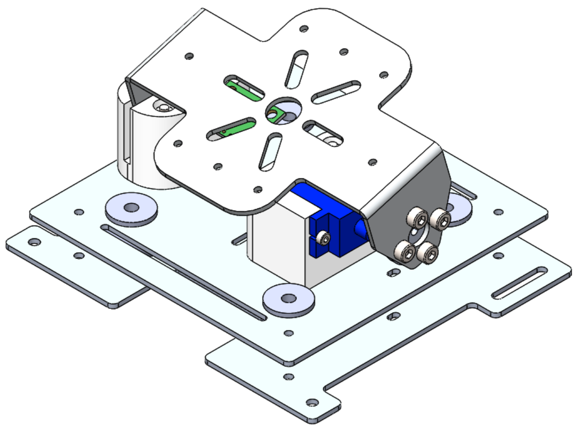
Auto-Leveling Gimbal
Technical Documentation

Davis Drone Club

September 10, 2018



Contents

1 Mechanical Design 1
  1.1 Sheet Metal Parts 1
  1.2 3D Printed Parts 1

2 Electrical Systems 2
  2.1 Microcontroller 2
  2.2 Inertial Measurement Unit 3
  2.3 Actuator 4

3 Algorithm Design 5
  3.1 Orientation Filter Algorithm 7
  3.2 Testing the Quaternion-Based Filter 10

4 Arduino Code 14
  4.1 Union Data Type 14
  4.2 Reading and Writing EEPROM 14
  4.3 Non-Blocking Serial Input 15
  4.4 Watchdog Timer 15

List of Figures

1 Sheet Metal Plates 1
2 Hardware Flowchart 2
3 Overall Software Flowchart 6
4 Euler and Quaternion Orientation Representation 8
5 Quaternion-Based Complimentary Filter Flowchart 9
6 Results of Varying Base Gain 11
7 Results of Varying ϵ 12
8 Results of Varying Threshold Range 13

List of Tables

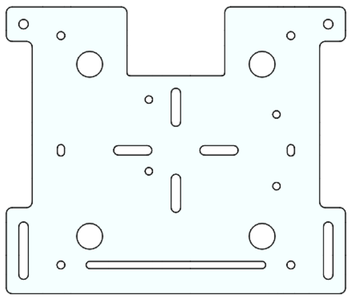
1 Arduino Nano Pin Assignments 3

1 Mechanical Design

The gimbal was designed to stabilize a small camera in one axis. To simplify the construction of the gimbal, a servo motor was used as an actuator and as one of the two axles for the rotating plate. The servo motor is mounted to a 3D printed mount, which is screwed to a sheet metal plate. Shock absorption balls are used to reduce vibration, and are mounted between two sheet metal plates. To maximize ground clearance to the camera, the gimbal assembly is mounted to the DJI Matrice 100 bottom battery mounting rails.

1.1 Sheet Metal Parts

The three sheet metal parts were designed using Solidworks and made with 1/16" aluminium cut at the UC Davis ESDC waterjet cutter. The parts are designed to mimic the design and hole layout of the expansion bay plates for the DJI M100, allowing for similar devices to be bolted onto either the expansion plate or the fabricated plates.


(a) Matrice Plate

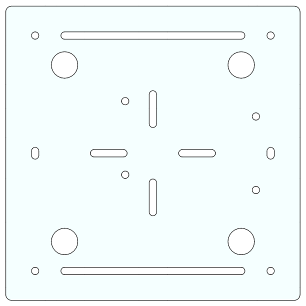

(b) Gimbal Plate

Figure 1: Sheet Metal Plates

1.2 3D Printed Parts

3D printed parts were designed with Solidworks and printed at the UC Davis Engineering Student Startup Center. Each file was exported as an .STL file, and processed with slicing software at the ESSC prior to being printed.

The servo adapter is used to attach the camera plate to the servo motor. The included plastic servo horn is fixed to the servo adapter using 2-part epoxy, with the toothed side of the servo horn facing away from the servo adapter.

2 Electrical Systems

The electrical system of the gimbal consists of an Arduino Nano microcontroller mounted to a piece of perforated prototyping board. Tracks on the perforated board distribute power from a 5v output Battery Elimination Circuit (BEC) used to lower the 22v Matrice power output to 5v. Figure 2 shows how the various hardware components of the gimbal are connected to one another.

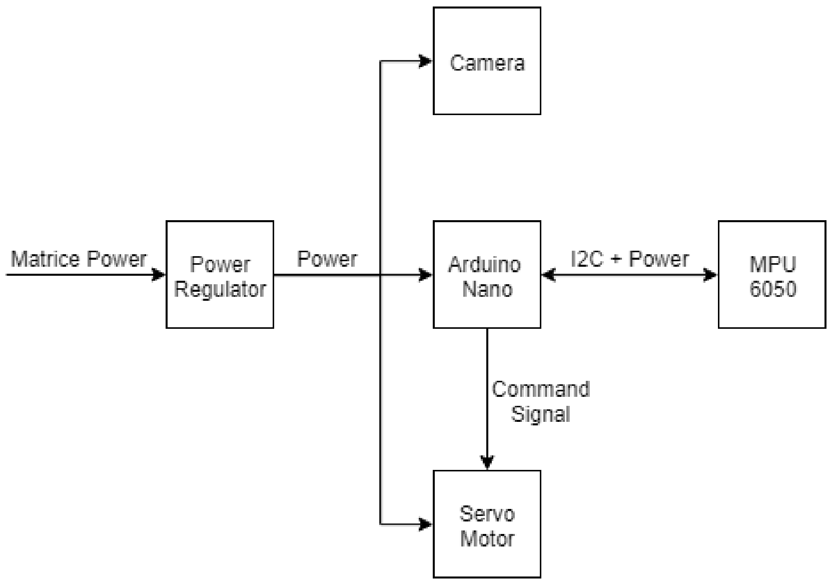


Figure 2: Hardware Flowchart

2.1 Microcontroller

The Arduino Nano microcontroller was chosen to operate the gimbal because of wide software support for Arduino type boards and because its small form factor would allow for convenient mounting on the drone or gimbal. Alternative options such as the Arduino Pro Mini were not used because such boards did not offer USB connectors, and had to be programed with additional FTDI programming chips.

2.1.1 Pin Assignments

The I2C SDA and SCL pins must be connected to pins A4 and A5 respectively due to the hardware of the Arduino Nano, however; the servo output wire can be changed to any arbitrary digital output pin as long as the software is updated to the correct pin.

Table 1: Arduino Nano Pin Assignments

Pin	**Description**
A4	MPU6050 SDA
A5	MPU6050 SCL
D3	Servo out wire
D13	Arduino built-in LED

2.2 Inertial Measurement Unit

An Intertial Measurement Unit is an integrated circuit which can measure orientation and angular velocities. Initially, an MPU9250 was used on the gimbal, however; this was switched to the MPU6050 sensor due to reliability issues with the MPU9250.

The MPU6050 is a six degree-of-freedom IMU, meaning that there are three acceleration axes and three gyroscope axes. As compared to the MPU9250, which is a nine degree-of-freedom IMU, with three acceleration, gyroscope, and magnetometer axes. While the magnetometer on the 9DOF IMU's can be useful for determining the yaw of the craft in a global reference frame, the one-axis gimbal can function with the 6DOF sensor.

The MPU6050 communicates via the I2C bus, which allows multiple sensors to be connected along the same two wire bus. The sensitivity of the gyroscope or accelerometer can be separately set by adjusting register values as described in the MPU6050 Register Map and Descriptions document. Data is retrieved by reading specific registers allocated to accelerometer/gyroscope data. While the measurements are 16bit integers, the MPU6050 registers are 8bits, with the 16bit measurement split across two registers in Most Significant Bit (MSB) first format, meaning that bits 15 to 8 are stored in the first register, and bits 7 to 0 are stored in the second. To combine these two, shift the first byte "left" by 8 bits and perform a logical and with the second byte.

2.3 Actuator

A servo motor was chosen to actuate the gimbal because it allowed for simple control over the angle between the camera plate and the gimbal plate. Unlike brushless DC (BLDC) motors, servo motors do not require special driver circuits to operate, and only require power, ground, and signal connections, however; servo motors have worse resolution when compared to BLDC motors, which limits the effectiveness of the gimbal.

The FMS 9g servo was used because it offered metal gears, which should provide better durability compared to plastic servos, however; the metal gear teeth may erode plastic servo horns, resulting in the camera plate and servo motor decoupling.

3 Algorithm Design

The algorithm for running the gimbal can be split into two major sections: the setup procedure and loop process. In the setup procedure, serial communication, I2C communication, the watchdog timer, and the PID class is instantiated. Gyroscope calibration values, PID values, and deadband values are also read from EEPROM memory. The loop process consists of reading data from the Inertial Measurement Unit (IMU), and processing the gyroscope and accelerometer values to yield a pitch angle. The pitch angle is then fed into the PID controller, which determines an output value for the servo. The servo is updated based on this value. The serial buffer is then checked for incoming commands, and the program enters an empty while loop to limit the loop frequency to 180Hz.

Figure 3 represents this chain of operations in a flowchart.

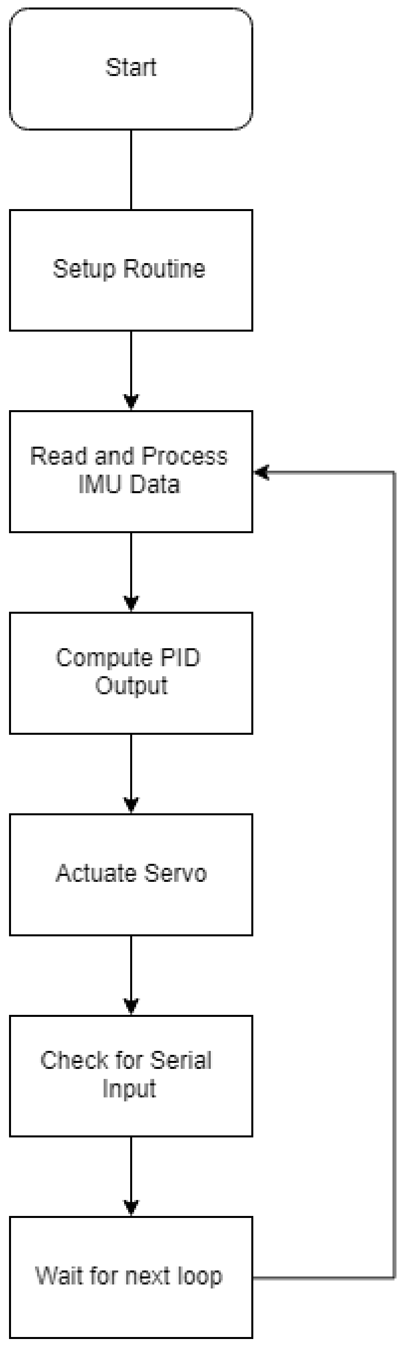


Figure 3: Overall Software Flowchart

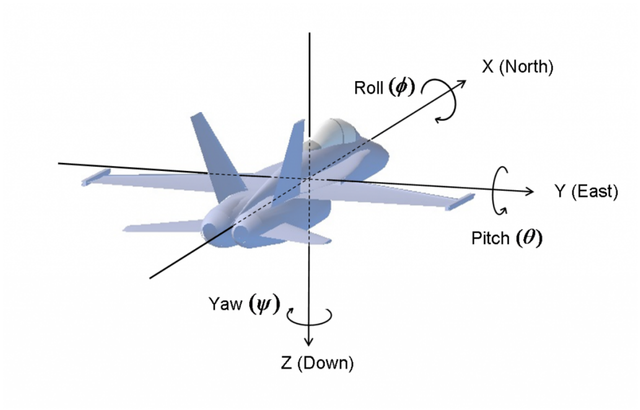
3.1 Orientation Filter Algorithm

The core of the gimbal's operation is the algorithm for processing data from the Inertial Measurement Unit (IMU). The key problem lies in combating the main issues regarding the two sensors available: gyroscope readings are precise, but drift over time; accelerometer readings are accurate, but suffer from noise and external forces. To get an accurate and precise measurement of the IMU's orientation, one must use an algorithm that combines data from the two sensors.

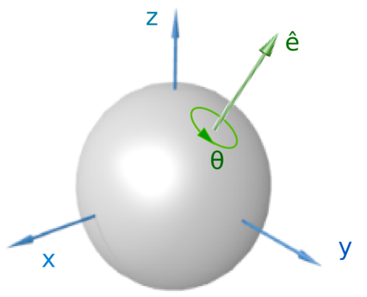
In gimbal firmware 2.2 and below, a simple Euler angle complimentary filter, based on the work of Joop Brokking, was used to compute the pitch angle of the IMU. While this approach offered an intuitive approach to processing data, the Euler angle method is susceptible to gimbal lock and suffered from large angle errors when subjected to high non-gravitational accelerations. To address this issue, an adaptive gain quaternion-based complimentary filter was implemented, based on the work of Roberto Valenti, Ivan Dryanovski, and Jizhong Xiao, published in the article "Keeping a Good Attitude: A Quaternion-Based Orientation Filter for IMUs and MARGs". The following section will detail the implementation of this algorithm, and draw comparisons between the Euler angle and Quaternion algorithm.

3.1.1 Implementing the Quaternion-Based Complimentary Filter

Quaternions are four-element vectors that describe the rotation of objects in 3D space. As opposed to Euler angles, which describe the amount of rotation in the yaw, pitch, and roll axes of a system, three of the four elements in the quaternion define a vector for the axis of rotation, while the fourth, scalar, element defines how much the object is rotated about that axis. Figure 4 shows how the two types of representation differ. Further details on quaternion orientation representation and mathematical handling can be found here.



(a) Euler Representation [Source]



(b) Quaternion Representation [Source]

Figure 4: Euler and Quaternion Orientation Representation

The algorithm can be split into a number of steps

1. Measure IMU and form quaternion ${}^{L}\omega_t = [0, \omega_x, \omega_y, \omega_z]$ where ω represents a reading from the IMU.

2. Rotate ${}^{L}\omega_t$ from local frame to global frame with previously calculated heading quaternion.
   $-\frac{1}{2}{}^{L}\omega_{q,t} \otimes^{L}_{G} q'_{t-1} =^{L}_{G} \dot{q}_{t-1}$

3. Integrate numerically to get gyroscope measurement of global frame orientation relative to local frame at time t.
   ${}^{L}_{G}q_{\omega,t} =^{L}_{G} q_{t-1} +^{L}_{G} \dot{q}_t \Delta t$

4. Convert the measured orientation vector to a 3x3 rotation matrix.

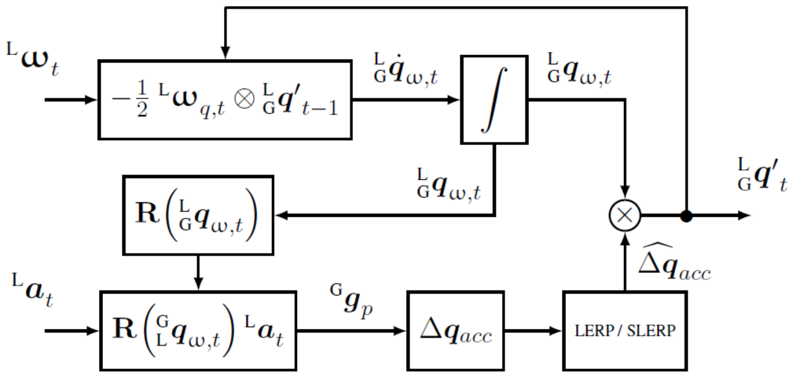
5. Rotate the normalized measured acceleration vector, ${}^{L}a_t$, by rotation matrix $R({}^{G}_{L}q_{\omega,t})$ $R({}^{G}_{L}q_{\omega,t}){}^{L}a_t \; =^{G} \; g_p$, where ${}^{G}g_p$ is the predicted acceleration vector based on the gyroscope estimation. If the gyroscope measurement is completely accurate, the predicted acceleration vector should be $[0, 0, 0, 1]$.

6. Because of measurement error and drift, the predicted acceleration vector will likely not be $[0, 0, 0, 1]$. Calculate the rotation quaternion required to rotate ${}^{G}g_p$ vector to $[0, 0, 0, 1]$.
$\Delta q_{acc} = \left[\sqrt{\frac{g_z+1}{2}}, -\frac{g_y}{\sqrt{2(g_z+1)}}, \frac{g_x}{\sqrt{2(g_z+1)}}, 0\right]$

7. Run LERP/SLERP complimentary filter on Δq_{acc} to yield $\widehat{\Delta q}_{acc}$

8. Take product of gyroscope orientation quaternion and acceleration compensation quaternion to yield final orientation quaternion.
${}^{L}_{G}q'_t = {}^{L}_{G}q_{\omega,t} \otimes \widehat{\Delta q}_{acc}$

Figure 5 arranges these steps into a flowchart. More detail about the mathematics can be found on pages 19315-19318 of the "Keeping a Good Attitude" paper.



${}^{L}\boldsymbol{\omega}_t$
$-\frac{1}{2}\,{}^{L}\boldsymbol{\omega}_{q,t} \otimes {}^{L}_{G}\boldsymbol{q}'_{t-1}$
${}^{L}_{G}\dot{\boldsymbol{q}}_{\omega,t}$
$\int$
${}^{L}_{G}\boldsymbol{q}_{\omega,t}$
${}^{L}_{G}\boldsymbol{q}_{\omega,t}$
${}^{L}_{G}\boldsymbol{q}'_t$
$\mathbf{R}\left({}^{L}_{G}\boldsymbol{q}_{\omega,t}\right)$
$\widehat{\Delta \boldsymbol{q}}_{acc}$
${}^{L}\boldsymbol{a}_t$
$\mathbf{R}\left({}^{G}_{L}\boldsymbol{q}_{\omega,t}\right){}^{L}\boldsymbol{a}_t$
${}^{G}\boldsymbol{g}_p$
$\Delta \boldsymbol{q}_{acc}$
LERP / SLERP

Figure 5: Quaternion-Based Complimentary Filter Flowchart

3.1.2 Implementing Adaptive Gain

While quaternions can resolve issues with gimbal lock (though not actually a concern for a one-axis gimbal), adaptive gain is used to partially resolve issues arising from angle deviations when the IMU is subjected to large, non-gravitational accelerations. Adaptive gain changes the influence the accelerometer has on the final reading, based on a calculated magnitude error. If the error is large, the algorithm will depend heavily on the gyroscope, which is not affected by non-gravitation forces. If the error is small, the algorithm will use both gyroscope and accelerometer readings to combat noise and drift.

In summary, the adaptive gain function changes the α value dynamically. The value for α can range from 0 to a pre-determined base rate $\bar{\alpha}$ depending on a function that depends on e_m, the magnitude of error.

$$\alpha = \bar{\alpha} f(e_m)$$
$$e_m = \frac{|\|^L\tilde{a}\| - g|}{g}$$

For e_m, $\|^L\tilde{a}\|$ represents the magnitude of the acceleration measured by the IMU in meters per second, prior to being normalized. The function used is a parametric function

$$\begin{cases} e_m < L, & 1 \\ L < e_m < U, & \text{linear between 0 and 1} \\ U < e_m, & 0 \end{cases}$$

where L is a lower threshold, and U is a upper threshold, both determined experimentally.

Once again, more details can be found in the "Keeping a Good Attitude" paper on page 19320.

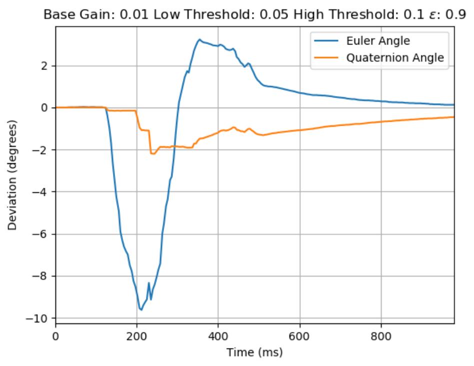
3.2 Testing the Quaternion-Based Filter

A perfect orientation estimation algorithm would be able to filter out all external non-gravitational forces. In theory, the adaptive gain algorithm should provide performance benefits for orientation estimation. In practice, the adaptive gain can help combat drift when the aircraft is accelerating/decelerating in windy conditions, or when the aircraft is entering/exiting turns.

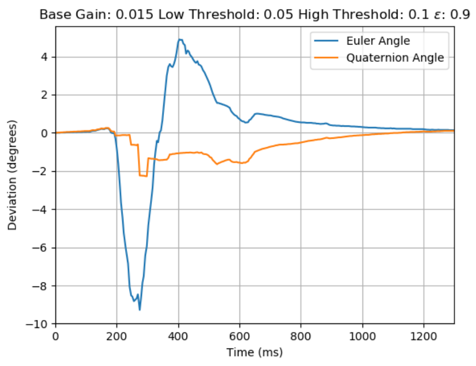
To test the resilience of the quaternion-based filter against external non-gravitational forces, an Inertial Measurement Unit (IMU) was placed on a flat surface, and quickly accelerated across the surface. Parameters for base gain, lower threshold, upper threshold, and ϵ were changed and compared to an Euler estimation algorithm.

3.2.1 Varying Base Gain

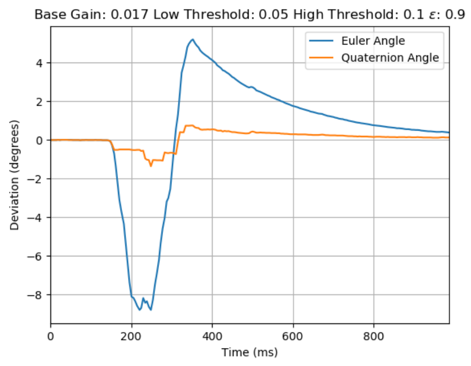
Base Gain $\bar{\alpha}$ seemed to have a large impact on the algorithm's performance. Figure 6 compares the quaternion based algorithm (orange) and Euler angle algorithm (blue) at base gain between 0.01 to 0.05.



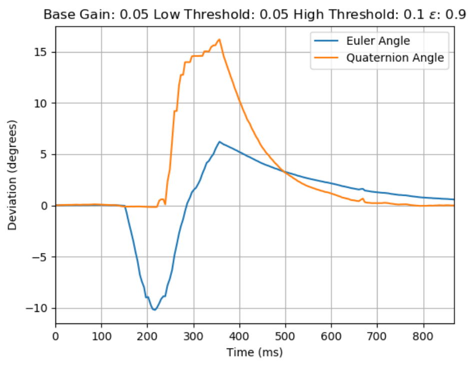
(a) Base Gain = 0.01



(b) Base Gain = 0.015



(c) Base Gain = 0.017



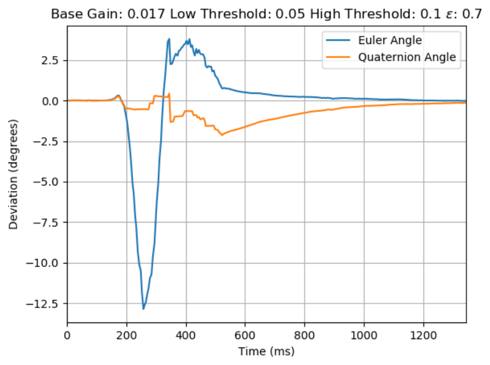
(d) Base Gain = 0.05

Figure 6: Results of Varying Base Gain

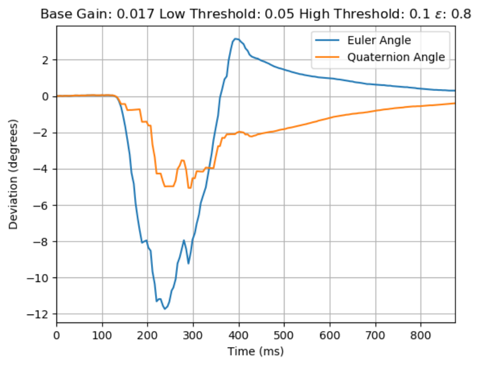
Large base gains can result in the algorithm overshooting, and performing worse than the regular Euler angle algorithm. In contrast, low base gains result in slow compensation and an elongated response. The run performed at base gain = 0.017 performed well, exhibiting less deviation than other trials and recovering sooner than the Euler algorithm.

3.2.2 Varying ϵ

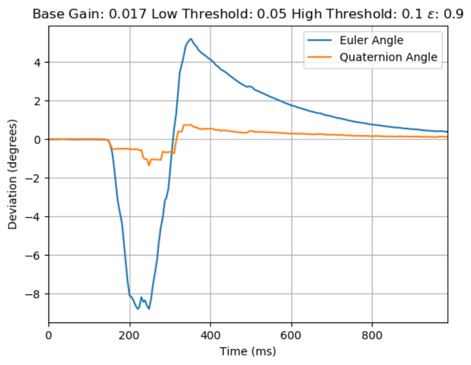
Changing ϵ seems to have a similar impact to performance as base gain. Figure 7 compares the quaternion based algorithm (orange) and Euler angle algorithm (blue) at ϵ between 0.7 to 0.9.



(a) $\epsilon = 0.7$



(b) $\epsilon = 0.8$



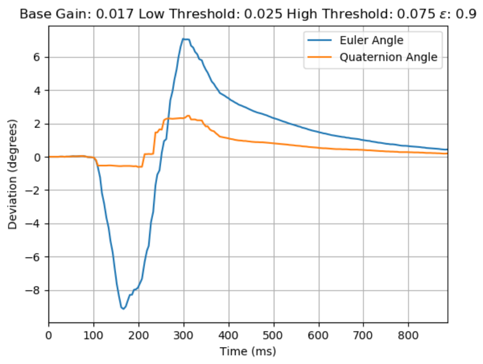
(c) $\epsilon = 0.9$

Figure 7: Results of Varying ϵ

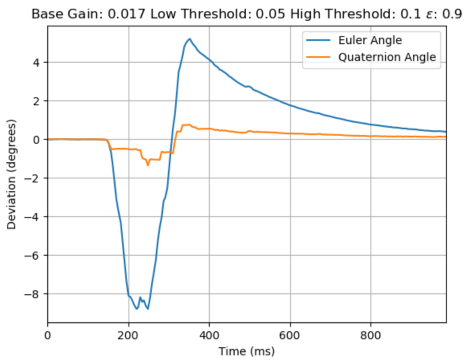
It seems that larger values of ϵ, around 0.9, perform better than lower values, however; it is interesting that $\epsilon = 0.8$ performs worse than $\epsilon = 0.7$. Further testing may be required.

3.2.3 Varying Threshold Values

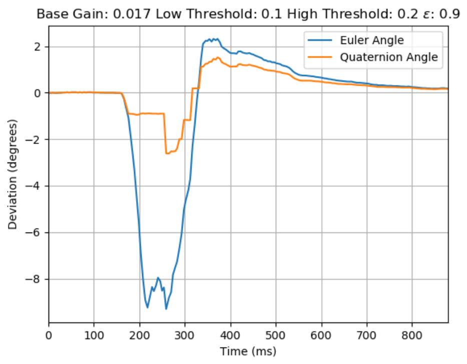
Three sets of threshold values were tested. 0.025-0.075, 0.05-0.1, and 0.1-0.2. Figure 8 compares the quaternion based algorithm (orange) and Euler angle algorithm (blue) for the three threshold sets.



(a) L = 0.025, U = 0.075



(b) L = 0.05, U = 0.1



(c) L = 0.1, U = 0.2

Figure 8: Results of Varying Threshold Range

It would appear that higher ranges of threshold cause the graph of the quaternion-based algorithm to act like the Euler algorithm. Lower threshold ranges result in smoother, but delayed responses. Further testing may be required to see how the shape of the response curve changes with threshold values.

4 Arduino Code

Software for the gimbal was written in Arduino code, which is based on the C++ language, on the Arduino IDE. The code can be uploaded to the Arduino Nano through the Arduino IDE. Files can be found on the Davis Drone Club Github. The gimbal code is designed to run on the 16MHz Arduino Nano. Arduino code is based on C and C++, so many resources for C/C++ can be used to understand Arduino code. This section will cover concepts of the code that may not be covered in most learning resources.

4.1 Union Data Type

The Union data type is used to store the value of numbers in both float and byte formats. This is useful because the PID and orientation algorithms use data type float for floating point calculations, while the EEPROM data storage requires data in bytes. The Union provides a convenient structure to store both data types under a single variable name. A detailed guide on unions can be found here.

```
int gyro_x, gyro_y, gyro_z;
float acc_x, acc_y, acc_z, acc_total_vector;
int temp;
float loop_timer;
```

The union is defined in the same way a structure is defined. First use the union statement, followed by the name of the union type. Then in the block enclosed by the brace, include a list of member definitions. In this case, we use float val_float and byte val_byte[4] to denote a float value and a 4 byte value (Arduino floats are 32-bit). After the braces, list the names of the variables. In this case we list kp, ki, kd, etc.

To access the data from a union, use the member access operator (.) by first indicating the variable, followed by the period, and then the desired member definition. For example, kp.val_float will yield the float value of kp, while kp.val_byte[0] will yield the first byte of kp.

4.2 Reading and Writing EEPROM

EEPROM is a section of non-volatile memory on the Arduino board, meaning that values stored in EEPROM are not lost when the board is powered off or reset. On the Arduino Nano there are 1024 bytes of EEPROM memory. One consideration when using EEPROM is that EEPROM has limited write cycles. After approximately 100,000 writes, there is a chance that memory will be corrupted or unreadable, however; there are no read limitations for EEPROM memory. Therefore, it is inadvisable to constantly write data the EEPROM. In this case, EEPROM data is only written when certain parameters are changed and read during setup.

To write to the EEPROM memory, use the EEPROM.h library. The EEPROM.update function further reduces write cycles by checking the value of the memory space first. If the data to be written is the same as the data already present, the memory will not be changed, saving one write cycle. The update function takes two parameters, an index and value.

More data about the EEPROM library can be found on the Arduino documentation.

4.3 Non-Blocking Serial Input

Non-blocking functions are functions that do not stop the execution of other commands. Non-blocking functions are especially useful for time sensitive programs with functions that take a long time to execute. In this case, reading and writing data to serial takes a considerable amount of time. In firmware version 2.2 and up, the reading functionality is non-blocking, as adapted from the Majenko Technologies Blog.

4.4 Watchdog Timer

The Arduino Watchdog Timer is a feature to avoid unwanted hangs. The watchdog timer will force the microcontroller to reset if a certain loop timer exceeds a given threshold time. For instance, if a Serial.println() command causes the Arduino to hang, the watchdog timer will automatically reset the Arduino. While the watchdog timer will not help with errors in the implementation of the code, as these errors will occur regardless of how many times the board is reset, the timer is helpful for preventing short term outages, such as issues with power supply or short circuits.

A useful resource for learning about implementing the watchdog timer can be found here.