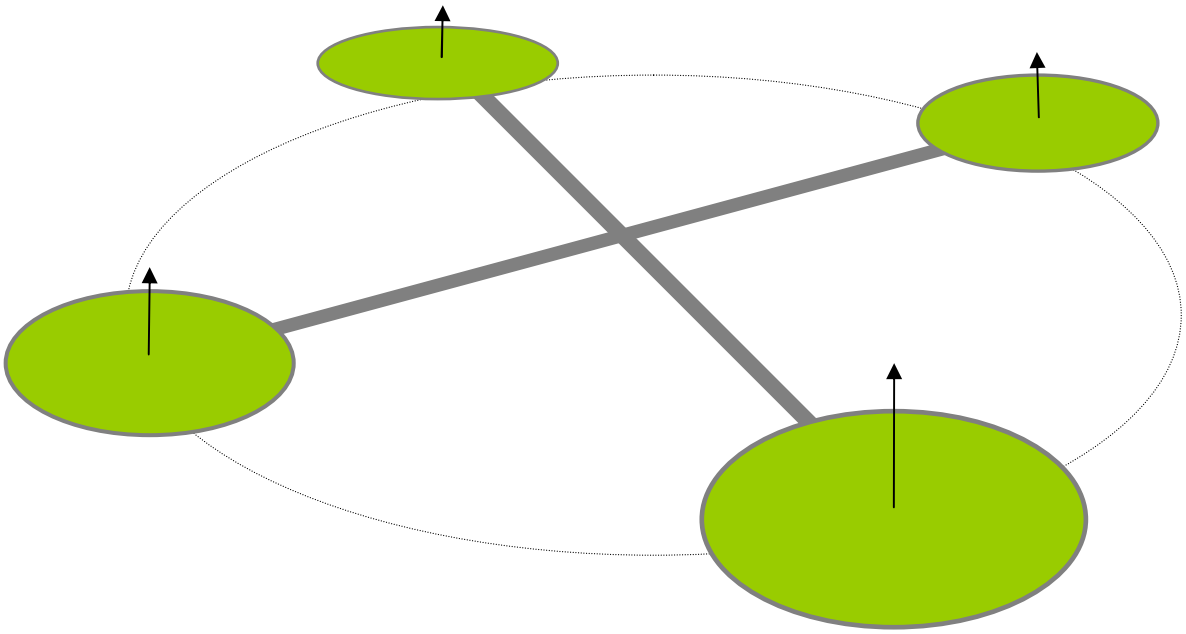
DIY Raspberry Pi
Autonomous Quadcopter
Construction Guide

DIY Raspberry Pi Python Autonomous Quadcopter



Catchy title, eh?

Towards the end of 2012, I'd done a few little projects with my Raspberry Pi, and was looking for a significant challenge for the next one. I was totally ignorant about quadcopters other than that they flew; I had no idea whether a Raspberry Pi could be used to control one. So I decided to try.

My key aims were:

- to use the Raspberry Pi to provide as much of the necessary function as possible
- to code it in Python rather than C because C is too closely tied to my day job.

I started out also with the aim of building a quadcopter controlled, via a Raspberry Pi remote control, by a human. I decided the first step of the project was to produce an autonomous quadcopter that flies according to preconfigured flight plans before inserting the human into the mix. This, it turns out, is a lot more complex than one with a human in control! I've now given up on the thought of adding a remote control, so this guide makes no further attempt to explain how to add a human into the control mix.

All flying machines and boats have a name; it is usually female. I named my quadcopter Phoebe. Throughout this guide, I use "Phoebe" in preference to "the quadcopter". Partly because it has fewer syllables, and partly, I've formed a love / hate relationship with her over the last 18 months, and I now feel awkward referring to her as "the | my | your quadcopter".

Introduction

This guide does not provide a set of concise instructions how to build your quadcopter; that's simply not possible. It attempts to provide guidelines as to the hardware and software required, along with explanation of how these work together to fly a quadcopter.

There are further details in the appendices with specific pieces of what you need, how to put them all together, and how to test them for flight, but once more, these are general guidelines since every quadcopter is different.

However, I hope that this guide does provide enough information that you can understand how quadcopters work and so cna build your own quadcopter with confidence and success!

The contents of the guide are based upon what I've learnt over the past 18 months, as I started from complete ignorance and journeyed to producing a fully functional autonomous quadcopter. All the details of this story are on my blog at blog.pistuffing.co.uk along with all the mistakes I made along the way.

This guide distils all the useful information from the blog, removing all the rubbish and wrong directions I took along the way. Hopefully this means you can build your quadcopter in perhaps under a month rather than the 18months it took me!

Chapter 1 - Quadcopter components

My aim for building a quadcopter was to use the Raspberry Pi to do as much as possible, using off-the-shelf parts only for hardware (i.e. the frame) and high powered electronics (i.e. the connection to motors from the Raspberry Pi).

Motors and Propellers

First the mandatory warning: propellers blades are called so as they cut through the air. They are also very capable of cutting through you. Always be careful and always be ready to run.

There's a huge variety of propellers; I can't really give advice as to what's best for your setup. The one piece of advice I have learned is that for a given set of props, you need to use a set of motors capable of running them both to speed them up and slow them down.

The lighter the propeller, the easier it is for a broader set of motors to be used.

However having spend an awful lot of money trying out motor / propeller combinations, my advice would be to buy matched pairs - probably the best I've used have been the combination of props and motors made by DJI for their F450 platform, and at the other end of the cost spectrum, the props and motors made by T-motor.

The motors used for quadcopters are brushless; that means the magnets are attached to the spindle, and coils are attached to the body; as current is passed though a could, the magnets are attracted to it, and the motors rotates until the motor and could is aligned.

At that point, the neighbour coil is powered up, and the current one powered down, and the motors moves round a little further. By doing this very quickly, the motors rotate.

I hope you can see though that the motor requires several wires to control which set of coils are powered. There are typically 3 wires, and the power is switched between pairs of the 3 to move the motors. Motor control boards are used to handle these three phases at the high currents required and control their speed. The boards are the key component of...

ESCs

ESC stands for electronic speed controller. It takes a single signal from the Flight Controller, and converts that to high power pulses to the motors. There's a microcontroller at its core.

If you're interested in what the microcontroller does in detail, I recommend googling "Simon Kirby ESC firmware" which is open source.

I use the 30A ESCs supplied by DJI for their F450 flamewheel kit.

Batteries

Quadcopters are generally powered by LiPo (lithium polymer) batteries. These have a huge power to weight ratio - for the amount of power they can deliver, they weigh very little. This is the primary reason quadcopters have recently become available at affordable prices and hence flying them is viable as an affordable hobby.

But this huge power to weight ratio makes them very similar to a bomb, and if mistreated, they will burst into flames or explode. Treat them with due respect:

- be careful not to short the power leads
- always use a LiPo charger

Without power for the Raspberry Pi and the motors, we're going nowhere. This was the first stage I got sorted.

A quadcopter requires vast amounts of power for the motors; at the same time, it needs to be as light as possible. As a result, virtually ever quadcopter on the planet uses Lithium Polymer (LiPo) batteries.

A single LiPo cell puts out a nominal 3.7 volts. Quadcopters are usually driven by several of these cells strung together in series to produce a battery (of cells) providing higher output voltages. For some reason, a magic language has developed to categorize the battery types. For example, the ones I use are

3S, 30C, 4000mAh, 1C

4000mAh was the only term I recognised; it has a storage capacity 4000mAh which means the battery will can provide power at the specified voltage at 4000mA (4.0 Amps) for an hour.

But what's the specified voltages? That's the **3S** - **3** x 3.7v LiPo cells in **S**eries make up a nominal 11.1 V battery. I believe cells can be wired together in parallel to increase their power giving a definition along the lines of 3S2P (3 series, 2 parallel = 6 cells), but don't quote me on that.

40 - 80C is the burst power the battery can produce - it's 30 time the capacity of 4000mAh - it says the battery can put out 120 - 240A as a power burst - and that's a lot.

The final 1**C** refers to how fast the battery can be charged compared to the storage **C**apacity - in this case, based on the storage capacity of 4000mAh, the maximum charge current is 4000mA or 4.0A. This is something you set up on your charger to protected the battery while charging.

With all that power in such a compact package, Lithium batteries resemble a bomb, and are known to explode if abused. They must be charged with care, and discharged with care, and used within safe bounds - critically then must not be discharged below a lower limit or else they become dangerous to recharge.

For this reason, the batteries contain safety circuitry to protect against short-circuits, and the batteries must be charged with specific LiPo chargers which are able to manage the charge rate according to what it detects from the battery.

The power for the motors is supplied by these batteries, but the Raspberry Pi needs a 5V feed - where does that come from?

In the world of quadcopters, all sorts of technical terms are hidden behind non-obvious terminology - the meaning of the battery symbols is one example. Another is the BEC or UBEC - the (Universal) Battery Electronic Controller - utterly meaningless. It's actually a voltage regulator to convert the battery voltage to the voltage level required by the control circuitry. I only discovered this a month or two ago. By then I'd been using a switching voltage regulator for 9 months to produce the 5V required by the Raspberry Pi - more on that in the section on circuitry.

Body

Some people do build their own frame very successfully - at its most basic, it's just a pair of square aluminium pipes set at right angles to each other. Have a look online if doing your own frame tickles your fancy. It tickled nothing for me, so I bought a kit containing the frame, ESCs, motors and propellers (see Appendix XX for details), allowing me to get on with the bits I was interested in.

Whether you buy or build the frame, you still need to find places for the battery / Raspberry Pi and sensors to live. They need to be very firmly attached, and protected from crashes. I ended up with a mix of Velcro, double sided sticky tape, duct-tape and blu-tack all chosen specifically for their purpose.

Case + Breadboard

TBD

Domes

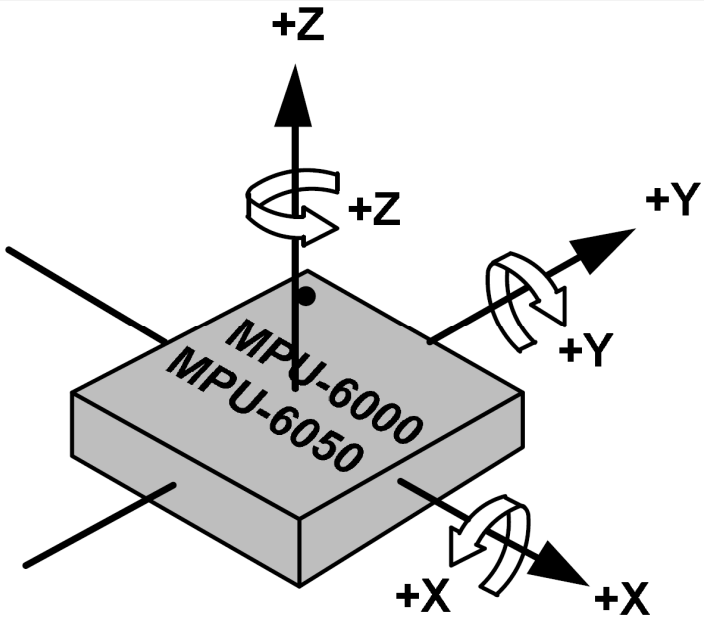
The Raspberry Pi + breadboard sit on top of the quadcopter platform. They have the highest position and hence require protection from crashes. I use a Perspex dome with a flange from e-bay, and it works perfectly. The flange is velcro'd to the arms of the platform.

I've tried various types of underside for the frame. Legs of various varieties break, and also increase the chance of propellers hitting the ground if the quad lands the ground while moving. I now have a sponge tennis training ball, cut in halt, and glued to Phoebe's underside. It's been perfect - far fewer landings that have the props hitting the ground, and soft landings for test-flights I'm forced to cancel mid-flight.

Sensors

Phoebe only has one sensor unit: the MPU 6050 made by Invensense. It is an IMU (Inertial Measurement Unit) which measures gravity using 3 accelerometers and rotation rate using 3 gyroscopes.

The 6 sensors follow the right-hand rule. The right-hand rule is an easy to remember guide to which way is positive for the sensors. The following diagram shows this:



For the accelerometer, to apply the right hand rule, using your thumb, index- and middle-finger, point them at 90 degrees to each other; then point one of your digits in the direction of the arrows in the diagram - you see your other two digits are pointing the same direction as the other two arrows. Your fingers and thumb are all pointing in the directions which produce positive values for the direction of acceleration.

For the gyroscope, make a fist and stick out your thumb. Point you thumb in the direction of one of the axes; your fingers then indicate which direction of rotation around that axis is positive.

For details about how to use the sensor, and how Phoebe uses the sensor, read the MPU6050 data sheets (google for them) and Phoebe's code.

We'll see the reason behind this as we come to discuss the math(s) required by the flight controller.

TBD - compass / magnetometer + altimeter / barometer + GPS

Flight Controller

The fact that helicopters and quadcopters fly is a remarkable feat of engineering and technology. How do they move forwards? How do they not get blown around by the wind? Why don't they just spin round and round and crash?

Planes are easy in comparison - think of all those paper planes you made as a child. Planes are innately stable.

Helicopters and quads are innately unstable. They need a pilot to keep them stable. It's slightly easier with a helicopter as they have a tail propeller to stop them spinning. They can also tilt their top propeller to make the helicopter go in a chosen direction.

Quadcopters don't have either. Instead they have a "flight controller" to maintain stability in the air. Normal quadcopters also have a human pilot, controlling which direction it moves via a remote control. In Phoebe's case, she doesn't have a pilot: she's autonomous, which in this case means she follows a pre-configured flight plan.

Phoebe's flight control is based on a Raspberry Pi, running software written in Python. It reads the sensors using the results to check what's happening compared to the flight plan, and changes how fast each of the propellers is spinning to fix any differences. A lot more on detail on this later.

Network connectivity

Whether autonomous or under user remote control, there needs to be some way to connect to Phoebe, regardless of her location - i.e. in the middle of nowhere with no internet or home WiFi access. Phoebe needs to be a wireless access point.

Given the right WiFi dongle, this isn't hard; the key term you are looking for is "soft AP" in the chipset of the dongle you use.

TBD - WAP instructions Appendix

Video camera

Phoebe supports videoing her flights using a RaspiCam attached to her frame and to the Raspberry Pi. The video starts off just prior to takeoff and ends just after landing. There is no further control over the video required.

Chapter 2 - Software requirements

Angles

Knowing the angle a quadcopter is leaning is critical. There are 4 angles that need to be know

- pitch - the angle theta (θ) that Phoebe is leaning forwards along the X axis by rotating around the Y axis
- roll - the angle phi (φ) that Phoebe is leaning sideways along the Y axis by rotating around the X axis
- yaw - the angle psi (ψ) Phoebe has rotated around the Z axis
- tilt - tilt is not strictly necessary as it is just the combination of pitch and roll used to measure that angle the Z axis from vertical. However it's a useful term to combine pitch and roll together.

Imagine you wish for your quadcopter to hover above a fixed point on the ground. Remember I mentioned quadcopters are inherently stable. So your quad is bound to tilt over at an angle, whether it's due to a gust of wind, an imbalanced body, or a number of other factors.

When your quadcopter tilts, the power from the propellers is now slightly not vertical. This means the force it was using to fight gravity is now a lesser amount and so your quad starts to descend, slowly at first, but accelerating. At the same time, there is now a small portion of the overall propeller power accelerating your quad horizontally.

A bit of trigonometry says that if the quadcopter is tilting at θ degrees, then the vertical force is (gravity * cosine θ) and the horizontal force is (gravity * sine θ). We need to know θ so that we can take corrective action get the quad horizontal again. But neither of the sensors put out angles, so how are the angles derived from the information we have?

The gyroscope output is a rotation rate. If from the start of a flight, we integrate the rotation rate, then the angle rotated is known. But there are two flaws with this method.

- unless Phoebe takes off from a horizontal platform, the gyro will not know the original starting angle, so the values derived from it will always be off by the tilt of the take-off platform.
- Integration of the gyro angle is not possible - the code does not run infinitely fast; therefore we do the next best option which is to sum the (value from the gyroscope multiplied by the elapsed time since the previous sample was taken). This leads to very accurate readings short-term still (order of a few seconds) but the value drifts over the longer-term.

The accelerometer output is acceleration force measured in g's. When Phoebe is not accelerating, then the only force acting upon her is gravity. If she's horizontal, then the output from the X, Y and Z accelerometers is 0, 0, 1g respectively. But if she tilts, then the measurement is gravity which is always pointing vertically is distributed around Phoebe's accelerometer X, Y, and Z axes. By a simple calculation, it is possible to calculate the angles of tilt between Phoebe's accelerometer X, Y and Z axis, and vertical / horizontal. However there are flaws with this method also:

- The angles are only accurate if Phoebe is not accelerating in addition to just gravity. Most of the time, this is true, but changes in what she is doing, such as take-off to hovering need acceleration / deceleration.
- The accelerometer is plagued with noise, primarily from the motors / propellers.

Together this means the accelerometer angles cannot be used short term, but averaging over the longer term produces very accurate values. Also, when sitting stable on the ground with the motors disengaged, the accelerometer can produce the angle of the platform she's taking off from.

The two sets of angles complement each other perfectly. The accelerometer reads the take-off platform angle; the gyro reads short term changes in angle without influence from noise, and the longer term averaged accelerometer values provide long-term accuracy. What's known as a complementary filter is used to merge these two sources of angles, filtering in the best of both, and filtering out the flaws of both.

Motion Velocities

To control Phoebe, you need to know the speed it's moving in the direction of the 3 axes. For example, a stable hover is 0 speed in the X, Y and Z axes.

TBD

Motion Direction

TBD - Compass / Magnetometer + Barometer / altimeter + GPS.

Motion Rotation

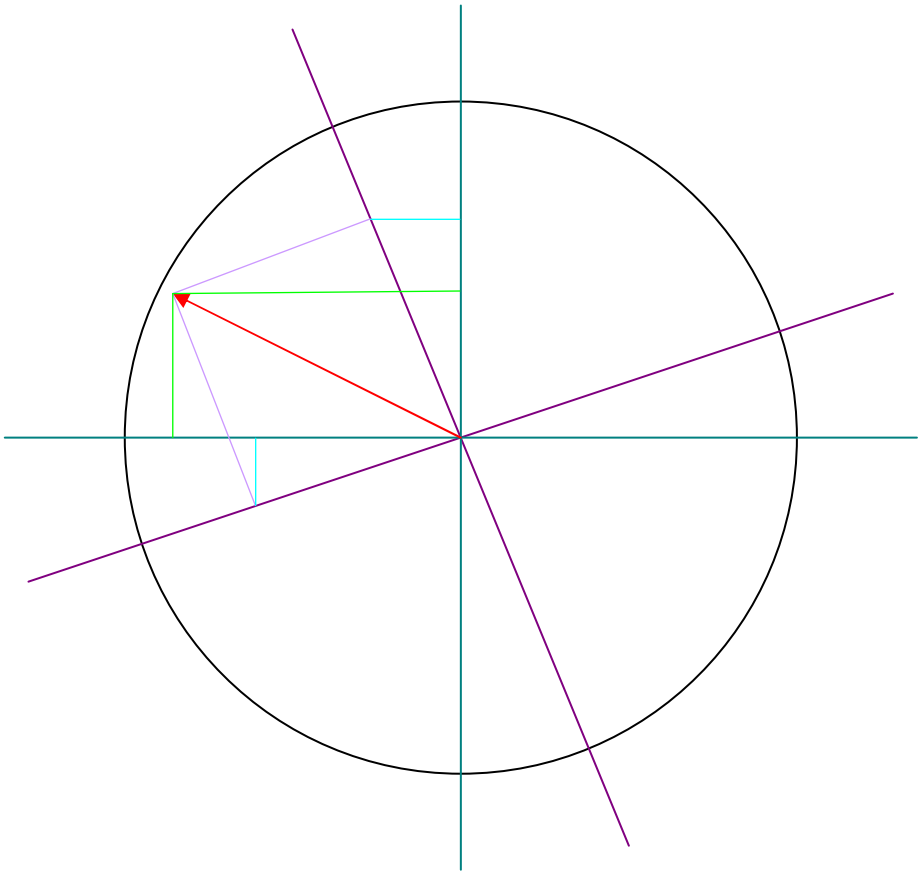
TBD -

Reference Frames / the Matrix

Assuming we have good quality sensor data, how do we use it? The bulk of that is covered later, but axes are critical to operation and it's worth covering them now.

From our point of view, it's easy to see which directions are horizontal (the horizon) and vertical (gravity), but a quad does not have eyes (though actually it has been done sort-of -- see blah). All it has is sensors, and the values these produce are not relative to the earth but to the quadcopter itself. They match the earth if the quad is hovering

stationary, but if the quad is at a tilt, the values from the sensors need convertion to earth coordinates.



PWM

Pulse Width Modulation takes several forms; in the context of driving a servo or brushless motors, it's normally a positive pulse between 1 and 2ms long. This pulse is sent to the ESC anywhere from 45 to several hundred times a second.

To achieve the necessary accuracy, normally microcontrollers are used. Luckily the Raspberry Pi provides the necessary PWM hardware, and is able to produce these pulses at up to 300Hz at a resolution of 1us - perfect.

Chapter 3 - PIDs

PIDs are the software glue that connects commands and sensors inputs together to control the power to the motors.

So far, we've not discussed error correction; for example, if you want your quad to take off vertically, surely you just power up all 4 motors at the same speed? Sadly not. None of the motors will be perfect matches, none of the propellers will be perfect matches, the frame won't be perfectly balanced, you have no control of the wind nor how horizontal the take-off platform is.

So sensor feedback, know as the 'input' to the PID is used to correct deviations from a desired 'target' setting. The 'target' is set by a user with a radio control transmitter via the joysticks, or can be hardcoded to follow a GPS defined route, or as a simple flight plan (take-off to 1m in 3s, hover for 10s, land in 3s). The different in 'target' - 'input' = 'error'. The code produces a corrective 'output' value which tries to keep 'error' minimized regardless to change in 'target'. To do so, the PID is run many times a second, constantly updating the corrective 'output'

The algorithm is called a PID because it produces 3 outputs

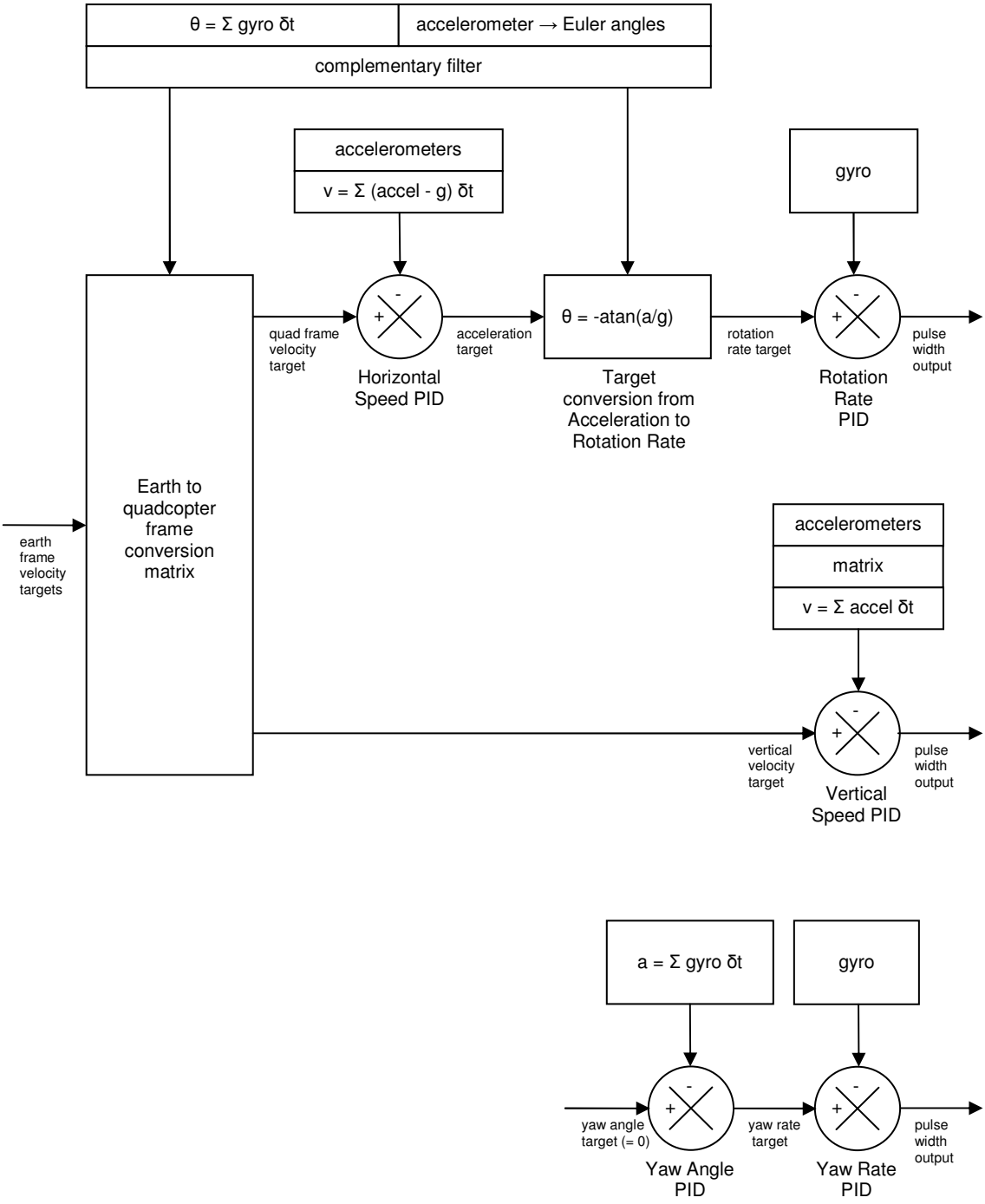
1. **P**roportional to the error
2. **I**ntegral of the error
3. **D**ifferential of the error

The PID algorithm is broadly used across many types of engineering projects; the use of feedback removes the need for a highly accurate mathematical model for the device being controlled.

Phoebe has 7 PIDs.

- 2 each for the X- and Y-axis horizontal speed control
- 1 for Z-axis vertical speed control.
- 2 for Z-axis yaw control

TBD - There are two extra PIDs required to provide the option of curves / circling. These are the lateral absolute angle PIDs required for centripetal force. The code exists in previous posts as the absolute angle PID. These work in parallel with the horizontal



There are 7 PIDs in operation.

- 2 each for the X- and Y-axis horizontal speed control
- 1 for Z-axis vertical speed control.
- 2 for Z-axis yaw control

Chapter 4 - Math(s)

Much of the math(s) is relatively simple if you have a reasonable level of trigonometry. The one piece that's critical, and that I simply didn't understand was the need to understand the reference frames and how to flip between them. Phoebe underwent to most significant recode I'd done at the point I finally understood the need.

Vectors

Vectors represent a value and a direction compared to a reference frame. Specifically to a quadcopter, they represent gravity as read by the accelerometer. So if your quadcopter is horizontal and not moving, then the vector is represented as 0, 0, 1 in the X, Y, and Z coordinates of the MPU6050 reference frame.

If you tilt your quadcopter, the portions of the force of gravity is detected by all three axes of the accelerometer.

Euler Angles

Given the vector of gravity distributed across the 3 accelerometer sensors, it is then possible to calculate the angles of tilt between the accelerometer axes (where the sensors are) and the earth axes (where gravity is) using the Euler angles calculation.

The Euler angles are used to

- calculate the angle of take-off
- convert between reference frames - see below

Frame Conversion Matrix

Once you know the angles between the quadcopter reference frame and the earth reference frame, you can convert values between the earth frame and the quadcopter frame using the following matrix. I won't (can't) explain the details here but instead I recomment you download and read section 2 of this document.

This matrix is used for two purposes in Phoebe's code:

1. convertion from horizontal and vertical earth frame targets to quadcopter frame targets
2. convertion from earth frame gravity to quadcopter frame gravity

$$
\begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{vmatrix}
\begin{vmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{vmatrix}
\begin{vmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{vmatrix}
$$

$$
\begin{vmatrix} qax \\ qay \\ qaz \end{vmatrix} = \begin{vmatrix} c\theta c\psi & c\theta s\psi & -s\theta \\ s\phi s\theta c\psi - c\phi s\psi & s\phi s\theta s\psi + c\phi c\psi & s\phi c\theta \\ c\phi s\theta c\psi + s\phi s\psi & c\phi s\theta s\psi - s\phi c\psi & c\phi c\theta \end{vmatrix} \begin{vmatrix} eax \\ eay \\ eaz \end{vmatrix}
$$

Appendix A - Parts List

Here's the list of components you'll need to build something similar to my quadcopter.

Hardware

DJI Flamewheel F450 ARF (almost ready to fly) kit
LiPo battery 11.1V 3S >3000mAh
LiPo balanced charger
T-motors 11 x 3.7 blades (optional upgrade)
T-motor 2216-11 motors (optional upgrade)

Breadboard

Power DC-DC converter
Cobbler
Sensors - alignment guaranteed by breadboard / breakouts for multiple sensors
Sounder
PWM out
Wires!

Software

Python
WAP
GitHub
RPIO
GPIO

Appendix B - Circuit Board Layout

TBD

Appendix C - Sensor Calibration

Because Phoebe is autonomous, the data read from the sensors must be as accurate as possible. For the gyro, this is easy, and the code performs the calibration automatically prior to every flight.

On the other hand, the accelerometer is a right PITA.

Appendix D - Tuning and Testing

TBD - for now see http://www.themagpi.com/issue/issue-20/

TESTCASE 1: Attach the propellers and make sure they rotate the right way

TESTCASE 2: Find approximate 0 vertical velocity speed = hover

TESTCASE 3: Stable hover from a horizontal platform with internal rotation rate PIDs

TEST FLIGHTS: Use the defaults (other than hover, vv* and rr* gains you've discovered in the previous tests) and see what happens. Good luck and enjoy! You're on your own from this point onwards.

Appendix E - Configuration, Diagnostics and Performance

Configuration

There are many command-line configuration parameters Phoebe provides to allow pre-flight changes without having to change the code between flights. The bare minimum is '-f' (flight mode).

-f - set set whether to fly')
-h - set the hover speed for manual testing')
-c - calibrate sensors against temperature and save')
-s - enable diagnostic statistics')
-n - disable drift control')
-v - video the flight')
-l ?? - set the processing loop frequency')
-a ?? - set attitude PID update frequency')
-m ?? - set motion PID update frequency')
--vvp - set vertical speed PID P gain')
--vvi - set vertical speed PID P gain')
--vvd - set vertical speed PID P gain')
--hvp - set horizontal speed PID P gain')
--hvi - set horizontal speed PID I gain')
--hvd - set horizontal speed PID D gain')
--rrp - set angular PID P gain')
--rri - set angular PID I gain')
--rri - set angular PID D gain')
--tc - select which testcase to run')
--dlpf - set the digital low pass filter')

Rather than explaining these here, I recommend reading the code to see exactly the effect of each - CheckCLI() is the function to start with.

Diagnostics

There comes a point when watching your quadcopter fly will not give you enough information to work out what happened when something goes wrong. To diagnose subtle problems, Phoebe can optionally provide megabytes of data per flight. This does slow her down by about 30% despite the fact the data is written to RAM and only copied to file at the end of a flight. Hence it's only recommended to run the code with diagnostics iff there's a problem.

The data is of the form of a CSV (comma separated variable) file which spreadsheets such as Microsoft Excel can parse. This then allows you to plot graphs of sensor data, PID variables and PWM outputs, allowing you to diagnose problems without reading the megabytes of data yourself.

Performance

There's not much you can do about performance - this section is purely for your information.

I've made Phoebe's code as efficient and fast as I can. With diagnostics disabled, the code loop at just over 200Hz (5ms per loop); with diagnostics it's about 130Hz. The PWM only needs updating at about 40Hz, so this allows averaging of sensor data, especially accelerometer data to remove noise.

I wouldn't recommend adding any significant new code unless you also consider the performance impact; I don't think Phoebe's code can be speeded up any more through tinkering but there is one other option worth considering if you need her to run a lot faster so you can fit in a lot of new code: Python is a completely interpreted language; the conversion to compiled code happens line by line as Python parses the script, and runs it via its interpreter.

There are speedier variants:

- Cython is precompiled Python scripts - your code will run at nearly the same speed as if you'd written it in C
- PyPy offers a middle ground of just in time (JIT) compiling, potentially increasing the speed of the code by a factor of 10.

Do bear in mind however that speeding up the code will generally NOT improve the performance of your quadcopter; the ESCs can only receive updates at about 400Hz; the sensors can only provide data at 1kHz. These are both hard-limits and more than fast enough to control a quadcopter - there's no way the motors can change the propeller speeds at this rate, and given the mass of the quadcopter, even if that was possible, the quad itself would not be able to react to the rapidly changing motor speeds. In fact, attempting to change the motors too quickly could make things worse by introducing more noise through jitter that the code is going to huge efforts to reduce.

But if there's other function you want to add to the code, then the use of Cython or PyPy will give you the space you need while maintaining the data feed to the motors at a sensible rate.

Alternatively, running the code faster allows longer term averaging between motor updates, thus reducing noise.

Appendix F - WiFi Access Point

TBD - For now see http://blog.pistuffing.co.uk/?p=594

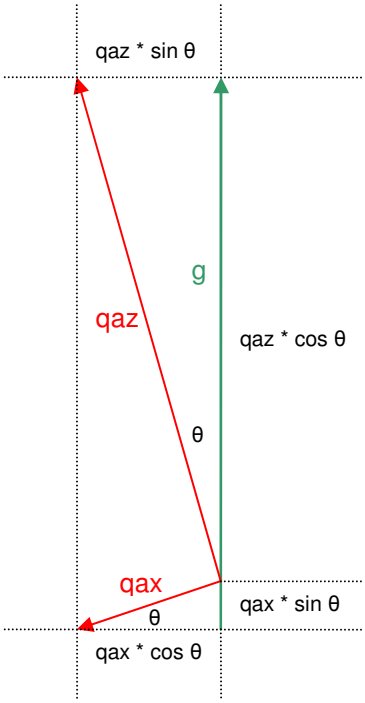
Appendix G - Code

I've left this until last as the code is "self documenting". There are at least as many lines of comments as there are code, describing what each small section of the code does. There is no point in repeating all that here.

The code is on GitHub at PiStuffing/Quadcopter, along with this document.

Although the code is "self documenting", it's worth giving you a quick overview to find your way around it.: in qc.py, after the imports at the top, the first few functions are low level for i2c, and the mpu6050 sensor; the next set are intermediate functions handling the ESCs and PIDs; the next set are helper functions for handling the command line, shutdown and other Phoebe function; finally right at the bottom is the main control loop which uses all the functions above to control Phoebe's flight.

If you want to read though the code, I'd recommend starting at "if keep_looping:" and work down from there.



Appendix H - Flight Plans

Currently Phoebe only support take-off, hover, and landing parts of a flight plan.

TBD

The next step is forward and lateral movement

The next step will be flying in circles

Finally, flight plans will be determined by GPS data points, and Phoebe will follow those pre-planned flight paths.