Unmanned Aerial Systems

Eugene Valetsky Supervisor: Dr Tim Baker

Note: referenced figures can be found on the last page.

1 Introduction

1.1 Background

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 July 2018.

1.2 Problem Definition

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.

1.3 Design Concept

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.

2 Design Requirements

2.1 System Plausibility

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.

Therefore, it was decided to build a simulation to test the validity of the concept that the MJE can be ignored. The goal of this simulation is to determine whether PID controllers are capable of stabilising the vehicle, the logic being that a significantly more advanced flight stack such as the PX4 should have no problems.

2.2 Control Software

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.

PX4 is fully capable of Software-in-the-Loop (SITL) simulation, using either the jMAVSim or Gazebo 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.

The goal is to write software that will run on Pete that will perform three main functions:

1. Communicate with and control PX4 over MAVLink.
2. Recognise the target using computer vision.
3. Communicate with a base station.

3 Design Methodology

- Simulation
    1. Create a simplified model of the vehicle.
    2. Determine equations of motion for this model.
    3. Creating PID controllers.
    4. Create a simulation based on these equations of motion and PID controllers.
    5. Test the validity of the core concept.
- Software: PX4 Control
    1. Research and download required libraries.
    2. Succesfully run a SITL PX4 simulation.
    3. Create a program that can communicate with PX4 over MAVLink.
    4. Use this program to send commands to and control PX4.
    5. Integrate with other software on Pete.
- Software: Computer Vision

    1. Research computer vision and download required libraries.
    2. Create a program that can recognise squares.
    3. Create a program that can recognise characters.
    4. Finalise this into a program that can recognise the target and provide useful information.
    5. Integrate with other software on Pete.
- Software: Communication
    1. Research and download required libraries.
    2. Create both sides of the program (on Pete and on the base station) and establish a connection.
    3. Adapt the program to send required information
    4. Integrate with other software on Pete.

4 Progress

4.1 Simulation

4.1.1 Equations of Motion

The vehicle is split into two sections, the outer quadcopter frame, *Quad*, and the gimballed section containing the MJE, *Jet*.

The quad is modeled as an inner thin, hollow cylinder, the *frame*, with four cyclindrical 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 jet is modelled as a cylinder.

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.) The multirotor uses the 'quad-x' configuration.

From these initial assumptions, equations of motion were determined. These consist of:

1. The total mass of the vehicle.
2. Sum of forces on the vehicle in x, y, and z directions.
3. Inertia of the quad.
4. Sum of torques on the quad.
5. Inertia of the jet.

4.1.2 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. 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 controllers are independent, as are the servo controllers.

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. The servo controller output determines the torque exerted by the servo.

4.1.3 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 1a. 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 1b 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 1c 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.

4.1.4 Refiment

It was realised that the mass of the quad had been implemented incorrectly. Additionally, the maximum torque of the servo had not been limited.

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.

The result of these refiments can be seen in figures 1d and 1e.

4.2 Software: PX4 Control

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.[2] 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 available documentation and examples for DroneKit mostly make use of the simple_goto () command. Unfortunately, as DroneKit is primarily designed to work with APM, the predecessor of PX4, this command does not work. 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. local_frame is the most immediately useful to us, as it is simpler and more intuitive to use.

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, send_ned_position().

Similarly, other commands have been created to suit our purposes. Some of these have been adapted from examples in the DroneKit and PX4 documentation.[2][4] 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.

4.3 Software: Computer Vision

A program was created that can recognise squares. This is done by detecting edges in the image, simplifying those edges to contours (a series of points), and seeing which of those contours fit our defintion of a square.

The program can also do Optical Character Recognition (OCR), using the Tesseract engine and pytesseract library.

This program was based on examples available online.[6][7][8]

4.4 Software: Communication

No progress has yet been made on this aspect of the project.

5 Conclusions & Further Work

5.1 Simulation

It appears that the addition of a gimballed jet to a quadcopter does not destabilise it. The extremely simple PID controllers used in our simulation are able to cope, 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 appears valid. Further refiments to the simulation could be made in two ways:

Firstly, the accuracy of the simulation itself could be increased. Many assumptions and simplifications have been made throughout, from assuming the controllers have access to perfect angle and position information, to ignoring the effects of friction and stiction in the gimbal. This would increase the validity of our conclusion.

Secondly, the PID controllers could be improved. It is suspected that the massive overshoot in height could be solved through the introduction of a 'limited integral' controller, that only uses the integral term when it is close to the setpoint. More research into the PX4 source code is required to determine what control algorithms it uses and to possibly implement them, increasing the similarity of our simulation's response to how a real PixHawk would respond.

5.2 Software: PX4 Control

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.

There is little further work to be done on this aspect of the project, other than integration with other software components as and when they are completed.

5.3 Software: Computer Vision

Currectly, the program suffers from an issue where it cannot detect shapes within other shapes, as it relies on defining shapes relative to the background. This is an important capability, as looking for concentric squares will likely be the best way to detect the target. More work is neeeded to find a solution for this, and to further adapt the program to search for our target specifically.

5.4 Overall Conclusion and Remaining Work

The simulation can be improved further, but for now seems to indicate that our concept is viable. Of the software that will go on Pete, PX4 communication over MAVLink is complete, computer vision is a work in progress, and communication with a base station is yet to be started.

Once the control software is complete, it is hoped to create a new simulation using Gazebo. Gazebo is an advanced physics simulator for robots, and crucially can integrate with the PX4 SITL simulation. Such a simulation would be the most accurate way of determing the validity of both the design concept and control software.

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.

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

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

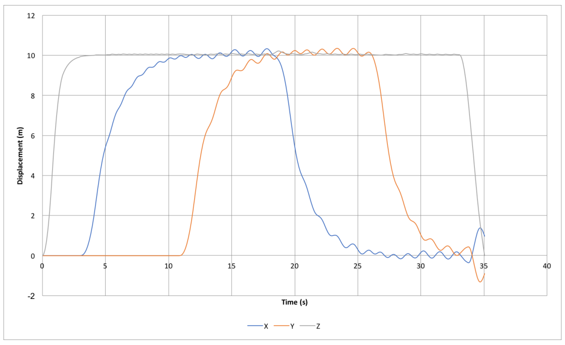
[4] DroneCode. PX4 Development Guide [Internet]. Updated 15 December 2017. [Last accessed 20 December 2017]; Availible from: https://dev.px4.io/en/.

[5] 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.

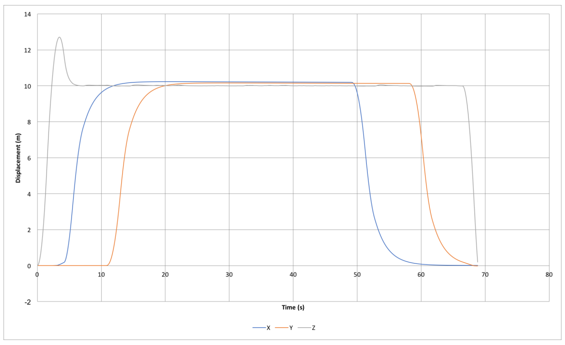
[6] OpenCV. OpenCV-Python Tutorials OpenCV 3.0.0-dev documentation [Internet]. Last updated 10 Novemeber 2014. [Accessed 22 December 2017]; Available from: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html.

[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/.

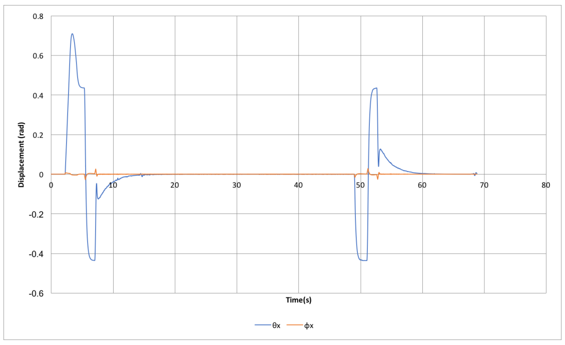
[8] Adrian Rosebrock. Using Tesseract OCR with Python [Internet]. 10 July 2017. [Accessed 22 December 2017]; Available from: https://www.pyimagesearch.com/2017/07/10/using-tesseract-ocr-python/.



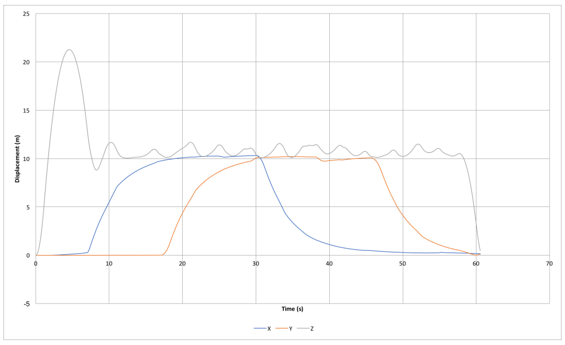
(a) Quad Only



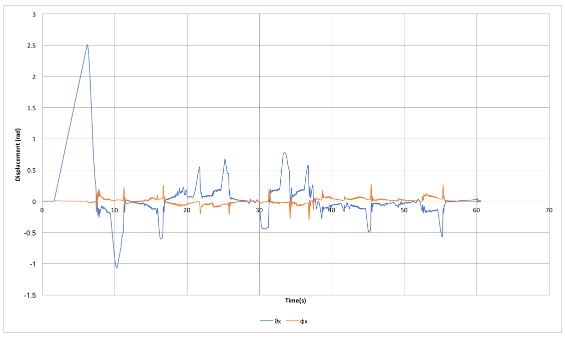
(b) With Jet



(c) With Jet



(d) With Refiments



(e) With Refiments

Figure 1: Simulation of a Square Path