1. INTRODUCTION

1.1 General Introduction

A Quadcopter is a multicopter that is propelled and lifted by four propellers (rotors). Opposed to fixed-wing aircraft (most common example is an airplane), a Quadcopter's lift is generated by revolving narrow-chord airfoils, symmetrically placed, are adjusted as a group. Control of motion is attained by altering the pitch and/or rotation rate of one or more propellers.

One reason a Quadcopter is seen as an alternative to some problems in vertical flight is because of torque-induced control issues (mostly the issue dealing with the tail rotor which generates no meaningful lift). Other advantages over a helicopter are: not requiring mechanical linkages to vary the propeller pitch angle as they spin, by using more propellers each individual propeller can be smaller in diameter possessing less kinetic energy, and by using smaller propellers, this reduces the damage done by the propellers should they come into contact with anything. The Helicopters use two propellers to control flight of pitch, roll, and yaw. With four propellers, there are two pairs of propellers and each pair rotates opposite direction. If one pair is slowed down then the Quadcopter will rotate in the same direction of the slowed down propellers, due to the opposite two motors greater speed overpowering the current slower motors.

The Quadcopter can have other applications instead of just a fun recreational model. One use is for the military, instead of sending someone into a dangerous area, the Quadcopter equipped with a camera would fly in and gather valuable reconnaissance information. Small-scale models can lead to the innovations in a full scale Quadcopter potentially solving the aforementioned flight issues presented by helicopters, maybe one-day evening replacing helicopters as a secondary means of air travel (behind airplanes).

The Quadcopter is not as expensive as the larger scale models, which range from a few inches to a few feet in diameter. The models can help people learn how the physics of a Quadcopter work and how it differs from other multicopter technology and other fixed-wing aircraft. The project encompasses many engineering issues and tasks that vary from physics, software, and hardware.

The software side is much different from the hardware. There are various things that need to done in software; for example, controlling the motors, interpreting controller inputs, communicating with the

Quadcopter, and various other algorithms required to fly. Software is also required for semi-autonomous flight, safety systems, and landing.

1.2 Overview of Concept

The Quadcopter as a bare machine is of very little use, but it is the power of algorithms running on integrated hardware that makes the Quadcopter a viable product for industrial and commercial use. Generally, by algorithms, we mean the software platform running on a processor, but Quadcopter falls under embedded system category where algorithms not only maintain the working of central processor which is microcontroller but also of wireless communication devices and IMU (Inertial Measurement Unit) devices. The wireless communication devices itself has a microprocessor which provide error detection and flow control in transmission and also the IMU device has high performance microprocessor to provide reliable data for calculations of yaw, pitch and roll angles for the Quadcopter. The Quadcopter has 4 degrees of freedom i.e. yaw – rotation about axis perpendicular to Quadcopter plane, pitch – rotation about y axis, roll – rotation about x –axis and throttle which is acceleration along the common axis of propellers. Some factors concerning the operation of Quadcopter are:

1.2.1 Weight

Weight is another major consideration for the Quadcopter, which is relative to the size. The design goal is to keep the weight very less, the chassis needs to be made of very lightweight materials, the light metal allows the chassis to be light and has the added benefit of being sturdy enough to support the weight of the motors and batteries. The motors and batteries also add a significant amount of weight to the design, so they must be as light as possible, while continuing to meet other specifications. The motors and batteries account for about half of our target weight limit.

1.2.2 Motors & Propellers

The right electronics are a first step toward creating flight, but a multi-rotor isn't going anywhere without good old-fashioned practical physics to pull it upwards. Your choice in motors plays a pivotal role in the success of a capable setup.

This is also the point when specifications start to get complicated. You'll want to do some research before settling on the right configuration for your quad. And motors are expensive, making it even more important to consider the options carefully.

Motors used these days are almost exclusively of the “brushless” variety. That equates to minimal friction. A cylindrical shell of magnets rotates on precision bearings around a core of tightly and neatly coiled wire. The propeller is fastened atop. Many Tom's Hardware readers already know the composition of an electrical motor, but for enthusiasts dabbling in multi-rotors, the inner workings are unimportant. So long as reasonable care is taken and dirt kept clear of the bearings, brushless motors are famously reliable.

Motors are assigned various notations, the most consequential being the Kv rating. Confusingly, Kv does not refer to kilovolts in this case. Rather, it's a motor velocity constant denoting the revolutions per minute (RPM) that a motor will turn when a 1 V potential difference is applied with zero load. This number is important, as it defines a multi-rotor's flight characteristics based on specifications like battery voltage and take-off weight.

Also vital are the propellers you choose. The variety of props is arguably greater than any other component we discuss; materials, dimensions, and price span a mind-bogglingly wide range. Generally, cheaper props are less precisely manufactured and more prone to creating vibration. This applies especially to the relatively larger end of the prop spectrum, with differences becoming less perceptible for smaller craft. Again, some vibration can be acceptable, bolstering the case for less expensive propellers. But if you're flying a quadcopter with the intention of producing well-shot footage, expect to spend more money on propellers.

It's worth pointing out that a majority of props are designed for airplane, though we're starting to see more optimized for multi-rotors. Graupner is a favorite brand amongst enthusiasts, and the company's E-Props, designed for electric power systems, are often cited as favorites. Other common options include Gemfan, APC, T-Motor, and RCTimer.

There are three simple measurements to keep in mind. The first is length, usually given in inches. The higher the Kv of your motors, the smaller your props need to be. Smaller props allow for greater speeds, but reduced efficiency. A larger prop setup (with correspondingly-low Kv motors) is easier to fly steadily, uses less current, and lifts more weight.

The best way to gauge the right range for motors and props is referring to manufacturer recommendations if you're building an ARTF kit. Or you can simply compare the setups of more experienced builders.

The second measurement, prop pitch, is less important, but of interest to more vigorous hobbyists. Prop dimensions are quoted in the form 9x4.7”, as a numerical example. The first number refers to the already-discussed length. The second is pitch, defined as the distance a prop would be pulled forward through a solid in a single full revolution, as if a screw through wood. The greater the pitch, the higher the thrust and necessary motor output. Typically, multi-rotors use props with pitches in the range of 3 to 5”. Lower pitches are more efficient, but lend a more sedate flying style.

Finally, we have bore measurement, which is simply the size of the hole in the center of the prop. This must be matched to the shaft of your chosen motors. Adapters are available to downsize a prop’s bore. Alternatively, some props, such as those produced by T-Motor, use a direct mounting system whereby screws secure the props directly to the motor head.

1.2.3 Electronic speed controllers (ESCs)

Electronic speed controllers (ESCs) are used in many R/C applications. They translate signal to electrical supply. On a multi-rotor, every motor gets its own ESC, each of which connects to the flight controller. After computing the inputs, the controller directs each ESC to adjust its speed in order for the craft to perform them.

ESC refresh rates vary. For multi-rotors, given the balance of multiple motors critical to the craft's ability to stay airborne, high refresh rates are more important than many other hobbies where ESCs are used.

In essence, we're talking about programmable microcontrollers, and they employ firmware to define and carry out their tasks. In the world of multi-rotors, SimonK is the supreme ruler of ESC firmware, creating revisions optimized for multi-rotor use, stripped of irrelevant features, and sporting refresh rates as high as 400 Hz or so. ESCs can be flashed or purchased with SimonK's optimizations pre-loaded.

The only other major factor to consider is an ESC's maximum current rating, which must exceed the current draw to each motor. Generally, 30 A for medium/large quads and 10 to 12 A for a small quad is plenty.

1.2.4 LiPo Battery

Clearly, those are high current draws. But such is the nature of multi-rotors. A medium-sized hex can easily pull 40 A on a steep ascent. As a result, hefty batteries are a necessity for decent flight times.

The industry standard is lithium-ion polymer (LiPo) batteries. Relatively lightweight, compact, and offering high discharge rates, LiPos are well-suited for multi-rotors.

Ready for another set of specifications? There are three to consider as you start perusing the cyber-aisles of LiPo batteries. The first is voltage. A single cell supplies a nominal voltage of 3.7 V (4.2 V at full charge). Each additional cell wired in series adds 3.7 V to the nominal voltage of that pack. Cell counts are denoted by the number of cells followed by "S". A 4S LiPo, therefore, is a battery of four 3.7 V cells at a summation of 14.8 V.

LiPo packs also have C ratings that indicate the maximum rate at which a pack can be discharged, with C standing for capacity. A 25C pack can be discharged at a rate 25 times its capacity.

Capacity, therefore, is the third important factor. It's measured in milliamp-hours (mAh). Let's say our 20C pack has a capacity of 4000 mAh. Given what we know about C ratings, we can do the math and determine its maximum discharge at up to 80,000 mA, or 80 A. Similar to ESCs, you need a discharge rate that's higher than the combined draw current of your motors.

LiPos connected in parallel add to capacity (rather than affecting voltage). In turn, the aforementioned S notation is modified. A 3S2P arrangement, for example, consists of two three-cell LiPos connected in parallel.

Batteries do not last forever. They vary in cost, and the pricier LiPos typically last for more cycles than the cheaper ones. A pack will “puff” in its plastic wrap as it gets to the end of its rope. Excessive heat after use is another bad sign.

The best way to prolong a LiPo’s life is to follow the 80% rule. You should try to avoid discharging more than 80% of the battery's listed capacity (a maximum of 4000 mAh from a 5000 mAh pack, for example). Also, monitor voltage when you're flying, and land before reaching 3.3 V per cell. Voltage falls more rapidly as charge is depleted, and at 3 V per cell, you might drop out of the sky. Some flight controllers have protection mechanisms to help prevent over-discharge.

1.2.5 Safety

Safety is one of the major issues with any flying device. Such devices can become dangerous when traveling at high enough speeds. Though it may not injure someone, it can be painful if one is hit by the Quadcopter. To limit the potential damages, the velocity will be limited to minimum level. This level depends on the final weight of the device, and needs a thorough analysis of the force exerted on collision. Another consideration is the performance under low-voltage conditions. When the battery level reaches a certain point, the Quadcopter must land at a safe speed instead of dropping from an unsafe height.

1.2.6 Reliability of Quadcopter

The Quadcopter needs to be semi-autonomous. This means that the Quadcopter will maintain its current position. It should drift very less. The Quadcopter will be able to self-correct when dropped from a certain height before it reaches the ground. Once the battery reaches a critical level, the Quadcopter will begin gently landing itself due to less power being supplied to the motors, and thus each motor will turn slower and the unit will descend until it is on ground level. Using the sensor data from the gyro/accelerometer, the Quadcopter will be able to judge its position as well as its velocity. If a large spike in sensor data indicates that a collision has occurred, the Quadcopter will attempt to stabilize itself. Failing to stabilize should result in an immediate shut down of the motors to prevent further harm to the device, and to whatever it happens to be running into.

1.3 Work Done

:

- **Electrical Wiring** – Wiring the individual electrical components together during building to ensure that proper communication needs between parts can occur.
- **PCB Design** – Schematic designed by Eagle CAD by careful placing of different components required.
- **PCB fabrication** – PCB is fabricated by CNC machine via copper clad.

- **Code Structure** – Maintain the overall structure of the code. Design how individual modules will interact with one-another, and how communications will occur.

- **Scheduler** – Design a scheduler that allows each module's task to complete within 10ms. If a task runs over the allotted time, the scheduler must run the next scheduled task as soon as possible, and continue this trend until all tasks have caught up to the allocated amount of time they should be running in. No task skips shall occur under any circumstances. If a task completes ahead of the allotted time, then the scheduler shall wait until the proper time to continue execution.

- **I2C Communications** – Implement a system for communication between the AVR and sensors. One sensors currently in use have 3.3-volt I2C communications. The AVR has built in hardware support for I2C, which is available. The accelerometer/gyroscope has a secondary set of I2C lines available for consolidating the sensor data into a single device interface.

- **Wireless Communications (Transmission)** – Implement the transmission of data across the wireless transceiver.

- **SPI Communications** – Design the SPI communication between the AVR and the wireless transceiver. The AVR has built in hardware support for SPI; however, there are deep connections to the I2C module. This means that there can be either hardware support for SPI or hardware support for I2C.

- **Balancing Systems** – This module is responsible for determining what corrective actions will keep the Quadcopter stable when in the air. This includes determining the percentage of total power that each motor will run at.

- **Control Interpretation Systems** – This module takes raw data provided by the sensors and turns it into a signal the Quadcopter can use to perform an action (e.g. "Turn Left").

- **Reading Sensor Data** – Communicates with all of the sensors and gathers the data onto the AVR. Place data into structures for other modules to process.

- **Input Filtering** – Filters input received from sensor modules to remove large outliers on data and smooth the changes between values to reduce jitter.

- **Initial Testing** – Responsible for testing the parts received to determine if they meet the requirements needed, and that they are functional. Tests combinations of parts to make sure they are compatible (e.g. Motors and Propellers).

- **Wireless Communications (Receiver, Comm. Methods)** – Implement the receiver of data across the wireless transceiver. Determine methods needed to insure reliable communications with minimal interference.

- **PWM Controls** – This module takes the percentage values for each motor and converts it into a PWM value for the motors.

- **Battery & Power Management** – Determine the optimal batteries available that meet the specifications. This software module is responsible for determining the current battery power available for the safety controls module.

- **Controls** – Connect the Quadcopter to the “Remote Control” AVR via whichever means are required.

- **Control Response Systems** – Adjusts the “corrective vectors” provided to the motors based on the control signal sent to the Quadcopter.

2. Problem Domain

The following are the problem encountered during the project:

- Quadcopter as a flying machine is very unstable in air, several methods and algorithms are required to reduce the instability of Quadcopter in free space or air. Quadcopter if unstable is highly dangerous, if not stabilized it can hit anyone and can cause serious injuries, on the other hand this can also critically damage the hardware components
- There is a major problem in wireless communication such as transmitting and receiving of data in packets and hence in acknowledgement of these packets by normal “Remote Control” available in market. The packets sent across a transmission channel via wireless device should be received at the receiving node with no information/data loss since the “Remote Control” has no reliability so data loss can occur and also the acknowledgement along with information such as Battery Level, Yaw, Pitch, Roll and Throttle values etc. cannot be transmitted back by wireless unit at Quadcopter to handheld controller since “Remote Control” available in market is not suitable for this type of use.
- Signal loss is highly probable when using wireless devices or units, which can cause the Quadcopter to relinquish the link or connection from handheld controller. In this case the Quadcopter will continue heading or moving in some direction and thus will be lost from reach of the person controlling it and hence can cause critical damage to property or any individual by injuring him/her.
- Power loss can cause similar effect as signal loss.
- Inertial Measurement Unit (IMU) consist of Accelerometer, Gyroscope if not calibrated on a scale will cause gibberish data as output for calculation of Yaw, Pitch and Roll angular values thus the IMU needs to be calibrated as finer it can be and moreover IMU can give much drift in gyroscopic angular values thus drift needs to be removed otherwise the Quadcopter won’t get stabilized.
- Imbalance of Propellers and motors on Quadcopter can produce unwanted vibration in Quadcopter frame and can distort the values coming from IMU i.e. from Accelerometer, Gyroscope and Magnetometer and thus cause instability in Quadcopter. Some methods for reducing these vibrations such as damping etc. have to be introduced.

3. Solution Domain

The following are the solution devised for the above mentioned problems

- In order to stabilize the Quadcopter, PID algorithm is implemented. PID stands for Proportional Integral Derivative. PID algorithm is generally used to stabilize an electrical signal or fluctuating value in field of electronics and aeronautics and thus it is used to stabilize the values of Yaw, Pitch and Roll generated as a result of control values transmitted from handheld controller via wireless units.
- We are using low power chip named "RF24" as wireless unit or device, it is a SPI device."RF24" is a simplex communication device i.e. the wireless unit can only transmit at a time or receive at a time but advantage is that either wireless unit or RF24 can become transmitter or receiver via a scheduler routine and it supports retransmission, auto acknowledgement and error checking. Thus communication is error free and supports the telemetry operation required for Quadcopter.
- Several complicated algorithms are created to handle the signal loss and power loss situations.
- We are using "MPU9250" a 9DOF sensor as IMU, it has 3-axis Accelerometer, 3-axis Gyroscope and 3-axis Magnetometer on a single chip, it is I2C device.MPU9250 is calibrated by algorithms involving embedded system programming and raw measurement and thus IMU is calibrated successfully. For removing the drift from gyroscopic angular values we have implemented the open source sensor fusion algorithm such as complimentary filter and Madgwick filter in order to stabilize the Yaw, pitch and roll angular measurement
- Several methods are done to remove the vibration from Quadcopter's structure

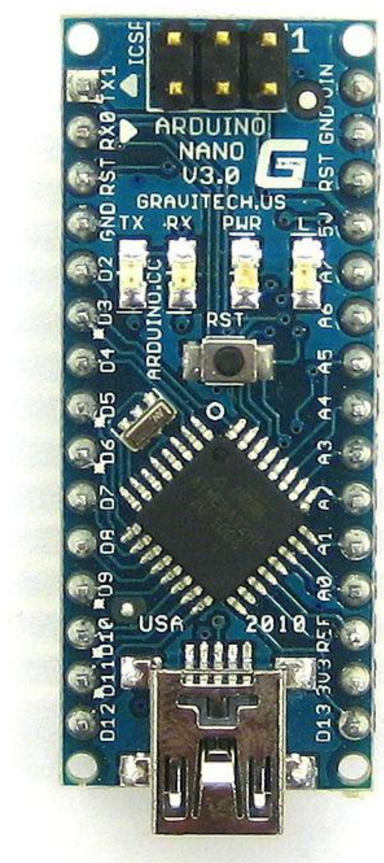
4. SYSTEM REQUIREMENTS SPECIFICATIONS

4.1 System Domain

The following are the equipment required:

S.No	Equipment	Quantity
1	Arduino Nano	2
2	Electronic Speed Controller	4
3	Brushless DC Motor	4
4	LiPo battery	1
5	Inertial measurement unit	1
6	Quad copter frame	1

- **Arduino Nano:**



Arduino is a software company, project, and user community that designs and manufactures computer hardware, open, and microcontroller-based kits for building digital devices and interactive objects that can sense and control physical devices.

The project is based on microcontroller board designs, produced by several vendors, using various microcontrollers. These systems provide sets of digital and analog I/O pins that can interface to various expansion boards (termed *shields*) and other circuits. The boards feature serial communication interfaces, including Universal Serial Bus (USB) on some models, for loading programs from personal computers. For programming the microcontrollers, the Arduino project provides an integrated development environment(IDE) based on a programming language named *Processing*, which also supports the languages C and C++.

The first Arduino was introduced in 2005, aiming to provide a low cost, easy way for novices and professionals to create devices that interact with their environment using sensors and actuators. Common examples of such devices intended for beginner hobbyists include simple robots, thermostats, and motion detectors.

Arduino boards are available commercially in preassembled form, or as do-it-yourself kits. The hardware design specifications are openly available, allowing the Arduino boards to be produced by anyone.

- **ESC or Electronic Speed Controller**

An **electronic speed control** or **ESC** is an electronic circuit with the purpose to vary an electric motor's speed, its direction and possibly also to act as a dynamic brake. ESCs are often used on electrically powered radio controlled models, with the variety most often used for brushless motors essentially providing an electronically generated three-phase electric power low voltage source of energy for the motor.

An ESC can be a stand-alone unit which plugs into the receiver's throttle control channel or incorporated into the receiver itself, as is the case in most toy-grade R/C vehicles. Some R/C manufacturers that install proprietary hobby-grade electronics in their entry-level vehicles, vessels or aircraft use onboard electronics that combine the two on a single circuit.

ESCs are normally rated according to maximum current, for example, 25 amperes or 25 A. Generally, the higher the rating, the larger and heavier the ESC tends to be which is a factor when calculating mass and balance in airplanes. Many modern ESCs support nickel metal hydride, lithium ion polymer and lithium iron phosphate batteries with a range of input and cut-off voltages. The type of battery and number of cells connected is an important consideration when choosing a Battery eliminator circuit (BEC), whether built into the controller or as a stand-alone unit. A higher number of cells connected will result in a reduced power rating and therefore a lower number of servos supported by an integrated BEC, if it uses a linear voltage regulator. A well designed BEC using a switching regulator should not have a similar limitation.

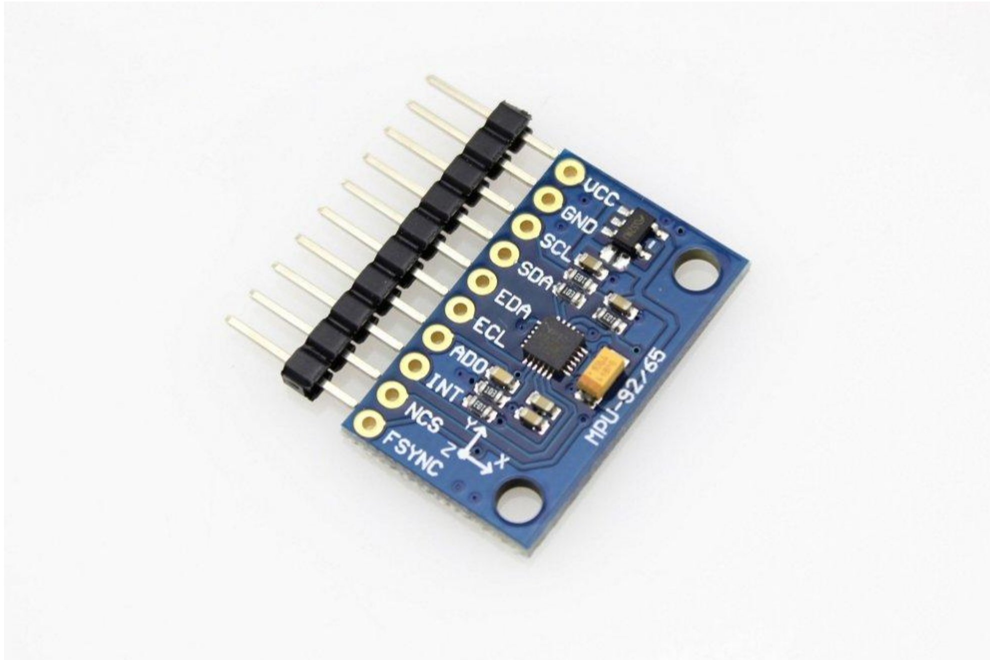
BLDC Motor:

Brushless DC electric motor (BLDC motors, BL motors) also known as electronically commutated motors (ECMs, EC motors) are synchronous that are powered by a DC electric source via an integrated inverter/switching power supply, which produces an AC electric signal to drive the motor. In this context, AC, alternating current, does not imply a sinusoidal waveform, but rather a bi-directional current with no restriction on waveform. Additional sensors and electronics control the inverter output amplitude and waveform (and therefore percent of DC bus usage/efficiency) and frequency (i.e. rotor speed).

- **IMU:**

An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surrounding the body, using a combination of accelerometers and gyroscopes, sometimes also magnetometers. IMUs are typically used to maneuver aircraft, including unmanned aerial vehicles(UAVs), among many others, and spacecraft, including satellites and landers. Recent developments allow for the production of IMU-enabled GPS devices. An IMU allows a GPS receiver to work when GPS-signals are unavailable, such as in tunnels, inside buildings, or when electronic interference is present

The IMU is the main component of inertial navigation systems used in aircraft, spacecraft, watercraft, drones, UAV and guided missiles among others.


VCC
GND
SCL
SDA
EDA
ECL
AD0
INT
NCS
FSYNC
MPU-92/65

5. IMPLEMENTATION

5.1 PID Algorithm

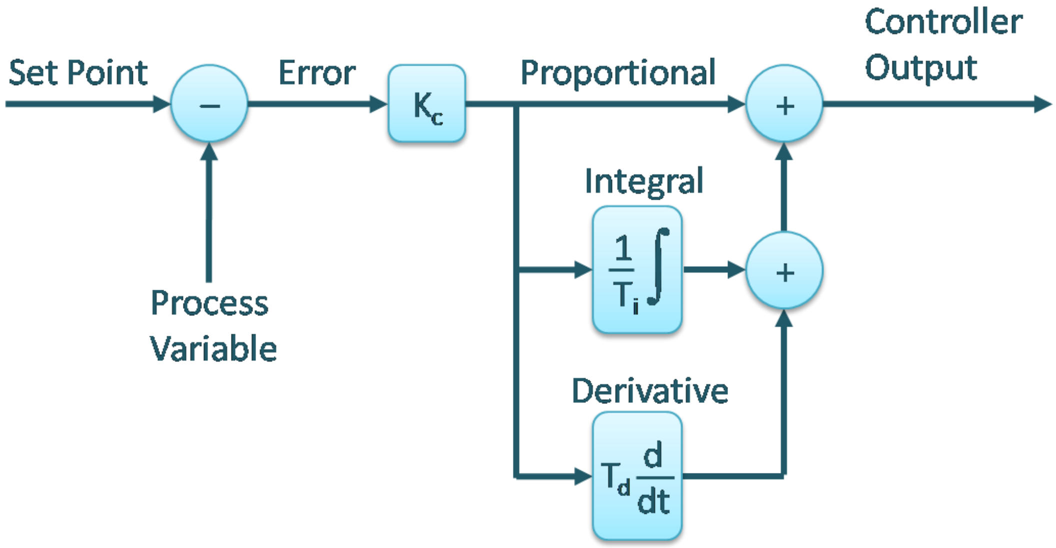


Fig: A block diagram of a PID controller in a feedback loop

A **proportional–integral–derivative controller** (**PID controller**) is a control loop feedback mechanism (controller) commonly used in industrial control systems. A PID controller continuously calculates an error value as the difference between a desired set point and a measured process variable. The controller attempts to minimize the error over time by adjustment of a control variable, such as the position of a control valve, a damper, or the power supplied to a heating element, to a new value determined by a weighted sum:

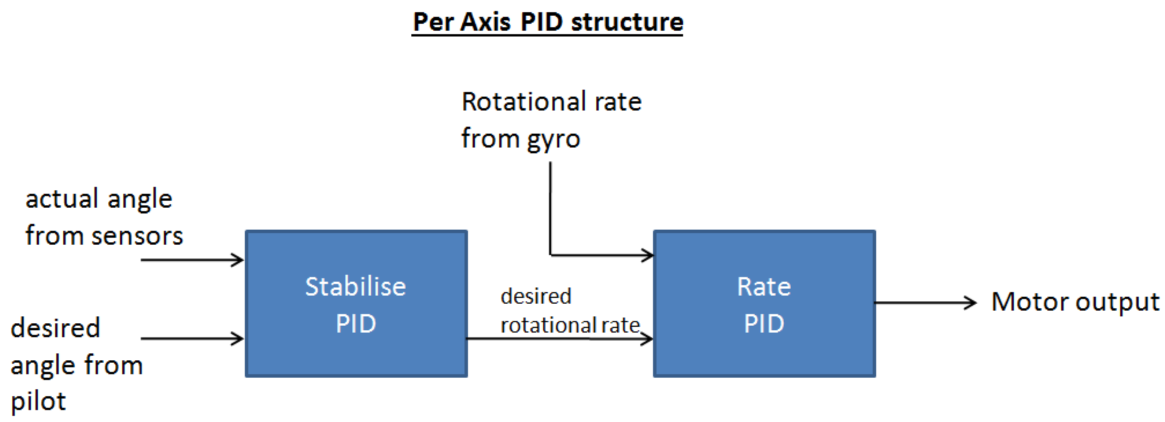
$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

where K_p, K_i, and K_d, all non-negative, denote the coefficients for the proportional, integral, and derivative terms, respectively (sometimes denoted *P*, *I*, and *D*). In this model,

- *P* accounts for present values of the error. For example, if the error is large and positive, the control output will also be large and positive.
- *I* account for past values of the error. For example, if the current output is not sufficiently strong, error will accumulate over time, and the controller will respond by applying a stronger action.
- *D* accounts for possible future values of the error, based on its current rate of change.[1]

As a PID controller relies only on the measured process variable, not on knowledge of the underlying process, it is broadly applicable. By tuning the three parameters of the model, a PID controller can deal with specific process requirements. The response of the controller can be described in terms of its responsiveness to an error, the degree to which the system overshoots a set point, and the degree of any system oscillation. The use of the PID algorithm does not guarantee optimal control of the system or even its stability.

5.1.1 Control loop structure for Quadcopter



To have any kind of control over the quadcopter or multicopter, we need to be able to measure the quadcopter sensor output (for example the pitch angle), so we can estimate the error (how far we are from the desired pitch angle, e.g. horizontal, 0 degree). We can then apply the 3 control algorithms to the error, to get the next outputs for the motors aiming to correct the error.

The sensed position or angles is the process variable (PV). The desired position or desired angles from pilot/Remote Controller is called the set point (SP). The input to the process (the electric current in the motor) is the output from the PID controller. It is called either the manipulated variable (MV) or the control variable (CV). The difference between the present position and the set point is the error (e), which quantifies whether the angle is too low or too high and by how much.

By measuring the position (PV), and subtracting it from the set point (SP), the error (e) is found, and from it the controller calculates how much electric current to supply to the motor (MV). The obvious method is **proportional** control: the motor current is set in proportion to the existing error. A more complex control may include another term: **derivative** action. This considers the rate of change of error, supplying more or less electric current depending on how fast the error is approaching zero. Finally, **integral** action adds a third term, using the accumulated position error in the past to detect whether the position of the mechanical arm is settling out too low or too high and to set the electrical current in relation not only to the error but also the time for which it has persisted. An alternative formulation of integral action is to change the electric current in small persistent steps that are proportional to the current error. Over time the steps accumulate and add up dependent on past errors; this is the discrete-time equivalent to integration.

If a controller starts from a stable state with zero error (PV = SP), then further changes by the controller will be in response to changes in other measured or unmeasured inputs to the process that affect the process, and hence the PV. Variables that affect the process other than the MV are known as disturbances. Generally, controllers are used to reject disturbances and to implement set point changes. A change in load on the arm constitutes a disturbance to the robot arm control process.

There are three parameters that a pilot can adjust to improve better quadcopter stability. These are the coefficients to the 3 algorithms we mentioned above. The coefficient basically would change the importance and influence of each algorithm to the output. Here we are going to look at what are the effects of these parameters to the stability of a quadcopter.

5.1.2 PID controller description

The PID control scheme is named after its three correcting terms, whose sum constitutes the manipulated variable (MV). The proportional, integral, and derivative terms are summed to calculate the output of the PID controller. Defining $u(t)$ as the controller output, the final form of the PID algorithm is:

$$\mathrm{u}(t) = \mathrm{MV}(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt}$$

Where,

K_p: Proportional gain, a tuning parameter

K_i: Integral gain, a tuning parameter

K_d: Derivative gain, a tuning parameter

e: Error $= SP - PV$

SP: Set Point

PV: Process Variable

t: Time or instantaneous time (the present)

τ: Variable of integration; takes on values from time 0 to the present t.

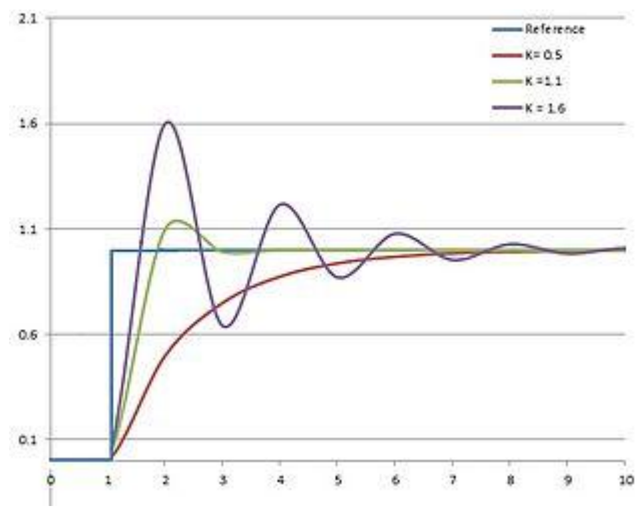
Equivalently, the transfer function in the Laplace Domain of the PID controller is

$$L(s) = K_p + K_i/s + K_d s$$

Where,

s: complex number frequency

5.1.3 Proportional term



Plot of PV vs time, for three values of K_p (K_i and K_d held constant)

The proportional term produces an output value that is proportional to the current error value. The proportional response can be adjusted by multiplying the error by a constant K_p, called the proportional gain constant.

The proportional term is given by:

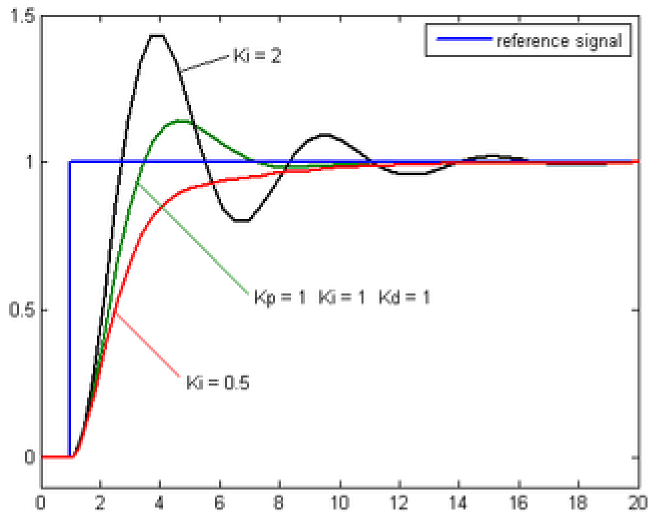
$$P_{\text{out}} = K_p\, e(t)$$

A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable. In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller. If the proportional gain is too low, the control action may be too small when responding to system disturbances. Tuning theory and industrial practice indicate that the proportional term should contribute the bulk of the output change.

5.1.4 Steady-state error

Because a non-zero error is required to drive it, a proportional controller generally operates with a so-called *steady-state error*. Steady-state error (SSE) is proportional to the process gain and inversely proportional to proportional gain. SSE may be mitigated by adding a compensating bias term to the set point or output, or corrected dynamically by adding an integral term.

5.1.5 Integral term



Plot of PV vs time, for three values of K_i (K_p and K_d held constant)

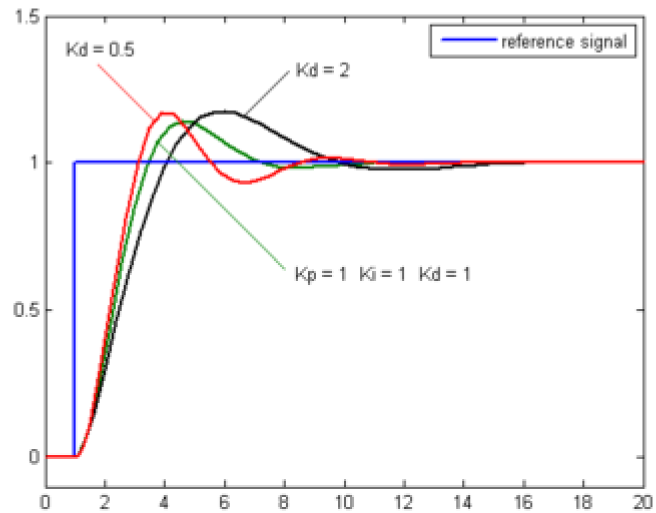
The contribution from the integral term is proportional to both the magnitude of the error and the duration of the error. The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. The accumulated error is then multiplied by the integral gain (K_i) and added to the controller output.

The integral term is given by:

$$I_{\text{out}} = K_i \int_0^t e(\tau)\, d\tau$$

The integral term accelerates the movement of the process towards set point and eliminates the residual steady-state error that occurs with a pure proportional controller. However, since the integral term responds to accumulated errors from the past, it can cause the present value to overshoot the set point value (see the section on loop tuning).

5.1.6 Derivative term



Plot of PV vs time, for three values of K_d (K_p and K_i held constant)

The derivative of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain K_d. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, K_d.

The derivative term is given by:

$$D_{\text{out}} = K_d \frac{de(t)}{dt}$$

Derivative action predicts system behavior and thus improves settling time and stability of the system.An ideal derivative is not causal, so that implementations of PID controllers include an additional low pass filtering for the derivative term, to limit the high frequency gain and noise Derivative action is seldom used in practice though - by one estimate in only 25% of deployed controllers- because of its variable impact on system stability in real-world applications

5.1.7 The effect of each parameter

The variation of each of these parameters alters the effectiveness of the stabilization. Generally, there are 3 PID loops with their own P I D coefficients, one per axis, so will have to set P, I and D values for each axis (pitch, roll and yaw).

To a quadcopter, these parameters can cause these behaviors.

- **Proportional Gain coefficient** – Your quadcopter can fly relatively stable without other parameters but this one. This coefficient determines which is more important, human control or the values measured by the gyroscopes. The higher the coefficient, the higher the quadcopter seems more sensitive and reactive to angular change. If it is too low, the quadcopter will appear sluggish and will be harder to keep steady. You might find the multicopter starts to oscillate with a high frequency when P gain is too high.

- **Integral Gain coefficient** – this coefficient can increase the precision of the angular position. For example, when the quadcopter is disturbed and its angle changes 20 degrees, in theory it remembers how much the angle has changed and will return 20 degrees. In practice if you make your quadcopter go forward and the force it to stop, the quadcopter will continue for some time to counteract the action. Without this term, the opposition does not last as long. This term is especially useful with irregular wind, and ground effect (turbulence from motors). However, when the I value gets too high your quadcopter might begin to have slow reaction and a decrease effect of the Proportional gain as consequence, it will also start to oscillate like having high P gain, but with a lower frequency.

- **Derivative Gain coefficient –** this coefficient allows the quadcopter to reach more quickly the desired attitude. Some people call it the accelerator parameter because it amplifies the user input. It also decreases control action fast when the error is decreasing fast. In practice it will increase the reaction speed and in certain cases an increase the effect of the P gain.

5.1.8 Stability

If the PID controller parameters (the gains of the proportional, integral and derivative terms) are chosen incorrectly, the controlled process input can be unstable, i.e., its output diverges, with or without oscillation, and is limited only by saturation or mechanical breakage. Instability is caused by *excess* gain, particularly in the presence of significant lag.

Generally, stabilization of response is required and the process must not oscillate for any combination of process conditions and set points, though sometimes marginal stability (bounded oscillation) is acceptable or desired.

Mathematically, the origins of instability can be seen in the Laplace domain.[15] The total loop transfer function is:

$$H(s) = \frac{K(s)G(s)}{1 + K(s)G(s)}$$

where

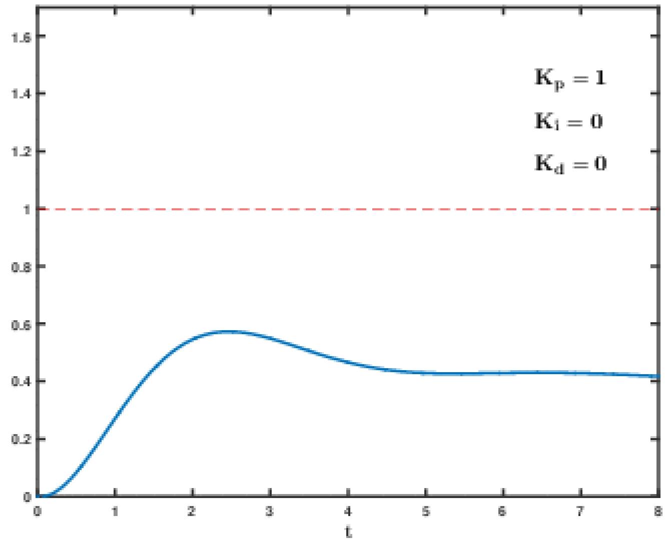
$K(s)$: PID transfer function

$G(s)$: Plant transfer function

The system is called unstable where the closed loop transfer function diverges for some s.[15] This happens for situations where $K(s)G(s) = -1$. Typically, this happens when $|K(s)G(s)| = 1$ with a 180 degree phase shift. Stability is guaranteed when $K(s)G(s) < 1$ for frequencies that suffer high phase shifts. A more general formalism of this effect is known as the Nyquist stability criterion.

5.1.9 Manual tuning

If the system must remain online, one tuning method is to first set K_i and K_d values to zero. Increase the K_p until the output of the loop oscillates, then the K_p should be set to approximately half of that value for a "quarter amplitude decay" type response. Then increase K_i until any offset is corrected in sufficient time for the process. However, too much K_i will cause instability. Finally, increase K_d, if required, until the loop is acceptably quick to reach its reference after a load disturbance. However, too much K_d will cause excessive response and overshoot. A fast PID loop tuning usually overshoots slightly to reach the set point more quickly; however, some systems cannot accept overshoot, in which case an *over-damped* closed-loop system is required, which will require a K_p setting significantly less than half that of the K_p setting that was causing oscillation.



Effects of varying PID parameters (K_p,K_i,K_d) on the step response of a system.

Effects of increasing a parameter independently					
Parameter	**Rise time**	**Overshoot**	**Settling time**	**Steady-state error**	**Stability**
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

These gains apply to the ideal, parallel form of the PID controller. When applied to the standard PID form, the integral and derivative time parameters T_i and T_d are only dependent on the oscillation period T_u.

5.1.10 Tuning of quadcopter PID Gains

It is usually tune one parameter at a time start with P gain, I and then D gain.

For **P gain**, first start low and work way up, until noticing it producing oscillation. Fine tune it until you get to a point it’s not sluggish and there is not oscillation.

For the **I gain**, again start low, and increase slowly. Roll and pitch your quad left and right, pay attention to the how long does it take to stop and stabilize. You want to get to a point where it stabilizes very quickly as you release the stick and it doesn't wander around for too long. You might also want to test it under windy condition to get a reliable I-value.

For **D gain**, it can get into a complicated interaction with P and I value. When using D gain, you need to go back and fine tune P and I to keep the plant well stabilized.

Quadcopters are symmetric so you can set the same PID Gain values for Pitch, and Roll. The value for Yaw is not as important as those of Pitch and Roll so it's probably OK to set the same values as for Pitch/Roll to start with (even it might not be the best). After your multicopter is relatively stable, you can start alter the Yaw gains. For non-symmetric multicopter like hexcopter or tricopter, you might want to fine tune the pitch and roll separately, after you have some flight experience.

5.1.11 IMU Data Fusing: Complementary, Madgwick Filter

An **inertial measurement unit**, or IMU, measures accelerations and rotation rates, and possibly earth's magnetic field, in order to determine a body's attitude. Two basic filter approaches are discussed, the **complementary filter** and the **Madgwick filter**. The article starts with some preliminaries, which I find relevant. It then considers the case of a single axis (called one dimensional or 1D). First the simplest method is discussed, where gyro bias is not estimated (called 1^{st} order). Then gyro bias estimation is included (called 2^{nd} order). Finally, the complete situation of three axes (called 3D) is considered, and some approximations and improvements are evaluated.

Notation: The discrete time step is denoted as Δt, and $n$or $k$is used as time-step index. The estimate of a quantity is indicated by a hat, e.g. $\hat{x}$, which will however often be dropped for simplicity, whenever confusion seems impossible. Bold symbols represent vectors or matrices in $\mathbb{R}^3$(vectors and matrices in e.g. state space won't be bold), and quaternions.

5.1.12 Preliminaries

5.1.12.1 Kinematics and IMU Algorithms

The task of attitude estimation corresponds to evaluating (computationally) the kinematic equation for the rotation of a body:

$$\dot{\mathbf{R}} = \mathbf{R}\boldsymbol{\Omega}_\times \quad \text{with} \quad \boldsymbol{\Omega}_\times = \boldsymbol{\omega}\mathbf{J} = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}, \qquad \text{Eq. (1.1)}$$

where $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)^T$ is the measured rotation rate vector. The rotation $\mathbf{R}$ represents the orientation of the body-fixed reference frame as observed in the earth reference frame. For any vector $\mathbf{v}$, the coordinates $\mathbf{v}_{earth}$ with respect to the earth frame become in the body-fixed frame $\mathbf{v}_{body} = \mathbf{R}^T\mathbf{v}_{earth}$, which evolve as $\dot{\mathbf{v}}_{body} = \boldsymbol{\Omega}_\times^T\mathbf{v}_{body} = -(\boldsymbol{\omega} \times \mathbf{v}_{body})$ (the minus sign comes in here since $\boldsymbol{\omega}$ represents the rotation of the body-fixed coordinate system).

As simple as it may appear, equation Eq. (1.1) presents us with some fundamental issues:

(1) Equation (1.1) is non-linear. This can complicate filter design enormously.

(2) Equation (1.1) is susceptible to numerical errors. Well, numerical errors are present in any calculation performed on a microprocessor, but in most cases they are well-behaved in the sense that they do not accumulate. However, for Eq. (1) the errors continuously grow if no counter-measures are taken, and $\mathbf{R}$ eventually ceases to represent a rotation. Importantly, this is related to the global non-commutativity of rotations (in 3 dimensions) and hence is fundamental. It is here where cool buzz words such as direction cosine matrix (DCM) or quaternions enter the game.

(3) The rotation $\mathbf{R}$ can be represented in several ways [RO], and each representation has its own set of advantages and disadvantages. Most well-known are the representations by a rotation matrix or DCM, Euler angles and related angles (Cardan, Tait-Brian), axis and angle, and

quaternions, but some more exist. Obviously, the algorithm will depend a lot on which representation is chosen.

You may note, no words were yet spent on measurement noise and data fusing; I haven't added it to the list since it's not really rooted in Eq. (1.1), although it's of course an important point – and in fact the topic of this article.

Anyhow, since the challenges are alike, all algorithms presented by the above authors exhibit a similar structure:

5.1.12.2 Discretization and Implementation Issues

Well, it's not big news that for a given system, described by e.g. a continuous-time transfer function $G(s)$, there are many different possible implementations in computer code, and that they do not all show identical performance even though they are derived from one and the same function $G(s)$.

One reason for arriving at different implementations is that there is no universal rule for converting a continuous- time transfer function $G(s)$ to a discrete-time transfer function $H(z)$, since it's approximate. The conversion from $G(s)$ to $H(z)$ is thus not unique, and typical choices are:

backward difference: $s = \frac{1}{\Delta t}(1 - z^{-1})$, Eq. (1.2)

bilinear transformation, expansion of $ln(z)$: $s = \frac{2}{\Delta t}\frac{1 - z^{-1}}{1 + z^{-1}}$, Eq. (1.3)

impulse invariance transformation: $\frac{G(z)}{1 - z^{-1}} = Z\left(\frac{G(s)}{s}\right)$, Eq. (1.4)

Another reason is that a given discrete-time transfer function $H(z)$ can be implemented in different ways, and the different implementations possibly show different behavior in various aspects, such as stability and high-frequency noise. The topic is not simple, and is beyond my competences, but the basic principle is clear.

Let's consider as an example, since it's so familiar, the PID controller $G(s) = K_p + \frac{1}{s}K_i + sK_d$. Using backward difference one finds $H(z) = K_p + \frac{K_i \Delta t}{1-z^{-1}} + \frac{K_d}{\Delta t}(1 - z^{-1})$, which can be implemented by first evaluating $I_n = \frac{K_i \Delta t}{1-z^{-1}} x_n$and then $y_n = K_p x_n + I_n + \frac{K_d}{\Delta t}(1 - z^{-1})x_n$, or by directly solving $y_n = H(z)x_n$. In the first case one obtains the **positional PID algorithm**

$$I_n = I_{n-1} + K_i \Delta t x_n \quad , \qquad \text{Eq.(1.5a)}$$

$$y_n = K_p x_n + I_n + \frac{K_d}{\Delta t}(x_n - x_{n-1}) \quad , \qquad \text{Eq. (1.5b)}$$

while the second case leads to the **velocity PID algorithm**

$$y_n = y_{n-1} + K_p(x_n - x_{n-1}) + K_i \Delta t x_n + \frac{K_d}{\Delta t}(x_n + 2x_{n-1} - x_{n-2}) \quad . \qquad \text{Eq. (1.6)}$$

Both derive from the same function $H(z)$, or $G(s)$, but differ e.g. with regards to wind up, overflow of internal variables, number of storage elements and so on.

Corollary #1: In the positional PID algorithm, Eq. (1.5), the order of the two equations can be reversed,

$$y_n = K_p x_n + I_{n-1} + \frac{K_d}{\Delta t}(x_n - x_{n-1}) \quad , \qquad \text{Eq.(1.7a)}$$

$$I_n = I_{n-1} + K_i \Delta t x_n \quad , \qquad \text{Eq.(1.7b)}$$

with an irrelevant change of the parameter $K_p \rightarrow K_p + K_i \Delta t$. That is, the order of their execution or implementation in code is irrelevant.

5.1.12.3 1D IMU Data Fusing – 1st Order (without Drift Estimation)

In this chapter we will consider the simplest case of IMU data fusing, namely that of fusing the angles for a single axis as determined from the time-integrated rotation rate and accelerometer data, without explicitly estimating the gyro's drift. It has relevance for applications, but here it is of interest mainly as a „warm up", and because it provides some insight which will help us to better understand the more advanced cases below.

In the following, θdenotes the estimated angle), $a$the angle calculated from the accelerometer measurements, and $\omega$the rotation rate measured by the gyro.

- **Complementary Filter**

The complementary filter fuses the accelerometer and integrated gyro data by passing the former through a 1st-order low pass and the latter through a 1st-order high pass filter and adding the outputs. The transfer function reads

, $$\theta = \frac{1}{1+Ts}a + \frac{Ts}{1+Ts}\frac{1}{s}\omega = \frac{a+T\omega}{1+Ts}$$ Eq. (2.1)

where Tdetermines the filter cut-off frequencies. Using backward difference yields.

$$1 + Ts = (1 + \tfrac{T}{\Delta t}) - \tfrac{T}{\Delta t}z^{-1}$$

Insertion into Eq. (2.1) and rearrangement leads to our final result

$$\theta_k = \alpha(\theta_{k-1} + \omega_k \Delta t) + (1-\alpha)a_k$$ Eq. (2.2)

where $\alpha = \frac{T}{\Delta t}/(1 + \frac{T}{\Delta t})$. This relation can be implemented in code in several ways; three of them were discussed in [SC] (though only #3 makes sense). For better comparison with the other cases below, the result is reformulated as

. $$\theta_k = \alpha\theta_{k-1} + (1-\alpha)a_k + \alpha\omega_k\Delta t$$ Eq. (2.3)

- **Madgwick Filter**

Here data fusing is done with a P controller and an integrating process, where the „accelerometer“ angle $a$becomes the set point and the rotation rate ωa disturbance (of type d_1). The transfer function is thus

, $$\theta = K_p\frac{1}{s}(a-\theta) + \frac{1}{s}\omega$$ Eq. (2.7) where $a-\theta$is the error input to the P controller. Rearranging this equation for θyields exactly the transfer function of the complementary filter, Eq. (2.1), which from standard arguments of control theory is however not surprising.

The controller Eq. (2.7) is usually implemented by first rearranging it to $\theta = \frac{1}{s}\left[K_p(a-\theta)+\omega\right]$, then separating it into $e = a-\theta$and $\theta = \frac{1}{s}(K_p e+\omega)$, and finally discretizing it as

$$\begin{aligned} e_k &= a_k - \theta_{k-1} \\ \theta_k &= \theta_{k-1} + (K_p e_k + \omega_k)\Delta t \end{aligned} \qquad \text{Eq. (2.8)}$$

That is, for the Madgwick filter one arrives at the update law

,

$$\theta_k = \alpha\theta_{k-1} + (1-\alpha)a_k + \omega_k \Delta t \qquad \text{Eq. (2.9)}$$

with $\alpha = 1 - K_p\Delta t$.

It is worthwhile to elaborate a bit further on the controller aspect. For a (simple) controller on finds in general that

,

$$y = \frac{G_c G_p}{1+G_c G_p} r + \frac{G_p}{1+G_c G_p} d_1 + \frac{1}{1+G_c G_p} d_2 \qquad \text{Eq. (2.10)}$$

where $G_c(s)$and $G_p(s)$are the controller and process transfer functions, respectively (the other symbols should be obvious). The conversion of the complementary filter transfer function, Eq. (2.1), into the controller form, Eq. (2.7), is accomplished by multiplying the nominator and denominator in Eq. (2.1) by $1/s$:

,

$$\theta = \frac{1}{1+Ts} a + \frac{Ts}{1+Ts}\frac{1}{s}\omega = \frac{K_p \frac{1}{s}}{1+K_p\frac{1}{s}} a + \frac{\frac{1}{s}}{1+K_p\frac{1}{s}}\omega \qquad \text{Eq. (2.11)}$$

and identifying $G_c(s) = K_p = 1/T$and $G_p(s) = 1/s$. Equations (2.1) and (2.7) are identical, but the former is expressed in terms of polynomials in $s$while the latter is expressed in terms of polynomials in $1/s$.

- **Comparison and Conclusions**

A couple of observations can be made from the above findings.

(1) The complementary and Madgwick filters are described by identical transfer functions.

(2) From (1) it follows that all three filters are identical at the level of the transfer function.

(3) Despite the algorithm of the Madgwick filter is **not** identical to that of the complementary and/ see Eq. (2.9) and Eqs. (2.2), (2.6).

It is worthwhile to discuss the last point further. For convenience the two update laws are reproduced:

,
$$\theta_k = \alpha\theta_{k-1} + (1-\alpha)a_k + \alpha\omega_k\Delta t \qquad \text{Eqs. (2.2),(2.6)}$$
.
$$\theta_k = \alpha\theta_{k-1} + (1-\alpha)a_k + \omega_k\Delta t \qquad \text{Eq. (2.9)}$$

One can look at the difference in two ways. Firstly, Eqs. (2.2,6) may be read to say that the angle is first advanced by integrating the rotation rate to give an updated angle and then filtered with $a_k$to give an improved angle. This may be expressed by the bracketing $\theta_k = \alpha[\theta_{k-1} + \omega_k\Delta t] + (1-\alpha)a_k$. In contrast, Eq. (2.9) may be read to say that the angle is first filtered with $a_k$and then advanced by integrating the rotation rate, corresponding to the bracketing $\theta_k = [\alpha\theta_{k-1} + (1-\alpha)a_k] + \omega_k\Delta t$. Secondly, the equations can be rearranged into the algorithms.

complementary filter (1D, 1st order) Madgwick filter (1D, 1st order)

$$\theta_k^- = \theta_{k-1} + \omega_k\Delta t \qquad \text{Eq. (2.12)}$$
$$e_k^- = a_k - \theta_k^-$$
$$\theta_k = \theta_k^- + K_0 e_k^-$$

$$\theta_k^- = \theta_{k-1} + \omega_k\Delta t \qquad \text{Eq. (2.13)}$$
$$e_k = a_k - \theta_{k-1}$$
$$\theta_k = \theta_k^- + K_p\Delta t e_k$$

They are essentially identical, except of the important difference that on the left the feedback error uses the updated angle θ_k^- while on the right it uses the previous angle estimate θ_{k-1}! The two algorithms cannot be directly converted into each other, even though they derive from the same transfer function, which should be considered a characteristic feature expressing the different underlying „philosophies “.

5.1.12.4 1D IMU Data Fusing – 2nd Order (with Drift Estimation)

In this chapter the single-axis filters will be improved by explicitly taking into account the bias/drift of the gyro sensors. To the best of my knowledge, a complementary filter accomplishing this task has not been described before, and hence the order of the discussion of the filters is changed as compared to Chapter 2.

- **Madgwick Filter**

The gyro drift estimation is facilitated by using a PI controller [RM08], and the transfer function is accordingly

$$\theta = \left(K_p + K_i \frac{1}{s}\right) \frac{1}{s}(a - \theta) + \frac{1}{s}\omega$$

, Eq. (3.5)

Following again the standard implementation (positional PID algorithm) one separates Eq. (3.5) into $e = a - \theta$, $I = K_i \frac{1}{s} e$, and $\theta = \frac{1}{s}(K_p e + I + \omega)$, and discretizes it as

$$\begin{aligned} e_k &= a_k - \theta_{k-1} \\ I_k &= I_{k-1} + K_i \Delta t e_k \\ \theta_k &= \theta_{k-1} + (K_p e_k + I_k + \omega_k)\Delta t \end{aligned}$$

Eq. (3.6)

This results in the update laws

$$\begin{aligned} I_k &= I_{k-1} + K_i \Delta t (a_k - \theta_{k-1}) \\ \theta_k &= \alpha \theta_{k-1} + (1 - \alpha) a_k + (\omega_k + I_k)\Delta t \end{aligned}$$

Eq. (3.7)

with $\alpha = 1 - K_p \Delta t$.

Here Corollary #1 is recalled, which tells that the sequence of the two equations in Eq. (3.7) can be reversed (with an insignificant parameter change).

- **Complementary Filter**

A complementary filter is easily derived by solving the transfer function of the Madgwick filter for the angle θ, which yields

$$\theta = \frac{1+\frac{K_p}{K_i}s}{1+\frac{K_p}{K_i}s+\frac{1}{K_i}s^2}a + \frac{\frac{1}{K_i}s^2}{1+\frac{K_p}{K_i}s+\frac{1}{K_i}s^2}\frac{1}{s}\omega$$

. Eq. (3.10)

Obviously, and not unexpectedly, this complementary filter is built from 2nd order filters. Note that the filter acting on the acceleration data actually consists of a low-pass plus band-pass filter.

This result has interesting consequences. Being 2nd order filters, the frequency response of the acceleration and rotation rate filters are characterized by the resonance frequency and damping factor

$$\omega_0 = \sqrt{K_i} \qquad \xi = \frac{K_p}{2\sqrt{K_i}}$$

, . Eq. (3.11)

The damping factor determines the overshoot at the resonance frequency. For high-pass (and low-pass) filters the frequency response is flat (and the step response non-oscillatory) for $\xi \geq 1$. This suggests the criterion

$$K_i \leq \frac{1}{4}K_p^2$$

Eq. (3.12)

in order to avoid overshoot in the gyro channel. In order to minimize also overshoot in the accelerometer channel, the damping should be somewhat larger than that; $\xi \approx 2$ might be a good compromise between smooth frequency response and fast bias estimation. Accordingly, as a rule of thumb $K_i \approx 0.05 \ldots 0.1\, K_p^2$.

Note that – unless K_i is set to very small values – the crossover frequency is determined now by K_i (or the inverse square root of it), and not by K_p as in the 1st order case! It could in fact be appropriate to use the parameters ω_0 or $T = 2\pi/\sqrt{K_i}$ and ξ instead of K_p and K_i; the tuning of the filter might be more intuitive to the user.

The complementary filter may be implemented as in Eq. (3.6), or with any of the algorithms used with advantage for digital filters. The direct form II would be a typical choice

- **Summary on 1D Filters**

As lengthy as it was, the above detailed discussion in chapters 2 and 3 of different approaches to the data fusing for a single axis result in a very short summary:

The three considered different approaches are in fact not that different, even in the 2nd order case.

As a bonus a criterion for the choice of the bias estimator gain K_i or K_1, respectively, has been obtained, as well as a potentially easier and/or more flexible direct implementation as a complementary filter.

There is a difference between the Kalman and Mahony&Madgwick filters in how the error is calculated. This may be interpreted as conceptually different „philosophies“, but besides that it’s not clear to me if this has also practical consequences, such as different stability properties or high-frequency noise. (Does anyone know?)

5.1.12.5 Further 3D Filters

- **Madgwick’s IMU Filter**

Madgwick has presented an interesting approach, which is based on formulating task T3 as a minimization problem and solving it with a gradient technique [SM]. I will argue here that this approach is – IMHO – not appropriate for IMUs which are using only gyro and accelerometer data (6DOF IMU).

Madgwick uses a quaternion approach to represent the attitude, which immediately poses the problem of how to convert the measured acceleration vector $\mathbf{a}$ into a quaternion. Madgwick has described the problem in clear detail: A body’s attitude (quaternion) cannot be unambiguously represented by a direction (vector) since any rotation of the body around that direction gives the same vector but a different quaternion. The solution manifold is a „line “and not a „point “. Or plainly: The body’s yaw angle is totally undetermined.

To tackle this problem he suggested to determine that rotation, which brings the gravity vector $\mathbf{g}^{earth} = -\mathbf{e}_z$in the earth frame in coincidence with the measured acceleration $\mathbf{g}^{body} = \mathbf{a}$in the body frame, that is to find the rotation $\mathbf{R}_a$for which $\mathbf{a} = -\mathbf{R}_a^T \mathbf{e}_z$, or the quaternion $\mathbf{q}_a$for which $\mathbf{a} = -\mathbf{q}_a^{-1} \mathbf{e}_z \mathbf{q}_a$, respectively. Converting the measured vector to a quaternion is very desirable since then data fusing could be done directly on quaternions, which has favorable mathematical properties [RO2]. In order to determine this rotation computationally, Madgwick suggested to formulate it as minimization problem and to solve it iteratively by the method of steepest decent.

This approach has two problems. The exact solution is not unique but there are infinitely many, and not the exact solution is calculated. One can hence expect that the yaw angle in the computed attitude $\mathbf{q}_a$is not only arbitrary, but determined by the noise introduced by the incomplete steepest decent. The yaw angle fluctuates.

At this point one could analyze the algorithm by asking: Let’s assume that our gyro and accelerometer data is perfect and exact, what is then the algorithm doing? Clearly, one would expect the algorithm to produce the exact attitude, and the data fusing filter not to introduce any corrections. For the filters described in the above this is obviously fulfilled. In Madgwick’s filter, the correction step $\delta\mathbf{s}$is however not zero (since $\mathbf{a} = -\mathbf{q}^{-1} \mathbf{e}_z \mathbf{q}$). That is, the filter in fact pushes the estimated attitude away from the correct attitude. In conclusion this is a signature of the noise mentioned before.

6 TESTING METHODOLOGY

6.1 Range Test

Range test was conducted to check the signal range between the remote controller and Quadcopter. In the test performed, various obstacles are introduced between wireless units on Quadcopter and Remote Controller, in which the communication seems to be unaffected with respect to clean line of sight communication.

6.2 IMU Test

IMU or Inertial measurement unit is tested while it is being calibrated. In testing phase, a number of tests were carried out to calibrate the IMU (3 – axis Accelerometer, 3- axis Gyroscope) by placing it on perpendicular and parallel surfaces with respect to ground. And also various sensitivity factors were varied (i.e. sensitivity factors of Accelerometer and Gyroscope) were varied to produce optimum readings from gyroscope and accelerometer.

6.3 PID Tuning

The various gain coefficients like proportional gain, derivative gain and Integral gain coefficients were adjusted to stabilize the vibrations and angular values such as Yaw, Pitch and Roll.

6.4 Propeller Balancing

Various methods were applied to remove imbalance in propellers which includes setting of Digital Low Pass Filter in IMU at 5 Hz.

7 CONCLUSION

7.1 Summary

Our Quadcopter was able to fly with stable angular values of Yaw, Pitch and Roll. We have used Arduino Nano board which has ATMEGA 328P microcontroller which does all the implemented operations despite operating on very low power.

We have used NRF24L01+ wireless chip which works on low power with consuming only 3.3 Volts which produce a feasible result that is the range test is done successfully.

We have used a single chip Inertial Measurement Unit (IMU) to measure the angular values which results in stable operations of Quadcopter.

7.2 Limitations

Because the microcontrollers internal flash memory is 32K in which bootloader acquires 1K and a code acquires approximately 29K and rest of the memory is used for dynamic memory allocation for variables the algorithms such as power loss and impulse detection were not yet implemented.

7.3 Recommended for future work

In future version of our Quadcopter the power loss and impulse detection algorithm will be implemented and also we will attach computer vision based processing to the Quadcopter which will use some Artificial Intelligence algorithms to do certain tasks.

8 CODE

```
#include "quadcopter_config.h"
#include <PID_v1.h>
#include <Servo.h>
#include <Wire.h>
#include"RF24.h"


float angleX,angleY,angleZ = 0.0;

int throttle=THROTTLE_RMIN;



double pid_roll_in,   pid_roll_out,   pid_roll_setpoint = 0;
double pid_pitch_in,  pid_pitch_out,  pid_pitch_setpoint = 0;
double pid_yaw_in,    pid_yaw_out,    pid_yaw_setpoint = 0;


int m0, m1, m2, m3;



void setup()
{

    Serial.begin(115200);
    if(Serial.available())
    Serial.println("Debug Output ON");


  motors_initialize();

 pinMode(PIN_LED, OUTPUT);
 digitalWrite(PIN_LED, LOW);



  rx_Initialize();
  pid_initialize();
  motors_arm();
  imu_setup();

}

void loop()
{
  receiver_update();
  imu_loop();
  control_update();


```

```
}
void control_update(){
  throttle=map(packet_rx.throttle,THROTTLE_RMIN,THROTTLE_RMAX,MOTOR_ZERO_LEVEL,MOTOR_MAX_LEVEL);


  setpoint_update();
  pid_update();
  pid_compute();

  m0 = throttle + pid_pitch_out - pid_roll_out + pid_yaw_out;
  m1 = throttle + pid_pitch_out + pid_roll_out - pid_yaw_out;
  m2 = throttle - pid_pitch_out + pid_roll_out + pid_yaw_out;
  m3 = throttle - pid_pitch_out - pid_roll_out - pid_yaw_out;

  if(throttle < THROTTLE_SAFE_SHUTOFF)
    {
      m0 = m1 = m2 = m3 = MOTOR_ZERO_LEVEL;
    }

  update_motors(m0, m1, m2, m3);
  m0=0;
  m1=0;
  m2=0;
  m3=0;
}


void setpoint_update() {

  if(packet_rx.roll > 508 - 20 && packet_rx.roll < 508 + 20)
    pid_roll_setpoint = 0;
  else
    pid_roll_setpoint = map(packet_rx.roll,ROLL_RMIN,ROLL_RMAX,ROLL_WMIN,ROLL_WMAX);



  if(packet_rx.pitch > 508 - 20 && packet_rx.pitch < 508 + 20)
    pid_pitch_setpoint = 0;
  else
    pid_pitch_setpoint = map(packet_rx.pitch,PITCH_RMIN,PITCH_RMAX,PITCH_WMIN,PITCH_WMAX);


  if(packet_rx.yaw > 522 - 20 && packet_rx.yaw < 522 + 20)
    pid_yaw_setpoint = 0;
  else
    pid_yaw_setpoint = map(packet_rx.yaw,YAW_RMIN,YAW_RMAX,YAW_WMIN,YAW_WMAX);

    Serial.println();
```

```
    Serial.println(pid_roll_setpoint);
    Serial.println(pid_pitch_setpoint);
    Serial.println(pid_yaw_setpoint);
}


Servo motor0;
Servo motor1;
Servo motor2;
Servo motor3;

void motors_initialize(){
  motor0.attach(PIN_MOTOR0);
  motor1.attach(PIN_MOTOR1);
  motor2.attach(PIN_MOTOR2);
  motor3.attach(PIN_MOTOR3);
  motor0.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor1.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor2.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor3.writeMicroseconds(MOTOR_ZERO_LEVEL);
}

void motors_arm(){
  motor0.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor1.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor2.writeMicroseconds(MOTOR_ZERO_LEVEL);
  motor3.writeMicroseconds(MOTOR_ZERO_LEVEL);
  delay(3000);
}

void update_motors(int m0, int m1, int m2, int m3)
{
  /*motor0.writeMicroseconds(m0);
  motor1.writeMicroseconds(m1);
  motor2.writeMicroseconds(m2);
  motor3.writeMicroseconds(m3);
  */
  Serial.print("\t Front left :");
  Serial.print(m0);
  Serial.print("\t Front Right :");
  Serial.print(m1);
  Serial.print("\t Rear left :");
  Serial.print(m3);
  Serial.print("\t Rear Right :");
  Serial.print(m2);
  Serial.println();
}
```

```
#include<SPI.h>
RF24 rx(7,8);
byte addresses[][6]={"tx","rx"};
struct packet
{
  int throttle;
  int yaw;
  int roll;
  int pitch;
}packet_rx;
void rx_Initialize(){
//  Serial.begin(115200); used for debugging
  rx.begin();
  rx.setChannel(115);
  rx.setDataRate(RF24_250KBPS);
  rx.setPALevel(RF24_PA_LOW);
  rx.openWritingPipe(addresses[1]);
  rx.openReadingPipe(1,addresses[0]);
  rx.startListening();
  rx.setAutoAck(0);
  rx.setCRCLength(RF24_CRC_16);
}
void receiver_update(){
  if(rx.available())
  {
    while(rx.available())
      {rx.read(&packet_rx,sizeof(packet_rx));
  }
  }
  Serial.print(packet_rx.throttle);
  Serial.print("\t");
  Serial.print(packet_rx.yaw);
  Serial.print("\t");
  Serial.print(packet_rx.pitch);
  Serial.print("\t");
  Serial.print(packet_rx.roll);
  Serial.print("\t");
}
```

```

:

// MPU9250 VCC - Arduino VCC
// MPU9250 GND - Arduino GND
// MPU9250 I2C SCL - Arduino I2C SCL
// MPU9250 I2C SCL - Arduino I2C SDA




// Arduino code ( GYRO + Accelerometer)

#include <SPI.h>
#include <Wire.h>

#define AK8963_ADDRESS   0x0C
#define WHO_AM_I_AK8963  0x00 // should return 0x48
#define INFO             0x01
#define AK8963_ST1       0x02  // data ready status bit 0
#define AK8963_XOUT_L  0x03  // data
#define AK8963_XOUT_H  0x04
#define AK8963_YOUT_L  0x05
#define AK8963_YOUT_H  0x06
#define AK8963_ZOUT_L  0x07
#define AK8963_ZOUT_H  0x08
#define AK8963_ST2       0x09  // Data overflow bit 3 and data read error status bit 2
#define AK8963_CNTL      0x0A  // Power down (0000), single-measurement (0001), self-test (1000) and Fuse ROM (1111) modes on bits 3:0
#define AK8963_ASTC      0x0C  // Self test control
#define AK8963_I2CDIS    0x0F  // I2C disable
#define AK8963_ASAX      0x10  // Fuse ROM x-axis sensitivity adjustment value
#define AK8963_ASAY      0x11  // Fuse ROM y-axis sensitivity adjustment value
#define AK8963_ASAZ      0x12  // Fuse ROM z-axis sensitivity adjustment value
#define XG_OFFSET_H 0x13 // User-defined trim values for gyroscope
#define XG_OFFSET_L 0x14
#define YG_OFFSET_H 0x15
#define YG_OFFSET_L 0x16
#define ZG_OFFSET_H 0x17
#define ZG_OFFSET_L 0x18
#define SMPLRT_DIV 0x19
#define CONFIG 0x1A
#define GYRO_CONFIG 0x1B
#define ACCEL_CONFIG 0x1C
#define ACCEL_CONFIG2 0x1D
#define LP_ACCEL_ODR 0x1E
#define WOM_THR 0x1F
#define MOT_DUR 0x20 // Duration counter threshold for motion interrupt generation, 1 kHz rate, LSB = 1 ms
```

```
#define ZMOT_THR 0x21 // Zero-motion detection threshold bits [7:0]
#define ZRMOT_DUR 0x22 // Duration counter threshold for zero motion interrupt generation, 16 Hz rate, LSB = 64 ms
#define FIFO_EN 0x23
#define I2C_MST_CTRL 0x24
#define I2C_SLV0_ADDR 0x25
#define I2C_SLV0_REG 0x26
#define I2C_SLV0_CTRL 0x27
#define I2C_SLV1_ADDR 0x28
#define I2C_SLV1_REG 0x29
#define I2C_SLV1_CTRL 0x2A
#define I2C_SLV2_ADDR 0x2B
#define I2C_SLV2_REG 0x2C
#define I2C_SLV2_CTRL 0x2D
#define I2C_SLV3_ADDR 0x2E
#define I2C_SLV3_REG 0x2F
#define I2C_SLV3_CTRL 0x30
#define I2C_SLV4_ADDR 0x31
#define I2C_SLV4_REG 0x32
#define I2C_SLV4_DO 0x33
#define I2C_SLV4_CTRL 0x34
#define I2C_SLV4_DI 0x35
#define I2C_MST_STATUS 0x36
#define INT_PIN_CFG 0x37
#define INT_ENABLE 0x38
#define DMP_INT_STATUS 0x39 // Check DMP interrupt
#define INT_STATUS 0x3A
#define ACCEL_XOUT_H 0x3B
#define ACCEL_XOUT_L 0x3C
#define ACCEL_YOUT_H 0x3D
#define ACCEL_YOUT_L 0x3E
#define ACCEL_ZOUT_H 0x3F
#define ACCEL_ZOUT_L 0x40
#define TEMP_OUT_H 0x41
#define TEMP_OUT_L 0x42
#define GYRO_XOUT_H 0x43
#define GYRO_XOUT_L 0x44
#define GYRO_YOUT_H 0x45
#define GYRO_YOUT_L 0x46
#define GYRO_ZOUT_H 0x47
#define GYRO_ZOUT_L 0x48
#define EXT_SENS_DATA_00 0x49
#define EXT_SENS_DATA_01 0x4A
#define EXT_SENS_DATA_02 0x4B
#define EXT_SENS_DATA_03 0x4C
#define EXT_SENS_DATA_04 0x4D
#define EXT_SENS_DATA_05 0x4E
#define EXT_SENS_DATA_06 0x4F
#define EXT_SENS_DATA_07 0x50
#define EXT_SENS_DATA_08 0x51
#define EXT_SENS_DATA_09 0x52
#define EXT_SENS_DATA_10 0x53
#define EXT_SENS_DATA_11 0x54
```

```
#define EXT_SENS_DATA_12 0x55
#define EXT_SENS_DATA_13 0x56
#define EXT_SENS_DATA_14 0x57
#define EXT_SENS_DATA_15 0x58
#define EXT_SENS_DATA_16 0x59
#define EXT_SENS_DATA_17 0x5A
#define EXT_SENS_DATA_18 0x5B
#define EXT_SENS_DATA_19 0x5C
#define EXT_SENS_DATA_20 0x5D
#define EXT_SENS_DATA_21 0x5E
#define EXT_SENS_DATA_22 0x5F
#define EXT_SENS_DATA_23 0x60
#define MOT_DETECT_STATUS 0x61
#define I2C_SLV0_DO 0x63
#define I2C_SLV1_DO 0x64
#define I2C_SLV2_DO 0x65
#define I2C_SLV3_DO 0x66
#define I2C_MST_DELAY_CTRL 0x67
#define SIGNAL_PATH_RESET 0x68
#define MOT_DETECT_CTRL 0x69
#define USER_CTRL 0x6A // Bit 7 enable DMP, bit 3 reset DMP
#define PWR_MGMT_1 0x6B // Device defaults to the SLEEP mode
#define PWR_MGMT_2 0x6C
#define DMP_BANK 0x6D // Activates a specific bank in the DMP
#define DMP_RW_PNT 0x6E // Set read/write pointer to a specific start address in specified DMP bank
#define DMP_REG 0x6F // Register in DMP from which to read or to which to write
#define DMP_REG_1 0x70
#define DMP_REG_2 0x71
#define FIFO_COUNTH 0x72
#define FIFO_COUNTL 0x73
#define FIFO_R_W 0x74
#define WHO_AM_I_MPU9250 0x75 // Should return 0x71
#define XA_OFFSET_H 0x77
#define XA_OFFSET_L 0x78
#define YA_OFFSET_H 0x7A
#define YA_OFFSET_L 0x7B
#define ZA_OFFSET_H 0x7D
#define ZA_OFFSET_L 0x7E
#define    MPU9250_ADDRESS            0x68



#define    GYRO_FULL_SCALE_250_DPS    0x00
#define    GYRO_FULL_SCALE_500_DPS    0x08
#define    GYRO_FULL_SCALE_1000_DPS   0x10
#define    GYRO_FULL_SCALE_2000_DPS   0x18

#define    ACC_FULL_SCALE_2_G        0x00
#define    ACC_FULL_SCALE_4_G        0x08
#define    ACC_FULL_SCALE_8_G        0x10
#define    ACC_FULL_SCALE_16_G       0x18
```

```
 enum Ascale {
AFS_2G = 0,
AFS_4G,
AFS_8G,
AFS_16G
};
enum Gscale {
GFS_250DPS = 0,
GFS_500DPS,
GFS_1000DPS,
GFS_2000DPS
};

enum Mscale {
  MFS_14BITS = 0, // 0.6 mG per LSB
  MFS_16BITS      // 0.15 mG per LSB
};



uint8_t Gscale = GFS_250DPS;
uint8_t Ascale = AFS_2G;
uint8_t Mscale = MFS_16BITS; // Choose either 14-bit or 16-bit magnetometer resolution
uint8_t Mmode = 0x02;        // 2 for 8 Hz, 6 for 100 Hz continuous magnetometer data read
float aRes, gRes, mRes;      // scale resolutions per LSB for the sensors





int16_t magCount[3];    // Stores the 16-bit signed magnetometer sensor output
  // Factory mag calibration
int16_t accelCount[3]; // Stores the 16-bit signed accelerometer sensor output
int16_t gyroCount[3]; // Stores the 16-bit signed gyro sensor output
float magCalibration[3]={0,0,0};
float magBias[3] = {290.71, 111.07, -76.17};
//float magBias[3] = {150.61, 165.72, -160.80};
float gyroBias[3] = {1.09, -1.8, 2.4}, accelBias[3] = {0.01013, 0.01801, -0.00537}; // Bias corrections for gyro and accelerometer


float GyroMeasError = PI * (40.0f / 180.0f); // gyroscope measurement error in rads/s (start at 40 deg/s)
float GyroMeasDrift = PI * (0.0f / 180.0f); // gyroscope measurement drift in rad/s/s (start at 0.0 deg/s/s)
// There is a tradeoff in the beta parameter between accuracy and response speed.
```

```
// In the original Madgwick study, beta of 0.041 (corresponding to GyroMeasError of 2.7 degrees/s) was found to give optimal accuracy.
// However, with this value, the LSM9SD0 response time is about 10 seconds to a stable initial quaternion.
// Subsequent changes also require a longish lag time to a stable output, not fast enough for a quadcopter or robot car!
// By increasing beta (GyroMeasError) by about a factor of fifteen, the response time constant is reduced to ~2 sec
// I haven't noticed any reduction in solution accuracy. This is essentially the I coefficient in a PID control sense;
// the bigger the feedback coefficient, the faster the solution converges, usually at the expense of accuracy.
// In any case, this is the free parameter in the Madgwick filtering and fusion scheme.
float beta = sqrt(3.0f / 4.0f) * GyroMeasError; // compute beta
float zeta = sqrt(3.0f / 4.0f) * GyroMeasDrift; // compute zeta, the other free parameter in the Madgwick scheme usually set to a small or zero value
#define Kp 2.0f * 5.0f // these are the free parameters in the Mahony filter and fusion scheme, Kp for proportional feedback, Ki for integral
#define Ki 0.0f

float pitch, yaw, roll;


float deltat = 0.0f, sum = 0.0f;          // integration interval for both filter schemes
uint32_t lastUpdate = 0, firstUpdate = 0; // used to calculate integration interval
uint32_t Now = 0;
uint32_t delt_t = 0, count = 0, sumCount = 0;  // used to control display output rate
double ax=0, ay=0, az=0, gx=0 , gy=0 , gz=0,pitch1,roll1,gxint=0, gyint=0,xf,yf,zf,mx, my, mz;; // variables to hold latest sensor data values
double yawc;
float q[4] = {1.0f, 0.0f, 0.0f, 0.0f};    // vector to hold quaternion
float eInt[3] = {0.0f, 0.0f, 0.0f};       // vector to hold integral error for Mahony method
void imu_setup()
{

  Wire.begin();
  //Serial.begin(115200);
getAres();
getGres();
   getMres();
Serial.println(" Calibrate gyro and accel");
// calibrateMPU9250(gyroBias, accelBias);
//Serial.println("accel biases (mg)"); Serial.println(1000.*accelBias[0]);
```

```
Serial.println(1000.*accelBias[1]);
Serial.println(1000.*accelBias[2]);
//Serial.println("gyro biases (dps)"); Serial.println(gyroBias[0]); Serial.println(gyroBias[1]); Serial.println(gyroBias[2]);
 initMPU9250();
   initAK8963(magCalibration);
}
void imu_loop()
{
 if (readByte(MPU9250_ADDRESS, INT_STATUS) & 0x01) { // check if data ready interrupt
// if (digitalRead(intPin)) { // On interrupt, read data
readAccelData(accelCount); // Read the x/y/z adc values
// Now we'll calculate the accleration value into actual g's
ax = (float)accelCount[0]*aRes - accelBias[0]; // get actual g value, this depends on scale being set
ay = (float)accelCount[1]*aRes - accelBias[1];
az = (float)accelCount[2]*aRes - accelBias[2];
readGyroData(gyroCount); // Read the x/y/z adc values
// Calculate the gyro value into actual degrees per second
gx = (float)gyroCount[0]*gRes; // get actual gyro value, this depends on scale being set
gy = (float)gyroCount[1]*gRes;
gz = (float)gyroCount[2]*gRes;
 readMagData(magCount);  // Read the x/y/z adc values
    // Calculate the magnetometer values in milliGauss
    // Include factory calibration per data sheet and user environmental corrections
    mx = (float)magCount[0]*mRes*magCalibration[0] - magBias[0];  // get actual magnetometer value, this depends on scale being set
    my = (float)magCount[1]*mRes*magCalibration[1] - magBias[1];
    mz = (float)magCount[2]*mRes*magCalibration[2] - magBias[2];
}
 Now = micros();
 deltat = ((Now - lastUpdate)/1000000.0f); // set integration time by time elapsed since last filter update
 lastUpdate = Now;
sum += deltat; // sum for averaging filter update rate
```

```
sumCount++;
MadgwickQuaternionUpdate(ax, ay, az, gx*PI/180.0f, gy*PI/180.0f, gz*PI/180.0f,  my,  mx, mz);
yawc   = atan2(2.0f * (q[1] * q[2] + q[0] * q[3]), q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] * q[3]);
yawc   *= 180.0f / PI;
yawc   -= 0.17f; // Declination at Danville, California is 13 degrees 48 minutes and 47 seconds on 2014-04-04
pitch1 = atan2(ay,sqrt(ax*ax + az*az))*180/PI;
roll1 = atan2(-ax,az)*180/PI;
//headx =  mx * cos(pitch1*PI/180) + my * sin(pitch1*PI/180) * sin(roll1*PI/180) + mz * sin(pitch1*PI/180) * cos(roll1*PI/180);
//heady =my * cos(roll1*PI/180) - mz * sin(roll1*PI/180);
//yawc = atan2(-heady,headx)*180/PI;
xf = .90 * ( roll1+ gx * deltat) + .10 * ax;
yf = .90 * (pitch1+ gy * deltat) + .10 * ay;
//zf = .90 * (gz * deltat) + .10 * mz;
//Serial.print("ax = "); Serial.print((int)1000*ax);
//Serial.print(" ay = "); Serial.print((int)1000*ay);
//Serial.print(" az = "); Serial.print((int)1000*az);
//Serial.print("\troll = "); Serial.print((int)roll1);
//Serial.print("\tpitch = "); Serial.print((int)pitch1);
//Serial.print("\tgx = "); Serial.print( gxint, 2);
//Serial.print("\tgy = "); Serial.print( gyint, 2);
Serial.print("\txf = "); Serial.print( xf, 2);
Serial.print("\tyf = "); Serial.print( yf, 2);
Serial.print("\tzf = "); Serial.print( gz, 2);
//Serial.print("\taccel bias x  "); Serial.print(accelBias[0] , 2);
// Serial.print("\taccel bias y "); Serial.print( , 2);
Serial.println("");
Serial.println(deltat);
sumCount = 0;
sum = 0;
delay(100);
}
void getMres() {
  switch (Mscale)
  {
   // Possible magnetometer scales (and their register bit settings) are:
   // 14 bit resolution (0) and 16 bit resolution (1)
     case MFS_14BITS:
           mRes = 10.*4912./8190.; // Proper scale to return milliGauss
           break;
     case MFS_16BITS:
           mRes = 10.*4912./32760.0; // Proper scale to return milliGauss
           break;
  }
}
```

```
void getGres() {
switch (Gscale)
{
// Possible gyro scales (and their register bit settings) are:
// 250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS (11).
// Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-bit value:
case GFS_250DPS:
gRes = 250.0/32768.0;
break;
case GFS_500DPS:
gRes = 500.0/32768.0;
break;
case GFS_1000DPS:
gRes = 1000.0/32768.0;
break;
case GFS_2000DPS:
gRes = 2000.0/32768.0;
break;
}
}
void getAres() {
switch (Ascale)
{
// Possible accelerometer scales (and their register bit settings) are:
// 2 Gs (00), 4 Gs (01), 8 Gs (10), and 16 Gs (11).
// Here's a bit of an algorith to calculate DPS/(ADC tick) based on that 2-bit value:
case AFS_2G:
aRes = 2.0/32768.0;
break;
case AFS_4G:
aRes = 4.0/32768.0;
break;
case AFS_8G:
aRes = 8.0/32768.0;
break;
case AFS_16G:
aRes = 16.0/32768.0;
break;
}
}
void readAccelData(int16_t * destination)
{
uint8_t rawData[6]; // x/y/z accel register data stored here
readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]); // Read the six raw data registers into data array
destination[0] = -((int16_t)rawData[0] << 8) | rawData[1] ; // Turn the MSB and LSB into a signed 16-bit value
destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
}
void readGyroData(int16_t * destination)
```

```
{
uint8_t rawData[6]; // x/y/z gyro register data stored here
readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]); // Read the six raw data registers sequentially into data array
destination[0] = -((int16_t)rawData[0] << 8) | rawData[1] ; // Turn the MSB and LSB into a signed 16-bit value
destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
}
void readMagData(int16_t * destination)
{
  uint8_t rawData[7];  // x/y/z gyro register data, ST2 register stored here, must read ST2 at end of data acquisition
  if(readByte(AK8963_ADDRESS, AK8963_ST1) & 0x01) { // wait for magnetometer data ready bit to be set
  readBytes(AK8963_ADDRESS, AK8963_XOUT_L, 7, &rawData[0]);  // Read the six raw data and ST2 registers sequentially into data array
  uint8_t c = rawData[6]; // End data read by reading ST2 register
    if(!(c & 0x08)) { // Check if magnetic sensor overflow set, if not then report data
    destination[0] = ((int16_t)rawData[1] << 8) | rawData[0] ;  // Turn the MSB and LSB into a signed 16-bit value
    destination[1] = ((int16_t)rawData[3] << 8) | rawData[2] ;  // Data stored as little Endian
    destination[2] = ((int16_t)rawData[5] << 8) | rawData[4] ;
   }
  }
}


void initAK8963(float * destination)
{
  // First extract the factory calibration for each magnetometer axis
  uint8_t rawData[3];  // x/y/z gyro calibration data stored here
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down magnetometer
  delay(10);
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x0F); // Enter Fuse ROM access mode
  delay(10);
  readBytes(AK8963_ADDRESS, AK8963_ASAX, 3, &rawData[0]);  // Read the x-, y-, and z-axis calibration values
  destination[0] =  (float)(rawData[0] - 128)/256. + 1.;   // Return x-axis sensitivity adjustment values, etc.
  destination[1] =  (float)(rawData[1] - 128)/256. + 1.;
  destination[2] =  (float)(rawData[2] - 128)/256. + 1.;
  writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down magnetometer
  delay(10);
  // Configure the magnetometer for continuous read and highest resolution
  // set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution in CNTL register,
```

```
  // and enable continuous mode data acquisition Mmode (bits [3:0]), 0010 for 8 Hz and 0110 for 100 Hz sample rates
  writeByte(AK8963_ADDRESS, AK8963_CNTL, Mscale << 4 | Mmode); // Set magnetometer data resolution and sample ODR
  delay(10);
}
void initMPU9250()
{
// wake up device
writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00); // Clear sleep mode bit (6), enable all sensors
delay(100); // Wait for all registers to reset
// get stable time source
writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01); // Auto select clock source to be PLL gyroscope reference if ready else
delay(200);
// Configure Gyro and Thermometer
// Disable FSYNC and set thermometer and gyro bandwidth to 41 and 42 Hz, respectively;
// minimum delay time for this setting is 5.9 ms, which means sensor fusion update rates cannot
// be higher than 1 / 0.0059 = 170 Hz
// DLPF_CFG = bits 2:0 = 011; this limits the sample rate to 1000 Hz for both
// With the MPU9250, it is possible to get gyro sample rates of 32 kHz (!), 8 kHz, or 1 kHz
writeByte(MPU9250_ADDRESS, CONFIG, 0x03);
// Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x04); // Use a 200 Hz rate; a rate consistent with the filter update rate
// determined inset in CONFIG above
// Set gyroscope full scale range
// Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values are left-shifted into positions 4:3
uint8_t c = readByte(MPU9250_ADDRESS, GYRO_CONFIG); // get current GYRO_CONFIG register value
// c = c & ~0xE0; // Clear self-test bits [7:5]
c = c & ~0x02; // Clear Fchoice bits [1:0]
c = c & ~0x18; // Clear AFS bits [4:3]
c = c | Gscale << 3; // Set full scale range for the gyro
// c =| 0x00; // Set Fchoice for the gyro to 11 by writing its inverse to bits 1:0 of GYRO_CONFIG
writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c ); // Write new GYRO_CONFIG value to register
// Set accelerometer full-scale range configuration
c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG); // get current ACCEL_CONFIG register value
// c = c & ~0xE0; // Clear self-test bits [7:5]
c = c & ~0x18; // Clear AFS bits [4:3]
c = c | Ascale << 3; // Set full scale range for the accelerometer
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c); // Write new ACCEL_CONFIG register value
// Set accelerometer sample rate configuration
```

```
// It is possible to get a 4 kHz sample rate from the accelerometer by choosing 1 for
// accel_fchoice_b bit [3]; in this case the bandwidth is 1.13 kHz
c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG2); // get current ACCEL_CONFIG2 register value
c = c & ~0x0F; // Clear accel_fchoice_b (bit 3) and A_DLPFG (bits [2:0])
c = c | 0x03; // Set accelerometer rate to 1 kHz and bandwidth to 41 Hz
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c); // Write new ACCEL_CONFIG2 register value
// The accelerometer, gyro, and thermometer are set to 1 kHz sample rates,
// but all these rates are further reduced by a factor of 5 to 200 Hz because of the SMPLRT_DIV setting
// Configure Interrupts and Bypass Enable
// Set interrupt pin active high, push-pull, hold interrupt pin level HIGH until interrupt cleared,
// clear on read of INT_STATUS, and enable I2C_BYPASS_EN so additional chips
// can join the I2C bus and all can be controlled by the Arduino as master
writeByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x22);
writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x01); // Enable data ready (bit 0) interrupt
delay(100);
}

void calibrateMPU9250(float * dest1, float * dest2)
{
uint8_t data[12]; // data array to hold accelerometer and gyro x, y, z, data
uint16_t ii, packet_count, fifo_count;
int32_t gyro_bias[3] = {0, 0, 0}, accel_bias[3] = {0, 0, 0};
// reset device
writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x80); // Write a one to bit 7 reset bit; toggle reset device
delay(100);
// get stable time source; Auto select clock source to be PLL gyroscope reference if ready
// else use the internal oscillator, bits 2:0 = 001
writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
writeByte(MPU9250_ADDRESS, PWR_MGMT_2, 0x00);
delay(200);
// Configure device for bias calculation
writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x00); // Disable all interrupts
writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00); // Disable FIFO
writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00); // Turn on internal clock source
writeByte(MPU9250_ADDRESS, I2C_MST_CTRL, 0x00); // Disable I2C master
writeByte(MPU9250_ADDRESS, USER_CTRL, 0x00); // Disable FIFO and I2C master modes
writeByte(MPU9250_ADDRESS, USER_CTRL, 0x0C); // Reset FIFO and DMP
```

```
delay(15);
// Configure MPU6050 gyro and accelerometer for bias calculation
writeByte(MPU9250_ADDRESS, CONFIG, 0x01); // Set low-pass filter to 188 Hz
writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00); // Set sample rate to 1 kHz
writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00); // Set gyro full-scale to 250 degrees per second, maximum sensitivity
writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00); // Set accelerometer full-scale to 2 g, maximum sensitivity
uint16_t gyrosensitivity = 131; // = 131 LSB/degrees/sec
uint16_t accelsensitivity = 16384; // = 16384 LSB/g
// Configure FIFO to capture accelerometer and gyro data for bias calculation
writeByte(MPU9250_ADDRESS, USER_CTRL, 0x40); // Enable FIFO
writeByte(MPU9250_ADDRESS, FIFO_EN, 0x78); // Enable gyro and accelerometer sensors for FIFO (max size 512 bytes in MPU-9150)
delay(40); // accumulate 40 samples in 40 milliseconds = 480 bytes
// At end of sample accumulation, turn off FIFO sensor read
writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00); // Disable gyro and accelerometer sensors for FIFO
readBytes(MPU9250_ADDRESS, FIFO_COUNTH, 2, &data[0]); // read FIFO sample count
fifo_count = ((uint16_t)data[0] << 8) | data[1];
packet_count = fifo_count/12;// How many sets of full gyro and accelerometer data for averaging
for (ii = 0; ii < packet_count; ii++) {
int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
readBytes(MPU9250_ADDRESS, FIFO_R_W, 12, &data[0]); // read data for averaging
accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1] ) ; // Form signed 16-bit integer for each sample in FIFO
accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3] ) ;
accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5] ) ;
gyro_temp[0] = (int16_t) (((int16_t)data[6] << 8) | data[7] ) ;
gyro_temp[1] = (int16_t) (((int16_t)data[8] << 8) | data[9] ) ;
gyro_temp[2] = (int16_t) (((int16_t)data[10] << 8) | data[11]) ;
accel_bias[0] += (int32_t) accel_temp[0]; // Sum individual signed 16-bit biases to get accumulated signed 32-bit biases
accel_bias[1] += (int32_t) accel_temp[1];
accel_bias[2] += (int32_t) accel_temp[2];
gyro_bias[0] += (int32_t) gyro_temp[0];
gyro_bias[1] += (int32_t) gyro_temp[1];
gyro_bias[2] += (int32_t) gyro_temp[2];
}
accel_bias[0] /= (int32_t) packet_count; // Normalize sums to get average count biases
accel_bias[1] /= (int32_t) packet_count;
accel_bias[2] /= (int32_t) packet_count;
gyro_bias[0] /= (int32_t) packet_count;
gyro_bias[1] /= (int32_t) packet_count;
gyro_bias[2] /= (int32_t) packet_count;
```

```
if(accel_bias[2] > 0L) {accel_bias[2] -= (int32_t) accelsensitivity;} // Remove gravity from the z-axis accelerometer bias calculation
else {accel_bias[2] += (int32_t) accelsensitivity;}
// Construct the gyro biases for push to the hardware gyro bias registers, which are reset to zero upon device startup
data[0] = (-gyro_bias[0]/4 >> 8) & 0xFF; // Divide by 4 to get 32.9 LSB per deg/s to conform to expected bias input format
data[1] = (-gyro_bias[0]/4) & 0xFF; // Biases are additive, so change sign on calculated average gyro biases
data[2] = (-gyro_bias[1]/4 >> 8) & 0xFF;
data[3] = (-gyro_bias[1]/4) & 0xFF;
data[4] = (-gyro_bias[2]/4 >> 8) & 0xFF;
data[5] = (-gyro_bias[2]/4) & 0xFF;
// Push gyro biases to hardware registers
writeByte(MPU9250_ADDRESS, XG_OFFSET_H, data[0]);
writeByte(MPU9250_ADDRESS, XG_OFFSET_L, data[1]);
writeByte(MPU9250_ADDRESS, YG_OFFSET_H, data[2]);
writeByte(MPU9250_ADDRESS, YG_OFFSET_L, data[3]);
writeByte(MPU9250_ADDRESS, ZG_OFFSET_H, data[4]);
writeByte(MPU9250_ADDRESS, ZG_OFFSET_L, data[5]);
// Output scaled gyro biases for display in the main program
dest1[0] = (float) gyro_bias[0]/(float) gyrosensitivity;
dest1[1] = (float) gyro_bias[1]/(float) gyrosensitivity;
dest1[2] = (float) gyro_bias[2]/(float) gyrosensitivity;
// Construct the accelerometer biases for push to the hardware accelerometer bias registers. These registers contain
// factory trim values which must be added to the calculated accelerometer biases; on boot up these registers will hold
// non-zero values. In addition, bit 0 of the lower byte must be preserved since it is used for temperature
// compensation calculations. Accelerometer bias registers expect bias input as 2048 LSB per g, so that
// the accelerometer biases calculated above must be divided by 8.
int32_t accel_bias_reg[3] = {0, 0, 0}; // A place to hold the factory accelerometer trim biases
readBytes(MPU9250_ADDRESS, XA_OFFSET_H, 2, &data[0]); // Read factory accelerometer trim values
accel_bias_reg[0] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
readBytes(MPU9250_ADDRESS, YA_OFFSET_H, 2, &data[0]);
accel_bias_reg[1] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
readBytes(MPU9250_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
accel_bias_reg[2] = (int32_t) (((int16_t)data[0] << 8) | data[1]);
uint32_t mask = 1uL; // Define mask for temperature compensation bit 0 of lower byte of accelerometer bias registers
uint8_t mask_bit[3] = {0, 0, 0}; // Define array to hold mask bit for each accelerometer bias axis
for(ii = 0; ii < 3; ii++) {
if((accel_bias_reg[ii] & mask)) mask_bit[ii] = 0x01; // If temperature compensation bit is set, record that fact in mask_bit
}
// Construct total accelerometer bias, including calculated average accelerometer bias from above
```

```
accel_bias_reg[0] -= (accel_bias[0]/8); // Subtract calculated averaged accelerometer bias scaled to 2048 LSB/g (16 g full scale)
accel_bias_reg[1] -= (accel_bias[1]/8);
accel_bias_reg[2] -= (accel_bias[2]/8);
data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
data[1] = (accel_bias_reg[0]) & 0xFF;
data[1] = data[1] | mask_bit[0]; // preserve temperature compensation bit when writing back to accelerometer bias registers
data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
data[3] = (accel_bias_reg[1]) & 0xFF;
data[3] = data[3] | mask_bit[1]; // preserve temperature compensation bit when writing back to accelerometer bias registers
data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
data[5] = (accel_bias_reg[2]) & 0xFF;
data[5] = data[5] | mask_bit[2]; // preserve temperature compensation bit when writing back to accelerometer bias registers
// Apparently this is not working for the acceleration biases in the MPU-9250
// Are we handling the temperature correction bit properly?
// Push accelerometer biases to hardware registers
writeByte(MPU9250_ADDRESS, XA_OFFSET_H, data[0]);
writeByte(MPU9250_ADDRESS, XA_OFFSET_L, data[1]);
writeByte(MPU9250_ADDRESS, YA_OFFSET_H, data[2]);
writeByte(MPU9250_ADDRESS, YA_OFFSET_L, data[3]);
writeByte(MPU9250_ADDRESS, ZA_OFFSET_H, data[4]);
writeByte(MPU9250_ADDRESS, ZA_OFFSET_L, data[5]);
// Output scaled accelerometer biases for display in the main program
dest2[0] = (float)accel_bias[0]/(float)accelsensitivity;
dest2[1] = (float)accel_bias[1]/(float)accelsensitivity;
dest2[2] = (float)accel_bias[2]/(float)accelsensitivity;
}

 void writeByte(uint8_t address, uint8_t subAddress, uint8_t data)
{
Wire.beginTransmission(address); // Initialize the Tx buffer
Wire.write(subAddress); // Put slave register address in Tx buffer
Wire.write(data); // Put data in Tx buffer
Wire.endTransmission(); // Send the Tx buffer
}
uint8_t readByte(uint8_t address, uint8_t subAddress)
{
uint8_t data; // `data` will store the register data
Wire.beginTransmission(address); // Initialize the Tx buffer
Wire.write(subAddress); // Put slave register address in Tx buffer
Wire.endTransmission(); // Send the Tx buffer, but send a restart to keep connection alive
// Wire.endTransmission(false); // Send the Tx buffer, but send a restart to keep connection alive
// Wire.requestFrom(address, 1); // Read one byte from slave register address
Wire.requestFrom(address, (size_t) 1); // Read one byte from slave register address
data = Wire.read(); // Fill Rx buffer with result
```

```
return data; // Return data read from slave register
}
void readBytes(uint8_t address, uint8_t subAddress, uint8_t count, uint8_t * dest)
{
Wire.beginTransmission(address); // Initialize the Tx buffer
Wire.write(subAddress); // Put slave register address in Tx buffer
Wire.endTransmission(); // Send the Tx buffer, but send a restart to keep connection alive
// Wire.endTransmission(false); // Send the Tx buffer, but send a restart to keep connection alive
uint8_t i = 0;
// Wire.requestFrom(address, count); // Read bytes from slave register address
Wire.requestFrom(address, (size_t) count); // Read bytes from slave register address
while (Wire.available()) {
dest[i++] = Wire.read(); } // Put read results in the Rx buffer
}
void MadgwickQuaternionUpdate(float ax, float ay, float az, float gx, float gy, float gz, float mx, float my, float mz)
        {
            float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];   // short name local variable for readability
            float norm;
            float hx, hy, _2bx, _2bz;
            float s1, s2, s3, s4;
            float qDot1, qDot2, qDot3, qDot4;

            // Auxiliary variables to avoid repeated arithmetic
            float _2q1mx;
            float _2q1my;
            float _2q1mz;
            float _2q2mx;
            float _4bx;
            float _4bz;
            float _2q1 = 2.0f * q1;
            float _2q2 = 2.0f * q2;
            float _2q3 = 2.0f * q3;
            float _2q4 = 2.0f * q4;
            float _2q1q3 = 2.0f * q1 * q3;
            float _2q3q4 = 2.0f * q3 * q4;
            float q1q1 = q1 * q1;
            float q1q2 = q1 * q2;
            float q1q3 = q1 * q3;
            float q1q4 = q1 * q4;
            float q2q2 = q2 * q2;
            float q2q3 = q2 * q3;
            float q2q4 = q2 * q4;
            float q3q3 = q3 * q3;
            float q3q4 = q3 * q4;
            float q4q4 = q4 * q4;

```

```
        // Normalise accelerometer measurement
        norm = sqrt(ax * ax + ay * ay + az * az);
        if (norm == 0.0f) return; // handle NaN
        norm = 1.0f/norm;
        ax *= norm;
        ay *= norm;
        az *= norm;

        // Normalise magnetometer measurement
        norm = sqrt(mx * mx + my * my + mz * mz);
        if (norm == 0.0f) return; // handle NaN
        norm = 1.0f/norm;
        mx *= norm;
        my *= norm;
        mz *= norm;

        // Reference direction of Earth's magnetic field
        _2q1mx = 2.0f * q1 * mx;
        _2q1my = 2.0f * q1 * my;
        _2q1mz = 2.0f * q1 * mz;
        _2q2mx = 2.0f * q2 * mx;
        hx = mx * q1q1 - _2q1my * q4 + _2q1mz * q3 + mx * q2q2 + _2q2 * my * q3 + _2q2 * mz * q4 - mx * q3q3 - mx * q4q4;
        hy = _2q1mx * q4 + my * q1q1 - _2q1mz * q2 + _2q2mx * q3 - my * q2q2 + my * q3q3 + _2q3 * mz * q4 - my * q4q4;
        _2bx = sqrt(hx * hx + hy * hy);
        _2bz = -_2q1mx * q3 + _2q1my * q2 + mz * q1q1 + _2q2mx * q4 - mz * q2q2 + _2q3 * my * q4 - mz * q3q3 + mz * q4q4;
        _4bx = 2.0f * _2bx;
        _4bz = 2.0f * _2bz;

        // Gradient decent algorithm corrective step
        s1 = -_2q3 * (2.0f * q2q4 - _2q1q3 - ax) + _2q2 * (2.0f * q1q2 + _2q3q4 - ay) - _2bz * q3 * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (-_2bx * q4 + _2bz * q2) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + _2bx * q3 * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
        s2 = _2q4 * (2.0f * q2q4 - _2q1q3 - ax) + _2q1 * (2.0f * q1q2 + _2q3q4 - ay) - 4.0f * q2 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + _2bz * q4 * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q3 + _2bz * q1) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + (_2bx * q4 - _4bz * q2) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
        s3 = -_2q1 * (2.0f * q2q4 - _2q1q3 - ax) + _2q4 * (2.0f * q1q2 + _2q3q4 - ay) - 4.0f * q3 * (1.0f - 2.0f * q2q2 - 2.0f * q3q3 - az) + (-_4bx * q3 - _2bz * q1) * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (_2bx * q2 + _2bz * q4) * (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + (_2bx * q1 - _4bz * q3) * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
        s4 = _2q2 * (2.0f * q2q4 - _2q1q3 - ax) + _2q3 * (2.0f * q1q2 + _2q3q4 - ay) + (-_4bx * q4 + _2bz * q2) * (_2bx * (0.5f - q3q3 - q4q4) + _2bz * (q2q4 - q1q3) - mx) + (-_2bx * q1 + _2bz * q3) *
```

```
    (_2bx * (q2q3 - q1q4) + _2bz * (q1q2 + q3q4) - my) + _2bx * q2 * (_2bx * (q1q3 + q2q4) + _2bz * (0.5f - q2q2 - q3q3) - mz);
            norm = sqrt(s1 * s1 + s2 * s2 + s3 * s3 + s4 * s4);    // normalise step magnitude
            norm = 1.0f/norm;
            s1 *= norm;
            s2 *= norm;
            s3 *= norm;
            s4 *= norm;

            // Compute rate of change of quaternion
            qDot1 = 0.5f * (-q2 * gx - q3 * gy - q4 * gz) - beta * s1;
            qDot2 = 0.5f * (q1 * gx + q3 * gz - q4 * gy) - beta * s2;
            qDot3 = 0.5f * (q1 * gy - q2 * gz + q4 * gx) - beta * s3;
            qDot4 = 0.5f * (q1 * gz + q2 * gy - q3 * gx) - beta * s4;

            // Integrate to yield quaternion
            q1 += qDot1 * deltat;
            q2 += qDot2 * deltat;
            q3 += qDot3 * deltat;
            q4 += qDot4 * deltat;
            norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);    // normalise quaternion
            norm = 1.0f/norm;
            q[0] = q1 * norm;
            q[1] = q2 * norm;
            q[2] = q3 * norm;
            q[3] = q4 * norm;

        }







PID roll_controller(&pid_roll_in,   &pid_roll_out,  &pid_roll_setpoint, ROLL_PID_KP, ROLL_PID_KI, ROLL_PID_KD, REVERSE);
PID pitch_controller(&pid_pitch_in, &pid_pitch_out, &pid_pitch_setpoint, PITCH_PID_KP , PITCH_PID_KI, PITCH_PID_KD, REVERSE);
PID yaw_controller(&pid_yaw_in,     &pid_yaw_out,   &pid_yaw_setpoint, YAW_PID_KP, YAW_PID_KI, YAW_PID_KD, DIRECT);


void pid_initialize() {
  roll_controller.SetOutputLimits(ROLL_PID_MIN,ROLL_PID_MAX);
  pitch_controller.SetOutputLimits(PITCH_PID_MIN,PITCH_PID_MAX);
  yaw_controller.SetOutputLimits(YAW_PID_MIN,YAW_PID_MAX);
  roll_controller.SetMode(AUTOMATIC);
  pitch_controller.SetMode(AUTOMATIC);
  yaw_controller.SetMode(AUTOMATIC);
  roll_controller.SetSampleTime(PID_SAMPLE_TIME );
  pitch_controller.SetSampleTime(PID_SAMPLE_TIME );
```

```
  yaw_controller.SetSampleTime(PID_SAMPLE_TIME );
}

void pid_update(){
  pid_roll_in = xf;
  pid_pitch_in = yf;
  pid_yaw_in = gz;
}

void pid_compute() {
   roll_controller.Compute();
   pitch_controller.Compute();
   yaw_controller.Compute();

}
```

9. REFERENCES(Website)

- **www.playground.arduino.cc**
- **Nrf24L01+ datasheets**
- **Invensense MPU9250 datasheet**