AUTONOMOUS FLIGHT: PARROT A.R. DRONE 2.0

Software Design & Development Project

Cathal Moran	13432808
Luke Burke	13490582
Deirdre Henry	97622923

Contents

Acknowledgements

Firstly we would like to thank the NUIG Information Technology department for the experience and education received throughout our time in the university. The lessons learned as part of our degree are highly valuable and we will take them with us in our future careers.

Secondly we would like to thank Dr. Finlay Smith and all the IT department staff involved in the Software Design & Development projects. In these past few months we have learned how to work as a group in order to achieve our goals whilst also gaining new skills.

Lastly we would like to thank our supervisor Dr. Seamus Hill, whom without this project would not have been achievable. We are all extremely grateful for his patience and guidance throughout this year

1. Introduction

For this project, we were asked to program a drone to fly autonomously while being able to detect a marker. This report will go through the different steps taken in order to achieve this. We hope that this document will provide others with a guide on how best to approach similar projects.

The document will provide a combination of detailed descriptions along with some snippets of code used in each of these steps taken. For this project, we were given the AR Parrot Drone 2.0 to work with.

2. Aims

Our aims for this project was to program a drone to fly autonomously around an indoor room. In doing this we set out to try and make the drone do certain movements and actions. These may change and differ due to drone or software limitations, however the fundamental idea to fly the drone autonomously remains constant throughout the project.

At the beginning of the year we decided on a set of goals that we wanted the drone to achieve. These included:

- To fly autonomously without any user input
- To fly around the walls of a square indoor room i.e. a perfect square
- To adjust the drone's altitude when flying over objects
- To search a room for a marker or object
- To calculate the area of the room
- Use the drone's cameras

Throughout this report we will address each of these aims and discuss the way in which they were carried and any issues that were encountered.

When working in a group project such as this it was important that we worked together in order to successfully complete such a challenging task. This included

learning new skills and testing each our individual intellectual abilities. That is why it was important to set personal goals as well as group targets. Some personal aims and objectives that we set out to achieve as individuals included:

- To learn new technologies
- To learn a new language or to build upon the knowledge that we have on a given language
- To gain an in-depth knowledge of drones and their future uses
- To improve our communication, teamwork and problem solving skills

3. Software Testing

As part of this project software testing was carried out throughout the process of writing code. Different aspects of software testing were used in order to ensure that the code was working properly and efficiently. This allowed us to put into practice what we have learned through programming languages and our time studying software engineering.

3.1 Unit Testing:

The purpose of unit testing is to ensure that each unit of the software performs as designed with no errors or flaws. This was essential for this project as with the drone we had specific time slots for testing code that we had written. This meant that if a large portion of the code was written at once and an error was thrown during the test, we would be wasting time trying to find the issue and correct it in a large chunk of code. Therefore, by breaking the code into smaller sections and testing each of these individually we were able to pinpoint any errors in a timely manner.

3.2 Integration Testing:

This process of testing is used to combine the previously tested units of code to see if there are any issues with the interaction of each element. This may be important when methods are written to be used in various pieces of the code such as the *tagDetection()* method for the drone. When each of the units were found to be

working individually we then combine these together in a larger chunk of code to ensure everything is working together.

3.3 System Testing:

The use of system testing in our project was for testing the system as a whole to ensure that each of our goals were met. This involved taking a look at our initial requirements and examining if each of these were involved in the final code itself. If there were any issues such as missing requirements or something that wasn't working as fluidly as we hoped, we would then analyse that specific code and begin testing from the beginning after fine tuning or alteration.

3.4 Acceptance Testing:

This testing level was used at the very end when preparation for the demonstration. This involved changing various pieces of code and adding extra comments. Some elements of the code had to be slightly modified to ensure the demonstration would run smoothly and the drone would fly the way in which we had envisioned.

4. About the AR Drone 2.0

4.1 Main Components

The AR Drone 2.0 (fig.1.1) is a quadricopter comprising of four rotors. The drone's four rotors are attached to the four ends of a crossing to which the battery and the RF hardware are attached. This crossing is referred to as the chassis. The chassis is made up of a combination of carbon and plastic, making it lightweight but strong at the same time.

Figure1.1: AR Parrot Drone 2.0

4.2 Propellers

The propellers are divided into two pairs. One pair rotating clockwise and another pair ant-clockwise. This is displayed below in (fig.1.2). Both the front left and back right propellers are rotating clockwise, where as the front right and the back left are rotating anti-clockwise. Movement is executed by each propeller varying the speed of their rotation.

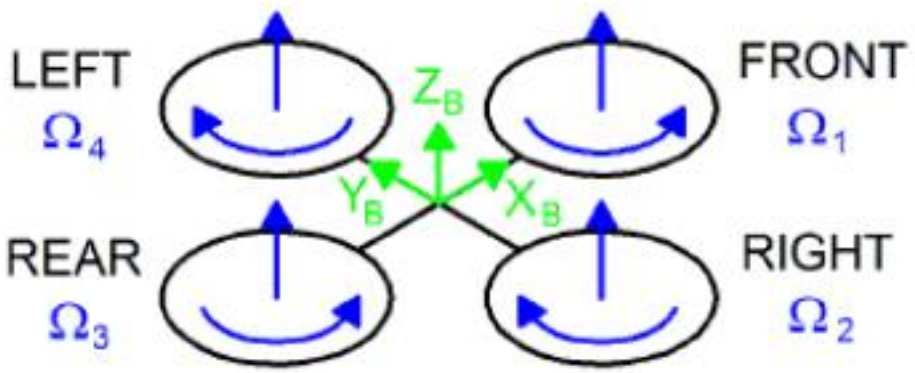


Figure1.2: Propeller movement (AR Drone Developer Guide, 2012)

4.3 Protective Cover

The AR Drone comes with a protective outer cover (fig.1.3). The cover is made out of EEP (expanded polypropylene) which is a very light material which helps with the flight of the drone. The cover's purpose is to prevent the propellers form hitting off objects and causing damage to either the drone or the object. It would be highly recommended to use this outer covering when flying the drone inside.

Figure1.3: Protective Cover

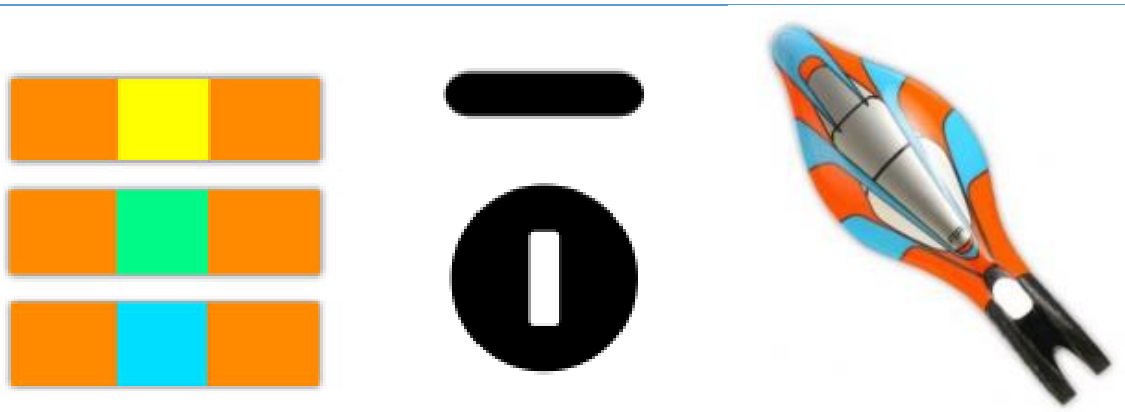
4.4 Battery

The AR parrot drone uses a 1500mAh, 11.1V LiPo battery. The battery takes approximately an hour to become fully charged and then typically last for about 15 minutes of flight depending on what kind of tasks are being carried out. We found that when the battery reached about 20% the drone would terminate flight or the program being ran and automatically land.

4.5 Camera

The drone possesses two cameras, one facing the front and one facing the ground. The higher quality front camera uses a resolution of 720p (1280 x 720), whereas the bottom facing camera is of much lower quality. The drone has the ability to detect 3 different markers. Both cameras are able to detect three different tags.

These are the three tags:



(a) 2D tags (b) Orientated roundel (c) 2D tags on out side shell

Figure 1.4: Detection Tags

4.6 Wi-Fi Network and Connection

The AR Drone can be controlled from any client device supporting Wi-Fi ad-hoc mode. The process is carried out as listed below:

- The Drone sets up a Wi-Fi network with an EESID usually called "ardrone_xxx" and then allocates a free, odd IP address.
- The user can then connect the client device to the EESID network created.
- The device used by the client then requests an IP address from the drone DHCP server.
- The Drone DHCP server grants the client device with an IP address which is 192.168.1.x
- The client can then start sending messages/requests to the AR Drone.
- The client can also use the Wi-Fi ad-hoc network. If the drone can detect a network which is already present with the SSID, it just joins the already present network.

5. Ubuntu Virtual Operating System

Initially we used the Ubuntu 16.06 operating system mounted on a virtual machine using VMware Workstation Player. This allowed us to use Ubuntu in the early stages of testing and getting accustomed to a Linux operating system.

However, by using a virtual machine it affected performance using Ubuntu and overall was not very affective. There were issues with the overall speed of the operating system and on some occasions there would be an error that would cause a login loop to occur and all data was lost and a fresh install had to be carried out.

Similar issues were also found with Ubuntu 14.04, although the overall speed of this version of Ubuntu seemed to be faster and smoother. We came to the conclusion that we would have to use an alternative approach for the long run.

6. Dual Booting Windows 10 with Ubuntu 14.04 (Trusty Tahr)

The solution to the performance issues found in using the virtual machine was to dual boot Ubuntu alongside Windows on one of our laptops. This would then be the designated laptop for testing code with the drone.

The first step in this process was to decide on which version of Ubuntu would be most beneficial to us. We chose Ubuntu version 14.04 as we found the performance on the virtual machine to be faster, and if we ran into any issues with software it would be easier to upgrade rather than downgrade.

The next step was to create a bootable USB drive. In order to do this Win32 Disk Imager was downloaded onto the laptop. This is a program that allows you to copy Linux disk images to a USB. The USB was inserted into the laptop and a quick format was carried out and any data was removed from the USB. Using Win32 Disk Imager the Ubuntu 14.04 image was written to the USB.

Next, 'fast start up' had to be turned off on Windows 10 as this does not allow for booting from a USB. A partition also had to be created in the hard drive. This is where Ubuntu

would be located. A total of 50GB was allocated as the recommended amount for good performance was above 30GB. As hard drive space was not an issue an extra 20GB of space was allocated. The laptop was then restarted and had to be booted into safe mode. In order to boot into the USB, the EFI USB device option was selected. Ubuntu was then opened and the option to try before installing was selected to ensure that the operating system was working without any issues.

When everything was found to be working in order, the option to install Ubuntu alongside Windows boot manager was selected. When the install was finished the option to continue testing was selected so that the boot order could be edited. The boot order was set so that Ubuntu boots first followed by Windows boot manager. The command for this in the shell was "sudo efibootmgr -o 4, 2". The computer then had to be restarted and the USB removed when prompted.

Now when the laptop starts you are given the option to boot into Ubuntu or Windows 10.

7. Atom

Atom is a text editor that allows for the use of many languages such as Python, Java, Objective-C, HTML and many more. It is a free and open source editor. This means that it is fully customizable for any user. This may not be of huge benefit for our project, however by using Atom it allows us to use a new text editor in order to further our knowledge even more.

Eclipse and notepad would have been mainly used in our studies so it is important to push the boundaries when given the chance to learn something new. The terminal window is used to compile and run any of the python programs that are written in Atom.

8. PS Drone API

The PS-Drone-API (application programming interface) is a fully featured SDK (Software Development Kit). It was developed as part of a Master of Computer

Science Dissertation by J. Philipp de Graaff. It is written in Python and is specifically written for Parrot's AR Drone 2.0.

Following much research by our group, together with numerous trials (and tribulations) with other SDK's, we found this to be the most concise, user friendly and easy to understand with a very quick setup time. In addition, as a group we were also happy to be learning a new programming language while completing this project.

The documentation is easy to navigate and learn from. It also provides a full set of possible uses for the AR Drone 2.0, including Sensor-Data, Configuration and Video-Support. When used with OpenCV2, it is possible to analyse video images data also.

A full list of commands and a description of all sensor data is available in the documentation, along with tutorials. These tutorials provide explanations of the most important commands for example:

3.3.6 *Basic movement*

Drone moves or turns to given direction, until it gets the command to change direction.

Usage: moveLeft(optional)
moveRight(optional)
moveForward(optional)
moveBackward(optional)
moveUp(optional)
moveDown(optional)
turnLeft(optional)
turnRight(optional)

Return: None

Name:	Type:	Description:
optional	float	Speed value from *0.0* to *1.0*; from no active movement to full thrust. If empty, the value from *speed* will be used.

Note: Very basic movements for first tests, with only one movement at a time. Setting all values to *0.0* does not make the drone stop, but glide in the same (horizontal) direction as before. To stop movement, use *stop()* instead.

Example: *drone.moveLeft(0.4)*

8.1 Requirements for Setup

The main requirements needed to use this API:

- Computer or virtual machine with an operating system like Linux
- AR Drone 2.0
- Python 2.x
- OpenCV2 (required for video-support)
- CPU equal 1.83GHz Intel Core Solo T1400, 2.4GHz E6600 Core 2 Duo CPU or better recommended for video-analysis

8.2 Additional/Alternative requirements that our team used:

- Dual Boot Ubuntu 14.04 (Trusty Tahr) with Linux operating system (section 6)
- Atom Text Editor (section 7)

When you have everything installed and you are ready to get started, simply enter python ps_drone.py and off you go. There is a simple demo programme which will allow you to control the drone with your keyboard to get you started.

At the beginning of our code you need to import the ps_drone API. It is then initialised to the variable drone.

```
# -*- coding: utf-8 -*-
##### Suggested clean drone startup sequence #####
import time, sys
import ps_drone                 #Imports the PS-Drone-API

drone = ps_drone.Drone()        #Initials the PS-Drone-API
drone.startup()                 #Connects to the drone and starts subprocesses

drone.reset()                   #Sets drone's LEDs to green when red
```

9. OpenCV

To use the drone's video function, we need to incorporate the OpenCV library.

OpenCV (Open Source Computer Vision Library) is an image processing library where all algorithms are provided in C++. However, it also has bindings for Python, Java and C, so functionality and fundamentals are applicable across the board. Bindings generators create a bridge between C++ and other languages. This allows

users to call C++ functions from Python and other languages. It was initially launched in 1999 by Intel. OpenCV2 was then introduced in 2009, and in December 2016 OpenCV3 was released.

OpenCV is made up of more than 2500 optimized algorithms, including a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to:

- detect and recognize faces
- identify objects
- classify human actions in videos
- track camera movements
- track moving objects
- extract 3D models of object
- produce 3D point clouds from stereo cameras
- stitch images together to produce a high-resolution image of an entire scene
- find similar images from an image database
- remove red eyes from images taken using flash
- follow eye movements
- recognise scenery and establish markers to overlay it with augmented reality, etc.

OpenCV was built to provide as a free and open infrastructure for computer vision applications. It is important to note, it is not a standalone programme. It is necessary to write all of the software that calls on it.

In order to use the drones videos, OpenCV2 is used for decoding, preparing and analyzing the images. In simple terms the way the video-stream works on the AR-Drone is by using "frames". The I-Frame captures and stores the complete image, like a photo. The P-Frame, then only stores the difference in the next image to the last frame and so on.

The PS_Drone API comes with OpenCV. Therefore it was not necessary to import OpenCV separately into our python code.

There are extensive tutorials online for OpenCV2 and Python (https://opencv-python-tutroals.readthedocs.io/en/latest/)

10. Start Up Code

For the drone to perform effectively, we have included code at the start of each of our programmes. This sequence allows the drone to start up in a 'healthy' state. This code performs a number of tasks prior to executing the main programming.

```
# -*- coding: utf-8 -*-
```

We start each of our programmes with the line above. While this looks like a comment, it is however an encoding declaration (Anon., n.d.) (https://docs.python.org/3/reference/lexical_analysis.html#encoding-declarations). This expression names the encoding of the source code file. This line of code must be on its own line and either the first or second line of code. If it is the second the first line of code must be a comment.

Initially we import the PS-Drone API. This is a fully featured API which is written in Python for the AR-Drone 2.0 specifically. The PS-Drone-API is initialised and assigned to drone.

```
import ps_drone                    #Imports the PS-Drone-API
```

The *startup()* method is called to connect to the drone, it configures some basic processes and it also activates the NavData.

```
drone = ps_drone.Drone()           #Initials the PS-Drone-API
drone.startup()                    #Connects to the drone and starts subprocesses
```

When the lights are red, the *drone.reset()* method is used to ensure that the LEDs on the drone are green prior to running any programmes. We now know that the drone is ready to fly!

```
drone.reset()                    #Sets drone's LEDs to green when red
```

As a full battery will only give you 15 minutes approximately of flight, it is important to keep track of the battery status, prior to each programme being run. A check is performed on how much battery power is left, *getBattery()*, which is printed to the screen before the drone moves. Otherwise, if the battery is empty, then the programme will exit.

```
while (drone.getBattery()[0]==-1):
    time.sleep(0.1)              #Reset completed

print "Battery: " + str(drone.getBattery()[0]) + "%" + str(drone.getBattery()[1])
if drone.getBattery()[1]== "empty": #Check Battery status
     sys.exit()
```

The drone provides two modes for sending its NavData, demo and full mode. We have set the code to *useDemoMode(true)*, which will send 15 NavData per second. NavData are sent as blocks, including the sensor-measurements and status information. There are 28 packages which contain a specific set of values. The package names are the same as in Parrot's SDK.

```
drone.useDemoMode(True)          #Use demo mode
drone.getNDpackage(["demo"])     #Packets, which shall be decoded
time.sleep(0.5)                  #Give it some time to fully awake
```

The *time.sleep()* is used to give the drone time to perform and complete whatever code is executed in the previous line. The time given depends on the task to be performed.

11. First Flight Code

For our flight we just wanted to experiment with some of the different commands available. We used the suggested start up sequence and included a number of commands to see how they would work. Some of the commands used were:

```
drone.takeoff()
time.sleep(7.5) #Drone starts
#Gives the drone time to start
```

This made the drone take off and then hover there for 7.5 seconds giving it time to start up and to prepare for the next command.

```
drone.moveForward()
time.sleep(2)
drone.stop()
time.sleep(2) #Drone flies forward...
#... for two seconds
#Drone stops...
#... needs time
```

This commanded the drone to fly forward for two second and then stop.

We used a similar command to get the drone to fly backwards, however we changed the speed at which the drone travelled at. The following code will get the drone to move backward at a speed of 0.25 of the default speed.

```
drone.moveBackward(0.25)
```

This is a very useful command as we can set the speed for each movement the drone makes. However, if there are lots of movements and we want the speed the same the whole way we can just set the speed for the whole journey:

```
drone.setSpeed(1.0)
```

To complete our first flight we got the drone to land using the following command:

```
33  drone.land() #Drone lands
```

12. Speed Code

In this project, it was important from the beginning to be able to control, measure and manipulate the speed of the drones' movement in any direction. When testing this, we wrote a piece of code called Speed.py which allowed us to test this locally (unit testing).

Like most of the other pieces of code written for the drone, the basic start up code was used followed by the main program code. For this code the demo package was decoded for use. The Speed.py program makes the drone fly forward for 3 seconds and prints the speed on screen every 0.2 seconds for the user to see. The speed that is printed is the estimated speed in X in mm/s. What this means is that the drone is moving in a forward direction or in reverse (pitch). There is also the ability to measure the yaw and roll of the drone and printing the results as the estimated speed in Y or Z mm/s.

To access the drone's speed configurations the code used is:

Drone.NavData['demo'][4][0]

In order to print this result, the configuration must be converted to a string using a wrapper str(code). The main program code can be seen in *Figure 1.5* below.

```
gliding = False
time.sleep(1)

start = time.time()
end = time.time()


drone.moveForward(0.2)
while (end - start < 3):       #While loop that lasts 3 seconds
    print "Estimated speed in X in mm/s: " +str(drone.NavData['demo'][4][0]) # Estimated speed in X in mm/s
    time.sleep(0.2)               #Print this value every 0.2 second(s)
    end = time.time()           #End value is reset every iteration to get the length of flight

drone.stop()
time.sleep(1)
drone.land()
```

Figure 1.5, Speed.py (Main Program)

13. Tag Detection

Tag detection with the AR Drone was an important aspect of this project. It incorporated the drone's front and ground cameras and also used OpenCV2 to analyse the images through the video feed to identify the roundel tags. The PS Drone API uses OpenCV2 software for tag detection.

The code for this test was *firstTagDetection.py*, an example made available by the maker of PS Drone API online. The marker used for this test was the oriented roundel which can be found at the end of this report. This was used to test the drone's ability to analyse images through either camera. However, when the code was first downloaded there were issues with detecting markers. This was due to the settings in detect: *detect_type*. When setting the configuration for detect type in the code, we discovered that there was a mistake. The detect type was set to 0 which in turn, disables the drone's ability to detect any markers. This was then changed to 10 as this allows for universal detection. By doing so this allowed for multiple tags to be detected.

Next, the camera in use had to be selected. The camera we chose for detecting the tags was the ground camera. This was chosen as we wanted the drone to land on the tag as a landing pad. Therefore, the *detections_select_h* was set to 0 as this turns off detection with the front camera. The *detections_select_v* configuration was set to 128, which detects the oriented roundel with the ground camera.

Variable names are given to certain 'characteristics' of the oriented roundel tag. These include horizontal and vertical position, the number of tags found, the distance the tag is from the drone and the orientation of the tag. By using a loop, if the tag is detected these characteristics are printed on the screen. If there is no tag detected, the screen will display "no tag detected". The tag is marked as detected when the distance from the drone is less than 300 mm. The full main program can be seen in *Figure 1.6* below.

```
# Setting up detection...
# Shell-Tag=1, Roundel=2, Black Roundel=4, Stripe=8, Cap=16, Shell-Tag V2=32, Tower Side=64, Oriented Roundel=128
drone.setConfig("detect:detect_type", "10")                    # Enable universal detection
drone.setConfig("detect:detections_select_h", "0")             # No detection with front-camera
drone.setConfig("detect:detections_select_v", "128")           # detection with ground cam
CDC = drone.ConfigDataCount
while CDC == drone.ConfigDataCount:    time.sleep(0.01)        # Wait until configuration has been set

# Get detections
stop = False
while not stop:
    drone.moveForward(0.1)

    NDC = drone.NavDataCount
    while NDC == drone.NavDataCount:   time.sleep(0.01)
    if drone.getKey():                 stop = True
    # Loop ends when key was pressed
    tagNum = drone.NavData["vision_detect"][0]                 # Number of found tags
    tagX =   drone.NavData["vision_detect"][2]                 # Horizontal position(s)
    tagY =   drone.NavData["vision_detect"][3]                 # Vertical position(s)
    tagZ =   drone.NavData["vision_detect"][6]                 # Distance(s)
    tagRot = drone.NavData["vision_detect"][7]                 # Orientation(s)


 # Show detections
 if tagNum:
    for i in range (0,tagNum):
        print "Tag no "+str(i)+" : X= "+str(tagX[i])+"  Y= "+str(tagY[i])+"  Dist= "+str(tagZ[i])+"  Orientation= "+str(tagRot[i])
        if (tagZ[i] <= 300):
            drone.stop()                   #Drone stops...
            time.sleep(2)                  #... needs, like a car, time to stop
            drone.land()
            stop = True
 else:   print "No tag detected"
```

Figure 1.6, firstTagDetection.py

14. Altitude Code

The program Atitude.py was created in order to set the altitude of the drone and test if the drone was flying at that height. This was carried out so that the drone could be set to a certain height if needs be.

For this test we set the max altitude to approximately 2 meters. The default altitude for the drone is 75cm. The code uses the same standard battery check at the beginning in order to test if the drone has enough battery life to carry out the given commands. Once this is done we set the demo mode to false as we needed the NavData packets fast for this program. The NavData packages that we decoded here were “demo” and “altitude”.

```
drone.getNDpackage(["demo","altitude"])
```

The drone then takes off and using *'drone.setConfig'* the max altitude was set to 1999mm (altitude measurements are in mm for the drone).

```
drone.setConfig("control:altitude_max","1999")
```

The drone was then programmed to fly upwards for two seconds. This was to test that the *maxAltitude* setting was working and so that we could set this as the new height for the program. Gliding was set to false and was the drone was given 1 more second to make sure everything was given a chance to get started. A simple timer loop was then used in order to send the altitude measurements back to the laptop and display them with a print statement. The loop was set to 30 seconds and to display the altitude every 1 second. The end value is reset at the end of every iteration in order to calculate the loop time every time it enters the loop.

```
while (end - start < 30):       #While loop that lasts 30 seconds
    print "Altitude: " + str(drone.NavData["demo"][3])
        #Prints the altitude value
```

After 30 seconds the loop is exited and the drone lands. The program is then terminated.

15. Height

We wanted to try and get the drone to measure the height of a room. We planned to do this by making the drone fly upwards slowly and using three variables which get measurements of the altitude every half a second.

```
alt1 = drone.NavData["pressure_raw"][0]
print "\nAltitude: " + str(drone.NavData["pressure_raw"][0])#Prints the altitude value
time.sleep(0.5)
alt2 = drone.NavData["pressure_raw"][0]
print "\nAltitude: " + str(drone.NavData["pressure_raw"][0])#Prints the altitude value
time.sleep(0.5)
alt3 = drone.NavData["pressure_raw"][0]
print "\nAltitude: " + str(drone.NavData["pressure_raw"][0])#Prints the altitude value
time.sleep(0.5)
```

Our idea was that we would create a while loop that would run while none of the altitude variables were the same. When the drone would reach the roof the variables would be getting the same result indicating this would be the height of the room.

```
while alt1 != alt2 and alt2 != alt3:
```

While none of the variables were the same the drone would keep measuring the altitude of the variables. When they became equal to each other it would print the height of the room

```
print "\nHeight of room: " + str(alt3 + 12.7) + "cm"
```

We added 12.7 cm to cater for the height of the drone.

Despite our efforts this idea did not work as the drone is unable to measure its height relative to the ground. It only knows its height from where it took off. When it reached the roof it was unable to realise that it was static at the same altitude. It kept printing the altitude as if there was no roof and the drone continued its flight upwards. As a result we were unable to measure the height of a room.

16. Square Flight Code

Once we got the drone up and flying our first objective was to write a simple program to make the drone fly around in square and land back where it started. We thought this would be a simple program to get started with and could help us with more complex programs such as navigating around a room.

Before writing the code, the mechanics of getting the drone to fly in the shape of a square had to be thought about. We came up with two different strategies to get the drone to complete the objective.

The first strategy was to get the drone to complete the square while facing the same way. It would do this by moving the drone forward, right, backwards and left. The following functions are the most important:

```
drone.moveForward(0.5)          #Drone flies forward...
time.sleep(2)                    #... for 3 seconds
drone.stop()                     #Drone stops...
```

This will make the drone fly forward for three seconds at a speed of 0.25 of the default speed and then stop.

```
drone.moveRight(0.25)
time.sleep(3)                    #... for two seconds
drone.stop()                     #Drone stops...
```

This instruction will cause the drone to carry out the same task however this time the drone will move to its right for three seconds. Two more commands are used to make the drone move backwards and then left which in theory should return the drone to its starting point.

```
drone.land()                     #Drone lands
```

The *drone.land()* function will then get the drone to land at its final destination.

The second strategy was to get the drone to move forward using the *drone.moveForward()* method and then getting the drone to turn 90 degrees to get the drone to face right and the drone could move forward again. We could get the drone to turn using the following method:

```
drone.turnAngle(70,1)           # Drone turns right 70 degrees at full speed
drone.stop()                    # Drone stops
time.sleep(0.3)                 # time to stop
```

This would get the drone to turn a total of 90 degrees at a speed of 0.25 of the default speed and then stop when the drone has turned. The drone can then be commanded to move forward again. These commands can be repeated in order to get the drone fly in the shape of the square.

Both strategies had their positives and negatives. The advantages of the first strategy of getting the drone the move forward, right, left etc were the time efficiency and accuracy. We knew that making the drone turn at each corner would take time. Also, the accuracy of each turn could be off at certain times. However, the obvious downside to this idea was that the drone would not be facing forward while travelling.

Our thoughts on the second strategy of using the *drone.turnangle()* command were the complete opposite of the first strategy. We knew that this strategy would be more time costly and may not be as accurate but the drone would be facing forward for the entire journey.

Taking the two strategies into account we decided to go with the second option for the simple fact that the drone would be facing forward while travelling. Although the first idea would be more efficient time wise, we thought it just wouldn't be practical not having the front camera of the drone facing forward. As a result, this is the idea we decided to start with to get the drone to fly in a simple square.

The following is a snippet of the code for our first attempt at flying in the shape of a square:

```
drone.takeoff()                 #Drone takes off
time.sleep(1)                   #Gives the drone time
drone.moveForward(0.5)          #Drone flies forward...
time.sleep(2)                   #... for 3 seconds
drone.stop()                    #Drone stops...
time.sleep(2)                   #... for 2 seconds, gives it time
drone.turnAngle(90,0.5)         #Drone turns right 90 degrees at
time.sleep(1.5)                 #... for one and a half seconds
drone.stop()                    #Drone stops
```

We repeated this code to allow the drone to complete the square. In theory we thought this would work perfectly however this was not the case. We encountered a number of problems, it was moving forward and turning but the drone was not landing anywhere near the spot it took off in the first place.

We took videos of the flights and analysed them to see where the flight was going wrong. From our analysis, we felt the problem was arsing from the fact that the drone would drift a little bit each time it would make a turn. It may only have been a couple of centimetres each time but over the course of the flight this could put the drone off quite a lot by the time it reaches the end of the flight. Upon further reading and research we found the following the line of code:

```
gliding = False
```

This improved the quality of the flight greatly as it would erase some of the drifting we were experiencing. Although there was a big improvement with flight it still wasn't right. We thought this might be down to the quality of the drone or some damage that may have been previously experienced.

As a result, we made a few tweaks to our code such as changing some of the angles at each turn to account for the drifting. So after some trial and error with the turning angles we got the drone to fly in the shape of a square and land roughly in the same area it took off.

17. Area Code

We wanted to write a program that would calculate the area of the square that drone has travelled in. To do this we had to figure out the distance the drone had travelled. In order to find the distance, we needed the speed of the drone and the time the drone flies for (*Distance = Speed * Time*).

To get the time we used a timer to check the time the flight was taking using the *time.time()* function.

To get the speed we can call the "demo" package and using the values [4] and [0] we can find the Estimated speed in mm/s when flying forward or backward.

```
speed = drone.NavData['demo'][4][0]
```

Then to calculate the distance, we can just multiply the speed (mm/s) of the drone by the time taken for the drone to travel that distance. For more accuracy, we took measurements every fifth of a second. So, we divided the speed (mm/s) by 5 and then multiplied it by 0.2 (fifth of a second).

```
speed = drone.NavData['demo'][4][0]
fifth = speed/5
distx  = distx + (fifth * 0.2)
```

Then in order to calculate the area of the square we multiplied the distance of two sides of the square the drone travelled

```
area = distx * disty
```

18. Final Code

The final code for the project demonstration comprises of multiple tests that were carried out throughout the year. By combining these and testing the final demo repeatedly, in order to perfect the flight and tag detection accuracy, we were able to complete the code. The code itself includes the start-up code that has been used in all of our tests previously. The packets that are decoded in the code are the demo, altitude and vision_detect packages.

There are two timers used in the final code. The first timer is the X timer. This timer is used for the first part of the rectangle the drone will measure. By using the same algorithm as in the area code seen in this report, we were able to calculate the distance for X. The X value is then stored as X and the distance is printed on the screen formatted to 2 decimals. This is done for the Y value also. By multiplying both of these the area is calculated and printed when the drone lands.

The *tagDetected()* method is called when a tag has been detected. This method contains a print statement which prints all of the characteristics of the tag that has

been found. An if statement is then used to see if the tag is less than 300mm away from the drone. If this is true, the drone lands on the marker and exits the code.

One issue that was prominent throughout the testing of this code was the 90 degree turn angles for the drone. Because of various limitations the drone does not complete 90 degree turns in an accurate manner and due to drift and various external factors that may cause this, different angle turns were made throughout the flight. This was done in order to correct the flight path of the drone and keep the flight path straight in order to land at the take-off position.

19. Conclusion

19.1 Limitations

During the course of the project we encountered a couple of problems. Overall the quality of the drone proved to be troublesome at times. It struggled to carry out some commands accurately and there was a large amount of error for certain tasks. This could have been down to a number of reasons such as damage to the drone or the standard of the drone.

- One of the main problems we experienced was the drifting of the drone. The drone tended to veer of track at times. We experienced this mostly when trying to get the drone to fly in a square. We would run the code a number of times but would end up with different results. We had to adjust our code and cater for the gliding we were experiencing.
- The drone cover also seemed to cause some trouble. The slightest damage caused to the cover seemed to affect the flight. The accuracy of the flight seemed to deteriorate over time as a result of damage to the cover.
- The battery life was also a hindrance at times as we had minimal time to test our code. The length of the battery life varied but typically we found we had about 10 minutes of testing our code. We also found that the performance of the drone deteriorated as the battery got lower.

- The quality of the bottom camera was quite poor. At times the drone would fail to locate markers. The quality of the camera hindered the ability of the drone to detect certain markers and objects.

19.2 Future Drone Uses

- This project could be further developed and up scaled to be used on much bigger projects. Our project could be used on a much larger scale than what was displayed for this particular project. It could be used to measure the areas of much bigger areas. The use of the drone to measure the area of a particular area could be a lot more practical than measuring the area by hand. Using the drone could prove to be much more efficient and could save time on a large scale project.
- The use of detecting markers could then be used to avoid obstacles and help avoid crashing. Instead of detecting markers the drone could be programmed to detect certain objects and then move in a certain way to avoid them. In the case of this project the drone will land when it detects a marker but in an up scaled version it could be programmed to move left or right to avoid it.

20. Appendix 1

```
# -*- coding: utf-8 -*-
##### Suggested clean drone startup sequence #####
import time, sys
import ps_drone                 #Imports the PS-Drone-API

drone = ps_drone.Drone()        #Initials the PS-Drone-API
drone.startup()                 #Connects to the drone and starts subprocesses

drone.reset()                   #Sets drone's LEDs to green when red

while (drone.getBattery()[0]==-1):
    time.sleep(0.1)             #Reset completed

print "Battery: " + str(drone.getBattery()[0]) + "%" + str(drone.getBattery()[1])
if drone.getBattery()[1]== "empty": #Check Battery status
     sys.exit()

drone.useDemoMode(True)         #Use demo mode
drone.getNDpackage(["demo", "altitude", "vision_detect"])    #Packets, which shall be decoded
time.sleep(0.5)                 #Give it some time to fully awake


                    ###### Main Program ######
drone.takeoff()
while drone.NavData["demo"][0][2]:
    time.sleep(0.1) #Still in landed-mode

gliding = False
time.sleep(1)                   #Gives the drone time to

xStart = time.time()          # Start timer for x
xEnd = time.time()            # End timer for x
x = 0                         # Set x = 0, this will be reset later for the Distance
y = 0                         # Set y = 0, this will be reset later for the Distance



# Setting up detection...
# Oriented Roundel=128
drone.setConfig("detect:detect_type", "10")                    # 10 = Enable universal detection
drone.setConfig("detect:detections_select_h", "0")             # Turn off detection for front
camera
drone.setConfig("detect:detections_select_v", "128")           # Detect "Oriented Roundel"
with ground-camera
CDC = drone.ConfigDataCount
while CDC == drone.ConfigDataCount:    time.sleep(0.01)         # Wait until configuration
has been set
```

```
# tagDetected function that will be called if a tag has been detected
def tagDetected ():
    for i in range (0,tagNum):
        # Print the tag number and coordinates
        print "Tag no "+str(i)+" : X= "+str(tagX[i])+"  Y= "+str(tagY[i])+"  Dist= "+str(tagZ
        [i])+"  Orientation= "+str(tagRot[i])
        # If the tag is closer than 300mm the drone will land
        if (tagZ[i] <= 300):



            drone.stop()                    #Drone stops...
            drone.land()
            sys.exit()


NDC = drone.NavDataCount
while NDC == drone.NavDataCount:    time.sleep(0.01)

# Drone flies forward at a speed of 0.2
drone.moveForward(0.2)

while (xEnd - xStart < 4):        #While loop that lasts 4 seconds

    tagNum = drone.NavData["vision_detect"][0]                # Number of found tags
    tagX =   drone.NavData["vision_detect"][2]                # Horizontal position(s)
    tagY =   drone.NavData["vision_detect"][3]                # Vertical position(s)
    tagZ =   drone.NavData["vision_detect"][6]                # Distance(s)
    tagRot = drone.NavData["vision_detect"][7]                # Orientation(s)

    print "Estimated speed in X in mm/s: " +str(drone.NavData['demo'][4][0]) # Estimated
    speed in X in mm/s
    speed = drone.NavData['demo'][4][0]
    fifth = speed/5                 # Divide speed by 5 to allow for acceleration at a fifth
    of a second
    x = x + (fifth * 0.2)           # Calculate the distance every 0.2 of a second and add
    to value x
    time.sleep(0.2)                 # Print this value every 0.2 second(s)
    xEnd = time.time()              # End value is reset every iteration to get the length
    of flight
    if tagNum:                      # if a tag is detected, the tagDetected() method is called
        tagDetected()
# Convert x to metres
x = x/100

# x distance is printed on the screen and formatted to 2 decimals
print "\nDistance: " + str('{0:.3g}'.format(x)) + " Metres"
```

```
drone.stop()                    # Drone stops
time.sleep(1)                   # Given time to stop
drone.turnAngle(90,1)           # Drone turns right 90 degrees at full speed
drone.stop()                    # Drone stops
time.sleep(1)                  # Given time to stop

yStart = time.time()            # y timer is started
yEnd = time.time()              # y end timer for when the timer is finished

drone.moveForward(0.2)          # Drone moves forward at a speed of 0.2
while (yEnd - yStart < 2.5):    # While loop that lasts 2.5 seconds

    tagNum = drone.NavData["vision_detect"][0]              # Number of found tags
    tagX =   drone.NavData["vision_detect"][2]              # Horizontal position(s)
    tagY =   drone.NavData["vision_detect"][3]              # Vertical position(s)
    tagZ =   drone.NavData["vision_detect"][6]              # Distance(s)
    tagRot = drone.NavData["vision_detect"][7]              # Orientation(s)

    print "Estimated speed in X in mm/s: " +str(drone.NavData['demo'][4][0]) # Estimated
    speed in X in mm/s

drone.stop()                    # Drone stops...
time.sleep(2)                   # time to stop
drone.turnAngle(70,1)           # Drone turns right 70 degrees at full speed
drone.stop()                    # Drone stops
time.sleep(0.3)                 # time to stop

xStart = time.time()            # Timer is used again and reset to the present time
xEnd = time.time()              # End timer is reset again

drone.moveForward(0.2)          # Drone moves forward at 0.2 speed
while (xEnd - xStart < 2.5):    # While loop that lasts 2.5 seconds

    tagNum = drone.NavData["vision_detect"][0]              # Number of found tags
    tagX =   drone.NavData["vision_detect"][2]              # Horizontal position(s)
    tagY =   drone.NavData["vision_detect"][3]              # Vertical position(s)
    tagZ =   drone.NavData["vision_detect"][6]              # Distance(s)
    tagRot = drone.NavData["vision_detect"][7]              # Orientation(s)

    time.sleep(0.1)              # Every 0.1 seconds
    xEnd = time.time()           # End value is reset every iteration to get the length of
    flight

    if tagNum:
        tagDetected()
```

```
drone.stop()                    # Drone stops...
time.sleep(2)                   # Time to stop
drone.turnAngle(110,1)          # Drone turns right 110 degrees at full speed
drone.stop()                    # Drone stops
time.sleep(0.3)                 # Time to stop

yStart = time.time()            # Timer starts
yEnd = time.time()              # Timer end variable


drone.moveForward(0.2)          # Drone moves forward at 0.2 speed
while (yEnd - yStart < 2.5):    # While loop that lasts 2.5 seconds

    tagNum = drone.NavData["vision_detect"][0]              # Number of found tags
    tagX =   drone.NavData["vision_detect"][2]              # Horizontal position(s)
    tagY =   drone.NavData["vision_detect"][3]              # Vertical position(s)
    tagZ =   drone.NavData["vision_detect"][6]              # Distance(s)
    tagRot = drone.NavData["vision_detect"][7]              # Orientation(s)

    time.sleep(0.1)             # Print this value every 1 second(s)
    yEnd = time.time()          # End value is reset every iteration to get the length of
    flight

    if tagNum:
        tagDetected()           # If a tag is detected the tagDetected() function is called


drone.stop()                    # Drone stops
time.sleep(2)                   # Time to stop
drone.turnAngle(110,1)          # Drone turns right 110 degrees at full speed
drone.stop()                    # Drone stops
time.sleep(0.5)                 # Time to stop


drone.land()                    #Drone lands

print 'Drone has landed.'

########### Program Finished ############
```

Bibliography

Anon., n.d. *Python Documentation.* [Online]
Available at: https://docs.python.org/3/reference/lexical_analysis.html#encoding-declarations

Atom. 2016. *Atom*. [ONLINE] Available at: https://atom.io/. [Accessed 1 January 2017].

Chao, W., 2014. Vision-based Autonomous Control and Navigation of a UAV.

J.Phillip de Graaf. 2014. *The PS-Drone-API. Programming a Parrot AR.Drone 2.0 with Python - The easy way*. [ONLINE] Available at: http://www.playsheep.de/drone/. [Accessed 26 February 2017].

OpenCV-Python Tutorials. 2013. *Install OpenCV-Python in Windows*. [ONLINE] Available at: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_setup_in_windows/py_setup_in_windows.html. [Accessed 2 March 2017].

Phillip de Graaf, J., 2014. The PS-Drone-API. Programming a Parrot AR.Drone 2.0 with Python – The easy way

Piskorski, S., Brulez, N., Eline, P. and D'Haeyer, F., 2012. Ar. drone developer guide. *Parrot, sdk*, *1*.