MONASH University

Computer Vision using a Quadrotor UAV

Project Report

Version: 1.0

Name: Steven Clementson

ID: 21467811

Supervisor: Dr. Wai Ho Li

TRC4000 17/10/2012

MONASH University

Department of Electrical and Computer Systems Engineering

TRC4001 Final Year Project, Semester 2, 2012 — Steven Clementson

Computer Vision Controlled Quadrotor UAV

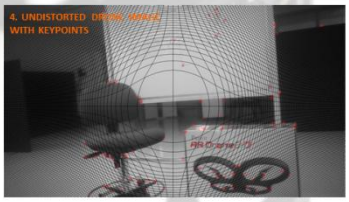
The Parrot AR.Drone 2.0 is a commercially available Quadrotor Unmanned Aerial Vehicle (UAV). Custom software has been added to the system to enable it to fly autonomously in an environment being viewed by a Kinect (RGB-D) camera.

PROCEDURE

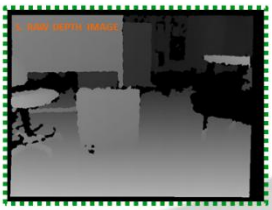
1. The default program provides flight stability via optical flow analysis on the downward camera.

2. The custom program runs alongside this on the drone, accessing the front camera.

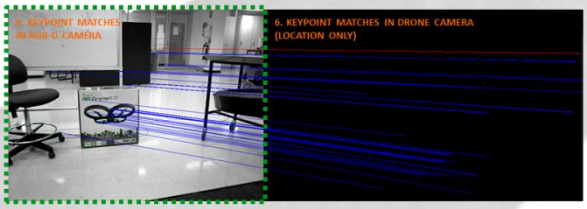
3. Camera frame is undistorted according to a quintic camera model.



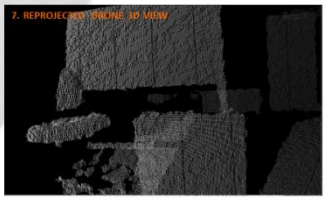
4. Keypoints and feature descriptors are calculated and sent to the server via WIFI.



5. Keypoints and descriptors are calculated from the RGB-D camera frame.



6. The drone pose relative to the RGB-D camera is calculated and sent to the drone via WIFI.

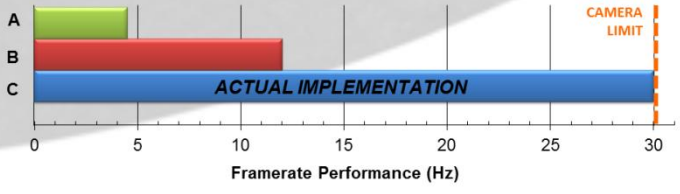


7. The drone uses the relative pose information to calculate desired motion. Motion commands are sent to the default control program via UDP.

IMPLEMENTATION STRATEGY

Options investigated include:

A. Complete remote processing - requires sending camera frames via WIFI
B. Replacement of default drone program - requires manual stabilisation
C. Addition of 'parasite' program to drone that commands the default program



FUTURE APPLICATIONS

The next step involves pre-mapping a room so that the drone can operate without the need for a fixed RGB-D camera. The drone could then be used to automatically patrol buildings as a security measure or investigate potentially dangerous buildings after earthquakes or nuclear accidents. Battery technology is the limiting factor, as the drone can fly for no more than 15 minutes per charge.

ACKNOWLEDGEMENT

The localisation aspect of the project was implemented with significant assistance from Winston Yii.

Table of Contents

1.0 Summary

The Parrot AR.Drone and AR.Drone 2.0 are quadrotor Unmanned Aerial Vehicles (UAVs) intended to be controlled by a smartphone device via a WIFI link. This project involves exploring the capabilities of the devices for use in common computer vision tasks.

The AR.Drone was first investigated by measuring the latency of sending video frames to a remote computer for processing. The median latency of 141ms and frame loss rate of 23% were deemed to be unacceptable for advanced computer vision and control algorithms. Next, the possibility of replacing the default drone program with custom code was investigated and found to be a viable option.

The drone video and sensor data were attained and motor control was achieved by manually tuning PID controllers. The disadvantage with this approach was found to be the loss of stabilisation afforded by optical flow analysis on the downward facing camera.

Next a simple object following application was created to compare the two control approaches. The lag of the drone behind the pendulum being followed when adjusting yaw angle was measured and found to be 3.4 times greater for off-board processing as compared to onboard processing. When altitude was adjusted the lag was 1.14 times greater. Hence it was decided that as much computer vision computation as possible should be performed onboard the drone.

When the AR.Drone 2.0 was attained a custom optical flow stabilisation algorithm was implemented to counter the flight stability issues. Due to the implementation being CPU-based rather than making use of the DSP, this ran at only 12.2Hz whereas the camera is capable of 60FPS. This was regarded as an unusable control method and hence ways to use the default stabilisation were investigated.

An approach was discovered whereby it is possible to gain exclusive access to the drone front camera (for use in computer vision tasks) whilst also allowing the default program to run and control the drone. This approach sees the custom program send UPD commands to the default program to control the drone's motors, whilst running onboard the drone itself.

The latency associated with sending commands this way was measured and found to be very low (0.275ms). The latency for the motors to respond to a command sent this way was also measured and found to be between 14 and 17ms. This result confirmed that this control approach was acceptable and hence a localisation application was implemented.

Localisation first required an accurate model of the drone camera, so this was created using a calibration toolbox in MATLAB. Next the FAST-9 keypoint algorithm and rHIPS descriptors were extracted from camera frames and it was found that this could be completed at or above the camera frame rate of 30FPS. Localisation against an AUSU Xtion PRO Live camera was achieved with the help of Winston Yii where the processing of the Xtion camera data was completed on a remote computer.

The pose of the drone was received via WIFI and used to generate control commands for the drone movement. This allowed the drone to navigate a series of waypoints or remain in a location relative to the Xtion camera as it was moved manually (Visual Servoing). Future work will involve expanding the system to allow it to navigate a fully mapped room and perform more or all processing onboard the drone.

2.0 Introduction

This final year project was completed as part of the requirement for the Bachelor of Mechatronics Engineering and was undertaken over the course of an 8 month period during 2012. The project was completed under the requirements set by the Monash University Faculty of Electrical and Computer Systems Engineering. The project supervisor was Dr. Wai Ho Li.

The project involved exploring two Unmanned Aerial Vehicles (UAVs); the AR.Drone and AR.Drone 2.0 created by Parrot. These are low cost ($200-350), commercially available platforms that are intended to be used as toys and controlled by a smartphone through a WIFI connection. The first half of the project involved the AR.Drone whilst the second half focussed on the AR.Drone 2.0, as it was released in June 2012.

The goals of the project were to implement computer vision applications using the drone platforms. The specific requirements were:

1. A literature review of previous work in the field of computer vision controlled quadrotors.
2. The applications should see that the device operates autonomously with the exception of an 'emergency stop' command from the user.
3. That the device can perform localisation in a mapped environment.
4. That the device can avoid obstacles and environment boundaries.

To achieve these goals, it was first necessary to explore many elements of the drone platforms to establish their capabilities with regard to computer vision. The application implemented on the original AR.Drone involved ascertaining the need for local processing as opposed to transferring camera image data to a server computer for processing. The AR.Drone 2.0 was tested to determine the optimal control approach and a pre-existing localisation algorithm was implemented to demonstrate the platform capabilities.

3.0 Background

Much work has been completed in the area of quadrotor control using computer vision so this review will focus on three key areas; work completed using the platforms created by Parrot, computer vision localisation for quadrotors and real-time onboard applications. Only GPS denied environment applications will be considered.

3.1 Previous Work using the AR.Drone

E. Deligne was one of the first to experiment with the AR.Drone and show the extent of the hacking possibilities for the device [1]. He establishes the TELNET access method, cross compilation using the CodeSourcery compiler and use of the V4L2 video driver for accessing the cameras - all of which were used extensively throughout this project.

C. Bills *et al.* were the first to perform autonomous control of the AR.Drone based on computer vision methods [2]. They use perspective cues from a single image to first classify the environment as corridor, stairs, open space or enclosed space. Navigation is performed using a Canny edge detector and Hough transform for corridors and stairs, whilst a simple exploratory routine is employed for other environments. In these cases the drone uses additional proximity sensors to avoid obstacles whilst searching for corridors and stairs. The authors followed the implementation approach endorsed by Parrot; using a remote computer for all processing. This limits the video resolution and only allows access to the downward camera as a picture in picture.

Researchers at the Exertion Games lab have advanced the AR.Drone into a jogging companion whereby the drone recognises a marker on the joggers t-shirt [3]. The drone then maintains a distance in front of the jogger as they run. This has obvious issues when the user runs toward an obstacle, as the drone has no ability to see in the direction it is flying and could also be dangerous to other joggers nearby.

Japanese researchers are working on an application whereby the drone is controlled by a blind user through head movements [4]. A PlayStation game controller is attached to the user's head and movements are registered by the controller's sensors and sent to the drone via WIFI. The drone state is reported back to the user through a braille display. This approach also runs on a remote computer and relies on additional sensors added to the drone.

N. Berezny and L. D. Greef use the drone to perform various computer vision tasks using OpenCV and the Robot Operating System (ROS) [5]. They perform all processing remotely and show that it is possible to have the drone follow blob objects – namely a ground based robot with a colourful marker. The authors noted the difficulty associated with limited camera resolution as provided by the drone via WIFI. A SURF-based localisation application was also developed, where the drone compared its camera view to a set of 36-48 images and attempted to find the closest match. This ran at 2-3FPS with an accuracy of around 80%. This result highlights the need for efficient and robust algorithms if computer vision is to be implemented onboard the drone.

A. Benini *et al.* implement an Extended Kalman Filter (EKF) to localise the drone [6]. They use the drone Inertial Measurement Unit (IMU) to calculate odometry and couple this with visual odometry. The computer vision algorithm involves searching for known markers placed at known locations. The EKF is used to correct for errors that build from the IMU odometry when a tag is recognised.

3.2 Computer Vision Localisation for Quadrotors

G. H. Lee *et al.* have implemented a Visual Simultaneous Localisation and Mapping (VSLAM) algorithm on an advanced quadrotor system that runs on a remote computer [7]. They use a downward facing camera to extract SIFT features and compute the essential matrix between two frames with the assistance of RANSAC. They use this approach to map and localise within an L-shaped set of QR-Codes placed on the floor. The system assumes all points viewed by the camera lie on the same plane for simplicity. V. Ghadiok *et al.* have implemented a nearly identical system that performs VSLAM on a downward camera using SIFT features, except that they have omitted the single plane assumption [8].

M. Bošnak and S. Blažič developed a very similar system which also performs VSLAM on a downward facing camera [9]. Instead of SIFT features however, they use glyph-based features together with tilt angles in a method called FastSLAM. This also relies on a set of markers placed in a series on the floor and is yet to be used to control a quadrotor.

C. Ratanasawanya *et al.* introduce a system that makes use of a forward facing camera on a quadrotor [10]. They extract features from a scene that contains only a rectangular marker to determine the location of the 4 marker corners. This is used to determine the pose relative to the marker and is performed on a remote computer.

It can be reasoned that for any system to be useful in a real-world environment, localising based on a downward facing camera is not possible. Indoor environments generally have floors that are uniform in appearance or have a repeating pattern that makes it impossible to localise against. As such applications that employ the use of markers or set patterns on the floor would require any working environment to be modified before the quadrotor could be used.

Localisation based on a forward facing camera is much more usable in a real environment however. As indoor environments are filled with furniture, posters and pictures on walls and other interesting features, it becomes possible for the UAV to operate in an unmodified environment. That is realm in which this project operated.

3.3 Real-time Onboard Applications

S. Erhard *et al.* were the first to present a quadrotor system that performed all processing onboard the device [11]. A Nokia N95 mobile phone was attached to an Ascending Technologies quadrotor which is capable of carrying a 300g payload. The drone first performs an exploration phase where a feature database is built with GPS data provided for ground truth. During the exploration phase the UAV compares features to generate a pose estimate. The system achieved a mean localisation error between 10.9 and 21.4m whilst operating at a frame rate of 1.5Hz which is clearly not useable in an indoor environment.

M. Achtelik *et al.* implemented VSLAM onboard a Pelican quadrotor that contained a 1.6GHz Atom processor [12]. This system ran at 10Hz by fusing IMU data with VSLAM information from a downward facing camera to achieve an RMS position error of 7cm whilst hovering. S. Shen *et al.* also run algorithms onboard using a 1.6GHz Atom processor [13]. Their system navigates multi-floor environments using a laser scanner and multiple cameras at 2Hz, providing an accuracy standard deviation below 10cm.

L. Meier *et al.* use a PIXHAWK Cheetah quadrotor platform to perform localisation, pattern recognition and obstacle avoidance onboard [14]. They use four cameras to provide two stereo views and perform processing on a Core 2 DUO processor computer that weighs 230g. Marker-based localisation is performed using a downward facing camera pair and obstacle avoidance is completed using a 3D occupancy grid generated from the forward facing camera pair.

In summary, it can be seen that there is great scope for advancement in the area of computer vision controlled autonomous quadrotors. Very little work has been done in the area of onboard processing, whilst the few that have entered this area are using devices that cost upward of $10,000. Work using the AR.Drone has been limited to remote processing, which limits the video quality and introduces large latencies.

Most localisation applications use markers in environments instead of natural features. This means the quadrotors cannot navigate everyday environments and often rely on downward facing cameras for stabilisation. Others use stereo cameras or laser range sensors for depth information, neither of which are available to the AR.Drone. This project has largely operated in unexplored territory and as such presents a useful contribution to the field.

4.0 AR.Drone 1.0

4.1 Platform Overview

The AR.Drone is a low cost device intended for use as a toy, so the hardware is limited in some ways compared to a more robust research platform. The following section gives an overview of the system capabilities.

4.1.1 Sensors

The drone contains several sensors that are used to estimate the position and movement of the device. The sensors included are outlined in figure 4.1 below.

Sensor	Parrot Specification	Notes
3 Axis MEMS Accelerometer	+/- 2g precision	Bosh BMA150
2 Axis MEMS Gyroscope	500 degree/second precision	InvenSense IDG500
1 Axis Yaw Precision piezoelectric Gyroscope	500 degree/second precision	Epson XV3700
Vertical Downward Camera	64° FOV 176X144 at 60 FPS	`/dev/video1`. V4L2 pixel format is YUV420
Horizontal Forward Camera	93° FOV VGA at 15 FPS	`/dev/video0`. V4L2 pixel format is YUV420
Ultrasound Sensor	Range 6m	Minimum height 30cm

Figure 4.1. AR.Drone sensor specifications [6, 15].

4.1.2 Computing Hardware

The processor of the drone is an ARM9 RISC 32bit 468MHz model (refer to section 10.1). This processor does not contain NEON capability. The drone also contains 128 MB of DDR RAM running at 200MHz.

4.1.3 Other Hardware

The AR.Drone uses WIFI b/g to connect to other devices and can be configured in ad-hoc or access-point mode. It was found that only access-point mode worked with android devices. The device also has one red and one green LED below each rotor that can be activated separately or together. There are exposed pins on the under-side of the drone that can be used as a USB or serial port. The USB port is disabled when the drone boots so custom drivers must be installed for its use [16].

4.1.4 Software

The drone runs Linux version 2.6.27.47-parrot. Busybox version 1.14.0 is installed to provide a set of tools to the user (refer to section 10.2). Code by Parrot that runs onboard the drone was cross-compiled using the Sourcery G++ Lite for ARM GNU/Linux version 2009q1-203 compiler [17]. The system version information was found by entering the following:

```
# cat proc/version
Linux version 2.6.27.47-parrot (aferran@Mykuntu)
(gcc version 4.3.3 (Sourcery G++ Lite 2009q1-203))
```

4.2 Control Approach

Parrot intends for users to write smartphone applications that interact with the AR.Drone via a WIFI connection [15]. This requires the video information from the front camera to be sent over a WIFI connection, which could generate a large latency in the system. The alternative approach is to hack into the drone system and insert a parasite program that runs onboard the drone itself. This method voids the warranty provided by the manufacturer but could provide a significant increase in system performance. To determine the best approach, first the latency of the WIFI connection was measured.

4.3 Platform Capability Tests

To determine the suitability of the device for use in computer vision applications, multiple properties of the drone were determined. The time delay between sending and receiving an image from the drone front camera via WIFI was measured using a pendulum and multiple views of the scene. The security level and system access abilities were also established through exploration of hacking abilities so that the possibility of running code on the drone was determined.

4.3.1 WIFI Latency

The experiment was set up using a simple pendulum that was in view of both the drone front camera and the laptop webcam. The drone was connected to from a simple program that saved camera frames from the drone with the time they were received as the filename. The computer also ran a simple OpenCV program that saved frames from the webcam in the same way. The pendulum position was then compared in each sequence to determine the time difference between pictures of the pendulum at the same position. Figure 4.2 depicts the experimental setup, whilst figure 4.3 shows two images taken at the same position in each view. The experiment was conducted in an environment with only one other WIFI network in range to minimise interference.

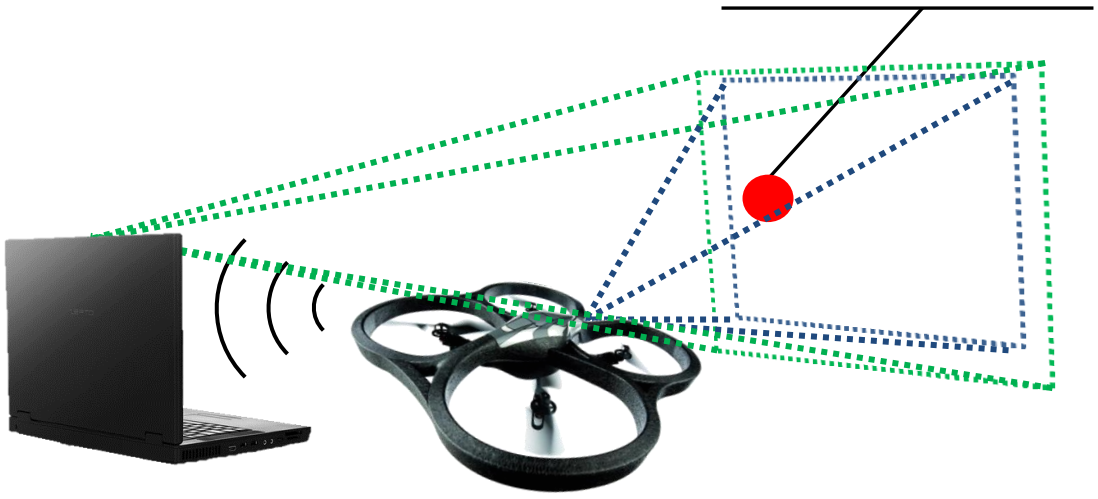


Figure 4.2. Experimental setup for WIFI latency test.

Figure 4.3. Drone camera view at time=9.704 (left) and webcam view at time=9.577 (right) where the pendulum is in the same position.

The median latency was calculated to be 141.17ms to transfer a colour image of size 320X240 pixels (.jpeg format) which was between 30,128 and 31,741 bytes. The latency results are summarised in figure 4.4.

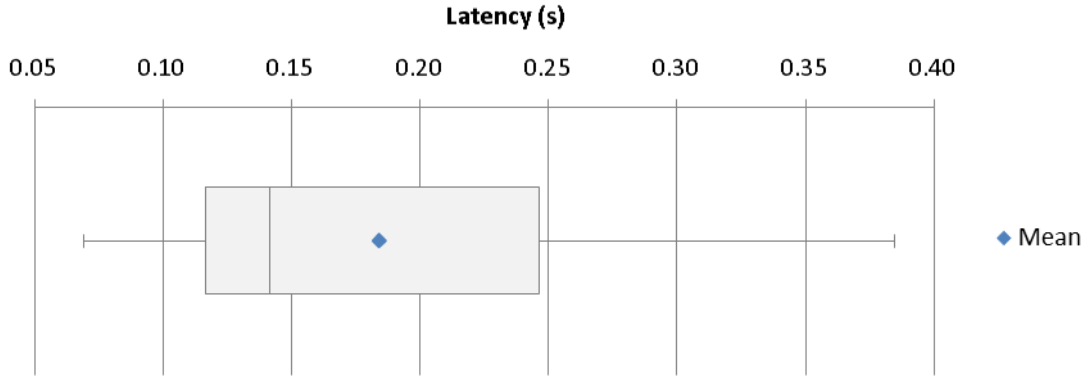


Figure 4.4. Distribution of WIFI image transfer time measurements for the AR.Drone in a low interference environment.

This latency was deemed to be significant considering no computer vision processing has taken place yet and commands must still be sent to the drone to complete the system loop. It should also be noted that the latency varied significantly with a range of 68.72 – 384.53ms. It was also found that a large percentage of frames were not received by the laptop, with a frame loss rate of 22.83% (42 of 184).

In light of this result it was determined that minimal information should be transferred via WIFI to and from the drone, and as such it would be preferable to perform computer vision tasks onboard the drone. This requires access to the drone operating system which is explored in section 4.3.2.

4.3.2 Hacking Possibilities

If code is to be run onboard the drone it must first be established that this is possible. The security protocols onboard the drone must be established to ascertain the extent to which it can be utilised.

4.3.2.1 System Access

The drone can be accessed in several ways. Firstly an FTP connection can be established in a number of ways. After connecting to the drone's WIFI network, the user can open an internet browser and navigate to either

```
ftp://192.168.1.1:21 or ftp://192.168.1.1:5551
```

Port 21 displays the contents of the `/data/video/` directory which is the default, whilst port 5551 connects to the `/update` directory which allows for firmware update files to be transferred.

The second option is to set up a permanent FTP folder which allows the user to copy, cut, paste and delete files directly to the `/data/video/` directory of the drone. This proved useful when a large number of output files were generated and needed to be copied from the drone.

The third option is to use the FTP command through a command prompt window which allows the standard functions to be utilised. This method also connects to the `/data/video/` directory by default.

Access to the drone Linux operating system is achieved through the use of a TELNET client. This is achieved by entering `telnet 192.168.1.1` into the command window. If the TELNET client is installed, it will open and the user will be greeted with the message:

```
BusyBox v1.14.0 (2012-06-01 16:35:43 CEST) built-in shell (ash)
Enter 'help' for a list of built-in commands.
```

The built-in commands available are listed in section 10.2. From here the user has root access to the drone. This allows the user to perform a variety of tasks such as moving, renaming, copying and deleting files, changing file permissions, displaying the contents of files and much more. The default program that controls the drone is called "program.elf" and it can be terminated by typing:

```
killall program.elf
```

Note that if the drone is flying it will crash land when this command is executed.

4.3.2.2 Program Compilation

Code must be compiled on an external computer and the final files copied across to the drone. This was done using an ARM cross-compiler called Sourcery G++ Lite for ARM GNU/Linux version 2009q3-67 by CodeSourcery.

4.3.2.3 Program Execution

Once the compiled executable program file is copied to the drone using one of the FTP methods described in section 4.3.2.1, the drone is connected to via TELNET. The program permissions must be modified to 'executable' using the command:

```
cd /data/video
chmod 777 program_name
```

It can then be run simply by entering:

```
./program_name
```

4.3.2.4 System Device Access

The drone's motors, cameras, sensors and LEDs can all be accessed from a program running on the drone once program.elf has been killed. The cameras can be accessed using the standard V4L2 camera driver protocol [18]. The motor board is sent commands to set the LEDs and motor voltages by opening the device `/dev/ttyPA1` [16]. This allows each motor to be controlled individually by sending a Pulse Width Modulation (PWM) signal to the motor.

The sensor data is read from the drone navboard (device `/dev/ttyPA2`). Before takeoff a "flat trim" operation must be performed to establish the offset of all sensors when the drone is on a level surface. Data from the navboard does not include the battery voltage – this can be monitored through the device `/dev/i2c-0`.

4.4 Computer Vision Application

To test the control response of the drone when performing a simple computer vision task, two approaches were applied for the same task. The task involved the drone following an object by adjusting the drone height and yaw only. The object was a red "blob" which was attached as a pendulum in front of the drone (refer to figure 4.5). The first approach was to receive front camera images from the drone via WIFI and process the images using OpenCV. The second application involved performing all processing onboard the drone.

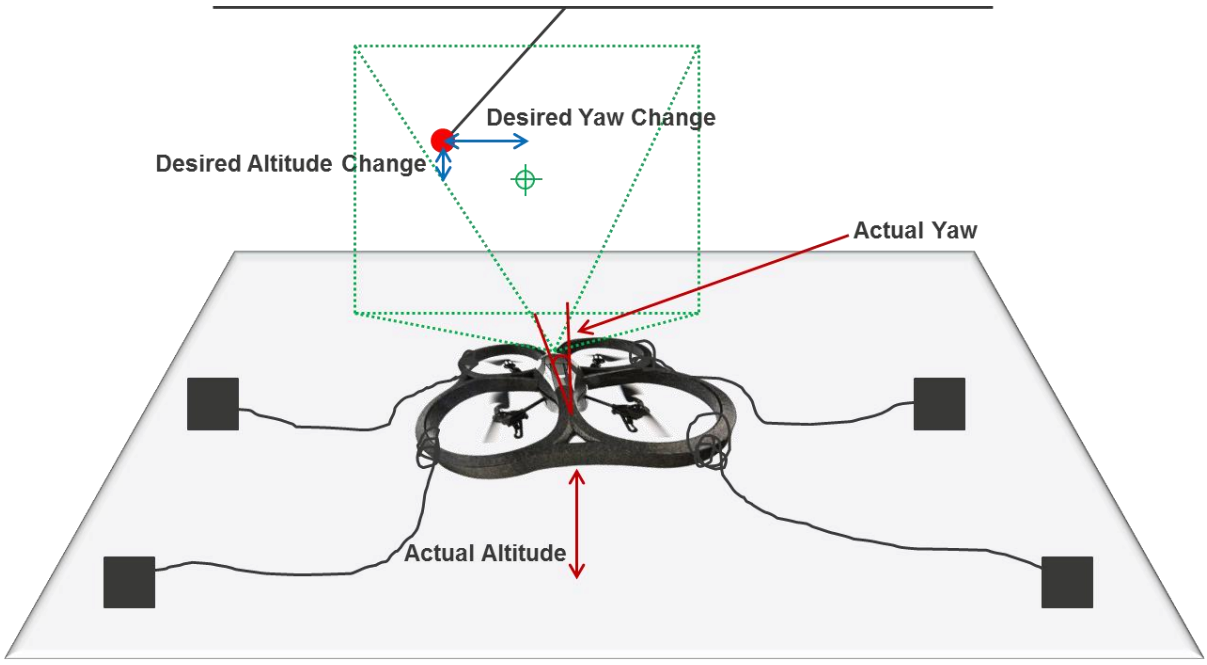


Figure 4.5. Experimental setup for the object following tests.

4.4.1 Remote Processing with OpenCV

This approach involved using the default drone program to send front camera images to the laptop via WIFI. The images were then processed using an OpenCV application that determined the centre of mass of the largest red blob in the frame. The blob offset from the centre of the frame was then calculated and a command was sent to the drone via WIFI to have it adjust to keep the blob in the centre of its view (refer to figure 4.6).

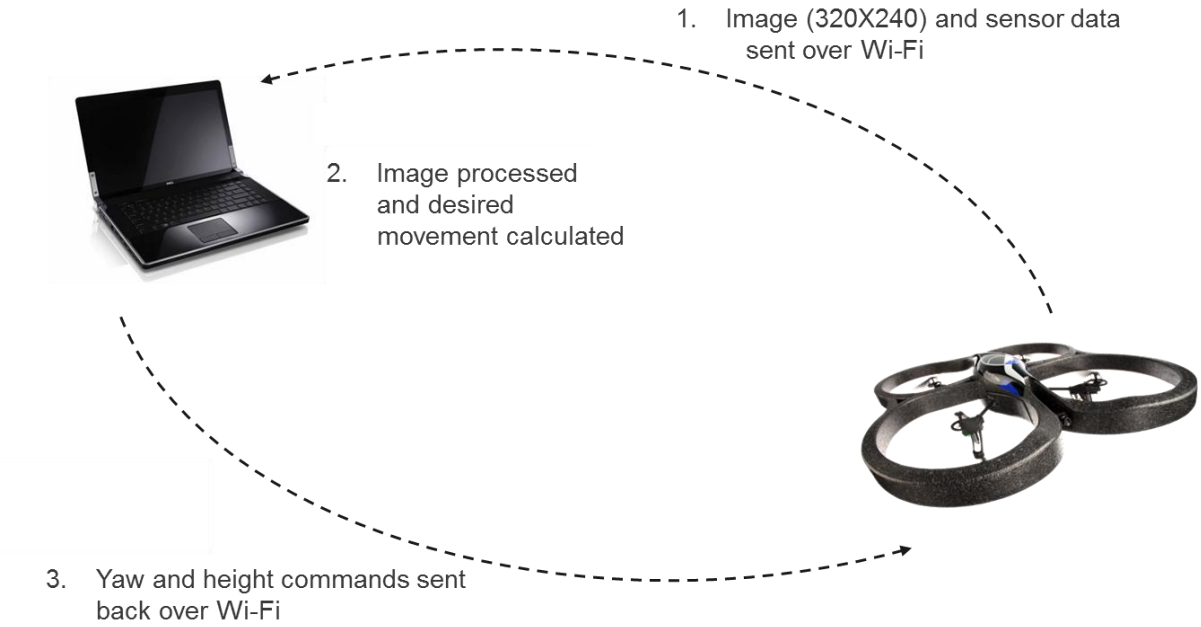


Figure 4.6. Data flow for the object following (remote computer) experiment.

The flow of processing undertaken in the OpenCV program is:

1. The received image is converted to the YUV colour space.
2. Image is then thresholded to a binary image based on pre-determined thresholds for the red object.
3. Erosion and dilation operations are applied to reduce image noise.
4. The centre of area of the remaining object is calculated.
5. The offset from the image centre is found and a command is sent to the drone.

Various stages of the program flow are illustrated in figures 4.7.1-4.7.2.

Figure 4.7.1. Raw input image (left), after conversion to YUV colourspace (centre) and the V channel upper and lower threshold values (right).

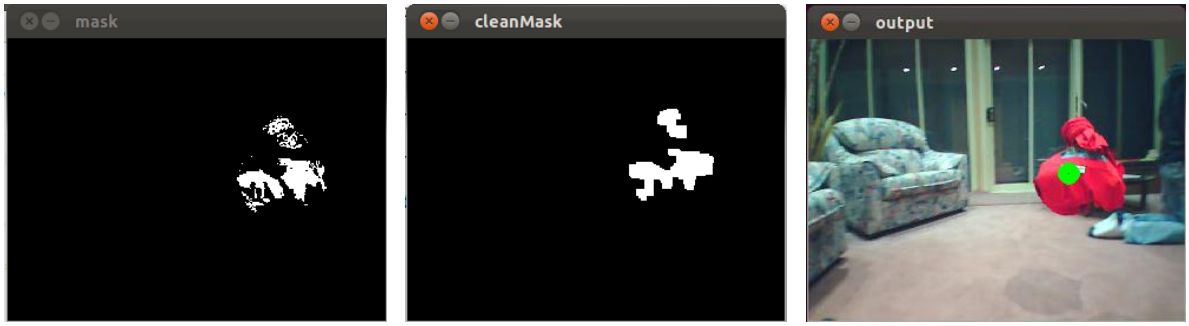


Figure 4.7.2. The binary thresholded image (left), the binary image after dilation and erosion operations (centre) and the location of the centre of the blob (right).

The outcome of this approach was recorded by comparing the desired position with the actual drone position as determined from the drone sensor data. Figure 4.8 depicts the yaw movement of the drone during a test where only yaw angle was adjusted (height remained constant). It was found that the actual yaw lagged behind the desired yaw by 710ms on average. This is a considerable delay, which results in the pendulum leaving the view of the drone camera for larger amplitude swings. Video of the experiments can be found on the project YouTube channel [19].

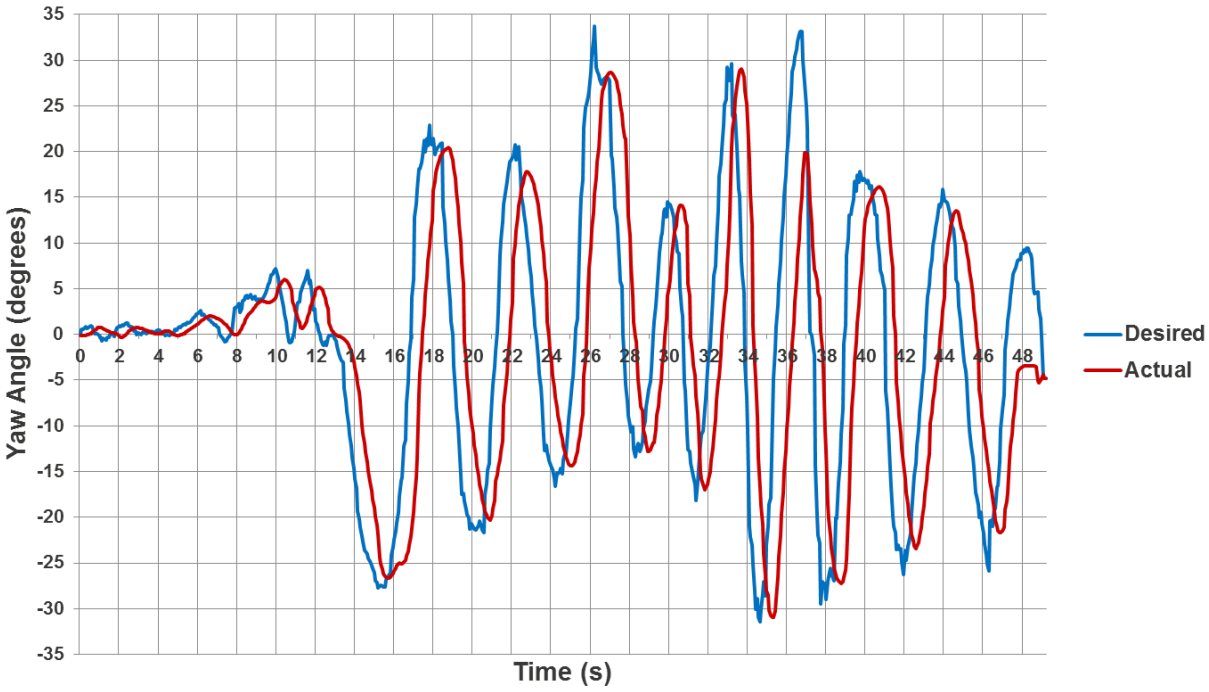


Figure 4.8. Drone yaw during the off-board processing object following experiment.

An experiment was also conducted to measure response in the height of the drone. Figure 4.9 compares the desired and actual height of the drone. The average delay between the desired and actual heights was found to be 625ms, which also generated problems when the blob was moved faster (but not unreasonably fast).

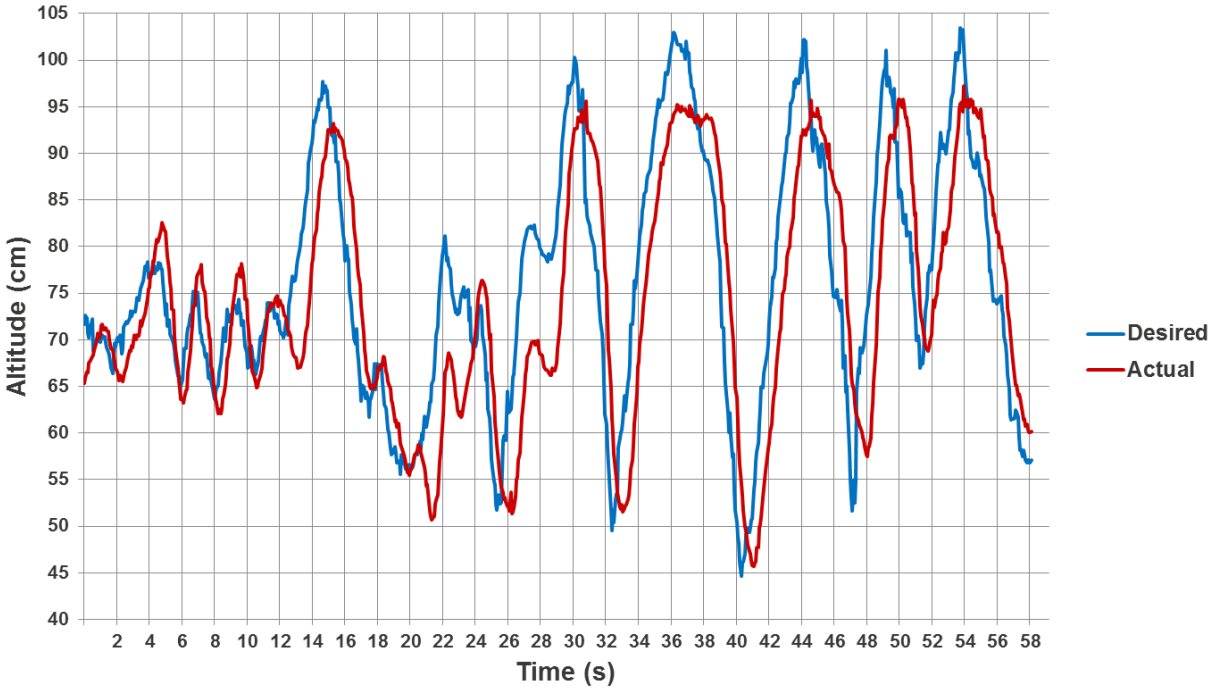


Figure 4.9. Drone height during the off-board processing object following experiment.

4.4.2 Onboard Processing

This experiment involved replacing the default drone program with a custom program. The program was based on the firmware created by Hugo Perquin [16]. Three threads are run on the drone; one to process the front camera video, another to update the PID controllers and a third to send commands to the motors. Figure 4.10 outlines the program flow.

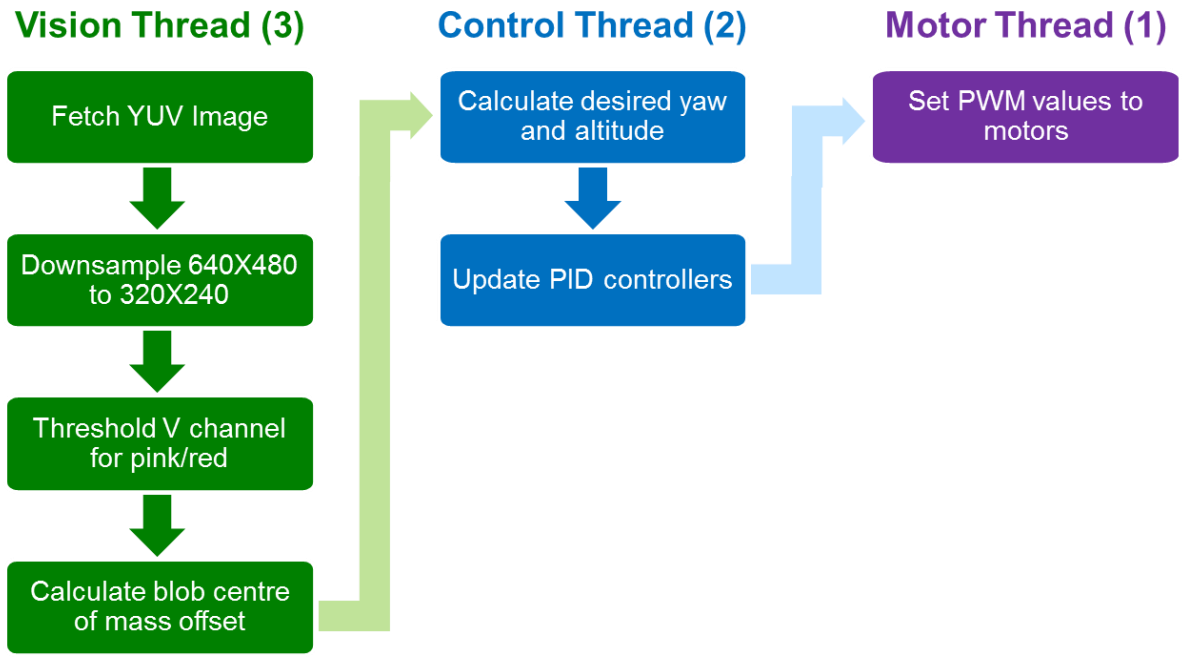


Figure 4.10. Program flow for the onboard implementation of the object following experiment. Thread priorities are in parentheses.

The vision thread runs at 15Hz (the frame rate of the camera) and takes 8.1ms on average whilst the control and motor threads run at 200Hz and take 0.05ms and 0.10ms respectively per cycle. This results in an overall CPU usage of 15.15%.

The PID controllers used were tuned manually using the Ziegler-Nichols method [20]. The system has a separate PID controller for yaw and height and these were tuned independently. Figure 4.11 shows the PID gains used.

Controller	Proportional Gain (P)	Integral Gain (I)	Derivative Gain (D)
Yaw	0.8	0.004	-0.065
Height	0.004	0.01	-0.002

Figure 4.11. PID contorller gains used for the onboard firmware replacement program.

The threshold values used to determine the presence of a blob were tested by saving images from the drone camera in which matching pixels were shaded green (refer to figure 4.12). The centre of area for the blob is indicated also to ensure the pixel coordinates values were being calculated correctly.

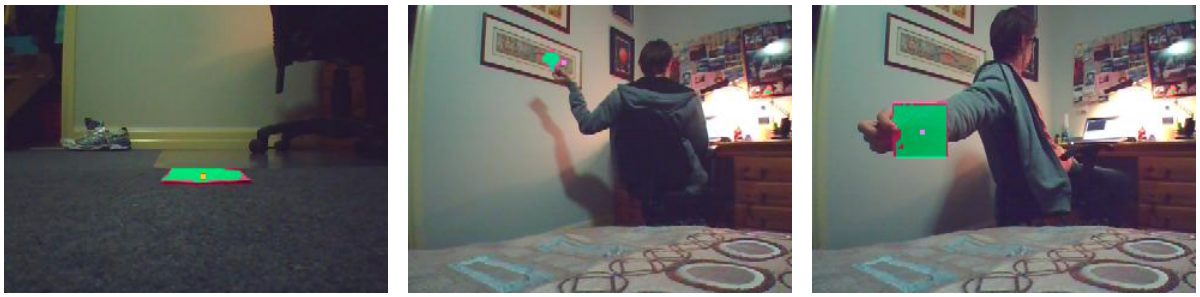


Figure 4.12. Sample frames used to establish upper and lower thresholds of the V channel in the image for detection of the desired object.

The result of the yaw-only experiment can be seen in figure 4.13. It shows that the drone follows much more closely to the desired yaw angle than for the experiment conducted in section 4.4.1. The average lag of the actual yaw angle as compared to the desired yaw angle was found to be 210ms. This resulted in the drone being able to follow the pendulum for swings of larger amplitude than was possible with the previous experiment. Video of the experiments can be found on the project YouTube channel.

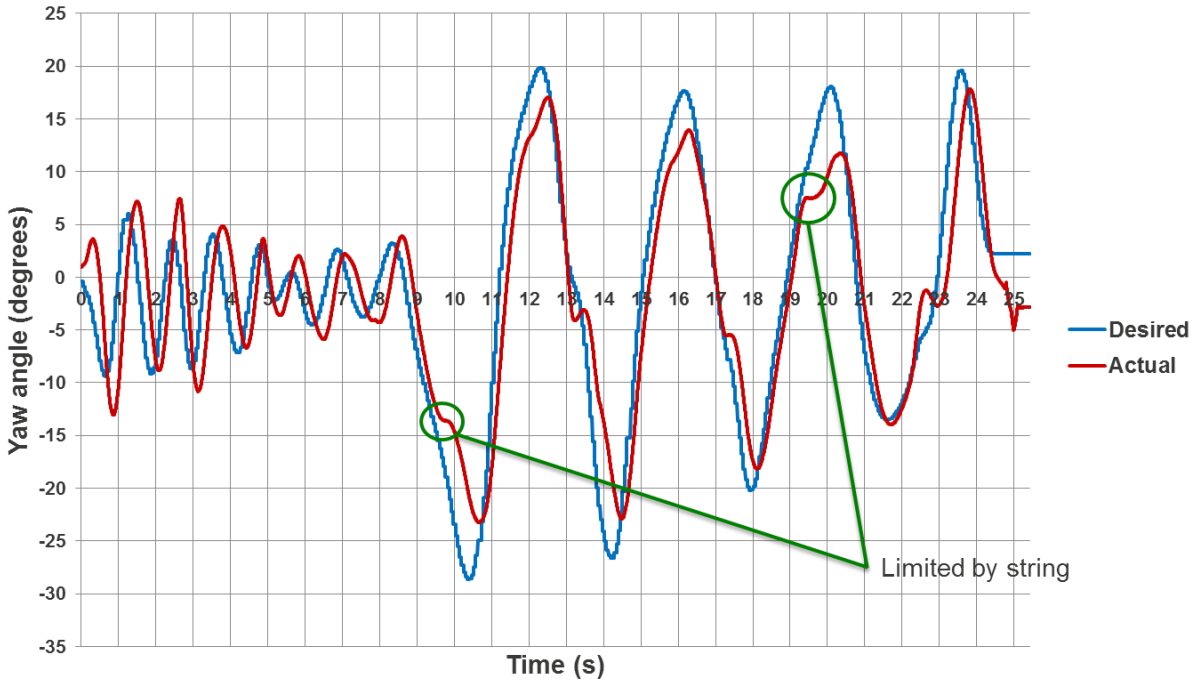


Figure 4.13. Drone yaw during the onboard processing object following experiment.

Figure 4.14 compares the desired and actual height of the drone. The average delay between the desired and actual heights was found to be 550ms, which is significantly greater than for the yaw experiment, possibly due to the motor power increase needed for the drone to gain height.

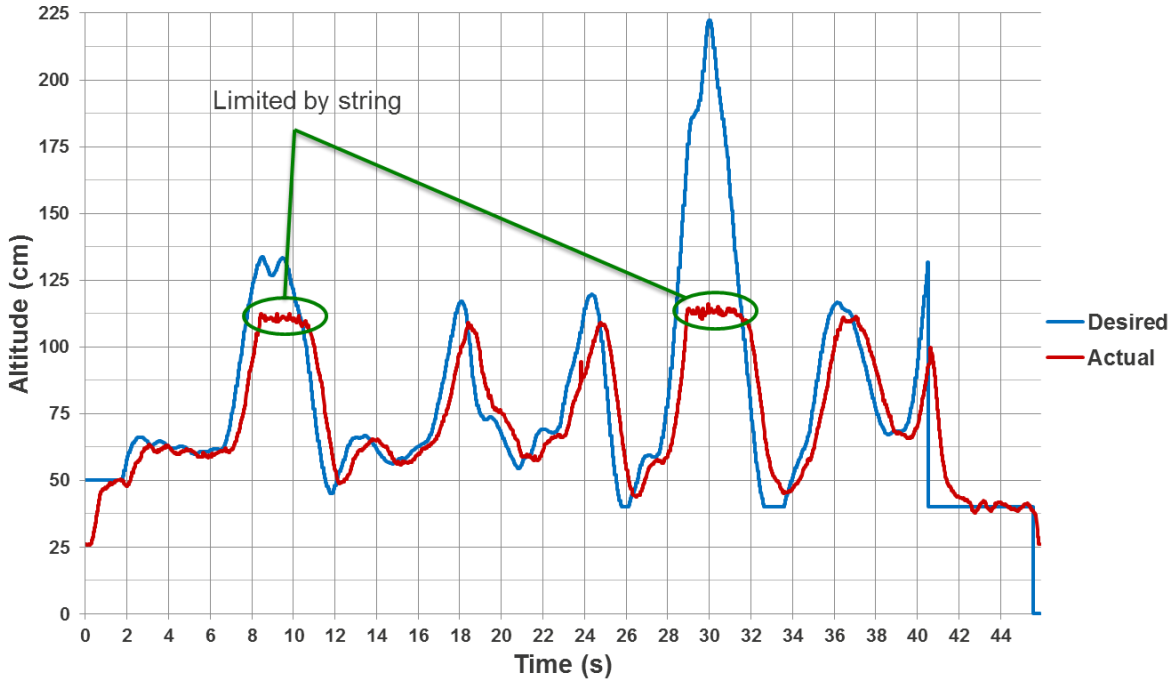


Figure 4.14. Drone height during the onboard processing object following experiment.

4.5 Results

It was found that the response of the system when sending camera frames to a remote computer via WIFI for processing was quite poor. The latency to send the images varied significantly and was as much as 385ms. The frame loss rate was also high, with more than one fifth of frames not being received by the laptop computer.

The simple object following tests showed that this latency introduces issues with system controllability and response. When the experiment was implemented onboard the drone however, performance increased. The main problem with this approach is that stabilisation provided by the downward camera is no longer available. This is due to the need for program.elf to be killed to gain access to the sensors and motors. Hence a custom stabilisation algorithm must be implemented to achieve a stable position in the horizontal plane.

5.0 AR.Drone 2.0

5.1 Platform Overview

The AR.Drone 2.0 represents a significant upgrade compared with the original version of the device. The changes are discussed below.

5.1.1 Sensors

The drone now contains more sensors with greater accuracy than the previous version of the drone. The sensors included are outlined in figure 5.1.

Sensor	Parrot Specification	Notes
3 Axis Accelerometer	+/- 50 mg precision	
3 Axis Gyroscope	2000 degree/second precision	
Pressure Sensor	+/- 10 Pa precision (80cm at sea level)	
3 Axis Magnetometer	6 degree precision	Needs to spin 360° to calibrate
Vertical Downward Camera	QVGA at 60 FPS	`/dev/video2`. V4L2 pixel format is UYVY
Horizontal Forward Camera	92° FOV HD720 at 30 FPS	`/dev/video1`. V4L2 pixel format is UYVY
Ultrasound Sensor		Minimum height 30cm

Figure 5.1. AR.Drone 2.0 sensor specifications [15].

Through experimentation it was found that it is possible to access the front camera and also run program.elf to make use of the in-built hovering stabilisation. The process is:

1. Modify the start-up script `bin/check_update.sh` at line 57 to remove the automatic launch of program.elf. The line should read:
   `#(/bin/program.elf ${PELF_ARGS}; gpio 181 -d ho 1) &`
2. Copy the modified script to the drone, overwriting the original and modify the file permissions using:
   `chmod 777 bin/check_update.sh`
3. Reboot the drone. Now program.elf will no longer run.
4. From the parasite program open the front camera using:
   `video_init("/dev/video1", 1280, 720, 30);`
5. Wait for 2 seconds.
6. Run program.elf from the parasite program using:
   `system("(../../bin/program.elf ${PELF_ARGS}; gpio 181 -d ho 1) &");`
7. Wait 7 seconds while program.elf starts. The drone can now be flown as per usual from the FreeFlight smartphone application but the app will report that the video has failed.

5.1.1.1 Bearing Calibration

The addition of a 3 axis magnetometer to the AR.Drone 2.0 was explored to determine absolute yaw angle. The navboard provides three values relating to the magnetometer; mag_x, mag_y and mag_z from which the drone's absolute bearing can be computed. To ascertain the relationship between these variables and the bearing, the drone was slowly rotated whilst on a level surface and the sensor readings were recorded. The relationship between mag_x, mag_y and the bearing is shown in figure 5.2.

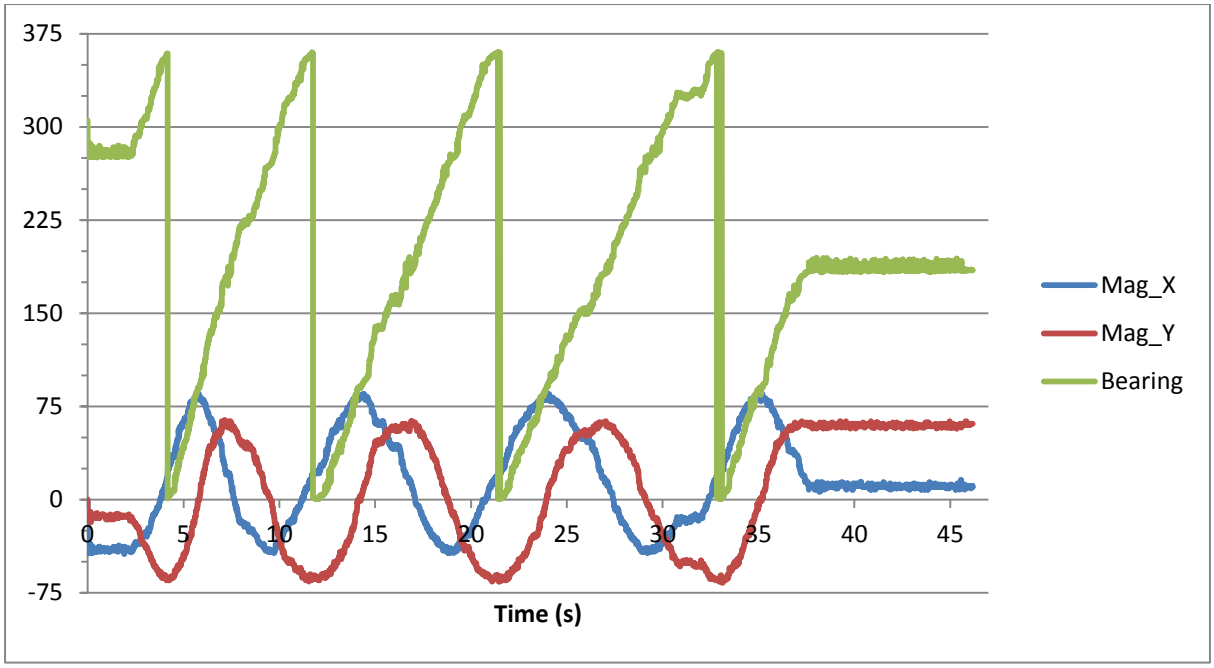


Figure 5.2. Relationship between magnetometer readings and bearing as the drone rotates continuously.

It can be seen that the two magnetometer readings generate sinusoidal waves separated by a 90° phase difference. This relationship was used to calculate bearing angle using the function listed in section 10.3 and shown as the green line in figure 5.2. It must be noted that the offset and scale factors need to be determined before this is possible. These values are found by commanding the drone to spin in a full circle after it takes off, whilst recording the maximum, minimum and mean values of the sensor output.

5.1.2 Computing Hardware

The processor of the drone has been upgraded to an ARM Cortex A8 32bit 1GHz model (refer to figure 10.1). This processor contains NEON capability for high speed optimisations. The drone also has an 800MHz video DSP – model number TMS320DMC64x. Use of the DSP was not explored in this project, but it would allow for more intensive computer vision tasks to be completed in real time. The drone also contains 1 GB of DDR2 RAM running at 200MHz.

5.1.3 Other Hardware

The AR.Drone 2.0 is equipped with a USB 2.0 port on board which can be used for data storage. The USB stick used must be FAT32 format. When a USB stick is inserted the drone lights should blink rapidly for a second to indicate the USB has been mounted. The USB directory is mounted in `data/video/usb/`. The WIFI chip has been upgraded to a b/g/n variant.

The drone has one red and one green LED below each rotor that can be activated separately or together. It was found that the LEDs must first be initialised by program.elf before they can be controlled. The LEDs were activated directly using the following commands from the TELNET client either whilst program.elf was running or after it had been killed:

All LEDs off: `echo -en \"\\x60\\x00\" >/dev/ttyO0`

All LEDs red: `echo -en \"\\x7f\\x00\" >/dev/ttyO0`

All LEDs green: `echo -en \"\\x60\\x1f\" >/dev/ttyO0`

All LEDs orange: `echo -en \"\\x7f\\x36\" >/dev/ttyO0`

It should be noted that if the LEDs are set whilst program.elf is running, then they will be overwritten almost instantly. To avoid this, the LEDs were set at a much higher frequency that that at which program.elf controls them (greater than 20Hz). Initially it was tried to set the LEDs using the `gpio` command but this proved ineffective.

5.1.4 Software

The drone now runs Linux version 2.6.32.9-g0d605ac with the same version of BusyBox as the original device. Through testing it was found that the drone has STL support included, so STL-port was not required. Code by Parrot that runs onboard the drone was cross-compiled using the Sourcery G++ Lite for ARM GNU/Linux version 2010.09-50 compiler. The system version information was found by entering the following:

```
# cat proc/version
Linux version 2.6.32.9-g0d605ac (aferran@FR-B-800-0053)
(gcc version 4.5.1 (Sourcery G++ Lite 2010.09-50))
```

5.2 Control Approach

It was shown in section 4.3.1 that due to the large latency associated with sending the drone forward camera frames over WIFI, this approach is not desirable. Therefore it is necessary to perform computer vision tasks onboard the drone. It was shown that access to the front camera can be achieved both while program.elf is running and whilst it is not (refer to section 5.1.1), so the two control possibilities are:

A. Control the drone directly as was done for the blob following in section 4.4.2. This requires implementation of an optical flow algorithm using the downward facing camera for stabilisation.
B. Allow program.elf to perform stabilisation tasks and control the drone by sending UDP commands to program.elf. This requires the parasite program to simulate a WIFI connection to the drone from onboard the drone.

The tests that were performed to evaluate the suitability of each approach are described in section 5.3.

5.3 Platform Capability Tests

For control approach A, a simple optical flow algorithm was implemented to determine if this could be used to stabilise the drone in-flight. As the downward camera runs at 60FPS it would be necessary for the algorithm to run at well above this rate, to allow for other tasks to be completed in parallel.

For control approach B, several tests were performed. Firstly the CPU usage of program.elf was measured to determine how much CPU power remained for other tasks whilst the default program was running. Next, multiple tests were conducted to determine the latency associated with sending control commands to program.elf via the UDP method. This control method was discovered by the creator of the ardudrone modification whereby the drone is controlled via an RC controller [21]. This is done by sending UPD commands to the UDP port 5556 from a program onboard the drone, just in the same way as would be done from a remote computer connected via WIFI.

Tests were also performed to determine the intrinsic parameters of the front camera. This allows for the lens distortion to be modelled in the software, hence generating more accurate computer vision outcomes.

5.3.1 Optical Flow Stabilisation

A basic algorithm was implemented using the vertical camera to perform optical flow. If the movement of the drone in the horizontal plane can be determined from adjacent camera frames, then the drone can be controlled to counter this movement, hence hovering over a single point. This algorithm was adapted from the "video_blocksum" function by Hugo Perquin [16]. The results of the processing time per frame are displayed in figure 5.3. It can be seen that the optical flow runs at a median frequency of 12.2Hz and hence cannot run at the frame rate of the downward camera (60 FPS). As such control method A was determined to be unsuitable.

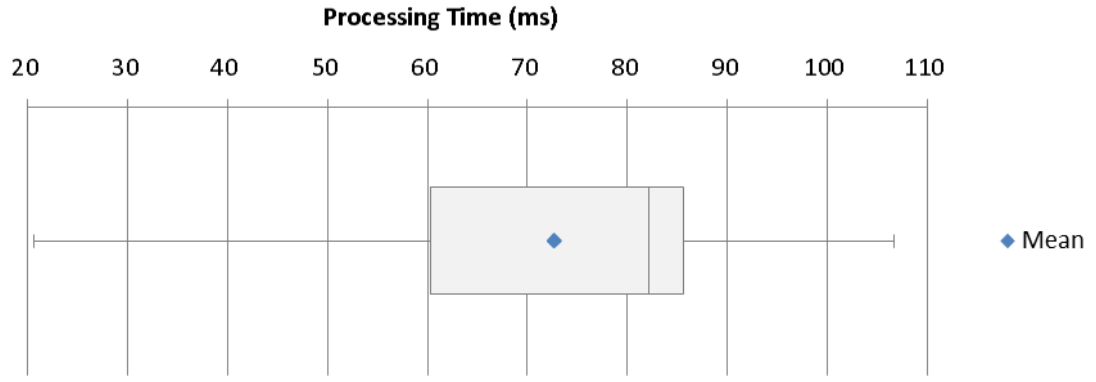


Figure 5.3. Distribution of processing time measurements for the optical flow stabilisation algorithm when implemented manually.

5.3.2 CPU Benchmarks

A variety of CPU benchmark tests were performed to determine how much of the CPU program.elf requires to run. The tests performed were from the "Classic Numeric PC Benchmarks" collection by Roy Longbottom [22]. The tests performed were a floating point benchmark (Whetstone); two different integer benchmarks (Dhrystone 1 & 2), a workstation floating point benchmark (LINPACK) and a supercomputer floating point benchmark (Livermore Loops).

For each test, two iterations were performed whilst program.elf was not running (control tests) and two iterations were performed whilst program.elf was running and stabilising the drone in flight. The results for each test are summarised in figure 5.4 and detailed in section 10.4.

Test	Program.elf CPU utilisation (%)
Whetstone MIPS	23.249
Dhrystone MIPS 1	25.237
Dhrystone MIPS 2	24.497
LINPACK	28.446
Livermore Loops	18.529
Average	**24.0**

Figure 5.4. Summary of the AR.Drone 2.0 CPU benchmarking results.

The results show that program.elf uses approximately 24% of the 1GHz CPU, thus leaving approximately 760MHz for the execution of custom code. This result shows that it is possible to run computer vision tasks onboard the drone in parallel to program.elf.

5.3.3 Program.elf Control Latency

Since it has been determined that program.elf can be used to control the drone movements, it is necessary to determine the latency associated with sending control commands to the drone. The first approach used to test this was as follows:

1. Set the drone in front of a mirror and use a lens to focus the light from one of the LEDs into the front camera (refer to figure 5.5).
2. Start a parasite program on the drone that has access to the front camera. Send a command to program.elf to turn the LEDs off.
3. Start a timer and send a command to program.elf from the parasite program to turn on the drone LEDs.
4. Check the video feed in the parasite program for a sudden change in intensity at a certain pixel location (constant time operation).
5. Stop the timer when the LED change is detected.

Figure 5.5. Drone front camera view when LED is focussed onto camera lens.

The results of this approach are summarised in figure 5.6 below. The median latency of 202ms is much larger than expected. It is known from the boot log however that program.elf operates a separate thread to update the LEDs so it is possible that the drone runs this thread at a lower priority or lower frequency than the thread to control the motors. As such the response of program.elf itself was next measured.

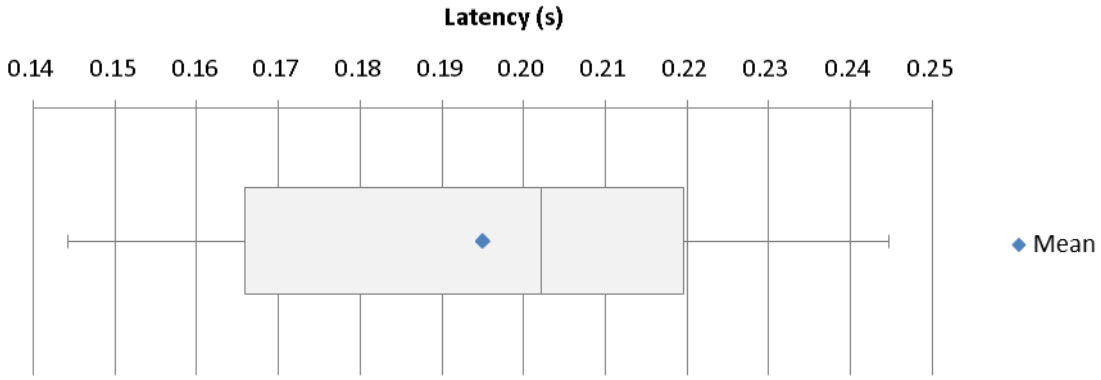


Figure 5.6. Distribution of latency measurements for program.elf to receive a command over the UDP port and activate an LED.

5.3.3.1 Program.elf Response Latency

If the user has access to the drone via TELNET when program.elf is launched then the program will print various messages to the user screen as it runs. Fortunately, the drone prints a message when an incoming connection is detected on the control port 5556. This message includes the drone system time, so it was possible to read the system clock and send a command to the drone from an external program running on the drone, then record the system time as printed by program.elf. The difference between these times represents the latency of program.elf to receive the command over the UDP port. The results of this latency test are summarised in figure 5.7.

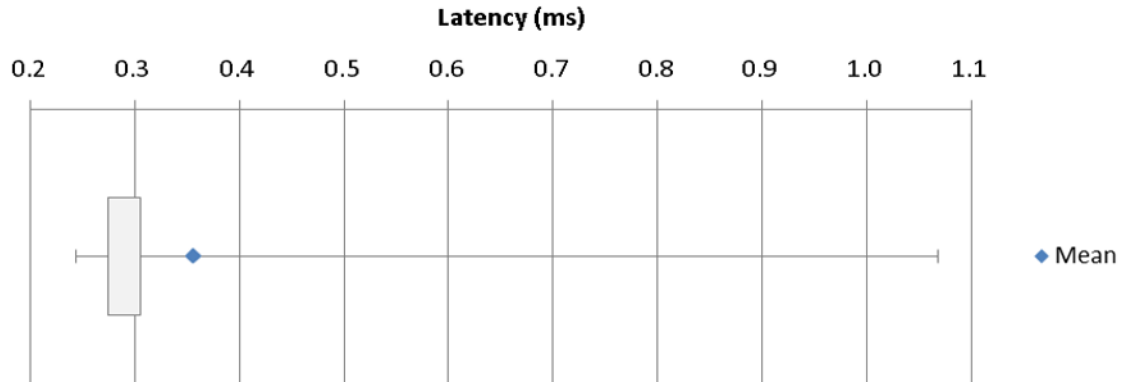


Figure 5.7. Distribution of latency measurements for program.elf to receive a command over the UDP port.

5.3.3.2 Motor Response Latency

Since it has been established that program.elf receives commands from the parasite program with a median latency of 0.275ms, it is now necessary to establish the latency of the motors to respond to a command. This experiment was conducted as follows:

1. Run program.elf
2. Run parasite program that sends “takeoff” command to program.elf.
3. Wait 1 second.
4. Directly set LEDs to orange from parasite program and send “emergency stop” command to program.elf (see section 5.1.3).
5. Use 1000FPS video camera to film drone blades and LEDs to determine time difference between LED turning orange and blades slowing (see figure 5.8).

Figure 5.8. Sequential video frames before (left) and after (right) the LEDs are activated orange.

The experiment was conducted 6 times and the video was analysed frame-by-frame. Due to the resolution of the latency measurement being limited by a half revolution of the drone blade, the latency can only be guaranteed to be inside a certain range of values. The results are summarised in figure 5.9.

Experiment Number	Minimum Latency (ms)	Maximum Latency (ms)
1	14	28
2	5	17
3	6	17
4	11.5	24
5	3	17
6	10	21.5
Combined Result	14	17

Figure 5.9. Summarised results of the motor response latency for program.elf on AR.Drone 2.0.

It can be seen that the latency of the motors responding to a command sent to program.elf via a second program running on the drone is between 14 and 17ms which is deemed to be low enough for the purposes required. As such it was determined that running computer vision tasks on the drone in parallel to program.elf would provide the best system response and flight stability.

5.3.4 Camera Calibration (MATLAB)

As the front camera is to be used for computer vision purposes, it cannot be assumed to be a pinhole camera. The camera was calibrated using the "Camera Calibration Toolbox for MATLAB" [23, 24]. This approach requires a set of images to be captured of a checkerboard pattern in different orientations using the drone camera (refer to section 10.5 for dataset used). The checkerboard corners are then selected manually and the program uses these to calculate the intrinsic parameters of the camera. The corners are then re-estimated automatically to increase the accuracy of the method. The full list of parameters obtained is included in figure 5.10.

Camera Parameter	Value	Error
Focal Length (f)	1127.03, 1121.63	1.92, 1.93
Principal Point (c)	638.74, 381.29	3.42, 2.74
Skew (alpha)	-0.000 499	0.000 466
Cubic Radial Distortion Coefficient (k1)	-0.554 805	0.008 645
Quintic Radial Distortion Coefficient (k2)	0.473 795	0.038 608
Septic Radial Distortion Coefficient (k5)	-0.262 756	0.048 599
Linear Tangential Distortion Coefficient (k3)	-0.000 365	0.000 593
Quadratic Tangential Distortion Coefficient (k4)	-0.000 570	0.000 527

Figure 5.10. Complete intrinsic parameters of the AR.Drone 2.0 front camera.

To reduce the complexity of the camera model used, the skew was assumed to be zero, as were the tangential distortion coefficients and radial distortion coefficients greater than 5rd order (k3-k5). When these parameters were forced to zero, the intrinsic camera parameters were as shown in figure 5.11. This results in a quintic model for the camera.

Camera Parameter	Value	Error
Focal Length (f)	1124.67, 1119.24	1.85, 1.87
Principal Point (c)	635.44, 380.02	1.54, 1.44
Distortion Coefficient (k1)	-0.512 645	0.003 882
Distortion Coefficient (k2)	0.267 903	0.007 229

Figure 5.11. Approximate intrinsic parameters of the AR.Drone 2.0 front camera, as used by the computer vision application.

The focal length was taken as the average of the two values in figure 5.11 (1122) as they are close enough for this to be a reasonable assumption and it reduces complexity of the camera model. Tests were conducted to evaluate the appropriateness of the model used. This was done by undistorting images from the front camera and visually assessing the straightness of lines in the image that should appear straight. A sample image pair is shown in figure 5.12.

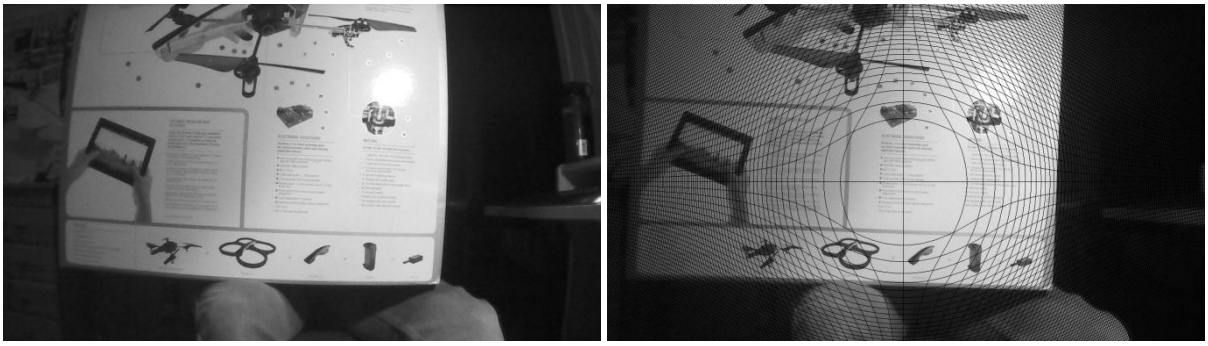


Figure 5.12. Raw camera frame (left) and after undistortion applied (right).

Note that the black lines in the undistorted image represent locations where no pixel in the distorted image maps to. This is due to the libCVD implementation of image undistortion not using interpolation to fill these gaps. As feature extraction will be computed from the original image, with only the location of those features being mapped to undistorted locations, these lines will not generate false keypoints (refer to section 5.4).

This design decision was made based on the limited processing power of the drone. If each camera frame were undistorted (with interpolation to fill the gaps) before keypoints were computed then the processing time per frame would be much larger. It was found through testing that matching performance was not significantly affected by attempting to match distorted keypoint descriptors.

The quintic camera model is depicted as a vector field in figure 10.14. To increase runtime efficiency of the undistortion process, a 1280X720 image containing image references is created before the main processing loop as a look-up table for the undistortion process. Once this table is created, a distorted pixel location (x_d, y_d) is mapped to an undistorted location (x, y) by accessing the look-up image at location (x_d, y_d):

```
(x, y) = lookupTable[x_d][y_d]
```

This is performed for each keypoint location before they are sent to the server, so has a bounded time cost of k*MAX_CORNERS (refer to section 5.4.2).

5.4 Computer Vision Application

The computer vision application implemented on the drone was adapted from work done by Winston Yii [25]. The goal is to use computer vision to navigate to set locations in the environment.

5.4.1 System Overview

The system requires an RGB-D camera (such as a Microsoft Kinect or ASUS Xtion Pro Live) to be set up in front of a scene. This camera is connected to a server computer which then communicates with the AR.Drone 2.0 via WIFI. The drone is set up so that the front camera views the same scene as the RGB-D camera (refer to figure 5.13). When the drone is flying it computes keypoints and

keypoint descriptors from the front camera image. The server computer also computes keypoints and keypoint descriptors from the RGB image.

The drone sends the keypoint information to the server for each camera frame and the server calculates matches between both keypoint sets. These matches are then refined using RANSAC and the drone pose relative to the RGB-D camera is computed. This pose is then sent to the drone via WIFI where the pose is filtered using a median filter and checking bounds. The desired motion of the drone is then calculated to enable it to fly to a goal location. Commands are then send to program.elf via the UDP connection to move the UAV. The flow of processing is depicted in figure 5.14.

Figure 5.13. Experimental setup for the localisation tests.

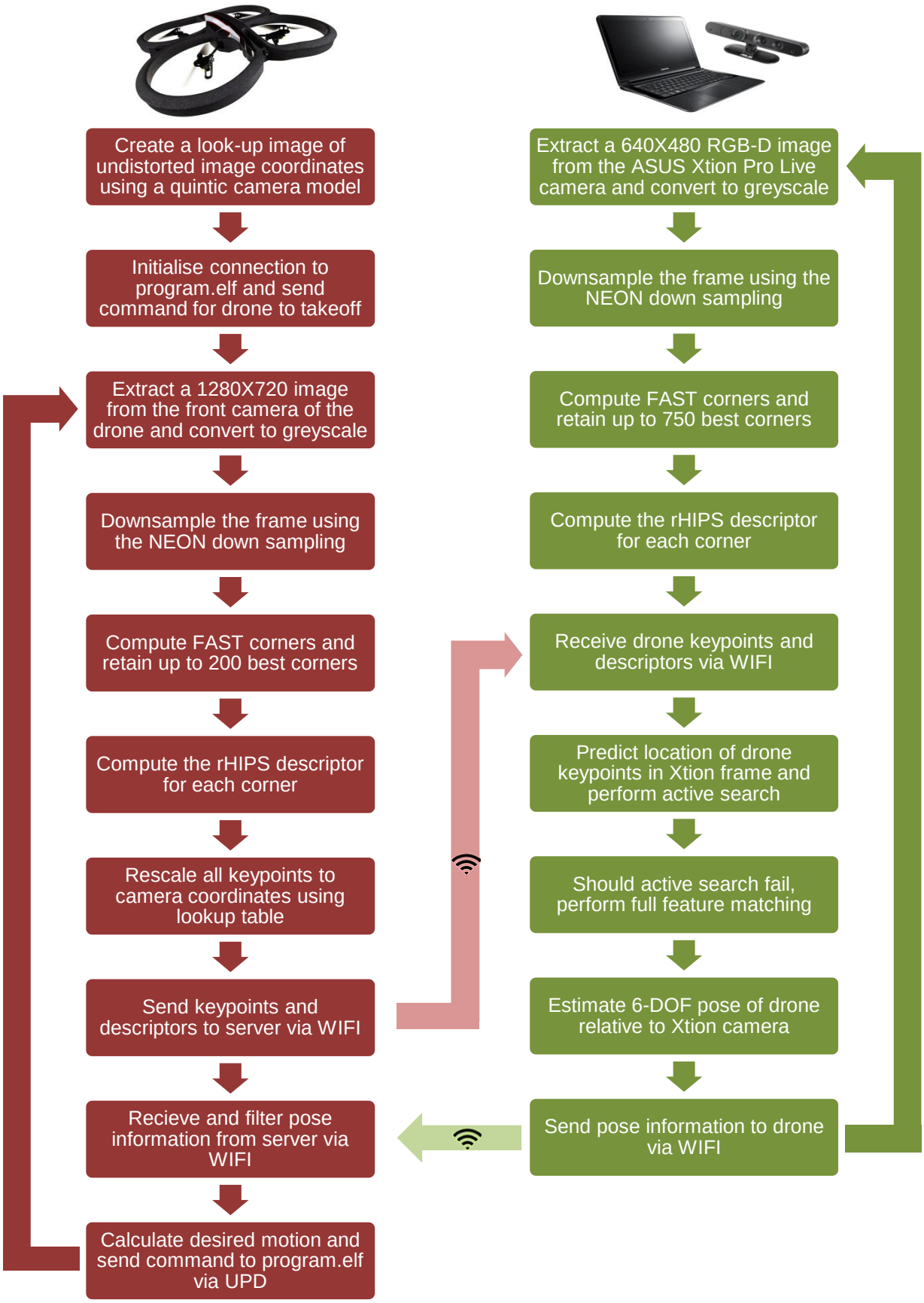


Figure 5.14. Flow of processing for the localisation application implemented on the AR.Drone 2.0.Multiple threads are used to ensure efficient CPU usage.

5.4.2 Drone-based Processing

The tasks completed onboard the drone will now be described in detail. When the program is launched some initialisation tasks are first performed. Firstly the front camera is opened as described in section 5.1.1 and program.elf is launched. A connection to program.elf is then established via UDP and an image is set up to undistort image pixel coordinates using the quintic camera model described in section 5.3.4. This image is essentially a look-up table that maps distorted pixel locations to their undistorted camera coordinates. Using a "look-up image" in this manner allows for constant time cost when keypoint coordinates are mapped to camera coordinates in the main processing loop. Next the drone is commanded to takeoff through a command to program.elf and the two main processing threads begin.

5.4.2.1 Computer Vision Thread

This involves obtaining a greyscale frame from the front camera (1280X720) and down sampling it by a factor of 2, then a factor of 4 to create some scale invariance. This down sampling is done using NEON optimisations provided by the libCVD library, so it takes very little time. Both down-scaled images are then processed using the FAST-9 algorithm to find FAST keypoints [26]. The 100 best keypoints are selected per image so that a set of up to 200 keypoints is retained (see figure 5.15).

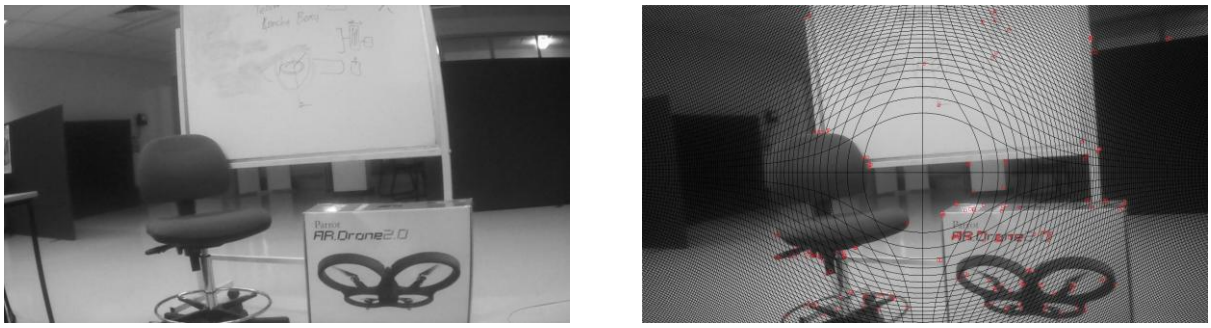


Figure 5.15. Raw drone image (left) and undistorted view with keypoints (right).

rHIPS feature descriptors are then calculated for each keypoint [27]. The set of keypoints and descriptors is sent to the server computer via WIFI. In turn the drone receives the estimated relative 6 degree of freedom (DOF) pose from the server via WIFI. Only the (x, y, z, yaw) information is considered, as the pitch and roll are assumed to be zero. This assumption is based on the consideration that the drone will remain roughly level in the air when flying, as the commands sent to move the drone will make only slight adjustments to the drone pitch and roll.

The (x, y, z, yaw) result is rejected if it contains all zeros (no match found) or if any of the values lie outside reasonable bounds. The bounds used are indicated in figure 5.16 whilst figure 5.17 shows the coordinate system used. If the pose is valid then it is added to a buffer of size 3 from which the median pose is selected as the actual pose. This pose is maintained as a global variable (protected by semaphores) to enable the control thread to access it at any time.

Relative Position	Minimum	Maximum
X direction	-4.0m	4.0m
Y direction	-1.5m	1.5m
Z direction	-4.0m	4.0m
Yaw angle	-60°	60°

Figure 5.16. Bounds used to reject pose estimates.

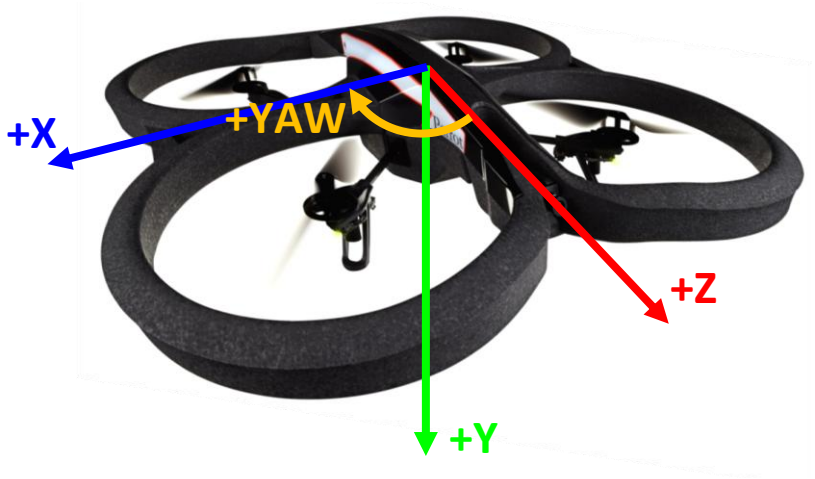


Figure 5.17. Drone coordinate system used for the localisation application.

5.4.2.2 Control Thread

The control thread queries the pose at or above 20Hz and calculates a command to send to program.elf. If the drone is within a small range of the goal then it enters hover mode whereas if it is away from the goal then a command is sent for 50ms. Yaw is adjusted first to ensure pitch and roll move the drone in the desired direction. If no valid pose estimate is available or it is stale (older than 0.3 seconds) then the drone enters hover mode for up to one second. If there is still no valid pose after one second then the drone yaws left and right by 45° until a valid pose is found.

Due to time constraints, a simple bang-bang controller was used to calculate commands to be sent to the drone when moving to the goal. The smallest values of pitch angle, roll angle, yawing rate and altitude rate that caused the drone to move were measured and these were used as the values sent to the drone (refer to figure 5.18). A proportional controller was not used because sending larger values of movement rate to the drone caused it to move too rapidly, resulting in camera blur that led to no matches and hence loss of pose information.

Drone Command	Absolute Minimum	Absolute Maximum
Pitch	0.05	0.25
Roll	0.04	0.35
Yaw Rate	0.15	0.6
Altitude Rate	0.25	0.6

Figure 5.18. Maximum and minimum values measured to cause the drone to move. Note that only the minimum values were used.

In future work a more complex PID controller can be developed that keeps an estimate of drone location based on previous reading of pose and velocity so that if the localisation fails, the drone still has an estimate of its state. This could be coupled with information from the drone sensors for greater accuracy, an area not explored in this project.

The goal location can be a set position relative to the Xtion camera so that as the user moves the camera, the drone flies to adjust its position. This technique is called Visual Servoing. An alternate approach is to keep the Xtion camera stationary and the goal location is set either as a hard-coded sequence of waypoints or by user-entered data.

Note that at this stage obstacle avoidance is not built into the system as the drone receives no information from the server about the environment. This could be added simply as an additional piece of information sent to the drone via WIFI. A basic way to implement this is to send a point coordinate to the drone that represents the location of the nearest obstacle relative to the drone (sensed from the Xtion depth information). Then the drone could perform a simple check to ensure it does not fly towards this point if it is closer than some threshold value.

5.4.3 Server-based Processing

The server receives a 640X480 RGB-D image from the Xtion camera which is converted to greyscale (refer to figure 5.19 - 5.20). This is down sampled in the same manner as for the drone, except that 5 levels of resolution are used; factors of 1, 1.5, 2, 3 and 4. FAST-9 corners and rHIPS feature descriptors are then computed to create a set of up to 750 keypoints.

When the server receives a set of keypoints from the drone it searches for matches between the sets of points. If no previous pose was found then this is done using a full search of every drone keypoint to every Xtion keypoint. If a previous pose exists however, then an active search is used where keypoints are compared with those inside a certain radius of their expected location. If the active search fails then a full search is run.

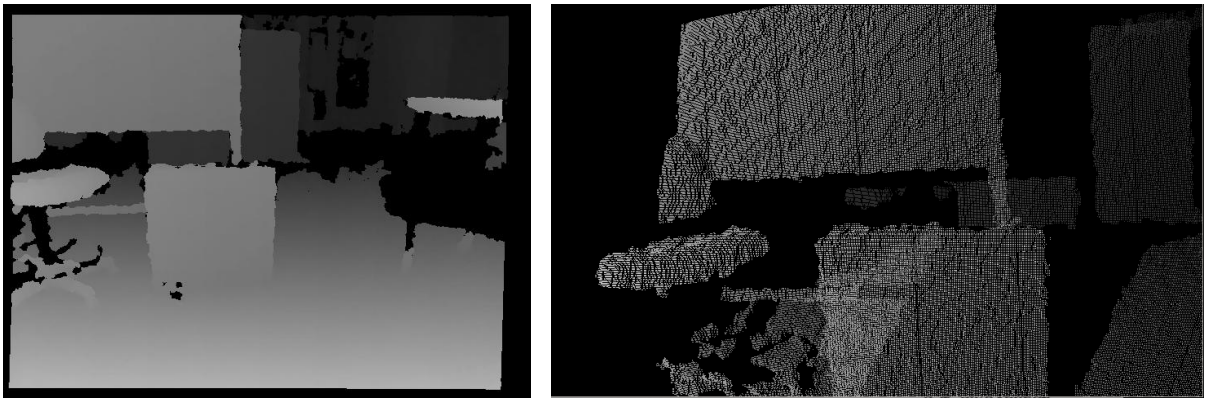


Figure 5.19. Xtion depth image (left) and drone depth image (right) as calculated by reprojection of depth information in drone viewpoint.

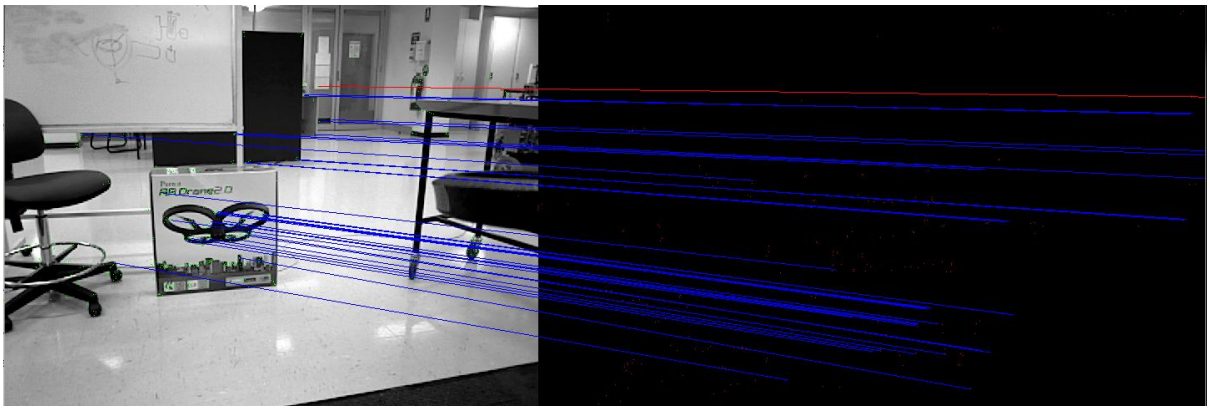


Figure 5.20. Keypoint matches between the Xtion image (left) and drone keypoints (right – location only). Blue lines indicate inlier matches whilst red lines indicate outlier matches after RANSAC.

The set of matches is then refined using the RANSAC algorithm (refer to figure 5.20) [28]. This selects a random set of matches from which to create the transformation between point sets. The remaining matches are tested against the transformation and if a large percentage of matches agree then this is taken to be the correct transformation. If few other matches agree then the process is repeated with another random set of matches.

Once the transformation between views has been found the drone position and orientation in space relative to the Xtion camera is sent to the drone via WIFI. If no transform has been found then a set of zeros is sent which the drone recognises as a failed result. The server also calculates the drone's view of the depth information as a means for the user to visually check the results of matching (refer to figure 5.19).

5.5 Results

5.5.1 Frame Rate Performance

The system runs at the frame rate of the AR.Drone front camera (30Hz) when the server processing is done on either a desktop or powerful laptop computer. This indicates that it in fact runs faster but is limited by the camera capabilities. The drone based processing time per video frame was measured and found to vary with the number of keypoints found (refer to figure 5.21). The median processing time of 25.57ms would give rise to a frame rate of 39.1FPS if the camera were capable of this.

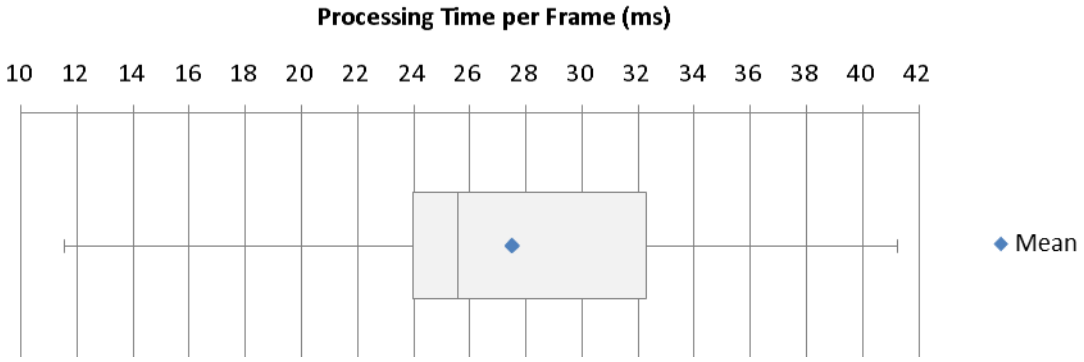


Figure 5.21. Distribution of proccessing times per camera frame onboard the drone.

5.5.2 Localisation Accuracy

Detailed results of the accuracy of localisation with respect to the Xtion camera can be found in Winston Yii's paper [24]. To determine the size of the median filter required the location of the drone as reported by the server computer was logged in a CSV file. A MATLAB program was then written that generates a video of a 3D plot of the drone location in space as it flies (refer to section 10.6).

Figures 10.17 - 10.19 show that there are a small percentage of outlier pose estimates, which justifies the choice of a 3 point moving median filter. This enables single outliers to be filtered from the data whilst maintaining a low delay between actual location being detected and being enacted upon by the drone.

5.5.3 Flight Control

Several tests were performed to determine the system's ability to move to a goal location and remain there. Firstly the default hovering routine provided by program.elf was tested for comparison.

5.5.3.1 Default Hover Routine (Control Tests)

The default hover routine was tested several times for a plain floor and it was found that it could not be performed indefinitely. In the first test the drone yawed left by 90° within the first 30 seconds of takeoff and hence little data could be logged regarding its location (due to not viewing the scene). In the second test the drone flew upwards, almost to the ceiling and seemed to be unable to stabilise at all. These results may have been caused by interference with the magnetometer from lab equipment or gusts from the air conditioner.

In the control tests for which the drone maintained a view of the scene the location was logged and plotted. Figure 5.22 shows the location of the drone for the first 32 seconds of one test flight, after which the drone yawed and could no longer view the scene. Refer to figure 10.20 for the result of a longer test. The errors associated with the drone drifting when in hover mode are likely due to the uniformity of the floor beneath the drone. When items were placed under the drone this drifting was greatly reduced.

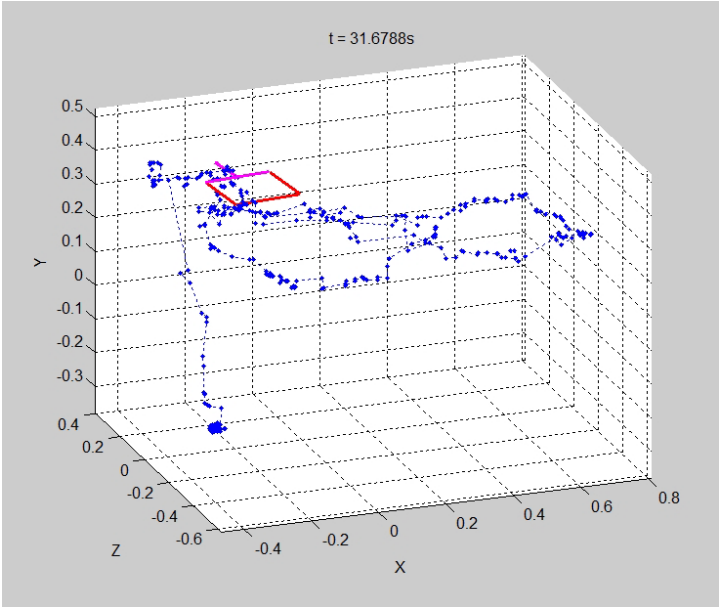


Figure 5.22. The drone path travelled when using the default hovering routine to remain stationary.

5.5.3.1 Localisation-based Control

Several tests were completed and it was found that the drone was able to remain at a desired location relative to the Xtion camera. It was necessary for the scene to have many features for reliable matching performance, so a large poster was used.

Issues were encountered when the drone entered hovering mode due to a lack of matches and the drone could turn to look away from the scene. The slow left and right yawing movements worked well to allow the drone to regain a view of the scene in several cases but not when the drone had moved significantly away from the scene.

Two modes of control were experimented with when the drone reached its goal location. Firstly the drone was commanded to enter program.elf's hover mode when the goal was reached (refer to figure 5.23). This acted to brake the drone when the hover mode started and worked well in most instances. When the drone left the goal due to drift in the hovering it would re-enable localisation-based control and move towards the goal once more. This method did suffer some odd problems where the hover mode would occasionally not relinquish control of the drone, which resulted in it becoming unresponsive. In these cases the drone had to be manually stopped and flipped over to abort the flight. Figure 10.21 depicts a longer test of this flight control method.

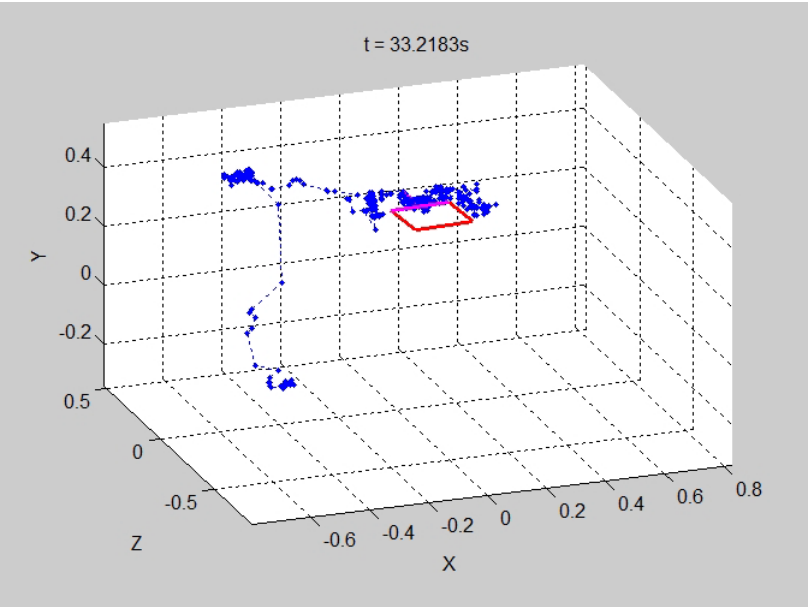


Figure 5.23. The drone path travelled when using localisation to move to the goal location. Hover mode was enabled once the goal was reached.

The second method involved commanding the drone to remain still once the goal was reached by sending all zeroes to program.elf. This resulted in overshoot issues as the drone would have momentum as it reached its goal location and it would continue beyond it. Figure 5.24 shows that the drone tended to oscillate past the goal in this mode whilst figure 10.22 depicts a longer test.

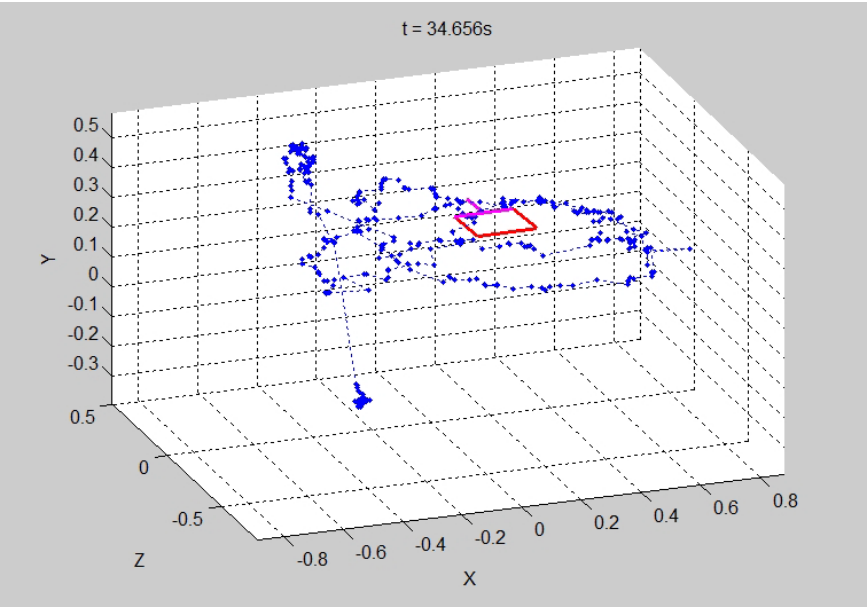


Figure 5.24. The drone path travelled when using localisation to move to the goal location. The drone was commanded to remain still once the goal was reached.

Finally a test was conducted where the drone had to navigate a series of four waypoints, using the default hover to stabilise when a waypoint was reached (refer to figure 5.25). The drone performed this test well, as there was little chance for errors to be introduced by the hovering before a new goal was set. Videos of the tests are available on the project YouTube channel.

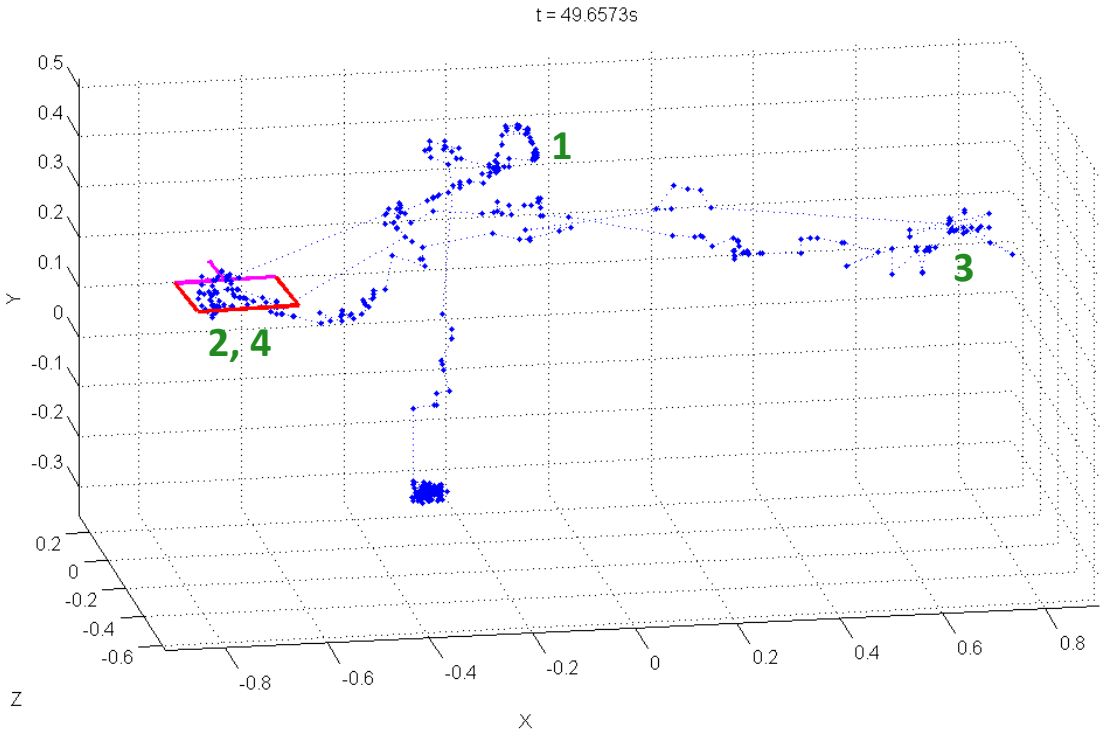


Figure 5.25. The drone path travelled when navigating a series of waypoints, hovering at each waypoint for one second.

6.0 Conclusion

Much has been learnt about the UAV devices produced by Parrot throughout this project. In regard to the aims of the project, several outcomes were accomplished. A literature review of relevant work has been completed and the objective of exploring the drone platforms with regard to their computer vision capabilities has been completed.

The applications implemented on both drones enabled autonomous flight, both in following an object and for navigating waypoints. The AR.Drone successfully performed localisation in a mapped environment where the map was considered to be a scene viewed by the Xtion camera. A full map of a room would have been preferred but this was not available at the time required, and building the map was outside the scope of the project.

The objective of avoiding obstacles and environment boundaries was not completed due to time constraints but this would be relatively simple to add to the application in a basic form as was discussed in section 5.4.2.2. The localisation program runs at 30FPS – the maximum capability of the drone camera, which indicates there is scope for moving more of the processing onboard the drone.

By completing several experiments over the course of the 8 month period, the key findings of this project are:

1. The latency associated with sending camera frames via WIFI to a remote computer for processing is large. Frame loss late is high and resolution is sub-optimal.
2. When performing a simple task such as blob following, the controllability of the system is greatly reduced by sending camera frames via WIFI to a remote computer for processing.
3. It is possible to completely replace the default program that runs on the drone with custom firmware. This can be used to access cameras and sensors whilst controlling the motors.
4. Replacing the default program necessitates implementing custom optical flow stabilisation using the downward facing camera.
5. Custom optical flow stabilisation code runs far too slowly when the implementation is CPU-based.
6. The default program uses approximately ¼ of the available processing power on board the drone, leaving ample room for a second program to run onboard the device.
7. It is possible to control the default program from another program onboard the drone by sending UDP commands.
8. It is possible to gain access to the drone front camera whilst also running the default program with stabilisation.
9. The latency associated with sending commands to the default program from onboard the drone is low enough for accurate control methods.
10. The drone front camera distortion can be accurately modelled by a quintic camera model.
11. Localisation and control can be performed at the frame rate of the drone camera when keypoints and descriptors are sent to a remote computer for matching against data from an RGB-D camera.
12. There is a great deal of scope for future work on the drone that could see full onboard processing for navigation in a large indoor environment.

The project has been a rewarding and informative experience in real time embedded systems.

7.0 Recommendations

Future work on the project can take a variety of directions. The following are a number of possible tasks that could be completed.

7.1 Map Navigation

The next logical step of the project is to use the Xtion camera to create a full map of a room by stitching together frames. Once a complete map is built and scaled to the correct size, with loop closure applied, the keypoints and descriptors for the whole room can be computed (offline). This information can then be used with the application seen in section 5.4 to allow the drone to fly around the whole room.

There are several advantages to this approach. Firstly, it requires less computation on the server, possibly allowing for more rigorous matching algorithms. Secondly, there are no longer any issues associated with the drone and Xtion camera views not overlapping. This was a problem with the application tested, because as soon as the drone looked away from the scene that the Xtion was viewing, localisation became impossible. Finally, there are no issues with blur created by moving the Xtion camera. The model of the environment can be made to fill all gaps so that the best matching possibilities arise.

7.2 Drone Sensor Aiding

The drone's sensor data can be used to determine the height and yaw angle information absolutely. This gives the advantage of reducing the search area within the 3D map of descriptors as certain keypoints could not possibly be within the drone's view. This could be further extended to allow for full sensor fusion. With this approach, if the drone loses relative pose information, it can use sensor odometry to make an estimate of its location until a new valid pose is found.

7.3 Full Onboard Processing

Once use of the DSP is achieved, it is reasonable that the tasks described in sections 7.1-7.2 could be completed entirely onboard the drone, removing reliance on any external computing power. Further, more of the code could be implemented using NEON SIMD instructions, providing significant speed increases.

7.4 Onboard Depth Camera

A preliminary test was performed with the AR.Drone where the Xtion camera was attached to the drone and it was flown manually. It was found that the drone could still fly but battery life was reduced to around 3 minutes. If the Xtion camera were modified to reduce its weight as much as possible, the camera could be attached to the drone and connected through the USB socket. This would allow the drone to navigate and explore un-mapped environments, either performing SLAM or simply avoiding obstacles.

7.5 Object Manipulation

An electromagnet gripper could be added to the drone, controlled through the USB port and powered by the drone battery. This would allow the drone to identify items that have moved in the environment compared with the map information. It could then pick up these (metallic) objects and return them to their original positions in the map.

8.0 Acknowledgements

8.1 Dr. Wai Ho Li

This project was an idea that I personally thought of and as such not an option provided by any supervisor at Monash University. When I approached Wai Ho with my idea he responded positively and was very much open to my thoughts, for which I am very grateful. As supervisor of the project, Wai Ho provided guidance and insight wherever it was needed. He is an excellent teacher and has directly and indirectly taught me a great deal through the final year of my degree.

8.2 Prof. Tom Drummond

Tom also provided a great deal of insight and help at the weekly group meetings. His expertise with libCVD and FAST corners proved invaluable in determining the source of some nasty program bugs.

8.3 Winston Yii

Winston's localisation application that computes the pose of a smartphone relative to a Kinect was the basis for the tests completed in section 5.4. Many days were spent in the lab modifying his program together for use on the AR.Drone 2.0. He spent much time ironing out issues with the WIFI communications and improving the robustness of the matching algorithms. This proved vital to the success of the experiment and is greatly appreciated.

9.0 References

[1] E. Deligne, “ARDrone corruption,” *Journal in Computer Virology*, vol. 8, no. 1–2, pp. 15–27, Dec. 2011.

[2] C. Bills, J. Chen, and A. Saxena, “Autonomous MAV flight in indoor environments using single image perspective cues,” *2011 IEEE International Conference on Robotics and Automation*, pp. 5776–5783, May 2011.

[3] E. Graether and F. Mueller, “Joggobot: a flying robot as jogging companion,” *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, pp. 1–4, 2012.

[4] K. Minatani and T. Watanabe, “A Non-visual Interface for Tasks Requiring Rapid Recognition and Response,” *Computers Helping People with Special Needs*, pp. 630–635, 2012.

[5] N. Berezny and L. D. Greef, “Accessible aerial autonomy,” *IEEE International Conference on Technologies for Practical Robot Applications*, pp. 53–58, 2012.

[6] A. Benini, A. Mancini, and S. Longhi, “An IMU/UWB/Vision-based Extended Kalman Filter for Mini-UAV Localization in Indoor Environment using 802.15.4a Wireless Sensor Network,” *Journal of Intelligent & Robotic Systems*, Aug. 2012.

[7] G. H. Lee, F. Fraundorfer, and M. Pollefeys, “MAV visual SLAM with plane constraint,” *2011 IEEE International Conference on Robotics and Automation*, pp. 3139–3144, May 2011.

[8] V. Ghadiok, J. Goldin, and W. Ren, “On the design and development of attitude stabilization, vision-based navigation, and aerial gripping for a low-cost quadrotor,” *Autonomous Robots*, vol. 33, no. 1–2, pp. 41–68, Mar. 2012.

[9] M. Bošnak and S. Blažič, “Sparse VSLAM with camera-equipped quadrocopter,” *Autonomous and Intelligent Systems*, pp. 135–140, 2012.

[10] C. Ratanasawanya, M. Mehrandezh, and R. Paranjape, *Nonlinear Approaches in Engineering Applications*. New York, NY: Springer New York, 2012, pp. 393–419.

[11] S. Erhard, K. E. Wenzel, and A. Zell, “Flyphone: Visual Self-Localisation Using a Mobile Phone as Onboard Image Processor on a Quadrocopter,” *Journal of Intelligent and Robotic Systems*, vol. 57, no. 1–4, pp. 451–465, Sep. 2009.

[12] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart, “Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments,” *2011 IEEE International Conference on Robotics and Automation*, pp. 3056–3063, May 2011.

[13] S. Shen, N. Michael, and V. Kumar, “Autonomous multi-floor indoor navigation with a computationally constrained MAV,” *2011 IEEE International Conference on Robotics and Automation*, pp. 20–25, May 2011.

[14] L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys, “PIXHAWK: A system for autonomous flight using onboard computer vision,” *2011 IEEE International Conference on Robotics and Automation*, pp. 2992–2997, May 2011.

[15] Parrot SA. (2012). *Parrot AR.Drone* [Online]. Available: http://ardrone.parrot.com/parrot-ar-drone/en

[16] H. Perquin. (2011, September 20). *Tech Toy Hacks* [Online]. Available: http://blog.perquin.com/blog/category/ardrone

[17] Mentor Graphics. (2012). *Sourcery CodeBench Lite Edition for ARM GNU/Linux* [Online]. Available: https://sourcery.mentor.com/GNUToolchain

[18] LinuxTV Developers. (2012). *Linux Media Infrastructure API* [Online]. Available: http://linuxtv.org/downloads/v4l-dvb-apis/index.html

[19] S. Clementson. (2012). *YouTube Channel* [Online]. Available: http://www.youtube.com/channel/UCNkXxiEbBStDsfbKlhoTtxg?feature=results_main

[20] Ziegler, J.G and Nichols, N. B., "Optimum settings for automatic controllers," *Transactions of the ASME*, vol. 64. pp. 759–768, 1942.

[21] Nosaari (user name). (2011, March). *Ardudrone – A WIFI-less RC Controlled AR.Drone mod* [Online]. Available: http://code.google.com/p/ardudrone

[22] R. Longbottom. (2012, January). *Roy Longbottom's PC Benchmark Collection* [Online]. Available: http://www.roylongbottom.org.uk

[23] J. Bouguet. (2010, July 9). *Camera Calibration Toolbox for Matlab* [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc

[24] Z. Zhang, "Flexible camera calibration by viewing a plane from unknown orientations," *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 00, no. c, pp. 0–7, 1999.

[25] W. Yii, W. H. Li, and T. Drummond, "Distributed Visual Processing for Augmented Reality," in *International Symposium on Mixed and Augmented Reality*, 2012.

[26] E. Rosten, R. Porter, and T. Drummond, "Faster and better: a machine learning approach to corner detection.," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 1, pp. 105–19, Jan. 2010.

[27] S. Taylor, E. Rosten, and T. Drummond, "Robust feature matching in 2.3µs," *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 15–22, Jun. 2009.

[28] Fischler, Martin A. and Bolles, Robert C., "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography", in Readings in computer vision: issues, problems, principles, and paradigms, 1987, pp. 726–740.

10.0 Appendix

10.1 CPU Information

Specification	AR.Drone	AR.Drone 2.0
Processor	ARM926EJ-S rev 5 (v5l)	ARMv7 Processor rev 2 (v7l)
BogoMIPS	233.47	998.36
Features	swp half thumb fastmult edsp java	swp half thumb fastmult vfp edsp neon vfpv3
CPU implementer	0x41	0x41
CPU architecture	5TEJ	7
CPU variant	0x0	0x3
CPU part	0x926	0xc08
CPU revision	5	2

Figure 10.1. CPU details comarison for both drones. Obtained by examining /proc/cpuinfo file.

10.2 Drone System Contents

ash	false	mv	sleep
busybox	fgrep	netstat	stat
cat	gdbserver	nfs.sh	stty
channelselector	getopt	pairing_setup.sh	sync
check_update.sh	grep	pidof	tar
checkplf	gunzip	ping	touch
chgrp	gzip	program.elf	true
chmod	hostname	ps	umount
chown	ip	pwd	uname
cp	ipcalc	random_ip	usleep
cttyhack	kill	random_mac	vi
date	ln	repairBoxes	watch
dd	ls	reset_config.sh	wifi_adhoc.sh
df	memory_check.sh	reset_dhcp.sh	wifi_infra.sh
dmesg	mkdir	rm	wifi_managed.sh
echo	mknod	rmdir	wifi_setup.sh
egrep	mktemp	sed	zcat
factory_reset_cb	mount	sh	

Figure 10.2. The contents of the /bin directory on the AR.Drone.

US00_check	board_check	init_gpios.sh	media-ctl
devmem2	mount_usb.sh	umount_usb.sh	dspbridge
parallel-stream.sh	yavta		

Figure 10.3. Additional system contents provided for the AR.Drone 2.0 (in addition to those listed in 10.2).

```
Built-in commands:
------------------
        . : [ [[ alias bg break cd chdir continue echo eval exec exit
        export false fg hash help jobs kill let local pwd read readonly
        return set shift source test times trap true type ulimit umask
        unalias unset wait
```

Figure 10.4. Additional commands provided by the Busybox component. Note these are the same for both drones.

10.3 Bearing Calculation

```
float findBearing(float mag_x, float mag_y){
      //Scale the magnetometer readings and clip them at +/-1
      float x_scaled = (mag_x-20)/60;
      if(x_scaled<-1) x_scaled = -1;
      if(x_scaled>1) x_scaled = 1;

      float y_scaled = (mag_y)/60;
      if(y_scaled<-1) y_scaled = -1;
      if(y_scaled>1) y_scaled = 1;

      //Find the quadrant of the angles and calculate the two possibilities
      float angle1, angle2;
      if(x_scaled<=0 && y_scaled<=0){
            angle1 = asin(x_scaled) + 2*PI;
            angle2 = acos(y_scaled) + PI;
      }
      else if(x_scaled<=0 && y_scaled>0){
            angle1 = -asin(x_scaled) + PI;
            angle2 = acos(y_scaled) + PI;
      }
      else if(x_scaled>0 && y_scaled<=0){
            angle1 = asin(x_scaled);
            angle2 = -acos(y_scaled) + PI;
      }
      else if(x_scaled>0 && y_scaled>0){
            angle1 = -asin(x_scaled) + PI;
            angle2 = -acos(y_scaled) + PI;
      }
      //Return the answer as the average of the two values
      return average(angle1, angle2);
}
```

Figure 10.5. Function used to compute bearing angle from magnetometer sensor readings.

10.4 AR.Drone 2.0 Benchmark Tests

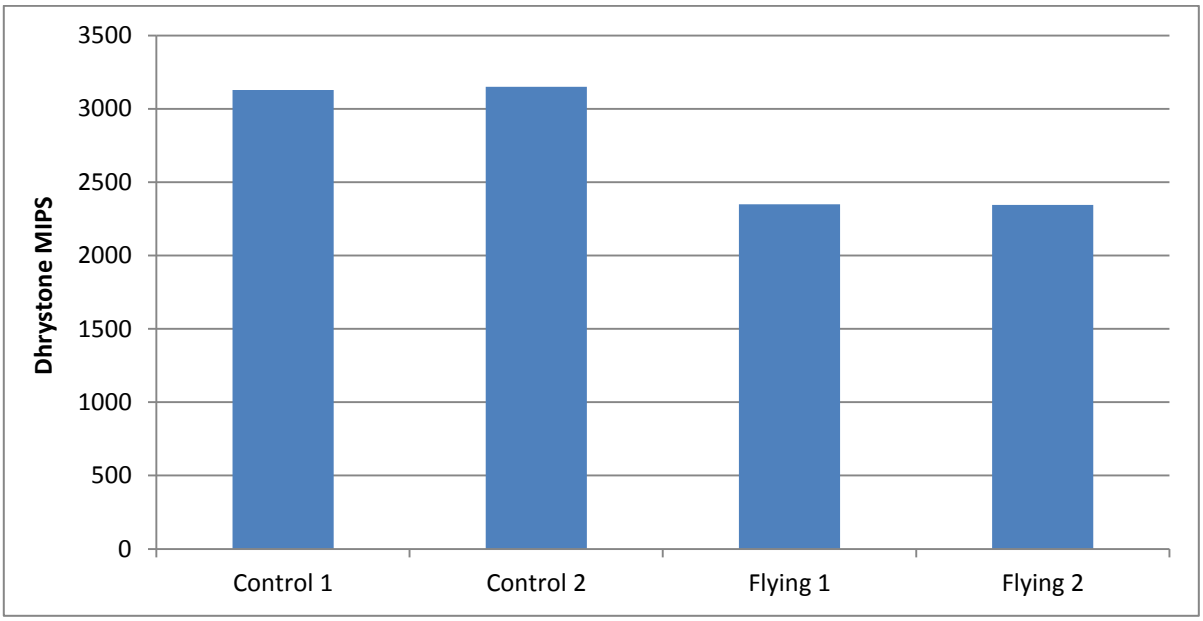


Figure 10.6. Dhrysone MIPS test 1.

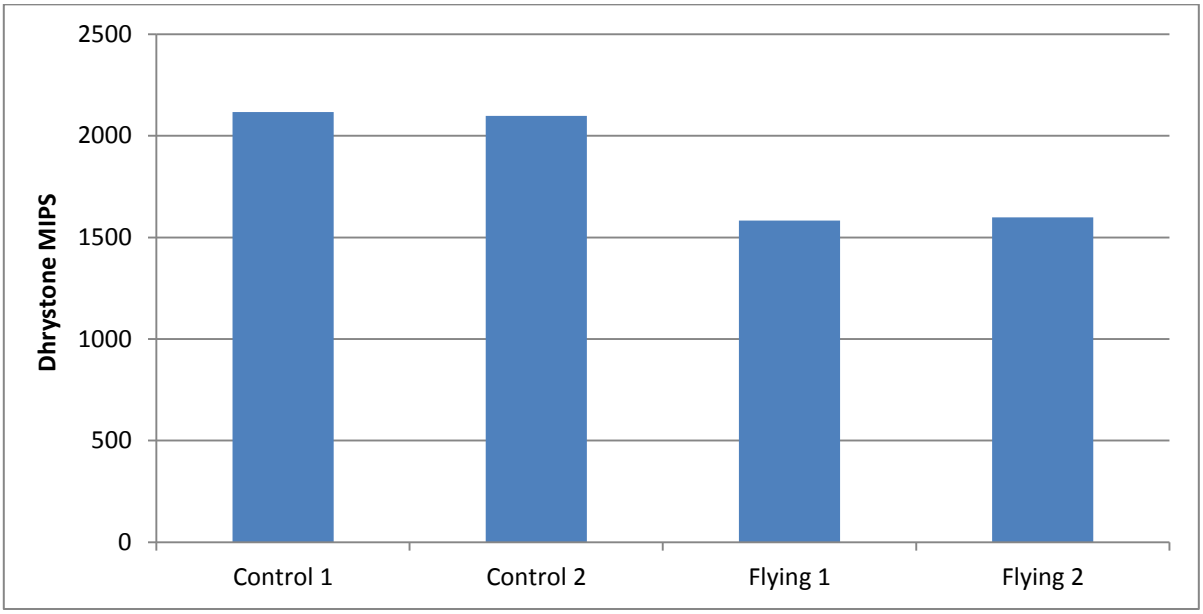


Figure 10.7. Dhrysone MIPS test 2.

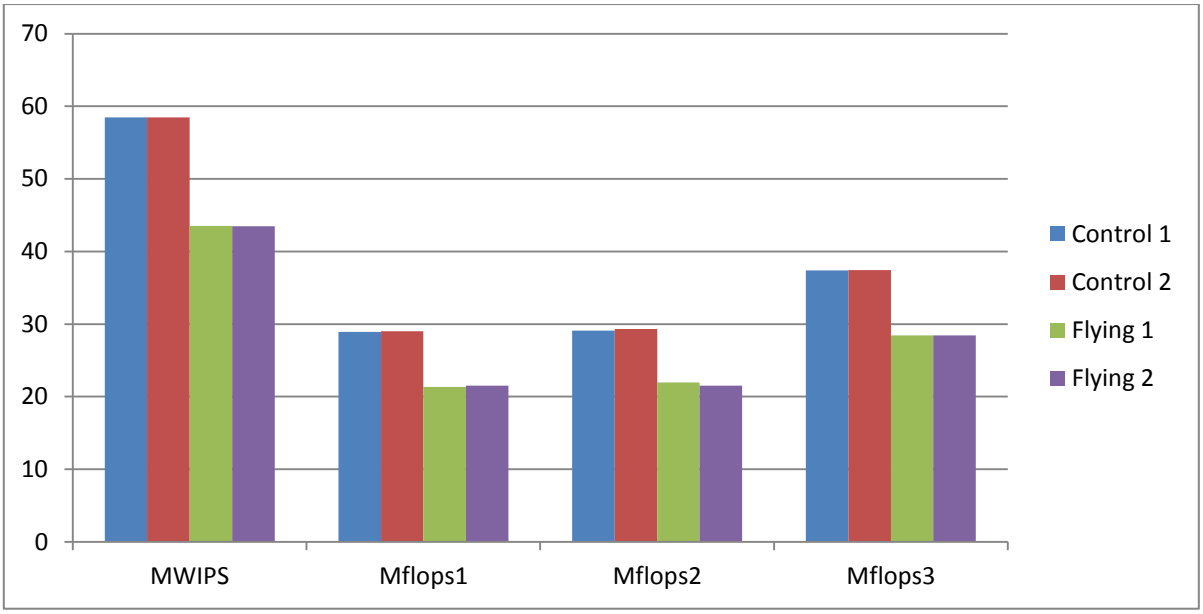


Figure 10.8. Whetstone MWIPS and MFLOPS tests.

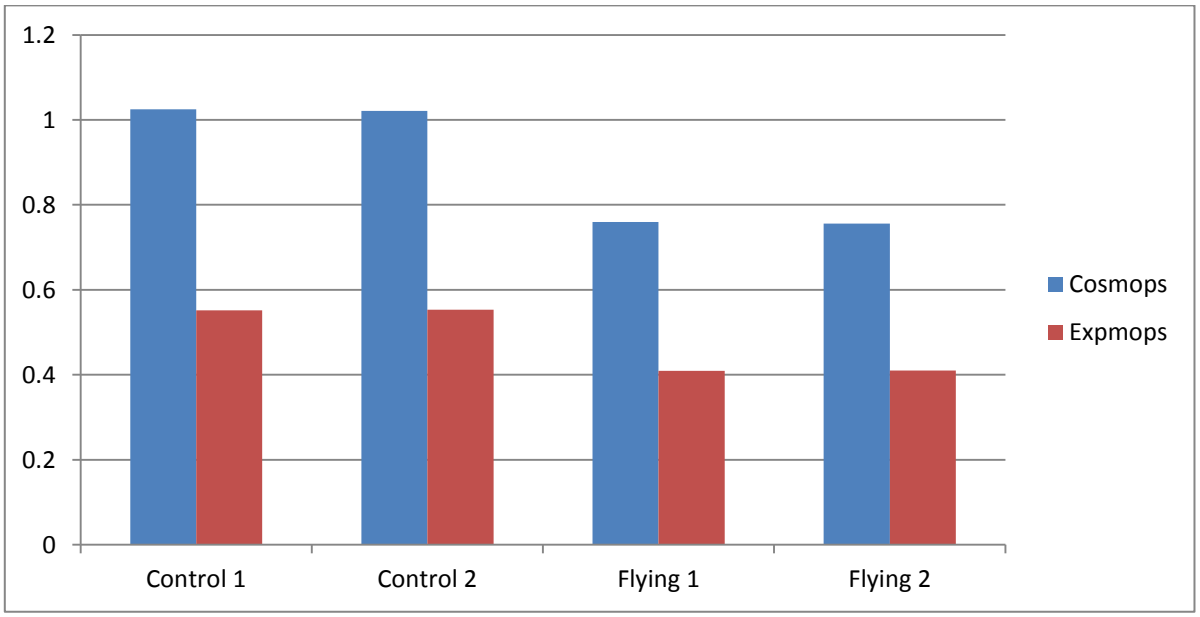


Figure 10.9. Whetstone CosMOPS and ExpMOPS tests.

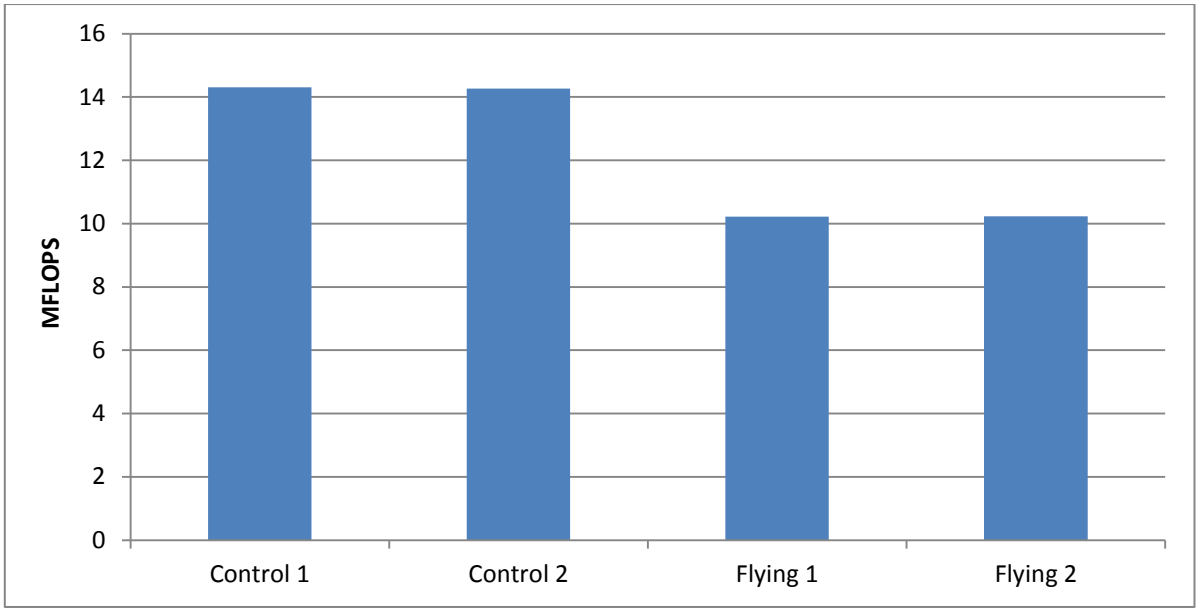


Figure 10.10. Linpack tests.

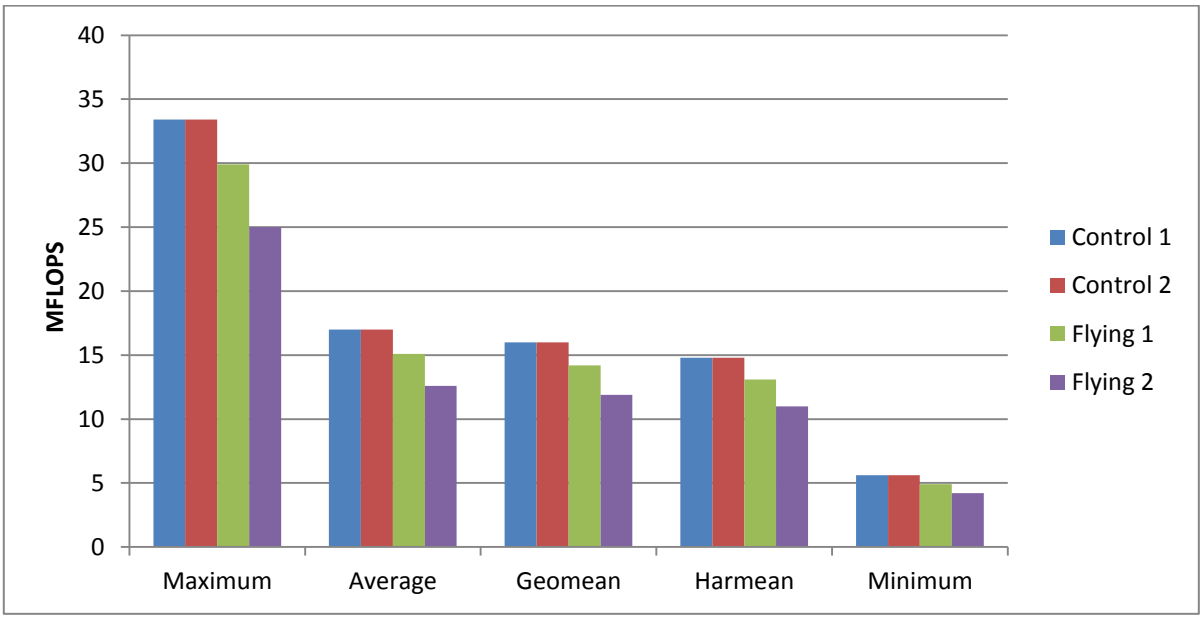


Figure 10.11. Livermore Loops tests.

10.5 AR.Drone 2.0 Front Camera Calibration

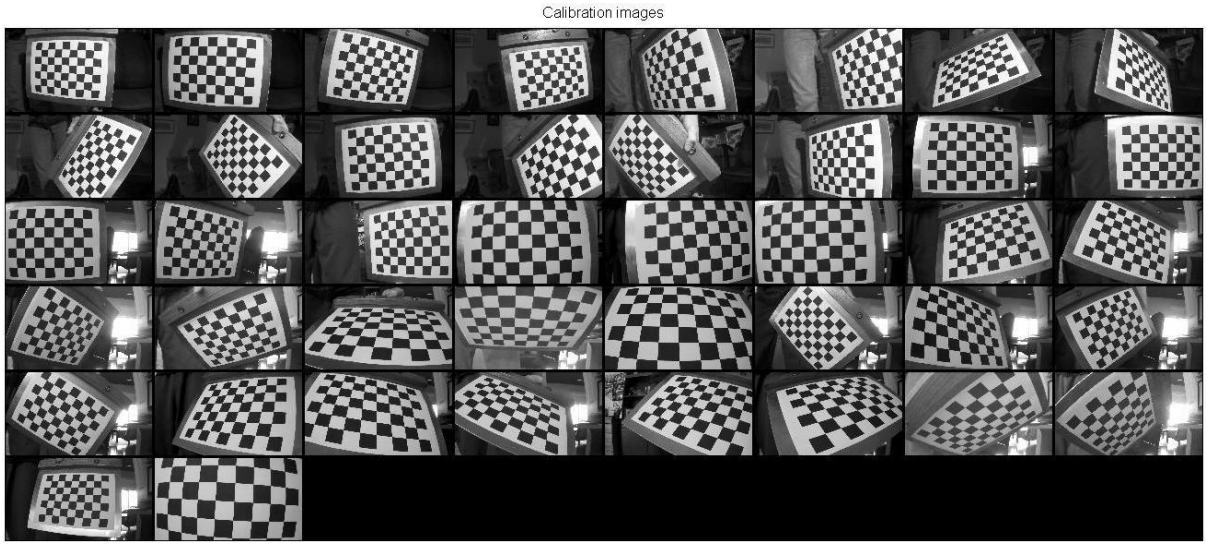


Figure 10.12. Image set used to calibrate the AR.Drone 2.0 front camera.

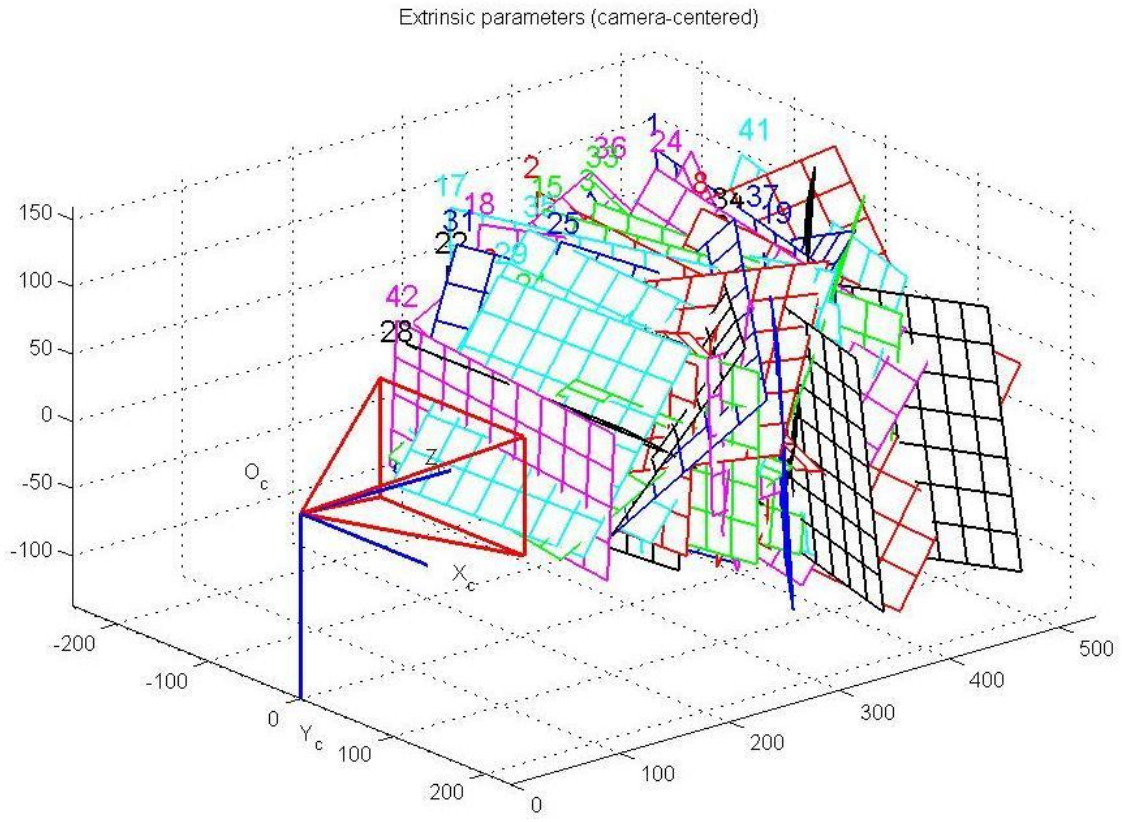


Figure 10.13. Location of the checkerboard relative to the camera in each of the calibration images.

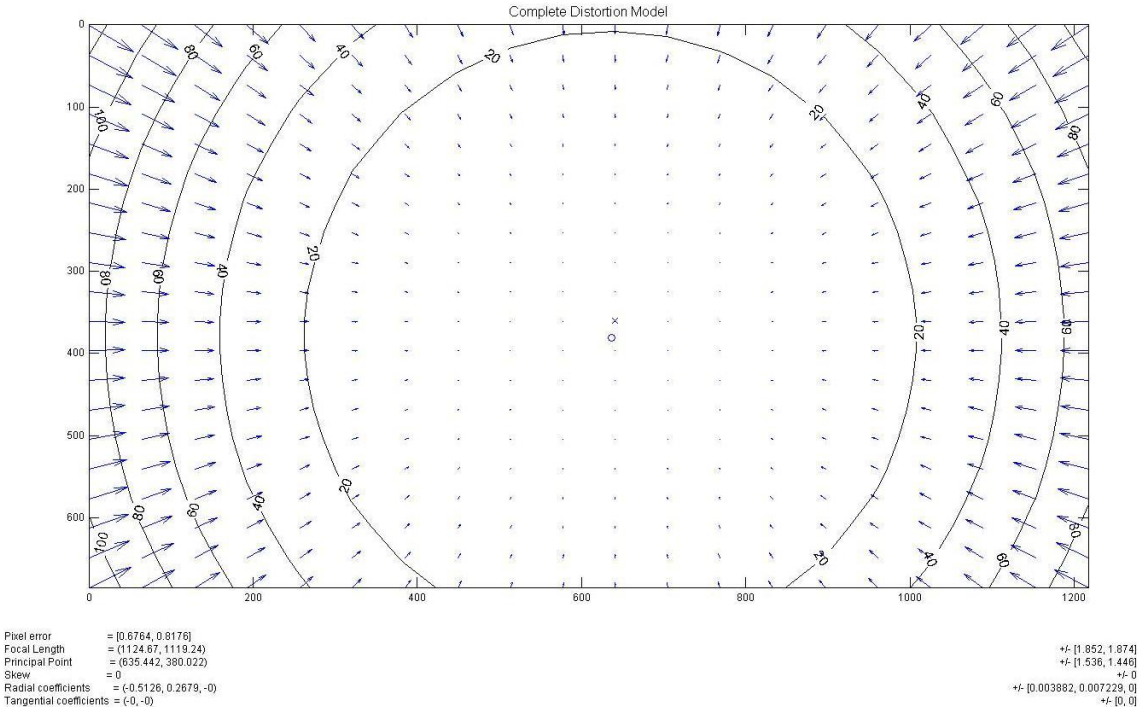


Figure 10.14. Vector field representing the distortion present in the AR.Drone 2.0 front camera.

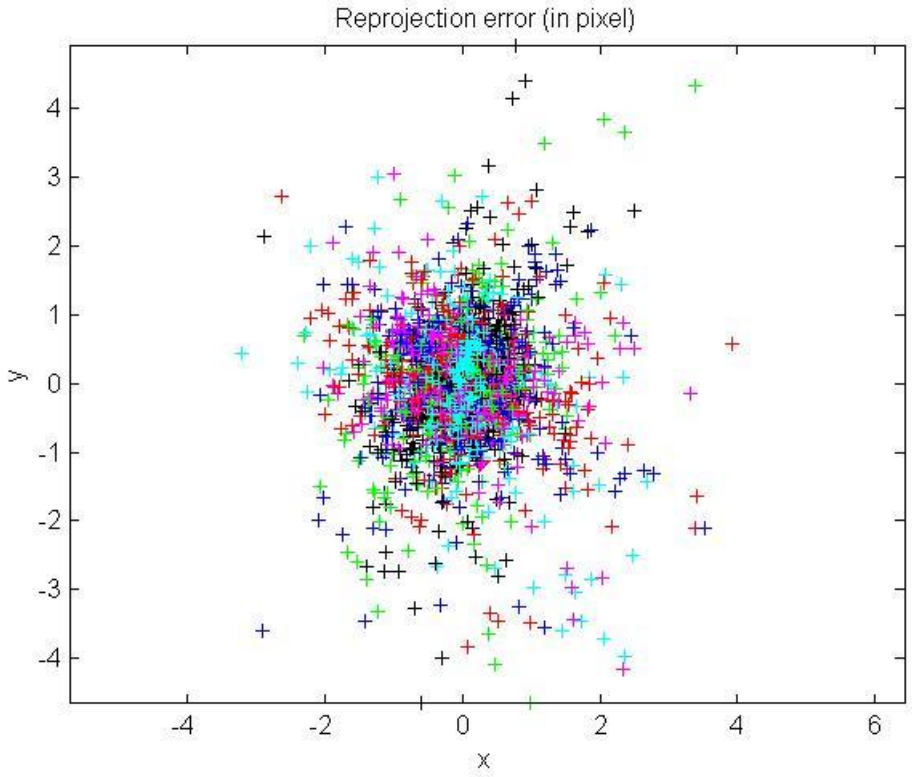


Figure 10.15. Error of keypoint locations on checkerboard when the camera distortion model is fitted.

10.6 MATLAB Plotting Program

```
% Read the data fron csv file and reverse z values
clear;
data = csvread('udp_log.csv', 1, 0);
data(:,3) = -data(:,3);

% Create the video writer object
vid = VideoWriter('plot_output.avi');
vid.Quality = 100;
vid.FrameRate = 30;
open(vid);

% Create the figure for plotting
handle = figure();
hold on; grid on; axis equal;
x_axes = [min(data(:, 2))-0.1 max(data(:, 2))+0.1];
y_axes = [min(data(:, 4))-0.1 max(data(:, 4))+0.1];
z_axes = [min(data(:, 3))-0.1 max(data(:, 3))+0.1];
axis([x_axes y_axes z_axes]);
view(-20, 20);
xlabel('X'); ylabel('Z'); zlabel('Y');
%set(handle, 'Renderer', 'OpenGL');

% Setup variables
previous_wait = data(1, 1);
previous_x = 0;
DELAY = 1/30;
L = 0.1;
pause();

% Loop through data, creating 3D plot and saving frames for the video
for count=1:size(data, 1)
    % Save the same frame again if gap in data
    gap = data(count,1)-previous_wait;
    fill_frames = gap/DELAY;
    for i=1:fill_frames
        title(['t = ', num2str(previous_wait + i*DELAY), 's']);
        writeVideo(vid, getframe(handle));
    end

    previous_wait = data(count,1);
    title(['t = ', num2str(previous_wait), 's']);

    if(previous_x ~= data(count, 2))
        cla;
        plot3(data(1:count, 2), data(1:count, 4), data(1:count, 3), 'b.:');

        % Compute location of drone outline vectors
```

```
        lcostheta = L*cos(data(count, 5));
        lsintheta = L*sin(data(count, 5));
        x = data(count, 2);
        y = data(count, 4);
        z = data(count, 3);
        previous_x = x;

        % Add drone outline to plot
        quiver3(x-lcostheta, y+lsintheta, z, -2.2*lsintheta, -
2.2*lcostheta, 0, '.r', 'LineWidth', 2);
        quiver3(x-lcostheta, y+lsintheta, z, 2.2*lcostheta,  -
2.2*lsintheta, 0, '.m', 'LineWidth', 2);
        quiver3(x+lcostheta-2*lsintheta, y-lsintheta-2*lcostheta, z,
2.2*lsintheta, 2.2*lcostheta, 0, '.r', 'LineWidth', 2);
        quiver3(x+lcostheta-2*lsintheta, y-lsintheta-2*lcostheta, z, -
2.2*lcostheta,  2.2*lsintheta, 0, '.r', 'LineWidth', 2);

        quiver3(x, y, z, 2*lsintheta, 1.5*lcostheta, 0,'m','LineWidth',2);
    end
    writeVideo(vid, getframe(handle));
end

close(vid);
```

Figure 10.16. Program listing for the MATLAB application that generates a video of the 3D path the drone flew.

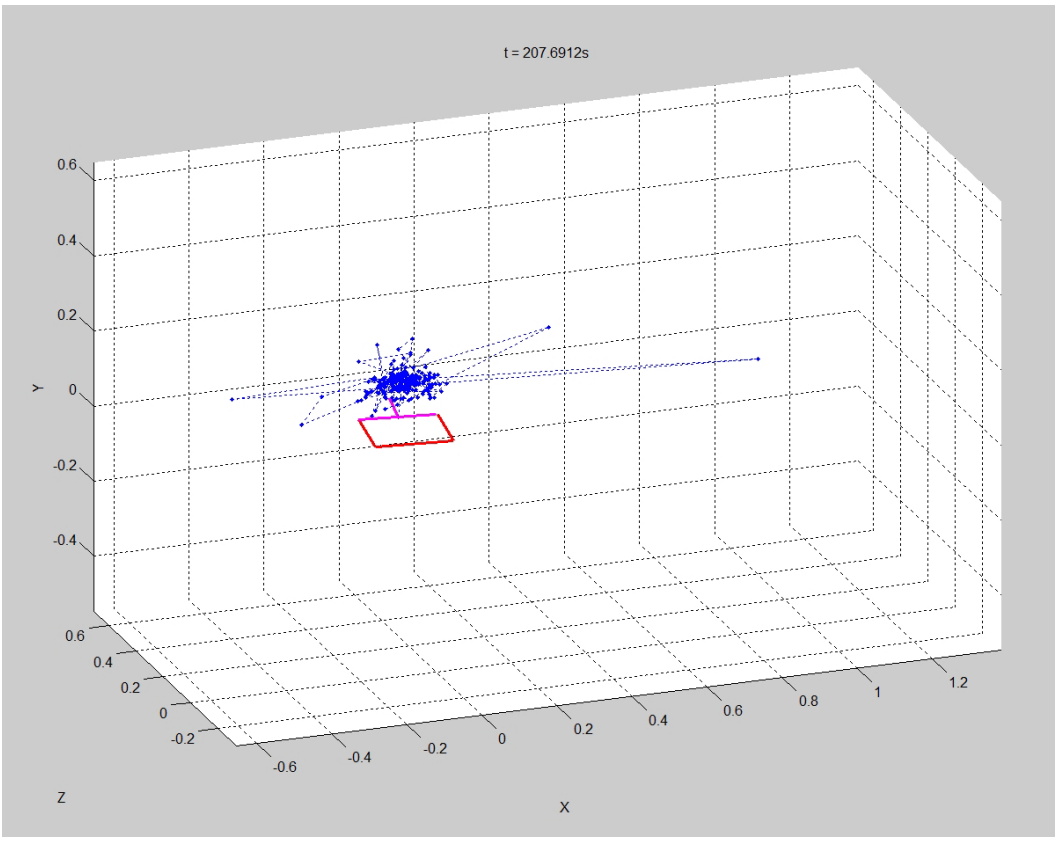


Figure 10.17. Plot of the drone position (without filtering) from time t=194 to t=208.

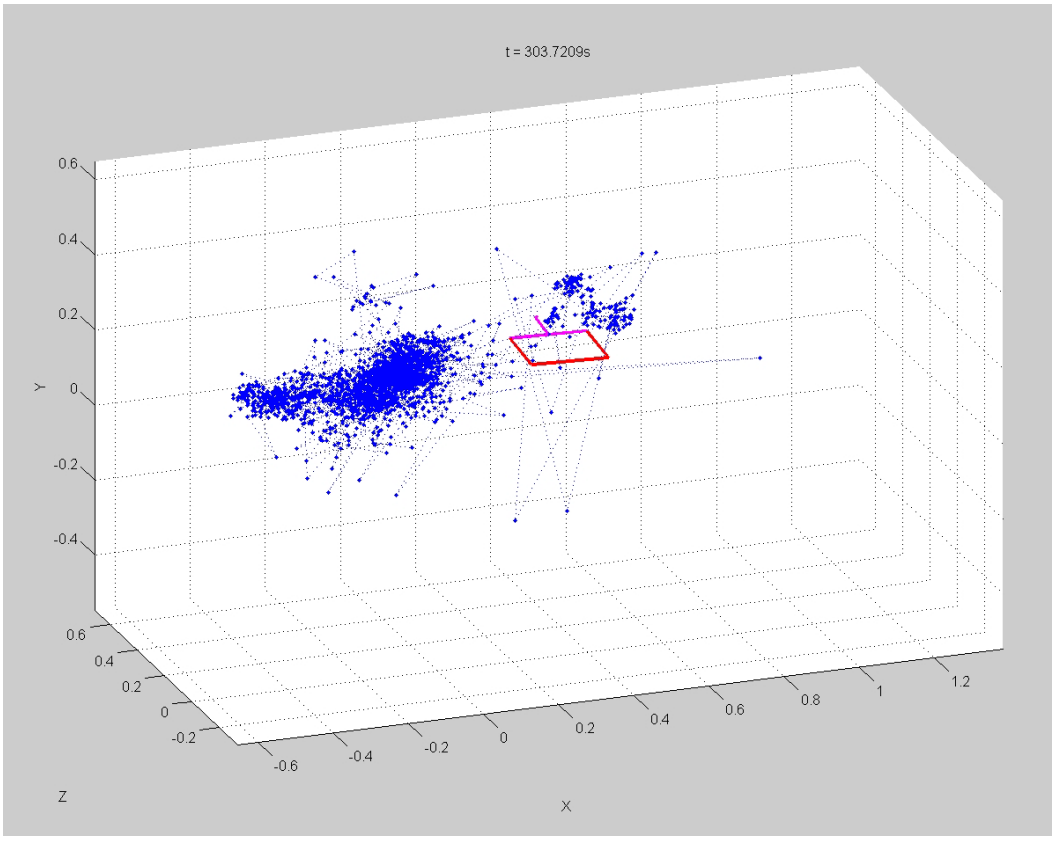


Figure 10.18. Plot of the drone position (without filtering) from time t=194 to t=304.

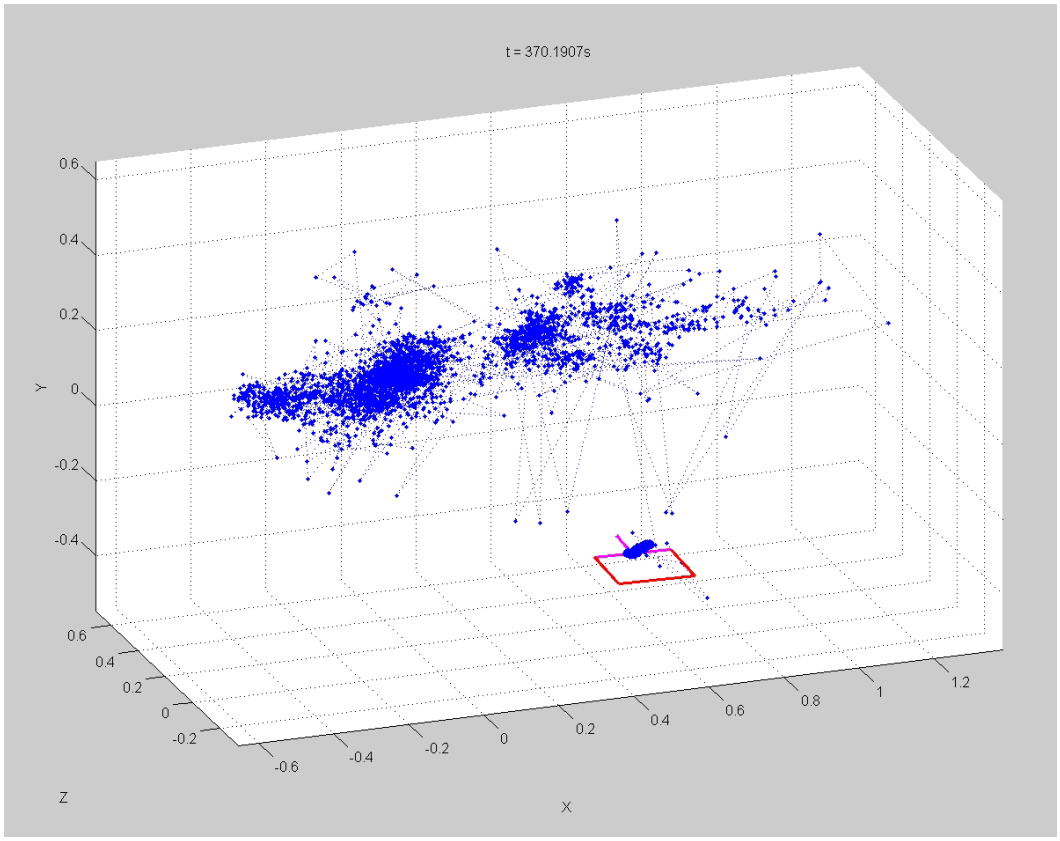


Figure 10.19. Plot of the drone position (without filtering) from time t=194 to t=370.

10.7 Localisation Results

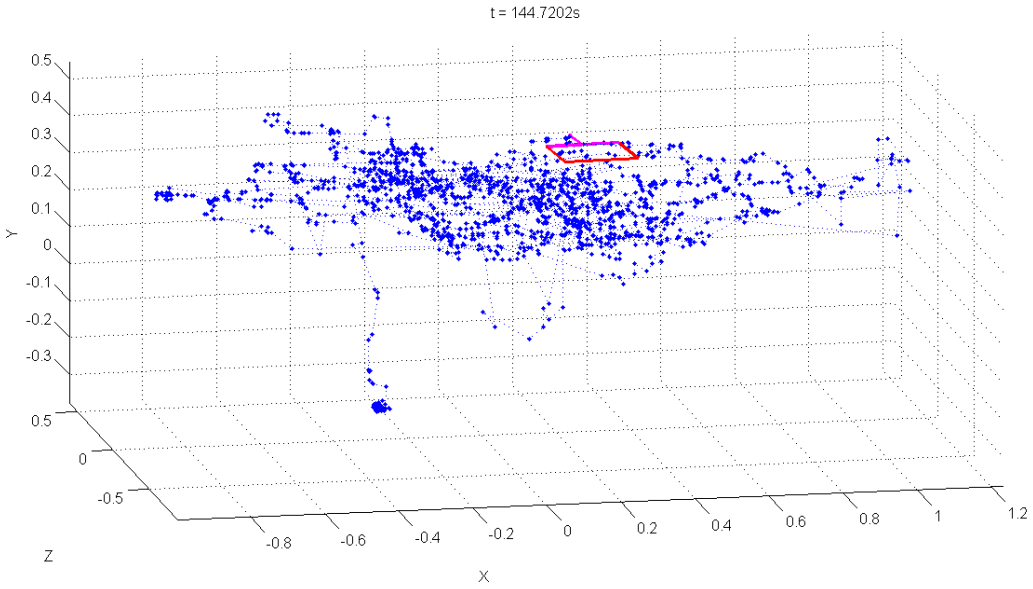


Figure 10.20. Path travelled by the drone when in hover mode for a period of 2:25.

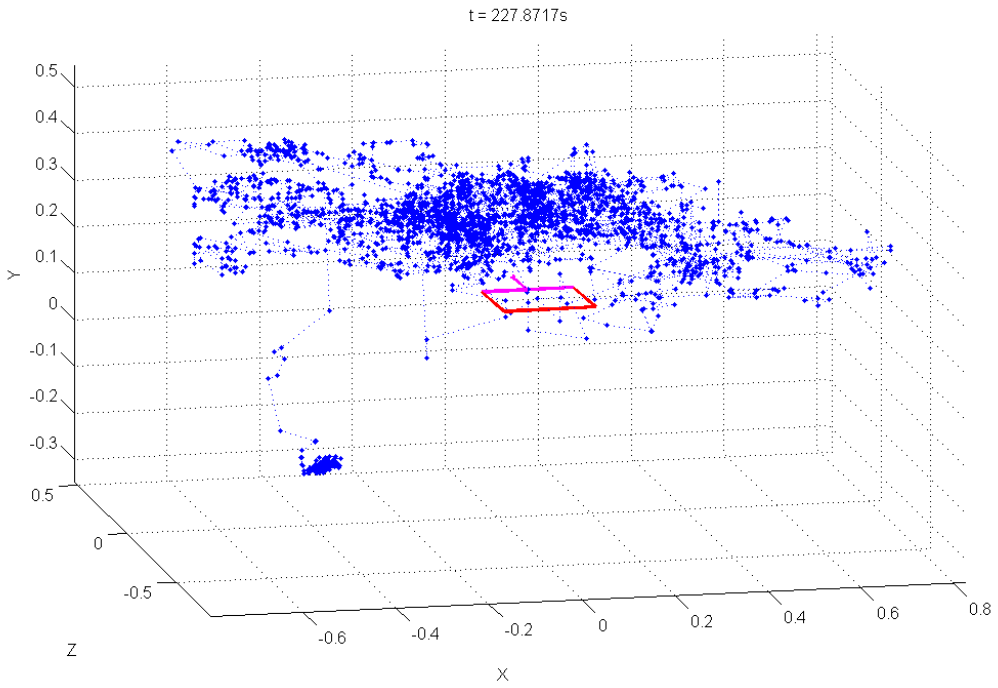


Figure 10.21. The drone path travelled when using localisation to move to the goal location for a period of 3:48. Hover mode was enabled once the goal was reached. Note that the drone was physically pushed away from the gaol multiple times.

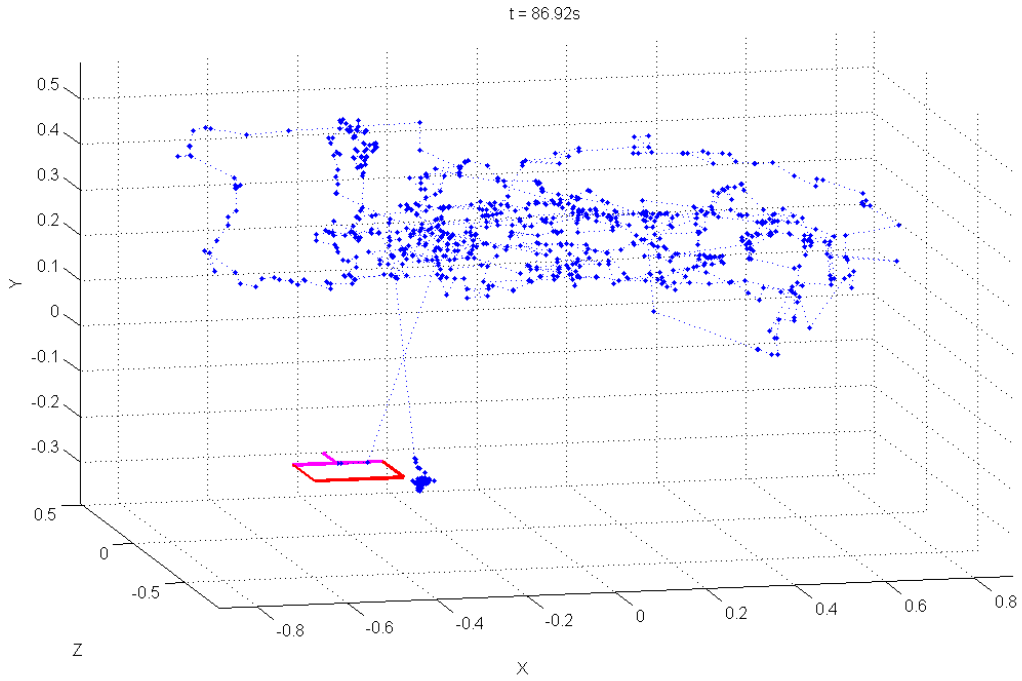


Figure 10.22. The drone path travelled when using localisation to move to the goal location for a period of 1:27. Hover mode was not enabled once the goal was reached.