Unmanned Aerial Systems

Eugene Valetsky Supervisor: Dr Tim Baker

April 3, 2018

Contents

List of Figures

I, Eugene Valetsky, confirm that the work presented in this report is my own. Where information has been derived from other sources, I confirm that this has been indicated in the report.

1 Abstract

The goal of this project is to design and test the control system for a vehicle to compete in the Institution of Mechanical Engineers (*IMechE*) Unmanned Aerial Systems (*UAS*) Challenge. This vehicle must be capable of autonomous flight, which will include GPS waypoint navigation, location of a ground target, and the accurate delivery onto a payload onto said target.

The UAS design uses a novel concept: a traditional quadcopter built around a central micro jet engine. The control system will consist of a PixHawk flight controller running the PX4 flight stack, augmented by a companion computer running custom-made software to handle the duties of control, communication, and computer vision.

A simulation was built to test the feasability of the chosen configuration, using a simplified model. Results showed that while stability was significantly impacted over a pure quadcopter, especially in height, it was still within acceptable margins. It was also found that stability is significantly improved with more powerful servo motors for the jet gimbal.

The three main modules for the companion computer software were built and tested: control of the PixHawk, communication with a ground station, and target-finding computer vision. Python was chosen as a programming language due to its simplicity and available libraries.

PixHawk control was tested using a software-in-the-loop (*SITL*) PX4 simulation and jMAVSim. The DroneKit SDK was used as the basis for the module, but it had to be modified, as it was originally designed to work with APM, the predecessor of PX4. Commands for connecting to the PX4, taking off, navigating it both using GPS coordinates and in the local frame of reference, navigating it using both absolute and relative coordinates, and returning to base were all created and tested thoroughly within the simulation enviroment of jAVSim.

Computer vision is done using the OpenCV-Python library. The target is recognised by searching for concentric squares in the image. Optical Character Recognition (*OCR*) is possible using the Tesseract and pytesseract libraries.

Communication will be by radio, and so will be limited to a serial connection. Functions were created which allowed the transfer of crucial information, such as the coordinates of a located target, over said serial connection using the JSON format. The socat command line utility was used to establish two virtual serial ports to allow testing.

These disparate modules were brought together, and allow a simulated drone to navigate to a target, by using the offset of the target from the center of the frame as an input for a command navigating to a target relative to the current position. Although modifications will need to be made before deployment on the real vehicle, the basis for an autonomous flight control system has been proven.

2 Introduction

The Unmanned Aerial System (UAS) Challenge was launched by the Institution of Mechanical Engineers (IMechE) in 2014 'with the key objectives of developing professional engineers and inspiring the next generation'[1]. The main goal is to design and build a UAS in order to autonomously deliver humanitarian aid such as medical supplies in a disaster zone. This requires a broad range of skills, from project management to airframe design to avionics system programming. To add to the challenge there is a strict weight limit and stringent safety regulations. Working together with another student, Ismail Ahmad, our goal is to follow the design and build cycle all the way through to competing at the IMechE UAS competition in June 2018.

One of the greatest individual challenges as part of the project is creating an autonomous flight system. The UAS must be capable of quickly and safely navigating to and locating a target using computer vision, dropping a payload accurately, and returning to base. This requires an understanding of the UASs flight dynamics as well as good programming and wiring capabilities. This is the aspect that I will be focusing on for my individual project, although I will be assisting with many parts of the project.

3 Design

The concept for our UAS uses a hybrid symbioisis between a *Micro Jet Engine*, henceforth refered to as MJE, and an external multi-rotor. The MJE is gimballed to always remain vertical and only provides lift, while the multirotor rotates around it providing stabilisation and control. This configuration means that standard quadcopter flight control software can be used rather than needing to come up with custom architecture.

The MJE provides a higher thrust to weight ratio than the equivalent electric motors and batteries. It does this without introducing the vibration issues of a piston engine, which would seriously impact stability and control on such a lightweight design. With the current MJE and electric motors we hope to lift a payload of 3kg while remaining under the 6.9kg weight limit.

Flight control will be handled with a Pixhawk running the PX4 flight stack. This will be coupled to a companion computer running software based on the DroneKit SDK, communicating with the PixHawk using the MAVLink protocol. The companion computer is responsible for communication, waypoint navigation, and target location and tracking using computer vision. It then communicates where to go to the PixHawk.

4 Simulation: System Plausibility

4.1 Concept

One of the main concepts behind the UAS concept is the idea that, if the MJE is gimballed to remain vertical, and its thrust vector is through the center of gravity of the vehicle, it exerts no horizontal forces or moments on the rest of the vehicle. This means its effect on the stability and control of the vehicle can be ignored. In turn, this means that regular quadcopter flight control software can be used, such as the PX4 flight stack.

Being able to use PX4 firmware running on a Pixhawk is crucial. The UAS will come to cost approximately £2,500. Using home-made flight control software is therefore extremely risky. PX4, on the other hand, is a project that has been worked on by thousands of people for years, and is infinitely more reliable than anything we could put together from scratch. Additionally, creating custom flight control software is both discouraged by the IMechE, who recomend adapting an off-the-shelf autopilot [2], and falls well outside the scope of a university third-year project.

Therefore, it was decided to build a simulation to test the validity of the concept that the MJE can be ignored.

4.2 Equations of Motion

The vehicle is split into two sections, the outer quadcopter frame, *Quad*, and the gimballed section containing the MJE, *Jet*. A cartesian reference frame was chosen, with the xy plane being the horizontal plane and z being height. +x is right, +y is forward, +z is up. A rotation about the x axis is pitch, about the y axis is roll, and about the z axis is yaw. These are labelled θ_x, θ_y, and θ_z for the quad and ϕ_x, ϕ_y, and ϕ_z for the jet respectively.

The gimbal is controlled by two servos, one moving the gimbal in roll and one in pitch. (There is no need for yaw control of the jet.) Note that this means $\theta_z = \phi_z$.

The multirotor uses the 'quad-x' configuration, as shown in Figure 1. Note that motors 1 and 3 spin clockwise, and motors 2 and 4 spin anticlockwise. In the quad-x configuration, pairs of motors control each of the flight axes. For example, to pitch forward, motors 1 and 2 are spun down, and motors 3 and 4 are spun up; to yaw left, motors 2 and 4 would be spun up whilst motors 1 and 3 are spun down. The flight control system combines these various control movements, as well as ensuring that the net upward thrust is mantained. This can be seen in Equations 11 through 14.

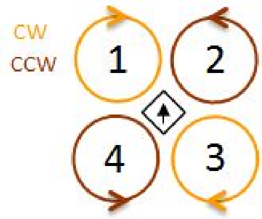


Figure 1: Quadcopter Configuration

4.2.1 Forces

Our gimbal design will transmit forces, but not moments. Thus, from a forces perspective, the vehicle is treated as a single unit.

The quad is modeled as an inner thin, hollow cylinder, the *frame*, with four cylindrical rods, the *arms*, extending outwards. The motors are point masses on the ends of the arms. It is assumed to be constructed of aircraft-grade aluminium. The total mass is therefore:

$$M_{TOT} = \underbrace{\rho\pi h(r_2 - r_1)}_{Frame} + 4 \cdot \underbrace{\rho\pi L r_r^2}_{Arms} + 4 \cdot \underbrace{M_M}_{Motors} + 2 \cdot \underbrace{M_G}_{Servos} + \underbrace{M_J}_{Jet} \tag{1}$$

The forces acting on the system are:

$$F_x = (F_{M1} + F_{M2} + F_{M3} + F_{M4}) \cdot \sin\theta_x \cdot \cos\theta_z + F_J \cdot \sin\phi_x \cdot \cos\phi_z \tag{2}$$

$$F_y = (F_{M1} + F_{M2} + F_{M3} + F_{M4}) \cdot \sin\theta_y \cdot \cos\theta_z + F_J \cdot \sin\phi_y \cdot \cos\phi_z \tag{3}$$

$$F_z = (F_{M1} + F_{M2} + F_{M3} + F_{M4}) \cdot \cos\theta_x \cdot \cos\theta_y + F_J \cdot \cos\phi_x \cdot \cos\phi_y \tag{4}$$

4.2.2 Quad Rotation

With the quad design as described in Section 4.2.1, the following inertias about the three defined axes are as follows:

$$\begin{aligned} I_{Qx} = I_{Qy} &= \underbrace{\frac{\pi\rho h_F}{12}(3(r_2^4 - r_1^4) + h_F^2(r_2^2 - r_1^2))}_{Frame} \\ &+ 4 \cdot \underbrace{\sin 45 \cdot (\frac{M_R L_R^2}{12} + M_R(r_2 + \frac{L_R}{2})^2)}_{Arms} \end{aligned}$$

$$
\begin{aligned}
&+ \quad 4 \cdot \underbrace{\sin 45 \cdot M_M (r_2 + L)^2}_{Motors} && (5)\\
I_{Qz} \quad &= \quad \underbrace{\frac{\pi \rho h_F}{2}(r_2^4 - r_1^4)}_{Frame}\\
&+ \quad 4 \cdot \underbrace{\frac{M_R L_R^2}{12} + M_R (r_2 + \frac{L_R}{2})^2}_{Rods}\\
&+ \quad 4 \cdot \underbrace{M_M (r_2 + L)^2}_{Motors} && (6)
\end{aligned}
$$

On the quad, the gimbal servos exert offset moments about the z-axis. This results in the following equations:

$$\tau_{Qx} = (-F_{M1} - F_{M2} + F_{M3} + F_{M4}) \cdot (L_R + r_2) \cdot \cos 45 \tag{7}$$

$$\tau_{Qy} = (F_{M1} - F_{M2} - F_{M3} + F_{M4}) \cdot (L_R + r_2) \cdot \cos 45 \tag{8}$$

$$\tau_{Qz} = (F_{M1} - F_{M2} + F_{M3} - F_{M4}) \cdot (L_R + r_2) \cdot \cos 45 + \tau_{Gx} \cdot r_1 + \tau_{Gy} \cdot r_1 \tag{9}$$

4.2.3 Jet Rotation

Since $\theta_z = \phi_z$, we are only interested in the rotation of the jet about the x and y axes. Since there is a rigid linkage between the servo and the gimbal, the position of the servo is proportional to the rotation of the jet.

The jet is modelled as a cylinder. However, it does not rotate about its origin in x and y, but about the appropriate servo. Including the inertia of the servos themselves (see Appendix C) results in the following inertias:

$$I_{Jx} = I_{Jy} = 3.89 \times 10^{-4} + M_J \cdot 3r_J^2 + h_J^2 + M_J l_c^2 \tag{10}$$

We know from Appendix C the maximum torque the servo can exert is 0.34 kg-m. The actual torque exerted is controlled by a PID controller in order to keep the jet vertical.

4.3 PID Control

8 PID controllers are needed in total: 3 position and 3 angle controllers for the quad, and 2 angle controllers for the jet gimbal. These controllers use the standard PID control logic:

$$
\begin{aligned}
error &= setpoint - actual\ value\\
integral &= integral + (error \times time\ period)\\
derivative &= (error - previous\ error)/time\\
output &= kP \times error + kI \times integral + kD \times derivative
\end{aligned}
$$

where kP, kI, and kD are constants to be optimised.

4.3.1 Quad Control

In a real quadcopter, the controller sends a signal to an *electronic speed control* (ESC), which in turn sends a *pulse width modulation* (PWM) signal to the motor, controlling its speed. In this model this has been simplified, such that the controller output directly controls the force exerted by the motors.

SP_x	Quad X position controller setpoint
SP_y	Quad Y position controller setpoint
SP_z	Quad Z position controller setpoint
SP_{θ_x}	Quad X angle controller setpoint
SP_{θ_y}	Quad Y angle controller setpoint
SP_{θ_z}	Quad Z angle controller setpoint
O_x	Quad X position controller output
O_y	Quad Y position controller output
O_z	Quad Z position controller output
O_{θ_x}	Quad X angle controller output
O_{θ_y}	Quad Y angle controller output
O_{θ_z}	Quad Z angle controller output

The controllers work in sequence, with the x and y position controllers determing the setpoint of the x and y angle controllers. The Z axis controller is independent.

$$O_x = SP_{\theta_x}$$
$$O_y = SP_{\theta_y}$$
$$F_{M1} = O_z - O_{\theta_x} + O_{\theta_y} + O_{\theta_z} \tag{11}$$
$$F_{M2} = O_z - O_{\theta_x} - O_{\theta_y} - O_{\theta_z} \tag{12}$$
$$F_{M3} = O_z + O_{\theta_x} - O_{\theta_y} + O_{\theta_z} \tag{13}$$
$$F_{M4} = O_z + O_{\theta_x} + O_{\theta_y} - O_{\theta_z} \tag{14}$$

4.3.2 Jet Gimbal Control

Servos are controlled by sending an electronic signal to tell them what position to rotate to.

SP_{ϕ_x}	Jet X angle controller setpoint
SP_{ϕ_y}	Jet Y angle controller setpoint
O_{ϕ_x}	Jet X angle controller output
O_{ϕ_y}	Jet Y angle controller output

$$\tau_{Jx} = O_{\phi x} \tag{15}$$
$$\tau_{Jy} = O_{\phi y} \tag{16}$$

4.4 Programming

The simulation itself was written in Processing. This is a language originally based on Java that is designed for ease of programming, especially with regards to displaying graphical elements. It was chosen over using MATLAB due to the author's increased familiarity with it, and the fact that this simulation has no need of advanced mathematical capability - there are no complex differential equations or matrix operations.

The program was built up in stages, with the physics of each stage checked before adding complexity. As far as possible, good object-oriented programming practice has been followed.

The quad section was implemented first. A controller was first tested solely in the z direction, and the response was used to calibrate PID values for the z controller. Angular controllers were then implemented, tested, and calibrated, before finally introducing x and y position controllers.

Testing involved flying a simple path: takeoff to 10m, a square path (10m north, 10m east, 10m south, 10m west), and then landing. This gave the results seen in Figure 2a. It can be seen that there is an oscilliation in x and y position. It proved difficult to eliminate, due to the not straightforward interaction of position and angle PID values. This, however proved not to be a problem. It can be seen in Figure 2b that the addition of the jet has, likely due to the added mass, damped out the oscillations in x and y position. It has also added an overshoot in z and a steady state error in x and y, but this can be corrected by fine-tuning of PID values. The flight time has also increased but this is to be expected with greater mass. In Figure 2c we can see how the quadcopter changes angle about the x axis, and how the jet initially starts to move in that direction but is quickly brought back to the vertical by the servo.

4.4.1 Programming Detail

Seen in Figure 3 is a partial UML[1]class diagram. This provides an overview of the various classes and objects used in the simulation and their relationships to each other.

It can be seen that there exists a PhysicalObject class, from which several other classes inherit, which contains atributes which cover information such as mass, position, velocity, and inertia. Quad contains QuadFrame, which itself contains 4 Motors, as well as Jet and 2 GimbalServos. QuadFrame, Jet, and GimbalServo are all interconnected, ensuring that changes to one will be accurately and correctly reflected in the other as required.

PID_Values is a class which describes one set of PID values and the relevant PID calculations, and is used by the GimbalServos and the Controller. As described in Section 4.3, standard PID control logic is used, with one addition: integral wind-up has been added. This prevents the integral term from growing disproportionately large, which otherwise occurs in this implementation as our feedback loop occurs several orders of magnitude faster than changes in the output. The code for this is:

```
float calculate(float time, float actual) {
    float error = setPoint - actual;
    integral += (error*time);
    if (integral > integralWindUp) {
        integral = 0;
    }
    float derivative = (error - prevError)/time;
    prevError = error;
    return (kP*error + kI*integral + kD*derivative);
}
```

Many classes contain an update() method. This takes the amount of time that has passed since the last update as an input, and uses this, current values of velocities and accelerations, as well as exerted forces and moments, to calculate the resultant movement of the object, as well as recording the result, by itself calling methods such as calculateMovement() and calculateRotation().

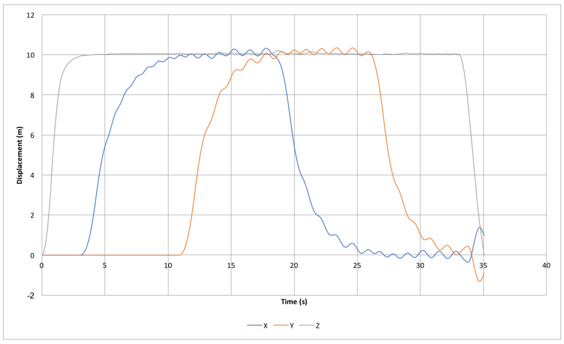
It should be noted that a minimal effort has been made to make use of object-oriented programming concepts such as encapsulation. The focus of programming this simulation was to produce usable results, not to produce clean, re-usable code.

The full code can be seen in Appendix

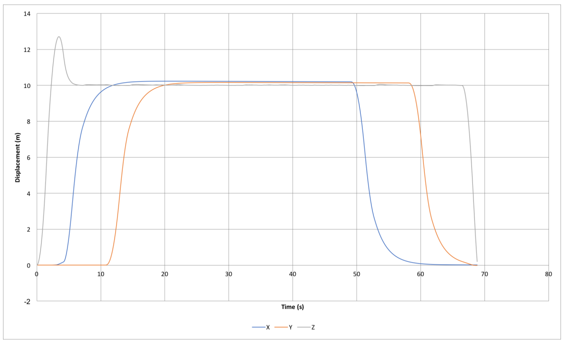
4.5 Refiment

It was realised that the mass of the quad had been implemented incorrectly (r_r had not been squared and the rod mass and motor mass had not been multiplied by 4 in equation 1).

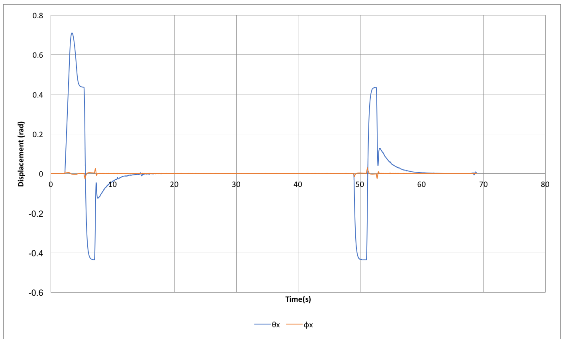
[1]Unified Modeling Language (UML) is a visual modelling language used to describe processes and structures, particularly in software modelling. More information is widely available online.



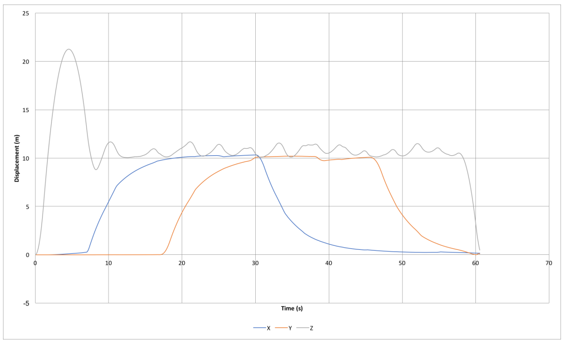
(a) Quad Only



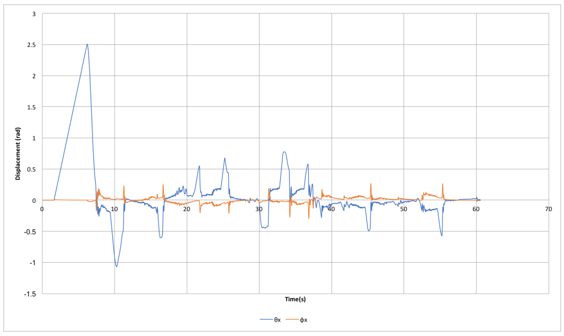
(b) With Jet



(c) With Jet



(d) With Refiments



(e) With Refiments

Figure 2: Simulation of a Square Path

Figure 3: XQuad2 UML Class Diagram

Additionally, the maximum torque of the servo had not been limited to 0.34kgm. Correcting these resulted in Figures 2d and 2e.

These refiments decreased the stability of the drone. It is still well within acceptable parameters for rotation, and for x and y positioning, but height now has considerable steady state error and oscillation.

With sufficient PID tuning, height could be stabilised, but not without a massive initial overshoot, and slightly increasing instability in x and y. This is because as the proportion of motor control that comes from height control increases (due to increasing k_p and k_i terms), the proportion of motor control left available for the other controllers decreases.

4.6 Conclusion

The reason for the overshoot in height appears to be that the jet adds mass to the system, without adding weight (as this is cancelled out by its thrust). This is not something that a controller would expect. However, we are not greatly concerened with the accuracy of the height-keeping of the system. As long as we stay above 20ft, the minimum height required by the regulations[2], it is not a concern.

It can be seen that the addition of a gimballed jet to a quadcopter does not catastrophically destabilise it. The extremely simple PID controllers used in our simulation are able to cope to a sufficient degree, implying that a significantly more advanced flight stack such as the PX4 should have no problems. Our initial assumption that the jet can simply be ignored is, indeed, valid.

5 Companion Computer Software

5.1 PX4 Control

We will be controlling the PixHawk flight controller using offboard commands from an external companion computer, henceforth refered to as Pete (since CC is a rather ambigous acronym). Pete will consist of three main components: communication with the PixHawk, communication with a ground station, and target-finding computer vision.

PixHawk 2 is a fully integrated flight controller running the PX4 flight stack. PX4 is an open source project, and is a further development of the popular ArduPilot flight control software. It is an excellent choice for this project as it is easily adaptable to different vehicles, and is capable of calculating many flight paramters by itself and continously optimise them. This minimises the amount of work that will be required to adapt it to our custom-built airframe.

PX4 is fully capable of Software-in-the-Loop (SITL) simulation, using either the jMAVSim or Gazebo simulation enviroments. This is crucial, as it allows code to be tested with no danger to the real vehicle. Pete can commicate with the simulated vehicle using MAVLink through a UDP port; this is exactly the same as how it would connect to a real vehicle. As far as Pete's code is concerned, there is no difference between a real and a simulated vehicle except for the UDP address.

Communication with the PX4 flight stack on the PixHawk is based on the DroneKit SDK. This is a platform for devoloping apps written in Python that run on a drone's companion computer. It communicates with the PixHawk using the MAVLink communication protocol.[5] The DroneCore API was also experimented with but proved significantly harder to implement and hence was abandoned.

PX4 has 12 flight modes when employed on a multirotor. We are interested in the autonomous modes, which include Hold, Return(RTL), Takeoff, Land, Mission, Follow Me, and Offboard. Of particular interest are Mission, in which ‘the vehicle follows a programmed mission’, and Offboard, where ‘the vehicle obeys a position, velocity or attitude setpoint provided over MAVLink’.[3]

The DroneKit SDK was used to create code that will run on Pete and communicate with PX4. This is an open source, freely available software development kit, specifically designed for offboard control of vehicles using companion computers to enable autonomous behaviour. It is available in three main implementation: on Android, on iOS, and in Python. It is the Python version we will be using here, as almost all available comapnion computers come with Python preinstalled and with a high level of support for it.

During research into the subject area, two alternative libraries were found that could have been used: MAVROS and DroneCore. These were rejected as they are both written in C++. Although code written in C++ is often faster, it also typically takes longer to write as it is a lower-level language with more need for ‘boilerplate’ code. We did not anticipate our control system to be advanced enough for the speed of code execution to make a significant difference, and developing a control system quicker was the larger priority. Additionally, much of the difference can be made up by using C or C++ libraries with Python implementations, as OpenCV, used later in this report, does.

The available documentation and examples for DroneKit mostly make use of the `simple_goto()` command. Unfortunately, as DroneKit is primarily designed to work with ArduPilot, the predecessor of PX4, this command was discovered to not work when tested in jMAVSim. Neither does `arm_and_takeoff()`, or the `vehicle.groundspeed` and `vehicle.airspeed` attributes, among others.

Additionally, it is usually assumed that one is operating in the `global_relative_frame`. There are three frames of reference available:

`global_frame`	GPS coordinates, with 0 altitude at sea level
`global_relative_frame`	GPS coordinates, with 0 altidue as ground level at the starting location
`local_frame`	Cartesian coordinates relative to the starting location

`local_frame` is the most immediately useful to us, as it is simpler and more intuitive to use. Note that this is the North East Down (NED) frame, i.e. -10m altitude is 10m above the ground.

Custom code has had to be written to take the place of these defunct commands and allow the vehicle to be controlled in the desired manner. First of all, we want to be able to directly send a position waypoint to PX4. This is a feature that is not fully implemented in DroneKit, and so a custom command with a custom MAVLink message has been created:

```
def send_ned_position(pos_x, pos_y, pos_z):
    """
    Move vehicle in direction based on specified velocity vectors.
    """

    msg = vehicle.message_factory.set_position_target_local_ned_encode(
        0,       # time_boot_ms (not used)
        0, 0,    # target system, target component
        mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
        0b0000111111111000, # type_mask
        pos_x, pos_y, pos_z, # x, y, z positions
        0, 0, 0, # x, y, z velocity in m/s
        0, 0, 0, # x, y, z acceleration (not supported yet)
        0, 0)    # yaw, yaw_rate (not supported yet)

    vehicle.send_mavlink(msg)
```

Similarly, other commands have been created to suit our purposes. Some of these have been adapted from examples in the DroneKit and PX4 documentation.[5][4]

```python
def arm_and_takeoff(targetAlt, accuracy=0.5):
    wp = get_location_offset_meters(home, 0, 0, targetAlt)
    cmds.add(PX4Command(wp, "TO"))
    cmds.upload()
    time.sleep(1)

    vehicle.mode = VehicleMode("MISSION")
    time.sleep(1)
    print("Vehicle mode should be MISSION: %s" % vehicle.mode.name)
    vehicle.armed = True
    while True:
        print " Altitude: ", vehicle.location.global_relative_frame.alt
        #Break and return from function just below target altitude.
        if vehicle.location.global_relative_frame.alt>=targetAlt-accuracy:
            print "Reached target altitude"
            break
        time.sleep(1)

def goto_absolute(pos_x, pos_y, pos_z, accuracy=0.5):
# Go to a position relative to the home position

    targetLocation = LocationLocal(pos_x, pos_y, -pos_z)

    send_ned_position(pos_x, pos_y, -pos_z)
    vehicle.mode = VehicleMode("OFFBOARD")
    print("Vehicle mode should be OFFBOARD: %s" % vehicle.mode.name)

    while True:
        send_ned_position(pos_x, pos_y, -pos_z)
        remainingDistance = get_distance_metres_local(vehicle.location.local_frame, targetLocation)
        if remainingDistance<=accuracy:
            print("Arrived at target")
            break
        print "Distance to target: ", remainingDistance
        time.sleep(0.1)

def goto_relative(pos_x, pos_y, pos_z, accuracy=0.5):
# Go to a position relative to the current posotion

    currentLocation = vehicle.location.local_frame
    targetLocation = get_location_metres_local(currentLocation, pos_x, pos_y, -pos_z)\

    send_ned_position(targetLocation.north, targetLocation.east, targetLocation.down)
    vehicle.mode = VehicleMode("OFFBOARD")
    print("Vehicle mode should be OFFBOARD: %s" % vehicle.mode.name)

    while True:
        send_ned_position(targetLocation.north, targetLocation.east, targetLocation.down)
        remainingDistance = get_distance_metres_local(vehicle.location.local_frame, targetLocation)
        if remainingDistance<=accuracy:
            print("Arrived at target")
            break
        print "Distance to target: ", remainingDistance
```

```
        time.sleep(0.1)

def setMaxXYSpeed(speed):
    vehicle.parameters['MPC_XY_VEL_MAX']=speed
    print("Set max speed to: %s" % vehicle.parameters['MPC_XY_VEL_MAX'])
    time.sleep(0.5)

def returnToLand():
    vehicle.mode = VehicleMode("RTL")
    time.sleep(1)
    print("Vehicle mode should be RTL: %s" % vehicle.mode.name)
    while vehicle.armed == True:
        print("Waiting for landing...")
        time.sleep(3)
```

arm_and_takeoff() replaces the command provided in DroneKit, and performs the same function. goto_absolute() and goto_relative () allow us to navigate to a position waypoint, simply defined as X meters north, Y meters east, and Z meters up from either the starting location or the current location. The accuracy attribute optionally allows us to change how close the vehicle must get to the waypoint to consider it to have reached it. setMaxXYSpeed() is a replacement for the vehicle.groundspeed attribute present in DroneKit, which does not work as intended with PX4, and allows us to set a maximum groundspeed for the vehicle. Finally, returnToLand() has the vehicle return to it's starting location and land safely.

Altogether, this means that the communication with the PixHawk has been simplified down to a few basic commands. An example mission could consist of the following: arming and taking off, following a series of GPS coordinates, locating a target using computer vision, calculating it's offset from directly underneath the drone, using goto_relative () to position itself precisely over the target, droping the payload, and returning to base and landing.

5.2 Computer Vision

The next component of the control system is a computer vision system capable of identifying the target. This takes the form of a red 1x1m square, incoporating an alphanumeric code in white, within a larger 2x2m white square[2]. The system would have to locate the target, calculate the offset from being centered underneath the vehicle, and to indentify the alphanumeric code.

Since an interface had already been programmed in Python, it was decided to program the computer vision in Python as well to allow for them to easily work in conjunction. To this end the OpenCV-Python library was chosen.

5.2.1 Square Detection

The target consists of two concentric squares and a central leter, as seen on Figure 4[2]. Therefore, a good way of identifying the target is looking for concentric squares. This is called feature detection. The other main method is known as creating a cascade, and allows objects whose features cannot be easily described mathematically, such as a chair, to be recognised. However, creating a cascade requires the processing of thousands of images and considerable time. As we are fortunate enough to have a feature which can be described easily mathematically (concentric squares), feature detection is the way to go.

First, squares are recognised. This is a multi-stage process:

1. Convert image to grayscale
2. Use a Gaussian Blur on the image

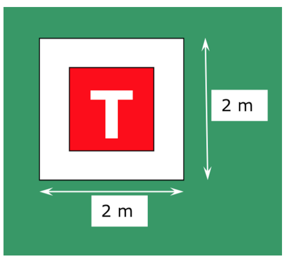


Figure 4: Target

3. Use Canny Edge Detection to find edges
4. Find complete contours
5. Find contours that have between 4 and 6 edges
    - An ideal square would have 4 edges, but images are rarely perfect.
6. Check that the contour is above a minimum size
7. Check that the solidity of the contour is above a threshold
    - This is done by drawing a rectangle (the *bounding rectangle*) that encloses the entire contour, and comparing the area of the rectangle to the area of the contour
    - An ideal square would have a solidity of 1
8. Check that the aspect ratio of the bounding rectangle is within thresholds
9. Contours that have passed all of these criteria must be squares

These contours are then drawn onto the image, showing us the squares. This method can be applied to images, or to video by applying it to each frame in the video. It is modified from examples available online.[6][7]

5.2.2 Optical Character Recognition

Reading the letter inside the target is also a requirement of our system. To this end, Optical Character Recognition (OCR) was added to our system. This is done using the freely available Tesseract OCR library, in conjuction with the pytesseract library that adapts it to Python. A simple function was created that takes an image and puts it through Tesseract, printing all that is written on that page. Once again, it was adapted from examples available online.[8]

It should be noted that the function which performs this OCR was discovered to be extremely inefficient, taking about 0.3 seconds to execute, due to the way that the Tesseract library is accessed. This would decrease the frame rate below usable levels if run routinely, and so will only be run a single time once the target has been positively identified.

5.2.3 Target Detection

The next step was to find the target by finding squares that are concentric to each other. This was done by finding the coordinates of the center of the square, using OpenCV's `moments()` function. This center was then compared to the centers of all other squares that had been found.

At this point, a `Square` class was made to improve the code under the concepts of object-oriented programming. A `concentric()` method was made in this class that compares squares against each other to see if their center points are within a certain distance of each other (by default, 5% of the size of the square). Two squares that are concentric are added to a new intance of a `Target` class. Repeated comparisons are avoided by using `intertools.combinations()` to make a list of all the unique pairs of squares before comparing.

Initially, it appeared that all squares were targets. The problem was discovered to be that there were in fact multiple, very similar squares being generated for each actual square on the image. For example, for a square whose border is a black line might have the outside of the line and the inside of the line counted as seperate squares. These squares are clearly concentric and thus all squares are targets. The problem was solved by creating a similar() method for Square and using it to eliminte squares that are too similar (again, 5% by default), ensuring that all squares are unique before running the concentric() method to find targets.

From here, it is trivial to calculate the offset of the target from the center of the image or camera frame. This offset will be used as an input into a control system to move the drone above the target.

5.2.4 Integrating into Control

Using the code developed in Section 5.1, a program was created to integrate the newly-developed computer vision. This takes off to 10m, and once there the webcam on the computer is turned on. It looks for a target, and uses the offset of the target from the center of the camera as an input for the desired position of the drone. This was verified to work as intended, validating all the work we have done up to this point.

A demonstration video of this was created. For obvious reasons, it could not be put in this report, but is available on request.

5.2.5 Investigating Frame Rate

The method of finding all squares, checking through them all for duplicates, then checking through them all for targets, intuitively seems inneficient. This could especially become an issue when deployed to a drone platform, which will likely have less processing power.

However, timing sections of the program using the timeit library revealed that the vast majority of the time that went into processing a frame went into receiving a frame from the camera, with the entire process of target detection taking approximately 10% of the required time per frame. This is because I/O (Input/Output) operations are very expensive in terms of computing time. Meanwhile all the square comparisons are simple number comparisons, which are cheap.

Thus, it was decided there was no need to optimise the target-finding algorithm, as it would have a minimal effect on frame rate. Any appreciable frame rate improvement will have to come from improving the speed of obtaining a frame from the camera.

5.3 Communication

Communication from Pete to a base station to transmit information will, by necessity, be over a serial connection.

The socat command line utility was used to establish two virtual serial ports which communicate with each other. These are then acessed using the pyserial library to write and read information from these serial ports.

5.3.1 Serialising Targets

It was quickly discovered that sending an entire image over a serial connection within a reasonable amount of time is not possible. Even sending the contours of a square does not work well, as this is a large set of points. Thus, it was decided to send only the most important information.

A Square2 and Target2 class were created. These inherit from Square and Target. The only difference is that they are missing the contours of the squares. Square and Target gained a to_json() method that converts their attributes to a string using JavaScript Object Notation

(JSON), a standard for transmitting information. Square2 and Target2 have a function from_json () which can recreate the squares from the JSON string.

This was tested by sending the targets over the virtual serial connection, and drawing them on the image. With minor modifications to the way that squares and targets are drawn, this worked sucessfully. Notably, squares are now defined largely based on their bounding rectangle and not on their own contours, which for a square or rectangle makes little difference.

5.4 Choosing a Computer

By this point, Pete has advanced about as far as is possible in a purely simulated enviroment. Further development required moving onto a real companion computer, and testing either with a real drone or a Hardware-in-the-Loop (HITL) simulation.

A comparison was made of availible companion computers on the market. The Intel Edison, Raspberry Pi3, Nvidia TX2, Odroid XU4, and Snapdragon Flight were compared on criteria such as processing power, weight, and cost. The full comparison can be seen in Appendix 5.

Specification	Raspberry Pi3	Nvidia TX2	Odroid XU4	Snapdragon Flight	Intel Edison
Linux Kernel Version	4.9	4.4	4.14	3.4	3.8
CPU Power	1.2 GHz x4	2 GHz x6	2 GHz x4	2.26 GHz x4	500 MHz x2
GPU Power	250 MHz	1300 MHz x256	~ 600 MHz	578 MHz	None
Memory	1 GB	8 GB	2 GB	2 GB	1 GB
Storage	MicroSD Card	32 GB	MicroSD Card	32 GB + microSD Card	4 GB
Voltage	5V	5.5V minimum	5V	5V	3.15-4.5V
Power Consumption	~ 3 W	7.5-15W	3-12W	3-4W	0.5W
Weight	45g	85g	48g	???	Low
Size	85.6x56.5	87x50mm	83x58mm	58x40mm	25x35.5mm
Wireless	Yes	Yes	No	Yes	Yes
Bluetooth	Yes	Yes	No	Yes	Yes
Ethernet	Yes	Yes	Yes	No	No
USB Ports	4	2	3	1	1
Other Ports	HDMI, MIPI, 3.5mm audio	HDMI, MIPI, CAN	HDMI	2xMIMO	None
Others Features				Inbuilt Cameras, LTE	
Issues			Ports are 1.8V, not the standard 3.3 or 5V		
Needs PixHawk?	Yes	Yes	Yes	No	Yes
Price	£30 + PixHawk + Periphals	£486	£75	$675	No longer for sale
Comments	Most support and documentation available, borderline as to whether its	Possibly overkill, but will definitely get the job done	Solid choice on paper but little community support available	Apparently a pain	Not powerful enough for computer vision
Choice	1=	1=	3	4	5

Figure 5: Companion Computer Selection

Of these, the Raspberry Pi3 was the preferred choice, as it is cheap and light, and has the greatest amount of community support and available documentation. It is, however, borderline as to whether it is powerful to run the required computer vision computation. The Nvidia TX2 is a close second. It is definetely powerful enough to meet our needs, but is significantly more expensive and has less support available. It was therefore decided to buy a Raspberry Pi, and to upgrade to the TX2 if it turns out to not be powerful enough.

6 Companion Computer Setup & Testing

A Raspberry Pi3 was obtained, and Raspbian was installed on it. It was discovered that the 8GB microSD card that the Pi3 comes with is not large enough for our purposes. The amount of free space after the operating system and default programs are installed is approximately

1.6GB, and OpenCV requires around 4GB free space to compile from source. Therefore, a 32GB microSD card was obtained and the setup process was restarted.

A Table of All Symbol Definitions

Note: values are only given for variables which are constants. No values are given for values derived from contants or for dynamic variables.

Symbol	*Definition*	*Value(where appropriate)*
	Positions and Angles	
θ_x	Quad Pitch	
θ_y	Quad Roll	
θ_z	Quad Yaw	
ϕ_x	Jet Pitch	
ϕ_y	Jet Roll	
ϕ_z	Jet Yaw	
	Dimensions	
h_f	Quad frame height	5 mm
r_1	Quad frame inner radius	180 mm
r_2	Quad frame outer radius	200 mm
L_R	Quad arm length	100 mm
r_r	Quad arm radius	2 mm
r_J	Jet radius	41 mm
h_J	Jet height	150 mm
l_c	Gimbal servo connecting rod length	100 mm
	Masses	
M_{TOT}	Total Mass	
ρ	Density of Quad	$2700kgm^3$
M_F	Quad frame mass	
M_R	Quad rod mass	
M_M	Motor mass	50 g
M_G	Gimbal servo mass	79 g
M_J	Jet mass	
	Inertias	
I_{Qx}	Quad inertia about x axis	
I_{Qy}	Quad inertia about y axis	
I_{Qz}	Quad inertia about z axis	
I_{Jx}	Jet inertia about x axis	
I_{Jy}	Jet inertia about y axis	
I_{Jz}	Jet inertia about z axis	
	Forces	
F_x	Sum of forces in x direction	
F_y	Sum of forces in y direction	
F_z	Sum of forces in z direction	
F_{M1}	Force from motor 1	
F_{M2}	Force from motor 2	
F_{M3}	Force from motor 3	
F_{M4}	Force from motor 4	
F_J	Force from jet	
	Torques and Moments	
τ_{Qx}	Sum of moments on quad about x axis	
τ_{Qy}	Sum of moments on quad about y axis	
τ_{Qz}	Sum of moments on quad about z axis	

Symbol	*Definition*	*Value(where appropriate)*
τ_{Gx}	Torque of jet gimbal servo about x axis	
τ_{Gy}	Torque of jet gimbal servo about y axis	
	PID Controllers	
SP_x	Quad X position controller setpoint	
SP_y	Quad Y position controller setpoint	
SP_z	Quad Z position controller setpoint	
SP_{θ_x}	Quad X angle controller setpoint	
SP_{θ_y}	Quad Y angle controller setpoint	
SP_{θ_z}	Quad Z angle controller setpoint	
O_x	Quad X position controller output	
O_y	Quad Y position controller output	
O_z	Quad Z position controller output	
O_{θ_x}	Quad X angle controller output	
O_{θ_y}	Quad Y angle controller output	
O_{θ_z}	Quad Z angle controller output	
SP_{ϕ_x}	Jet X angle controller setpoint	
SP_{ϕ_y}	Jet Y angle controller setpoint	
O_{ϕ_x}	Jet X angle controller output	
O_{ϕ_y}	Jet Y angle controller output	

B Table of Abbreviations

In order of appearance:

Abbreviation	*Definition*
IMechE	Institution of Mechanical Engineers
UAS	Unmanned Aerial Systems
GPS	Global Positioning System
SITL	Software In The Loop
SDK	Software Development Kit
APM	ArduPilot Mega
OCR	Optical Character Recognition
JSON	JavaScript Object Notation
MJE	Micro Jet Engine
PID	Proportional Integral Derivative
ESC	Electronic Speed Control
PWM	Pulse Width Modulation
UML	Unified Modelling Language
UDP	User Datagram Protocol
API	Application Programming Interface
RTL	Return To Land
NED	North East Down
I/O	Input/Output
HITL	Hardware In The Loop

C Servo Information

Servo information was based on a sample servo that may well end up being used on the vehicle, the Futaba BLS177SV.

Torque (at 6.6V):	34 kg-cm
Speed (at 6.6V):	0.12sec/60°
Weight:	79g

We can see from the datasheet that it takes the servo 0.12 seconds to rotate 60°. Assuming it takes 0.01 of those seconds to accelerate to full speed, this gives an acceleration of $873 rad/s^2$. Since we know it exerts a torque of 0.34kgm, this means its inertia must be 3.89×10^{-4}.

References

[1] IMechE. About UAS Challenge - IMechE [Internet]. [Accessed 20 December 2017]; Availible from: https://www.imeche.org/events/challenges/uas-challenge/about-uas-challenge.

[2] Institution of Mechanical Engineers. University UAS Challenge 2018 Competition Rules [Internet]. Issue 7.1. [Accessed 27 December 2017]; Available from: https://www.imeche.org/docs/default-source/1-oscar/uas-challenge/uas-challenge--competition-rules.pdf?sfvrsn=2.

[3] DroneCode. PX4 Autopilot User Guide [Internet]. Updated 20 December 2017. [Last accessed 20 December 2017]; Availible from: https://docs.px4.io/en/.

[4] DroneCode. PX4 Development Guide [Internet]. Updated 15 December 2017. [Last accessed 20 December 2017]; Availible from: https://dev.px4.io/en/.

[5] 3D Robotics. DroneKit-Python Documentation [Internet]. Updated 21 April 2017. [Accessed 20 December 2017]; Available from: http://python.dronekit.io.

[6] OpenCV. OpenCV-Python Tutorials OpenCV 3.0.0-dev documentation [Internet]. Last updated 10 Novemeber 2014. [Accessed 22 December 2017]; Available from: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html.

[7] Adrian Rosebrock. Target acquired: Finding targets in drone and quadcopter video streams using Python and OpenCV [Internet]. 4 May 2015. [Accessed 22 Decemeber 2017]; Available from: https://www.pyimagesearch.com/2015/05/04/target-acquired-finding-targets-in-drone-and-quadcopter-video-streams-using-python-and-opencv/.

[8] Adrian Rosebrock. Using Tesseract OCR with Python [Internet]. 10 July 2017. [Accessed 22 December 2017]; Available from: https://www.pyimagesearch.com/2017/07/10/using-tesseract-ocr-python/.