Kalman Filter Implementation
AE 5621 – GNC

Max Heil, Kyle Frith

December 2, 2024

Introduction

The drone industry has experienced exponential growth in the past two decades, primarily driven by advancements in aerospace technology. The Federal Aviation Administration (FAA) has eased restrictions on controlled airspaces for drones, potentially paving the path for revolutionizing industries that utilize drones. Agriculture, logistics, and surveillance drone usage have skyrocketed since this gradual change. Among these developments, small quadrotors have emerged as market leaders due to their affordability and versatility, despite challenges in flight performance.

At the core of a successfully performing drone lies the need for accurate motion and acceleration measurement, which heavily relies on an efficient and cost-effective Inertial Measurement Unit (IMU). IMUs have decreased in cost significantly while maintaining high-performance stability integration. Paired with a microcontroller, such an Arduino, these IMUs hold significant promise for improving the flight stability and maneuverability of drones.

The focus of this project is to implement a Kalman Filter to enhance the performance of an IMU by filtering noise and providing accurate estimates of pitch and roll angles. The Kalman Filter is widely recognized in the aerospace industry as a leader in real-time system identification and addressing inaccuracies of cheaper IMUs. By using MATLAB to integrate this filter into a BNO055 9-axis IMU, this project aims to demonstrate improved motion tracking in various pitch and roll maneuvers.

Hardware Description

The hardware provided for this project consists of the following:

1. Arduino UNO Rev3
2. BNO-055 9-axis IMU
3. Mini breadboard
4. 4 jumper wires (for wiring setup)

The Arduino UNO is a popular microcontroller board based on the ATmega328P microcontroller. It is widely used for electronics projects and prototyping due to its simplicity and versatility. For this project, it will be useful for interfacing the IMU with computer software. It is powered through a USB connection to a computer. The USB connection is also used for programming and serial communications. There are several standard 0.1-inch pitch headers for easy connection. The board is configured for male pins but can use stacking headers for shields. There is also an ICSP header that allows for direct programming of the board using an In-Circuit Serial Programmer. Several LED indicators on the board allow the user to tell whether it is powered on and functioning properly.

The BNO-055 9-axis IMU is integrated into the Arduino board to allow for sensor fusion. It combines a 3-axis gyroscope, 3-axis accelerometer, and 3-axis magnetometer, along with an ARM Cortex-M0 processor to perform real-time sensor fusion and communicate orientation data. The device combines raw motion sensing with onboard sensor fusion algorithms to deliver a mostly ready-to-use output. However, as with many cheap IMUs, sensor noise is visible when monitoring the system. Thus, it is important that noise filtering algorithms are implemented on the software side to deliver drift-compensated outputs to the user.

The main purpose of the breadboard is simply to connect the IMU and the Arduino boards. The mini breadboard completes the setup and allows the testbed to function properly. The jumper wires are used to connect the breadboard to the Arduino board. The IMU is soldered to the breadboard using pins.

Figure 1 shows the completed setup that includes the IMU, Arduino, and breadboard.

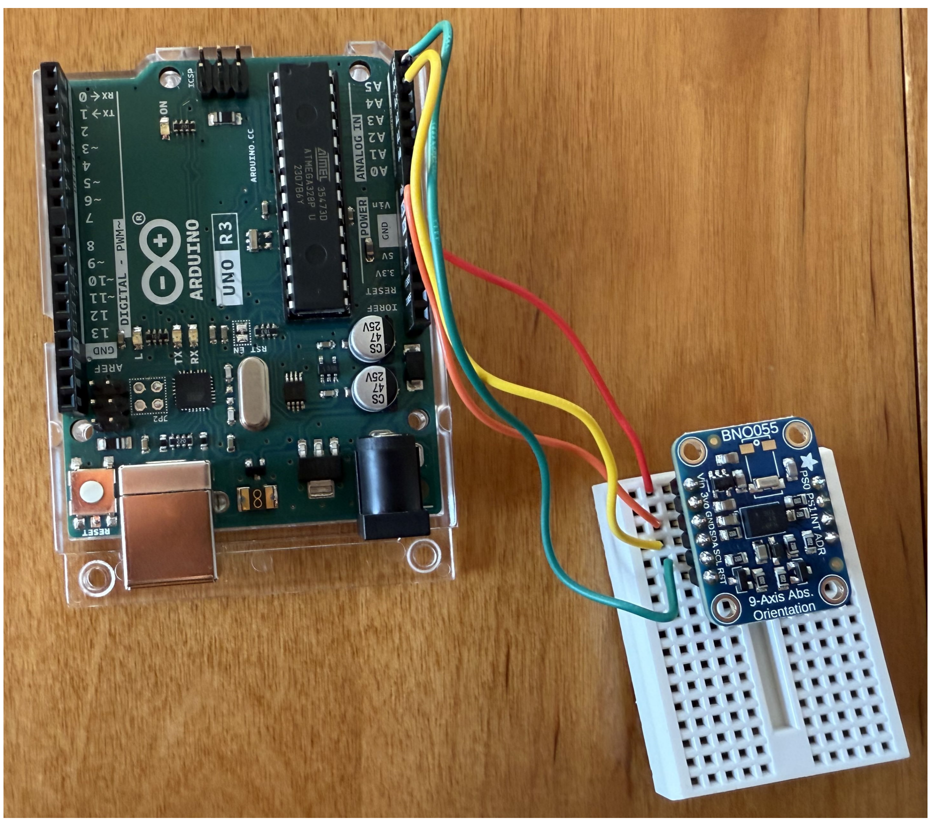

Figure 1: Wired Arduino board and IMU for full hardware setup

Additionally, Figure 2 displays the required wiring diagram that was used to complete the setup above.

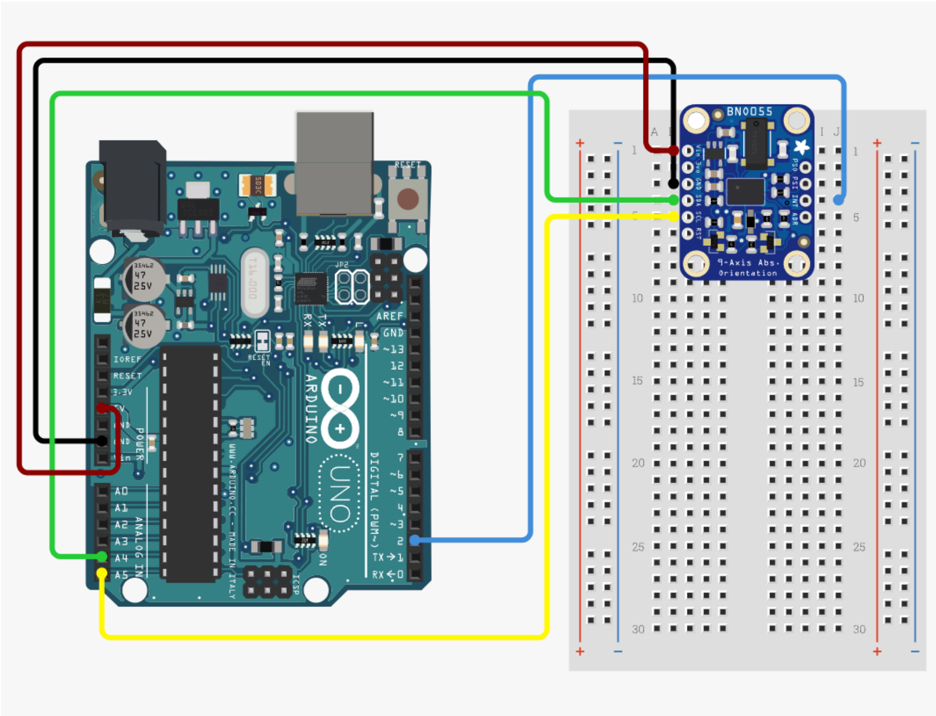

Figure 2: Full wiring diagram for hardware setup

The ports used on the Arduino board are A5, A4, GND, and 5V. These were connected to SCL, SDA, GND, and Vin on the IMU board respectively. The 5V pin provides the power supply to the voltage input in the IMU and the ground pins on each board are simply ground wires. I2C communication is done through the A4 pin on the Arduino which is connected to the SDA or Serial Data pin on the sensor. The A5 pin from the Arduino connects to the Serial Clock (SCL) pin on the sensor to allow for smooth data exchange. This setup is quite simple, yet effective when using the Arduino and IMU provided.

Theory and Implementation

A Kalman Filter is an effective state estimator for a process by predicting the future state based on the measurement of the prior state while accounting for error in measurement and sensing. The filter that is implemented in this work is an Extended Kalman Filter (EKF) which offers state estimation for non-linear systems. Reliance on Euler angle computations from gyroscope and accelerometer readings can lead to computational errors and undetermined states. The EKF will be computed using quaternions which avoid any singularities in the computation.

The procedure of the filter is a recursive loop that consists of projecting the future state, projecting the future error covariance, updating the estimate, and finally updating the error covariance. Future state prediction is given by

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, w_{k-1}) \tag{1}$$

where $\hat{x}_k^-$ is the projected state, $\hat{x}_{k-1}$ is the previous state measurement, u_{k-1} is the previous measured input, and w_{k-1} is random measurement noise from the previous step. For our purposes, w_{k-1} cannot be known, thus our new state prediction is

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2}$$

Reading of the gyroscope rates allows input to the system for Euler angles ϕ, θ, ψ. Input from the gyros is given as:

$$\dot{\phi} = p + q \sin\psi \tan\theta + r \cos\psi \tan\theta \tag{3}$$

$$\dot{\theta} = q \cos\psi - r \sin\psi \tag{4}$$

$$\dot{\psi} = q \sin\psi \sec\theta + r \cos\psi \sec\theta \tag{5}$$

P, q, r can be expressed as $\omega_1, \omega_2, \omega_3$ respectively. Equations (3)–(5), when expressed in quaternion matrix form, become the following:

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & \omega_3 & -\omega_2 & \omega_1 \\ -\omega_3 & 0 & \omega_1 & \omega_2 \\ \omega_2 & -\omega_1 & 0 & \omega_3 \\ -\omega_1 & -\omega_2 & -\omega_3 & 0 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{6}$$

Discretizing by derivative approximation, combining equations (2) and (6) yields the state predictor

$$\hat{x}_k^- = \begin{bmatrix} q_{1k} \\ q_{2k} \\ q_{3k} \\ q_{4k} \end{bmatrix} = \frac{h}{2} \begin{bmatrix} 2/h & \omega_3 & -\omega_2 & \omega_1 \\ -\omega_3 & 2/h & \omega_1 & \omega_2 \\ \omega_2 & -\omega_1 & 2/h & \omega_3 \\ -\omega_1 & -\omega_2 & -\omega_3 & 2/h \end{bmatrix} \begin{bmatrix} q_{1k-1} \\ q_{2k-1} \\ q_{3k-1} \\ q_{4k-1} \end{bmatrix} \tag{7}$$

Covariance projection is given by

$$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{8}$$

where P_k^- is the projected covariance, A_k is a Jacobian matrix with respect to x, W_k is a Jacobian matrix with respect to w_k, and Q_{k-1} is process noise covariance. Because w_k is unknown, W_k will be assumed to be an identity matrix; Q_{k-1} will also be assumed to be constant and an identity matrix with an attached scalar ~ 0.0001 to account for the accuracy of the sensors. Equation (8) thus becomes

$$P_k^- = A_k P_{k-1} A_k^T + Q \tag{9}$$

The Jacobian A_k in quaternion format taken with respect to ω yields the same result as equation (6).

$$A_k = \frac{1}{2}\begin{bmatrix} 0 & \omega_3 & -\omega_2 & \omega_1 \\ -\omega_3 & 0 & \omega_1 & \omega_2 \\ \omega_2 & -\omega_1 & 0 & \omega_3 \\ -\omega_1 & -\omega_2 & -\omega_3 & 0 \end{bmatrix} \tag{10}$$

Kalman gain can be calculated from

$$K_k = P_k^- H_k^T \left(H_k P_k^- H_k^T + V_k R_k V_k^T\right)^{-1} \tag{11}$$

For our project, V_k and H_k are assumed to be identity matrices. The new form for Kalman gain is:

$$K_k = P_k^- \left(P_k^- + R_k\right)^{-1} \tag{12}$$

where R_k is the covariance of the measurement sensors. The covariance of the measurement sensors was calculated by allowing the IMU to lay flat on a table with no motion for approximately 3 minutes and using MATLAB to calculate the covariance of each sensor. Table 1 presents the results of these calculations.

Sensor	**Covariance**
Accel x	1.732×10^{-7}
Accel y	1.327×10^{-7}
Accel z	4.473×10^{-7}
Gyro x	4.125×10^{-7}
Gyro y	6.884×10^{-7}
Gyro z	4.011×10^{-7}

Table 1: Covariance calculations from sensors

Once the Kalman gain is computed, the update portion of the filter can start. The equation for the state update is given by

$$\hat{x}_k = \hat{x}_k^- + K_k\big(z_k - h(\hat{x}_k^-, 0)\big) \tag{13}$$

where z_k is a measurement provided by a sensor. The measurement will be provided by the accelerometers of the IMU, and Euler angles can be calculated as

$$\theta = \sin^{-1}\left(\frac{a_x}{g}\right) \tag{14}$$

$$\phi = \tan^{-1}\left(\frac{a_y}{a_z}\right) \tag{15}$$

$$\psi = 0 \tag{16}$$

Due to the sensitive nature of the onboard magnetometer, the measurement of ψ was consistently 0. When expressed in quaternion form, the measurement matrix becomes

$$z_k = \begin{bmatrix} q_{1k} \\ q_{2k} \\ q_{3k} \\ q_{4k} \end{bmatrix} = \begin{bmatrix} \sin\frac{\phi}{2}\cos\frac{\theta}{2} \\ \cos\frac{\phi}{2}\sin\frac{\theta}{2} \\ -\sin\frac{\phi}{2}\sin\frac{\theta}{2} \\ \cos\frac{\phi}{2}\cos\frac{\theta}{2} \end{bmatrix} \tag{17}$$

Update to error covariance is then given by:

$$P_k^- = (I - K_k)\, P_k^- \tag{18}$$

Once the state update is performed, a direction cosine matrix (DCM) can be constructed from new quaternions and the new best estimate for Euler angles:

$$\phi = \tan^{-1}\left(\frac{2(q_1q_2 + q_3q_4)}{q_1^2 - q_2^2 - q_3^2 + q_4^2}\right) \tag{19}$$

$$\theta = \sin^{-1}\left(\frac{-2(q_1q_3 + q_2q_4)}{\sqrt{(q_1^2 - q_2^2 - q_3^2 + q_4^2)^2 + (2(q_1q_2 + q_3q_4))^2}}\right) \tag{20}$$

$$\psi = \tan^{-1}\left(\frac{2(q_2q_3 + q_1q_4)}{q_3^2 - q_1^2 - q_2^2 + q_4^2}\right) \tag{21}$$

Experimental Results

The IMU was rotated from -30° to 30° for 10 seconds in only roll, only pitch, and combined roll and pitch maneuvers. Figures 3–5 present the results of the testing.

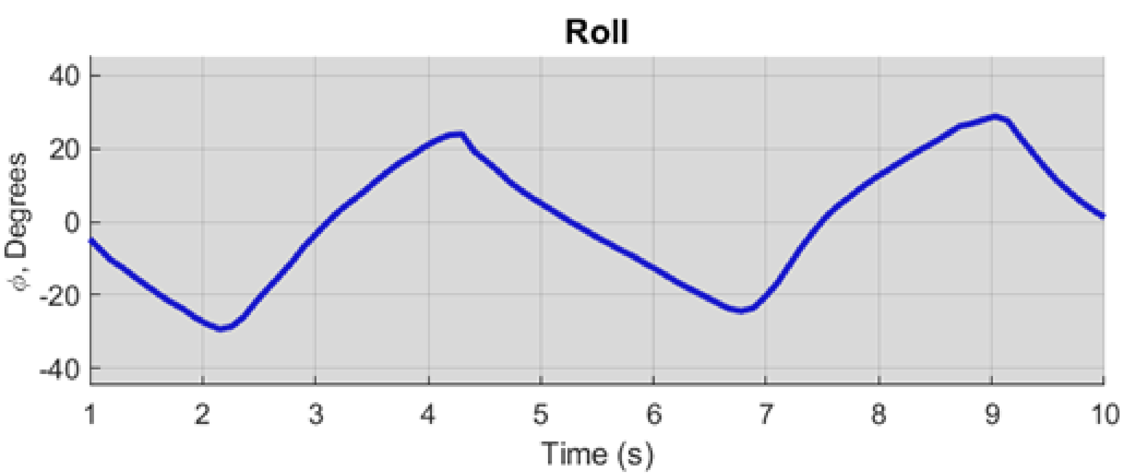

Figure 3: Roll from -30 degrees to 30 degrees, 10 seconds

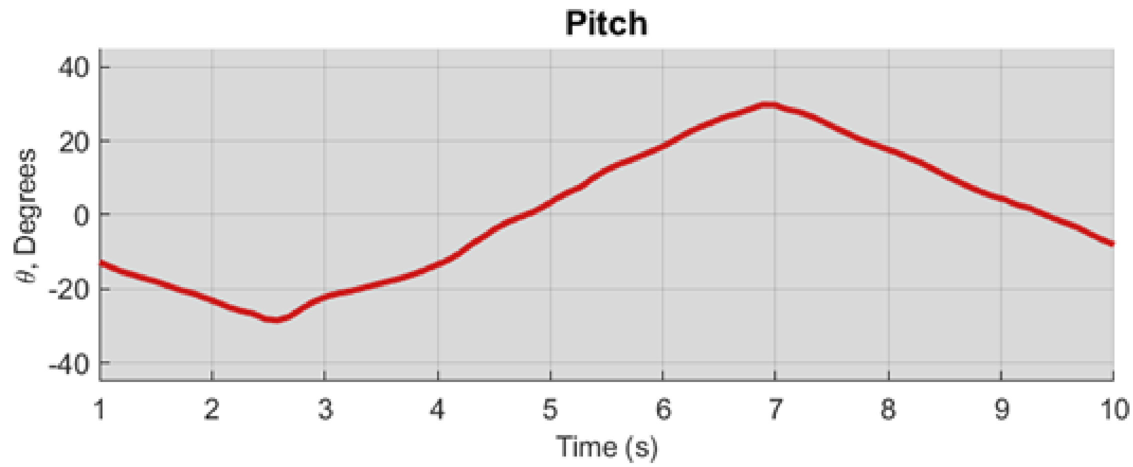

Figure 4: Pitch from -30 degrees to 30 degrees, 10 seconds

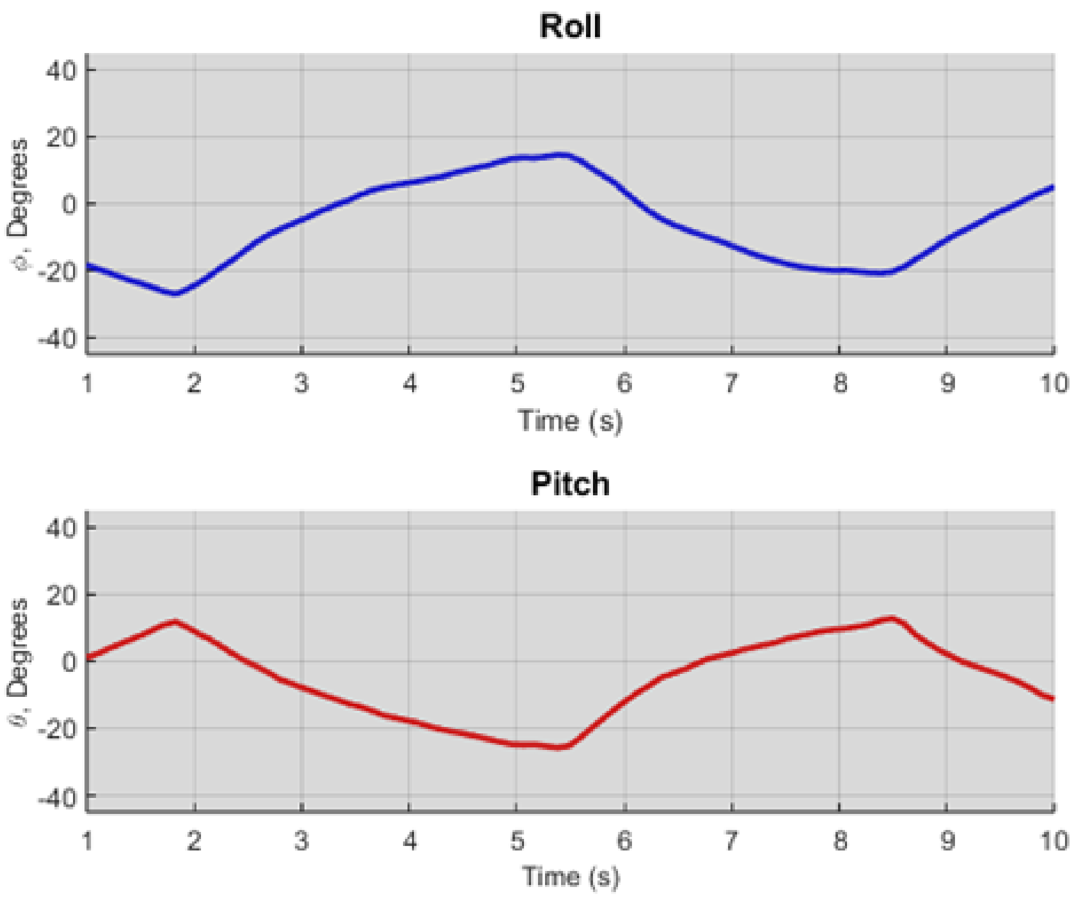

Figure 5: Pitch and Roll from -30 degrees to 30 degrees, 10 seconds

Clearly, the filter performed excellent state estimation and was quick to respond to gradual changes in angle and low frequencies. From Figure 5 with combined maneuvering, the filter performed well, and the results are similar to both Figures 3 and 4. Next, the same tests were performed over a test period of 2 seconds. Figures 6–8 present the results.

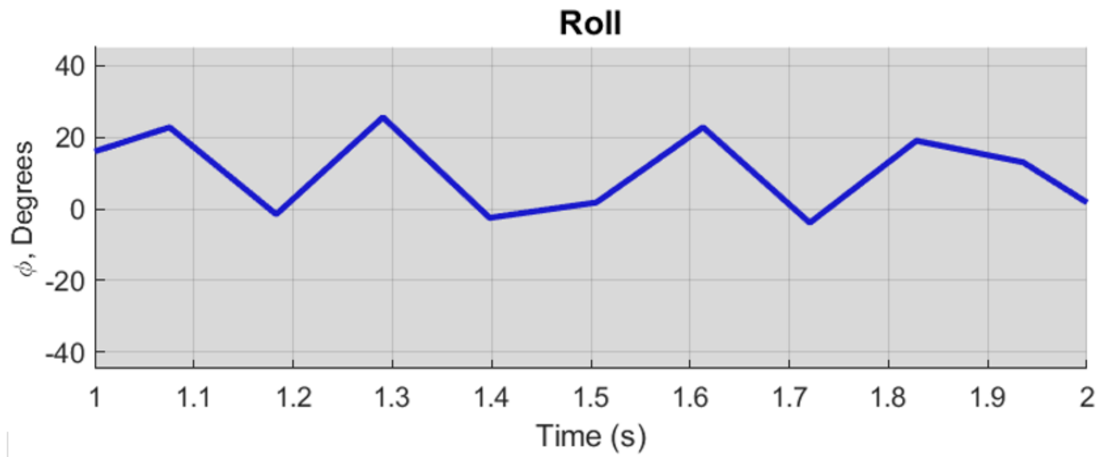

Figure 6: Roll from -30 degrees to 30 degrees, 2 seconds

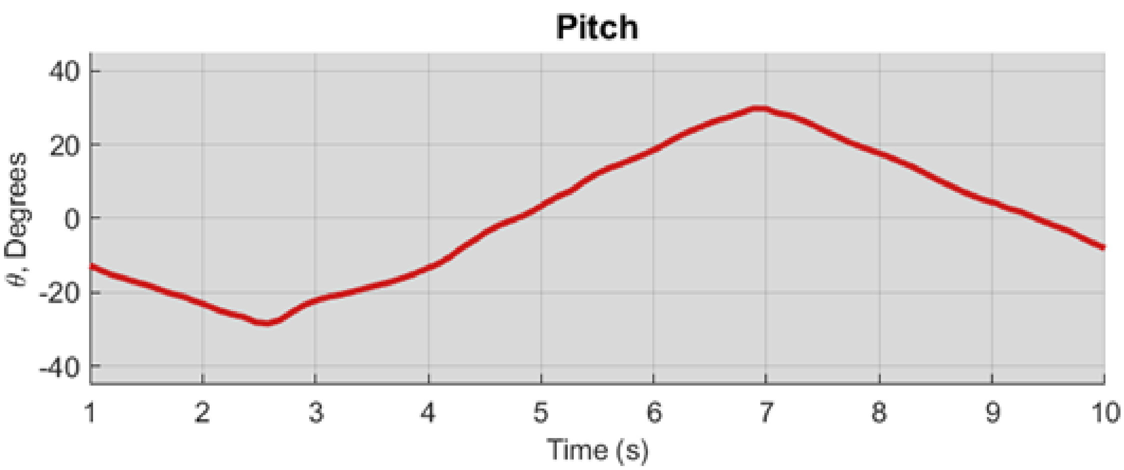

Figure 7: Pitch from -30 degrees to 30 degrees, 2 seconds

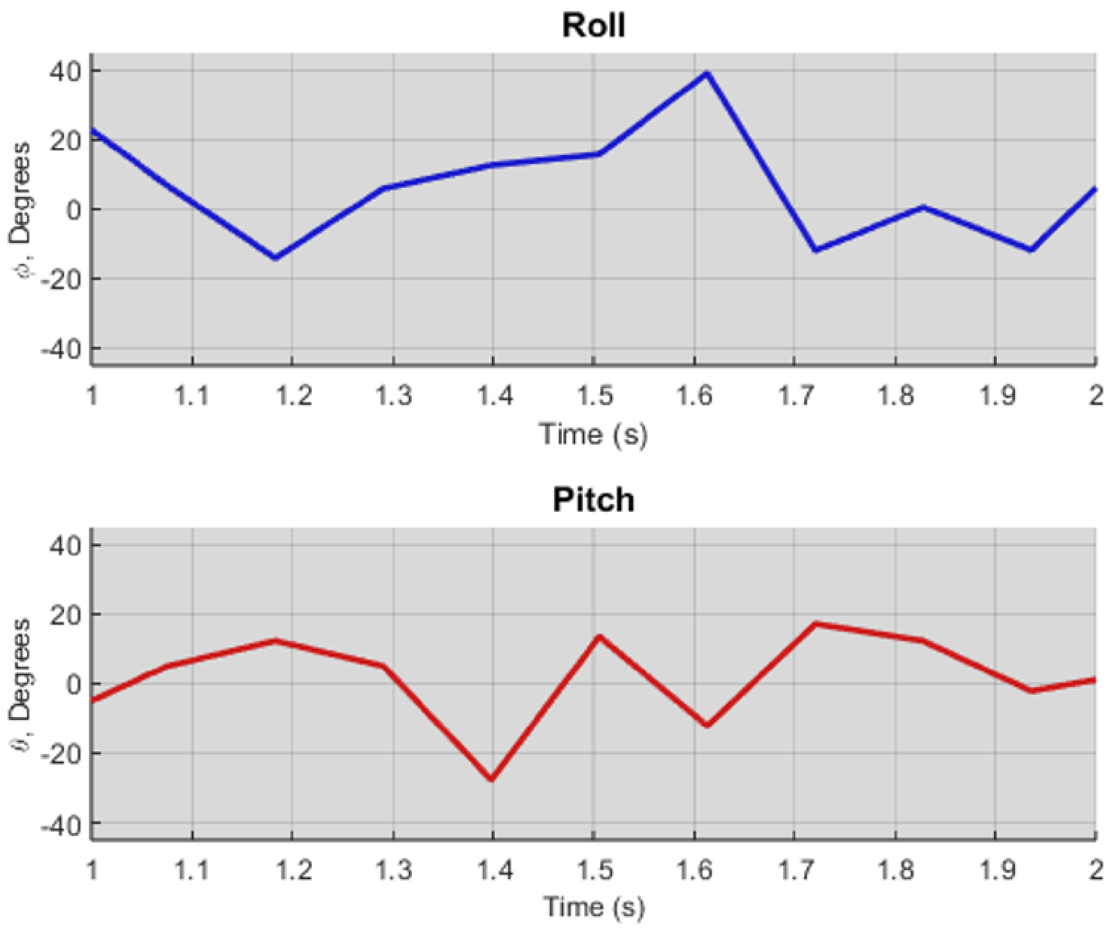

Figure 8: Pitch and Roll from -30 degrees to 30 degrees, 2 seconds

From Figure 6 it is seen that for high frequency oscillation, the filter was not as responsive, producing much more angular lines when roll response was plotted against time. The corresponding low frequency maneuver from Figure 3 was much smoother with rounder peaks. Similarly, from Figure 7 it is seen that pitch estimation also produced angular plots. Combined maneuvering produced similar results.

Conclusion

Implementation of an Extended Kalman Filter (EKF) utilizing an inexpensive IMU and Arduino-based microcontroller is an effective state estimator for certain frequency maneuvers. Through the filtering of sensor noise and accurate estimation of pitch and roll angles, the EKF provides a practical solution for enhancing overall flight stability.

Experimental results showed that the EKF effectively handled low-frequency maneuvers, maintaining smooth and accurate responses for both pitch and roll tests. However, in cases where high frequency of oscillation maneuvers is required around the pitch and roll, the EKF and IMU were slow to respond. This suggests that the current IMU's sampling rate may constrain its performance in highly dynamic scenarios. To better the performance under these conditions, a more robust IMU utilizing a higher sampling rate would be required. Future improvements to the project could involve expanding the scope to include yaw estimation and finding better tuning parameters of the EKF to further refine the system's robustness and accuracy.

In conclusion, this project highlights the viability of using a low-cost Arduino and IMU in combination with a filtering system to efficiently and accurately estimate future states. It also presents the challenges of using such a system which include limited capabilities and output delay when performing high-frequency maneuvers. Ultimately, this kind of project paves the way for broader accessibility and applicability of drone technology in various industries.

Appendix: MATLAB Code

Listing 1: kalmanfilterintegration.m

```
% Plug in arduino first, be sure matlab recognizes it
% evaluate each line at a time
% Part 1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear; clc; close all;

a = arduino;     % Connect Arduino to Matlab
imu = bno055(a);  % Interface BNO-055 IMU with Arduino
status = readCalibrationStatus(imu)   ; % Calibrate IMU
tconversion = 53.76839/500;

%%%%% Desired Duration %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% for 10s duration, uncomment these
% imu_lim = 94;
% tlim = 10;

% for 30s duration uncomment these
imu_lim = 280;
tlim = 30;

% for const streaming, no time limit
% imu_lim = 1000000;
% tlim = 1000000;


% system declarations / initialization

h = 0.01;  % sample rate
Q = 0.001*eye(4);  % sensor noise estimation
x = getquaternions(0,0) ; % Initialization of quaternions
P = 0*eye(4)  ;  % Initial P val
R = 0.001*eye(4) ;  % Covariance of measurment sensor
pitch = [];  %
roll = [];
H = eye(4);

% Live plot
figure()
tiledlayout(2,1);
nexttile;

% Accelerometer (x,y,z)
acc_x_line = animatedline;
axis([1 tlim -3 3]);

acc_x_line.LineStyle = '-';
acc_x_line.LineWidth = 2;
acc_x_line.Color = '[0.99 0.01 0.01]';
```

```
acc_x_line_title = title('Accelerometer', 'FontSize', 11);
acc_x_line_xlabel = xlabel('Time (s)');
acc_x_line_xlabel.FontSize = 9;
acc_x_line_ylabel = ylabel('Acceleration (g)');
acc_x_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

acc_y_line = animatedline;
axis([1 tlim -3 3]);

acc_y_line.LineStyle = '-';
acc_y_line.LineWidth = 2;
acc_y_line.Color = '[0.99 0.35 0.01]';
acc_y_line_title = title('Accelerometer', 'FontSize', 11);
acc_y_line_xlabel = xlabel('Time (s)');
acc_y_line_xlabel.FontSize = 9;
acc_y_line_ylabel = ylabel('Acceleration (g)');
acc_y_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

acc_z_line = animatedline;
axis([1 tlim -3 3]);

acc_z_line.LineStyle = '-';
acc_z_line.LineWidth = 2;
acc_z_line.Color = '[0.99 0.70 0.01]';
acc_z_line_title = title('Accelerometer', 'FontSize', 11);
acc_z_line_xlabel = xlabel('Time (s)');
acc_z_line_xlabel.FontSize = 9;
acc_z_line_ylabel = ylabel('Acceleration (g)');
acc_z_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

acc_legend = legend([acc_x_line acc_y_line acc_z_line],{'x axis', 'y axis'
    , 'z axis'});
acc_legend.FontSize = 12;
acc_legend.Location = 'northeastoutside';
acc_legend.NumColumns = 1;
acc_legend.TextColor = 'w';
acc_legend.Title.String = 'Accelerometer Data';
acc_legend.Title.Color = 'w';
acc_legend.Title.FontSize = 11;
hold off;

nexttile;

%Gyroscope (x,y,z)

gyro_x_line = animatedline;
axis([1 tlim -10 10 ]);

```

```
gyro_x_line.LineStyle = '-';
gyro_x_line.LineWidth = 2;
gyro_x_line.Color = '[0.01 0.01 0.99]';
gyro_x_line_title = title('Gyroscope', 'FontSize', 11);
gyro_x_line_xlabel = xlabel('Time (s)');
gyro_x_line_xlabel.FontSize = 9;
gyro_x_line_ylabel = ylabel('Angular Velocity Rad/s');
gyro_x_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

gyro_y_line = animatedline;
axis([1 tlim -10 10 ]);

gyro_y_line.LineStyle = '-';
gyro_y_line.LineWidth = 2;
gyro_y_line.Color = '[0.01 0.31 0.99]';
gyro_y_line_title = title('Gyroscope', 'FontSize', 11);
gyro_y_line_xlabel = xlabel('Time (s)');
gyro_y_line_xlabel.FontSize = 9;
gyro_y_line_ylabel = ylabel('Angular Velocity Rad/s');
gyro_y_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

gyro_z_line = animatedline;
axis([1 tlim -10 10 ]);

gyro_z_line.LineStyle = '-';
gyro_z_line.LineWidth = 2;
gyro_z_line.Color = '[0.01 0.70 0.99]';
gyro_z_line_title = title('Gyroscope', 'FontSize', 11);
gyro_z_line_xlabel = xlabel('Time (s)');
gyro_z_line_xlabel.FontSize = 9;
gyro_z_line_ylabel = ylabel('Angular Velocity Rad/s');
gyro_z_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.45 0.45 0.5]');
grid on; hold on;

gyro_legend = legend([gyro_x_line gyro_y_line gyro_z_line],{'x axis', 'y
    axis', 'z axis'});
gyro_legend.FontSize = 12;
gyro_legend.Location = 'northeastoutside';
gyro_legend.NumColumns = 1;
gyro_legend.TextColor = 'w';
gyro_legend.Title.String = 'Gyroscope Data';
gyro_legend.Title.Color = 'w';
gyro_legend.Title.FontSize = 11;
hold off;

% Roll and Pitch Plot %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Live plot
figure()
```

```
tiledlayout(2,1);
nexttile;

% Roll
roll_line = animatedline;
axis([1 tlim -180 180]);

roll_line.LineStyle = '-';
roll_line.LineWidth = 2;
roll_line.Color = '[0.1 0.1 0.80]';
roll_line_title = title('Roll', 'FontSize', 11);
roll_xlabel = xlabel('Time (s)');
roll_line_xlabel.FontSize = 9;
roll_line_ylabel = ylabel('\phi, Degrees','FontSize', 8);
roll_line_ylabel.FontSize = 9;
set(gca, 'Color', '[0.85 0.85 0.85]');
grid on; hold on;

nexttile;

% Pitch
pitch_line = animatedline;
axis([1 tlim -180 180]);

pitch_line.LineStyle = '-';
pitch_line.LineWidth = 2;
pitch_line.Color = '[0.80 0.1 0.1]';
pitch_line_title = title('Pitch', 'FontSize', 11);
pitch_xlabel = xlabel('Time (s)');
pitch_xlabel.FontSize = 9;
pitch_ylabel = ylabel('\theta, Degrees','FontSize', 8);
pitch_ylabel.FontSize = 9;
set(gca, 'Color', '[0.85 0.85 0.85]');
grid on; hold on;

%Live streaming data

imu_count = 0;
imu_data = [];

while imu_count < imu_lim
    imu_count = imu_count + 1;

    imu_read = read(imu);   % read imu data
    imu_matrix = imu_read{:,:};
    imu_mean=mean(imu_matrix);

    % accelerometer
    acc_x = imu_mean(:,1);
    acc_y = imu_mean(:,2);
    acc_z = imu_mean(:,3);

    acc_x_g = acc_x/9.81;
    acc_y_g = acc_y/9.81;
```

```
    acc_z_g = acc_z/9.81 - 1;

    % gyroscope
    gyro_x = imu_mean(:,4);
    gyro_y = imu_mean(:,5);
    gyro_z = imu_mean(:,6);

    gyro_x_deg = rad2deg(gyro_x);
    gyro_y_deg = rad2deg(gyro_y);
    gyro_z_deg = rad2deg(gyro_z);


    %%%%%%%%%%      Kalman Filter     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    qdot = getqdot(gyro_x,gyro_y,gyro_z,x);
    xkminus = statePredict(gyro_x,gyro_y,gyro_z,x,h);
    pminus = pPredict(P,Q,gyro_x,gyro_y,gyro_z);
    kgain = pminus*inv(pminus+R);
    z = measureAccel(acc_x,acc_y,acc_z);
    x = xkminus + kgain*(z-xkminus);
    [roll(imu_count),pitch(imu_count)] = getEuler(x,imu_count);
    P = (eye(4) - kgain)*pminus;
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % magnetometer
    mag_x = imu_mean(:,7);
    mag_y = imu_mean(:,8);
    mag_z = imu_mean(:,9);

    % orientation
    ort_x_rad = imu_mean(:,12);
    ort_y_rad = imu_mean(:,11);
    ort_z_rad = imu_mean(:,10);

    ort_x = rad2deg(ort_x_rad);
    ort_y = rad2deg(ort_y_rad);
    ort_z = rad2deg(ort_z_rad);

    imu_data = [imu_data; [imu_count,acc_x_g,acc_y_g,acc_z_g,...
        gyro_x, gyro_y,gyro_z,mag_x,mag_y,mag_z,ort_x,ort_y,ort_z]];

    show_imu = [acc_x_g,acc_y_g,acc_z_g,gyro_x, gyro_y,gyro_z,...
        mag_x,mag_y,mag_z,ort_x,ort_y,ort_z];
    %disp(show_imu);

    if imu_count == 500
        disp('END SESSION')
    end

    addpoints(acc_x_line,imu_count*tconversion,acc_x_g);
    addpoints(acc_y_line,imu_count*tconversion,acc_y_g);
    addpoints(acc_z_line,imu_count*tconversion,acc_z_g);

    addpoints(gyro_x_line,imu_count*tconversion,gyro_x);
```

```
    addpoints(gyro_y_line,imu_count*tconversion,gyro_y);
    addpoints(gyro_z_line,imu_count*tconversion,gyro_z);

    addpoints(roll_line,imu_count*tconversion,roll(end));
    addpoints(pitch_line,imu_count*tconversion,pitch(end));


    drawnow;

end

imu_table = array2table(imu_data,'VariableNames', {'Time',...
    'Acc x', 'Acc y', 'Acc z','Gyro X','Gyro Y','Gyro Z',...
    'Mag X','Mag Y','Mag Z','ORT X','ORT Y','ORT Z'});


%%%%%%%%%%%%%% Function Calls %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function qq = getquaternions(phi, theta)  % create quarternion vector
   given theta and phi
    qq(1) = sin(phi/2)*cos(theta/2);
    qq(2) = cos(phi/2)*sin(theta/2);
    qq(3) = -sin(phi/2)*sin(theta/2);
    qq(4) = cos(phi/2)*cos(theta/2);

    qq =qq';

end

function newQ = statePredict(p,q,r,x,h) % Implement state predictor
   equation of EKF
    A = (h/2)*[ 2/h,r,-q,p;
                -r,2/h,p,q;
                 q,-p,2/h,r;
                -p,-q,-r,h/2] ;
    newQ = A * x;

end

function z = measureAccel(x,y,z) % Determination of Euler angles theta and
    phi from measurment
    theta = asin(x/9.81);
    phi = atan2(y,z);
    z = getquaternions(phi,theta);
end

function p = pPredict(P,Q,p,q,r) % Predict covariance in EKF
    A = getJacobian(p,q,r);

    p = A*P*A' + Q;

end

function A = getJacobian(p,q,r) % Calculation of Jacobian Matrix for EKF
```

```
    A = 0.5*[ 0,r,-q,p;
              -r,0,p,q;
               q,-p,0,r;
              -p,-q,-r,0] ;
end

function [phi,theta] = getEuler(Q,imu_count)  % Calculation of Euler
    Angles from updated Quartenrion
    clc;

    q1 = Q(1);
    q2 = Q(2);
    q3 = Q(3);
    q4 = Q(4);

    a11 = q1^2 - q2^2 - q3^2 + q4^2;
    a12 = 2 * ( q1 * q2 + q3 * q4 );
    a13 = 2 * ( q1 * q3 - q2 * q4 );
    a23 = 2 * ( q2 * q3 + q1 * q4 );
    a33 = q3^2 - q1^2 - q2^2 + q4^2;

    phi = atan2( a23, a33 );
    phi = rad2deg(phi);
    theta = atan2( -a13, sqrt( a11 * a11 + a12 * a12 ) );
    theta = rad2deg(theta);

    tsec = imu_count *53.768392/500;  % time conversion

    fprintf('Pitch is %2.2f degrees \n\n',theta);
    fprintf('Roll is %2.2f degrees \n\n', phi);
    fprintf('Current Time Duration: %2.2f s\n\n', tsec);
end
```