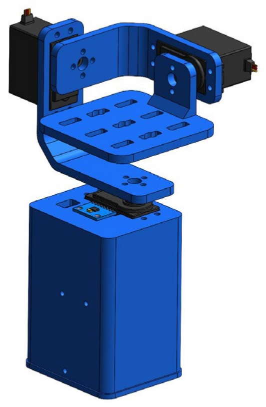


Fig. 2. Computer Aided Design of a Gimbal Prototype

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.

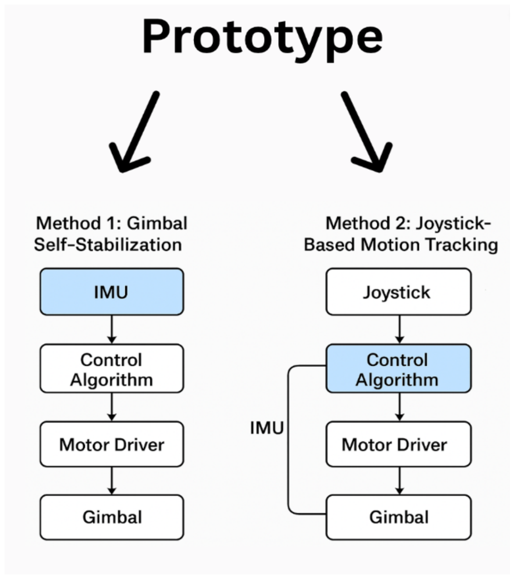


Fig. 3. Methodology Mind map

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^{2}C 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.

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.

Table II. 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 I^{2}C communication, allowing the Arduino to receive motion data from the sensor for orientation tracking.

Table III. 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 IV. 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.

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.

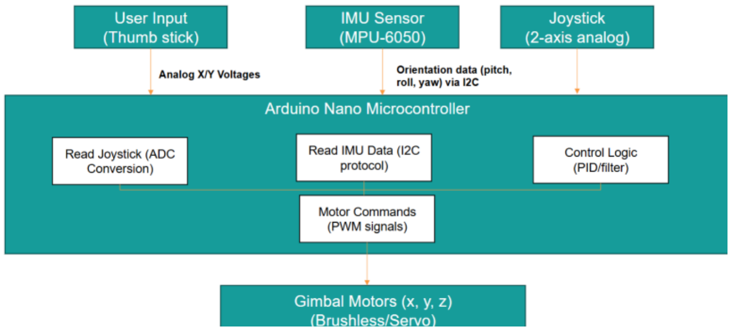

User Input (Thumb stick)
IMU Sensor (MPU-6050)
Joystick (2-axis analog)
Analog X/Y Voltages
Orientation data (pitch, roll, yaw) via I2C
Arduino Nano Microcontroller
Read Joystick (ADC Conversion)
Read IMU Data (I2C protocol)
Control Logic (PID/filter)
Motor Commands (PWM signals)
Gimbal Motors (x, y, z) (Brushless/Servo)

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

IV. Experiment

A. Final Prototype

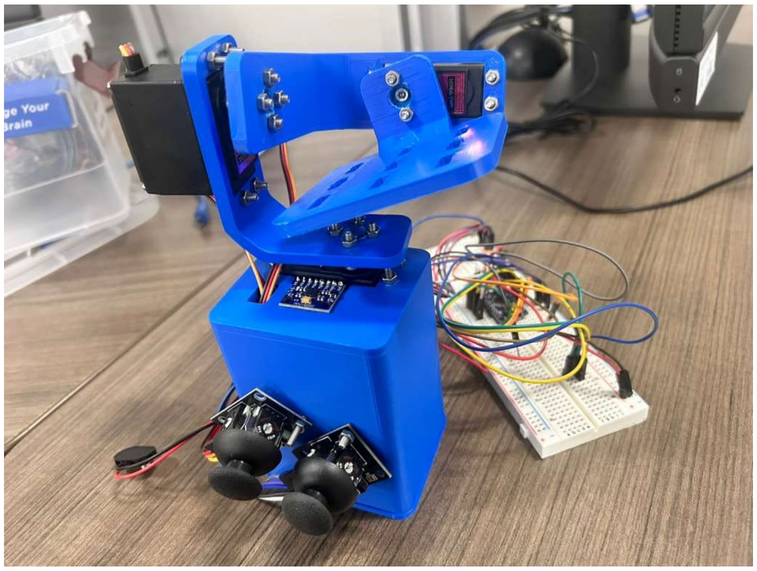


Fig. 5. Final Design of Gimbal Prototype

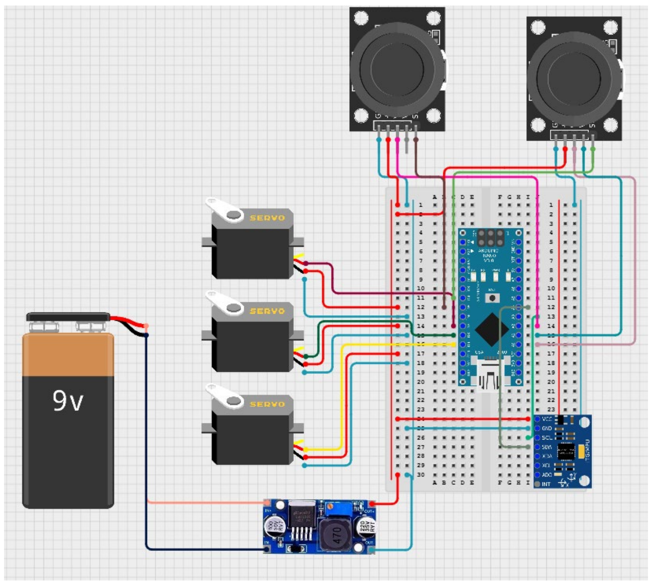


Fig. 6. Gimbal and Joystick Circuit Diagram

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 diagram illustrates 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 **Arduino Nano**, mounted on a breadboard, serving as the main microcontroller. It interfaces with sensors and actuators through digital and analog pins.

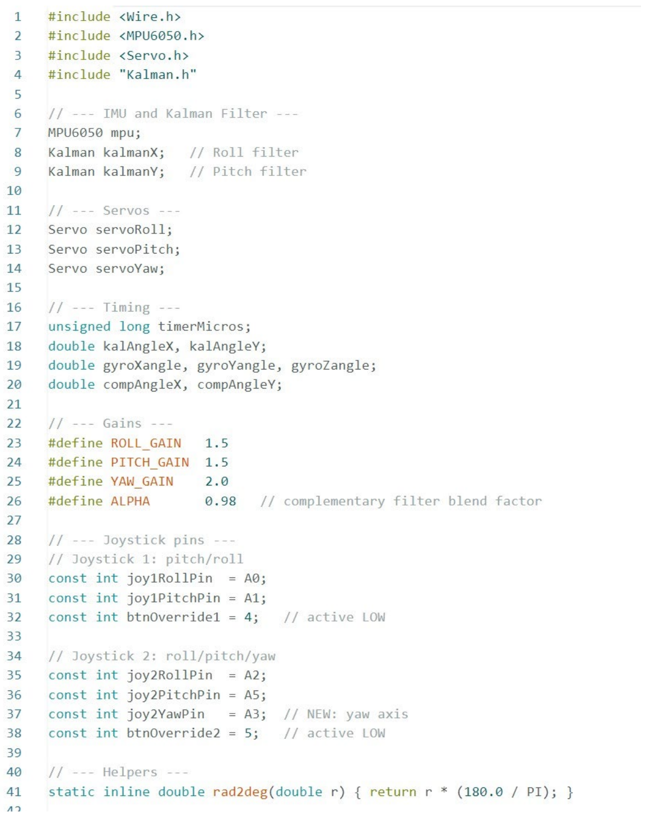
To the left and right of the Nano, **two joystick modules** are connected via multiple wires. These joysticks provide analog input signals for manual control of roll, pitch, and yaw. Each joystick has three axes (X, Y, and button), and their outputs are routed to analog input pins on the Nano.

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, providing 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.

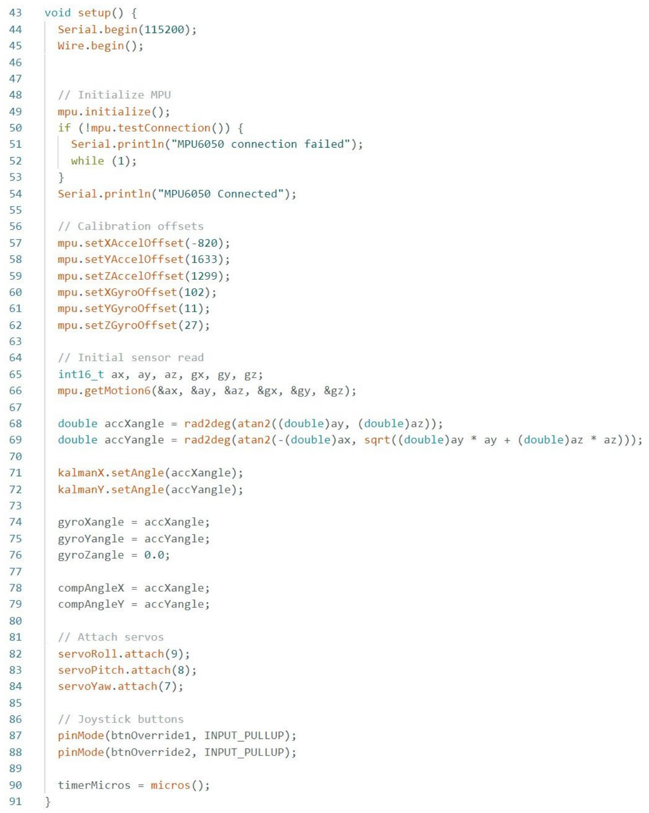
B. Arduino IDE Program



```
#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

// --- Servos ---
Servo servoRoll;
Servo servoPitch;
Servo servoYaw;

// --- Timing ---
unsigned long timerMicros;
double kalAngleX, kalAngleY;
double gyroXangle, gyroYangle, gyroZangle;
double compAngleX, compAngleY;

// --- Gains ---
#define ROLL_GAIN   1.5
#define PITCH_GAIN  1.5
#define YAW_GAIN    2.0
#define ALPHA       0.98   // complementary filter blend factor

// --- Joystick pins ---
// 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  = A2;
const int joy2PitchPin = A5;
const int joy2YawPin   = A3;  // NEW: yaw axis
const int btnOverride2 = 5;   // active LOW

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

Fig. 7. Initialization

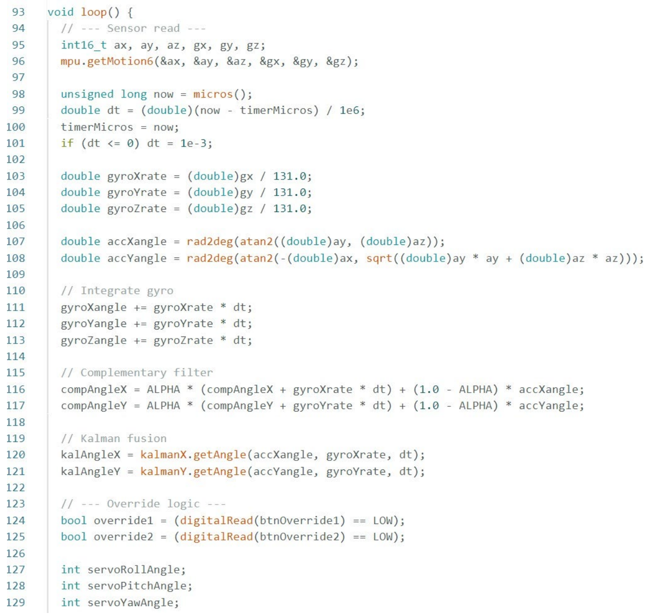
This section of the code sets up the hardware and control parameters for a joystick-controlled gimbal stabilization system. It begins by including the necessary libraries for sensor communication, servo control, and Kalman filtering. The MPU6050 sensor is used to measure motion, and Kalman filters are applied to estimate roll and pitch angles more accurately. Three servo motors are declared to control the roll, pitch, and yaw axes of the gimbal. Timing variables are prepared for tracking sensor updates, and gain values are defined to adjust the responsiveness of each axis. The joystick pins are assigned to analog inputs, allowing the user to manually control the gimbal's orientation. Two buttons are used to activate manual override modes. A helper function is also included to convert radians to degrees, which simplifies angle calculations later in the code.



```
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(-820);
  mpu.setYAccelOffset(1633);
  mpu.setZAccelOffset(1299);
  mpu.setXGyroOffset(102);
  mpu.setYGyroOffset(11);
  mpu.setZGyroOffset(27);

  // 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;
  gyroZangle = 0.0;

  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();
}
```

Fig. 8. Void Setup

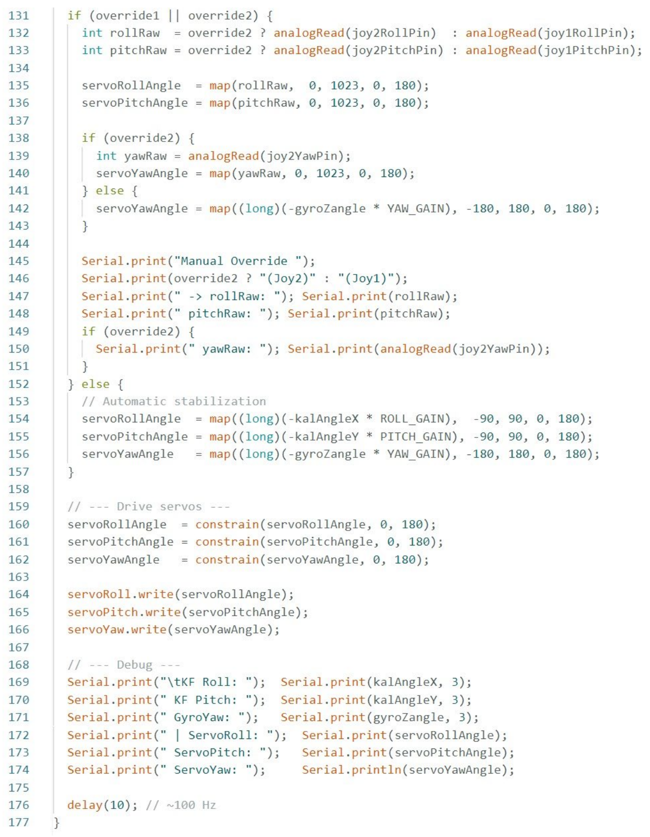
This section of the code initializes the hardware components used in the gimbal stabilization system. It begins by setting up serial communication and starting the I^{2}C interface for sensor data transfer. The MPU6050 sensor is then initialized and tested to confirm a successful connection. Calibration offsets are applied, from previous experimental data, to correct for sensor bias in both the accelerometer and gyroscope readings. After the initial sensor data is collected, roll and pitch angles are calculated using basic trigonometric functions and converted to degrees. These angles are used to initialize the Kalman filters and reference variables for angle tracking. The servos controlling the gimbal axes are attached to specific pins D7, D8, D9, and a button input is configured for manual control. Finally, the system timer is started to track update intervals for motion processing.



```
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;

  // 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;
```

Fig. 9. Void Loop

This part of the code runs continuously to read motion data from the MPU sensor and calculate orientation angles in real time. It begins by collecting raw accelerometer and gyroscope values, then computes the time difference between readings to track motion accurately. Using trigonometric functions, it calculates roll and pitch angles from the accelerometer data. The gyroscope data is integrated over time to estimate angle changes, which are then combined with accelerometer readings using a complementary filter. A Kalman filter is also applied to improve accuracy by reducing noise. Finally, the code checks if either override button is pressed to prepare for manual joystick control of the gimbal system.



```
  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("\tKF 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.println(servoYawAngle);

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

Fig. 10. Void Loop (continued)

This section of the code handles both manual and automatic control of the gimbal system using joystick input and sensor data. It begins by reading analog signals from the joysticks to determine user input for roll, pitch, and yaw angles. If joystick override is active, the system maps these inputs to servo positions, allowing direct control. For yaw, joystick 2 provides input when selected; otherwise, yaw is stabilized using gyroscope data. The code also prints raw input values for debugging. When no override is active, the system uses Kalman filter outputs to automatically stabilize roll and pitch. Finally, the calculated angles are sent to the servos, and a short delay ensures smooth updates.

C. *Evaluations from MATLAB*

INCLUDE 3-6 MATLAB DATA HERE

Fig. 11. Comparison of Kalman and complementary filter roll angle output for linear true rotation.

IV. Discussion and Recommendations

TALK ABOUT THE DEMO AND THE RESULTS AND FEEDBACK FOR FUTURE EXPERIMENTS

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.