Implementation and Evaluation of Orientation Estimation Filters on a 6-Axis IMU Using ESP32

Tofigh HOSSEINI

September 2025

Abstract

This project presents the implementation and comparative analysis of multiple orientation estimation filters on a 6-axis Inertial Measurement Unit (IMU) consisting of a 3-axis accelerometer and a 3-axis gyroscope. The filters implemented include a low-pass filter, complementary filter, Madgwick filter, Mahony filter, and an Extended Kalman Filter (EKF). The system is built on an ESP32 microcontroller, which performs real-time sensor data acquisition, filter processing, and orientation estimation. Processed orientation data is transmitted via UDP to a host computer for visualization and further evaluation. The performance of each filter is assessed in terms of stability, responsiveness, and robustness against noise and drift. The results provide practical insights into the trade-offs between computational cost, accuracy, and real-time feasibility of different orientation estimation algorithms on resource-constrained embedded systems.

1 Introduction

Inertial Measurement Units (IMUs) are widely used in robotics, drones, wearable devices, and various control systems for estimating orientation and motion. A typical 6-axis IMU consists of a 3-axis accelerometer and a 3-axis gyroscope, which together provide information about linear acceleration and angular velocity. However, due to inherent sensor limitations such as noise, bias drift, and sensitivity to dynamic motion, raw IMU data cannot be directly used for reliable orientation estimation. To address these challenges, sensor fusion algorithms and filtering techniques are applied to combine accelerometer and gyroscope data, thereby improving accuracy and robustness.

This project focuses on the implementation and evaluation of different orientation estimation filters on an ESP32 microcontroller using a 6-axis IMU. The filters implemented include a simple low-pass filter, a complementary filter, the Madgwick filter, the Mahony filter, and an Extended Kalman Filter (EKF). Each of these algorithms offers unique advantages: while low-pass and complementary filters are computationally efficient but less accurate, the Madgwick and Mahony filters provide fast convergence and robustness to noise, and the EKF offers a probabilistic framework for optimal state estimation at the expense of higher computational cost.

The ESP32 platform was selected due to its processing power, low cost, and built-in wireless communication capabilities, making it suitable for real-time embedded applications. Orientation estimates obtained from the filters are transmitted to a host computer over UDP for visualization and analysis. By comparing the performance of these filters in terms of accuracy, responsiveness, and computational efficiency, this work provides insights into the practical trade-offs involved in implementing IMU-based orientation estimation on resource-constrained hardware.

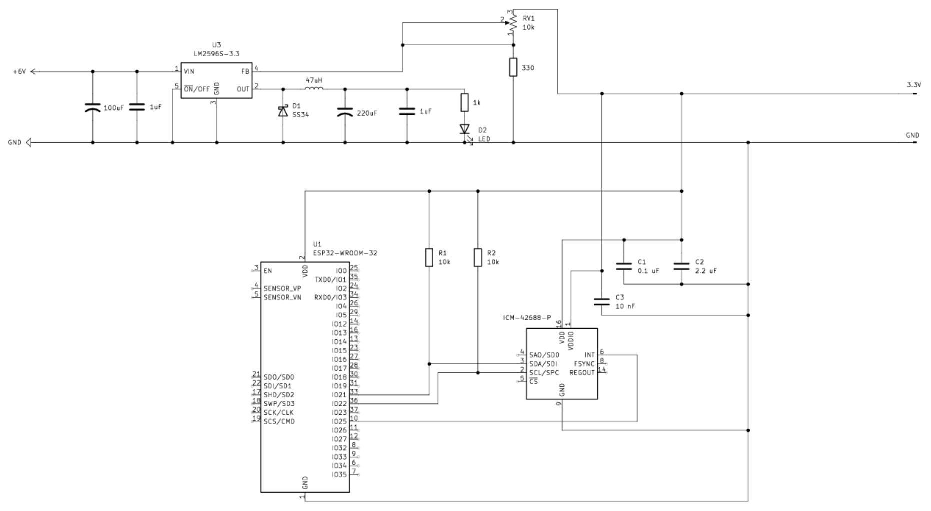

Figure 1: Circuit diagram of the hardware setup.

2 Hardware Setup

The hardware platform for this project consists of an ESP32 development board interfaced with a 6-axis Inertial Measurement Unit (IMU) via the I^{2}C communication protocol. The system is powered by a battery source regulated through a buck converter to provide stable operation at 3.3 V.

2.1 Power Supply

Four AA batteries connected in series are used as the primary power source, providing a nominal voltage of approximately 6 V. Since both the ESP32 and the IMU require 3.3 V for operation, the battery output is passed through a buck converter which regulates the voltage down to 3.3 V. The regulated output is then used to power both the ESP32 and the IMU sensor module. This setup ensures portability and reliable power delivery for real-time embedded processing.

2.2 ESP32 and IMU Connection

The ESP32 communicates with the IMU using the I^{2}C protocol. In this configuration, the ESP32 acts as the master device while the IMU functions as the slave device. The 3.3 V output from the buck converter powers both the ESP32 and the IMU, while the ground connections are shared across all components to maintain a common reference.

2.3 Circuit Diagram

The overall hardware connection is illustrated in Figure 1 and 2. The diagram shows the battery pack, buck converter, ESP32, and IMU connections via the I^{2}C interface.

2.4 I^{2}C Protocol and IMU Configuration

The I^{2}C protocol enables the ESP32 to communicate with the IMU using its unique 7-bit address. In this project, the I^{2}C bus is configured with a clock frequency of 500 kHz for fast data transfer. The IMU is initialized at startup by configuring its internal registers through I^{2}C commands. Specifically, the gyroscope full-scale range is set to $\pm 2000°/s$ while the accelerometer full-scale range is configured to $\pm 16\,g$, providing a good balance between sensitivity and noise

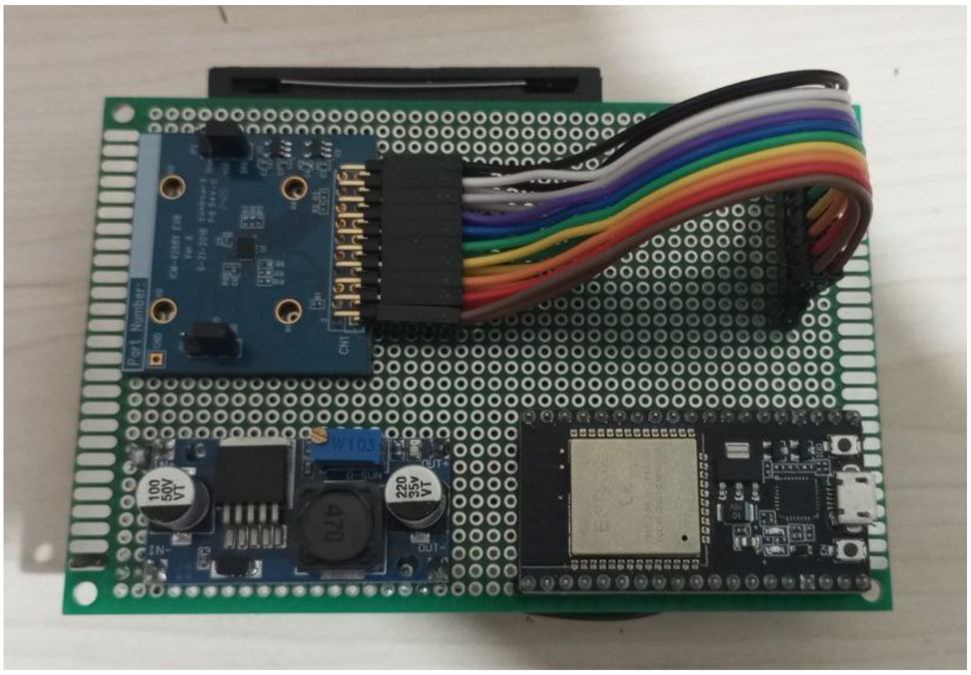

Figure 2: Real hardware setup.

tolerance. These configurations ensure reliable sensor measurements for the filtering algorithms implemented in this work.

2.5 I^{2}C Protocol

The Inter-Integrated Circuit (I^{2}C) protocol is a synchronous, multi-master, multi-slave, two-wire communication standard widely used in embedded systems. It requires only two lines:

- **SDA (Serial Data Line):** Bidirectional line used for data transmission.
- **SCL (Serial Clock Line):** Line driven by the master device to provide the clock signal.

Communication on the bus is initiated by the master device, which generates a *start condition*, followed by a 7-bit slave address and a read/write control bit. Each addressed slave device acknowledges the master by pulling the SDA line low during the acknowledgment cycle. Data transfer occurs in bytes, with each byte followed by an acknowledgment. The communication ends with a *stop condition* generated by the master.

In this project, the ESP32 functions as the master while the IMU module operates as the slave. The I^{2}C bus is configured at a clock speed of 500 kHz, enabling high-speed data acquisition. The protocol allows efficient register access within the IMU for configuration and measurement retrieval using only two signal lines, thereby reducing wiring complexity.

2.6 IMU Configuration

After establishing the I^{2}C connection between the ESP32 and the IMU, the next step is to configure the sensor registers. The configuration process involves setting the output data rate (ODR), measurement ranges for the gyroscope and accelerometer, and enabling the required sensor axes.

For example, the gyroscope sensitivity can typically be configured to values such as ± 250, ± 500, ± 1000, or ± 2000 dps. Similarly, the accelerometer can be set to ranges such as $\pm 2g$, $\pm 4g$, $\pm 8g$, or $\pm 16g$. A smaller range provides higher resolution for low-dynamic motion, while a larger range allows measurement of high-dynamic movements without saturation.

The configuration is performed by writing to specific IMU registers over the I^{2}C bus. For instance, setting the gyroscope range might involve writing to the `GYRO_CONFIG` register:

```
// Example: Configure gyroscope to ±500 dps
uint8_t config = 0x08;
i2cWrite(IMU_ADDR, GYRO_CONFIG, config);
```

Similarly, the accelerometer range can be set through the `ACCEL_CONFIG` register:

```
// Example: Configure accelerometer to ±4g
uint8_t config = 0x08;
i2cWrite(IMU_ADDR, ACCEL_CONFIG, config);
```

Once the sensitivity ranges are configured, the sampling rate and digital low-pass filters can be adjusted by writing to the appropriate control registers. For example, configuring a 1 kHz output data rate with a 98 Hz bandwidth low-pass filter ensures a balance between responsiveness and noise reduction.

After initialization, raw accelerometer and gyroscope data can be read by accessing the data registers in consecutive I^{2}C transactions:

```
i2cRead(IMU_ADDR, ACCEL_XOUT_H, buffer, 6); // Read accel values
i2cRead(IMU_ADDR, GYRO_XOUT_H, buffer, 6);  // Read gyro values
```

These raw measurements are later used as inputs to the implemented filtering algorithms.

3 Low-Pass Filter

The low-pass filter is designed to attenuate high-frequency noise in the gyroscope measurements while preserving the low-frequency dynamics that correspond to the actual motion of the system. The general form of the first-order low-pass filter applied to a signal $x(t)$ can be written as:

$$y[k] = (1-\alpha)\, y[k-1] + \alpha\, x[k]$$

where $y[k]$ is the filtered output, $x[k]$ is the raw input signal, and α is the filter coefficient. The value of α depends on the sampling interval T_s and the filter time constant τ:

$$\alpha = \frac{T_s}{T_s + \tau}, \quad \tau = \frac{1}{2\pi f_c}$$

with f_c denoting the cutoff frequency.

Code Implementation

In the implementation, the raw gyroscope data is filtered before being integrated into the orientation quaternion. A short excerpt is shown below:

```
// Apply low-pass filtering to gyroscope data
filtered_gx_ = (1.0f - alpha_) * filtered_gx_ + alpha_ * imu.gx;
filtered_gy_ = (1.0f - alpha_) * filtered_gy_ + alpha_ * imu.gy;
filtered_gz_ = (1.0f - alpha_) * filtered_gz_ + alpha_ * imu.gz;
```

The filtered angular rates are then used to update the quaternion representing the current orientation.

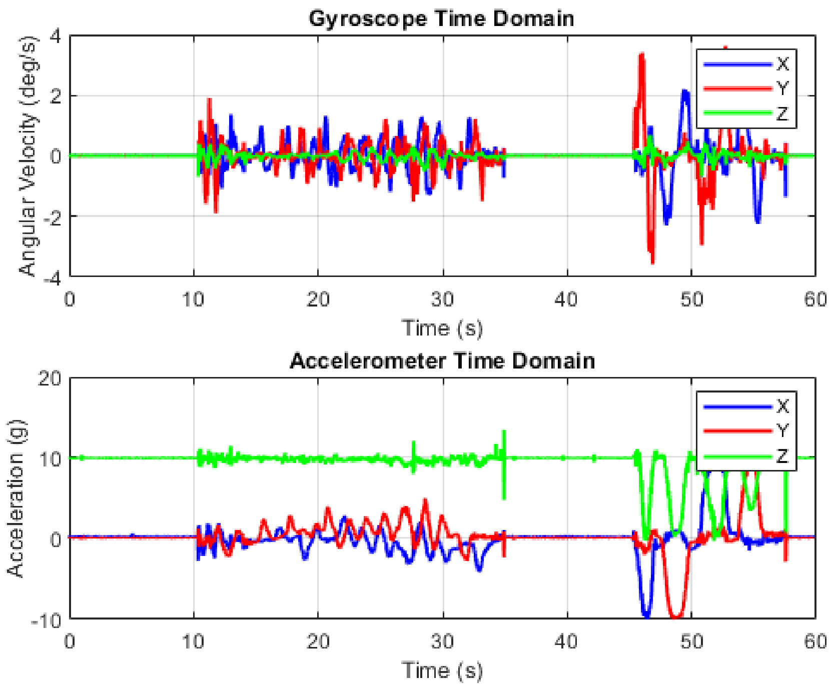

Figure 3: Raw IMU data plotted over 60s with slight movements.

Cutoff Frequency Selection

The cutoff frequency was determined empirically using frequency-domain analysis of the IMU data. Raw accelerometer and gyroscope signals were recorded in both stationary and lightly dynamic conditions. A Fast Fourier Transform (FFT) was then applied to the signals to inspect their spectral content. It was observed that the useful motion signals were concentrated at lower frequencies, while high-frequency components corresponded primarily to noise. The cutoff frequency f_c was chosen at the point where the magnitude spectrum exhibited a rapid decrease, ensuring that motion information was preserved while noise was attenuated. Figures 3 4 and is given two figures plotting the raw IMU data. Figure 4 shows the frequency spectrum of this data. Looking at this figure, one can select the cutoff frequency as 10Hz.

4 Complementary Filter

The complementary filter is a widely used method for fusing gyroscope and accelerometer measurements in order to estimate orientation. The key idea is that gyroscope integration provides good short-term accuracy but suffers from drift, while accelerometer readings provide a drift-free but noisy estimate of inclination. By blending these two sources of information, the complementary filter achieves a robust estimate of attitude.

The filter operates by applying a high-pass characteristic to the gyroscope data and a low-pass characteristic to the accelerometer data, then combining them. Conceptually:

$$\theta_{\text{est}} = \alpha\left(\theta_{\text{gyro}}\right) + (1-\alpha)\left(\theta_{\text{accel}}\right)$$

where α is a blending parameter determined by the cutoff frequency of the low-pass filter.

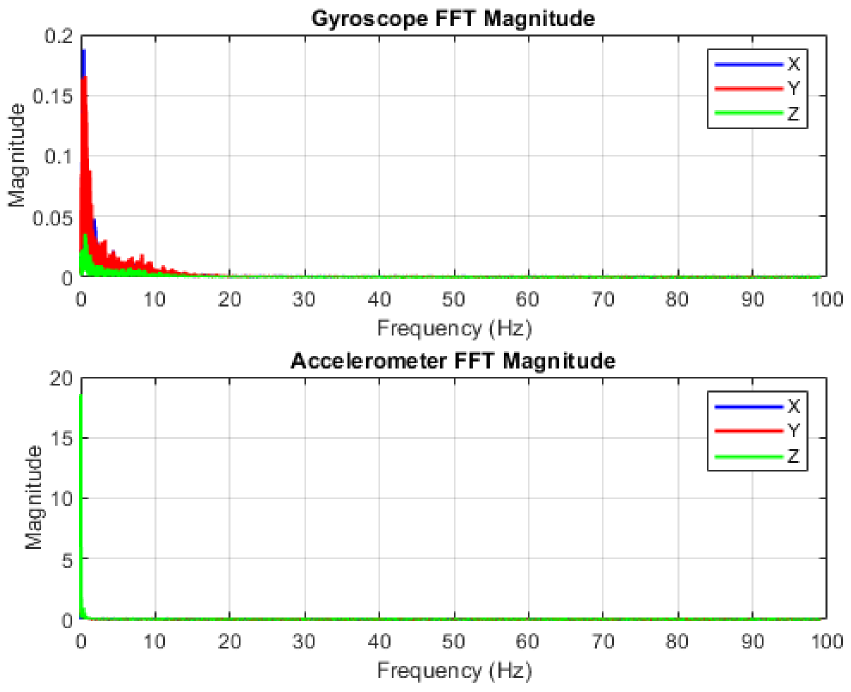

Figure 4: Frequency spectrum of raw IMU data.

Implementation

In this project, both accelerometer and gyroscope measurements are converted into quaternions. The gyroscope quaternion is updated by integrating angular rates, while the accelerometer quaternion is derived from the gravity vector. The two orientations are then blended together. For blending, the mathematically ideal approach is to use spherical linear interpolation (SLERP). However, SLERP requires trigonometric operations and is computationally more demanding. Since orientation updates between time steps are small in this application, a normalized linear interpolation (NLERP) provides nearly identical results with much lower computational cost.

An excerpt of the implementation is shown below:

```
// Low-pass filtered accelerometer vector is converted to quaternion
Quaternion q_accel = eul2quat(e_accel);

// Gyroscope quaternion update
Quaternion q_gyro = q_;
q_gyro = integrateGyro(q_gyro, imu.gx, imu.gy, imu.gz, dt);

// Blend using normalized linear interpolation
Quaternion q_fused = nlerp(q_gyro, q_accel, alpha_);
q_ = q_fused;
```

Here, `nlerp()` performs a linear interpolation between the two candidate orientations and then normalizes the result to ensure it remains a unit quaternion.

Discussion

The complementary filter provides a simple yet effective orientation estimation method. It ensures that short-term dynamics are captured from the gyroscope while long-term drift is corrected using the accelerometer. By using NLERP instead of SLERP, the implementation remains efficient for real-time execution on the ESP32 without significant loss of accuracy.

5 Madgwick Filter

The Madgwick filter is a nonlinear complementary filter designed to estimate orientation in quaternion form using only accelerometer and gyroscope measurements (and optionally magnetometer). The method combines gyroscope integration with a gradient descent correction step that aligns the estimated gravity direction with the measured acceleration vector.

5.1 Theory

Let the orientation of the sensor with respect to the inertial frame be represented by the quaternion

$$q = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T.$$

The time derivative of the quaternion due to angular velocity $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$ from the gyroscope is

$$\dot{q} = \frac{1}{2} q \otimes \begin{bmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix},$$

where $\otimes$ denotes quaternion multiplication. This corresponds to integrating the gyroscope signals directly, which suffers from drift due to bias.

To correct this drift, the accelerometer is used as a reference for the gravity vector. Given a normalized acceleration measurement

$$\mathbf{a} = \frac{1}{\|\mathbf{a}\|} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix},$$

the filter defines an objective function

$$\mathbf{f}(q) = \begin{bmatrix} 2(q_1 q_3 - q_0 q_2) - a_x \\ 2(q_0 q_1 + q_2 q_3) - a_y \\ 2(\frac{1}{2} - q_1^2 - q_2^2) - a_z \end{bmatrix},$$

which represents the difference between the estimated and measured gravity directions.

A gradient descent step is performed to minimize $\|\mathbf{f}(q)\|$, yielding a correction direction $\mathbf{s} = \nabla_q \mathbf{f}(q)$. The corrected quaternion derivative is

$$\dot{q} = \frac{1}{2} q \otimes \begin{bmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} - \alpha \mathbf{s},$$

where α is a tunable step size.

5.2 Implementation

In practice, the algorithm integrates the corrected quaternion derivative using the sampling period dt. A simplified excerpt from the implementation is shown below:

```
// Gradient descent correction in MadgwickFilter::update
// Normalize accelerometer
float a_norm = sqrt(ax*ax + ay*ay + az*az);
if (a_norm < 1e-6f) { skipUpdate(...); return; }
ax /= a_norm; ay /= a_norm; az /= a_norm;

// Objective function terms (gravity misalignment)
float f1 = 2.0f * (qx * qz - qw * qy) - ax;
float f2 = 2.0f * (qw * qx + qy * qz) - ay;
float f3 = 0.5f - qx*qx - qy*qy - az;

// Apply correction
dw -= alpha_ * s0;
dx -= alpha_ * s1;
// ...
```

5.3 Discussion

The Madgwick filter is known for:

- Low computational cost (suitable for embedded systems).
- Fast convergence due to gradient descent correction.
- Sensitivity to the tuning of α: too high causes noise amplification, too low slows convergence.

It is particularly effective for real-time orientation estimation in systems without magnetometers.

6 Mahony Filter

The Mahony filter is another nonlinear complementary filter based on proportional-integral (PI) feedback control. It uses the accelerometer to correct drift in gyroscope integration, with the integral term compensating for long-term biases.

6.1 Theory

The quaternion dynamics due to gyroscope signals are again given by

$$\dot{q} = \frac{1}{2} q \otimes \begin{bmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

To correct for drift, the accelerometer provides a reference for the gravity vector. Let the normalized acceleration be

$$\mathbf{a} = \frac{1}{\|\mathbf{a}\|} \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}.$$

From the current quaternion estimate q, the estimated gravity vector in the sensor frame is

$$\mathbf{v} = \begin{bmatrix} 2(q_1 q_3 - q_0 q_2) \\ 2(q_0 q_1 + q_2 q_3) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}.$$

The error between the measured and estimated gravity vectors is computed as the cross product

$$\mathbf{e} = \mathbf{a} \times \mathbf{v}.$$

This error is then fed into a PI controller:

$$\boldsymbol{\omega}_{\text{corrected}} = \boldsymbol{\omega} + K_p\mathbf{e} + K_i \int \mathbf{e}\, dt,$$

where K_p is the proportional gain and K_i is the integral gain.

6.2 Implementation

The corrected angular velocity is used in the quaternion integration. A simplified implementation excerpt is shown below:

```
// Error between measured and estimated gravity
float ex = (ay * vz - az * vy);
float ey = (az * vx - ax * vz);
float ez = (ax * vy - ay * vx);

// PI feedback
iBx += Ki_ * ex * dt;
iBy += Ki_ * ey * dt;
iBz += Ki_ * ez * dt;

gx += iBx + Kp_ * ex;
gy += iBy + Kp_ * ey;
gz += iBz + Kp_ * ez;
```

6.3 Discussion

Key characteristics of the Mahony filter:

- Explicit PI structure allows tuning for stability vs. responsiveness.
- Integral term compensates for constant gyroscope biases.
- More robust than Madgwick in noisy accelerometer environments.

However, the filter may converge more slowly than Madgwick if K_p and K_i are not properly tuned.

7 Extended Kalman Filter (EKF)

The Kalman Filter (KF) is an optimal recursive estimator for linear systems with Gaussian noise. It combines prior state predictions with noisy sensor measurements to produce statistically optimal state estimates. The Extended Kalman Filter (EKF) generalizes this framework to nonlinear systems by linearizing the process and measurement models around the current estimate.

7.1 State Definition

In this project, the state vector is chosen as

$$\mathbf{x} = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 & b_x & b_y & b_z \end{bmatrix}^T,$$

where $q = [q_0, q_1, q_2, q_3]^T$ is the orientation quaternion, and $\mathbf{b} = [b_x, b_y, b_z]^T$ is the gyroscope bias.

7.2 Process Model

The quaternion evolves according to gyroscope angular velocity $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$:

$$\dot{q} = \tfrac{1}{2}\,\Omega(\boldsymbol{\omega} - \mathbf{b})q,$$

where $\Omega(\boldsymbol{\omega})$ is the quaternion kinematic matrix

$$\Omega(\boldsymbol{\omega}) = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix}.$$

The bias is modeled as constant:

$$\dot{\mathbf{b}} = \mathbf{0}.$$

Discretizing with timestep dt:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k,$$

where $\mathbf{w}_k \sim \mathcal{N}(0, Q)$ is process noise.

7.3 Measurement Model

The accelerometer provides a measurement of gravity in the sensor frame:

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k,$$

where

$$h(q) = R(q)^T\mathbf{g}, \quad \mathbf{g} = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}.$$

Here $R(q)$ is the rotation matrix from quaternion q, and $\mathbf{v}_k \sim \mathcal{N}(0, R)$ is measurement noise.

7.4 Covariance and Gain Matrices

- P : State covariance matrix (uncertainty in the estimate).
- Q : Process noise covariance (model uncertainty, gyro noise).
- R : Measurement noise covariance (accelerometer noise).
- F : Jacobian of the process dynamics $\frac{\partial f}{\partial x}$.
- H : Jacobian of the measurement model $\frac{\partial h}{\partial x}$.
- K : Kalman gain, balancing prediction and measurement.

7.5 EKF Algorithm

The EKF proceeds in two steps:

Prediction

$$\mathbf{x}_{k|k-1} = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1})$$

$$P_{k|k-1} = F_k P_{k-1} F_k^T + G_k Q G_k^T$$

Update

$$\mathbf{y}_k = \mathbf{z}_k - h(\mathbf{x}_{k|k-1})$$

$$S_k = H_k P_{k|k-1} H_k^T + R$$

$$K_k = P_{k|k-1} H_k^T S_k^{-1}$$

$$\mathbf{x}_k = \mathbf{x}_{k|k-1} + K_k \mathbf{y}_k$$

$$P_k = (I - K_k H_k) P_{k|k-1}$$

Finally, the quaternion is normalized to ensure it remains a valid rotation.

7.6 Implementation

An excerpt from the project implementation is shown below.

```
// Time update (prediction)
void ExtendedKF::timeUpdate(const ImuData& imu, float dt) {
BLA::Matrix<3,1> w = {imu.gx, imu.gy, imu.gz};
w = w - b;
// Quaternion derivative
BLA::Matrix<4,4> omega = {...};
BLA::Matrix<4,1> qdot = 0.5f * omega * q;

// State propagation
x = x + f * dt;
normalizeQuaternion();

// Covariance propagation
P = F * P * ~F + G * Q * ~G;
}

// Measurement update
void ExtendedKF::measurementUpdate(const ImuData& imu, float dt) {
BLA::Matrix<3,1> z = {imu.ax, imu.ay, imu.az};
BLA::Matrix<3,1> y = z - ~R * g_vec; // residual

BLA::Matrix<3,7> H = {...};          // measurement Jacobian
BLA::Matrix<3,3> S = H * P * ~H + R; // innovation covariance
K = P * ~H * Inverse(S);             // Kalman gain

x = x + K * y;                       // state update
normalizeQuaternion();
P = (I - K * H) * P;                 // covariance update
}
```

7.7 Discussion

The EKF provides a statistically grounded, recursive estimation framework that:

- Handles nonlinear dynamics via local linearization.
- Estimates both orientation and gyroscope bias simultaneously.
- Weighs prediction and measurement adaptively using covariance.

While more computationally demanding than complementary or Madgwick/Mahony filters, the EKF is more general, robust, and interpretable, making it a gold standard in state estimation.

8 UDP Communication and Data Transmission

In this project, the **ESP32** transmits filtered orientation data from the IMU to a computer over WiFi using the **User Datagram Protocol (UDP)**. UDP is a lightweight, connectionless communication protocol built on top of the IP layer. Unlike TCP, it does not establish a connection before sending data and does not guarantee delivery, ordering, or error correction. However, this simplicity makes UDP highly efficient and suitable for real-time applications such as streaming sensor data, where low latency is preferred over guaranteed reliability.

8.1 UDP Communication Principle

UDP communication involves three essential components:

- **Sender (ESP32):** Packages IMU and filter data into structured packets and transmits them over WiFi.
- **Receiver (PC):** Listens on a specific IP address and port, decoding and visualizing incoming data.
- **Protocol structure:** Each packet contains a fixed header, sequence number, timestamp, and multiple floating-point values representing orientation data.

The ESP32 connects to a WiFi network and initializes a UDP socket for communication using the Arduino `WiFiUDP` library:

```
#include <WiFi.h>
WiFiUDP udp;
const char* ssid = "*********";
const char* password = "*************";
const char* hostIP = "192.168.1.21";
const int udpPort = 3333;

void setup() {
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
delay(500);
Serial.print(".");
}
Serial.println("Connected to WiFi");
udp.begin(1234);
}
```

8.2 Packet Structure and Filter Configuration

Each UDP packet sent by the ESP32 has a fixed layout, ensuring consistent parsing on the receiver side:

- 1 byte — Packet header (`0xAA`)
- 4 bytes — Sequence number
- 4 bytes — Timestamp (`micros()`)

- 8 floats (32 bytes) — Two quaternions representing the outputs of two filters

Thus, each transmitted packet has a total size of `41 bytes`. Before starting the transmission, the PC sends a configuration byte indicating which filters should run on the ESP32. The high nibble corresponds to the first filter, and the low nibble to the second filter. Upon receiving the configuration, the ESP32 acknowledges it by replying with `0x55`.

```
void waitForConfig() {
uint8_t buf[2];
while(true) {
int len = udp.parsePacket();
if (len == 2) {
udp.read(buf, 2);
if (buf[0] == 0xAA) {
selectFilters(buf[1]);
udp.beginPacket(udp.remoteIP(), udp.remotePort());
udp.write(0x55); // ACK
udp.endPacket();
break;
}
}
delay(10);
}
}
```

8.3 Sending Filtered Data via UDP

Once the filter configuration is set and IMU data becomes available, the ESP32 continuously reads sensor data, updates both selected filters, and transmits the results over UDP.

The following code snippet shows the core data transmission logic:

```
void loop() {
if (!imu.available()) return;
unsigned long now = micros();
ImuData d = imu.readScaled();
float dt = (now - lastMicros) / 1e6f;
lastMicros = now;

filter1->update(d, dt);
filter2->update(d, dt);

Quaternion q1 = filter1->getQuaternion();
Quaternion q2 = filter2->getQuaternion();

uint8_t packetBuf[SAMPLE_BYTES];
size_t offset = 0;

packetBuf[offset++] = 0xAA;
memcpy(packetBuf + offset, &seq, sizeof(seq)); offset += 4;
uint32_t ts = (uint32_t)(micros() & 0xFFFFFFFFUL);
memcpy(packetBuf + offset, &ts, sizeof(ts)); offset += 4;

memcpy(packetBuf + offset, &q1, sizeof(q1)); offset += sizeof(q1);
memcpy(packetBuf + offset, &q2, sizeof(q2)); offset += sizeof(q2);

IPAddress remoteIP;
remoteIP.fromString(hostIP);
udp.beginPacket(remoteIP, udpPort);
udp.write(packetBuf, SAMPLE_BYTES);
udp.endPacket();
seq++;
}
```

8.4 Advantages of Using UDP

The main advantages of using UDP in this project are:

- **Low latency:** UDP introduces minimal transmission overhead, allowing real-time streaming of IMU data.
- **Simplicity:** No connection management or retransmission handling is required.
- **Flexibility:** Easy integration with Python or MATLAB scripts for visualization and analysis.

In summary, UDP provides an ideal balance between speed and simplicity for transmitting orientation data from the ESP32 to a host computer, enabling fast and responsive visual feedback during filter testing.

9 Practical Demonstration and Discussion

In the accompanying video, the implemented filters are shown running live on IMU data. One important observation is the behavior of the **yaw angle**.

Since this implementation only uses the accelerometer and gyroscope, the yaw (heading) is **unobservable**. The accelerometer provides a reference to the gravity vector, which allows correction of roll and pitch. However, gravity carries no information about the rotation around the vertical axis. As a result, any small bias or noise in the gyroscope accumulates over time, causing the yaw to **drift**.

This effect is not caused by an error in the code, but is instead a **fundamental limitation** of the chosen sensor setup. Without an external reference, such as a magnetometer, GPS, or visual odometry, the yaw angle will always remain unbounded in the long term.

In the demonstration video, this effect is clearly visible: while roll and pitch stabilize well due to gravity, the yaw gradually drifts away. Possible extensions to the system include adding a magnetometer to make yaw observable, or integrating additional sensor modalities for more robust orientation tracking.