IMU 6 DOF Attitude Estimation with Kalman Filtering Sensor Fusion

Michael Pittenger
Department of Electrical and Biomedical Engineering
University of Nevada, Reno
mpittenger@unr.edu

***Abstract*— This paper explores the implementation of Kalman filtering techniques for attitude estimation using data from a low-cost IMU. The focus is on fusing accelerometer and gyroscope measurements to achieve accurate and reliable 6-DOF orientation tracking. The project demonstrates the effectiveness of Kalman filtering in mitigating sensor noise and drift, ensuring robust performance under various conditions. Key results include enhanced stability in pitch and roll estimations, with the potential for future hardware-based optimizations.**

Keywords—Kalman Filter, IMU, Microcontroller, Arduino, Attitude Estimation, Gyroscope, Accelerometer, Euler Angles

I. INTRODUCTION

An Inertial Measurement Unit (IMU) is a device that measures the force, angular rate, and sometimes orientation of the device, using a combination of accelerometers, gyroscopes, and sometimes magnetometers [1]. An IMMU is an IMU with a magnetometer included. The device used in this report is a 6 DoF (Degree of Freedom) IMU that measures the forces acted on the body by a 3-axis accelerometer measuring meters per second (m/s) in the xyz directions, and also by a 3-axis gyroscope measuring angles radian per second (rad/s). IMMUs with an additional 3-axis magnetometer can track 9 DoFs.

IMUs are often incorporated into systems that utilize the raw IMU data to calculate state variables such as attitude/orientation, angular rates, linear velocity, and position [1]. It is crucial for any navigational vehicle or device to be able to calculate these variables in order for it to properly take measurements or maneuver in reference to its surroundings. This project examines the application of IMU data in orientation/attitude estimation, which uses the raw IMU data to calculate the pitch and roll of the body of the device.

The combination of such sensors is a widely used way for machines to detect their tilt and orientation compared to a reference point [2]. The data from an accelerometer can be used to detect tilt, but it is generally noise, susceptible to external forces, and is not stable enough to prevent measurement drift in the long term. The gyroscope is used to detect angular acceleration and is not susceptible to external forces in the same way that an accelerometer is. Combined, these sensors can be used to more accurately detect the proper tilt and orientation of an object. Despite this, cheaper IMU sensors are still considerably noisy devices, so data fusion and error reduction algorithms, such as the Kalman Filter, are often implemented to increase reliability and accuracy.

This project uses data from an Adafruit ISM330DHCX - 6 DoF IMU and the Extended Kalman Filter is implemented using an Arduino MEGA2560 Microcontroller. Both the IMU module and the microcontroller are low cost hardware. More expensive IMUs and microcontrollers would yield better results, but the less expensive devices are more accessible and the noise issues from the low cost IMU are alleviated with a properly implemented Kalman Filter. The Arduino microcontroller processes the IMU data at a baud rate of 115200, which is sufficiently fast for this application.

The paper is structured as follows: Section II covers device specifications. Section III explains the circuit setup and testing. Section IV discusses basic Kalman filter results, without sensor fusion or bias correction. Section V adds sensor fusion and bias calculation to the Kalman filter and compares the output. Section VI attempt to implement the live readings of the Kalman filter through the Arduino IDE. Section VII concludes the paper and compares the efficacy of the experiment.

II. DEVICES

The IMU is an Adafruit ISM330DHCX [3], which supports 6 degrees of freedom (3 accelerometer xyz, 3 gyroscope xyz). The accelerometer capabilities are: ±2/±4/±8/±16 g at 1.6 Hz to 6.7KHz update rate, and the gyroscope capabilities are: ±125/±250/±500/±1000/±2000/±4000 dps at 12.5 Hz to 6.7 KHz. The IMU communicates through SPI or I2C interfacing and operates by a 3V or 5V power input [4], which allows for communication with the Arduino Mega 2560 microcontroller.

The Arduino Mega 2560 [5] is a microcontroller board based on the ATmega2560. It features 256KB of Flash memory, 8KB of SRAM, and 4KB of EEPROM. The microcontroller operates at a clock speed of 16 MHz and is powered by a 5V input, with a voltage range of 7V to 12V for optimal performance. The board includes 54 digital input/output pins, 15 of which can be used as PWM outputs, and 16 analog input pins. It also has 4 UARTs (hardware serial ports), which support communication via UART, SPI, and I2C protocols. The Mega 2560 supports both USB and external power for communication and power supply, making it compatible with various sensors and peripherals. The Arduino was chosen due to the accessibility of the microcontroller and its numerous libraries and IDE that help with prototyping.

The Adafruit IMU module comes with free to use Arduino and python libraries that extracts the measurements of the IMU in m/s and rad/s. Adafruit also provides circuit diagrams to aid in the connections between the modules, shown in **Figure 1**. The example code provided in the Adafruit LSM6DS under the ISM330DHCX module is shown in **Figure 2**.

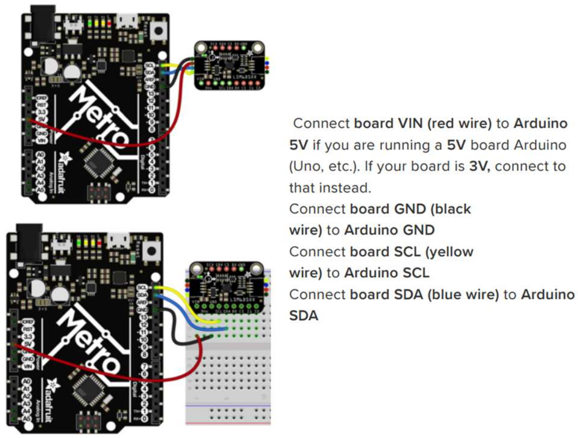

Figure 1: IMU I2C to Arduino Connection

```
# SPDX-FileCopyrightText: Copyright (c) 2020 Bryan Siepert for Adafruit Industries
#
# SPDX-License-Identifier: MIT
import time
import board
from adafruit_lsm6ds.ism330dhcx import ISM330DHCX

i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a microcontroller
sensor = ISM330DHCX(i2c)

while True:
    print("Acceleration: X:%.2f, Y: %.2f, Z: %.2f m/s^2" % (sensor.acceleration))
    print("Gyro X:%.2f, Y: %.2f, Z: %.2f radians/s" % (sensor.gyro))
    print("")
    time.sleep(0.5)
```

Figure 2: Adafruit Sample Code

For experimental purposes, the data monitoring software is run in the Arduino IDE along with the provided LSM6DF library, and the values are exported in a CSV data format. Once the data is in a CSV file, it can be loaded into MATLAB where the data analysis and Kalman filtering is processed. After model verification, the logic can be implemented on the Arduino IDE for on-line Kalman filtering and attitude estimation. The Adafruit 3D Model Viewer [6] can be used to test if the calculated angles are plausible.

III. Arduino Sensor Testing

In this section we discuss the reading of the sensor (acc xyz, gyro xyz), and the equations used for pitch and roll (not yaw). The measurements given by the sensor and calculated by the provided libraries are temperature, acceleration xyz (m/s), and gyroscope xyz (m/s). The temperature is not needed and is ignored. The sensor setup is shown in **Figure 1** and is shown on the breadboard in **Figure 3**.

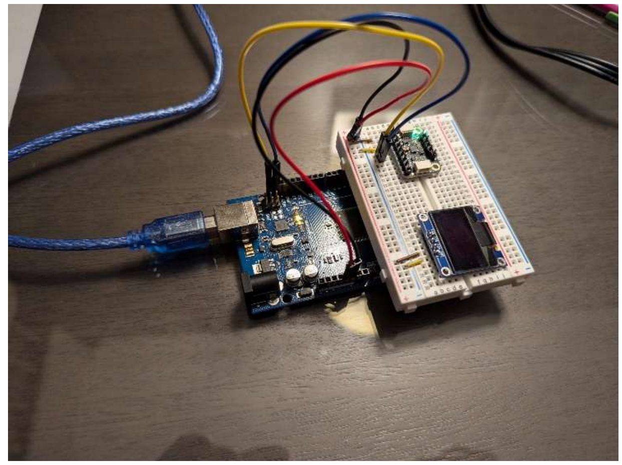

Figure 3: Arduino IMU Demo Circuit

The example code was loaded to test the functionality of the IMU. All of the readings are as in the expected range, with a z-direction acceleration of 10 m/s because of the acceleration due to gravity, and the rest of the values being close to zero. The average values of these sensor readings while at rest are the biases, and must be accounted for in our data processing phase. Noise can also be seen, which is also measured and accounted for in our covariance analysis in Section IV.

To calculate the orientation of the IMU, we use the known equations of the Euler angles to convert the raw sensor readings into the pitch and roll of the device. Euler angles are calculated using quaternions, but an approximation can be made as follows [7]

$$\boldsymbol{\theta_{AccPitch}} = \boldsymbol{atan2}(\boldsymbol{AccData_z},\ \boldsymbol{AccData_x}) \qquad (\mathbf{1})$$

$$\boldsymbol{\theta_{AccRoll}} = \boldsymbol{atan2}(\boldsymbol{AccData_z},\ \boldsymbol{AccData_y}) \qquad (\mathbf{2})$$

Where $\boldsymbol{\theta_{AccPitch}}$ and $\boldsymbol{\theta_{AccRoll}}$ are the Euler angles across the x-axis and y-axis respectively, and $\boldsymbol{AccData_x}$, $\boldsymbol{AccData_y}$, and $\boldsymbol{AccData_z}$ are the accelerometer readings in m/s in the x-axis, y-axis, and z-axis respectively.

The angle for yaw cannot be calculated from a 6 DoF IMU, as a magnetometer is required for accurate yaw calculation.

Converting the live signals to angles gives the live attitude estimation shown in **Figure 4**. Without proper liftering techniques, the system is subject to significant noise and drift over time. The accelerometer in particular is especially susceptible to noise through measurement and vibrations, with the gyroscope being susceptible to drift over time.

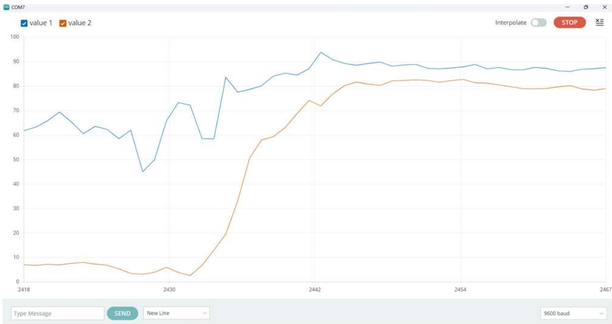

Figure 4: Live Arduino Roll from 0 to 90 Degrees

Adafruit provides a 3D model viewer that takes the Euler angles from a microcontroller serial output and displays them using the 3D graphic of a rabbit. The rabbit being moved by the angles calculated from the IMU data is shown in **Figure 5**. The rabbit model moved as the IMU moved in real space, showing that the IMU readings and Euler angle calculations were accurate. The noise in the readings were very visible on the 3D model, which is what we hope to minimize through the sensor fusion and bias correction.

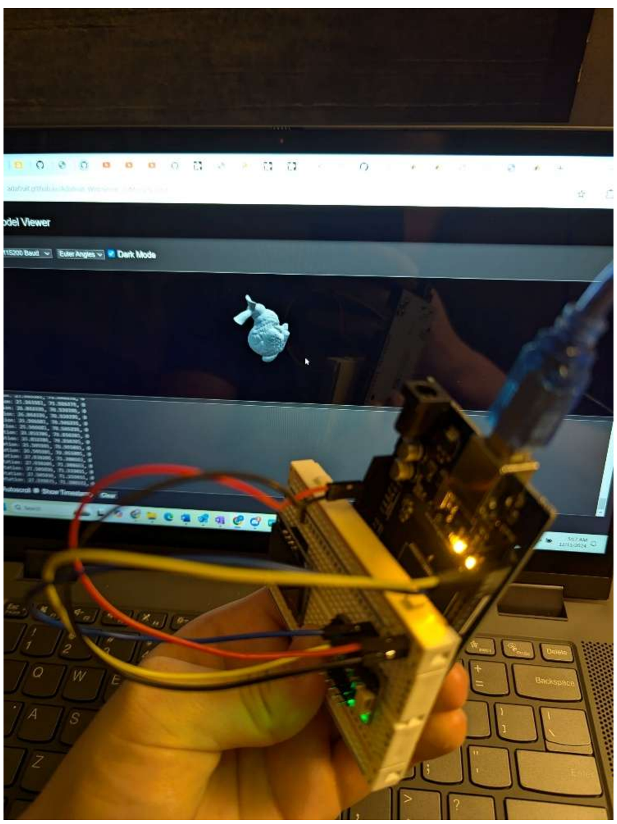

Figure 5: Adafruit 3D Model Viewer

IV. Kalman Filter Without Bias

One method for filtering the noise out of the attitude estimation calculated from the IMU would be to run the individual Euler angles through a generic linear Kalman filter. This method doesl not consider the information from the gyroscope sensor and will only rely on the readings provided by the accelerometer. The Kalman filter process flow being followed comes from [8] and is shown in **Figure 6**.

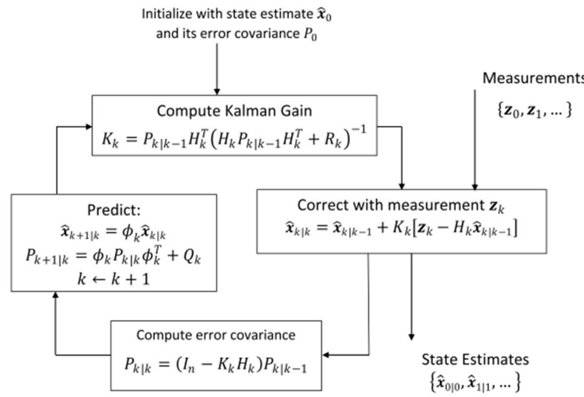

Figure 6: Block Diagram for Discrete Time Kalman Filter

For this the following calculations were taken from [7]. The models of the acceleration and gyroscope sensor readings can be implemented as followed:

$$\boldsymbol{\alpha}_{Acc} = \boldsymbol{\alpha}_{extra} - \boldsymbol{g} + \boldsymbol{b}_a + \boldsymbol{n}_a \tag{3}$$

Where, $\boldsymbol{\alpha}_{extra}$, $\boldsymbol{g}$, $\boldsymbol{b}_a$, and $\boldsymbol{n}_a$ are the external acceleration, gravitational acceleration, accelerometer bias and noise respectively and:

$$\boldsymbol{\omega}_{gyro} = \boldsymbol{\omega} + \boldsymbol{b}_g + \boldsymbol{n}_g \tag{4}$$

Where $\boldsymbol{\omega}$, $\boldsymbol{b}_g$, and $\boldsymbol{n}_g$ are the gyroscope measurement, bias, and noise.

The state space model is as follows:

$$\boldsymbol{x}_k = \boldsymbol{F}\boldsymbol{x}_{k-1} + \boldsymbol{B}\boldsymbol{u}_k + \boldsymbol{w}_k \tag{5}$$

$$\boldsymbol{z}_k = \boldsymbol{H}\boldsymbol{x}_k + \boldsymbol{v}_k \tag{6}$$

$$\boldsymbol{x}_k = [\boldsymbol{\theta}]_k \tag{7}$$

$$\boldsymbol{F} = [\mathbf{1}] \tag{8}$$

$$\boldsymbol{B} = [\mathbf{0}] \tag{9}$$

$$\boldsymbol{H} = [\mathbf{1}] \tag{10}$$

Where $\boldsymbol{x}_k$ is the angle at time $\boldsymbol{k}$ in degrees, $\boldsymbol{F}$ is the state transition model, $\boldsymbol{B}$ is the control-input model, $\boldsymbol{u}_k$ is the gyroscope measurement in degrees/second, $\boldsymbol{w_k}$ is the process noise, $\boldsymbol{z_k}$ is the measurement, $\boldsymbol{H}$ is the observation model, and $\boldsymbol{v_k}$ is the measurement noise.

The angular data recorded from the IMU was processed by the given Kalman filter model with the results shown in **Figure 7**. An R value must be chosen and influences how much the estimated values are susceptible to noise and therefore also influence how much the estimated values lag behind the true values. Through experimentation, an R value of 0.3 was chose. This R value filters out a considerable amount of noise, without having the estimated values lag too much to be effective.

The IMU was smoothly rotated along its pitch-axis and roll-axis, and then it was shook along its x-axis, y-axis, and z-axis to simulate noise or vibrations. The results of the pitch angle through the Kalman filter implemented on recorded data in MATLAB are shown in **Figures 7-9**.

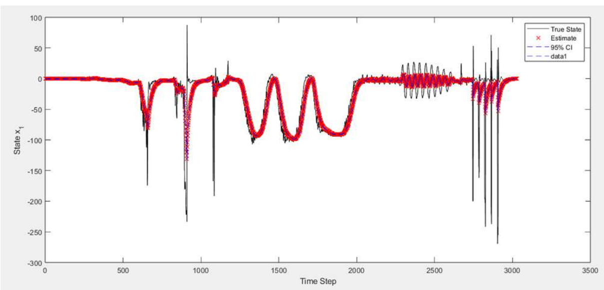

Figure 7: Pitch Angle Kalman Filter

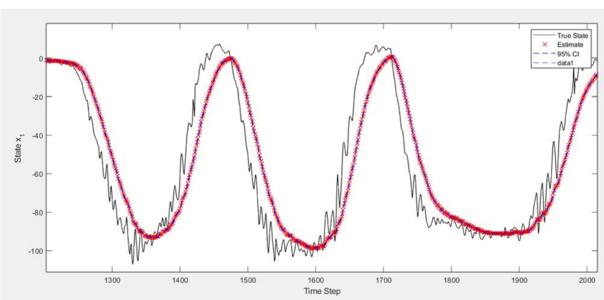

Figure 8: Pitch Angle Kalman Filter Rotation

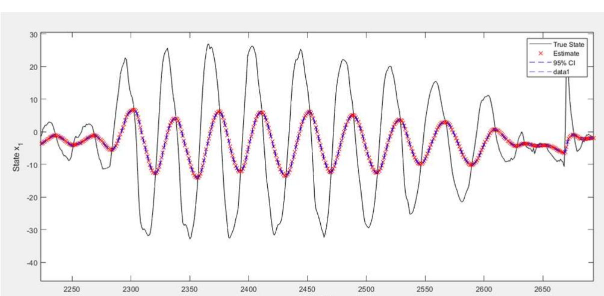

Figure 9: Pitch Angle Kalman Filter Shake in X-Direction

As can be seen, the Kalman filter is effective in reducing a significant amount of the noise seen by the sensor, without introducing too much lag into the output. The results are considerably more reliable than the unfiltered measurements, but without a proper bias implemented corrector, the device will drift over time and become unreliable.

V. Kalman Filter With Bias

In order to minimize the impact of drift over time, measurements from the gyroscope can be used as a bias for the accelerometer readings. The additional sensor readings implemented through said sensor fusion greatly enhances the reliability of the systems attitude estimation. The Kalman filter is updated to include the measurements from both sensors and their relation to one another in the sensor fusion algorithm [7].

The state space model is as follows:

$$\boldsymbol{x_k} = \boldsymbol{F}\boldsymbol{x_{k-1}} + \boldsymbol{B}\boldsymbol{u_k} + \boldsymbol{w_k} \tag{11}$$

$$\boldsymbol{z_k} = \boldsymbol{H}\boldsymbol{x_k} + \boldsymbol{v_k} \tag{12}$$

$$\boldsymbol{x_k} = \begin{bmatrix} \boldsymbol{\theta} \\ \dot{\boldsymbol{\theta}}_{\boldsymbol{b}} \end{bmatrix}_{\boldsymbol{k}} \tag{13}$$

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{1} & -\boldsymbol{\Delta t} \\ \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \tag{14}$$

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{\Delta t} \\ \boldsymbol{0} \end{bmatrix} \tag{15}$$

$$\boldsymbol{H} = [\boldsymbol{1} \quad \boldsymbol{0}] \tag{16}$$

Where $\boldsymbol{x_k}$ is the angle at time $\boldsymbol{k}$ in degrees, $\boldsymbol{F}$ is the state transition model, $\boldsymbol{B}$ is the control-input model, $\boldsymbol{u_k}$ is the gyroscope measurement in degrees/second, $\boldsymbol{w_k}$ is the process noise, $\boldsymbol{z_k}$ is the measurement, $\boldsymbol{H}$ is the observation model, and $\boldsymbol{v_k}$ is the measurement noise.

$\boldsymbol{x_k}$ is the system state vector at time k, with the outputs of the filter being the angle and the bias based on the accelerometer and gyroscope measurements. The bias is the amount that the gyroscope has drifted. F is the state transition matrix which is applied to the $\boldsymbol{x_{k-1}}$ state. $\boldsymbol{u_k}$ is the control input of the filter, which is the gyroscope measurement in degrees per second at time k (angular rate theta dot) and the bias. The B matrix is created by multiplying the angular rate by the change in time and the bias by 0 because the bias cannot be calculated directly based on angular velocity.

This allows us to rewrite the state equation as:

$$\boldsymbol{x_k} = \boldsymbol{F}\boldsymbol{x_{k-1}} + \boldsymbol{B}\dot{\boldsymbol{\theta}}_{\boldsymbol{k}} + \boldsymbol{w_k} \tag{17}$$

The noise variables $\boldsymbol{w_k}$ and $\boldsymbol{v_k}$ are assumed to be Gaussian white noise. The covariance matrix Q_k represents the variance of the accelerometer and the bias, which depends on the current time and thus is multiplied by delta t. The variance of the sensors can also be measured beforehand by letting the sensors collect data while stationary. The variance Q was measured by recording the data collected by the IMU while stationary for a long period of time and taking the variance of the data collected. The noise is as follows:

$$w_k \sim N(0, Q_k) \quad (18)$$

$$v_k \sim N(0, R) \quad (19)$$

$$R = E[v_k \quad v_k^T] = var(v_k) \quad (20)$$

Where Q_k is the variance of the measured noise while the device at a steady state, and R is the tunable variable that will change how the sensitive the system is to noise.

The implementation of the Kalman filter is through the following steps:

Predict:

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + B\dot{\theta}_k \quad (21)$$

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q_k \quad (22)$$

Update:

$$y_k = z_k - H\hat{x}_{k|k-1} \quad (23)$$

$$S_k = HP_{k|k-1}H^T + R \quad (24)$$

$$K_k = P_{k|k-1}H^T S_k^{-1} \quad (25)$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k y_k \quad (26)$$

$$P_{k|k} = (I - K_k H)P_{k|k-1} \quad (27)$$

Where $\hat{x}_{k|k-1}$ is priori state estimate of the current state at time k based on the estimates of the states before it. $\hat{x}_{k-1|k-1}$ is the previous state which is the previous estimate based on the previous states and estimates. $P_{k|k-1}$ is the priori error covariance matrix at time k previous error covariance matrix $P_{k-1|k-1}$. y_k is the innovation which is the difference between the measurement and the priori state. S_k is the innovation covariance. K_k is the Kalman gain. $P_{k|k}$ is the updated covariance matrix.

The Kalman filter algorithm used in the previous section was updated to implement the new state equations and matrices. This new method takes into account the accelerometer as well as the gyroscope, and a bias value is created through the Kalman gain, K, and its estimation of the unreliability of the angle readings over time. The algorithm will be applied on both the pitch and roll angles. The results of the algorithm on the pitch angle data collected previous is shown in **Figure 10-13**.

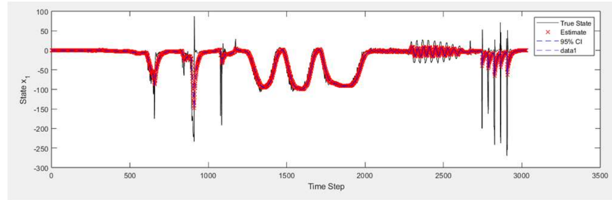

Figure 10: Pitch Angle Kalman Filtering with Bias

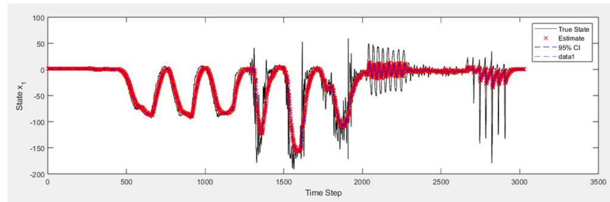

Figure 11: Roll Angle Kalman Filtering with Bias

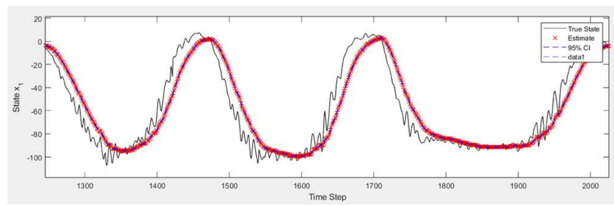

Figure 12: Pitch Angle Kalman Filter Rotation with Bias

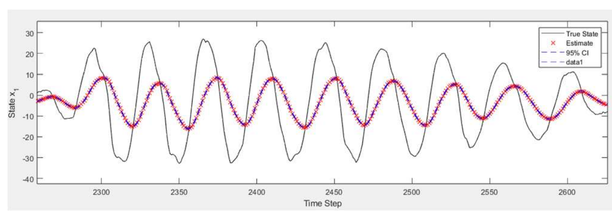

Figure 13: Pitch Angle Kalman Filter Shake in X-Direction with Bias

The results are marginally better, but very similar due to the data only being recorded over a short period of time. As time goes on, the bias and gyroscope measurements will have more of an impact on the noise and drift of the measurements and predicted values as opposed to the filter only using accelerometer measurements, which is expected to drift and become unreliable as time passes.

VI. Live Arduino Kalman Filter Results

The algorithm written and analyzed on MATLAB was ported over to C++ to be ran in the Arduino IDE in real time. The pitch angle and its estimated value via the Kalman filter was output to the serial monitor, shown in **Figure 14**. Both angles and their estimated Kalman filter values are displayed in **Figure 15**.

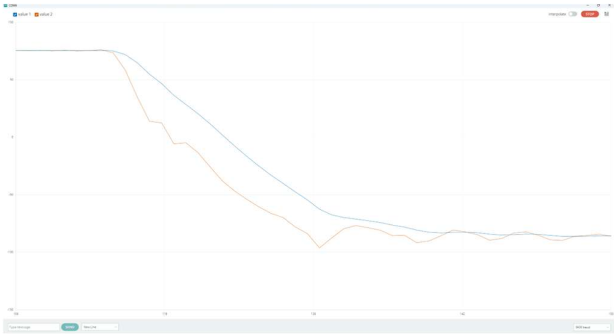

Figure 14: Live Pitch Angle Kalman Filtering with Bias

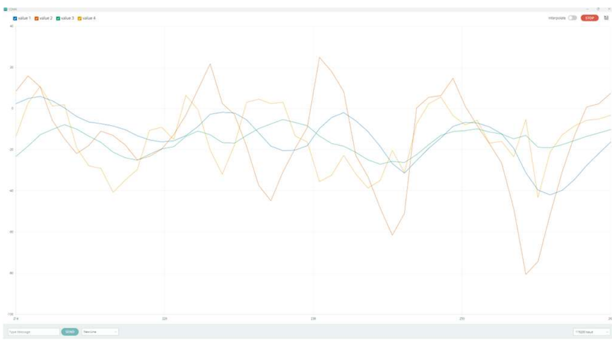

Figure 15: Live Pitch and Roll Angle Kalman Filtering with Bias

The serial output was also used on the previously shown Adafruit 3D model viewer, and the rabbit model is moving in space visibly smoother than was testing without the filters.

VII. Conclusion

In conclusion, the implementation of the Kalman filter for attitude estimation using a 6-DoF IMU demonstrated promising results in reducing noise and drift, essential for reliable orientation tracking. By fusing accelerometer and gyroscope measurements, the system provided more accurate pitch and roll estimations, despite the inherent noise of low-cost sensors. The performance was further enhanced by incorporating bias correction through sensor fusion, which mitigated long-term drift, a common challenge in IMU-based systems. This project successfully showcased the potential of Kalman filtering for improving sensor data accuracy, offering a foundation for future advancements in IMU-based attitude estimation systems. Further optimization, particularly through hardware enhancements and fine-tuning the filter parameters, could yield even more precise results in real-time applications.

References

[1] "Inertial Measurement Unit," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Inertial_measurement_unit. [Accessed: 12-Dec-2024].

[2] M. Karooza, "Attitude Estimation Using IMU Sensors," Karooza.net. [Online]. Available: https://karooza.net/attitude-estimation-using-imu-sensors. [Accessed: 13-Dec-2024].

[3] "LSM6DSOX and ISM330DHC - 6 DoF IMU," Adafruit Learning System. [Online]. Available: https://learn.adafruit.com/lsm6dsox-and-ism330dhc-6-dof-imu/arduino. [Accessed: 12-Dec-2024].

[4] STMicroelectronics, "ISM330DHC: iNEMO inertial module - 3D accelerometer and 3D gyroscope," Datasheet, Rev. 3, Nov. 2020. [Online]. Available: https://www.st.com/resource/en/datasheet/ism330dhc.pdf. [Accessed: 12-Dec-2024].

[5] Microchip Technology Inc., "ATmega2560: 8-bit AVR microcontroller with 256KB self-programming Flash program memory," Datasheet, 2023. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega2560-2561-Complete.pdf. [Accessed: 12-Dec-2024].

[6] "Adafruit WebSerial 3D Model Viewer," Adafruit Documentation. [Online]. Available: https://adafruit.github.io/Adafruit_WebSerial_3DModelViewer/. [Accessed: 12-Dec-2024].

[7] L. Lauszus, "A practical approach to Kalman filter and how to implement it," TKJ Electronics Blog, Sep. 10, 2012. [Online]. Available: https://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/#comment-57783. [Accessed: 12-Dec-2024].

[8] M. S. Fadali, Introduction to Random Signals, Estimation Theory, and Kalman Filtering. Springer, 2024.

FinalProjectTest.ino

```
// Michael Pittenger
// EE 782 Project Test Code

#include <Adafruit_ISM330DHCX.h>

Adafruit_ISM330DHCX ism330dhcx;

// IMU variables
float accx;
float accy;
float accz;
float gyrx;
float gyry;
float gyrz;

unsigned long startTime;

void setup(void) {
  Serial.begin(115200);
  while (!Serial)
    delay(10); // will pause Zero, Leonardo, etc until serial console opens

  Serial.println("Adafruit ISM330DHCX test!");

  if (!ism330dhcx.begin_I2C()) {
    Serial.println("Failed to find ISM330DHCX chip");
    while (1) {
      delay(10);
    }
  }

  Serial.println("ISM330DHCX Found!");

  // Set accelerometer and gyro ranges and data rates
  Serial.print("Accelerometer range set to: ");
  switch (ism330dhcx.getAccelRange()) {
    case LSM6DS_ACCEL_RANGE_2_G:
      Serial.println("+2G");
      break;
    case LSM6DS_ACCEL_RANGE_4_G:
      Serial.println("+4G");
      break;
    case LSM6DS_ACCEL_RANGE_8_G:
      Serial.println("+8G");
      break;
    case LSM6DS_ACCEL_RANGE_16_G:
      Serial.println("+16G");
      break;
  }

  Serial.print("Gyro range set to: ");
  switch (ism330dhcx.getGyroRange()) {
    case LSM6DS_GYRO_RANGE_125_DPS:
      Serial.println("125 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_250_DPS:
      Serial.println("250 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_500_DPS:
      Serial.println("500 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_1000_DPS:
      Serial.println("1000 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_2000_DPS:
      Serial.println("2000 degrees/s");
      break;
    case ISM330DHCX_GYRO_RANGE_4000_DPS:
      Serial.println("4000 degrees/s");
      break;
  }

  // Initialize accelerometer and gyro data rate
  Serial.print("Accelerometer data rate set to: ");
  switch (ism330dhcx.getAccelDataRate()) {
    case LSM6DS_RATE_SHUTDOWN:
      Serial.println("0 Hz");
      break;
    case LSM6DS_RATE_12_5_HZ:
      Serial.println("12.5 Hz");
      break;
    case LSM6DS_RATE_26_HZ:
      Serial.println("26 Hz");
      break;
    case LSM6DS_RATE_52_HZ:
      Serial.println("52 Hz");
      break;
    case LSM6DS_RATE_104_HZ:
      Serial.println("104 Hz");
      break;
    case LSM6DS_RATE_208_HZ:
      Serial.println("208 Hz");
      break;
    case LSM6DS_RATE_416_HZ:
      Serial.println("416 Hz");
      break;
    case LSM6DS_RATE_833_HZ:
      Serial.println("833 Hz");
      break;
    case LSM6DS_RATE_1_66K_HZ:
      Serial.println("1.66 KHz");
      break;
    case LSM6DS_RATE_3_33K_HZ:
      Serial.println("3.33 KHz");
      break;
    case LSM6DS_RATE_6_66K_HZ:
      Serial.println("6.66 KHz");
      break;
  }

  Serial.print("Gyro data rate set to: ");
  switch (ism330dhcx.getGyroDataRate()) {
    case LSM6DS_RATE_SHUTDOWN:
      Serial.println("0 Hz");
      break;
    case LSM6DS_RATE_12_5_HZ:
      Serial.println("12.5 Hz");
      break;
    case LSM6DS_RATE_26_HZ:
      Serial.println("26 Hz");
      break;
    case LSM6DS_RATE_52_HZ:
      Serial.println("52 Hz");
      break;
    case LSM6DS_RATE_104_HZ:
      Serial.println("104 Hz");
      break;
    case LSM6DS_RATE_208_HZ:
      Serial.println("208 Hz");
      break;
    case LSM6DS_RATE_416_HZ:
      Serial.println("416 Hz");
      break;
    case LSM6DS_RATE_833_HZ:
      Serial.println("833 Hz");
      break;
    case LSM6DS_RATE_1_66K_HZ:
      Serial.println("1.66 KHz");
      break;
    case LSM6DS_RATE_3_33K_HZ:
      Serial.println("3.33 KHz");
      break;
    case LSM6DS_RATE_6_66K_HZ:
      Serial.println("6.66 KHz");
      break;
  }

  ism330dhcx.configInt1(false, false, true); // accelerometer DRDY on INT1
  ism330dhcx.configInt2(false, true, false); // gyro DRDY on INT2

  // CSV header for data logging
  Serial.println("Time (ms), AccX, AccY, AccZ, GyroX, GyroY, GyroZ");

  // Start time for 20 seconds
  startTime = millis();
}

void loop() {
  // Check if 20 seconds have passed
  // if (millis() - startTime >= 20000) {
  //   Serial.println("20 seconds have passed. Stopping.");
  //   while(1);  // Stop the program by entering an infinite loop
  // }

  // Get a new normalized sensor event
  sensors_event_t accel;
  sensors_event_t gyro;
```

```
  sensors_event_t temp;
  ism330dhcx.getEvent(&accel, &gyro, &temp);

  // Update sensor values
  accx = accel.acceleration.x;
  accy = accel.acceleration.y;
  accz = accel.acceleration.z;
  gyrx = gyro.gyro.x;
  gyry = gyro.gyro.y;
  gyrz = gyro.gyro.z;

  float theta_pitch = atan2(accel.acceleration.z, accel.acceleration.x);
  float theta_roll = atan2(accel.acceleration.z, accel.acceleration.y);

  // Log the data in CSV format with increased precision (6 decimal places)
  unsigned long currentTime = millis();  // Get time in milliseconds
  // Serial.print(currentTime);              // Print time (in ms)
  // Serial.print(",");                      // Separator
  Serial.print("Orientation: ");
  Serial.print(theta_pitch*57.2957795-90, 6); // multply by
57.29577951308232087679815481410 5 for degrees
  Serial.print(", ");
  Serial.print(theta_roll*57.2957795-90, 6);
  Serial.print(", ");
  Serial.print(0);
  // Serial.print(accx, 6);                  // Accelerometer X with 6 decimal places
  // Serial.print(",");                      // Separator
  // Serial.print(accy, 6);                  // Accelerometer Y with 6 decimal places
  // Serial.print(",");                      // Separator
  // Serial.print(accz, 6);                  // Accelerometer Z with 6 decimal places
  // Serial.print(",");                      // Separator
  // Serial.print(gyrx, 6);                  // Gyroscope X with 6 decimal places
  // Serial.print(",");                      // Separator
  // Serial.print(gyry, 6);                  // Gyroscope Y with 6 decimal places
  // Serial.print(",");                      // Separator
  // Serial.println(gyrz, 6);                // Gyroscope Z with 6 decimal places
  Serial.println();
  // delay(100);  // Adjust delay as needed
}
```

FinalProjectFinal.ino

```
// Michael Pittenger
// EE 782 Project Final Code

#include <Adafruit_ISM330DHCX.h>
#include <Arduino.h>
Adafruit_ISM330DHCX ism330dhcx;

// IMU variables
float accx;
float accy;
float accz;
float gyrx;
float gyry;
float gyrz;

unsigned long startTime;

// Kalman Filter variables
float F[2][2] = {{1, 0}, {0, 1}}; // State transition matrix
float B[2] = {0, 0};             // Input matrix
float H[2] = {1, 0};             // Measurement matrix
float Q[2][2] = {{0.0014, 0}, {0, 0.03}}; // Process noise covariance
float R = 0.03;                  // Measurement noise covariance
float xhatpx[2] = {0, 0};          // Predicted state
float xhatpy[2] = {0, 0};          // Predicted state
float P_est[2][2] = {{0, 0}, {0, 0}}; // Error covariance
float P_esty[2][2] = {{0, 0}, {0, 0}}; // Error covariance
float gyro = 0;                  // Gyro input
float measurementx = 0;           // Measurement input
float measurementy = 0;           // Measurement input
float deltat = 0.1;              // Time step

void setup(void) {
  Serial.begin(115200);
  while (!Serial)
    delay(10); // will pause Zero, Leonardo, etc until serial console opens

  Serial.println("Adafruit ISM330DHCX test!");

  if (!ism330dhcx.begin_I2C()) {
    Serial.println("Failed to find ISM330DHCX chip");
    while (1) {
      delay(10);
    }
  }

  Serial.println("ISM330DHCX Found!");

  // Set accelerometer and gyro ranges and data rates
  Serial.print("Accelerometer range set to: ");
  switch (ism330dhcx.getAccelRange()) {
    case LSM6DS_ACCEL_RANGE_2_G:
      Serial.println("+-2G");
      break;
    case LSM6DS_ACCEL_RANGE_4_G:
      Serial.println("+-4G");
      break;
    case LSM6DS_ACCEL_RANGE_8_G:
      Serial.println("+-8G");
      break;
    case LSM6DS_ACCEL_RANGE_16_G:
      Serial.println("+-16G");
      break;
  }

  Serial.print("Gyro range set to: ");
  switch (ism330dhcx.getGyroRange()) {
    case LSM6DS_GYRO_RANGE_125_DPS:
      Serial.println("125 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_250_DPS:
      Serial.println("250 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_500_DPS:
      Serial.println("500 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_1000_DPS:
      Serial.println("1000 degrees/s");
      break;
    case LSM6DS_GYRO_RANGE_2000_DPS:
      Serial.println("2000 degrees/s");
      break;
    case ISM330DHCX_GYRO_RANGE_4000_DPS:
      Serial.println("4000 degrees/s");
      break;
  }

  // Initialize accelerometer and gyro data rate
  Serial.print("Accelerometer data rate set to: ");
  switch (ism330dhcx.getAccelDataRate()) {
    case LSM6DS_RATE_SHUTDOWN:
      Serial.println("0 Hz");
      break;
    case LSM6DS_RATE_12_5_HZ:
      Serial.println("12.5 Hz");
      break;
    case LSM6DS_RATE_26_HZ:
      Serial.println("26 Hz");
      break;
    case LSM6DS_RATE_52_HZ:
      Serial.println("52 Hz");
      break;
    case LSM6DS_RATE_104_HZ:
      Serial.println("104 Hz");
      break;
    case LSM6DS_RATE_208_HZ:
      Serial.println("208 Hz");
      break;
    case LSM6DS_RATE_416_HZ:
      Serial.println("416 Hz");
      break;
    case LSM6DS_RATE_833_HZ:
      Serial.println("833 Hz");
      break;
    case LSM6DS_RATE_1_66K_HZ:
      Serial.println("1.66 KHz");
      break;
    case LSM6DS_RATE_3_33K_HZ:
      Serial.println("3.33 KHz");
      break;
    case LSM6DS_RATE_6_66K_HZ:
      Serial.println("6.66 KHz");
      break;
  }

  Serial.print("Gyro data rate set to: ");
  switch (ism330dhcx.getGyroDataRate()) {
    case LSM6DS_RATE_SHUTDOWN:
      Serial.println("0 Hz");
      break;
    case LSM6DS_RATE_12_5_HZ:
      Serial.println("12.5 Hz");
      break;
    case LSM6DS_RATE_26_HZ:
      Serial.println("26 Hz");
      break;
    case LSM6DS_RATE_52_HZ:
      Serial.println("52 Hz");
      break;
    case LSM6DS_RATE_104_HZ:
      Serial.println("104 Hz");
      break;
    case LSM6DS_RATE_208_HZ:
      Serial.println("208 Hz");
      break;
    case LSM6DS_RATE_416_HZ:
      Serial.println("416 Hz");
      break;
    case LSM6DS_RATE_833_HZ:
      Serial.println("833 Hz");
      break;
    case LSM6DS_RATE_1_66K_HZ:
      Serial.println("1.66 KHz");
      break;
    case LSM6DS_RATE_3_33K_HZ:
      Serial.println("3.33 KHz");
      break;
    case LSM6DS_RATE_6_66K_HZ:
      Serial.println("6.66 KHz");
      break;
  }

  ism330dhcx.configInt1(false, false, true); // accelerometer DRDY on INT1
  ism330dhcx.configInt2(false, true, false); // gyro DRDY on INT2

  // CSV header for data logging
  Serial.println("Time (ms), AccX, AccY, AccZ, GyroX, GyroY, GyroZ");
```

```
  // Start time for 20 seconds
  startTime = millis();

  xhatpx[0] = 0;
  xhatpy[0] = 0;
}

void loop() {

  // Get a new normalized sensor event
  sensors_event_t accel;
  sensors_event_t gyro;
  sensors_event_t temp;
  ism330dhcx.getEvent(&accel, &gyro, &temp);

  // Update sensor values
  accx = accel.acceleration.x;
  accy = accel.acceleration.y;
  accz = accel.acceleration.z;
  gyrx = gyro.gyro.x;
  gyry = gyro.gyro.y;
  gyrz = gyro.gyro.z;

  float theta_pitch = atan2(accel.acceleration.z, accel.acceleration.x);
  float theta_roll = atan2(accel.acceleration.z, accel.acceleration.y);

  static unsigned long lastTime = 0;
  unsigned long currentTime = millis();

  // Calculate delta time
  deltat = (currentTime - lastTime) / 1000.0; // Time in seconds
  lastTime = currentTime;

  // Update matrices based on deltat
  F[0][1] = -deltat;
  B[0] = deltat;

  // Mock data for gyro and measurement
  measurementx = theta_pitch * 57.2957795 - 90; // Sensor reading
  measurementy = theta_roll * 57.2957795 - 90; // Sensor reading

  // Pitch angle
  // Predictor step
  float xhatx[2];
  xhatx[0] = F[0][0] * xhatpx[0] + F[0][1] * xhatpx[1] + B[0] * gyrx;
  xhatx[1] = F[1][0] * xhatpx[0] + F[1][1] * xhatpx[1] + B[1] * gyrx;

  float P_pred[2][2];
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      P_pred[i][j] = F[i][0] * P_est[0][j] + F[i][1] * P_est[1][j] + Q[i][j];
    }
  }

  // Measurement update (corrector)
  float K[2]; // Kalman gain
  float S = H[0] * (P_pred[0][0] * H[0] + P_pred[0][1] * H[1]) + R;
  K[0] = (P_pred[0][0] * H[0] + P_pred[0][1] * H[1]) / S;
  K[1] = (P_pred[1][0] * H[0] + P_pred[1][1] * H[1]) / S;

  // Update state estimate
  xhatpx[0] = xhatx[0] + K[0] * (measurementx - H[0] * xhatx[0]);
  xhatpx[1] = xhatx[1] + K[1] * (measurementx - H[0] * xhatx[0]);

  // Update covariance
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      P_est[i][j] = (1 - K[i] * H[j]) * P_pred[i][j];
    }
  }

  // Print results for debugging
  // Serial.print("Estimated State: ");
  // Serial.print(xhatpx[0]);
  // Serial.print(", ");
  // // Serial.println(xhatpx[1]);
  // // Serial.print(", ");
  // Serial.print(theta_pitch * 57.2957795 - 90);
  // Serial.print(", ");

  // Roll angle
  // Predictor step
  float xhaty[2];
  xhaty[0] = F[0][0] * xhatpy[0] + F[0][1] * xhatpy[1] + B[0] * gyry;
  xhaty[1] = F[1][0] * xhatpy[0] + F[1][1] * xhatpy[1] + B[1] * gyry;

  float P_predy[2][2];
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      P_predy[i][j] = F[i][0] * P_esty[0][j] + F[i][1] * P_esty[1][j] + Q[i][j];
    }
  }

  // Measurement update (corrector)
  float Ky[2]; // Kalman gain
  float Sy = H[0] * (P_predy[0][0] * H[0] + P_predy[0][1] * H[1]) + R;
  Ky[0] = (P_predy[0][0] * H[0] + P_predy[0][1] * H[1]) / Sy;
  Ky[1] = (P_predy[1][0] * H[0] + P_predy[1][1] * H[1]) / Sy;

  // Update state estimate
  xhatpy[0] = xhaty[0] + Ky[0] * (measurementy - H[0] * xhaty[0]);
  xhatpy[1] = xhaty[1] + Ky[1] * (measurementy - H[0] * xhaty[0]);

  // Update covariance
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      P_esty[i][j] = (1 - Ky[i] * H[j]) * P_predy[i][j];
    }
  }

  // Print results for debugging
  // Serial.print("Estimated State: ");
//   Serial.print(xhatpy[0]);
//   Serial.print(", ");
//   // Serial.println(xhatpy[1]);
//   // Serial.print(", ");
//   Serial.print(theta_roll * 57.2957795 - 90);
//   Serial.println();
//   delay(100); // Wait for readability


  // Serial Print
  Serial.print("Orientation: ");
  Serial.print(xhatpx[0], 6); // multply by 57.295779513082320876798154814105 for
degrees
  Serial.print(", ");
  Serial.print(xhatpy[0], 6);
  Serial.print(", ");
  Serial.print(0);
  Serial.println();

}
```

CovarianceCalc.m

```
% Michael Pittenger
% EE 782 Final Project
% Covaraince calculation

clc; clear; close all;

% Import general reading data
gen_data = readtable('general_readings.csv');
m_data = readtable('movement_readings.csv');

% Column extraction
gen_accx = gen_data(:,4);
gen_accy = gen_data(:,5);
gen_accz = gen_data(:,6);
gen_gyrx = gen_data(:,7);
gen_gyry = gen_data(:,8);
gen_gyrz = gen_data(:,9);

gen_pitch = gen_data{:, 2};
gen_roll = gen_data{:, 3};
m_pitch = m_data{:, 2};
m_roll = m_data{:, 3};

% Covariance calculation for each column
Qpitch = var(gen_pitch)
Qroll = var(gen_roll)
Qbiasx = var(gen_gyrx);
Qbiasy = var(gen_gyry);
Rpitch = var(m_pitch)
Rroll = var(m_roll)

bias_accx = mean(gen_accx)
bias_accy = mean(gen_accy)
bias_accz = mean(gen_accz)
bias_gyrx = mean(gen_gyrx)
bias_gyry = mean(gen_gyry)
bias_gyrz = mean(gen_gyrz)
```

SingleVariableKF.m

```
% Michael Pittenger
% EE 782 Final Project
% Single Variable Kalman Filter

clc;
clear;
close all;

% Parameters
Phi = [1];  % State transition matrix (1x1)
[m, n] = size(Phi);  % State size is 1
B = [0];  % Input matrix (1x1)
H = [1];  % Measurement matrix (1x1)
Q = [0.00141757967789652];  % Process noise covariance for pitch
Q = [0.000908186547736352]; % Process noise covariance for roll
R = 0.3 * eye(n);  % Measurement noise covariance (1x1)

% Load data
data = readmatrix('movement_readings.csv');
measurements = data(:, 2);  % 2 for pitch, 3 for roll
num_steps = length(measurements);  % Number of time steps

% Initialization
x = zeros(n, num_steps);  % True state (1xnum_steps)
xhatp = zeros(n, num_steps);  % Predicted state (1xnum_steps)
P_est = zeros(n, n, num_steps);  % Error covariance (1x1xnum_steps)
RMS_error = zeros(1, num_steps);

% Initial conditions
x(:, 1) = measurements(1);  % Initialize true state with first measurement
xhatp(:, 1) = measurements(1);  % Initialize estimated state
P_est(:, :, 1) = zeros(n, n);  % Initial error covariance

for k = 2:num_steps
    % Measurements
    x(:, k) = measurements(k);  % True state from data
    z = H * x(:, k);  % Measurement (scalar)

    % Predictor
    xhat = Phi * xhatp(:, k-1) + B;
    P_pred = Phi * P_est(:, :, k-1) * Phi' + Q;

    % Corrector
    K = P_pred * H' / (H * P_pred * H' + R);  % Kalman gain
    xhatp(:, k) = xhat + K * (z - H * xhat);  % State estimate
    P_est(:, :, k) = (eye(n) - K * H) * P_pred;  % Update error covariance

    % RMS Error
    RMS_error(k) = sqrt(trace(P_est(:, :, k)));
end

% Plotting
time = 1:num_steps;

figure;
for i = 1:n
    subplot(n + 1, 1, i); % Additional subplot for RMS Error
    plot(time, x(i, :), 'k', 'DisplayName', 'True State');
    hold on;
    plot(time, xhatp(i, :), 'rx', 'LineWidth', 0.5, 'DisplayName', 'Estimate');
    sigma = sqrt(squeeze(P_est(i, i, :)))';
    plot(time, xhatp(i, :) + 2 * sigma, 'b--', 'DisplayName', '95% CI');
    plot(time, xhatp(i, :) - 2 * sigma, 'b--');
    xlabel('Time Step');
    ylabel(['State x_' num2str(i)]);
    legend;
end

% Add RMS Error to the last subplot
subplot(n + 1, 1, n + 1);
plot(time, RMS_error, 'b.', 'DisplayName', 'RMS Error');
xlabel('Time Step');
ylabel('RMS Error');
legend;

ylim([0 1]);
```

WithBias.m

```
% Michael Pittenger
% EE 782 Final Project
% Sensor Fusion and Biasing Kalman Filter

clc;
clear;
close all;

for j = 2:3
    angle = j; % 2 for pitch, 3 for roll

    % Parameters
    deltat = 0;
    F = [1 -deltat; 0 1];  % State transition matrix (2x2)
    [m, n] = size(F);  % State size is 2
    B = [deltat; 0];  % Input matrix (2x1)
    H = [1 0];  % Measurement matrix (1x2)
    if angle == 2
        Q = [0.00141757967789652 0; 0 0.03]; % Process noise covariance for pitch
    else
        Q = [0.000908186547736352, 0; 0, 0.03]; % Process noise covariance pitch and roll
    end

    R = 0.3;  % Measurement noise covariance (scalar)

    % Load data
    data = readmatrix('movement_readings.csv');
    measurements = data(:, angle); % 2 for pitch, 3 for roll
    num_steps = length(measurements);  % Number of time steps
    gyro = data(:, angle+5); % gyro readings in the x and y axis directions

    % Initialization
    x = zeros(n, num_steps);  % True state (2xnum_steps)
    xhatp = zeros(n, num_steps);  % Predicted state (2xnum_steps)
    P_est = zeros(n, n, num_steps);  % Error covariance (2x2xnum_steps)
    RMS_error = zeros(1, num_steps);

    % Initial conditions
    x(1, 1) = measurements(1)';  % Initialize true state with first measurement
    xhatp(1, 1) = measurements(1)';  % Initialize estimated state
    P_est(:, :, 1) = zeros(n, n);  % Initial error covariance

    for k = 2:num_steps
        % Calculate change in time between measurements
        deltat = (data(k, 1) - data(k-1, 1)) * 0.1;  % Change in time

        % Measurements
        x(:, k) = measurements(k, :)';  % True state from data
        z = H * x(:, k);  % Measurement (scalar)

        % Predictor
        xhat = F * xhatp(:, k-1) + B * gyro(k-1)';
        P_pred = F * P_est(:, :, k-1) * F' + Q;

        % Corrector
        K = P_pred * H' / (H * P_pred * H' + R);  % Kalman gain
        xhatp(:, k) = xhat + K * (z - H * xhat);  % State estimate
        P_est(:, :, k) = (eye(n) - K * H) * P_pred;  % Update error covariance

        % RMS Error
        RMS_error(k) = sqrt(trace(P_est(:, :, k)));
    end

    % Plotting
    time = 1:num_steps;

    figure;
    for i = 1:n
        subplot(n + 1, 1, i); % Additional subplot for RMS Error
        plot(time, x(i, :), 'k', 'DisplayName', 'True State');
        hold on;
        plot(time, xhatp(i, :), 'rx', 'LineWidth', 0.5, 'DisplayName', 'Estimate');
        sigma = sqrt(squeeze(P_est(i, i, :)))';
        plot(time, xhatp(i, :) + 2 * sigma, 'b--', 'DisplayName', '95% CI');
        plot(time, xhatp(i, :) - 2 * sigma, 'b--');
        xlabel('Time Step');
        ylabel(['State x_' num2str(i)]);
        legend;
    end

    % Add RMS Error to the last subplot
    subplot(n + 1, 1, n + 1);
    plot(time, RMS_error, 'b.', 'DisplayName', 'RMS Error');
    xlabel('Time Step');
    ylabel('RMS Error');
    legend;
    ylim([0 1]);
end
```