Unmanned Aerial Systems

Eugene Valetsky   Supervisor: Dr Tim Baker

April 22, 2018

GitHub Repository: `https://github.com/eugeneval/Project_Titan`

Word Count: 7122

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.

Contents

List of Figures

1 Abstract

The goal of this project is to design and test, in a simulated environment, 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[2].

Design Concept Validation

The UAS design uses a novel concept: a traditional quadcopter built around a central micro jet engine[12]. Equations of motion were derived from first principles for a simplified model of the airframe, and a computer simulation was built in Processing[13] to test the feasibility of the chosen configuration, especially with regards to the impact that the jet engine would have on stability. Results showed that while stability was significantly impacted over a pure quadcopter, especially in height, it was still within acceptable margins.

Control System Design

A control system was designed consisting of a PixHawk flight controller running the PX4 software[3], augmented by a companion computer running custom-made software to handle the duties of control, communication, and computer vision. Python[14] was chosen as a programming language due to its simplicity, abundance of available relevant libraries, and good cross-platform compatibility.

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. These were tested using a software-in-the-loop (*SITL*) PX4 simulation and the jMAVSim simulation environment. The DroneKit SDK was used as the basis for control of the PixHawk, but it had to be modified, as it was originally designed to work with APM, the predecessor of PX4[5]. 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 environment 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 and tested which allowed the transfer of crucial information, such as the coordinates of a located target, over said serial connection using the JSON format.

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. A specific companion computer was chosen, the Raspberry Pi3. The control system was ported to the companion computer, and modifications were made to account for the details of running on specific hardware.

Conclusion

The airframe concept has been validated in a simulated environment and stability has been shown to be within acceptable margins. Control software has been designed, created, and tested in a simulated environment. The control system is ready for deployment to the real vehicle as and when it is complete.

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. I will also be assisting with other parts of the project, but this is outside the scope of this report.

2.1 Airframe Design

The concept for our UAS uses a hybrid symbiosis between a *Micro Jet Engine*, henceforth referred to as MJE, and an external multi-rotor. The MJE is gimballed to always remain vertical and only provides lift, while the multi-rotor 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 imposed by the rules[2]. MJE's are also capable of running on a wide variety of fuels, which would be an advantage in a disaster zone where fuel and electricity supplies may be limited or unavailable[12].

The chosen airframe configuration is a novel airframe that, to the extent of research done into the subject, has only been tried by several hobbyists and never as a serious technical endeavour. We were to discover only quite late in the design process (see Section 7.1) a possible reason as to why - the central MJE greatly complicates the carriage of payload.

Further detail as to the physical design of the airframe is outside the scope of this project, as this is Ismail Ahmad's responsibility[12]. A render of his design for the airframe can be seen in Figure 1[12].

(a) Bare

(b) With Bodyshell
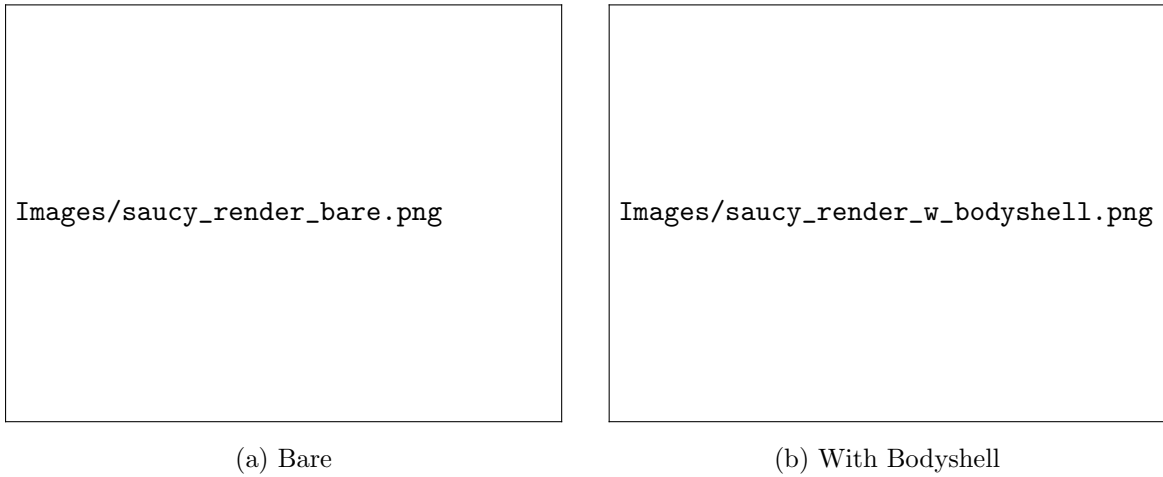
Figure 1: Renders of Airframe Design

2.2 Aim and Objectives

Ultimately, the long term goal of this project is to take part in the IMechE UAS Challenge. However, within the extent of the third-year project, the objectives are as follows:

1. Determine, via simulation or otherwise, whether the stability of the airframe configuration is within acceptable parameters.

2. Investigate and design a control system for the UAS. This control system must be able to:

    (a) Travel to predetermined GPS waypoints.

    (b) Recognise and navigate to a target using computer vision.

    (c) Communicate the GPS coordinate of the target to a ground station.

3. Test the control system in a simulated environment.

As this is an open-ended design project, the exact details of the final solution cannot

be determined at this stage. As such, these aims and objectives have been expanded on in more detail in the relevant sections after initial research and design has been undertaken.

2.3 Control System Design

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.

This setup of two computers working in tandem, a low-level flight controller and a high-level companion computer, is the currently widely accepted method for adding complex autonomy to small unmanned aerial vehicles. Computer vision is a computationally intensive task; we do not want it to 'distract' the flight controller from actually flying the vehicle, as flight control might be running at refresh rates of up to 50 times a second. The separation of flight control ensures the stability and reliability of the vehicle.

As the MJE is both expensive ($\tilde{\pounds}1300$), and fragile, it was decided to go for a top of the range flight controller in an effort to avoid costly crashes. Two main options were considered - the DJI A4 and the PixHawk 2. These are both considered excellent quality flight controllers, and also both allows for external control with a companion computer, using either the DJI SDK for the A4 or one of DroneCore, DroneKit, or MAVROS for the PixHawk. In the end, the PixHawk was chosen as it is open source, allowing for easier modification to our needs; there is also more documentation, tutorials, and other information available online for it.

Reasons for the choice of the DroneKit SDK over alternatives are explained in Section 4.1. Companion computer selection is covered in Section 7.2. The full code can be found on GitHub at `https://github.com/eugeneval/Project_Titan`.

3 Simulation of System Plausibility

3.1 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 recommend 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.

3.2   Aims & Objectives

The specific objectives of this section of the project are:

1. Create a simplified model of the airframe, consisting of easy to model shapes such as cylinders and cuboids.
2. Derive equations of motion for this simplified model, describing how it will respond to forces, moments, and torques.
3. Design a PID control system for the airframe
4. Design and create a simulation environment incorporating this model, equations of motion, and controller, which will follow a test flight path and output relevant values to be analysed.
5. Test the model as a quadcopter only (not including the MJE).
6. Test the model with the MJE included.
7. Decide, based on results of the simulation, whether the stability of the airframe configuration is within acceptable parameters.

3.3   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 multi-rotor uses the 'quad-x' configuration, as shown in Figure 2. 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
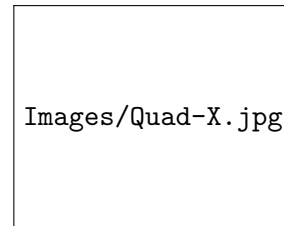

Figure 2: Quadcopter Configuration

8

motors 3 and 4 are spun up; to yaw left, mo-
tors 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 maintained. This can be seen in Equations 11 through 14.

3.3.1 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 modelled 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:

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

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

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

3.3.2 Quad Rotation

With the quad design as described in Section 3.3.1, the following inertias about the three
defined axes are as follows:

$$
\begin{aligned}
I_{Qx} = I_{Qy} \quad = \quad & \underbrace{\frac{\pi \rho h_F}{12} (3(r_2^4 - r_1^4) + h_F^2 (r_2^2 - r_1^2))}_{Frame} \\
+ \quad & 4 \cdot \sin 45 \cdot \underbrace{(\frac{M_R L_R^2}{12} + M_R (r_2 + \frac{L_R}{2})^2)}_{Arms} \\
+ \quad & 4 \cdot \sin 45 \cdot \underbrace{M_M (r_2 + L)^2}_{Motors} \tag{5}
\end{aligned}
$$

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

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

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

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

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

3.3.3 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, calculated in Section 3.3.4 results in the following inertias:

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

It can be seen in Section 3.3.4 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.

3.3.4 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}.

3.4 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:

$$error = setpoint - actual\ value$$

$$integral = integral + (error \times time\ period)$$

$$derivative = (error - previous\ error)/time$$

$$output = kP \times error + kI \times integral + kD \times derivative$$

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

3.4.1 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.

$$
\begin{aligned}
SP_x & \quad \text{Quad X position controller setpoint} \\
SP_y & \quad \text{Quad Y position controller setpoint} \\
SP_z & \quad \text{Quad Z position controller setpoint} \\
SP_{\theta_x} & \quad \text{Quad X angle controller setpoint} \\
SP_{\theta_y} & \quad \text{Quad Y angle controller setpoint} \\
SP_{\theta_z} & \quad \text{Quad Z angle controller setpoint} \\
O_x & \quad \text{Quad X position controller output} \\
O_y & \quad \text{Quad Y position controller output} \\
O_z & \quad \text{Quad Z position controller output} \\
O_{\theta_x} & \quad \text{Quad X angle controller output} \\
O_{\theta_y} & \quad \text{Quad Y angle controller output} \\
O_{\theta_z} & \quad \text{Quad Z angle controller output}
\end{aligned}
$$

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

$$O_x = SP_{\theta_x}$$

$$O_y = SP_{\theta_y}$$

$$F_{M1} = O_z - O_{\theta_x} + O_{\theta_y} + O_{\theta_z} \tag{11}$$

$$F_{M2} = O_z - O_{\theta_x} - O_{\theta_y} - O_{\theta_z} \tag{12}$$

$$F_{M3} = O_z + O_{\theta_x} - O_{\theta_y} + O_{\theta_z} \tag{13}$$

$$F_{M4} = O_z + O_{\theta_x} + O_{\theta_y} - O_{\theta_z} \tag{14}$$

3.4.2 Jet Gimbal Control

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

$$
\begin{aligned}
SP_{\phi_x} & \quad \text{Jet X angle controller setpoint} \\
SP_{\phi_y} & \quad \text{Jet Y angle controller setpoint} \\
O_{\phi_x} & \quad \text{Jet X angle controller output} \\
O_{\phi_y} & \quad \text{Jet Y angle controller output}
\end{aligned}
$$

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

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

3.5 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[13]. 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 3. It can be seen that there is an oscillation in x and y position of approximately 0.2m. It proved difficult to eliminate, due to the not straightforward interaction of position and angle PID values, and could potentially cause issues with the accuracy of payload delivery.

This, however proved not to be a problem. It can be seen in Figure 4a 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 of several meters and a steady state error in x and y of about 0.2m, but this can be corrected by fine-tuning of PID values. The flight time has also increased but this is to be expected due to the increase in mass causing a decrease in acceleration. In Figure 4b 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.

3.5.1 Programming Detail

Seen in Figure 5 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 attributes which cover information such as mass, position, velocity,

---

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

Graphs/square_path_quad_only.png

Figure 3: Simulation of a Square Path - Quad Only

(a) Position

(b) Angle

Figure 4: Simulation of a Square Path - With Jet

Figure 5: XQuad2 UML Class Diagram

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\textunderscore 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 3.4, 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:

```
1  float calculate(float time, float actual) {
2      float error = setPoint - actual;
3      integral += (error*time);
4      if (integral > integralWindUp) {
5          integral = 0;
6      }
7      float derivative = (error - prevError)/time;
8      prevError = error;
9      return (kP*error + kI*integral + kD*derivative);
10 }
```

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 to calculate the resultant movement of the object, as well as recording the result. It uses the current values of velocities and accelerations, as well as exerted forces and moments, by 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 the GitHub repository, located at `https://github.com/eugeneval/Project_Titan`.

3.6 Refinement

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). Additionally, the maximum torque of the servo had not been limited to 0.34kgm. Correcting these resulted in Figures 6a and 6b.

These refinements decreased the stability of the drone. It is still well within acceptable parameters for rotation, and for x and y positioning; in fact the steady state error in x and y has been eliminated after further turning of the PID values. The overshoot in height has increased further, and is now over 100%. There is also a steady state error approximately

16

(a) Position

(b) Angle

Figure 6: Simulation of a Square Path - With Refinements

1m, and an oscillation also of approximately 1 m.

3.7 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 concerned 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.

A simplified model of the airframe configuration has been created and equations of motion derived. A simulation environment has been designed and created from the ground up, and the model created in this simulation. Several configurations have been tested and the results analysed.

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. We have completed our objectives for this section, as defined in Section 3.2, determining that the stability of the airframe configuration is within acceptable parameters.

4 Companion Computer Software: PX4 Control

4.1 Introduction

We will be controlling the PixHawk flight controller using offboard commands from an external companion computer, henceforth referred to as Pete (since CC is a rather ambiguous 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 parameters by itself and continuously 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 environments[4]. This is crucial, as it allows code to be tested with no danger to the real vehicle. Pete can communicate 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.

PX4 has 12 flight modes when employed on a multi-rotor. 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 using the MAVLink communication protocol. 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 implementations: on Android, on iOS, and in Python. It is the Python version we will be using here, as almost all available companion 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 on time was the larger priority. Python is also more suited to be used as a cross-platform language, which allows for code to be created in one operating system, MacOS, and eventually run in another, specifically Linux (which all available companion computer run on a distribution of). 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.

4.2 Aims and Objectives

1. Research autonomous vehicle flight control solutions and decide on a configuration. *(Already completed)*
2. Download and set up PX4, JMAVSim, and Dronekit.
3. Run a SITL PX4 simulation, in conjunction with a jMAVSim simulation, and become familiar with this simulation environment.
4. Create code using DroneKit that will communicate with the PX4 simulation and JMAVSim.
5. Create code using DroneKit that will allow for a predetermined flight path to be followed, and test using PX4 and jMAVSim.
6. Create code using DroneKit that will allows for real-time flight control relative to the current position (required to allow the vehicle to navigate to the target in response to

an input from the computer vision module), and test using PX4 and JMAVSim.

7. Package created code in a module to allow for easy implementation in a full control system.

4.3   Implementation

The available documentation and examples for DroneKit mostly make use of the simple_goto() command[4][5]. 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 altitude 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):

    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[4][5].

```
def arm_and_takeoff(self, targetAlt, accuracy=0.5):
```

```python
 2      """Overrides DroneKit command. Takeoff to a target altitude, then loiter.
        """

 3

 4      wp = get_location_offset_meters(self.home_location, 0, 0, targetAlt)
 5      self.commands.add(PX4Command(wp, "TO"))
 6      self.commands.upload()
 7      time.sleep(1)

 8

 9      self.mode = VehicleMode("MISSION")
10      time.sleep(1)
11      print("Vehicle mode should be MISSION: %s" % self.mode.name)
12      self.armed = True
13      while True:
14          print " Altitude: ", self.location.global_relative_frame.alt
15          if self.location.global_relative_frame.alt>=targetAlt-accuracy:
16              print "Reached target altitude"
17              self.mode = VehicleMode("LOITER")
18              time.sleep(0.5)
19              break
20          time.sleep(1)

21

22  def goto_absolute(self, pos_x, pos_y, pos_z, accuracy=0.5, wait=True, text=
        True):
23      """Go to a position relative to the home position"""

24

25      targetLocation = LocationLocal(pos_x, pos_y, -pos_z)

26

27      self.__send_ned_position(pos_x, pos_y, -pos_z)
28      self.mode = VehicleMode("OFFBOARD")
29      if text: print("Vehicle mode should be OFFBOARD: %s" % self.mode.name)

30

31      while True:
32          self.__send_ned_position(pos_x, pos_y, -pos_z)
33          remainingDistance = get_distance_metres_local(self.location.
        local_frame, targetLocation)
34          if remainingDistance<=accuracy:
35              if text: print("Arrived at target")
36              break
37          if wait == False: break
38          if text: print "Distance to target: ", remainingDistance
39          time.sleep(0.1)

40

41  def goto_relative(self, pos_x, pos_y, pos_z, accuracy=0.5, wait=True, text=
        True):
42      """Go to a position relative to the current posotion"""

43

44      currentLocation = self.location.local_frame
```

```
45      targetLocation = get_location_metres_local(currentLocation, pos_x, pos_y,
        -pos_z)\

46

47      self.__send_ned_position(targetLocation.north, targetLocation.east,
        targetLocation.down)
48      self.mode = VehicleMode("OFFBOARD")
49      if text: print("Vehicle mode should be OFFBOARD: %s" % self.mode.name)

50

51      while True:
52          self.__send_ned_position(targetLocation.north, targetLocation.east,
        targetLocation.down)
53          remainingDistance = get_distance_metres_local(self.location.
        local_frame, targetLocation)
54          if remainingDistance<=accuracy:
55              if text: print("Arrived at target")
56              break
57          if wait == False: break
58          if text: print "Distance to target: ", remainingDistance
59          time.sleep(0.1)

60

61 def setMaxXYSpeed(self, speed):
62      """Required due to DroneKit's vehicle.groundspeed not working with PX4"""

63

64      self.parameters['MPC_XY_VEL_MAX']=speed
65      print("Set max speed to: %s" % self.parameters['MPC_XY_VEL_MAX'])
66      time.sleep(0.5)

67

68 def returnToLand(self):
69      """RTL and wait for auto disarm on land"""

70

71      self.mode = VehicleMode("RTL")
72      time.sleep(1)
73      print("Vehicle mode should be RTL: %s" % self.mode.name)
74      while self.armed == True:
75          print("Waiting for landing...")
76          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 respectively. 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. These

commands were all tested using a SITL PX4 simulation in conjunction with jMAVSim.

4.4 Conclusion

The code that has been written 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, dropping the payload, returning to base, and landing.

The created code has been throughly tested using a SITL PX4 simulation and jMAVSim, and packaged into a module which can be easily integrated into an overall control system. Using object-oriented programming principles will allow the code to be easily extended or modified later if this is required.

5 Companion Computer Software: 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, incorporating 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 identify 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.

5.1 Aims and Objectives

1. Research computer vision solutions and decide on a configuration. *(Already completed)*
2. Download and compile OpenCV-Python and other relevant packages, and become familiar with them.
3. Create code that can recognise the IMechE target using computer vision, and test it.
4. Create code that can read the alphanumeric code inside the IMechE target, and test it.
5. Create code that can calculate the offset of the target from directly underneath the vehicle (we will use this offset as an input for navigation), and test it.
6. Package created code in a module to allow for easy implementation in a full control system.
7. Integrate this new module with the existing control module, and test it using PX4 and jMAVSim.

5.2 Square Detection

The target consists of two concentric squares and a central letter, as seen on Figure 7[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 optimal method.

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

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

Figure 7: Target

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].
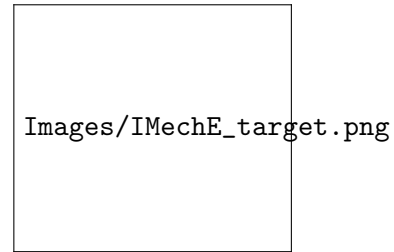
5.3 Optical Character Recognition

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 conjunction 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.

5.4 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 instance 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 separate 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 eliminate 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.

5.5 Integrating into Control

Using the code developed in Section 4.3, 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.

Multithreading was used to optimise performance and increase effectiveness. Unfortunately, it could not be fully implemented, as OpenCV's VideoCapture(), used to receive input from the camera, is only capable of functioning inside the main thread. This is a known unresolved issue with OpenCV itself. This meant that receiving input could not occur in a subthread, which is unfortunate as input/output operations are normally the most desirable operation to keep in subthreads. In this version of the program, the main thread handles computer vision and a subthread contains the vehicle class which communicates with the PixHawk.

A demonstration video of this was created. For obvious reasons, it could not be put in this report, but is available at `https://youtu.be/sFyZphH8_CM`.

5.6   Investigating Frame Rate

The method of finding all squares, checking through them all for duplicates, then checking through them all for targets, intuitively seems inefficient. 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.

5.7   Conclusion

A module has been created, throughly tested, and integrated that allows for IMechE targets to be recognised and used to navigate the vehicle.

Investigations have been made into multithreading and frame rate. These can be taken further once a specific companion computer is chosen, as any improvements that can be made here are heavily dependent on the specific architecture of the chosen computer and operating system.

Potentially, some of the parameters used to identify the target will need to be fine-tuned once deployed on a real vehicle to account for how the target may look different. The code has been written in such a way that these parameters are easy to modify.

6   Companion Computer Software: Communication

Communication from Pete to a base station to transmit information will, by necessity, be over a serial connection, as this is how most low-end radio links are designed.

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

6.1   Aims and Objectives

1. Research serial communication solutions and decided on a configuration. *(Already completed)*
2. Download and become familiar with relevant libraries.
3. Create code that will allow for relevant information to be sent over a serial connection, and test it using socat.
4. Package created code in a module to allow for easy implementation in a full control system.
5. Integrate this new module with existing software, and test it.

6.2   Implementation

Several methods for serialising information was investigated. Python has a library called pickle, which converts an object to a serial format by 'pickling' it, and 'unpickling' into the complete object at the other end of the serial connection. This method was investigated, but has certain limitation with the types of data that can be pickled and unpickled, and is also somewhat unreliable.

Instead, the JavaScript Object Notation (JSON) was used. This is a standard method of formatting the attributes of objects to a text string, which can then be sent over a serial connection. JSON is a well established method for transmitting object attributes over many different kinds of connections between different devices and programming languages, and is a good solution as it is inherently simple and reliable.

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 successfully. 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.

6.3 Conclusion

Through the usage of JSON and the pyserial library the location of a target can be sent over a serial connection. This has been tested using socat and integrated into existing functionality.

The UAS Challenge requires the GPS coordinates of a target to be relayed to the base station. This will be done by knowing the current location of the vehicle using our control module, and knowing the offset of the target from directly underneath the vehicle. Calibration will be needed upon deployment on a real vehicle to ensure accuracy, but otherwise this module is complete.

7 Companion Computer Software: Other Concerns

7.1 Jet Engine Control & Payload Delivery

The thrust output of the MJE will need to be controlled to compensate for the change in weight as payloads are dropped. Additionally, by more closely controlling jet thrust it may be possibly to reduce the instability in height noted in Section 3.6.

The MJE is controlled with a FADEC that takes a PWM (Pulse Width Modulation) signal as an input. PWM is a standard method of sending a digital signal to control a single variable such as speed on a motor, widely used in electronics; most companion computers are capable of creating such a signal using output pins and libraries for it in Python are widely available. This input signal can vary from 1000 to 2000 microseconds, and the FADEC is fully configurable as to what this range represents. In our case, it has been configured so that 1000 microseconds corresponds to idle and 2000 microseconds corresponds to maximum thrust.

Additionally, a mechanism to control the payload delivery mechanism is also required. Development of the specifics of payload delivery is being undertaken by a 1st year student, Arkadiy Serezhkin. The current solution will likely use a servo mechanism of some description. An example of such a

Images/makeblock_gripper.jpg
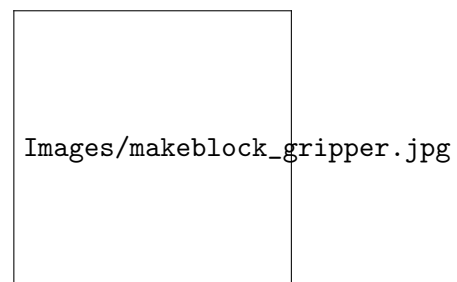
Figure 8: Makeblock 86502 Robot Gripper

28

mechanism would be the Makeblock 86502
Robot Gripper[9], seen in Figure 8.

The issue of a moving center of gravity
when a payload is dropped is also an issue
that is currently being investigated. Placing the payload underneath the jet engine, with a
cone to divert the jet exhaust, was rejected due to the large loss of thrust from the jet this
would cause. Placing the payloads evenly distributed, as close to the center of the vehicle
as possible, still raises issues with one of the electric motors having to take on too much of
the weight. Whatever the final solution, it may well factor in to the programming required
for the payload drop mechanism.

The full implementation of this will depend on the specific companion computer chosen,
as the libraries to output signals using the output pins of a companion computer are specific
to that companion computer, as well as the final payload deliver method chosen. For now,
a Payload class has been created, and incorporated into the Vehicle. Later, the specific code
for controlling payload drops, and varying jet engine thrust in response to payload drops,
will be incorporated into this class.
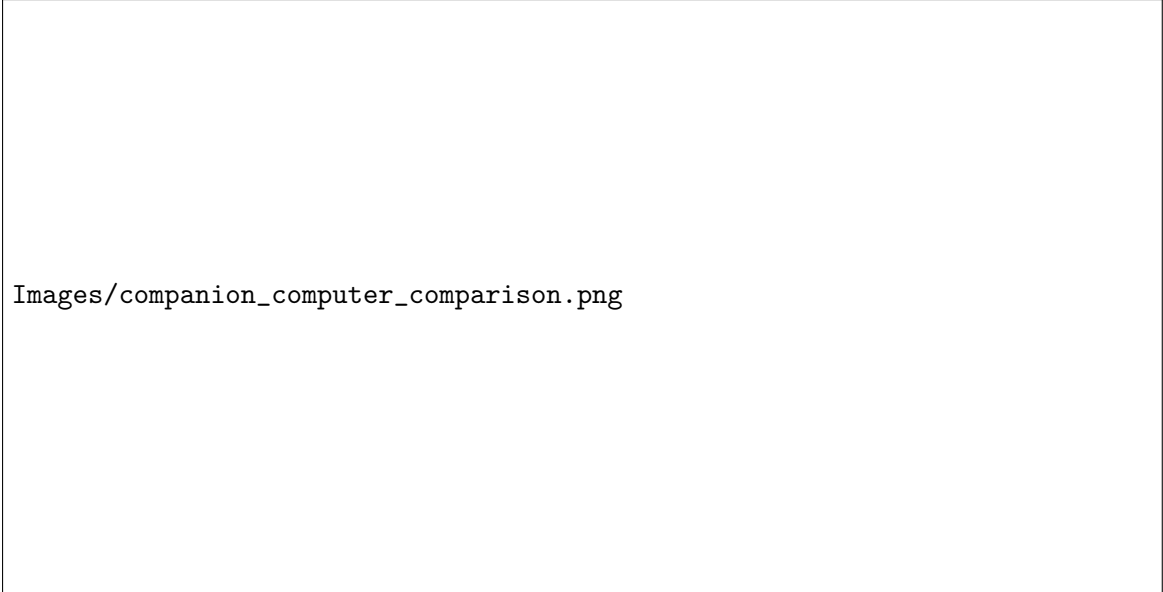
7.2 Choosing a Computer

By this point, Pete has advanced about as far as is possible in a purely simulated environ-
ment. 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 available 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
Figure 9.

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[10].
The Nvidia TX2 is a close second. It is definitely 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.

7.3 Companion Computer Setup & Testing

A Raspberry Pi3, along with camera, was obtained, and Raspbian (a distribution of Linux
specifically designed for the Pi) 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

Figure 9: Companion Computer Selection

OpenCV requires around 4GB free space to compile from source. Therefore, a 32GB microSD card was obtained and the setup process was restarted. OpenCV was successfully compiled, and other required libraries such as DroneKit were also acquired.

Slight modification to the existing code needed to made, mainly that the picamera library was used as an intermediary between the Pi's camera and the existing computer vision code.

When tested the code performed as expected, communicating with the simulation running on the main computer over a UDP connection, and recognising targets using the Pi's camera. This is important because it means the companion computer is ready to be installed on the vehicle, and also validates the choice of Python as a cross-platform programming language.

8   Conclusion & Further Work

1. The stability of the airframe configuration has been examined in a simulated environment, and shown to be within acceptable parameters in the x and y directions. There is a large overshoot and not inconsiderable oscillation in z (height), but this is acceptable as the required precision in height for our vehicle is very low.

2. A control system has been designed and tested within a simulated environment, which can:
   (a) Travel to predetermined GPS waypoints.
   (b) Recognise and navigate to a target using computer vision.
   (c) Communicate the GPS coordinate of the target to a ground station.

3. The control system has been uploaded to and adapted for a companion computer.

More detail on each of these completed objectives is available within the sub-conclusions seen throughout the report.

The control system has been tested as far as is possible within the current simulated environment. The next stage is preferably deployment on a real platform, where further testing and development can take place. This is dependent on Ismail delivering a working airframe; at the time of writing, all design work has been completed with manufacturing to commence shortly[12].

An alternative would be to move to a more advanced simulation environment, specifically Gazebo, which is an advanced robotics and physics simulation library and which is capable of interfacing with a PX4 simulation[4]. Enough research and investigation has been done into this to discover that creating an accurate simulation in Gazebo, with our custom airframe modelled correctly and integrated with the PX4 simulation, is almost ambitious enough to form a third-year project in and of itself.

The design of the control system so far allows for a great deal of flexibility and expansion through the use of object-oriented programming, although further work could be done in this area, especially when it comes to the usage of OpenCV and multithreading (This is a known issue, as described in Section 5.5). The use of the picamera module rather than OpenCV's VideoCapture may allow image processing to be moved into a subthread and proper multithreading implemented. Additionally, the code that has been written could be better packaged, although this would only really become an issue if it was being distributed.

A   Table of All Symbol Definitions

Note: values are only given for variables which are constants. No values are given for values derived from constants 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	$2700 kgm^3$
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	4 kg
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	

Symbol	Definition	Value(where appropriate)
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	
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 alphabetical order:

Abbreviation	Definition
API	Application Programming Interface

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

References

[1] IMechE. About UAS Challenge - IMechE [Internet]. [Accessed 20 December 2017]; Available 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]; Available 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 December 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/`.

[9] Rapid Eletronics Limited. Makeblock 86502 Robot Gripper up to 1.5KG with 12V Motor [Internet]. [Accessed 16 April 2018]; Available from: `https://www.rapidonline.com/Makeblock-86502-Robot-Gripper-up-to-1-5KG-with-12V-Motor-75-0703?IncVat=1&pdg=aud-313476734810:pla-339342696145:kwd-339342696145:cmp-757438067:adg-44804851896:crv-207912323492:pid-75-0703:dev-c&gclid=CjwKCAjwk9HWBRApEiwA6mKWaUnklYF9qNoe1kG8GOwc0quum8lmKrSm6ndSPCsK9-yZVVFx949oThoCedMQAvD_BwE`

[10] Dries Hulens, Jon Verbeke and Toon Goedemé. How to choose the best embedded processing platform for on-board UAV image processing. In: Computer Vision, Imaging and Computer Graphics Theory and Applications edition:VISIGRAPP 2015 pages:455-472.

[11] Andrew Gibiansky. Quadcopter Dynamics, Simulation, and Control [Internet]. 23rd Nov 2012. [Accessed 23 November]; Available from: `http://andrew.gibiansky.com/downloads/pdf/Quadcopter%20Dynamics,%20Simulation,%20and%20Control.pdf`

[12] Syed Ismail Ahamad. Unmanned Ariel Systems. MECH3002 Final Report. 25 April 2018.

[13] Processing 3. Processing Foundation. `https://processing.org`.

[14] Python 2.7. Python Software Foundation. `https://www.python.org/`.

[15] socat - Multipurpose relay (SOcket CAT). Gerhard Rieger. `http://www.dest-unreach.org/socat/doc/socat.html#CREDITS`.