Vision-based Algorithms for UAV Mimicking Control System

by

Pablo Martinez

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering
University of Alberta

Abstract

Vision-based algorithms designed to detect and track UAVs from an onboard moving platform have been the focus of active and extensive research over the last decade, and dozens of algorithms have been tested, compared and optimized. However, the existing approaches tend to rely on specific features such as color or edges which may not be able to detect and track various types of flying quadcopters. This thesis implements a modified version of an existing vision-based algorithm, the Cascade Classifier, originally designed to recognize facial features and humans, and demonstrates its capability of detecting and track any type of quadcopter with great accuracy over a variety of backgrounds, in both indoors and outdoors flight conditions. The Cascade Classifier algorithm is demonstrated on two specific quadcopter models used for this study, the 3DR Solo and the Parrot AR.Drone 2.0. This thesis introduces a novel method to reduce the amount of information which needs to be processed by vision-based algorithms when tracking physical objects undergoing non-random motion in 3D space. This method employs a Kalman filter to predict the estimated position, velocity and acceleration of the tracked object in order to reduce the image area in which the tracked quadcopter is believed to be. This enables the Cascade Classifier algorithm, or any other type of vision-based detection algorithm to track the target vehicle while greatly reducing the required image processing time. Experimental testing proves that the proposed algorithm obtains good detection and tracking performance in real-time for both quadcopter types in indoor and outdoor flight scenarios, as well as the successful performance of the mimicking control system design.

Preface

This thesis is an original work by Pablo Martinez. No part of this thesis has been previously published.

Acknowledgements

The author of this thesis would like to thank Dr. Martin Barczyk for the support, suggestions and supervision of this thesis. I would like to thank Brendan Calef as my second pilot, without him the initial experimental tests would have been more difficult to take; Ryan Satyabrata for his help with Android multi-threading and coding, Alexander Pon for his amazing cooperation in the algorithm training optimization and Adam Casey for his help porting all into ROS to get more experimental data.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Computer vision has long been of interest to engineering and industrial applications as a substitute to human eyes, for applications such as inspection, surveillance and guidance. Given the ongoing increases in computing power and decrease in cost, size and weight of computer and video hardware, computer vision-based algorithms are now able to perform detection and command appropriate actions faster than even the most experienced human operators. An example of vision systems used today are camera networks, utilizing many wired or wireless cameras spread throughout a site and transmitting video streams to a central computer. Using such systems, recognition and tracking of one or more objects [66, 49] or humans [38, 18] in both single view and multi-view [35, 27] settings have been demonstrated. In order to allow a single camera to move around and obtain multiple camera-like coverage, mounting the camera on an aerial vehicle such as a quadcopter is a logical development.

The quadrotor UAV platform offers a number of advantages over fixed-wing (plane) or two-rotor (helicopter) vehicles, namely mechanical simplicity, ability for hover and flight in any direction, as well as high agility. Given the proliferation of commercially available off-the-shelf quadrotors over the last decade, the cost of purchasing and maintaining a fleet of such vehicles has become affordable. This has also increased the interest of such units for commercial or civil purposes such as photography, journalism, delivery, art or racing, in addition to military and law enforcement focused interests [51, 46].

Academic interest in autonomous quadrotor modeling and flight control can be traced back to the early 2000's. Early work involved using simple PID or LQR control to control the attitude and altitude of a quadrotor and performing easy maneuvers using data from an onboard IMU-based state estimation system [48, 36, 41, 40]. Today, quadcopters have demonstrated formation flight, aggressive acrobatics, as well as autonomous recon of unknown environments [55, 12, 64, 50, 30], with special mention to the Dr. D'Andrea's work in quadrotor aggressive maneuvers and acrobatics at ETH Zurich and Dr. Kumar studies on autonomous recon and multiple quadrotor communications (swarm control) at Pennsylvania State University. Therefore, this thesis will assume that quadcopter motion control is a solved problem and any required motions can be achieved.

Detection and active tracking of target UAVs, which involves measuring its relative position and velocity, can be of interest for military and law enforcement, as well as operators of airports, nuclear power plants or other critical infrastructure. Rogue UAVs are difficult to detect and agile, and can pose an air threat [58], i.e. recently Quebec police reported the increasing use of quadrotors to smuggle cell phones, sim cards, tobacco, marijuana and other drugs into the Bordeaux jail [19]. The ability to detect, track and follow an unauthorized UAV using an onboard monocular camera is a useful capability and an interesting engineering problem [51]. Existent commercial quadrotors already offer follow modes, such as DJI Phantom 4 follow-me and obstacle avoidance modes, however they are not fully vision-based modes.

1.1.1 Motivation of Research

In order to detect and track any kind of target using vision-based sensing, certain defining features of the target need to be identified. Those features can be simply detectable such as color, shape, edges/corners, blobs or ridges, or more complex combinations thereof. This will be the main focus of the vision-based detection algorithm when determining whether or not there is a target in any frame of the onboard video stream. Quadcopters vary in colors and shapes, but their main structure tends to be constant: a fixed X-wing shape with a rotor fixed at each of the four corners. The first stages of work in this thesis are to identify the types of features which can be used to detect and track a quadcopter flying over a variety of backgrounds, and then select a vision-based algorithm to do this. Given the computational constraints involved with onboard or mobile ground station computers, we need to select an algorithm capable of running in real-time, then optimize its accuracy in a variety of conditions.

1.2 State of the Art

1.2.1 Detection and Active Tracking Algorithms

The background subject of detecting and tracking flying targets has been studied in depth since the 1960's military and aerospace companies [3]. This has traditionally been done using big sector antennas or radars, which can be seen at for instance airport control towers [2, 52]. Radar technology has also been miniaturized to fit onboard small UAVs [20]. However, given the cost and power requirements of such systems, radars and antennas have been replaced by various types of lower-cost sensors, such as infrared and ultrasonic range finders or optical flow estimators, often used in tandem (sensor fusion) to increase accuracy. Using such onboard sensors, augmented with lightweight inertial measurement units, have made UAVs capable of performing complex maneuvers while perceiving their environment through obstacle detection and collision avoidance [33, 6].

The decreasing costs of quadrotors, coupled with lightweight onboard sensors and wireless communication, allows cooperative flying between UAVs. Within the target tracking area, for either one or multiple targets, several approaches have recently been demonstrated: active sensing based on cooperative target tracking using UAVs [54], or using matched regional adjacency graphs to correlate video feeds from multiple UAVs to achieve multiple target detection [67]. Aside from multi-vehicle research, image processing for detection and tracking purposes has been extensively studied by the computer vision research community, for applications which vary from satellites (remote imaging) [53, 44] to quadrotors.

Regarding single-camera imaging onboard a quadrotor, performing autonomous tracking of multiple moving objects, either on the ground or in the air, has been demonstrated [29, 14, 11, 65]. However, most if not all of the tracking algorithms rely on markers or specific features of the target to be able to identify it and achieve real-time detection. Thus, these algorithms lack the capacity to be easily adapted to other environments or target types.

1.2.2 Trajectory Tracking Control Systems for UAVs

As discussed in Section 1.1, quadrotor control systems have been developed, tested and optimized for almost two decades. The mimicking control system in this thesis belongs to the class of trajectory tracking control systems, but with a modification in the control loop's reference. Instead of starting with precisely known desired trajectory as in most cases [31, 16], the reference trajectory is actively updated based on the target's motions. This is a similar problem to an infinite-horizon linear quadratic tracking optimal control used to track an unknown trajectory of a moving surface vehicle target [26, 21]. In our case, the trajectory of the target is less certain due to its capability of moving in any direction in 3D space.

1.3 Outline of Thesis

This chapter provided an overview of vision-based algorithms and the motivations for undertaking this research, followed by a survey of related literature. An itemized statement of contributions will be given in Section 1.3.1.

Chapter 2 is a general background summary on the hardware and software that were used for the research presented in this thesis. We first review the specifications of two chosen quadcopters, focusing on the relevant hardware and software features of each UAV type including sensors, connectivity protocols, and code development capacity. Next, we cover the main hardware components in more depth such as GPS, Wi-Fi and several inertial sensors whilst reviewing the principal equations for GPS positioning, Wi-Fi connectivity delays and the inertial sensor accuracy for both quadcopter types. Finally we detail the software programs and libraries used for the development of this thesis and the reasons why these were picked.

In Chapter 3 we cover the vision-based algorithms which were tested, the results obtained, and discuss their limitations for detecting and tracking quadcopters. First, we review the basic color-based algorithms and give the problems associated with dealing with moving backgrounds and moving cameras. Afterwards we design a slight variation of the Histogram of Oriented Gradients (HOG) algorithm, adding Features from Accelerated Segment Test (FAST) and Hungarian Assignment as well as Kalman Filtering, which is shown to be promising but not sufficient for our specific application. Finally we cover the Cascade Classifier Algorithm and the steps required to customize it in order to make it suitable for quadcopter detection and tracking. This algorithm is then optimized using the available tuning parameters, and its results are compared with the previously tested algorithms.

Chapter 4 treats, for both quadcopter models, the logic and control system designs as well as the motion estimation results from the vision-based algorithms seen in Chapter 3. Furthermore, we detail the user interface created to interact with the pursuing quadcopter, broken down into three principal modules: setup, scan, and scan and follow states. For the last state, both quadcopters has different physical inputs, thus some changes had to be made to use the results from the vision-based algorithm: GPS transformations were needed for the outdoor quadcopter model, while thrust calibrations were needed for the indoor model. The results for the Cascade Classifier were proven to be sufficiently reliable for the mimicking algorithm to work, although performance was degraded by background noise in certain situations. A novel method was introduced to reduce background noise while also improving the accuracy of the detection and tracking algorithms — we named this the image reduction method. A Discrete Kalman Filter or Unscented Kalman Filter predicts the position of the centroid of the target and, limiting the processed image area to a smaller zone around the predicted centroid, the probability of finding the target is maximized. This method speeds up image processing, saving memory and computational time, which are important when operating on platforms with limited computing power.

Chapter 5 summarizes the work done in the thesis and the resulting findings. Possible future research tasks which build on the present thesis are discussed.

1.3.1 Statement of Contributions

The following items are claimed as research contributions of this thesis (listed in order of appearance):

- Modifying the HOG algorithm for quadrotor detection and integrating FAST and Hungarian Assignment with Kalman filtering as seen in Section 3.3, which provides an explicit flow chart for how these algorithms are used.

- Optimizing Cascade Classifier training for quadcopter detection and tracking for both indoor and outdoor flying as seen in Section 3.4.2.

- Developing a model to obtain the 3D relative motion of a target quadcopter, given the target's centroid 2D pixel image coordinates and its corresponding bounding box obtained from a vision-based algorithm, as seen in Section 4.4.

- Developing a novel method to reduce the computation load for the vision-based algorithms. This method applies one step-ahead predictors through DKF or UKF to reduce the image area to be analyzed for the target to be found, and is covered in Section 4.5.

Chapter 2

Hardware and Software Background

2.1 Quadcopter specifications

The scanning algorithm was developed for two separate quadcopter UAVs: an outdoor model, the 3DR Solo, and an indoor model, the AR.Drone Parrot 2.0. Both of these models are equipped with on-board processors, however these are already dedicated to running the various on-board control and communication routines, and so it was decided that running processor-intensive video processing on these would not be feasible.

Two different approaches were considered: either adding a second onboard processor, e.g. a Raspberry Pi, to run the scanning algorithm, or run the algorithm on a ground computer and send the relevant results to the drone. The advantage of an onboard processor is avoiding the obvious time delays associated with transmitting data. This delay is more pronounced for outdoor flights, since it increases proportionally with distance. However, running the algorithm on a ground computer is an easier approach, because it allows using the stock flight configuration, and makes it easy to regain manual flight control in the event of a software fault. The resulting time delays may lead to some desynchronization effects in the mimicking, but will not destabilize the onboard flight control system.

A short-to-medium range wireless communication protocol is thus needed to communicate with the drone with as little delay as possible. An easy solution is to use the Wi-Fi IEEE 802.11n protocol, already supported by both drone models and whose propagation delay of about 1 μs per 300 meters is acceptable.

The following two drones are controlled via Wi-Fi from a custom Android app which allows the user to send commands and view the remote video and telemetry information in nearly real time. Since the objective is to extend the drones' existing flight control systems to allow mimicking, it is paramount that the platform be open-source. Other important parameters are the quality of the onboard camera and the accuracy of the onboard sensors.

2.1.1 3DR Solo

The 3DR Solo drone is a quadcopter specifically designed for outdoor videography by 3DR, based on a classic quadrotor design. The hardware is shock mounted inside an ABS plastic hard case, making the drone resistant to crashes to a certain degree. Inside the hard case, individual compartments are shielded with EMI-absorbing polymer shields to avoid interference between sensors. With an average mass of 1.6 kg, the drone easily resists winds up to 12 km/h. It has a battery life of 15-20 minutes, depending on the situation, and is powered by four 880kV motors with 10" diameter glass-reinforced nylon self-tightening propellers with a 4.5" pitch. The maximum communication range is 1.6 km for the stock antennas, which can be extended by further amplifying the signal.

Figure 2.1: 3DR Solo Drone and Controller

The 3DR Solo is powered by two 1 GHz ARM Cortex-A9 Linux-based microprocessors, one on the drone and one in the controller, connected via a dedicated Wi-Fi signal employing the 3DR Link protocol. This allows the drone to stream live 720p HD video from the onboard GoPro Hero 4 camera to the 3DR Solo Android app running on a ground computer. The drone is equipped with an off-the-shelf 3-axis stabilized gimbal to obtain stable shots even in turbulent flight conditions.

The control system uses the well-known Pixhawk 2 with APM:Copter as the internal software. The source code for the Solo App for Android, Pixhawk, and all the flight software developed by 3DR is available on GitHub. In addition, 3DR has made available a developer platform called DroneKit to help developers design applications to connect with the drone in both Python and Android, which will be described in depth in section 2.3.3.

The figure below is a schematic diagram of the main elements of the Solo architecture.

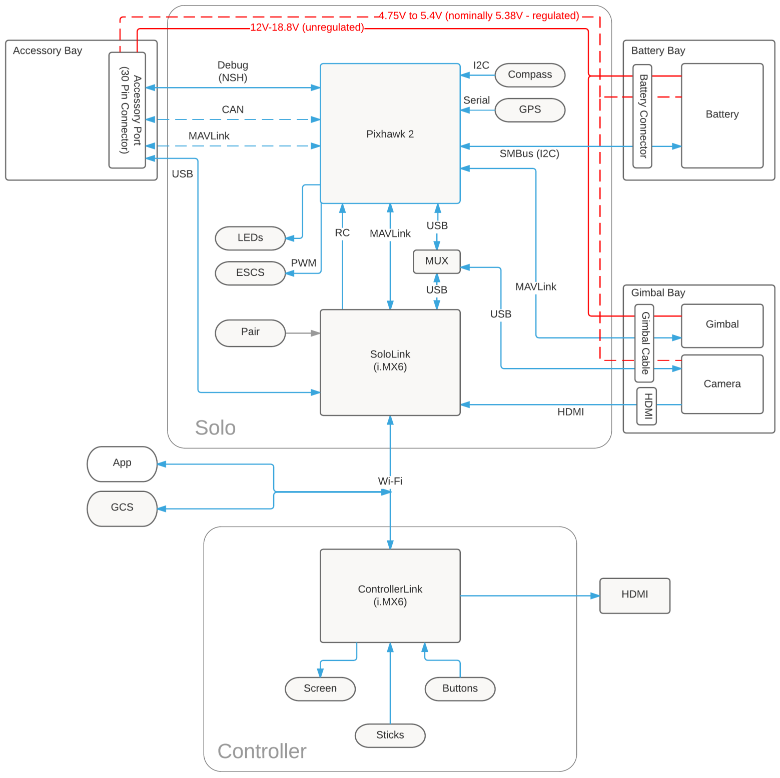


Figure 2.2: 3DR Solo System Diagram [47]

2.1.2 Parrot AR.Drone 2.0

The Parrot AR.Drone 2.0 is a smaller and lighter drone based on a classic quadrotor platform, designed primarily for indoor flights but capable of flying outdoors as well. It has a battery life of 12 minutes and a maximum range of 50 meters, and is equipped with 4 ”in-runner” type brushless 14.5 W motors running at 28 500 rpm. The four propellers are supported on carbon-fiber tubes connected to a plastic fiber-reinforced central cross member. The carbon structure carries an Expanded Polypropylene (EPP) body carrying the Lithium-Polymer battery, onboard camera and inertial sensors, and processing unit. A Styrofoam hull covers the body and surrounds the propellers. The Styrofoam hull makes indoor flight testing safe by shielding the propellers against contact with walls and other obstacles.

Figure 2.3: Parrot AR.Drone 2.0

The Parrot AR.Drone 2.0 runs Linux 2.6.32 on an ARM Cortex A8 1 GHz 32-bit processor with a dedicated video DSP chip running at 800 MHz, allowing its 720p 30fps HD camera video to be transmitted live through Wi-Fi to the AR Free Flight Android app written by Parrot. The source code for the AR Free Flight App can be found on GitHub. The onboard computer runs multiple threads simultaneously: Wi-Fi communications, video data sampling and compressing for wireless transmission, image processing, sensors acquisition, state estimation and closed-loop control. The data acquisition and control threads are run at a 200 Hz rate [10]. The onboard code is closed-source, however.

The drone's onboard sensors are listed in Table 2.1.

Table 2.1: AR.Drone 2.0 Sensors [42]

Sensor Type	**Accuracy**
3-Axis Gyroscope	2,000 degrees/s
3-Axis Accelerometer	+/- 50 mg
3-Axis Magnetometer	6 degrees
Barometer	+/- 10 Pa
Altitude Ultrasound Sensor	-

2.2 Hardware

Throughout this Section, the hardware components making up the control system of each drone will be described. These devices provide the data to the onboard flight control, and as such, the performance of the system depends on them.

2.2.1 GPS

The Global Positioning System (GPS) provides location and time information anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites. The GPS system is used by military, civil, and commercial users all around the world. Competing Global Navigation Satellite Systems (GNSS) such as Galileo (European Union), GLONASS (Russia) and BeiDou (China) are also in operation.

The GPS concept is based on time and the known position of specialized satellites. The satellites carry very stable atomic clocks which are synchronized to each other and to ground clocks. GPS satellites continuously transmit their current time and position. A GPS receiver monitors multiple satellites and triangulates the signals in order to determine its precise position. At a minimum, four satellites must be in view of the receiver for it to compute four unknown quantities (three position coordinates and its clock deviation from the true time). [39, p. 16]

The receiver reports position as a set of coordinates, latitude and longitude, whose format is usually in degrees with minutes and seconds or decimal degrees as in figure 2.4.

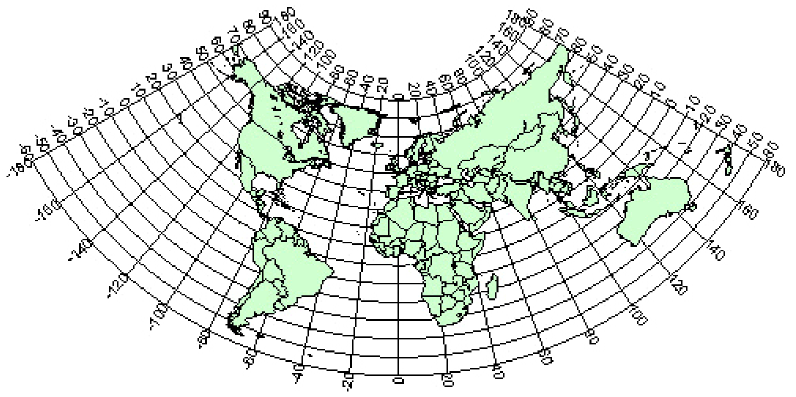


Figure 2.4: World Geodetic Coordinates [57]

For outdoor flights, GPS provides positioning information to the drone. For the 3DR Solo, this is used to obtain a hover within a radius of around a half meter.

Additionally, the input of the Solo flight control systems consists of a set of desired latitude and longitude coordinates, which are compared against the measured coordinates. This means that the implementation of the mimicking algorithm for the 3DR Solo needs to output a set of GPS coordinates for the drone to fly to.

Because the scanning algorithm reports the position of the second drone with respect to a reference frame fixed to the UAV body, its results need to be converted into geodetic longitude and latitude coordinates for the flight control system. The GPS transformation used by the control system will be covered in more depth in section 4.2.6.

Although GPS is a powerful technology for localizing the UAV, it is not perfect. Signal lock is not perfect and can be lost during flight. That can happen if the line of sight between the GPS receiver and the satellites is blocked, e.g. when flying close to tall buildings. This leads to a loss of hover stabilization, and may lead to a drone crash if the human operator is unable to manually take over the flight controls. Another issue is magnetic interference, such as when the drone is close to a ferromagnetic structure, in which case the estimates of attitude may be wrong.

2.2.2 Wi-Fi

Wi-Fi is a technology which allows electronic devices to connect to a wireless LAN network, mainly using UHF to SHF ISM radio bands around 2.4 and 5 GHz frequencies. The communication protocol is specified in the IEEE 802.11 standard. Devices which use Wi-Fi include computers, consoles, smartphones, digital cameras, tablet computers or wireless printers. Although signal interference may happen and the protocol is not heavily encrypted, it is sufficiently good for flight purposes and is used by both the Solo and the Parrot drones.

For indoor Wi-Fi connections, interference problems are typically low given that the distance between the emitter and the receiver is small, although signal loss is important through walls, as represented in figure 2.5. Given the high frequency of Wi-Fi signals, the time delay of the signal traveling through the air is almost non-existent.

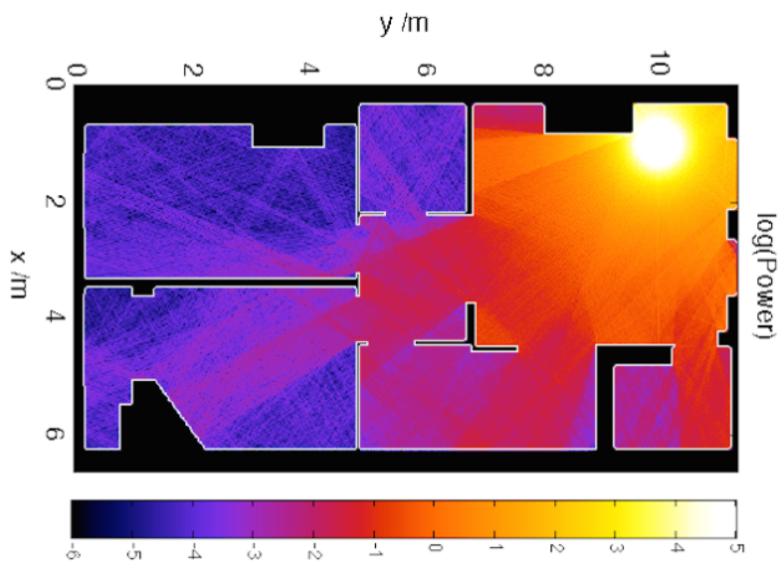


Figure 2.5: Wi-Fi Signal Propagation Indoors

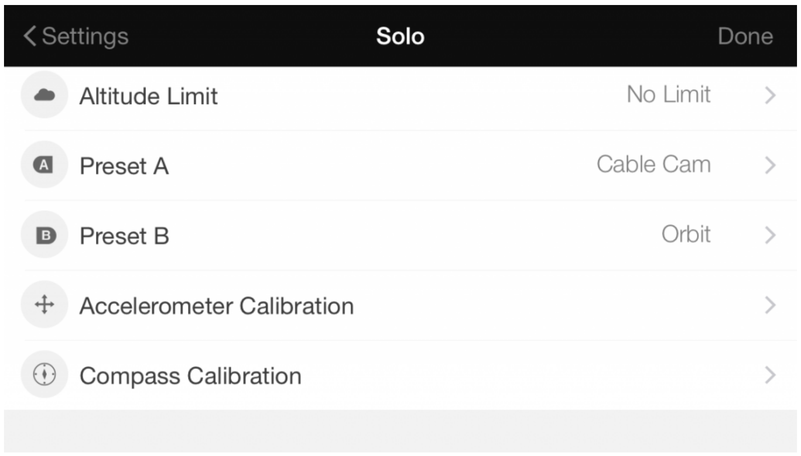
Source: www.technews.tw

In the case of outdoor Wi-Fi connections, long distances may cause signal or packet loss. Moreover, any large obstacles such as trees or buildings can interfere in the signal transmission between controller and UAV.

2.2.3 Inertial Sensors

In both UAV models, the use of low cost inertial sensors means that sensor bias, misalignment angles and scale factors are significant and cannot be neglected. For this reason, a calibration has to be performed in order to obtain correct readings.

The 3DR Solo has the software capability to automatically calibrate compass, level sensors, accelerometers and gyroscopes, as shown below:

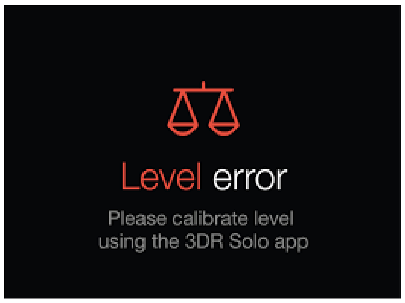


Settings | Solo | Done

Altitude Limit	No Limit
Preset A	Cable Cam
Preset B	Orbit
Accelerometer Calibration	
Compass Calibration	

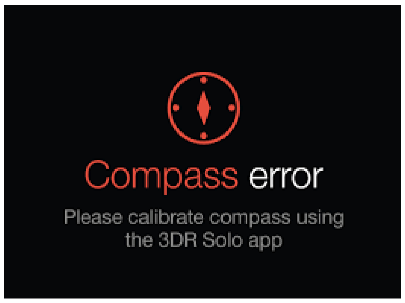
Figure 2.6: Solo App Calibration Options

Moreover, it actively validates the output of those sensors and give errors when the readings are incorrect, as can be seen in figures 2.7 and 2.8.



Level error

Please calibrate level using the 3DR Solo app

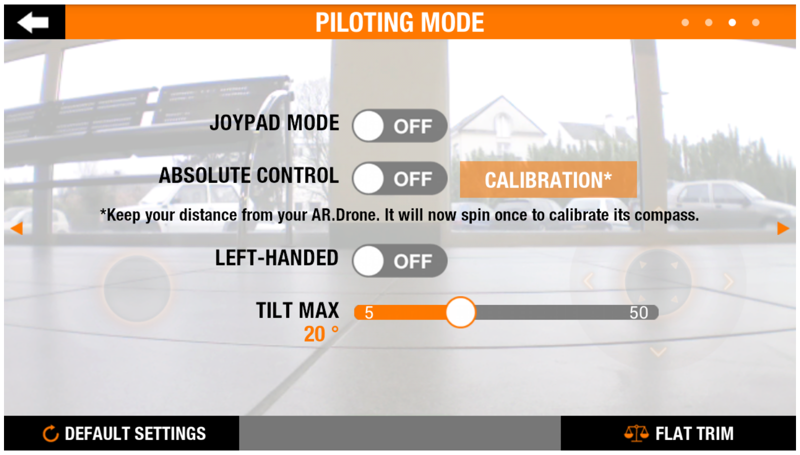


Compass error

Please calibrate compass using the 3DR Solo app

Figure 2.7: Level Calibration Error   Figure 2.8: Compass Calibration Error

Meanwhile, the Parrot AR.Drone 2.0 inertial sensor calibration is not as advanced. Although the compass and accelerometer sensors can be calibrated in the application settings, as seen in figure 2.9, other data (e.g. tilt sensing) cannot be modified.



PILOTING MODE

JOYPAD MODE OFF

ABSOLUTE CONTROL OFF CALIBRATION*

*Keep your distance from your AR.Drone. It will now spin once to calibrate its compass.

LEFT-HANDED OFF

TILT MAX 20 ° (5 – 50)

DEFAULT SETTINGS | FLAT TRIM

Figure 2.9: Parrot Free Flight App Calibration Button

2.3 Software

Through this Section, the software used to develop the system algorithms and flight control and telemetry systems will be detailed.

2.3.1 Matlab

Matlab or matrix laboratory is a numerical computing environment. This software allows to easily manipulate matrices, plotting functions or data, implement algorithms or create graphical user interfaces (GUI). Matlab was originally designed for numerical computing, but thanks to additional toolboxes it can be used for feedback control and image processing, the main topics of this research project.

For the present work, Matlab was used for initial development, testing and optimization of the vision-based algorithm for UAV mimicking, as well as testing the designs on videos recorded during hardware experiments. However, when attempting to use Matlab's code generation feature to port the algorithms onto the ground computer for testing, the system refused to proceed due to a software digital rights management (DRM) lock built into the Computer Vision Toolbox. This motivated moving further development to open-source software tools.

2.3.2 Android

Android is an operating system (OS) developed by Google, based on the Linux kernel and designed for touchscreen mobile devices or smartphones.

Android's source code is released by Google under open source licenses, which makes the OS and any App customizable. Its application programming interface or API is well documented and easy to use. An API is a set of subroutine definitions, protocols and tools to build software, in this case, for Android. Google's primary integrated development environment (IDE) for native Android applications is Android Studio, based on JetBrains' IntelliJ IDEA software, and is available for Windows, Linux or Mac OS X under Apache License 2.0. Its IDE relies on Oracle's Java development kit or JDK.

The Android API most interesting feature is to be able to access the different threads of the microprocessor directly and customize the use. This way any App can use several threads to maximize its performance.

2.3.3 DroneKit

DroneKit is a native C library, designed by the creators of Ardupilot and developed by 3DR, to allow remote devices to communicate with Ardupilot's software running onboard UAVs. The DroneKit library has been ported to Android, which allows developers to create new applications quickly and easily. DroneKit-Android provides interfaces for Android applications to control Ardupilot based vehicles using MAVLink.

MAVLink or Micro Air Vehicle Link is a protocol for communicating with small unmanned vehicles. It is designed as a header-only message marshaling library. MAVLink was first released in early 2009 by Lorenz Meier under LGPL license. Nearly all commercial Ardupilot based UAVs use MAVLink as the primary communication protocol.

A MAVLink message contains all the telemetry needed to check the state of the UAV while flying, where conventional software debugging is not feasible. A sample received message is shown below.

```
<message id = "24" name="GPS_RAW_INT">
        <description>The global position, as returned by
           the Global Positioning System (GPS). This is
           NOT the global position estimate of the
           system, but rather a RAW sensor value. See
           message GLOBAL_POSITION for the global
           position estimate. Coordinate frame is
           right-handed, Z-axis up (GPS
           frame).</description>
        <field type="uint64_t"
           name="time_usec">Timestamp (microseconds
           since UNIX epoch or microseconds since system
           boot)</field>
        <field type="uint8_t" name="fix_type">0-1: no
           fix, 2: 2D fix, 3: 3D fix. Some applications
           will not use the value of this field unless
           it is at least two, so always correctly fill
           in the fix.</field>
        <field type="int32_t" name="lat">Latitude
           (WGS84), in degrees * 1E7</field>
        <field type="int32_t" name="lon">Longitude
           (WGS84), in degrees * 1E7</field>
        <field type="int32_t" name="alt">Altitude
           (WGS84), in meters * 1000 (positive for
           up)</field>
        <field type="uint16_t" name="eph">GPS HDOP
           horizontal dilution of position in cm
           (m*100). If unknown,set to: UINT16_MAX</field>
        <field type="uint16_t" name="epv">GPS VDOP
           vertical dilution of position in cm (m*100).
           If unknown,set to: UINT16_MAX</field>
        <field type="uint16_t" name="vel">GPS ground
           speed (m/s * 100). If unknown,set to:
           UINT16_MAX</field>
        <field type="uint16_t" name="cog">Course over
           ground (NOT heading, but direction of
           movement) in degrees * 100, 0.0..359.99
           degrees. If unknown,set to: UINT16_MAX</field>
        <field type="uint8_t"
           name="satellites_visible">Number of
           satellites visible. If unknown,set to
           255</field>
```

DroneKit-Android allows the user to access pre-coded actions in the control system such as take off, landing, return home (which makes the drone return to where it took off), return to me, rotate, pause in the current location, or move to a specified location.

2.3.4 OpenCV

OpenCV or Open Source Computer Vision is a library of programming functions mainly aimed at real-time computer vision applications, originally developed by Intel. The library is cross-platform and free for use under the open source BSD license. OpenCV runs in Android as native code with the Native Development Kit (NDK), but because this kit is still experimental, a lot of time was spent in order to make the overall system work.

The Cascade Classifier (see section 3.4) and the Kalman Filter (see section 3.3.3) are available within OpenCV 2.4.11.

Chapter 3

Vision-Based Detection-Tracking Algorithms

3.1 Overview

Computer vision offers the ability to detect and track moving objects, which is useful in applications such as surveillance, manufacturing or self-perception. The fundamental challenges that drive much of the research in this field are the enormous data bandwidths associated with high-resolution video, the need for real-time performance and a typically vague or ill-defined specification of the tracking problem itself.

This chapter addresses the development of a vision-based detection-and-tracking algorithm for a small UAV. It should autonomously perform detection and tracking of a single moving target UAV which may be actively moving and changing its pose. The available UAV hardware imposes hard constraints on camera resolution and computational power, which directly affects the choice of algorithm to be implemented. Moreover, the UAV-mounted camera is likely to be moving, and is also subject to significant vibrations from the airframe, even when a 3-axis camera stabilization gimbal is used such as on the 3DR Solo drone. Among other things, this implies that background removal will be quite challenging.

In order to select an algorithm for detection and tracking, consider the following. The cameras onboard both UAVs are capable of capturing ARGB-8888 1280x720 resolution video at 30 frames per second, and the 3DR Solo camera supports even higher numbers. However a single 1280x720 video frame in ARGB-8888 takes 3.6864 MB of memory, meaning the video frames will need to be downsized, down-sampled and pre-processed in other ways (for instance employing a grayscale version of the image reduces memory usage to 921.6 kB) before being used in the vision algorithm. Given that computational time is fixed, a balance needs to be made between the bandwidth of the input video and the complexity of the vision algorithm processing this data.

3.2 Color-Based Algorithms

The simplest type of vision-based detection-tracking approach that can be implemented to process the video stream is an algorithm based exclusively on color information. Color-based algorithms have been successfully used for mobile detection and tracking purposes over many decades [15].

A simple tracking method which employs color information is based on tracking regions of normalized color which are similar from frame to frame and comparing them against a template representation of the UAV, taken to be rectangular for convenience. Specifically, the region to be tracked is characterized by a color vector (3.1) which represents the averaged color of pixels within the region R_i. The size of this region cannot be overly large in order to keep the range of detection realistic.

$$(\alpha_i, r_i, g_i, b_i) = \frac{\sum_{(x,y)\in R_i}(\alpha(x,y), r(x,y), g(x,y), b(x,y))}{|R_i|} \tag{3.1}$$

A sample vector $(\hat{\alpha}, \hat{r}, \hat{g}, \hat{b})$ is considered as the ideal color vector for the object being detected and tracked, and it can be computed a priori. As only one UAV is expected to be in the frame, the best fit between this sample vector and the region R_i is then considered to be the location of the desired target. The fitting between the sample vector and the color vectors obtained from the frame is done by employing a goodness of fit test [15].

In order to make the color-based algorithm more robust, an Extended Kalman Filter can be added on. This allows keeping track of the target even under high occlusion conditions [28].

Although detection and tracking of mobile objects can be accomplished through color-based algorithms, the problem is that the algorithm fails in environments where the tracked object and the background have similar normalized color vectors, for instance when a dark background (e.g. trees) is behind the black-colored UAV. Since this situation is often encountered in our flight testing conditions, color-based algorithms were found to be insufficient for our purposes.

3.3 Histogram of Oriented Gradients (HOG)-Based Algorithm

3.3.1 Blob Analysis and HOG Detection

In computer vision, blob detection methods are aimed at detecting parts of a digital image which differ in properties, such as brightness or color, from their neighboring areas. A blob is a relatively small region of an image in which these properties are approximately constant; thus all the points inside a blob can be considered to be similar to each other in some sense. The minimum size of blob regions needs to be specified at the initialization of the algorithm; in our case, we will use 20x20 pixel blobs.

The detected blobs are invariant to translations, rotations and uniform rescaling of the image. Input images to a computer vision system are, however, subject to perspective distortions from the optical camera used to acquire them. To obtain blob descriptors which are more robust to such perspective transformations, a natural approach is to devise a blob detector which is invariant to affine transformations. In this way, we can define affine-adapted versions of blob detectors such as the Laplacian of Gaussian (LoG) operator, the difference of Gaussians (DoG), the determinant of the Hessian, and the Hessian-Laplace operator [32].

Blobs may signal the existence of objects or their parts in the image, which is useful for object recognition and/or tracking. Because the UAV target has a uniform color, blob analysis can be performed on grayscale images in order to reduce computational power, while retaining the ability to detect possible UAVs in the image.

The results of the blob analysis will be a function of the brightness of edges, as determined by the LoG operator. In this way, the algorithm is able to identify regions of the digital image which exhibit marked differences between the blob's brightness and its surroundings. Given that the target UAV is known to be black-colored in our case, LoG results can be filtered to retain only the blobs with a positive gradient edge. However, as for the color-based algorithms, dark-colored blobs defined by their brightness are not specific to the UAV, for instance shadows in the image will also be identified as a positive result by this approach. This makes it impossible, in most cases, to use blob detection on its own to track target UAVs.

Although blob detection is not able to fully determine and track the UAV's position, the results identify regions of interest (ROI) in the image which can be analyzed further. We thus look for another algorithm which can be run on these identified regions.

The histogram of oriented gradients (HOG) is a feature descriptor used in computer vision and image processing. This technique counts the occurrences of gradient orientation in regions of an image, more specifically in the regions identified by the blob analysis. The HOG algorithm is a detection method based on evaluating normalized local histograms of image gradient orientations in a dense grid. The basic idea is that local object appearance and shape is well characterized by the distribution of local intensity gradients or edge directions, even without precise knowledge of the corresponding gradient or edge positions [37, 45, 7].

As the HOG algorithm results are not affected by scaling or rotation, we can create a template for each UAV which is obtained by running a clear digital image of a given UAV through the HOG algorithm (see Figures 3.1 and 3.2).

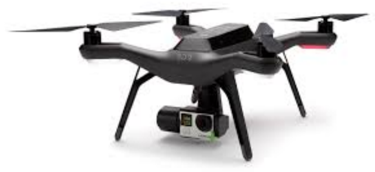
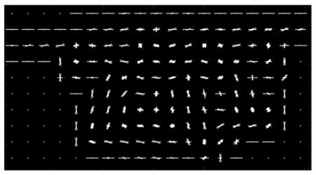



Figure 3.1: HOG Algorithm Template for 3DR Solo

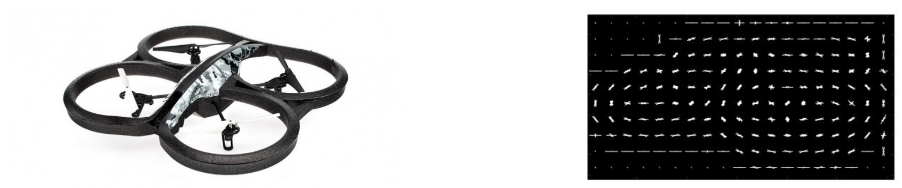


Figure 3.2: HOG Algorithm Template for Parrot AR.Drone 2.0

3.3.2 Features from Accelerated Segment Test (FAST) Algorithm

Although the UAV templates are matrices with relatively small dimensions, this approach is not efficient from a computational point of view since much of their data is either redundant or of no importance. It would be preferable to define each UAV by a cloud of points instead of a matrix. The HOG algorithm can be used in conjunction with the Features from Accelerated Segment Test (FAST) algorithm to find feature points in the UAV. The specified feature points are the characteristic corners of each UAV as shown in Figures 3.3 and 3.4.

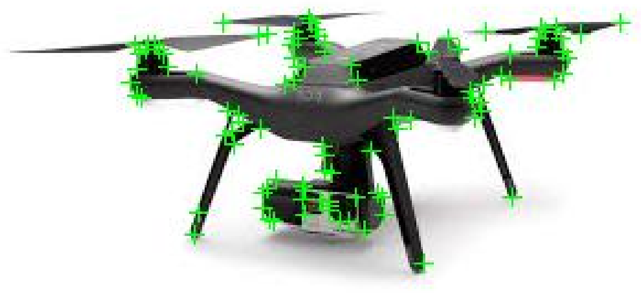


Figure 3.3: HOG Algorithm Template with FAST for 3DR Solo

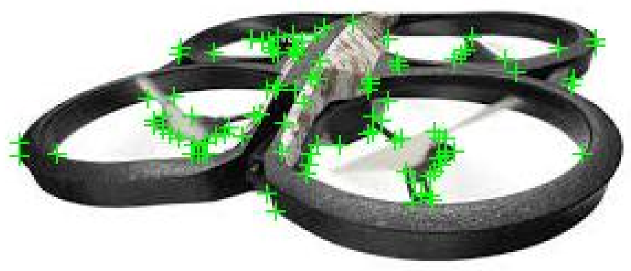


Figure 3.4: HOG Algorithm Template with FAST for Parrot AR.Drone 2.0

The template is loaded in the initialization process of the algorithm and serves as a comparison array for each ROI obtained from blob analysis. This process allows to determine whether a blob corresponds to the target UAV or if it should be considered as background noise. While not done here, this approach would work for tracking multiple UAVs as well.

The overall process is summarized in Figure 3.5. Both blob analysis and the HOG algorithm are available as software modules in OpenCV and Matlab, which can be used for implementation and simulation purposes, respectively.

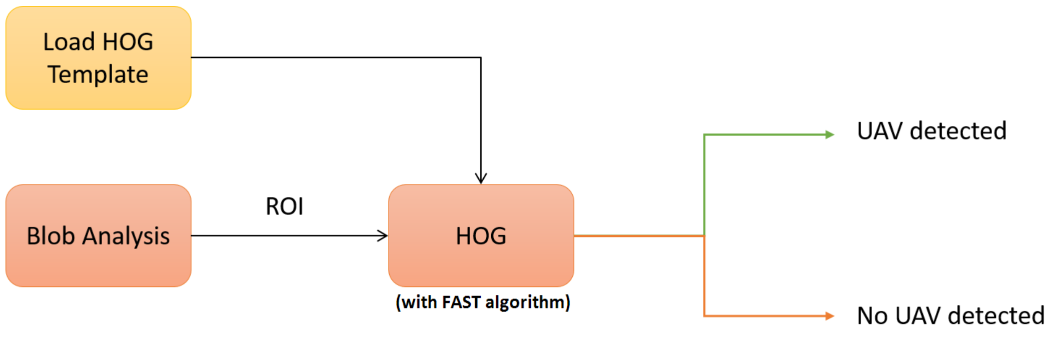


Figure 3.5: Blob Analysis with HOG-FAST Detection Process Diagram

While the algorithms presented in this Section provide a computationally efficiently way to detect UAVs in a single video frame, the detections are carried out independently of each other. In order to keep track of the UAV's movements from frame to frame, we need a way to associate a UAV detected in frame n at pixel coordinates $[x_n^n, y_p^n]$ to the UAV detected in frame $n+1$ at pixel coordinates $[x_p^{n+1}, y_n^{n+1}]$, using the assumption that they are the same UAV.

3.3.3 Hungarian Assignment Algorithm and Kalman Filtering

The Hungarian Assignment Algorithm is used for tracking multiple objects by associating detections to tracks [34]. A track is an individual detection followed over time, here a UAV in a video. This algorithm allows to algebraically determine whether a positive detection corresponds to an existing track or if it is a new track. It also determines if there are missing tracks. The assignment of tracks to detections is based on a cost function which the algorithm tries to minimize. In case of multiple detections, the cost function is a cost matrix in which each detection is associated to a cost for each existing track as well as to the creation of a new track. The lower the cost, the more likely that a detection gets assigned to a track.

The cost function is proportional to the distance between the previous track location and the detection location (in pixels). The cost of creating a new track is always higher than the cost of assigning the detection to an existent track, and in fact the algorithm tries to minimize the number of tracks which helps to keep the memory usage down. Moreover, in order to reduce the required computational power, we assign a non-assignment cost as the upper limit of the cost for a case where a detection will not be assigned to a track. The algorithm also deletes tracks that have been missing during a certain amount of sample periods.

The Hungarian Assignment Algorithm has its own limits: being a purely algebraic algorithm, it cannot discern whether a given detected UAV location makes sense physically, e.g. if the UAV was moving towards the east, the next UAV detection cannot be located in the west. This physical sense is provided by employing a Kalman Filter.

The Kalman Filter is an efficient and real-time computational solution for tracking a dynamic target under noisy measurements, which is done by minimizing the covariance of the error of the estimate. It is one of the most widely used methods for tracking and estimation applications due to its simplicity, optimality and robustness [25]. In computer vision, Kalman filtering is used to track specific pixel regions through uncertain motions and noisy measurements. More about Kalman filtering can be found in Section 4.5. The Kalman filter itself solves the problem of optimally estimating the state of the discrete-time linear system 3.2 subject to process and measurement noise:

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ z_k &= Hx_k + v_k \end{aligned} \tag{3.2}$$

where x is the state vector, u is the process input, z is the measurement vector, A is the state transition matrix, B is the input matrix, H is the measurement matrix, and v and w are the measurement and process noises, respectively.

Given a specific video frame, let's assume that the HOG-FAST algorithm (see Sections 3.3.1 and 3.3.2) returns at least one positive UAV detection. The resulting array consists of a set of parameters describing each detected blob, such as its size and centroid, which are used to define the dimensions of a bounding box. The parameters obtained completely characterize each blob, and thus any of them can be used with a Kalman filter, which in the present case is applied to the location of a blob's centroid.

A blob centroid is defined by a pair of pixel coordinates $[x_n^p, y_n^p]$ which act as the state vector x in the Kalman Filter. Since the blob does not possess an input, we take $B = 0$. The A and H matrices are obtained by assuming a constant velocity motion model of the blob as discussed below. The measurement noise v represents the sensor noise associated to the video camera, while the process noise w represents noise effects which affect the motion model such as vibrations of the UAV on which the camera is mounted on.

The motion model of the blob is constant velocity. The constant velocity model has the following state transition and measurement matrices for a Kalman filter:

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad ; \quad H = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

The Kalman Filter is often conceptualized as two distinct phases: “Prediction” and “Correction”. The “Prediction” phase uses the state estimate from the previous sample time to produce an estimate of the state at the current time. The predicted state $\hat{x}_{k|k-1}$ is also called the a priori state estimate. The “Correct” phase uses the a priori state estimate together with the current measurement to refine the state estimate. The improved estimate is then called the a posteriori state estimate, $\hat{x}_{k|k}$.

The a priori state estimate is always computed, irrespective of the output of the detection algorithm. In case this algorithm does not detect a UAV, the state estimation returns the UAV location for this frame based solely on the motion model of the filter. Conversely in case of a detection, the a posteriori state estimate improves the measurement accuracy of the detection algorithm by complementing it with the target UAV’s motion model used by the Kalman filter. Both estimates are computed as shown in Equations 3.3:

$$\begin{aligned} \hat{x}_{k|k-1} &= A\hat{x}_{k-1|k-1} \\ \hat{x}_{k|k} &= \hat{x}_{k|k-1} + P_{k|k-1} H^T S_k^{-1} \end{aligned} \tag{3.3}$$

where $P_{k|k-1}$ is the predicted (a priori) estimate covariance and S_k is the covariance of the residual error. Both are computed using the covariance matrix of the process noise $Q(w)$ and the covariance matrix of the observation noise $R(v)$ as follows:

$$\begin{aligned} P_{k|k-1} &= AP_{k-1|k-1}A^T + Q_k(w_k) \\ S_k &= HP_{k|k-1}H^T + R_k(v_k) \\ \text{where,} \quad P_{k-1|k-1} &= (I - P_{k-1|k-2}H^T S_{k-1}^{-1} H)(AP_{k-2|k-2}A^T + Q_{k-1}) \end{aligned} \tag{3.4}$$

Therefore, referring to the diagram in Figure 3.5, the overall system with blob analysis, HOG-FAST detection algorithm with Hungarian Assignment algorithm, and Kalman filtering tracking is summarized in Figure 3.6.

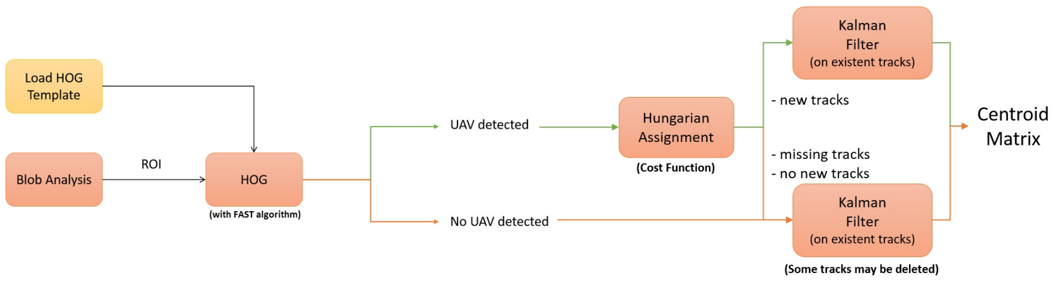


Figure 3.6: Flowchart of Algorithm to Estimate Centroid of a Target UAV from a Digital Image. From left to right: Blob Analysis with HOG-FAST, Hungarian Assignment Algorithm with Kalman Filter.

In the next section, we will implement the algorithms discussed in Section 3.3 and discuss their performance.

3.3.4 Algorithm Implementation

The components discussed in Sections 3.3.1, 3.3.2 and 3.3.3, namely blob analysis, the FAST algorithm, the HOG algorithm, the Hungarian Assignment algorithm and the Kalman Filter, are all available in Matlab in the Computer Vision System Toolbox. Thus, before implementation onboard UAV hardware, the system will be tested within Matlab.

First, testing was performed using a recorded video of a 3DR Solo drone taken with a moving camera (camera carried in hand). The UAV first flies away to the limit of the controller transmission range, then flies towards the camera such that the drone takes up a large part of the image viewpoint. In this way we can evaluate the algorithm's performance at the full range of distances. Running the algorithm on the recorded video yields the results shown in Figure 3.7.

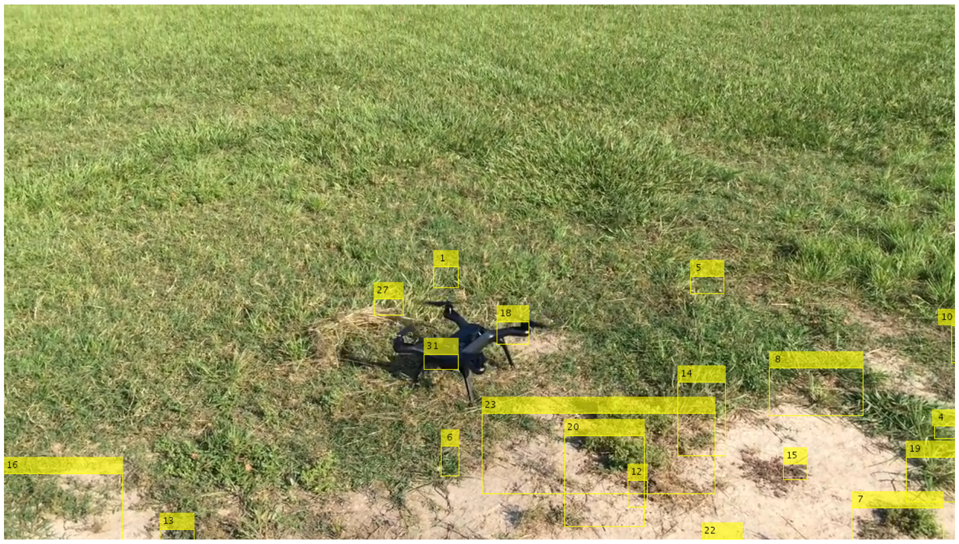


Figure 3.7: Blob Analysis Test with Moving Camera: HOG-FAST Detection Process plus Hungarian Assignment Algorithm with Kalman Filter Tracking

Within Figure 3.7, the yellow boxes labeled with numbers correspond to the blobs which were assigned tracks. The numbering is incremented at every detection, so in another words, the first detection assigned a track is labeled by the number '1', and the last detection assigned a track has the highest number. Some intermediate numbers may disappear due to being assigned to tracks which have been lost. We see 4 blobs around the UAV (labeled '1', '18', '27' and '31') which were unable to create a bounding box around the UAV and thus create a positive detection. This no-detection is due to the amount of other blobs which were detected at the same time. The analyzed blobs are noisy when the camera is in motion, and in this case the HOG algorithm cannot discern between blobs to decide which one corresponds to the UAV. Therefore, even if the Hungarian Algorithm is able to keep track of blobs for a short amount of time, the blobs appear and disappear very fast making the Kalman Filter useless, since it requires a stable detection over a sufficiently large number of frames to adapt its gain and filter out noise.

After further testing with other videos recorded of the 3DR Solo, we discovered that the noisy results from the blob analysis were due to the motion of the camera. The more stable the camera is, the less the number of detected blobs. However, even small vibrations in the camera due to factors such as wind gusts or walking over uneven surfaces would trigger the detection of noisy blobs within the video. We can conclude that blob analysis works well mainly when the camera is static.

At this point, we still haven't evaluated how the UAV detecting and tracking parts of the algorithm perform. In order to do this, we chose to retest the 3DR Solo drone in flight and to record the video using a static camera.

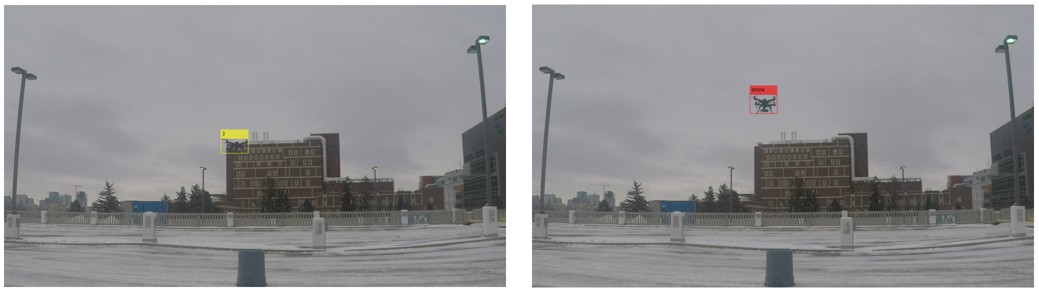


Figure 3.8: Blob Analysis Test with Static Camera: HOG-FAST Detection Process plus Hungarian Assignment Algorithm with Kalman Filter Tracking

As seen in the left image of Figure 3.8, the UAV is now the only detected blob in the entire frame which gets assigned a track. For a few seconds following take-off from the gray platform, the background of the UAV is dark (a building) and therefore the HOG algorithm is unable to determine whether or not the blob associated to the track corresponds to a UAV. However, the tracking of the blob is correct despite the noisy background. In the right image of Figure 3.8, the UAV is flying above the building, with a cloudy sky as background. As soon as this happens, the detected track gets labeled as "drone", indicating that the HOG algorithm is able to recognize the blob as a 3DR Solo drone. This behavior lasts for the remainder of the video. The tracking performance will be discussed in Section 4.4.3.

Despite the good performance of the overall algorithm, this solution is not appropriate for our application due to the requirement of a static camera. Because the use of the tracking process gave good results, we will look for another algorithm which does not employ blob analysis and so can work with cameras in motion.

3.4 Cascade Classifier

In their groundbreaking papers, Viola and Jones [61, 62] described a face detection algorithm which was able to process images quite rapidly (up to 15 fps on conventional computers circa 2000) while achieving high accuracy. The paper changed the way images where processed and how features were used by detection algorithms. The first innovation was "Integral Imaging" where the value of the image at point (x, y) is replaced by the sum of all the pixels above and to the left. This eased the way in which features are created (see Figure 3.9). The second innovation was to build an efficient classifier of features with a variant of the AdaBoost learning algorithm [17] to select, from all the available features, the most critical ones. The third one was a method to combine classifiers which allowed the background regions of each image to be discarded quickly and thus focus computational time on regions of the face within a frame.

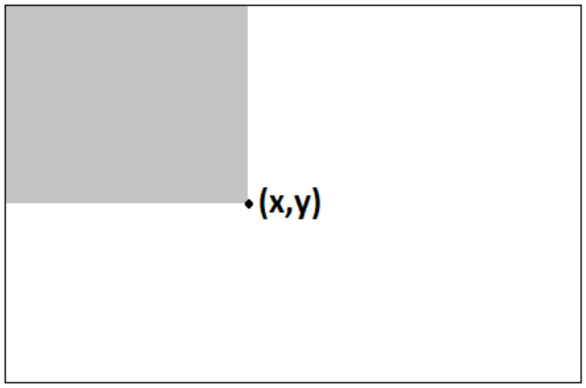


Figure 3.9: Integral Imaging Graphical Description

The resulting algorithm is a learning algorithm which is divided into two very different steps. The first is to train the algorithm using positive and negative images in order to eliminate weak classifiers and features from the object to track. The training is computationally quite demanding, on the time scale from hours to even weeks. However, the resulting selection of strong classifiers and their thresholds are summarized into a single XML file which can then be used by other devices. Thus, the training step can be performed offline on a powerful computer and the results exported to e.g. an Android device which will run the second step, the classification algorithm, in real time. The XML file can easily be employed with various languages such as C++, Python or Java, to name a few.

The training step can be performed either through an OpenCV module in Python or C++, or within Matlab running on Windows/Mac OS/Linux. Both options create an XML file which is cross-compatible with the other system. Due to ease of implementation, the training was performed in Matlab. The real-time classification was implemented in Android.

The Cascade Classifier algorithm was first designed, analyzed and optimized for facial detection [63, 13]. However, we wonder if the trainer can be used to detect any kind of object by the selection of positive and negative images given to it. Thus, in the next Sections, we will tailor the cascade trainer for UAV detection, and analyze the impact of each user parameter (true positive rate, feature type and false alarm rate) on detection accuracy as well as computational time in order to optimize the performance of the cascade classifier algorithm.

3.4.1 Cascade Classifier Trainer Overview

The training of the Cascade Classifier for UAV detection uses a set of N positive images of each UAV, either flying or on the ground and in different backgrounds, as well as a set of M negative images with outdoor or indoor backgrounds, not specific to the test flight zone, and including other flying objects which could look like a UAV, e.g. birds.

Within Matlab, the set of positive images can be created using the “Training Image Labeler” App. This allows to pick the ROI within images in order to avoid having to manually crop each input image. The App transforms the set of positive images into a table which is ready to use. The set of negative images is read from a specified folder as an *imageDatastore*.

The Matlab Cascade Classifier trainer (*trainCascadeObjectDetector* command) produces a single XML file but requires several parameters to be specified in addition to the positive and negative sets of images. The parameters available for the trainer are:

- Object Training Size: this is a two-entry vector which specifies the height and width, in pixels, of the object to be detected. Before the training starts, the trainer resizes the positive and negative samples to the specified object size. By default, the trainer determines this size automatically based on the median width-to-height ratio of the positive instances.

- Negative Samples Factor: this defines the number of negative images used at each training stage relative to the number of positive samples used. By default, the trainer uses double the number of negative images from the positive images. The number of negative images used at each stage is thus

$$M = NegativeSamplesFactor \ * \ N$$

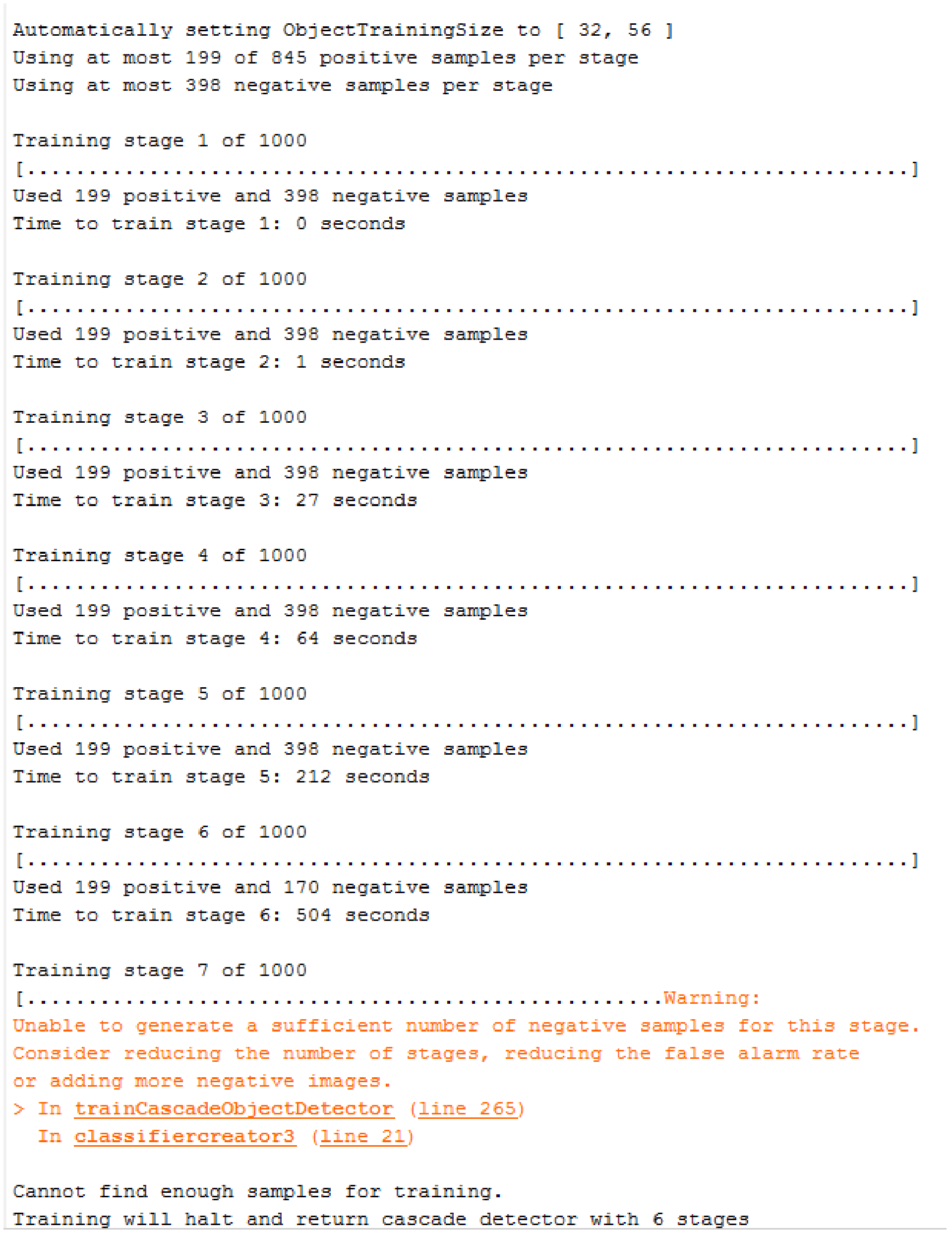
- Number of Cascade Stages: this specifies the number of stages for the trainer. Increasing the number of stages results in a more accurate detector but also increases training time as well as the size of the cascade classifier XML file. Higher numbers of stages may also require more training images, since at each stage a certain part of positive and negative samples can be eliminated. The number of eliminations depends on the values of *FalseAlarmRate* and *True-PositiveRate*. By default, the trainer uses all the images available, maximizing the number of stages.

- Acceptable False Alarm Rate: the maximum fraction of negative training samples which can be incorrectly classified as positive. The false alarm rate is a number between 0 and 1. The overall false alarm rate can be calculated as the specified false alarm rate to the power of the number of stages. Lower values for acceptable false alarm rate increase the complexity of each stage. By default, this value is set to 0.5.

- Minimum True Positive Rate: the minimum fraction of correctly classified positive training samples. A true positive occurs when a positive sample is correctly classified. The true positive rate is a number between 0 and 1. The overall true positive rate can be calculated as the specified true positive rate to the power of the number of stages. By default, this is set to 0.995.

- Feature Type: specifies the type of feature used by the cascade classifier. The three options available are: Haar-like features (Haar), local binary patterns (LBP) and histogram of oriented gradients (HOG).

An example of the output of the code used to train the Cascade Classifier detection algorithm code can be seen in Figure 3.10.



```
Automatically setting ObjectTrainingSize to [ 32, 56 ]
Using at most 199 of 845 positive samples per stage
Using at most 398 negative samples per stage

Training stage 1 of 1000
[........................................................................]
Used 199 positive and 398 negative samples
Time to train stage 1: 0 seconds

Training stage 2 of 1000
[........................................................................]
Used 199 positive and 398 negative samples
Time to train stage 2: 1 seconds

Training stage 3 of 1000
[........................................................................]
Used 199 positive and 398 negative samples
Time to train stage 3: 27 seconds

Training stage 4 of 1000
[........................................................................]
Used 199 positive and 398 negative samples
Time to train stage 4: 64 seconds

Training stage 5 of 1000
[........................................................................]
Used 199 positive and 398 negative samples
Time to train stage 5: 212 seconds

Training stage 6 of 1000
[........................................................................]
Used 199 positive and 170 negative samples
Time to train stage 6: 504 seconds

Training stage 7 of 1000
[....................................................Warning:
Unable to generate a sufficient number of negative samples for this stage.
Consider reducing the number of stages, reducing the false alarm rate
or adding more negative images.
> In trainCascadeObjectDetector (line 265)
  In classifiercreator3 (line 21)

Cannot find enough samples for training.
Training will halt and return cascade detector with 6 stages
```

Figure 3.10: Matlab Cascade Classifier Trainer Output Example

Certain factors need to be considered when setting the parameter values. On one hand, a small set of training images will lead to a lower overall number of stages. It will need a lower false positive rate, thus achieving a less accurate detection, but which is also less computationally demanding. On the other hand, a large training set (in the thousands of images) increases the number of stages and can employ a higher false positive rate for each stage, leading to a more accurate detection algorithm but which is also more computationally demanding. As well, we can increase the true positive rate to reduce the chance of missing the target, but a high true positive rate can prevent meeting the specified false positive rate per stage, making the detector more likely to produce false detections. However, increasing the number of stages or decreasing the false alarm rate can reduce the number of false detections. Therefore, we need to balance getting a reasonably accurate algorithm which keeps the required computational power as small as possible.

3.4.2 Cascade Classifier Trainer Optimization

The optimization of the training parameters is focused on accuracy and computational time. Given the large amount of parameters available, some of them will be fixed in order to reduce the number of variables in the optimization problem: the object training size will be set automatically, the negative sample factor will be kept at its default (value of 2), and the number of cascade stages will be set automatically but we will limit it by choosing a maximum number of positive instances and negative images for training [43]. Specifically, we will employ 845 positive images and 3144 negative images.

The rest of the parameters, namely the false alarm rate, the true positive rate and the feature type, will be optimized. Table 3.1 defines the range or options for each of the parameters.

Table 3.1: Cascade Classifier Trainer Parameter Ranges

Parameter	**Range**	**Step**
True Positive Rate FoV	0.975-0.995	0.005
False Positive Rate FoV	0.1-0.9	0.1
Feature Type	Haar - LBP - HOG	N/A

Once all the cascade classifiers were created, they were tested on different short videos: first, a video of a 3DR Solo flying through a clear background (sky); second, a video of the 3DR Solo standing still on the ground; and finally, a video of the 3DR Solo flying in front of a noisy background (trees). The detection algorithm accuracy as well as the computational time were measured in all videos. Tables 3.2 to 3.4 show the measured computational time needed to run the detection algorithm. Tables 3.5 to 3.13 show the accuracy results for the different cascade classifiers created.

Table 3.2: Cascade Classifier Trainer Computational Time Estimation Results for HOG Features (in seconds)

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	72.51	72.9	72.3	71.8	71.8
0.2	72.3	72.4	72.3	72.0	71.9
0.3	72.38	72.5	72.4	72.0	71.9
0.4	72.43	72.5	72.4	72.0	71.8
0.5	72.37	72.5	72.4	72.0	71.9
0.6	72.45	72.5	72.4	72.0	71.9
0.7	72.45	72.4	72.3	72.1	72.0
0.8	72.39	72.5	72.3	72.0	71.9
0.9	72.39	72.5	72.3	72.0	71.9

Table 3.3: Cascade Classifier Trainer Computational Time Estimation Results for Haar Features (in seconds)

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	17.6	14.4	14.2	14.4	14.5
0.2	17.0	14.4	14.1	14.4	14.4
0.3	17.2	14.3	14.1	14.4	14.4
0.4	17.2	14.4	14.1	14.4	14.4
0.5	17.2	14.4	14.1	14.4	14.4
0.6	17.3	14.3	14.1	14.4	14.4
0.7	17.2	14.3	14.1	14.4	14.4
0.8	17.3	14.3	14.1	14.4	14.4
0.9	17.3	14.3	14.0	14.4	14.4

Table 3.4: Cascade Classifier Trainer Computational Time Estimation Results for LBP Features (in seconds)

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	19.36	16.9	16.6	15.5	15.9
0.2	19.29	17.4	16.7	15.6	15.8
0.3	19.31	17.3	16.7	15.7	15.8
0.4	19.25	17.2	16.6	15.6	15.7
0.5	19.26	17.3	16.7	15.5	15.8
0.6	19.25	17.3	16.9	15.6	15.8
0.7	19.28	17.2	16.7	15.6	15.8
0.8	19.22	17.2	16.8	15.5	15.9
0.9	19.22	17.3	16.8	15.6	15.9

Table 3.5: Cascade Classifier Trainer Efficiency Results for HOG Features with Clear Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	88.7	24.0	21.3	5.3	1.7
0.2	95.3	4.0	24.0	2.7	2.0
0.3	91.3	31.3	10.7	0.7	2.0
0.4	89.3	8.3	15.3	2.3	3.3
0.5	87.3	26.3	36.0	0.7	1.7
0.6	96.7	5.3	32.7	4.3	3.3
0.7	86.0	16.0	28.0	3.7	3.3
0.8	85.3	15.0	6.7	0.7	3.3
0.9	96.7	14.7	6.7	0.7	3.3

Table 3.6: Cascade Classifier Trainer Efficiency Results for Haar Features with Clear Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	92.7	92.0	95.3	71.3	99.3
0.2	100.0	94.7	100.0	78.7	99.3
0.3	100.0	56.3	91.3	90.0	99.3
0.4	100.0	94.7	100.0	90.0	99.3
0.5	100.0	88.3	99.3	90.0	99.3
0.6	100.0	93.3	99.3	70.7	99.3
0.7	100.0	70.0	99.3	70.7	99.3
0.8	100.0	67.3	99.3	70.7	99.3
0.9	100.0	67.3	99.3	70.7	99.3

Table 3.7: Cascade Classifier Trainer Efficiency Results for LBP Features with Clear Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	48.1	0.0	0.6	0.6	0.0
0.2	7.2	0.0	0.0	0.0	0.0
0.3	3.1	0.0	0.0	0.0	0.0
0.4	19.2	0.0	0.0	0.0	0.0
0.5	9.7	0.0	0.0	0.0	0.0
0.6	11.1	0.0	0.0	0.0	0.0
0.7	10.6	0.0	0.0	0.0	0.0
0.8	10.6	0.0	0.0	0.0	0.0
0.9	10.6	0.0	0.0	0.0	0.0

Table 3.8: Cascade Classifier Trainer Efficiency Results for HOG Features with Static UAV

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	24.2	0.0	0.0	5.3	1.7
0.2	6.7	0.0	0.0	2.7	2.0
0.3	5.8	0.0	0.0	0.7	2.0
0.4	11.7	0.0	0.0	2.3	3.3
0.5	5.8	0.0	0.0	0.7	1.7
0.6	0.8	0.0	0.0	4.3	3.3
0.7	3.3	0.0	0.8	3.7	3.3
0.8	0.0	0.0	0.0	0.7	3.3
0.9	10.0	0.0	0.0	0.7	3.3

Table 3.9: Cascade Classifier Trainer Efficiency Results for Haar Features with Static UAV

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	0.0	0.0	0.0
0.3	0.0	0.0	0.0	0.0	0.0
0.4	0.0	0.0	0.0	0.0	0.0
0.5	0.0	0.0	0.0	0.0	0.0
0.6	0.0	0.0	0.0	0.0	0.0
0.7	0.0	0.0	0.0	0.0	0.0
0.8	0.0	0.0	0.0	0.0	0.0
0.9	0.0	0.0	0.0	0.0	0.0

Table 3.10: Cascade Classifier Trainer Efficiency Results for LBP Features with Static UAV

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	0.0	0.0	0.0
0.3	0.0	0.0	0.0	0.0	0.0
0.4	0.0	0.0	0.0	0.0	0.0
0.5	0.0	0.0	0.0	0.0	0.0
0.6	0.0	0.0	0.0	0.0	0.0
0.7	0.0	0.0	0.0	0.0	0.0
0.8	0.0	0.0	0.0	0.0	0.0
0.9	0.0	0.0	0.0	0.0	0.0

Table 3.11: Cascade Classifier Trainer Efficiency Results for HOG Features with Noisy Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	0.0	0.0	0.0
0.3	0.0	0.0	0.0	0.0	0.0
0.4	0.0	0.0	0.0	0.0	0.0
0.5	0.0	0.0	0.0	0.0	0.0
0.6	0.0	0.0	0.0	0.0	0.0
0.7	0.0	0.0	0.0	0.0	0.0
0.8	0.0	0.0	0.0	0.0	0.0
0.9	0.0	0.0	0.0	0.0	0.0

Table 3.12: Cascade Classifier Trainer Efficiency Results for Haar Features with Noisy Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	0.0	0.0	0.0
0.3	0.0	0.0	0.0	0.0	0.0
0.4	0.0	0.0	0.0	0.0	0.0
0.5	0.0	0.0	0.0	0.0	0.0
0.6	0.0	0.0	0.0	0.0	0.0
0.7	0.0	0.0	0.0	0.0	0.0
0.8	0.0	0.0	0.0	0.0	0.0
0.9	0.0	0.0	0.0	0.0	0.0

Table 3.13: Cascade Classifier Trainer Efficiency Results for LBP Features with Noisy Background

False Alarm Rate	**True Positive Rate**				
	0.995	0.990	0.985	0.980	0.975
0.1	12.6	0.0	0.0	11.5	2.0
0.2	0.0	0.0	0.6	0.0	1.1
0.3	9.8	0.0	0.0	0.9	1.1
0.4	9.2	1.2	1.4	0.9	1.1
0.5	16.2	0.0	5.8	0.9	1.1
0.6	2.3	1.8	5.8	0.0	1.1
0.7	0.0	0.0	5.8	0.0	1.1
0.8	5.8	3.5	5.8	0.0	1.1
0.9	3.9	3.5	5.8	0.0	1.1

The results provide the following conclusions: the LBP features are not suited for this kind of detection problem. Between Haar and HOG, Haar offers better accuracy with clear backgrounds, but fails to detect the target while it's static or over a noisy backgrounds; however, HOG still offers good performance with high true positive rates and can also to some extent detect the UAV while it static. Further, the Haar cascade classifier is around 5 times faster than the HOG cascade classifier.

Based on these results, it is clear that the best result for both features, all things considered, occurs when the false alarm rate is at its minimum (0.1) and the true positive rate is at its maximum (0.995). Therefore, we will create and retest the cascade classifiers in the lowest ranges of the false alarm rate (between 0.001 and 0.1) and the highest ranges of the true positive rate (between 0.995 and 1). The results of the new tests can be found in Tables 3.14 to 3.17.

Table 3.14: Cascade Classifier Trainer Efficiency Results for HOG Features with Clear Background

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	98.0	98.0	97.7	97.0	92.0
0.002	78.7	62.3	91.0	97.0	92.3
0.003	84.0	70.7	92.7	100.0	95.0
0.004	85.7	77.3	91.3	98.7	95.3
0.005	86.3	77.3	82.0	98.7	95.3
0.006	79.0	76.7	80.3	95.3	93.3
0.007	81.7	73.7	82.3	89.3	92.7
0.008	82.0	73.7	76.7	92.7	89.3
0.009	84.7	76.7	81.7	85.3	93.0
0.01	79.3	73.3	76.3	87.7	89.0
0.02	66.0	78.3	74.0	100.0	90.0
0.04	72.7	78.3	87.7	81.0	95.3
0.06	86.7	69.7	53.7	96.0	98.7
0.08	75.0	97.3	97.7	100.0	96.7

Table 3.15: Cascade Classifier Trainer Efficiency Results for Haar Features with Clear Background

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	99.0	100.0	100.0	100.0	100.0
0.002	97.3	100.0	100.0	100.0	100.0
0.003	99.3	100.0	100.0	98.3	100.0
0.004	100.0	100.0	100.0	99.0	100.0
0.005	100.0	100.0	100.0	99.0	100.0
0.006	100.0	100.0	100.0	99.0	100.0
0.007	99.3	100.0	100.0	97.7	100.0
0.008	99.7	100.0	100.0	97.7	100.0
0.009	100.0	100.0	100.0	98.3	100.0
0.01	99.7	100.0	98.3	98.3	100.0
0.02	99.7	87.3	100.0	100.0	100.0
0.04	98.7	95.7	91.7	89.7	100.0
0.06	92.7	98.0	100.0	100.0	97.0
0.08	99.0	99.3	98.0	100.0	100.0

Table 3.16: Cascade Classifier Trainer Efficiency Results for HOG Features with Static UAV

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	100.0	70.8	90.8	60.0	0.0
0.002	100.0	17.5	97.5	58.3	0.0
0.003	100.0	17.5	98.3	70.0	4.2
0.004	100.0	99.2	61.7	81.7	5.0
0.005	100.0	79.2	90.0	80.0	1.7
0.006	100.0	84.2	90.0	85.0	13.3
0.007	100.0	78.3	81.7	25.0	15.8
0.008	100.0	80.8	23.3	27.5	34.2
0.009	100.0	100.0	25.0	80.8	38.3
0.01	100.0	98.3	23.3	79.2	33.3
0.02	100.0	92.5	100.0	87.5	38.3
0.04	100.0	94.2	90.8	15.0	1.7
0.06	100.0	72.5	27.5	90.0	2.5
0.08	100.0	100.0	100.0	12.5	6.7

Table 3.17: Cascade Classifier Trainer Efficiency Results for Haar Features with Static UAV

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	100.0	0.0	0.0	0.0	0.0
0.002	100.0	1.7	0.0	0.0	0.0
0.003	100.0	1.7	0.0	0.0	0.0
0.004	100.0	0.8	0.0	0.0	0.0
0.005	100.0	0.0	0.0	0.0	0.0
0.006	100.0	0.0	0.0	0.0	0.0
0.007	100.0	0.8	0.0	0.0	0.0
0.008	100.0	0.0	0.0	0.0	0.0
0.009	100.0	0.8	0.0	0.0	0.0
0.01	100.0	0.8	0.0	0.0	0.0
0.02	100.0	0.0	0.0	0.0	0.0
0.04	100.0	0.0	0.0	0.0	0.0
0.06	100.0	0.8	0.0	0.0	0.0
0.08	100.0	8.3	0.8	0.8	0.0

Table 3.18: Cascade Classifier Trainer Efficiency Results for HOG Features with Noisy Background

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	0.0	0.0	0.0	7.9	0.0
0.002	0.0	0.0	0.9	7.9	0.0
0.003	0.0	0.0	0.9	0.0	0.3
0.004	0.0	0.0	1.5	0.0	0.3
0.005	0.0	3.3	3.3	0.0	0.3
0.006	0.0	3.3	3.3	0.0	0.0
0.007	0.0	3.3	0.9	0.0	0.0
0.008	0.0	3.3	3.3	0.0	0.0
0.009	0.0	3.3	3.3	0.0	0.0
0.01	0.0	3.3	3.3	0.0	0.0
0.02	3.5	0.0	0.0	0.0	0.0
0.04	0.0	0.0	0.0	0.7	0.0
0.06	0.0	0.6	0.0	0.0	0.0
0.08	0.4	0.0	0.0	0.0	0.0

Table 3.19: Cascade Classifier Trainer Efficiency Results for Haar Features with Noisy Background

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	100.0	0.0	0.0	0.0	0.0
0.002	100.0	1.7	0.0	0.0	0.0
0.003	100.0	1.7	0.0	0.0	0.0
0.004	100.0	0.8	0.0	0.0	0.0
0.005	100.0	0.0	0.0	0.0	0.0
0.006	100.0	0.0	0.0	0.0	0.0
0.007	100.0	0.8	0.0	0.0	0.0
0.008	100.0	0.0	0.0	0.0	0.0
0.009	100.0	0.8	0.0	0.0	0.0
0.01	100.0	0.8	0.0	0.0	0.0
0.02	100.0	0.0	0.0	0.0	0.0
0.04	100.0	0.0	0.0	0.0	0.0
0.06	100.0	0.8	0.0	0.0	0.0
0.08	100.0	8.3	0.8	0.8	0.0

In order to find the best solution, we formulate an optimization problem to maximize a cost function. The cost function will be a sum of the efficiencies of the algorithms weighted by the background or conditions of the detection:

$$J = max \sum_{i=1}^{3} \alpha_i d_i \tag{3.5}$$

where d_i is the efficiency of the cascade classifier and α_i is the weight of the i^{th} type of detection problem. The weights $[\alpha_1,\alpha_2,\alpha_3]$ respectively correspond to a high contrast between background and detection target, i.e. drone flying in a clear blue sky, a static target detection, and a low contrast between the background and the detection target, i.e. drone flying in front of a dark background.

The weights were tuned by hand, taking into account the amount of time the detection algorithm runs in each type of conditions, as well as the importance of said conditions for good behavior of the algorithm. UAVs typically fly in open spaces and at high altitudes, meaning a high-contrast background, however it is also important that a UAV can be detected even if it is static on the ground. The weights should be tuned along these lines. For this reason, we take $\alpha_1 = 2/3$, $\alpha_2 = 1/4$ and $\alpha_3 = 1/12$. The efficiency cost function results are shown in Tables 3.20 and 3.21.

Table 3.20: Cascade Classifier Trainer Efficiency Cost Function Results for HOG Features

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	90.3	83.0	74.0	80.3	61.3
0.002	77.4	45.9	85.1	79.9	61.5
0.003	81.0	51.5	86.4	84.2	64.4
0.004	82.1	76.3	76.4	86.2	64.8
0.005	82.6	71.6	77.4	85.8	64.0
0.006	77.7	72.4	76.3	84.8	65.6
0.007	79.4	69.0	75.4	65.8	65.7
0.008	79.7	69.6	57.2	68.7	68.1
0.009	81.4	76.4	61.0	77.1	71.6
0.01	77.9	73.7	57.0	78.2	67.7
0.02	69.3	75.3	74.3	88.5	80.8
0.04	73.4	75.8	81.2	57.8	64.0
0.06	82.8	64.6	42.7	86.5	66.4
0.08	75.0	90.1	90.1	69.8	66.1

Table 3.21: Cascade Classifier Trainer Efficiency Cost Function Results for Haar Features

False Alarm Rate	**True Positive Rate**				
	0.999	0.998	0.997	0.996	0.995
0.001	91.3	67.3	66.8	67.3	67.7
0.002	90.5	67.1	68.0	67.3	67.7
0.003	92.1	68.0	68.0	66.5	68.0
0.004	92.5	67.8	68.0	66.9	68.0
0.005	92.5	67.7	67.4	66.9	68.0
0.006	92.2	67.7	67.4	66.9	67.4
0.007	91.7	67.0	67.4	65.9	67.4
0.008	91.6	66.8	67.4	65.9	67.6
0.009	91.8	67.0	67.4	66.3	67.6
0.01	91.6	67.0	66.3	66.3	67.9
0.02	91.5	58.2	66.7	66.7	66.7
0.04	90.8	63.9	61.2	59.8	66.7
0.06	86.9	65.6	66.7	66.7	64.7
0.08	91.2	68.6	65.6	66.9	66.7

Based on the results from our cascade classifier optimization, we will further test the cascade classifiers highlighted in green in Tables 3.20 and 3.21 in Section 4.4.3. Note that in Table 3.21, the column corresponding to the highest true positive rate is highlighted in gray since the efficiency results are unexpectedly high and so must be treated with caution. Some of these will be analyzed further.

3.4.3 Cascade Classifier Implementation

The Cascade Classifier algorithm is implemented in an Android environment, using the OpenCV library, in three steps:

- Load OpenCV library: the OpenCV library should be loaded when the App initializes, making all the internal methods available immediately. This way, we avoid the situation where the App calls for a library method before it has been successfully loaded, which would lead to a software crash. To load external libraries while an Android App initializes, the loading method is called as a static variable. The OpenCV method called is OpenCVLoader.*initDebug*() which opens a debugging thread for the OpenCV methods while the App is executing (see [1]).

- Load Cascade Classifier XML File: as soon as the App is fully operational, we need to load the XML file created during training (see Section 3.4.2). For this, the XML file should be kept in the device memory and accessible to the App. The method to read the XML file on the device and write it into the App is:

```
public void load_cascade(){
try {
InputStream is =
   getResources().openRawResource(R.raw.solocascade);
File cascadeDir =
   getDir("Cascade",Context.MODE_PRIVATE);
File mCascadeFile = new
   File(cascadeDir,"solocascade.xml");
FileOutputStream os = new
   FileOutputStream(mCascadeFile);

byte[] buffer = new byte[4096];
int bytesRead;
while ((bytesRead = is.read(buffer)) != -1) {
os.write(buffer,0,bytesRead);
}
is.close();
os.close();

droneClassifier = new
   CascadeClassifier(mCascadeFile.getAbsolutePath());
droneClassifier.load(mCascadeFile.getAbsolutePath());
if(droneClassifier.empty()) {
showToast("SHORT","(!)Error loading XML File!");
} else {
showToast("SHORT","Success loading cascade
   classifier algorithm.");
}
} catch (IOException e) {
```

```
e.printStackTrace();
}
}
```

- Use the “CascadeClassifier” class: the use of the loaded Cascade Classifier XML file is done through the “CascadeClassifier” class. This class allows access to the detection algorithm through the method CascadeClassifier.*detectMultiScale* (see [1]).

Chapter 4

UAVs Logic And Control System

4.1 Overview

As stated in Sections 2.1.1 and 2.1.2, the two UAVs (Solo and AR.Drone 2.0) run onboard control systems which stabilize attitude and position. These controllers are already well tuned and provide robust performance in flight conditions. Consequently, we will continue using the stock onboard control system, and use it to track reference trajectories generated by our vision-based following algorithm (see Chapter 3), which runs as an Android App on a ground computer. This approach lets us take advantage of the UAV's well-tuned control performance, and provides the important safety feature of reverting to default flight operation in case of a software bug or loss of contact with the ground computer.

The existing UAV control system will be adapted to our vision-based algorithm as shown in Figure 4.1.

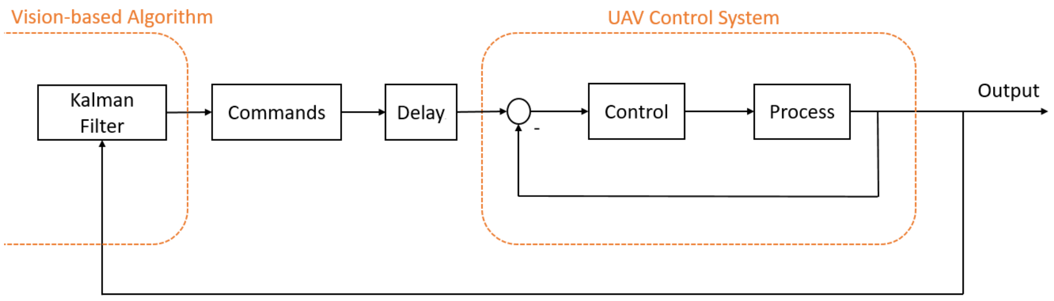


Figure 4.1: Control System Modification for Mimicking UAV

The Kalman Filter inside the vision-based algorithm uses the measured centroid of the UAV in its calculations. As discussed in the previous Chapter, the result of the algorithm is a vector $[\delta x,\delta y,\delta z]_N$, representing the desired motion of the UAV in the navigation reference frame. This vector is translated into a series of commands transmitted over Wi-Fi to the UAV. The delay block represents the inherent transmission and processing delays within the system.

4.2 Logic and Control Strategy for 3DR Solo

4.2.1 Overview

The control strategy for the 3DR Solo is to make the flight autonomous, from the take-off to the UAV following phase, with only a minimum number of user actions on the graphical interface of the Android App. In other words, no stick inputs on the manual flight controller are required for the Detection and Following algorithms to operate.

The overall system design is illustrated as a state machine in Figure 4.2. Each state involves issuing one or more commands to the Solo, then letting them complete while monitoring the system's output. Each state block creates and deletes software threads as needed. The control strategy for the 3DR Solo is divided into three main states: "Setup", "Scan" and "Scan and Follow".

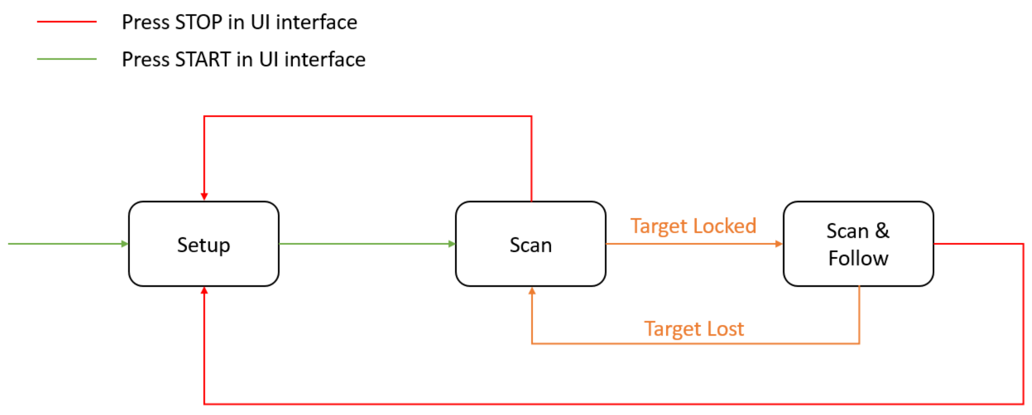


Figure 4.2: Simplified State Machine for 3DR Solo Control Strategy

The "Setup" state handles all the pre-flight steps, including enabling the camera and the actively stabilized camera gimbal, plus the initial flight phase of arming the motors followed by autonomous take-off to a pre-specified altitude. This state is triggered by the **Start Setup** button in the user interface of the App. If any command in the setup sequence fails to execute, the system returns to the previous state, which waits for the user to press **Start Setup**. Once the Setup state completes successfully, the **Start** button becomes available.

The "Scan" state follows the "Setup" state and is triggered by the user pressing the **Start** button in the user interface of the App. Within this state, the vision-based algorithm from Section 3.4 will run persistently in its own thread. This state puts the UAV in hover, runs the vision algorithm for 10 seconds, then commands a 90° clockwise turn and loops back to running the vision algorithm. At any time, if a target UAV is detected by the vision algorithm over at least 40 consecutive frames, the system proceeds to the following state.

The final state, "Scan and Follow", is triggered by finding a target UAV. This state executes a pursuit control while running the vision-based algorithm, each in its own thread. This state runs until the target is lost, at which point the system returns to the previous state, "Scan".

Within any state, pressing the **Stop** button in the user interface App reverts the system to the end of the "Setup" state, i.e. hovering in place and waiting for the user to press the **Start** button.

In the following subsections, we will describe the user interface and its components. Then, we will detail the individual states and transition conditions, either logical or mathematical, of the 3DR Solo control strategy.

4.2.2 User Interface

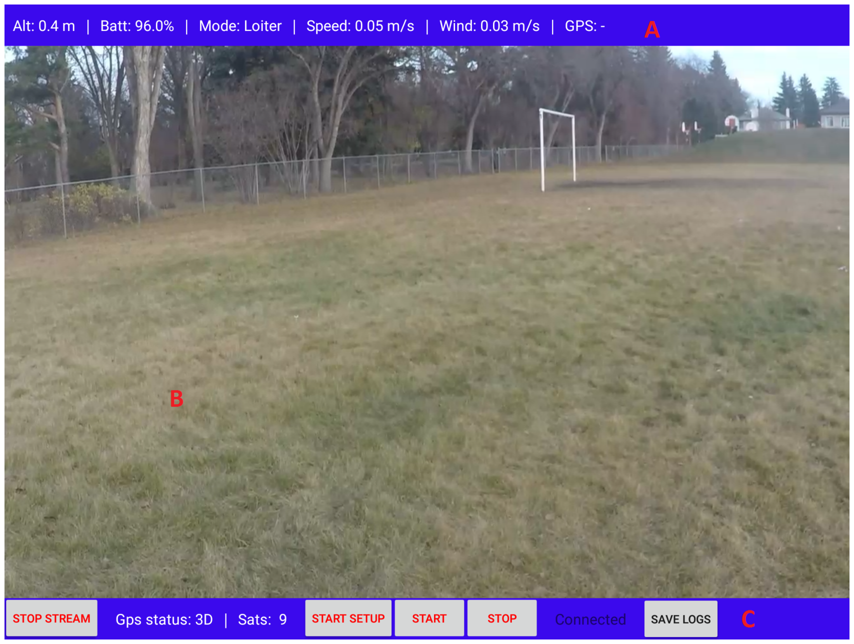

Alt: 0.4 m | Batt: 96.0% | Mode: Loiter | Speed: 0.05 m/s | Wind: 0.03 m/s | GPS: -
A
B
STOP STREAM
Gps status: 3D | Sats: 9
START SETUP
START
STOP
Connected
SAVE LOGS
C

Figure 4.3: User Interface for the 3DR Solo Android App

The App's user interface (UI) is divided into three zones denoted as A, B and C in Figure 4.3. The first zone (A) displays real-time information about the UAV status such as altitude, battery charge (in %), the flight mode, ground speed, wind speed and GPS location. Zone B is a surface texture where the video streamed from the UAV is displayed. Zone C contains relevant information about the status of the connection with the UAV, the GPS lock-on status, the number of detected GPS satellites, as well as a series of user buttons:

- **Start Stream**: connects the UAV's onboard GoPro Hero 4 camera and starts streaming real time video from it. The video is also recorded onto the camera's SD card. Once clicked this changes into the **Stop Stream** button.

- **Stop Stream**: ends connection with the onboard camera and freezes the video display (zone B in Figure 4.3) on the last frame. The video recording onto the SD card is stopped. Once clicked this changes back into the **Start Stream** button.

- **Start Setup**: triggers the "Setup" state, described in Section 4.2.3.

- **Start**: triggers the "Scan" state, described in Section 4.2.4.

- **Stop**: when the UAV is in flight, allows the operator to regain manual control of the UAV. This is a safety measure for testing and can be used to terminate the running algorithm at any time.
- **Save Logs**: saves all the logs of the App into a .txt file for analysis and debugging.

4.2.3 Setup State

As mentioned in Section 4.1, the "Setup" state prepares the UAV to run the vision-based following algorithm, starting with pre-flight preparations and ending with hovering in the air waiting for the user to launch the scanning algorithm. This state employs three separate threads: the main thread, the UI thread and the UAV control thread. The main thread and the UI thread are created when the App launches, while the UAV control thread is created only when needed and destroyed as soon as possible in order to keep the number of threads to a minimum and economize memory and computational load.

The entire process is automatic and does not require user intervention. Figure 4.4 illustrates the details of the "Setup" state.

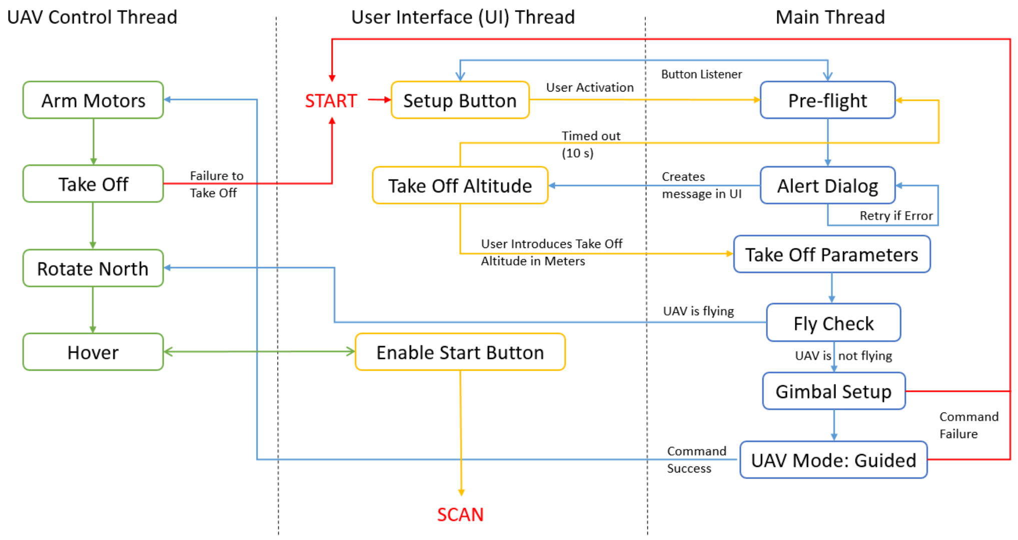


Figure 4.4: 3DR Solo State Machine of the Setup State

The state is triggered by clicking the **Start Setup** button in the UI. This is done through a button listener, a method to communicate actions from the UI to the main thread in real time provided by the Android API. A listener behaves as a boolean variable: it is set to true when clicked and false when it is not. The moment the button is clicked, it stays enabled such that it cannot be clicked twice.

Once the button is enabled, the main thread executes the pre-flight sequence. This establishes connection between the UAV and the App via MAVLink over UDP and awaits an acknowledgment from the UAV. Typically this connection was already established when the Android device first connected to the UAV's Wi-Fi, and this step only verifies the connection, as shown in Figure 4.5. The setup process will not continue until the connection is successful.

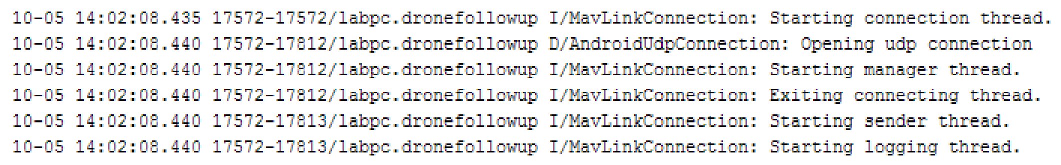

```
10-05 14:02:08.435 17572-17572/labpc.dronefollowup I/MavLinkConnection: Starting connection thread.
10-05 14:02:08.440 17572-17812/labpc.dronefollowup D/AndroidUdpConnection: Opening udp connection
10-05 14:02:08.440 17572-17812/labpc.dronefollowup I/MavLinkConnection: Starting manager thread.
10-05 14:02:08.440 17572-17812/labpc.dronefollowup I/MavLinkConnection: Exiting connecting thread.
10-05 14:02:08.440 17572-17813/labpc.dronefollowup I/MavLinkConnection: Starting sender thread.
10-05 14:02:08.440 17572-17813/labpc.dronefollowup I/MavLinkConnection: Starting logging thread.
```

Figure 4.5: Sample MAVLink Log of Solo-App UDP Connection

If the connection is successful, the next stage creates an alert dialog. A dialog is a pop-up screen which freezes the UI until either the dialog is closed or a certain action is completed by the user, here inputting a desired altitude. An alert dialog is already provided by the Android API and does not need to be implemented. The alert dialog appears in the UI as shown in Figure 4.6. Once the user enters a valid number into the dialog, it is stored for later use and the dialog is dismissed.

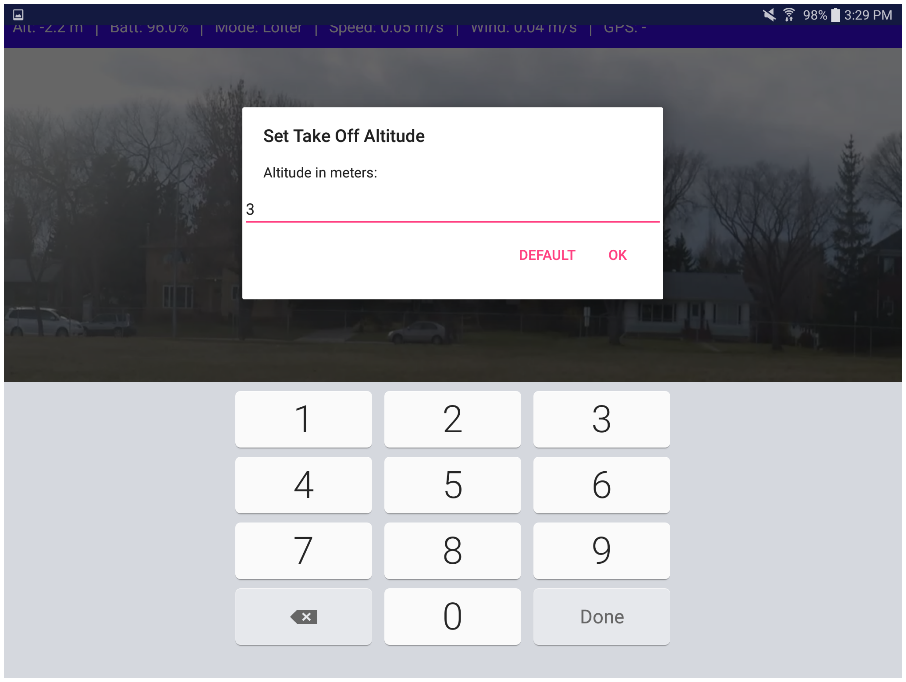

98% 3:29 PM
Alt. -2.2 m | Batt. 98.0% | Mode: Loiter | Speed: 0.05 m/s | Wind: 0.04 m/s | GPS: -
Set Take Off Altitude
Altitude in meters:
3
DEFAULT
OK
1 2 3
4 5 6
7 8 9
0
Done

Figure 4.6: Alert Dialog in the Setup State

Because the setup can be restarted at any time, including while the UAV is airborne, a Fly Check had to be introduced. This works by reading a boolean value within the Solo's control system. If the UAV is not flying then the takeoff sequence continues, otherwise the system skips ahead to the step of turning the hovering UAV towards the North, explained further below.

The next step initializes the gimbal, creating the corresponding listeners to its parameters and commanding the camera to point forward. Then, the UAV mode is set to guided, which is required to allow the UAV to receive autonomous flight commands.

If all the previous steps were successful, then the UAV arms its motors and takes off. The use of the commands 'arm' and 'take off' requires some caution. First, the 'arm' command will arm or disarm the UAV motors if they are disarmed or armed, respectively, independently of the altitude of the UAV. This is the reason why the fly check step was implemented: to avoid issuing this command when the **Start Setup** button is pressed once the UAV is in the air. Second, the 'take off' command requires not issuing any other commands until the UAV takes off and climbs to the altitude pre-specified by the user. In order to accomplish this, the speed of the UAV take off is set to its default value, and the following time delay is observed before issuing further commands:

$$\text{delay} = \text{altitude} * 500 + 1000$$

where the delay is in milliseconds and the altitude in meters.

Once the take off is successfully completed, the UAV is commanded to turn North, then hovers in place waiting for the user to press the **Start** button in the UI. The 'rotation' command is used to turn the UAV. The turn direction can be specified as relative or absolute, where the former makes the UAV rotate a specified amount of degrees from its original position, and the latter orients the UAV with respect to geographic north. Relative mode specifies the turn as a number between -180 and 180 (degrees), while absolute mode specifies it as a number between 0 and 360, where 0 denotes North. To make the UAV hover, the 'pause' command is used. This commands the UAV to maintain a level attitude and constant position in the air through closed-loop stabilization, meaning the performance is subject to the accuracy of the onboard sensors and state estimation system.

Until the "Scan" state is triggered, both the UI and control threads stay in the 'Enable Start Button' and 'Hover' blocks in Figure 4.4, respectively.

4.2.4 Scan State

As described in Section 4.1, the "Scan" state implements the vision-based algorithm of Section 3.4.3 and handles all the details required for its correct execution. This state runs the same threads as the "Setup" state, namely the main, UI and UAV control threads, and creates a new thread for the scanning algorithm.

The scan thread is responsible for running extensive computations for the vision-based algorithm and is implemented separately from the UI thread. This way, the mathematical calculations running on the CPU do not freeze the UI and do not block any user interaction. The whole process does not require user action other than triggering entry into this state, whose details are illustrated in Figure 4.7.

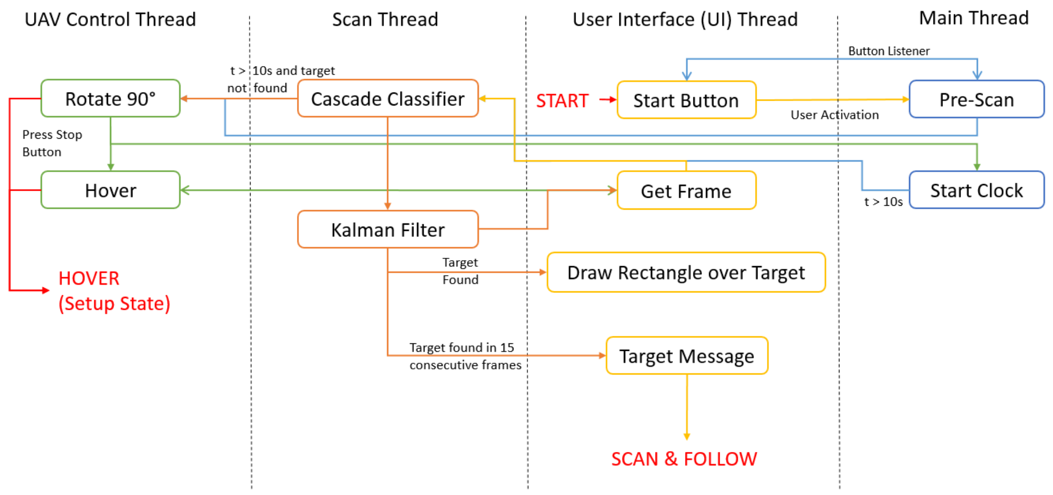


Figure 4.7: 3DR Solo State Machine of the Scan State

As in the “Setup” state, the “Scan” state is triggered by a listener in the **Start** button in the UI. Once clicked, this button stays enabled such that it cannot be clicked twice.

After clicking the **Start** button, the main thread creates the scan thread and runs the main loop in which the vision-based algorithm is executed. The timing is performed by the main thread. The scan thread is entered when the UAV is hovering and is run until a target is positively identified or 10 seconds have elapsed. In the latter case, the UAV performs a 90° clockwise relative rotation (see section 4.2.3) and the scan thread is redone.

While running, the UI thread stores the bitmap of every video frame streamed from the camera but only one at a time. The Android bitmap is a matrix which stores the color value of each pixel in ARGB_8888 (8-bit packed values of alpha, red, green and blue) format. This bitmap is then converted to grayscale to conform with OpenCV standards, using the following equation [1]:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

where Y is the grayscale value and R, G and B are the red, green and blue color values of the pixel.

The grayscale matrix is inputted into the Cascade Classifier algorithm (see Section 3.4) at each frame. If a match is made, a counter is started for the number of consecutive frames in which the target was detected. For each frame where a target is positively detected, the UI thread draws a red rectangle around it to give visual feedback to the user, as shown in Figure 4.8. If a target is detected over 15 consecutive frames, the UI thread displays the message “Target Locked” to the user and proceeds to enter the “Scan and Follow” state. Conversely, if a target is not locked onto within 10 seconds, the system executes a rotation and the scanning process is restarted.

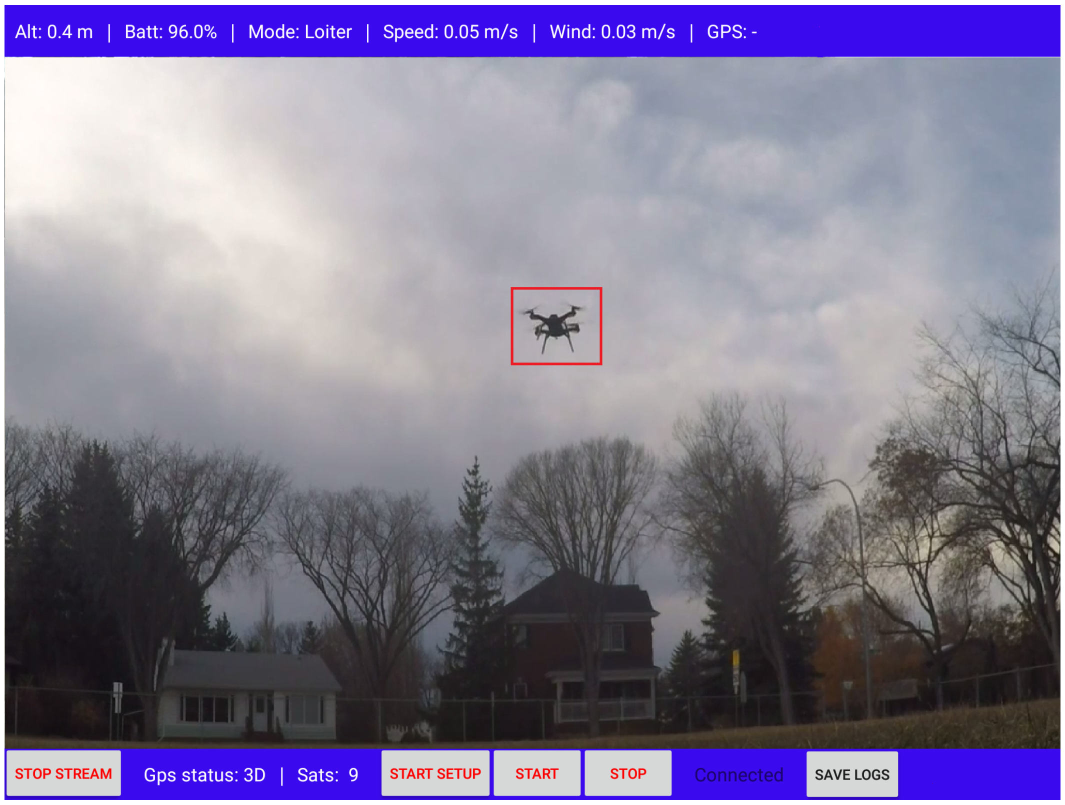

Alt: 0.4 m | Batt: 96.0% | Mode: Loiter | Speed: 0.05 m/s | Wind: 0.03 m/s | GPS: -
STOP STREAM
Gps status: 3D | Sats: 9
START SETUP
START
STOP
Connected
SAVE LOGS

Figure 4.8: User Interface Target Detection Example

Since this state runs autonomously, an emergency exit from the state is implemented, which can be triggered by either the user or the state of the battery. If the user presses the **Stop** button in the UI, they regain manual control of the UAV. The same things happens if the battery charge of the UAV falls below 25%. The latter provides the user sufficient time to find a safe landing spot and set the UAV down.

4.2.5 Scan and Follow State

As described in Section 4.1, the "Scan and Follow" state runs the vision-based Cascade Classifier algorithm (c.f. Section 3.4.3) in addition to the control system which allows the UAV to follow its target. This state runs the same threads as the "Scan" state: the UI thread, the UAV control thread and the scan thread. However, the control thread will now be used to make the UAV to follow its target. The state does not require user interaction and is shown in detail in Figure 4.9.

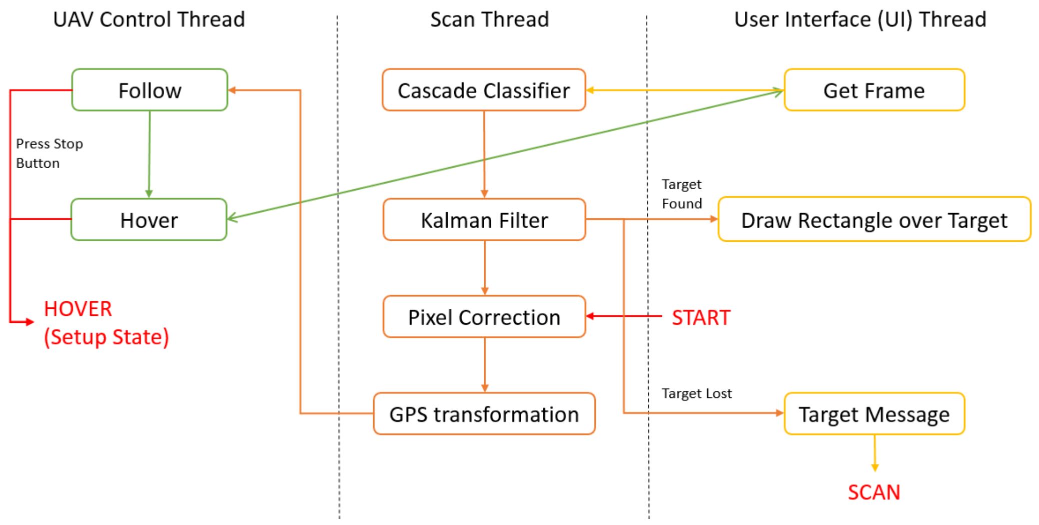


Figure 4.9: 3DR Solo State Machine of the Scan and Follow State

The "Scan and Follow" state is triggered from within the "Scan" state once a UAV target is locked, designating it as a the target to follow for the control system. The pair of pixels in which the UAV target's centroid is located $[x, y]$ and its bounding box size $[z]$ are obtained from the preceding "Scan" state. First, the pixel correction transforms the input position of the UAV into a real relative distance in meters from the center of the field of view of the camera, as described in Section 4.4. Next, the relative distance is transformed into GPS geodesic coordinates, described in Section 4.2.6.

Once the GPS transformations are done, we have two sets of parameters: a set of geodesic coordinates $[\lambda,\phi]$ and a relative movement $[\Delta x,\Delta y,\Delta z]$. Both are needed by the Follow control system. The set of coordinates will be the position to which the UAV will be commanded to fly to, but these are subject to accuracy errors of the onboard state estimation system. Thus the relative position of the target will be employed to fine-tune the position of the drone relative to its target.

Once the flight motion is done, the drone hovers while a new scan is performed, identically to the "Scan" state. Due to the fast processing of the frames and the high frame rate (30 FPS in the lowest case scenario), the UAV appears as being in constant movement.

If the target is not detected in a frame, for instance due to occlusion of the camera, the Kalman Filter will continue to predict the position of the target UAV (c.f. Section 3.3.3). If the target is not detected over 30 consecutive frames, the "Target Locked" message disappears and the system reverts back to the "Scan" state.

4.2.6 GPS Transformations

As previously mentioned in Chapter 2, the control system for the 3DR Solo accepts target positions specified as a set of geodetic coordinates (latitude and longitude) in decimal degrees. However, the actual control calculations are performed in the (orthogonal) ECEF frame, and the results are translated into the geodetic coordinates.

As seen in Section 3.4, the vision-based tracking algorithm returns, after running a Kalman Filter (c.f. Section 4.4), the desired motion [Δx,Δy,Δz] of the UAV in the navigation North-East-Down reference frame, in units of centimeters. This reference frame can rotate in relation to the ECEF frame so it is paramount to know the absolute direction in which the camera is looking at. The GPS transformation equations depend on whether the camera points North, South, East or West, but this orientation is available from the on-board compass on the UAV. Due to the complexity of the calculus of small 3D motions in geodetic coordinates, we will estimate the small changes in latitude and longitude from small changes in North and East positions [59]. Therefore, the flat Earth Δx and Δy coordinates are then transformed to North East coordinates:

$$\begin{bmatrix} \delta N \\ \delta E \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{4.1}$$

where ψ is the angle in degrees, clockwise, between the x-axis and the North. Then, to convert the North East coordinates to geodetic latitude and longitude coordinates, the estimation uses the radius of curvature in the prime vertical R_N and the radius of curvature in the meridian R_M. Both are defined in Equation 4.2.

$$\begin{aligned} R_N &= \frac{R}{\sqrt{1-(2f-f^2)\sin^2\lambda_0}} \\ R_M &= R_N \frac{1-2f+f^2}{1-(2f-f^2)\sin^2\lambda_0} \end{aligned} \tag{4.2}$$

where R is the equatorial radius and f is the flattening of the planet. Small changes in both latitude and longitude are then approximated by the following equations:

$$\begin{aligned} \delta\lambda &= \tan^{-1}(\frac{1}{R_M})\delta N \\ \delta\phi &= \tan^{-1}(\frac{1}{R_N\cos\lambda})\delta E \end{aligned} \tag{4.3}$$

Then, the output latitude and longitude are obtained adding the small variation to the previous location. The altitude is the negative flat z-axis value minus the reference height, h_{ref}.

$$\begin{aligned} \lambda &= \lambda_0 + \delta\lambda \\ \phi &= \phi_0 + \delta\phi \\ h &= -\Delta z - h_{ref} \end{aligned} \tag{4.4}$$

These geodetic [λ,ϕ,h] coordinates represent the position in which the drone must be in order to mimic the movement of the target UAV.

4.3 Logic and Control Strategy for Parrot AR.Drone 2.0

4.3.1 Overview

The control strategy for the Parrot AR.Drone 2.0 is to make the flight autonomous, from the take-off to the UAV following phase, with only a minimum number of user actions on the graphical interface of the Android app. In other words, no inputs on the virtual sticks are required for the Detection and Following algorithms to operate. For the Parrot UAV, most of the autonomous flight functionality is already implemented in the open-source Parrot SDK, and our Detection and Following algorithms were implemented by adding new modules into the SDK codebase.

The overall system design for the Parrot AR.Drone 2.0 can be defined in the same three main states as for the 3DR Solo (see Figure 4.2), but the main and most important difference is that the Parrot AR.Drone 2.0 is unable to sense and feedback its own position in space, due to the lack of GPS signals in the indoor lab. For this reason, the control strategy is an open-loop feed-forward control. In the future, an indoor optical motion capture system such as the one seen in [5] will be installed in order to provide this missing data.

As before, each state block creates and deletes software threads as needed in order to reduce memory use and computation time on the ground computer tablet.

The "Setup" state handles all the pre-flight steps, including connecting via Wi-Fi to the UAV, enabling the camera and starting the video stream from it, plus the initial flight phase of arming the motors followed by autonomous take-off to a pre-specified altitude. This state is triggered by the **Piloting** button in the dashboard activity section within the user interface (UI) of the app running on the ground computer.

The "Scan" state follows the "Setup" state and is triggered by the user pressing the **Target** button in the video activity section of the UI. Within this state, the vision-based algorithm from Section 3.4 runs persistently in its own thread. This state puts the UAV in hover, runs the vision algorithm for 10 seconds, then commands a 90° clockwise yaw and loops back to the vision algorithm. At any time, if a target UAV is detected by the vision algorithm over at least 40 consecutive frames, the system moves to the next state.

The next and final state, "Scan and Follow", is triggered when a target UAV is found. This state executes a pursuit control while running the vision-based algorithm, each in its own thread. This state runs until the target is lost, at which point the system returns to the previous state, "Scan".

Within any state, pressing the **Target** button in the video activity section of the UI reverts the system to the end of the "Setup" state, i.e. hovering in place and waiting for the user to press the **Start** button.

In the following subsections, we will describe the user interface and its components. Then, we will provide the details of the individual states as well as the transition conditions between them.

4.3.2 Parrot App User Interface

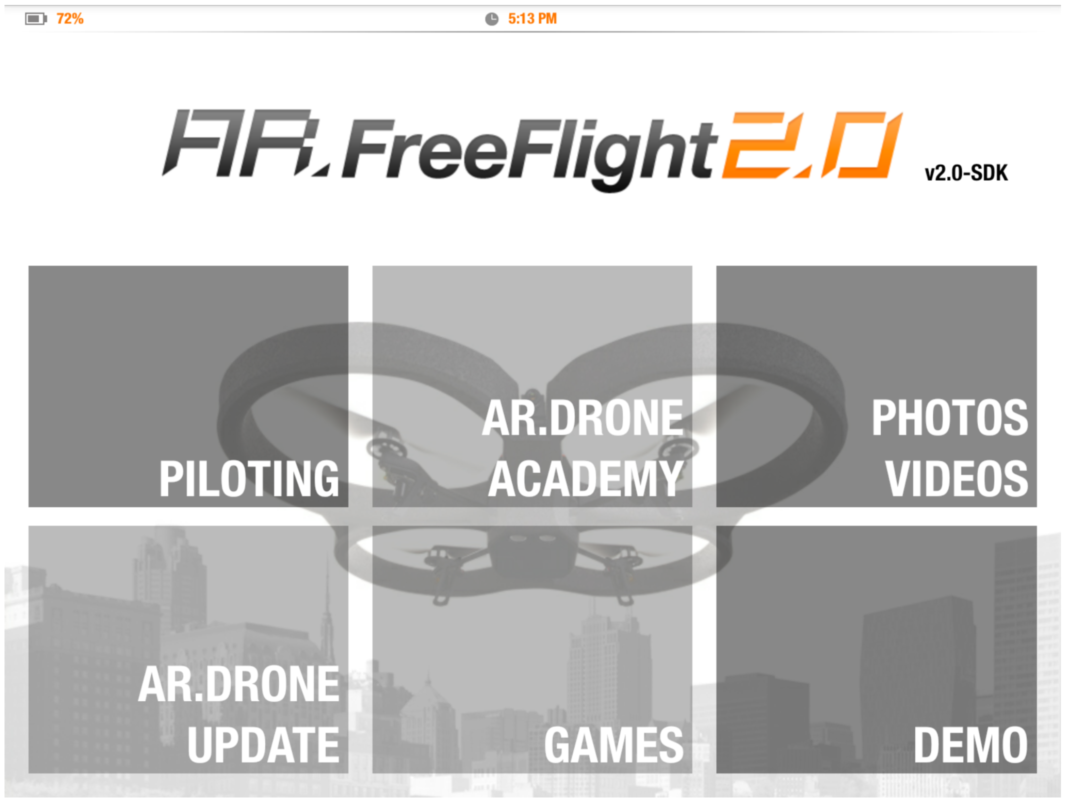


Figure 4.10: Parrot App Initial Menu UI

The Parrot SDK App for Android starts with a tabloid-like high-level menu shown in Figure 4.10. While a wide range of modes, such as accessing a library of photos and videos taken by the drone, or accessing an online academy which teaches how to fly the Parrot UAV, we are only interested in the piloting mode. Clicking the "Piloting" button enters the flight user interface.

This UI is divided into three zones denoted as A, B and C in Figure 4.11. The top zone (A) displays real-time information about the UAV status including the battery charge (in %), a button to access UAV connection settings, the Wi-Fi signal intensity, the emergency status signal which flashes when an emergency occurs during flight (this triggers a forced landing and shutdown of the motors), plus buttons for recording videos and taking pictures from the onboard camera. Zone B is a custom surface texture displaying the video streamed from the UAV's camera. Zone B is a touch-surface which displays virtual joysticks on the left and right sides of the display, allowing the user to control the UAV directly from the App UI. Zone C has one toggle button used to autonomously take off or land the UAV, depending on its state, and a second toggle button (blue or red ⊕ icon) which allows triggering or interrupting the vision-based algorithm thread, respectively. This button will be henceforth be referred to as the **Scan** or **Stop** button, respectively. Other than the start/stop button, all the elements of the UI are provided by the stock SDK code.

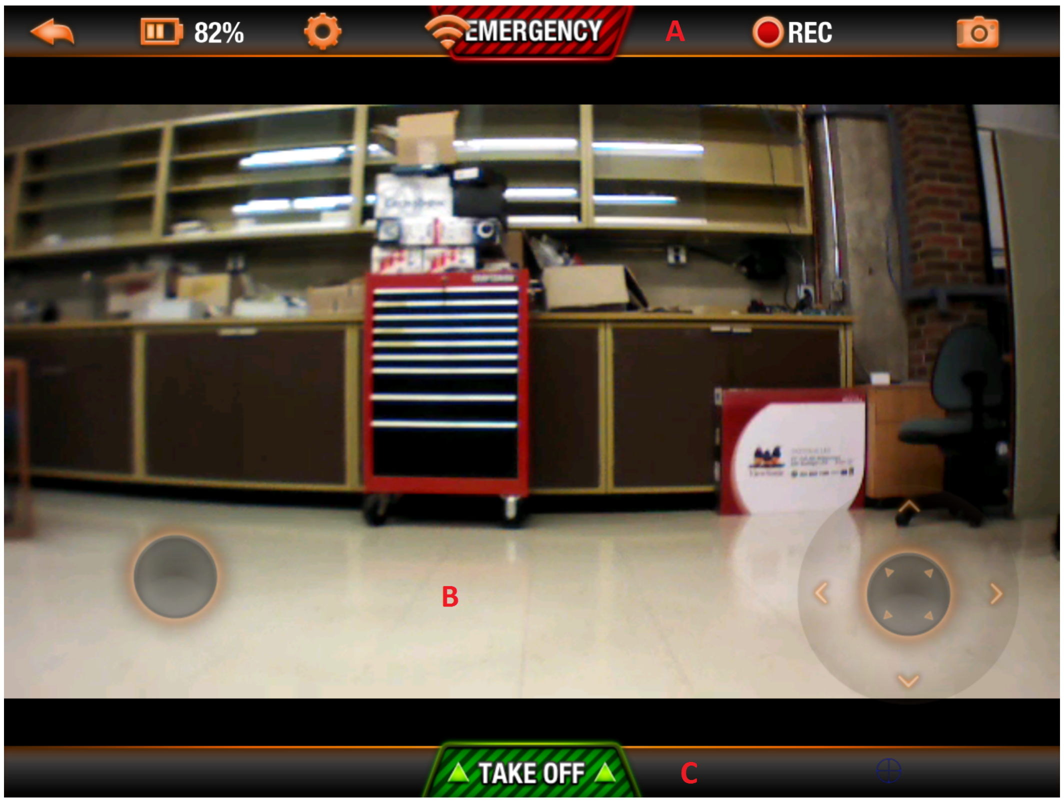


Figure 4.11: Flight User Interface

4.3.3 Setup State

As mentioned in Section 4.3.1, the “Setup” state prepares the UAV to run the vision-based pursuit algorithm, starting with pre-flight preparations and ending at hovering in the air waiting for the user to launch the scanning algorithm. This state employs a number of software threads to accomplish its tasks, but we will focus mainly on our custom-written threads which implement the vision-based algorithm. We will focus primarily on the main thread, the UI thread and the UAV control thread. The main thread and the UI thread are created when the App launches, while the UAV control thread is created only when needed and removed immediately after, in order to minimize the number of running threads and thus economize memory and computational load.

The “Setup” state is automated and requires user input at one point only. This state is triggered by the **Piloting** button in the activity section of the Parrot App UI. Once the button is clicked, the main thread executes the pre-flight sequence. This establishes a connection between the UAV and the App over Wi-Fi and waits for an acknowledgment from the UAV. Typically this connection is already established by the tablet before the App is launched, and in this case the thread only confirms the connection. The setup process will not proceed until the Wi-Fi is successfully established.

Once the connection is successful, the next stage initializes a live video stream from the onboard camera, which is displayed within Zone B of the UI.

If all the previous stages complete successfully, the UAV arms its four motors then sends a confirmation of success to the ground tablet. Upon receipt of this confirmation, the App displays the “Take Off” button and waits for it to be pressed, which lets the user decide when the conditions are right for take off and flying. After the button is pressed and the take off sequence is completed successfully, the UAV is commanded to hover in place and the app starts waiting for the user to press the **Scan** button in the UI. The performance of the UAV in maintaining a steady hover is subject to the accuracy of the onboard sensors, optical flow calculations and the performance of the onboard state estimation system.

4.3.4 Scan State

As described in Section 4.3.1, the “Scan” state implements the vision-based algorithm described in Section 3.4.3 and handles the details of its execution. This state runs the same threads as the “Setup” state, namely the main, UI and UAV control threads, and also creates a new thread to run the scanning algorithm.

The scan thread requires extensive computation resources to run the vision-based algorithm. It is thus implemented separately from the UI thread such that its computations do not freeze the UI and do not block any user interaction. The entire process does not require any user interactions, and its details are almost identical to the ones for the 3DR Solo as illustrated in Figure 4.7.

As in the “Setup” state, the “Scan” state is triggered by a listener in the **Scan** button within the UI. Once clicked, this button changes into the **Stop** button, which allows interrupting the algorithm at any time.

After clicking the **Scan** button, the main thread launches the scan thread which runs the main loop of the vision-based detection algorithm. Timing is performed by the main thread. The scan thread is entered when the UAV is hovering and runs until either a target is positively identified or 10 seconds have elapsed. In the latter case, the UAV performs a 90° clockwise rotation and the scan thread is restarted. The ‘turn’ command is used to turn the UAV. The turn direction can be specified as clockwise or counterclockwise, where the former makes the UAV rotate a specified amount of degrees from its original position towards the right and the latter towards the left. Similarly to the process described in Section 4.3.6, the ’turn’ command needs to be calibrated.

While running, the UI thread stores a bitmap of the latest video frame streamed from the camera. This bitmap is then converted to grayscale as required by OpenCV to speed up calculations, as discussed in Section 4.2.4. The grayscale image is inputted into the Cascade Classifier algorithm (see Section 3.4) at each frame. The detection logic for the Parrot AR.Drone 2.0 is exactly the same as the one specified for the 3DR Solo in Section 4.2.4.

Since the “Scan” state runs autonomously, an emergency exit from the state is implemented, which is triggered either by the user or by the state of the battery. If the user presses the **Stop** button in the UI, they regain manual control of the UAV. The same thing occurs if the battery charge of the UAV falls below 10%. The latter condition provides sufficient time for the user to safely land the UAV.

4.3.5 Scan and Follow State

As described in Section 4.1, the “Scan and Follow” state runs the Cascade Classifier vision algorithm (c.f. Section 3.4.3) as well as the control system which allows the UAV to pursue its target. This state runs the same threads as the “Scan” state: the UI thread, the UAV control thread and the scan thread. However, the control thread is now used to make the UAV follow its target. This state does not require user interaction and is shown in detail in Figure 4.12.

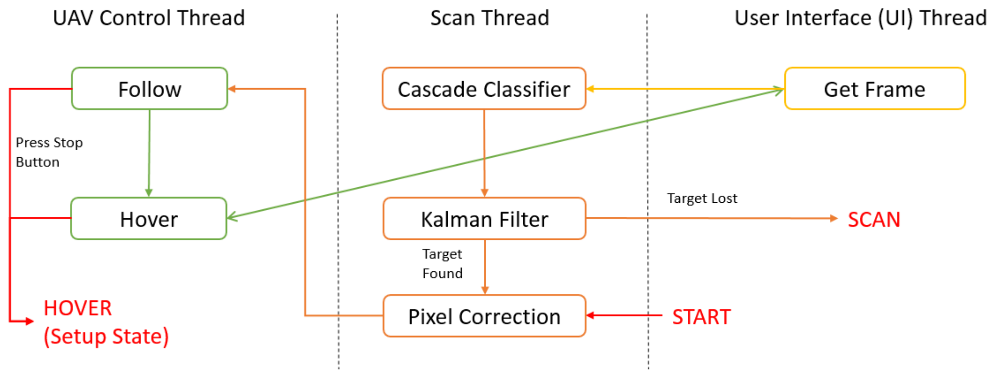


Figure 4.12: Parrot AR.Drone 2.0 State Machine of the Scan and Follow State

The “Scan and Follow” state is entered from within the “Scan” state once a UAV target is locked, designating it as the target to follow for the control system. The UAV target’s pixel location $[x, y]$ and its size $[z]$ is obtained from the earlier “Scan” state. First, as described in Section 4.4, the perspective correction equations transform the input position of the UAV into a real relative distance in meters from the center of the field of view of the camera.

The vision algorithm outputs a set of coordinates $[\delta x, \delta y, \delta z]$ which represent the relative movement to be mimicked by the UAV in the interval between frames. The control system executes the movements along the three axis independently. This is accomplished using the ‘set’ command provided by the SDK. The ‘set roll’ and ‘set pitch’ commands set the fraction of the maximum inclination angle, which is preset in the quadcopter settings, while the ‘set gaz’ command sets the fraction of the maximum vertical speed. Because the Parrot does not provide position measurement feedback (c.f. Section 4.3.1), this adds the problem of having to time the commands issued by the control system. Therefore, an extensive calibration of the relation between input power, time of execution (which will always be smaller than the frame rate) and distance needs to be carried out (see Section 4.3.6).

Once the flight motion is done, the drone hovers while a new scan is performed, identically to the “Scan” state. Due to the fast processing of the frames and the high frame rate of the video feed (30 FPS for the Parrot AR.Drone 2.0), the UAV appears as being in constant movement.

If the target is not detected in a given frame, for instance due to occlusion of the camera, a Kalman Filter continues to predict the position of the target UAV (c.f. Section 3.3.3). If the target is not detected over 30 consecutive frames, the “Target Locked” message disappears and the system reverts back to the “Scan” state.

4.3.6 Input Thrust Calibration

The Parrot AR.Drone 2.0 control system defines a body-fixed reference frame whose origin is placed at the UAV's center of gravity, as illustrated in Figure 4.13. The UAV moves forward and backwards along the Pitch axis, left and right along the Roll axis, and up and down along the Gaz axis (the french-language equivalent of the Thrust axis).

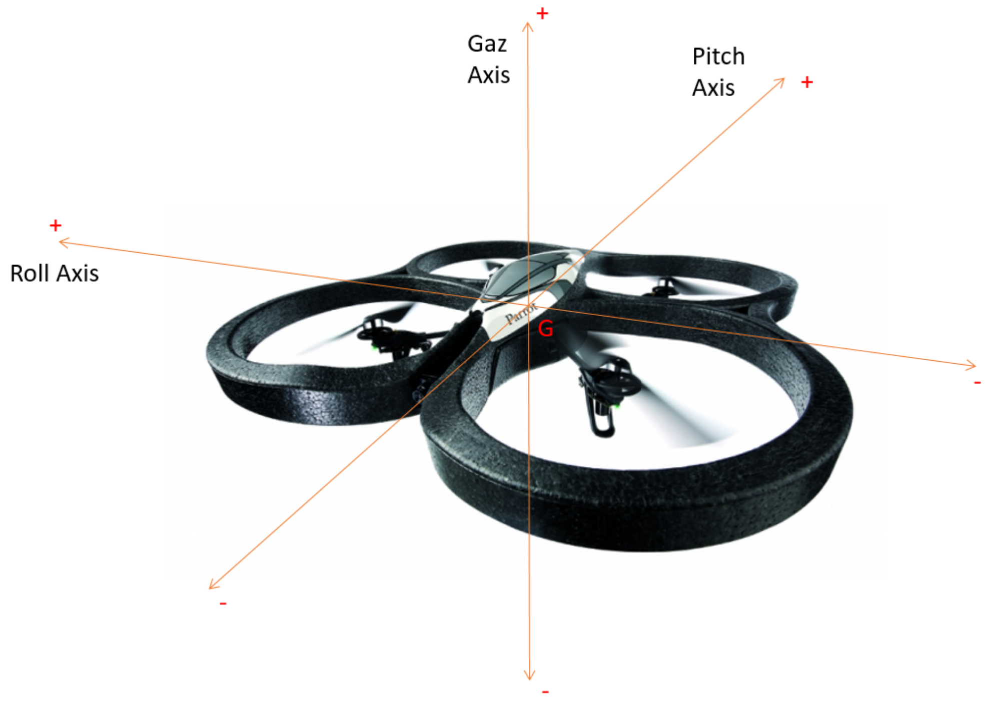


Figure 4.13: Parrot AR.Drone 2.0 Motion Axis Representation

The Parrot AR.Drone 2.0 native commands 'setRoll', 'setPitch' and 'setGaz' command movements of the UAV along each axis at a specified “power” value. “Power” is a real number between -1 and 1 which represents a normalized version of the tilting angle or vertical speed at which the UAV will perform motion along the specific axis. However, the tilting angle relates directly with the velocity of the quadcopter on each axis (pitch or roll), for instance the UAV will move to the right at its maximum speed when the 'setRoll' command is set to 1, and conversely will move to the left at its maximum speed when the 'setRoll' command is set to -1. Once a ‘set’ command is issued, the UAV maintains the commanded speed indefinitely, meaning the commands need to be progressively issued in order to obtain smooth flight motions.

Because the outputs from the vision-based algorithm as well as the Kalman filter provide estimates of the relative movement of the target UAV in meters, we need to establish a mathematical model which relates input power and time of execution of the velocity command with the resulting distance traveled.

In order to obtain this model, a set of experiments were carried out. Each of the three 'set' commands were executed individually for a specific amount of time and a given power value. The distance flown along the given axis was measured using a laser distance meter (Leica Disto D2). An example of the method called in the control thread to perform this test is the following:

```
private void testingMove() {
  AsyncTask.execute(new Runnable() {
  @Override
  public void run() {
    long startTime = System.currentTimeMillis();
    int testTime = 200;
    droneControlService.setProgressiveCommandEnabled(1);
    Log.d("ControlDroneParrot","Started moving. For " +
       Integer.toString(testTime) + "ms.");
    while(System.currentTimeMillis() - startTime <
       testTime) {
      // droneControlService.setPitch(power); Going
         forward (-1) and backwards (1)
      // droneControlService.setRoll(power); Going Left
         (-1) right (1)
      // droneControlService.setGaz(power); Going up (1)
         and down (-1)
      droneControlService.setGaz(1);
      }
    droneControlService.setGaz(0);
    Log.d("ControlDroneParrot","Finished moving.");
    }
  });
}
```

The Parrot AR.Drone 2.0 streams video from its onboard video camera to the Parrot App at 30 frames per second. The control system for the UAV has to perform the 3-axis movement in-between frames, meaning each command may run over a maximum of 0.03 seconds. Preliminary testing showed that for this UAV hardware, even at full power it was not possible to perform a measurable movement in such a short interval of time, meaning the vision system cannot run at 30 frames per second. Further testing showed that at full power, the minimum time required for the UAV to exhibit measurable movement (not considered drifting) is around 0.2 seconds (see Figure 4.14). We can thus conclude that the Parrot AR.Drone 2.0 will be limited to 1 frame per second, which will limit the achievable performance by the control system.

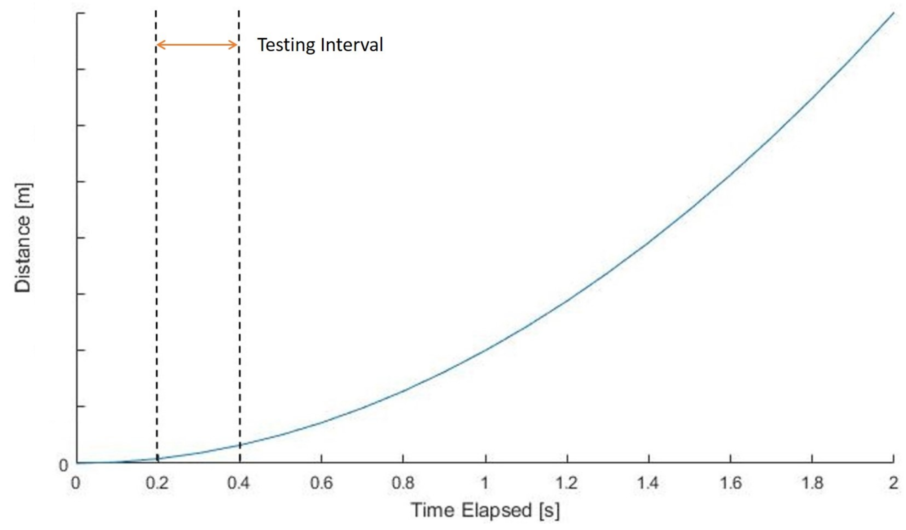
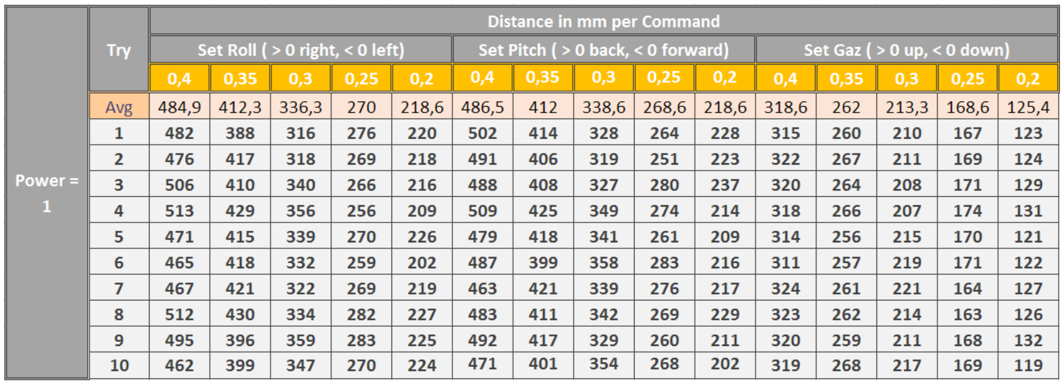


Figure 4.14: Parrot AR.Drone 2.0 Input Expected Thrust Calibration Results

Testing was performed to obtain a relationship between the time spent at full input power and the resulting distance traveled. Given the quadratic relationship between thrust and distance, as show in Figure 4.14, the testing interval will be limited to values near 0.2 seconds to allow linearization without inferring excessive error. The results of these tests are shown in Table 4.15:

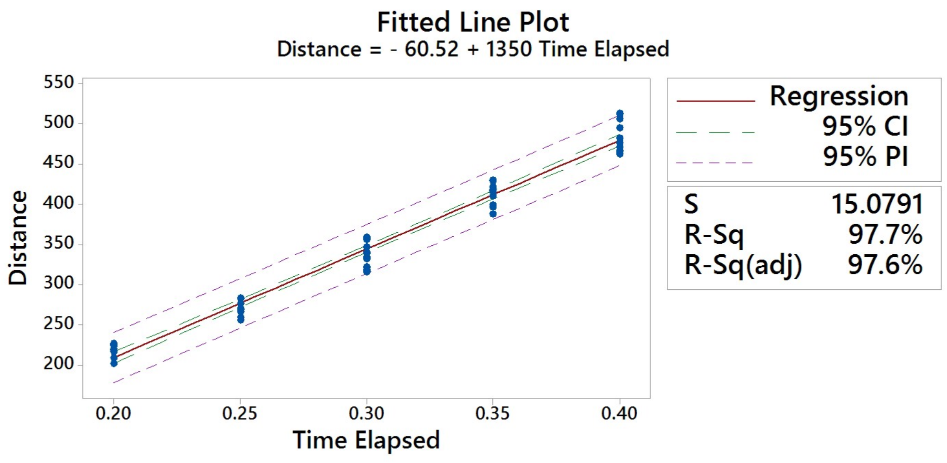


	Try	Distance in mm per Command														
		Set Roll (> 0 right, < 0 left)					Set Pitch (> 0 back, < 0 forward)					Set Gaz (> 0 up, < 0 down)				
		0,4	0,35	0,3	0,25	0,2	0,4	0,35	0,3	0,25	0,2	0,4	0,35	0,3	0,25	0,2
Power = 1	Avg	484,9	412,3	336,3	270	218,6	486,5	412	338,6	268,6	218,6	318,6	262	213,3	168,6	125,4
	1	482	388	316	276	220	502	414	328	264	228	315	260	210	167	123
	2	476	417	318	269	218	491	406	319	251	223	322	267	211	169	124
	3	506	410	340	266	216	488	408	327	280	237	320	264	208	171	129
	4	513	429	356	256	209	509	425	349	274	214	318	266	207	174	131
	5	471	415	339	270	226	479	418	341	261	209	314	256	215	170	121
	6	465	418	332	259	202	487	399	358	283	216	311	257	219	171	122
	7	467	421	322	269	219	463	421	339	276	217	324	261	221	164	127
	8	512	430	334	282	227	483	411	342	269	229	323	262	214	163	126
	9	495	396	359	283	225	492	417	329	260	211	320	259	211	168	132
	10	462	399	347	270	224	471	401	354	268	202	319	268	217	169	119

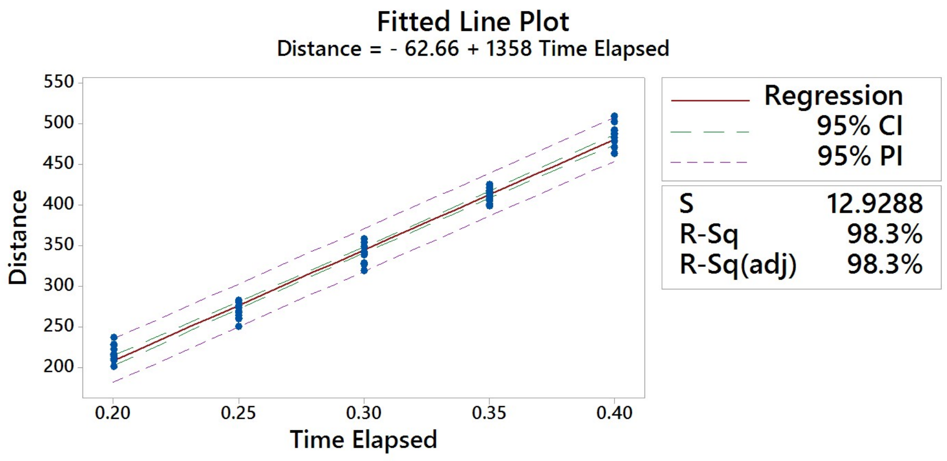
Figure 4.15: Parrot AR.Drone 2.0 Input Thrust Calibration Results

Note that the pitch and roll axes exhibit nearly identical results, which is as expected due to highly symmetric design of the Parrot AR.Drone 2.0. The relationship is obtained through regression analysis with a standard linear model: y = b_0 + b_1x.

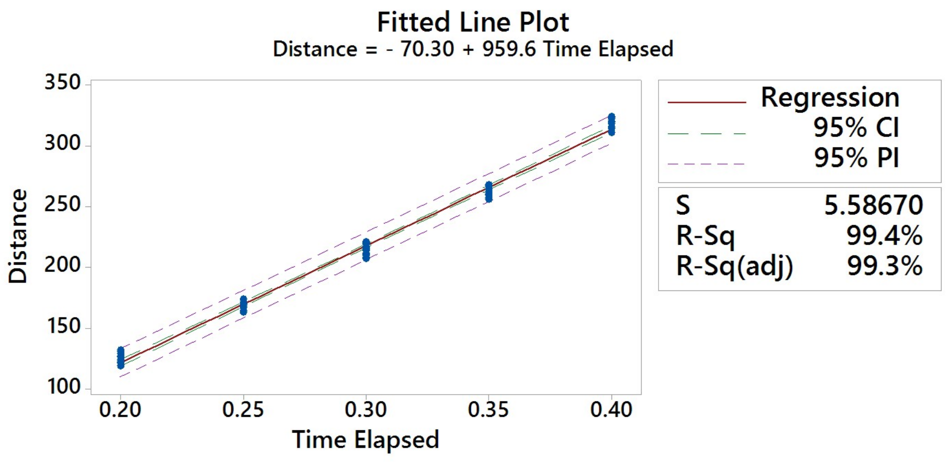
The fitting results for movements along the three individual axes is shown in Figure 4.16.



(a) Regression Analysis For The Roll Axis Movement



(b) Regression Analysis For The Pitch Axis Movement



(c) Regression Analysis For The Gaz Axis Movement

Figure 4.16: Regression Analysis Fitting Results

The results of the regression analysis provide the linear fit between the time elapsed during the command execution and the distance travelled by the UAV. The results are as follows:

$$\begin{aligned} D_{Roll} &= -60.52 + 1350t \\ D_{Pitch} &= -62.66 + 1358t \\ D_{Gaz} &= -70.30 + 959.6t \end{aligned} \tag{4.5}$$

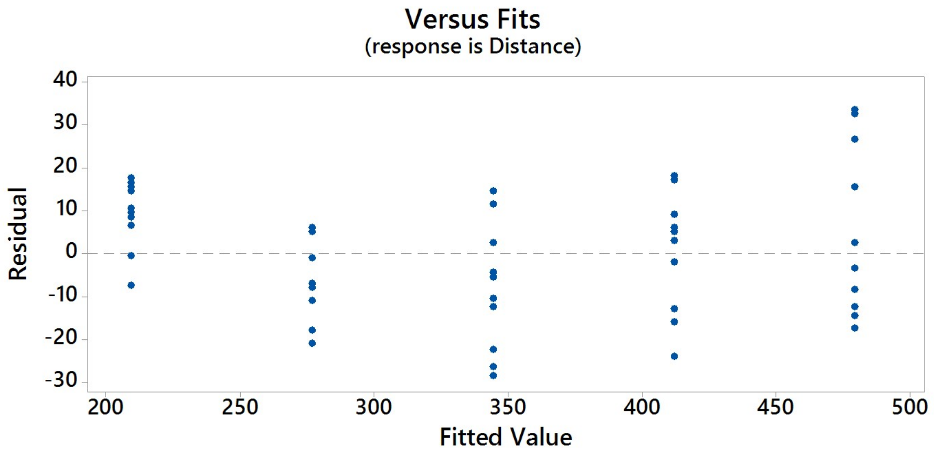
where D$_{Roll}$, D$_{Pitch}$ and D$_{Gaz}$ are the distances along each axis in millimeters, and t is the time elapsed in seconds during the execution of each command.

For all three models, the coefficient of determination R^2 is close to one, indicating that a linear model is appropriate, however the estimation may be biased. Therefore, all three linear models are validated through a residual analysis performed in MiniTab, checking the homogeneity of the "residual versus fit" plot and the isotropy of the residuals in the "normal probability" plot.

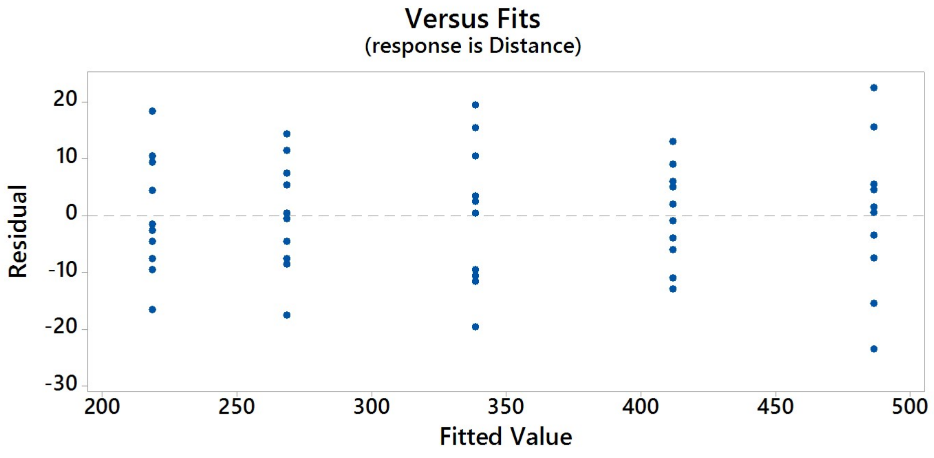
Figures 4.17a, 4.17b and 4.17c represent the scatter plot of the distance residuals vs the fitted values (estimated distance). From them we note that the magnitude of the residuals is reasonable (less than 10% error) and that they are centered around zero. Based on these observations we are able to conclude homogeneity of all the experimental residuals.

Figures 4.18a, 4.18b and 4.18c represent the normal probability plot of the distance residual for each motion. From them, we see that all the residuals are equally distributed with respect to the null residual along the fitting line and that there are no outliers. Thus, we consider all the experimental residuals to be isotropically distributed.

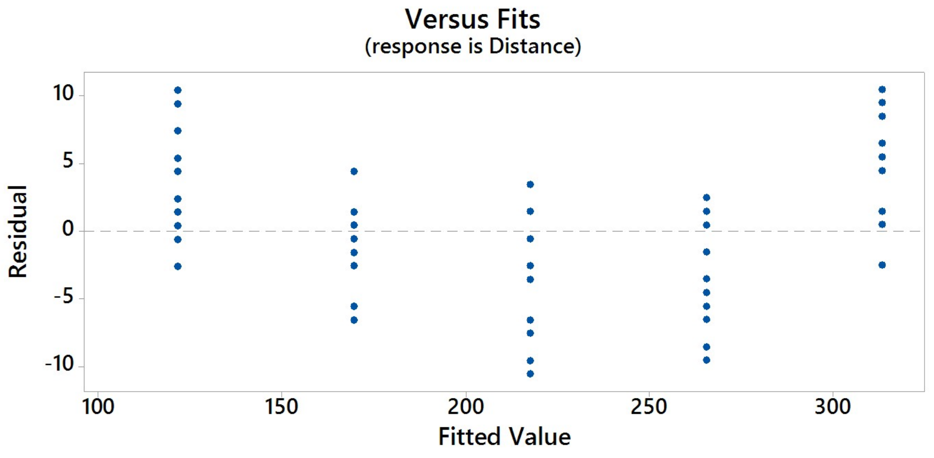
We can thus conclude that the linear models assumed in Equations 4.5 are indeed valid.



(a) Residual Versus Fit For The Roll Axis Movement

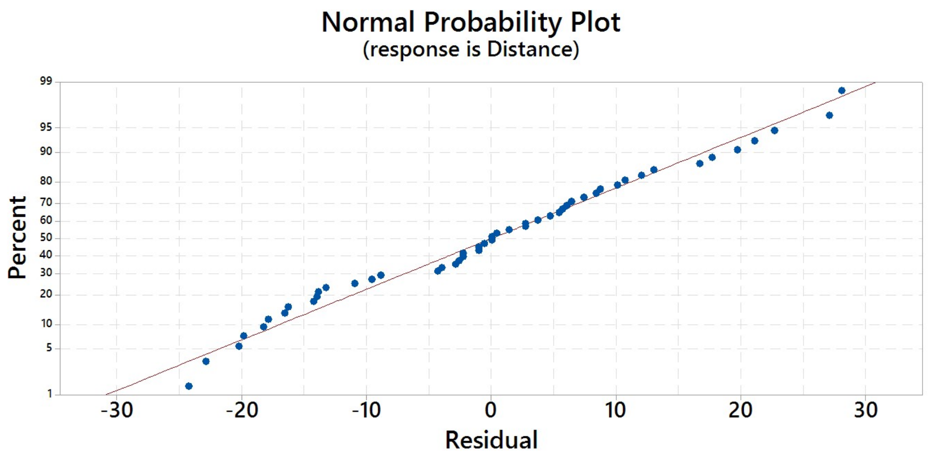


(b) Residual Versus Fit For The Pitch Axis Movement

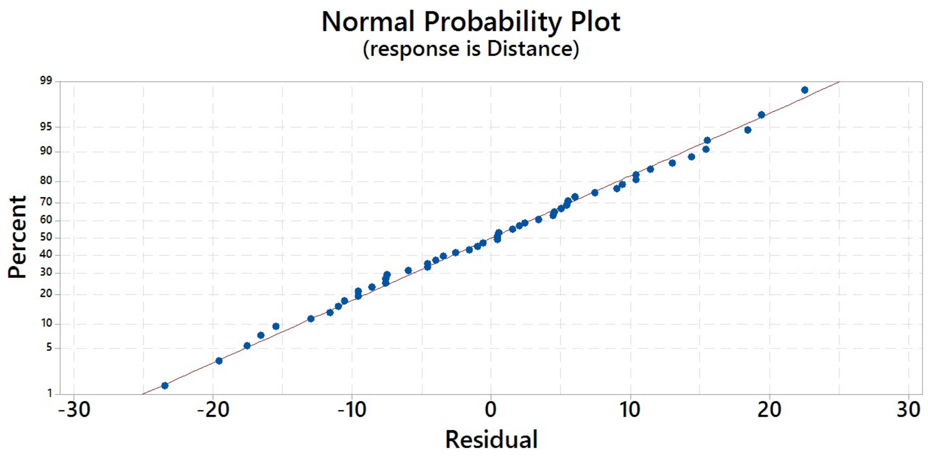


(c) Residual Versus Fit For The Gaz Axis Movement

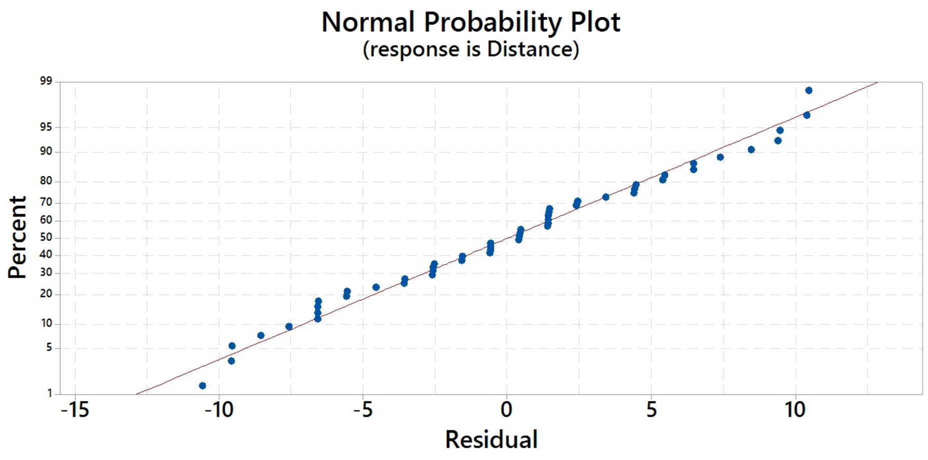
Figure 4.17: Residual Versus Fit Results



(a) Normal Probability Residual For The Roll Axis Movement



(b) Normal Probability Residual For The Pitch Axis Movement



(c) Normal Probability Residual For The Gaz Axis Movement

Figure 4.18: Normal Probability Residual Results

4.4 Motion Estimation

4.4.1 Overview

The vision-based tracking algorithm described in Section 3.4 does not output data which can be directly used for closed-loop target tracking. For a given frame number n, the data consists of a bounding box which delineates the dimensions of the detected UAV. This box is described by an array of 4 variables: the first two, $[x_n^{bbox}$,$y_n^{bbox}]$, are the pixel coordinates of the top left corner of the bounding box, while the third and fourth are its width, b_n^w, and height, b_n^h, respectively. The centroid of the target UAV can be calculated as:

$$x_n^p = x_n^{bbox} + \text{round}(\frac{b_n^w}{2})$$
$$y_n^p = y_n^{bbox} + \text{round}(\frac{b_n^h}{2})$$

where $[x_n^p$,$y_n^p]$ are the pixel coordinates of the centroid of the target UAV. The set of pixel coordinates $[x_p$,$y_p]$ needs to be converted into a set of relative movement axes [X,Y,Z] in the UAV reference frame in order to provide a valid reference trajectory for the position control system of the UAV. This requires calculating a 3D movement from a 2D image (see Figure 4.19).

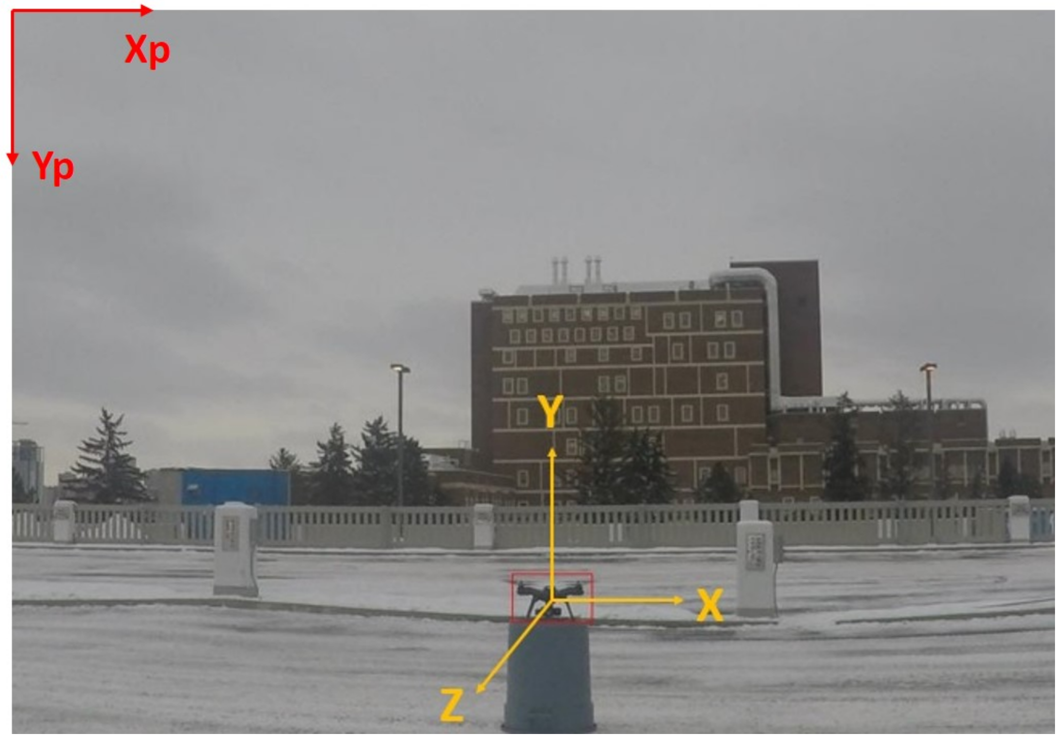


Figure 4.19: Target Reference Representation in Camera Frame

In order to estimate the relative depth movement of the UAV, or more generally the relative 3D movement of the UAV, we will subtract the results of the vision-based algorithm over two consecutive frames and then transform this result from pixels to centimeters. For this last step, the exact dimensions of the target UAV need to be known. The equations in (4.6) summarize the distance estimation calculations for the nth frame. These equations assume that the pursuer is stationary because there is currently no easy way of obtaining precise velocity of the pursuing vehicle.

$$\begin{aligned}
\delta x_n &= x_{uav} f_{lx}(x_n^{corr} - x_{n-1}^{corr}) \\
\delta y_n &= y_{uav} f_{ly}(y_n^{corr} - y_{n-1}^{corr}) \\
\delta d_n &= \sqrt{|b_n^h b_n^w - b_{n-1}^h b_{n-1}^w|} \\
\delta r_n^2 &= f_{lx}((x_n^{corr})^2 - (x_{n-1}^{corr})^2) + f_{ly}((y_n^{corr})^2 - (y_{n-1}^{corr})^2) \\
\delta z_n &= z_{uav}\text{sign}(b_n^h b_n^w - b_{n-1}^h b_{n-1}^w)\sqrt{|\delta d_n^2 - \delta r_n^2|}
\end{aligned} \tag{4.6}$$

where f_{lx} and f_{lx} are the focal length coordinates of the camera as calculated in Section 4.4.2, [δx_n,δy_n,δz_n] is the relative movement of the target UAV in the nth frame, [x_n^{corr},y_n^{corr}] are the corrected set of pixel coordinates of the target UAV (see Section 4.4.2), δd_n is the estimated relative distance in a straight line between the target UAV and the pursuing UAV, and [x_{uav},y_{uav},z_{uav}] is the relative size vector of the target UAV in cm/pixel, which is calculated in the first frame at which the target UAV has been locked on by the vision-based algorithm. This vector needs to be re-computed whenever a lock has been lost then re-established.

4.4.2 Camera Parametrization for 2D Depth Estimation

In order to get a correct estimation of the motion of the target UAV, it is important to identify the optical parameters of the camera. We are mostly interested in obtaining the angles for the vertical and horizontal field of view (see Figure 4.20) of the GoPro Hero4 Black camera equipped onboard the 3DR Solo, as well as the distortion in the distance measurements caused by the lens being spherical. For the Parrot AR.Drone 2.0, no camera parametrization was performed as the on-board camera field of view is relatively small and no pronounced fish-eye effect was observed.

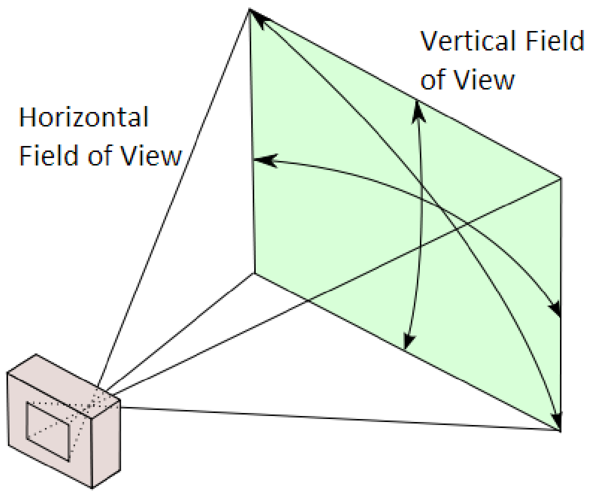


Figure 4.20: Graphic Representation of the Field of View

To obtain the distortion parameters, we use the camera calibration module available in OpenCV. The algorithm outputs five distortion parameters in a row matrix $[k_1,k_2,p_1,p_2,k_3]$, related to the pixel correction of the image (see equations (4.7)). The distortion in the GoPro Hero 4 Black is modeled by a radial factor, which manifests itself as a fish-eye effect in the image, and a tangential distortion due to the image not being parallel to the imaging plane. For a set of pixel coordinates $[x_p,y_p]$, we denote their corresponding position in the corrected output image as $[x_p^{corr},y_p^{corr}]$. The distortion can be corrected by the following formulas [8, 9]:

$$\begin{aligned} x &= \frac{x_p - x_{max}}{f_{lx}} \\ y &= \frac{y_p - y_{max}}{f_{ly}} \\ r^2 &= x^2 + y^2 \\ x_p^{corr} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_p^{corr} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^2) + 2p_2 xy \end{aligned} \tag{4.7}$$

To identify the distortion parameter values, a black-and-white chessboard pattern shown in Figure 4.21 is printed. In order to employ the OpenCV camera calibration algorithm, we need to take snapshots of this pattern with the camera from different angles and distances. Because of noise effects in the input images, a high number of snapshots is preferred. For this test, we will use 20 different snapshots.

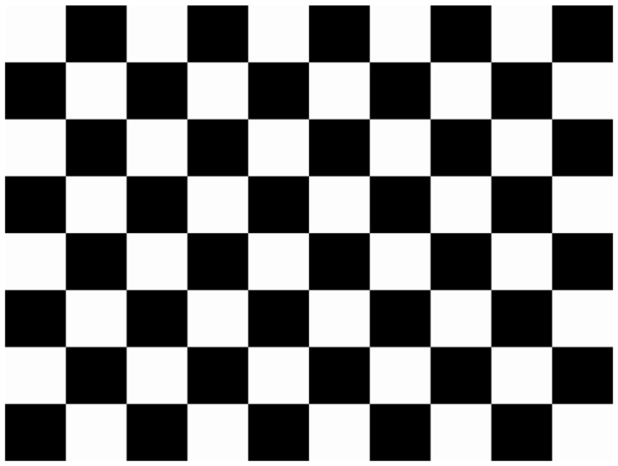


Figure 4.21: Image of the Test Chessboard

The resulting identified parameter values are given in Table 4.1.

Table 4.1: OpenCV GoPro Hero 4 Black Calibration Results

Parameter	**Estimation**
Horizontal FoV	93.5°
Vertical FoV	61.7°
k_1	-0.25722 ± 0.00228
k_2	0.09022 ± 0.00276
k_3	-0.00060 ± 0.00020
p_1	0.00009 ± 0.00018
p_2	-0.01662 ± 0.00098
Pixel error	[0.30 0.28]

The calibration vector is thus $[k_1,k_2,p_1,p_2,k_3]$ = [-0.25722, 0.09022, -0.00060, 0.00009, -0.01662] and the focal length $f_l = [f_{lx},f_{ly}]$ can be calculated from the horizontal and vertical field of view using the following Equations:

$$f_{lx} = \frac{I_w}{2\tan(0.5\text{HFoV})}$$
$$f_{ly} = \frac{I_h}{2\tan(0.5\text{VFoV})}$$

where I_w is the image width and I_h is the image height. Therefore, the focal length is [602.05 602.71] pixels.

4.4.3 Impact of the Vision-based Tracking Algorithm

Using the provided formulas, the onboard control algorithm can keep track of the movement of the target UAV throughout a sequence of frames in the video stream. In this section, the impact of the efficiency and accuracy of the distance estimation on the vision-based algorithms discussed in Sections 3.3.3 and 3.4 will be discussed.

In order to fairly compare the HOG-based and the Haar-based Cascade Classifier vision algorithms, a video of the 3DR Solo was recorded from a static GoPro Hero4 Black camera. The video consists of the 3DR Solo moving through a predetermined path where the UAV's motion per frame is known. The predetermined motion can be seen in Figure 4.22. Each black dot in the figure represents 6 inches or 15.24 centimeters, which corresponds to the expected movement of the UAV per frame.

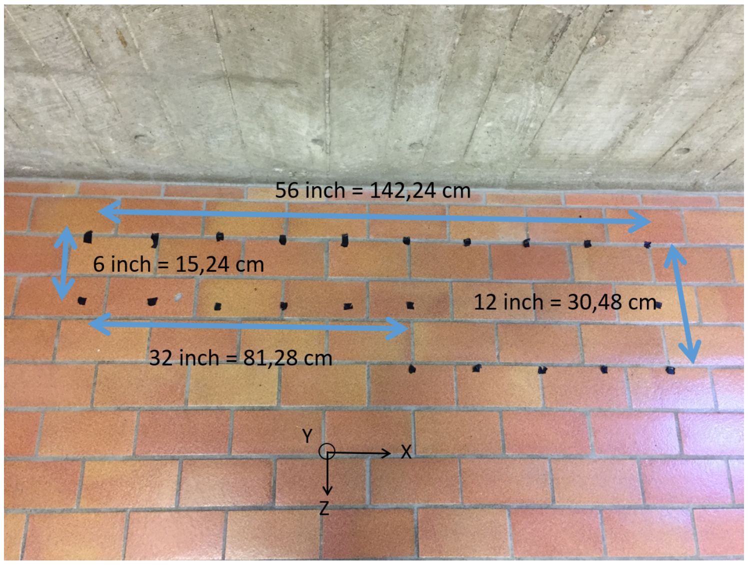


Figure 4.22: Predetermined Path for the Cascade Classifier Initial Test Diagram

Given that we know certain parameters of the motion of the UAV along all axes, we will employ these to evaluate the accuracy of the distance estimation for each Cascade Classifier. These known parameters are:

- The expected movement between frames (or linear speed) in each axis: 6 inches/frame or 15.24 cm/frame for the X and Z axis.
- The total motion along the three axes: 56 inches or 142.24 cm in the X axis back and forth and 12 inches or 30.48 cm back and forth. No movement is expected along the Y axis.

Recall that all distance estimations are calculated relative to the motion of the body-fixed frame of the target UAV viewed from a ground-fixed inertial frame. Figures 4.23 to 4.35 represent the distance estimation results [x,y,z] compared to the reference trajectory [x_{ref},y_{ref},z_{ref}] as well as errors along each axis [e_x,e_y,e_z] from the three Cascade Classifier types listed in Section 3.4.2.

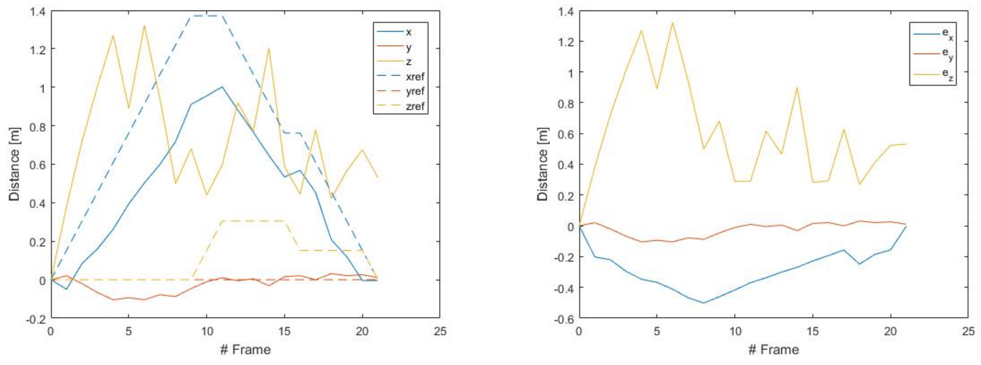


Figure 4.23: Distance Estimation Results for Cascade Classifier with Haar Features, 0.002 False Alarm Rate and 0.997 True Positive Rate

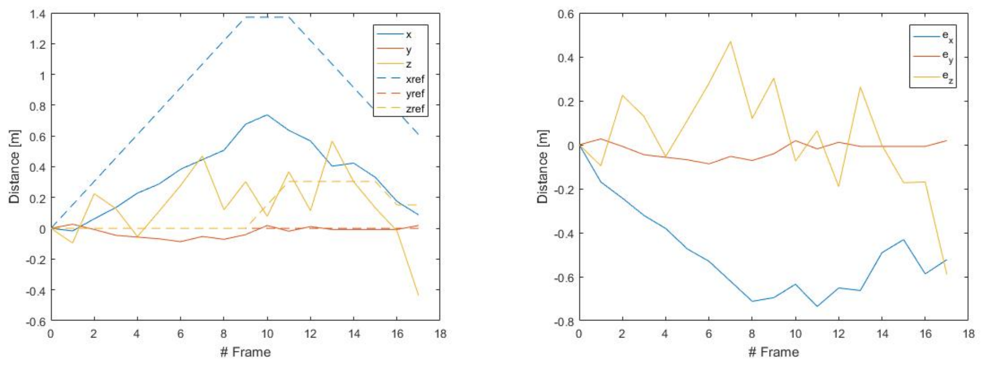


Figure 4.24: Distance Estimation Results for Cascade Classifier with Haar Features, 0.003 False Alarm Rate and 0.995 True Positive Rate

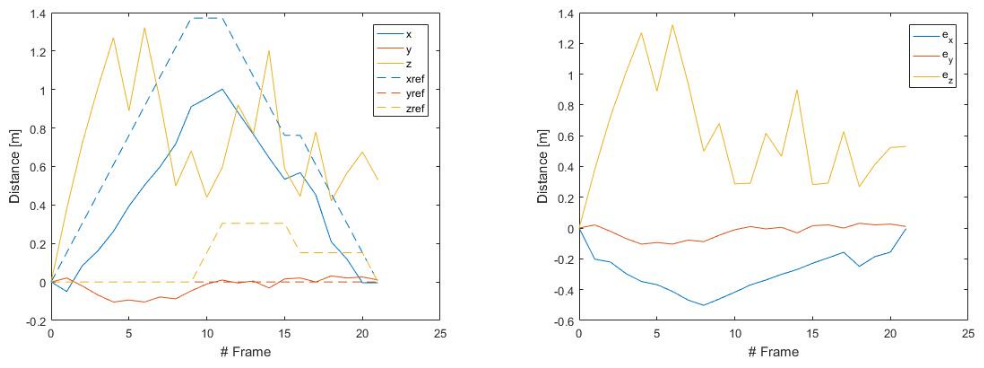


Figure 4.25: Distance Estimation Results for Cascade Classifier with Haar Features, 0.003 False Alarm Rate and 0.997 True Positive Rate

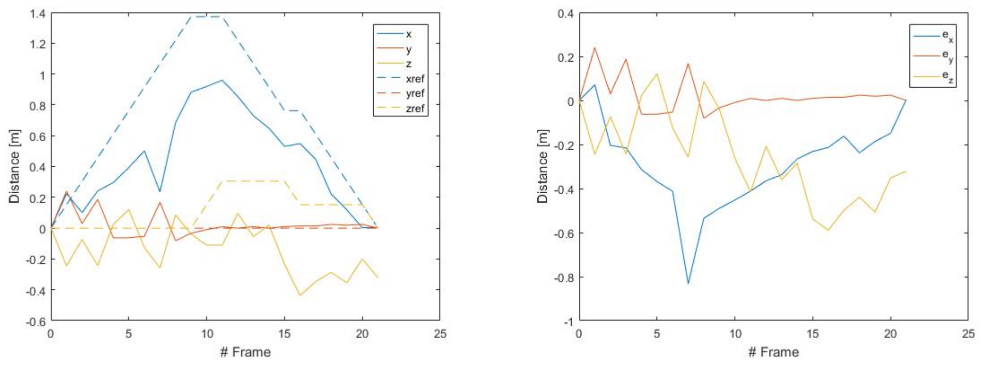


Figure 4.26: Distance Estimation Results for Cascade Classifier with Haar Features, 0.003 False Alarm Rate and 0.998 True Positive Rate

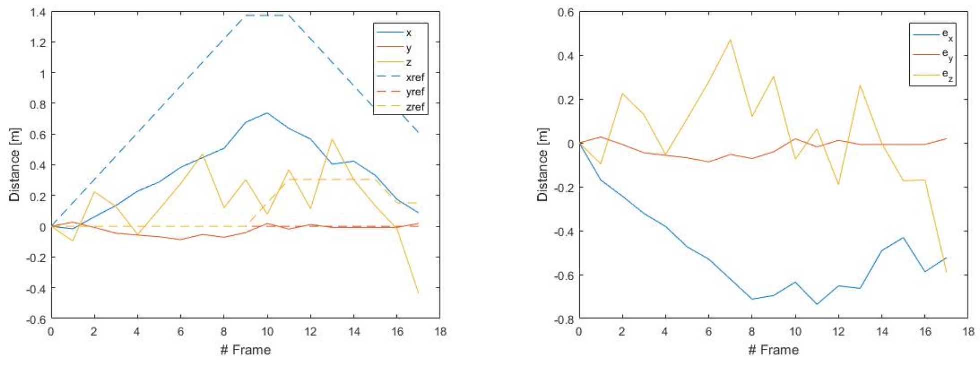


Figure 4.27: Distance Estimation Results for Cascade Classifier with Haar Features, 0.004 False Alarm Rate and 0.995 True Positive Rate

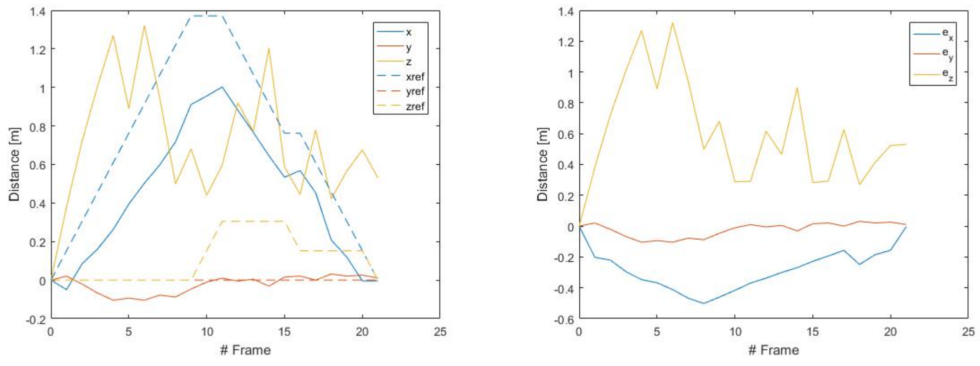


Figure 4.28: Distance Estimation Results for Cascade Classifier with Haar Features, 0.004 False Alarm Rate and 0.997 True Positive Rate

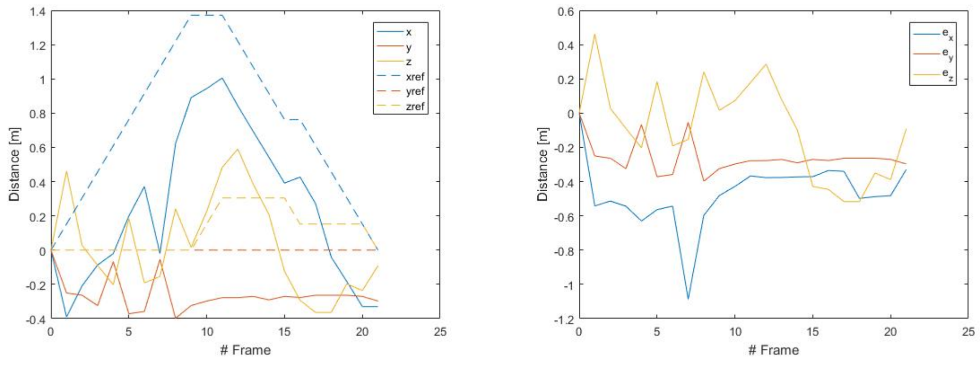


Figure 4.29: Distance Estimation Results for Cascade Classifier with Haar Features, 0.004 False Alarm Rate and 0.999 True Positive Rate

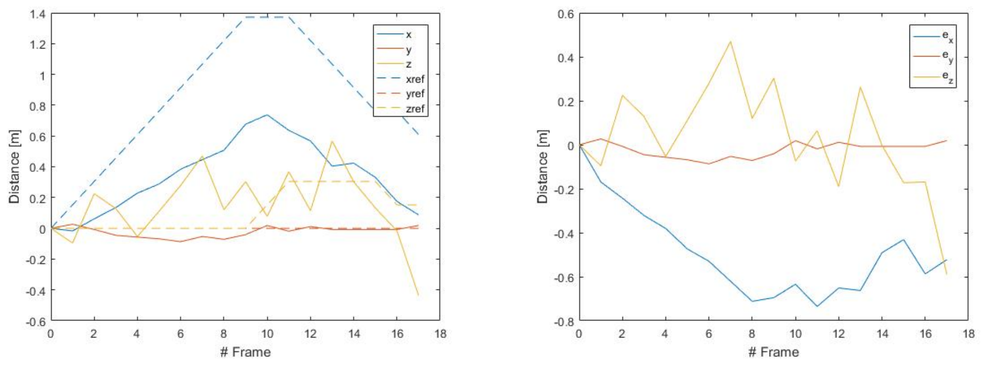


Figure 4.30: Distance Estimation Results for Cascade Classifier with Haar Features, 0.005 False Alarm Rate and 0.995 True Positive Rate

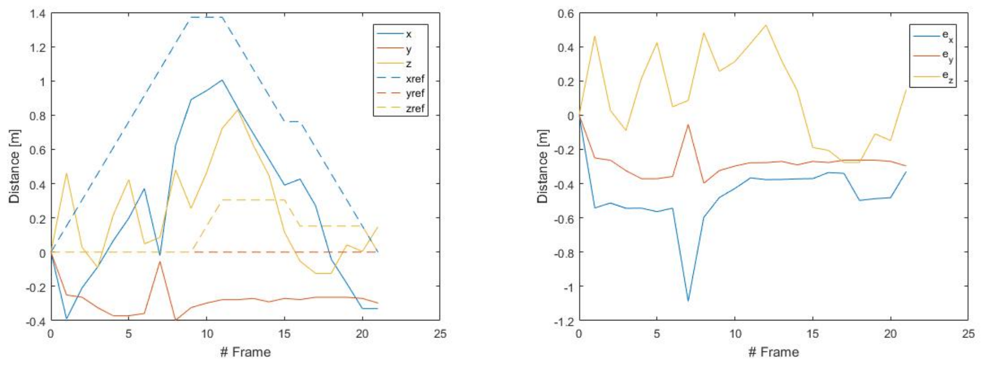


Figure 4.31: Distance Estimation Results for Cascade Classifier with Haar Features, 0.005 False Alarm Rate and 0.999 True Positive Rate

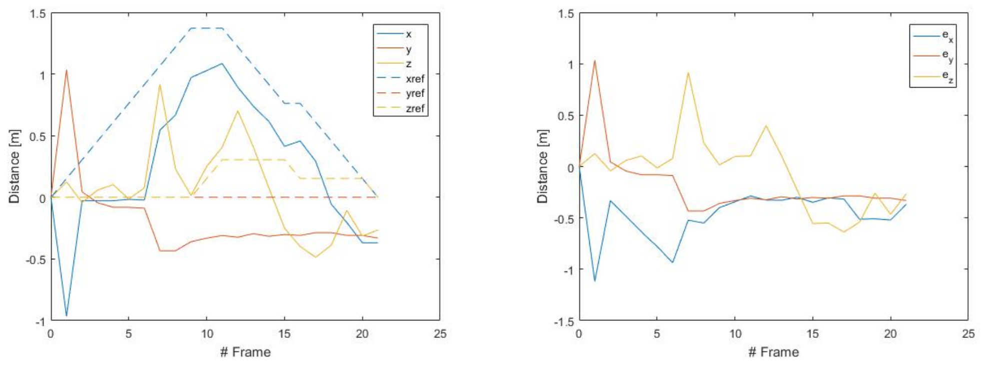


Figure 4.32: Distance Estimation Results for Cascade Classifier with Haar Features, 0.08 False Alarm Rate and 0.998 True Positive Rate

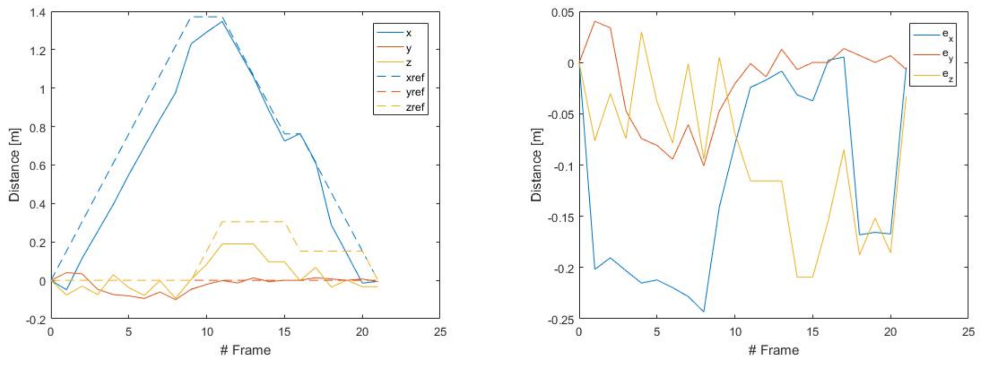


Figure 4.33: Distance Estimation Results for Cascade Classifier with HOG Features, 0.001 False Alarm Rate and 0.999 True Positive Rate

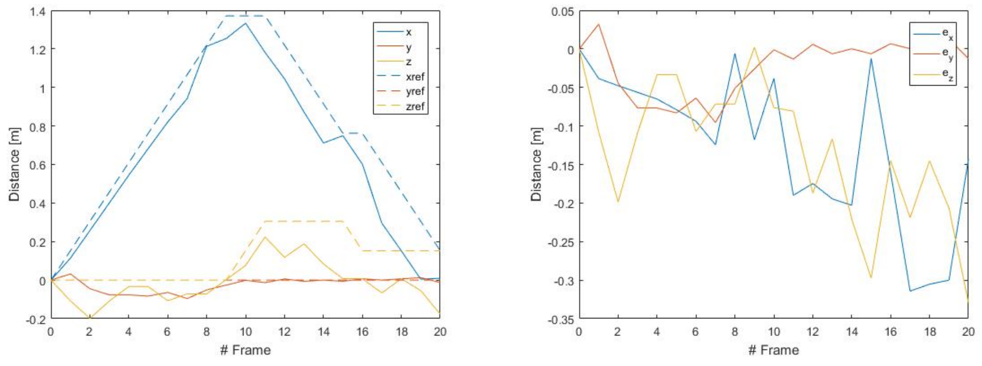


Figure 4.34: Distance Estimation Results for Cascade Classifier with HOG Features, 0.08 False Alarm Rate and 0.997 True Positive Rate

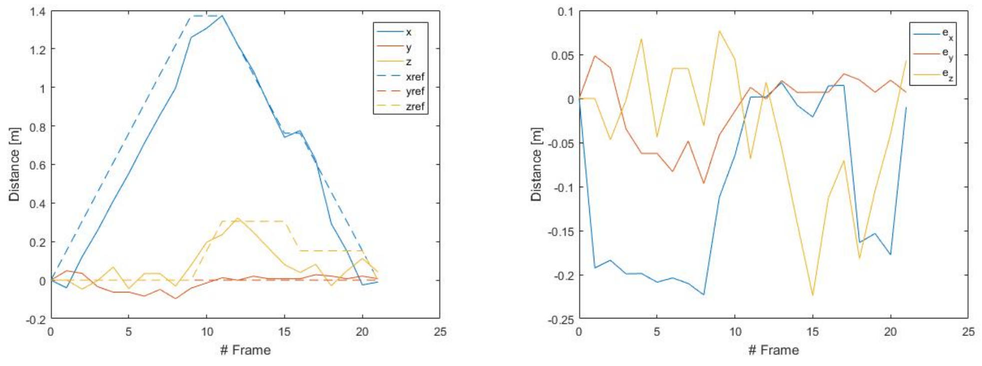


Figure 4.35: Distance Estimation Results for Cascade Classifier with HOG Features, 0.08 False Alarm Rate and 0.998 True Positive Rate

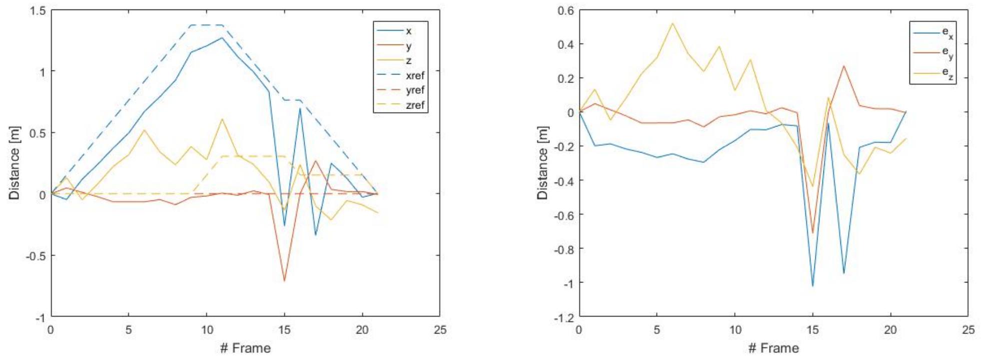


Figure 4.36: Distance Estimation Results for Cascade Classifier with HOG Features, 0.1 False Alarm Rate and 0.9 True Positive Rate

In order to compare the results, we performed a statistical analysis of the error signals (see Tables 4.2 and 4.3). Recall that FAR is the False Alarm Rate and TPR is the True Positive Rate of each cascade classifier.

Table 4.2: Statistical Study of the Distance Estimation Error Signals for the “Optimal” HOG Cascade Classifiers

Cascade Classifier	Axis	Mean [m]	Standard Deviation [m]	Variance [m]	Mean Squared Error [m^2]
HOG Features 0.001 FAR 0.999 TPR	x	-0.1160	0.0942	0.0089	0.0219
	y	-0.0200	0.0405	0.0016	0.0020
	z	-0.0905	0.0713	0.0051	0.0130
HOG Features 0.08 FAR 0.997 TPR	x	-0.1269	0.0982	0.0096	0.0253
	y	-0.0235	0.0370	0.0014	0.0019
	z	-0.1312	0.0909	0.0083	0.0251
HOG Features 0.08 FAR 0.998 TPR	x	-0.1036	0.0958	0.0092	0.0195
	y	-0.0102	0.0398	0.0016	0.0016
	z	-0.0367	0.0799	0.0064	0.0074
HOG Features 0.1 FAR 0.9 TPR	x	-0.2402	0.2561	0.0656	0.1203
	y	-0.0322	0.1669	0.0279	0.0276
	z	-0.0349	0.2596	0.0674	0.0656

Table 4.3: Statistical Study of the Distance Estimation Error Signals for the "Optimal" Haar Cascade Classifiers

Cascade Classifier	Axis	Mean [m]	Standard Deviation [m]	Variance [m]	Mean Squared Error [m^2]
Haar Features	x	-0.2790	0.1366	0.0187	0.0957
0.002 FAR	y	-0.0222	0.0463	0.0021	0.0025
0.997 TPR	z	0.6003	0.3399	0.1155	0.4707
Haar Features	x	-0.4923	0.2037	0.0415	0.2816
0.003 FAR	y	-0.0225	0.0345	0.0012	0.0016
0.995 TPR	z	0.0342	0.2421	0.0586	0.00565
Haar Features	x	-0.2790	0.1366	0.0187	0.0957
0.003 FAR	y	-0.0222	0.0463	0.0021	0.0025
0.997 TPR	z	0.6003	0.3399	0.1155	0.4707
Haar Features	x	-0.2868	0.1998	0.0399	0.1204
0.003 FAR	y	0.0203	0.0797	0.0064	0.0065
0.998 TPR	z	-0.2514	0.2070	0.0428	0.1041
Haar Features	x	-0.4923	0.2037	0.0415	0.2816
0.004 FAR	y	-0.0225	0.0345	0.0012	0.0016
0.995 TPR	z	0.0342	0.2421	0.0586	0.00565
Haar Features	x	-0.2790	0.1366	0.0187	0.0957
0.004 FAR	y	-0.0222	0.0463	0.0021	0.0025
0.997 TPR	z	0.6003	0.3399	0.1155	0.4707
Haar Features	x	-0.4666	0.1922	0.0369	0.2530
0.004 FAR	y	-0.2604	0.0979	0.0096	0.0769
0.999 TPR	z	-0.0875	0.2741	0.0751	0.0794
Haar Features	x	-0.4923	0.2037	0.0415	0.2816
0.005 FAR	y	-0.0225	0.0345	0.0012	0.0016
0.995 TPR	z	0.0342	0.2421	0.0586	0.00565
Haar Features	x	-0.4627	0.1895	0.0359	0.2484
0.005 FAR	y	-0.2742	0.0906	0.0082	0.0830
0.999 TPR	z	0.1167	0.2593	0.0672	0.0778
Haar Features	x	-0.4649	0.2405	0.0579	0.2713
0.08 FAR	y	-0.1885	0.3058	0.0935	0.1248
0.998 TPR	z	-0.0608	0.3623	0.1313	0.1290

Based on the test results, we can observe that:

- The Haar-based Cascade Classifier algorithms have, in general, big problems with getting a stable bounding box detection, making the depth or Z axis estimation incorrect (up to 100% error) as well as noisy. Since our distance estimation relies heavily on the accuracy of the initially detected bounding box, any error in these first detections will be accumulated during the entire experiment. This explains why the shape of the distance estimation in both X and Y axis is qualitatively correct, but the actual values are incorrect (up to 15% error).

- Given that the distance estimation will serve as reference for the quadcopter's position control system, the signal must have as little noise as possible. Although the Haar-based Cascade Classifiers were shown to be about five times faster than the HOG-based ones, in addition to having a good detection accuracy, due to both the distance estimation errors along each of the axes and the exhibited noise we consider that the Haar-based Cascade Classifiers cannot be used for experimental flight testing.

- Meanwhile, the HOG-based Cascade Classifier algorithms exhibit a distance estimation error roughly one order of magnitude less than the Haar-based Cascade Classifiers. This is due to having a more stable bounding box throughout the experiment, leading to more stable and accurate estimations. Therefore, the use HOG-based Cascade Classifiers for UAV detection and tracking is advised.

Out of all the HOG-based Cascade Classifiers, and focusing on the detection error along the Z axis, we pick the 0.08 false alarm rate and 0.998 true positive rate as the optimal parameters for detection and tracking of quadcopters.

The best-performing Cascade Classifier from the above analysis will be retested to evaluate the impact of the background on distance estimation in a more realistic flight experiment. This consists of the drone taking off to 3 meters, moving left 2.25 m and then landing. As before, a video of the 3DR Solo was recorded from a static GoPro Hero4 Black camera. The distances were pre-measured in order to serve as a ground truth for testing the accuracy of the tracking algorithm. The drone takes off from an elevated plastic platform in order to eliminate magnetic field distortions due to ferromagnetic materials in the ground.

The upcoming figures will illustrate the results of running two vision-based algorithm designs in Matlab. We will analyze the raw output data of the optimal HOG-based Cascade Classifier and compare it with the output of the HOG-FAST with Kalman Filter algorithm seen in Section 3.3.4. Based on Chapter 3, some differences can be highlighted in the behavior of the two algorithms:

- The HOG-FAST with Kalman Filter algorithm needs a set of initialization frames, thus the tracking of the target UAV will be delayed compared to the Cascade Classifier.

- The HOG-FAST with Kalman Filter algorithm depends on motion to be able to detect the target UAV, i.e. a stable hover or a stationary UAV on the ground would reset the algorithm, and require a set of frames with movement to reinitialize it. Meanwhile, the Cascade Classifier detection is not affected by the motion of the target UAV.

By choosing to use HOG instead of Haar features for the detection Cascade Classifier, we have selected the more computationally demanding algorithm. In order to improve performance, we will analyze the raw data output from the Cascade Classifier to see if the amount of data obtained could be reduced. This is further discussed in Section 4.5.

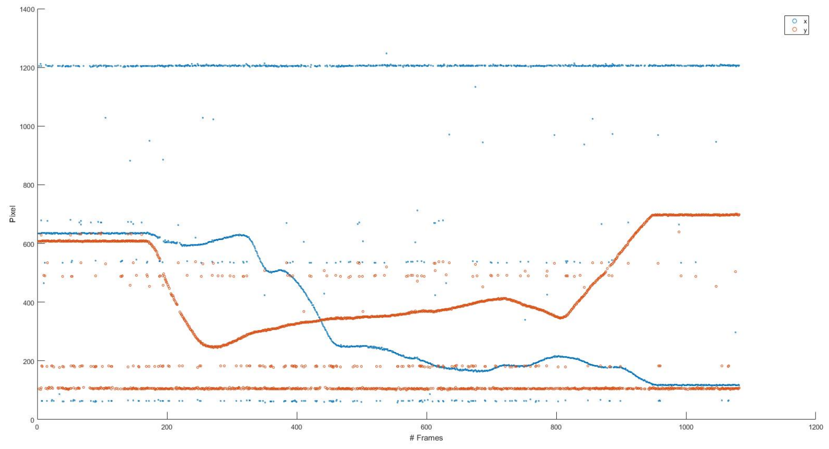


Figure 4.37: Pixel Coordinates of the Target UAV Centroid

Figure 4.37 shows the trajectory of position coordinates $[x_n^p, y_n^p]$ during the flight experiment. Since the camera is static, and thus the background does not move, the curve represents the motion of the target UAV, while the other horizontal sets of points are false positives of the Cascade Classifier algorithm for this particular experiment. The presence of false positives is typical when using the Cascade Classifier algorithm without filtering.

In the present case, the output noise is simple to filter out. The Cascade Classifier output data is an array containing a set of bounding boxes which describe all target UAV matches identified in each frame. This array is ordered from best to worst match, with all results above the minimum boundary specified by the algorithm. Thus, by assuming a single target UAV, we retain only the best match at each frame, and discard the remaining results as noise.

Since we expect some UAV motions to be lost during the initialization phase of the HOG-FAST with Kalman Filter algorithm, in order to compare the results of both algorithms for distance estimation accuracy we will manually set the first non-zero value of each axis distance estimation to equalize the initial point of both algorithms.

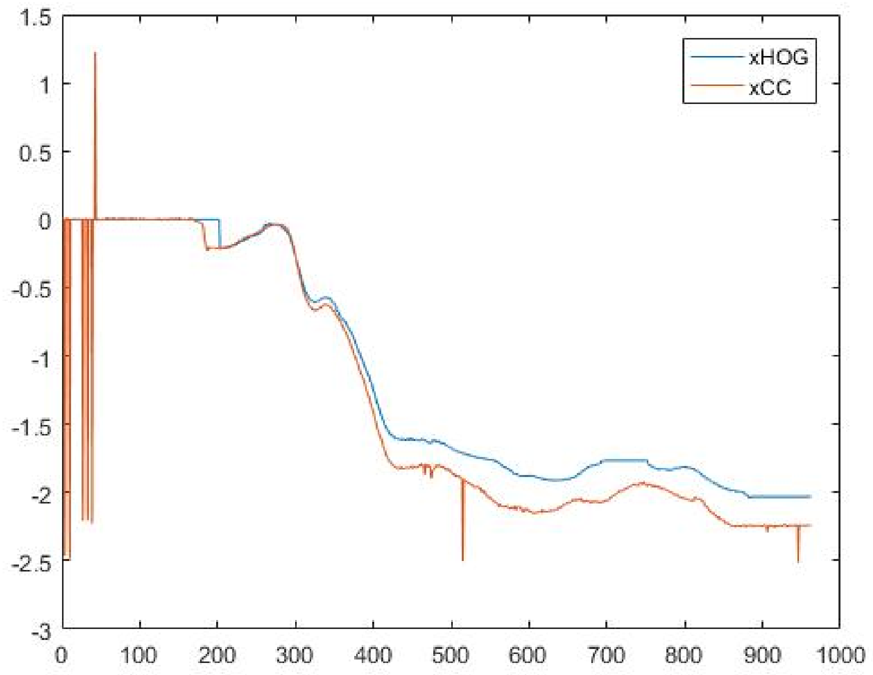


Figure 4.38: Distance Estimation for HOG-FAST with Kalman Filter and Cascade Classifier along the X Axis

Figure 4.38 shows both the HOG-FAST with Kalman Filter and Cascade Classifier distance estimations along the X axis. We see that there is no distance estimation for the HOG-FAST with Kalman Filter algorithm during the first 243 frames nor for the last 118, which together accounts for a substantial 33.36% of the entire experiment. As explained above, the cause of this is the lack of motion in the pre-take off and post-landing regimes. Meanwhile, the Cascade Classifier can detect the target UAV throughout the entire length of the experiment. We can also observe that the Cascade Classifier outputs motion earlier than the HOG-FAST with Kalman Filter, due to the initialization period required by the latter. By retaining only the best match, the noise in the Cascade Classifier estimates is drastically reduced as compared to Figure 4.37, although not completely eliminated. The residual noise is caused by frames in which the target UAV was considered a false positive or no match was found.

The results of both algorithms are empirically correct, with both trajectories tending to x = -2.3 m as expected. However, the accuracy of the two algorithms is different, with the HOG-FAST with Kalman Filter algorithm ending at x= -2.031 m and the Cascade Classifier ending at x= -2.243 m, representing 11.69% and 2.48% relative error, respectively.

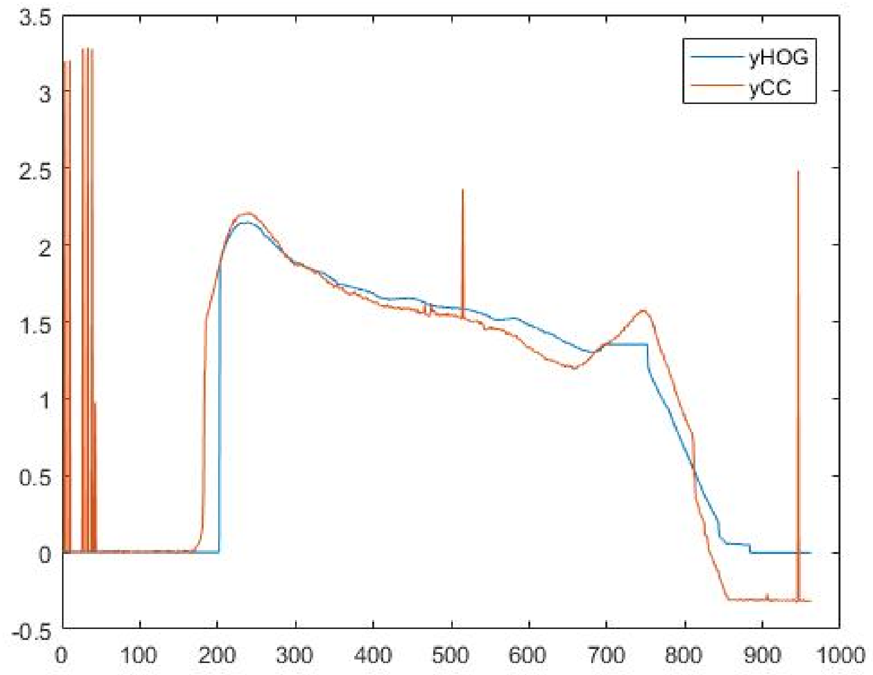


Figure 4.39: Distance Estimation for HOG-FAST with Kalman Filter and Cascade Classifier along the Y Axis

Figure 4.39 shows the HOG-FAST with Kalman Filter and Cascade Classifier distance estimations along the Y axis. We observe the same qualitative behaviors for the HOG and Cascade Classifier algorithms as for the X axis. The result of both algorithms are empirically correct, with both trajectories first tending to y = 3 - 0.65 = 2.35 m, remaining stable as the UAV moves along the X axis, and finally tending to y = -0.65 m.

The altitude loss observed in this experiment is due to hardware and/or software issues in the 3DR Solo. Based on extensive flight testing, we have observed that the UAV loses altitude over the first 10 seconds of hovering, while reporting the measured altitude as constant. This error stabilizes once the error reaches 1.1 meters. The cause of this effect may be due to a low-cost GPS receiver, lack of an accurate altimeter, or a poorly tuned state estimation system onboard the UAV. Unfortunately, this means the altitude reporting during hovering cannot be trusted.

Nevertheless, the results can still be evaluated in the takeoff and landing phases. The HOG-FAST with Kalman Filter algorithm ends up at y=2.146 m and y=-0.004 m or 8.68% and 34.0% relative error respectively. The Cascade Classifier algorithm ends up at y=2.212 m and y=-0.3122 m or 5.48% and 17.77% relative error respectively. We can conclude that for the X and Y axes, the Cascade Classifier is more accurate than the HOG-FAST with Kalman Filter algorithm, in addition to being faster and more reliable for detection tasks.

For depth estimation, the previous approach cannot be used. As shown in Figure 4.40, noise greatly affects the estimated depth values, making it impossible to accurately measure its values.

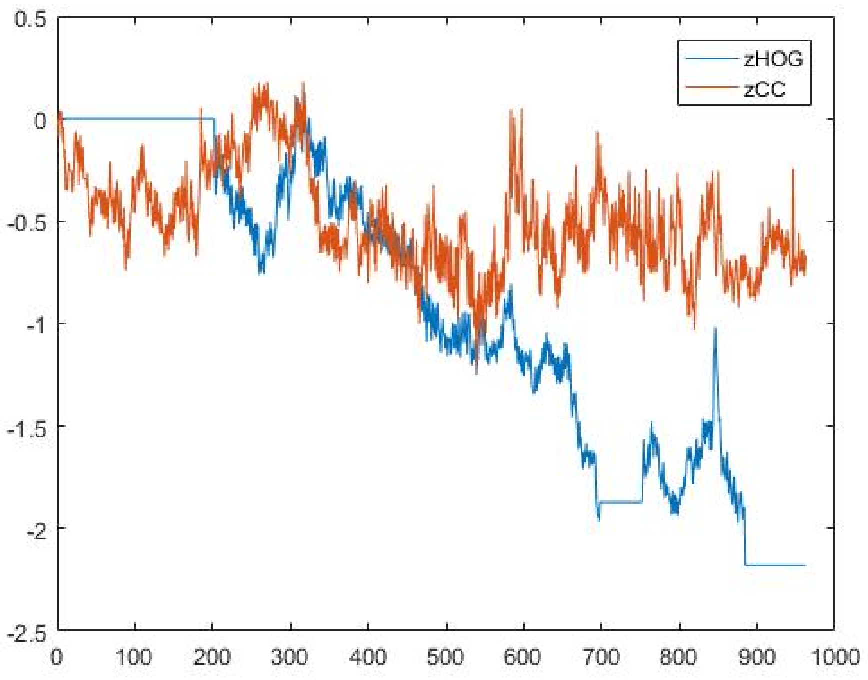


Figure 4.40: Distance Estimation for HOG-FAST with Kalman Filter with Kalman and Cascade Classifier in the Z Axis

Depth estimation is very sensitive to bounding box variations and possible background noise. In the present flight experiment, the true depth of the target UAV is close to constant. While the HOG-FAST with Kalman Filter depth estimation diverges, the Cascade Classifier depth estimation has variable but bounded error and does not diverge. Nevertheless, the depth estimation obtained as-is is unusable as a reference trajectory for any control system. We thus need to find a way to get a more consistent and stable depth estimation, either by stabilizing the bounding box size or by reducing the amount of background noise.

Whereas the stability of the bounding box output of the vision-based algorithm is of utmost importance for depth estimation, it is not the only factor which limits depth tracking. The bounding box size is limited by the Cascade Classifier to a minimum and a maximum, meaning the depth estimation has an effective working range inside which the depth error will be bounded, but this error will increase indefinitely outside this range. In the present case, the minimum bounding box size is 60x60 pixels while the maximum is the size of the video frame, here 1280x720 pixels. Given an initial bounding box size, we can determine the effective range for the depth estimation, shown in Figure 4.41.

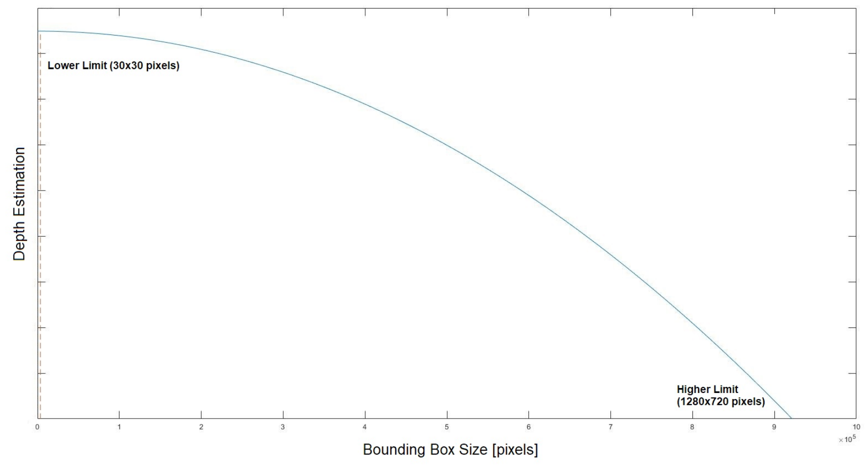


Figure 4.41: Cascade Classifier Depth Estimation Range

Given the quadratic relationship between the depth estimation and the bounding box size, the error beyond the lower limit is not negligible. The detection error will increase considerably once the lower limit is reached. Moreover, note that if the first detection gives a bounding box of minimum or maximum size, then the depth estimation will be limited to negative (target getting closer) or positive (target getting further) depth respectively.

4.5 Image Reduction

Thus far, when running the vision-based algorithms of Chapter 3, the full frame was processed in order to detect the target UAV. However, this approach is computationally expensive, especially when using high-resolution images. For target tracking, we know that the sought features are located close to the estimated target location. In addition, features risk being affected by background noise, which leads to false positive identifications and may degrade the performance of the mimicking system. Therefore, if the target's position can be predicted beforehand, an area of interest in the video frame can be defined in which the target UAV is likely to be located. The vision-based algorithm could then focus on this specific area of the image rather than the full frame, reducing computational time and reducing the effect of background noise on false positive identifications.

Because the target UAVs position needs to be available in real time, a one-step ahead predictor is needed. Among the various possible approaches, a Kalman filter can be used. Kalman filters are recursive state estimators which employ a dynamics model of the system. The state estimation is causal, i.e. uses only the current measurements as well as the estimated state propagated from the previous time-step. In contrast to other Bayesian estimation techniques, no history of observations or estimates is needed, making Kalman filters well suited for environments where computational power is limited [23]. We will be focusing our attention on the Discrete linear Kalman Filter (DKF) and the Unscented Kalman Filter (UKF) algorithms [24].

4.5.1 Target Motion Model

Both filters use a system dynamics model in order to predict the target's motion. The target UAV's flight is governed by Newton's laws of motion, meaning that its trajectory cannot change instantaneously. Based on observations of real UAV flight, where the stick inputs lead to aerodynamic force generation on the vehicle, we assume that the target motion in the plane can be modeled as having a constant jerk, meaning that acceleration is a slowly varying signal. In discrete time, given a pair of pixel coordinates $r_k = [x_p,y_p]$ within the kth frame, the target's motion dynamics are thus governed by:

$$\begin{aligned} r_{k+1} &= r_k + \Delta T v_k + \frac{\Delta T^2}{2} a_k + \frac{\Delta T^3}{6} j_k + w_r \\ v_{k+1} &= v_k + \Delta T a_k + \frac{\Delta T^2}{2} j_k + w_v \\ a_{k+1} &= a_k + \Delta T j_k + w_a \\ j_{k+1} &= j_k + w_j \end{aligned} \tag{4.8}$$

where v_k, a_k and j_k are the planar velocity, acceleration and jerk at the k^{th} frame, respectively, ΔT is the period of time between frames and $w_k = [w_r, w_v, w_a, w_j]^T$ is the vector of position, velocity, acceleration and jerk noise, respectively. The previous equations are a discrete state-space system with zero deterministic inputs, process noise w_k and $x_k = [r_k, v_k, a_k, j_k]$ as the state vector. This is written as

$$\begin{aligned} X_{k+1} &= AX_k + w_k \\ Y_{k+1} &= CX_{k+1} + n_{k+1} \end{aligned} \tag{4.9}$$

where the process noise, w_k, is assumed to be zero-mean Gaussian with known covariance $Q = E\langle w_k w_k^T \rangle$, and n_k is the measurement noise, assumed to be zero-mean Gaussian with known covariance $R = E\langle n_k n_k^T \rangle$. We assume both noise vectors are uncorrelated. The output of the system is the planar position of the target, r_k. The matrices A and C in (4.9) are thus given as (4.10). Note that both matrices are time-invariant [56].

$$A = \begin{bmatrix} 1 & 0 & \Delta T & 0 & \frac{\Delta T^2}{2} & 0 & \frac{\Delta T^3}{6} & 0 \\ 0 & 1 & 0 & \Delta T & 0 & \frac{\Delta T^2}{2} & 0 & \frac{\Delta T^3}{6} \\ 0 & 0 & 1 & 0 & \Delta T & 0 & \frac{\Delta T^2}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta T & 0 & \frac{\Delta T^2}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & \Delta T & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \Delta T \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \; C \; = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.10}$$

Note that ΔT can be associated to the inverse of the framerate. Therefore, all its expected possible values will be small and even smaller for its squared and cubic values.

4.5.2 Discrete-time Kalman Filter (DKF)

To predict the position of the target UAV in the frame, r_k, a discrete-time Kalman filter is designed. For a linear system, a Kalman filter is an optimal estimator provided all noise is normally distributed, i.e. the filter estimates the nominal state from indirect and uncertain outputs. The DKF minimizes the steady-state error covariance matrix P as Equation (4.11):

$$P = \lim_{t\to\infty} E[(X - \hat{X})(X - \hat{X})^T] \tag{4.11}$$

Kalman filtering is a convenient form of recursive estimation for real-time processing. Kalman filters are usually computed in two steps: first, an *a priori* estimation is calculated from the previous data; then, the computed estimation is corrected by the actual measurement. Given the system model in the previous section, the *a priori* $\hat{X}_k^-$ and *a posteriori* $\hat{X}_k^+$ estimated solutions of the DKF are given by

$$\begin{aligned} \hat{X}_{k+1}^- &= A\hat{X}_k^+ \\ \hat{X}_{k+1}^+ &= C\hat{X}_{k+1}^- + K_k(y_k - C\hat{X}_{k+1}^-) \end{aligned} \tag{4.12}$$

where K_k is the filter gain, determined by solving the algebraic Riccati equation. The results are given as

$$\begin{aligned} K_k &= P_k^- C^T (CP_k^- C^T + R)^{-1} \\ \text{where,}\quad P_k^- &= AP_(k-1)^+ A^T + Q \\ P_k^+ &= (I - K_k C)P_k^- \end{aligned} \tag{4.13}$$

The DKF design will be implemented as the block diagram shown Figure 4.42.

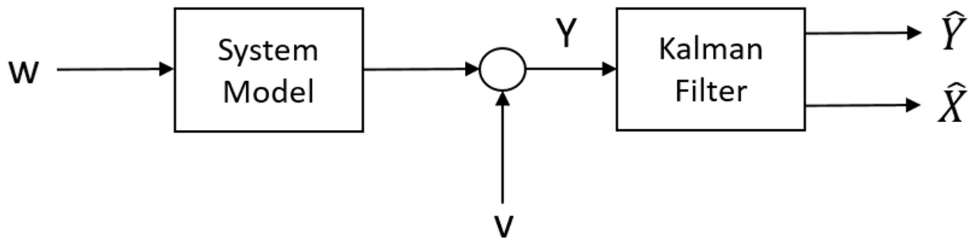


Figure 4.42: Discrete Kalman Estimator Schematic

In the present model, the system can be verified to be uncontrollable (as the input matrix B is zero) but observable. The conditions for convergence of the Kalman filter are the observability of the system model and that matrix M defined in Equation (4.14), must be positive semi-definite.

$$\mathrm{M} = \begin{bmatrix} \mathrm{Q} & \mathrm{N} \\ \mathrm{N}^T & \mathrm{R} \end{bmatrix} \tag{4.14}$$

where $N = E\langle w_k v_k^T \rangle = 0$. The eigenvalues of $M^* = (M + M^T)$ define the positiveness of M. Let $\lambda = [\lambda_1, ..., \lambda_9]$ the set of eigenvalues of M^*. Given the system model in Section 4.5.1, $Q \in \mathbb{R}^8$ and $R \in \mathbb{R}$. Therefore, λ is composed of the diagonal values of Q and R. If Q ≥ 0 and R ≥ 0, then the associated discrete Kalman filter will be stable. Note that if M^* is positive semi-definite, the filter convergence may have some error in the steady-state.

4.5.3 Unscented Kalman Filter (UKF)

In cases where the system model is highly nonlinear, the UKF tends to perform better than the regular linearization-based Extended Kalman Filter (EKF). Even if the system model is linear, the UKF uses a deterministic sampling method known as the "unscented transform" which results in a more accurate estimation of the true mean and covariance of the model [22].

Just like the DKF, the UKF is implemented in two steps: a predictive step and a correction step. The predictive step needs the state and covariance matrix to be augmented with the mean and covariance of the process noise, as follows:

$$\begin{aligned} X_k^a &= [X_k^T E(w_k^T)]^T \\ P_k^a &= \begin{bmatrix} P_k & 0 \\ 0 & \mathrm{Q} \end{bmatrix} \end{aligned} \tag{4.15}$$

A set of 2N$_a$+1 sigma points, σ_k^i, is derived from the augmented state and covariance matrix where N$_a$ is the dimension of the augmented state. All the sigma points are propagated through the state function (4.9) and then recombined to produce the predicted state and covariance by Equation (4.16):

$$\begin{aligned} \hat{X}_{k+1}^- &= \sum_{i=0}^{2N_a} W_s^i \sigma_k^i \\ P_{k+1}^- &= \sum_{i=0}^{2N_a} W_c^i [\sigma_k^i - \hat{X}_{k+1}^-][\sigma_k^i - \hat{X}_{k+1}^-]^T \end{aligned} \tag{4.16}$$

where W_s and W_c are the weights for the state and covariance, respectively, defined as Equation (4.17):

$$\begin{aligned} W_s^0 &= \frac{\alpha^2(N_a+\kappa) - N_a}{\alpha^2(N_a+\kappa)} \\ W_c^0 &= W_s^0 + 1 - \alpha^2 + \beta \\ W_s^i = W_c^i &= \frac{1}{2\alpha^2(N_a+\kappa)} \quad , i = 1, ..., 2N_a \end{aligned} \tag{4.17}$$

where α, β and κ are parameters of the UKF. The values used are $\alpha = 0.001$, $\beta = 2$ and $\kappa = 0$. If the distribution of the states is Gaussian, the estimation will be optimal under those parameters [4].

The correction step uses an augmented state and covariance matrix, exactly as in the prediction step, except that the measurement noise mean and covariance are used:

$$\begin{aligned} X_k^a &= [X_k^T E(v_k^T)]^T \\ P_k^a &= \begin{bmatrix} P_k & 0 \\ 0 & \mathrm{R} \end{bmatrix} \end{aligned} \tag{4.18}$$

As before, a set of 2N$_a$+1 sigma points, γ_k^i, is derived from the augmented state and covariance matrix where N$_a$ is the dimension of the augmented state. All the sigma points are propagated through the measurement function (4.9) and then recombined to produce the predicted state and covariance, Equation (4.19):

$$\begin{aligned}\hat{X}_{k+1}^{+} &= \hat{X}_{k+1}^{-} + K_k(y_k - \sum_{i=0}^{2N_a} W_s^i \gamma_k^i)\\ P_{k+1}^{+} &= P_{k+1}^{-} - K_k P_{yk} K_k^T\end{aligned} \tag{4.19}$$

where K_k is the UKF Kalman gain, computed as the product of the state-measurement cross-covariance matrix, P_{xy}, and the inverse of the updated measurement covariance matrix P_y:

$$\begin{aligned}K_k &= P_{xyk} P_{yk}^{-1}\\ P_{xyk} &= \sum_{i=0}^{2N_a} W_c^i [\sigma_k^i - \hat{X}_{k+1}^{-}][\gamma_k^i - \sum_{i=0}^{2N_a} W_s^i \gamma_k^i]^T\\ P_{yk} &= \sum_{i=0}^{2N_a} W_c^i [\gamma_k^i - \sum_{i=0}^{2N_a} W_s^i \gamma_k^i][\gamma_k^i - \sum_{i=0}^{2N_a} W_s^i \gamma_k^i]^T\end{aligned} \tag{4.20}$$

4.5.4 Predicted Motion Boundary Area

Given the *a posteriori* estimation for either of the Kalman filter designs, $\hat{r}_{k+1}^{-}$, we can define a geometrical zone χ in which the probability of finding r_{k+1} is maximized, as shown in Figure 4.43.

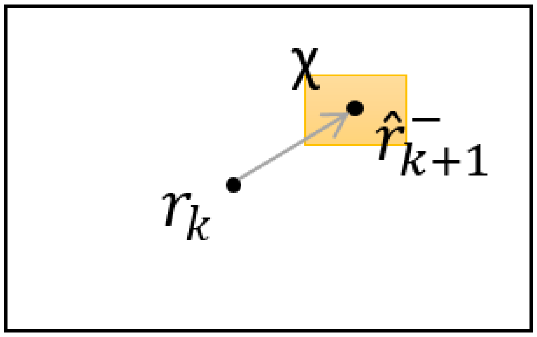


Figure 4.43: Predicted Motion Boundary Area Representation

The area will be defined as a bounding box around $\hat{r}_{k+1}^{-}$. The width and height of χ must be inline with the predicted motion of the target during the time elapsed between frames, as expressed in Equation (4.8). In that time, the target's acceleration can linearly change from its maximum positive value to its minimum negative value along both axes. Assuming the maximum possible change in acceleration, we can define upper and lower limits of the motion change in between frames for the velocity and position of the target. Recall that $\hat{r}_k = [\hat{x}_k, \hat{y}_k]$. Given the Kalman filter prediction step, Equations (4.12) and (4.16), the area χ can be defined by the following inequalities:

$$\begin{aligned} \hat{x}_{k+1}^{-} - \Delta x - \Delta v_x \leq x_{k+1} \leq \hat{x}_{k+1}^{-} + \Delta x + \Delta v_x \\ \hat{y}_{k+1}^{-} - \Delta y - \Delta v_y \leq y_{k+1} \leq \hat{y}_{k+1}^{-} + \Delta y + \Delta v_y \end{aligned} \tag{4.21}$$

Therefore, the area χ will be centered around $\hat{r}_{k+1}^{-}$ with $\Delta x + \Delta v_x$ and $\Delta y + \Delta v_y$ as the bounding box width and height, respectively, where Δx and Δy represent the previously corrected error for both pixel coordinates (see Equation (4.22)) and Δv_x and Δv_y represent the maximum variation of the velocity between frames along both frame axes. Note that Equations (4.21) are in pixels, therefore only integer values can be obtained from them. The area χ will thus increase with the Kalman filter error and decrease with the frame rate.

$$\begin{aligned} \Delta x &= |\hat{x}_k^{+} - x_k| \\ \Delta y &= |\hat{y}_k^{+} - y_k| \end{aligned} \tag{4.22}$$

The value of Δv_x and Δv_y varies from one target to another as the maximum achievable velocity, v_{max}, varies. Given a certain target, the maximum variation of the velocity can be computed as:

$$\begin{aligned} \Delta v_x &= \frac{v_{max}^{x}}{2 f x_{UAV}} \\ \Delta v_y &= \frac{v_{max}^{y}}{2 f y_{UAV}} \end{aligned} \tag{4.23}$$

where f is the video sequence frame rate in frames per second and x_{UAV} and y_{UAV} represent the cm/pixel ratio in the frame and are considered constant throughout the video stream. These are obtained at the first detection by the vision-based algorithm as discussed in Section 4.4. For a more pragmatic approach, to get a squared bounding box, set the width and height as the maximum of $\Delta x + \Delta v_x$ and $\Delta y + \Delta v_y$.

In most cases, the χ area width and height depends on the value of Δv, since this tends to be greater than the error of the Kalman filter. Recall that it is inversely proportional to the frame rate and the cm/pixel ratio of the frame. Whereas the target gets smaller as it moves away from the camera, the vision-based algorithm outputs a bounding box around the target centroid which has a minimal size of 60x60 pixels. Thus, even at higher frame rates and with a large cm/pixel ratio, the χ area should have a width and height larger than the minimum achievable output of the vision-based algorithm, otherwise it will fail to detect the target. It is thus advisable to limit the minimum size of the reduced image.

4.6 Image Reduction Implementation

To test the performance of the DKF and UKF designed previously, we will use the same experimental test as in Section 4.4.3. A 1280x720 video stream of a flying quadcopter is recorded in a built-up area at 30 frames per second. The video is then processed by the optimized HOG-based Cascade Classifier with Kalman filtering and image reduction. The tracking results and computational times were obtained through Monte Carlo simulation using a total of 15416 simulations. The tracking results can be seen in Figure 4.44, with its outliers highlighted. Over the length of the video, the vision-based algorithm gives consistent results at all frames except for seven, at which the algorithm gave false positive results due to background noise.

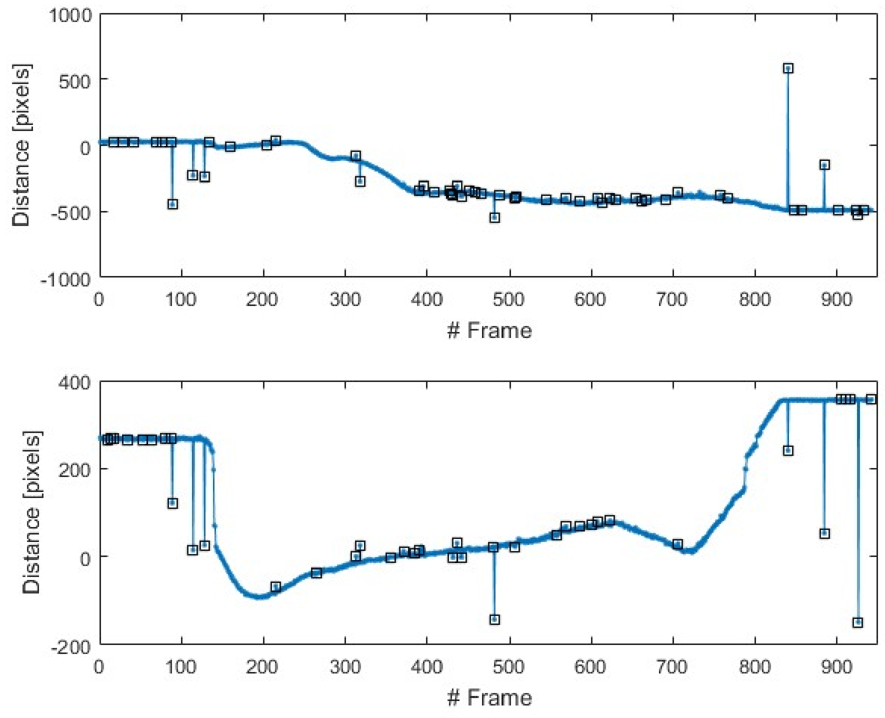


Figure 4.44: HOG-based Cascade Classifier Tracking Results and Outlier Detection in X and Y Axes Respectively

For this experimental test, the process and measurement noise covariances were determined following the conditions of the experiment and the vision-based algorithm used.

We assume the process noise affects mostly the jerk while the acceleration, velocity and position remain unaffected. The process noise covariance matrix is Q = diag(Q_x, Q_y, Q_{vx}, Q_{vy}, Q_{ax}, Q_{ay}, Q_{jx}, Q_{jy}), where (Q_x,...,Q_{jy}) are the variances of the process noise for the position, velocity, acceleration and jerk in the x and y axes, respectively. We assign $Q_{ax} = Q_{ay} = Q_{vx} = Q_{vy} = Q_{rx} = Q_{ry} = 0$. For the specific quadcopter used for the flight experiment, the variance for the jerk is set to $Q_x = Q_y = 0.1$ [56].

The measurement noise will represent the assumed error induced when the 3D motion of the target is transformed into a 2D motion on a video frame. The measurement noise can be compared to the quantization error of any digital image processing as seen in Figure 4.45. In all cases, the maximum error possible is half a pixel along both axes, therefore we will take R = 0.25.

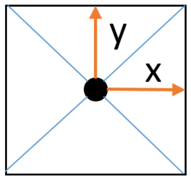


Figure 4.45: Pixel Quantization Error Diagram

Figures 4.46 and 4.47 show the results obtained using a DKF and UKF, respectively, for the vision-based algorithm's tracking results along both axes. The resulting prediction and corresponding error (with the upper and lower bounds) are also shown.

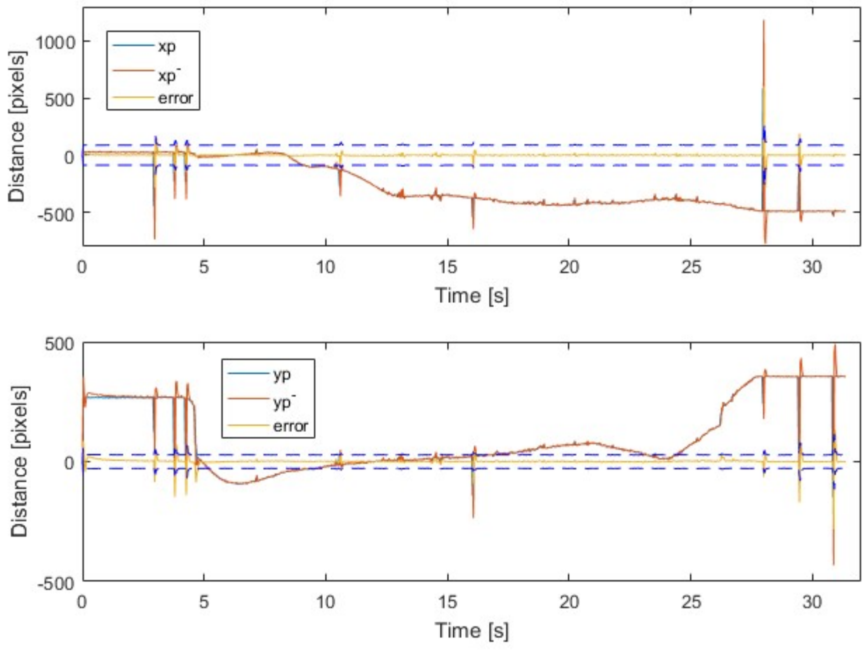


Figure 4.46: Average Discrete Kalman Filter Results

The results in Figure 4.46 show that the DKF prediction error is correctly bounded, except for the seven false positive frames and the frames during which the DKF converges. The results show a negligible drop of 1.38% in accuracy of position tracking with this approach. Meanwhile, in terms of image processing costs, the number of pixels to be processed per frame decreased from 921600 to 8281 pixels, a 99.10% reduction.

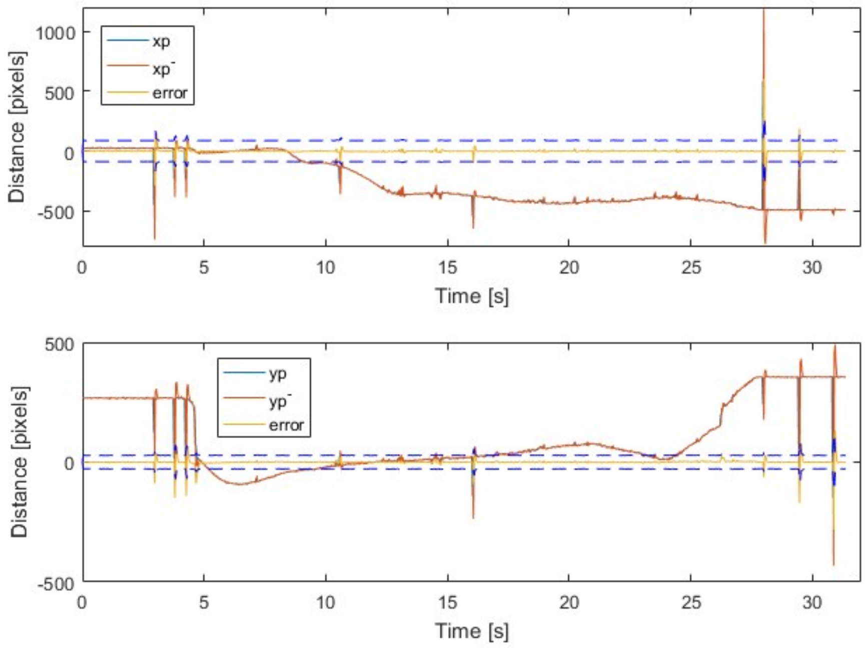


Figure 4.47: Average Unscented Kalman Filter Results

The results in Figure 4.47 show that the UKF prediction error is bounded, except for the same seven false positive frames as in the DKF. However, this version of the Kalman filter does not require a period of initialization to achieve convergence. The results show a drop of 1.06% in the accuracy of the position tracking. Meanwhile, the number of pixels to be processed per frame decreases from 921600 to 6561 pixels, a 99.29% reduction.

Regarding computational time, both the DKF and UKF offer very similar results, with a difference of approximately 0.1%. On average over the 15416 simulations, processing the whole image takes 0.1113 seconds to be processed. Under image reduction, the average computational time reduced to 0.0013 seconds, a reduction of two orders of magnitude. In other words, we can save up to 98.83% of the computational time to process the image. We also save, on average, 6.98 MB of memory, which is important for implementation in a resource-limited computer equipped onboard a typical UAV.

Finally, we fuse the image reduction method with the vision-based detection algorithm. The results for the centroid pixel coordinates can be seen in Figures 4.48 and 4.49.

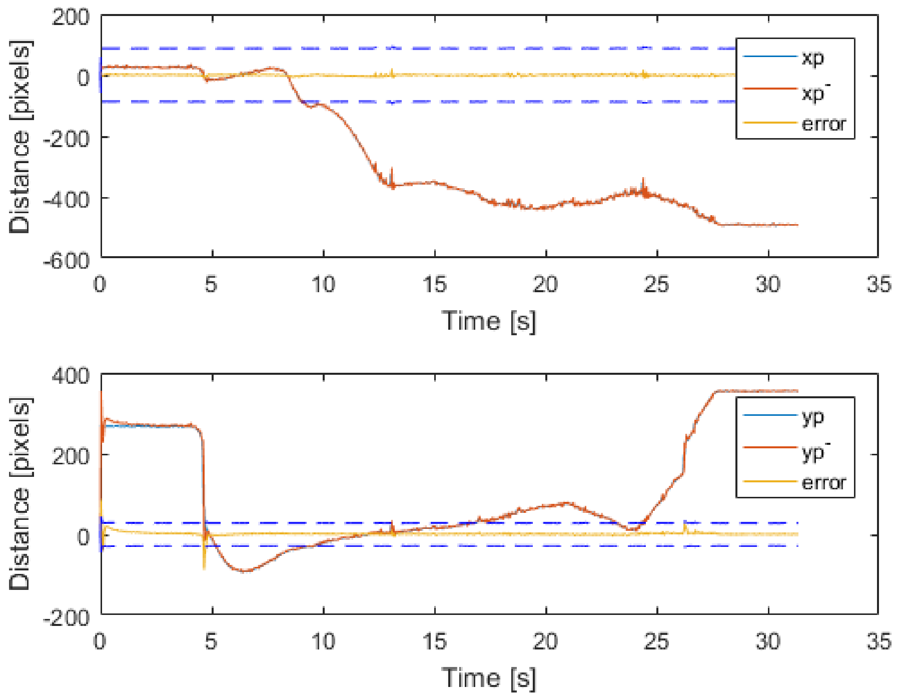


Figure 4.48: Discrete Kalman Filter Implementation Results

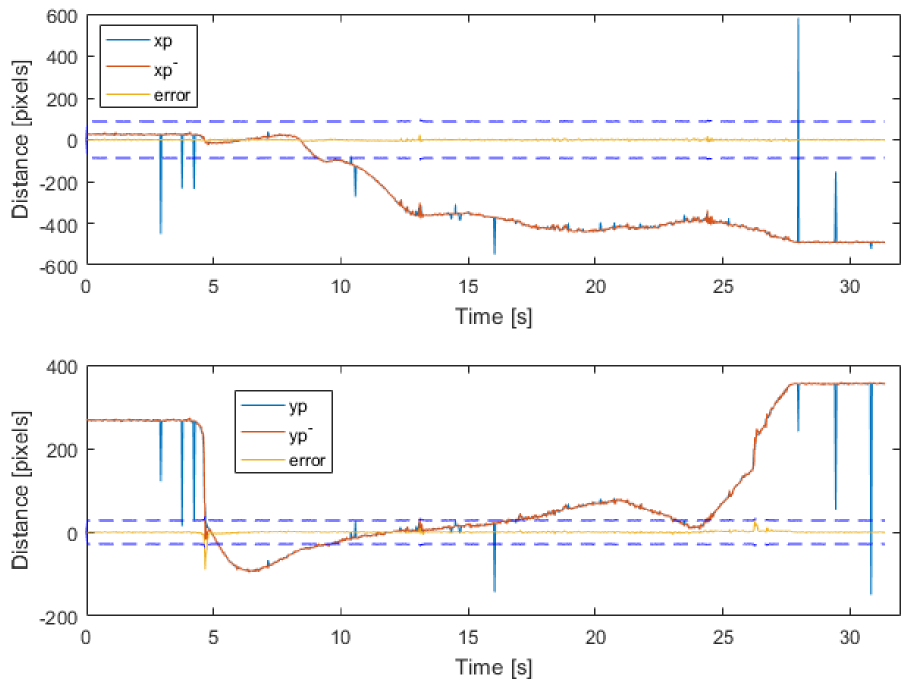


Figure 4.49: Unscented Kalman Filter Implementation Results

As predicted, the accuracy of the vision-based algorithm increases, e.g. the earlier outliers which were due to background noise are now avoided. Since all the outliers in this video are gone, the error is kept close to zero throughout most of the experiment. Both Kalman filters offer very similar performance for the positive detection rate: a 0.21% and 0.43% increase is observed for the DKF and the UKF, respectively. Although the increase itself is not significant, it does however confirm that the method does not degrade the detection rate of the algorithm, which was not the case in our previous experiments.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis has successfully validated the initial hypothesis that existing vision-based algorithms could be tuned or modified to provide real-time detection and 3D tracking of agile UAVs from another UAV using a commercial monocular camera, in both indoor or outdoor conditions. Simple control systems enabling mimicking behavior were designed for two different UAV models.

Between all the tested vision-based algorithms, the Cascade Classifier was shown to offer the best performance in terms of accuracy and computational time. Considering both parameters, the training of the Cascade Classifier was optimized for indoor and outdoor conditions. Outdoor flight testing provided a better detection rate but worse accuracy results as compared to indoor flight tests. The initial vision designs were very sensitive to background noise, which was not acceptable since their outputs are used as the reference trajectory for the mimicking control design.

To solve the issue of background noise, a novel method was introduced to reduce the problem of background noise in the input video stream. This method employs a DKF or UKF to reduce the size of the image region of interest, which reduces false positive detections as well as the computational load of the overall algorithm. Using this method was shown to provide better reliability and a higher framerate, improving the accuracy and detection rate for both indoor and outdoor environments.

5.2 Limitations of the Work

While the results of this thesis are promising, some limitations were found in both the chosen vision-based algorithm and the UAV mimicking control strategy:

- The depth estimation calculations used to estimate the relative movement of the target UAV are limited to a certain range of distances, which depend on the initial distance of the target UAV to the camera. This limitation is due to the fact that the vision-based algorithm calculates a bounding box around the target. The minimum size of this bounding box is user-specified, but values lower than approximately 30x30 pixels give unstable results.

- The mimicking control system accuracy for the 3DR Solo is directly related to the accuracy of the on-board GPS receiver. Whereas the existing receiver achieves a positioning accuracy of approximately ± 1 meter, the motions of the mimicking UAV in the time between frame scans are on the order of centimeters or less. Therefore, the performance of the mimicking UAV will be limited.

- The mimicking control system for the Parrot AR.Drone 2.0 is a feedforward "bang-bang" control design, due to the position of the vehicle not being reported by the Parrot API, and the control inputs requiring full throttle to achieve movement along the lateral axes.

- Although the Kalman filtering used for centroid pixel coordinates has shown excellent performance for recorded video streams, it has not been completely implemented in the onboard vehicle due to the Kalman filter implementation within OpenCV being buggy, and no UKF support being available. A custom DKF and UKF implementation would need to be added to the onboard real-time code.

- The proposed image reduction method will work only if the vision-based detection and tracking algorithm are able to provide stable detection of the target UAV. The method needs to be broadened to handle and recover from detection and tracking losses which may occur during experimental conditions.

5.3 Future Work

Further works needs to be done to iterate on the results presented in this thesis. Most of the points discussed in this section will focus on reducing or eliminating the limitations listed in Section 5.2.

While the issues with the mimicking control system accuracy for the 3DR Solo are due to insufficiently precise positioning sensors, the Parrot AR.Drone 2.0 system is less complicated and thus easier to modify. By installing an external optical motion capture system in the lab, the system could gain a high-accuracy position aiding signal, which could be used to implement a true closed-loop control design. However, the introduction of such a system would force the use of a ground computer, and thus require a change in the software framework. Regarding the 3DR Solo, a solution for the GPS positioning inaccuracy would be to replace the existing receiver with an after-market GPS module employing RTK, such as the Hero+ RTK GPS module. RTK-GPS or Real-Time Kinematic GPS is one of the most precise commercially available positioning technologies, "with which users can obtain centimeter level accuracy of the position in real-time by processing carrier-phase measurements of GPS signals" [60].

More real-world testing is required to fully assess the performance of the image reduction technique and to evaluate the gains in scanning frame rate. Also, more work needs to be done for handling the transition between reduced image processing and whole-image processing, e.g. when the target is visually lost and needs to be found again.

Finally, regarding the depth perception limits, two approaches can be considered: either employ a different vision-based detection algorithm which handles larger depth ranges, or adapt the existing system to use a secondary approach to depth estimation, for instance stereo vision or a dedicated depth sensor. Adding a second camera would entail challenges with mounting two cameras and synchronizing their video feeds, but the results could be on par with the current performance along the X and Y lateral axes. A dedicated depth sensor, such as ultrasound or radar, could prove to be an easier approach, but it will also be associated with extra noise which would need to be filtered out by the onboard software.

Bibliography

[1] *OpenCV documentation*, 2017.

[2] A. L. Hume; C. J. Baker. Netted radar sensing. *CIE International Conference on Radar Proceedings*, 2001.

[3] Yaakov Bar-Shalom and Xiao-Rong Li. *Multitarget-multisensor tracking: Principles and techniques.* Yaakov Bar-Shalom; 1 edition, 1995.

[4] S. Bitzer. The ukf exposed: How it works, when it works and when its better to sample. Technical report, 2016.

[5] Michael Bloesch, Sammy Omari, Marco Hutter, and Roland Siegwart. Robust visual inertial odometry using a direct ekf-based approach. *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference*, 2015.

[6] Marcin Odelga; Paolo Stegagno; Heinrich H. Blthoff. Obstacle detection, tracking and avoidance for a teleoperated uav. *IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[7] Patrick Lambert Bogdan Ionescu, Didier Coquin and Vasile Buzuloiu. Dynamic hand gesture recognition using the skeleton of the hand. *EURASIP Journal on Applied Signal Processing 2005:13, 21012109*, 2005.

[8] J. Y Bouguet. Camera calibration toolbox for matlab. *Computational Vision at the California Institute of Technology*, 2015.

[9] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*, 2008.

[10] P.J. Bristeau, F. Callou, D. Vissiere, and N. Petit. The navigation and control technology inside the ar.drone micro uav. *18th IFAC World Congress*, August 2011.

[11] Aryo Wiman Nur Ibrahim; Pang Wee Ching; G.L. Gerald Seet; W.S. Michael Lau; Witold Czajewski. Moving objects detection and tracking framework for uav-based surveillance. *Fourth Pacific-Rim Symposium on Image and Video Technology (PSIVT)*, 2010.

[12] R. D'Andrea and G. E. Dullerud. Distributed control design for spatially interconnected systems. *IEEE Transactions on Automatic Control*, 2003.

[13] M. Day and J.A. Robinson. Constructing efficient cascade classifiers for object detection. *Signal and Image Processing Applications (ICSIPA) 2011 IEEE International Conference*, pages 46–51, 2011.

[14] Mennatullah Siam; Mohamed ElHelw. Robust autonomous visual detection and tracking of moving targets in uav imagery. *IEEE 11th International Conference on Signal Processing (ICSP)*, 2012.

[15] P. Fieguth and D. Terzopoulos. Color-based tracking of heads and other mobile objects at video frame rates. *Computer Vision and Pattern Recognition, 1997. Proceedings.*, 1997.

[16] Gianluca Antonelli; Elisabetta Cataldi; Filippo Arrichiello; Paolo Robuffo Giordano; Stefano Chiaverini; Antonio Franchi. Adaptive trajectory tracking for quadrotor mavs in presence of parameter uncertainties and external disturbances. *IEEE Transactions on Control Systems Technology*, 2017.

[17] Y. Freund and R.E. Shapire. A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence 14*, 5:771–780, 1999.

[18] Li Zhang G. Schrotter. Robust model driven matching method for face analysis with multi image photogrammetry. *18th International Conference on Pattern Recognition, ICPR*, 2006.

[19] Misha Gajewski. Quebec prison smuggling goes high-tech with drones. Technical report, 2017.

[20] B. Ajith Kumar; D. Ghose. Radar-assisted collision avoidance/guidance strategy for planar flight. *IEEE Transactions on Aerospace and Electronic Systems*, 2001.

[21] Yongduan Song; Junxia Guo. Neuro-adaptive fault-tolerant tracking control of lagrange systems pursuing targets with unknown trajectory. *IEEE Transactions on Industrial Electronics*, 2016.

[22] Fredrik Gustafsson and Gustaf Hendeby. Some relations between extended and unscented kalman filters. *IEEE Transactions on Signal Processing. 2: 545555.*, 2012.

[23] S. J. Julier and J. K. Uhlmann. A new extension of the kalman filter to nonlinear systems. *In Proc. of AeroSense: The 11th Int. Symp. on Aerospace/Defence Sensing, Simulation and Controls*, 1997.

[24] Simon J. Julier and Jeffrey K. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 2004.

[25] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[26] Bahare Kiumarsi; Frank. L. Lewis; Mohammad-Bagher Naghibi-Sistani; Ali Karimpour. Optimal tracking control of unknown discrete-time linear systems using input-output measured data. *IEEE Transactions on Cybernetics*, 2015.

[27] Ryo Furukawa; Ryusuke Sagawa; Amael Delaunoy; Hiroshi Kawasaki. Multi-view projectors/cameras system for 3d reconstruction of dynamic scenes. *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2011.

[28] Malik M. Khan, Tayyab W. Awan, Intaek Kim, and Youngsung Soh. Tracking occluded objects using kalman filter and color information. *International Journal of Computer Theory and Engineering, Vol. 6, No. 5, October 2014*, 2014.

[29] Michael Teutsch; Wolfgang Krger. Detection, segmentation, and tracking of moving objects in uav videos. *IEEE Ninth International Conference on Advanced Video and Signal-Based Surveillance (AVSS)*, 2012.

[30] G. Loianno, Y. Mulgaonkar, C. Brunner, D. Ahuja, A. Ramanandan, M. Chari, S. Diaz, and V. Kumar. A Swarm of Flying Smartphones. *IEEE/RSJ International Conference on Intelligent Robots and Systems IROS*, 2016.

[31] D. A. Mercado; P. Castillo; R. Lozano. Quadrotor's trajectory tracking control using monocular vision navigation. *International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015.

[32] Krystian Mikolajczyk and Cordelia Schmid. Scale and affine invariant interest point detectors. *International Journal of Computer Vision 60(1), 6386, 2004*, 2004.

[33] Nils Gageik; Paul Benz; Sergio Montenegro. Obstacle detection and collision avoidance for a uav with complementary low-cost sensors. *IEEE Access*, 2015.

[34] J. Munkres. Algorithms for assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 1957.

[35] Marta Marron; Juan Carlos Garcia; Miguel Angel Sotelo; Daniel Pizarro Perez; Ignacio Bravo Munoz. Extraction of 3d features from complex environments in visual tracking applications. *Instrumentation and Measurement Technology Conference Proceedings, IMTC/IEEE*, 2007.

[36] T. Hamel N. Guenard and V.Moreau. Dynamic modeling and intuitive control strategy for an x4-flyer. *5th International Conference on Control and Automation*, 2005.

[37] Bill Triggs Navneet Dalal. Histograms of oriented gradients for human detection. *International Conference on Computer Vision And Pattern Recognition (CVPR 05), Jun 2005, San Diego, United States*, 2005.

[38] Shankar S. Sastry Nikhil Naikal, Pedram Lajevardi. Joint detection and recognition of human actions in wireless surveillance camera networks. *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[39] National Research Council (U.S.). Committee on the Future of the Global Positioning System; National Academy of Public Administration. *The global positioning system: a shared national asset: recommendations for technical improvements and enhancements.* National Academies Press, 1995.

[40] R. Mahony P. Pounds and P.Corke. Modelling and control of a quad-rotor robot. *Proceedings of the Australasian Conference on Robotics and Automation*, 2006.

[41] R. Mahony P. Pounds and P.Corke. System identification and control of an aerobot drive system. *Proceedings of Information, Decision and Control*, 2007.

[42] Parrot. *Parrot AR.Drone 2.0 User Manual*, 2013.

[43] Alexander Pon. Evaluation of viola-jones classifier parameters in detecting unmanned aerial vehicles from another unmanned aerial vehicles onboard camera. Technical report, University of Alberta, 2017.

[44] Loretta Ichim; Dan Popescu. Remote image classification based on patch dissimilarity. *24th Mediterranean Conference on Control and Automation (MED)*, 2016.

[45] Mei-Chen Yeh Qiang Zhu, Shai Avidan and Kwang-Ting Cheng. Fast human detection using a cascade of histograms of oriented gradients. 2007.

[46] Giuseppe Loianno; Gareth Cross; Chao Qu. Flying smartphones. *IEEE Robotics and Automation Magazine*, 2015.

[47] 3D Robotics. *3DR Solo User Manual*, 2016.

[48] A. Noth S. Bouabdallah and R. Siegwart. Pid vs lq control techniques applied to an indoor micro quadrotor. *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, 2004.

[49] Miwa Hayashi Sai Vaddi, Hui-Ling Lu. Computer vision based surveillance concept for airport ramp operations. *Digital Avionics Systems Conference (DASC), IEEE/AIAA 32nd*, 2013.

[50] Martin Saska, Tomas Baca, Justin Thomas, Jan Chudoba, Libor Preucil, Tomas Krajnik, Jan Faigl, Giuseppe Loianno, and Vijay Kumar. System for deployment of groups of unmanned micro aerial vehicles in GPS-denied environments using onboard visual relative localization. *Autonomous Robots*, 2016.

[51] David Schneider. Flying selfie bots. *Top Tech North America IEEE Spectrum Magazine*, 2015.

[52] K. H. Bethke; B. Rode; A. Schroth. Combination of low power radars and non-rotating sector antennas for surveillance of ground moving traffic on airports. *Sensors, IEEE*, 2002.

[53] Shahid Shafique; Noor M. Sheikh. Simple algorithm for detection of elliptical objects in remotely sensed images for uav applications. *6th International Bhurban Conference on Applied Sciences and Technology*, 2009.

[54] Lin Wang; Fei Su; Huayong Zhu; Lincheng Shen. Active sensing based cooperative target tracking using uavs in an urban area. *2nd International Conference on Advanced Computer Control (ICACC)*, 2010.

[55] S. Lupashin; M. Hehn; M. W. Mueller; A. P. Schoellig; M. Sherback and R. D'Andrea. A platform for aerial robotics research and demonstration: the flying machine arena. *Mechatronics*, 2014.

[56] Robert A. Singer. Estimating optimal tracking filter performance for manned maneuvering targets. *IEEE Transactions on Aerospace and Electronic Systems*, 1970.

[57] The Environmental Software Source. *Map Projections and Coordinate Conversions.*

[58] Jan Farlik; Miroslav Kratky; Josef Casar; Vadim Stary. Radar cross section and detection of small unmanned aerial vehicles. *17th International Conference on Mechatronics - Mechatronika (ME)*, 2016.

[59] B. L. Stevens and F. L. Lewis. *Aircraft Control and Simulation.* John Wiley and Sons, 2003.

[60] Tomoji Takasu and Akio Yasuda. Development of the low-cost rtk-gps receiver with an open source program package rtklib. *International Symposium on GPS/GNSS, ICC Jeju, Korea*, 2009.

[61] P. Viola and M.J. Jones. Rapid object detection using a boosted cascade of simple features. *Conference On Computer Vision And Pattern Recognition*, 2001.

[62] P. Viola and M.J. Jones. Robust real-time face detection. *International Journal of Computer Vision 57*, 2:137–154, 2004.

[63] Y.Q. Wang. An analysis of the viola-jones face detection algorithm. *Image Processing On Line*, 2014.

[64] G. Cross Y. Mulgaonkar and V. Kumar. Design of small, safe and robust quadrotor swarms. *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[65] Jian Wang Yisha Liu, Nan Jiang and Yiwen Zhao. Vision-based moving target detection and tracking using a quadrotor uav. *11th World Congress on Intelligent Control and Automation (WCICA)*, 2014.

[66] R. Miller Z. Sun, G. Bebis. On-road vehicle detection using optical sensors: a review. *The 7th International IEEE Conference on Intelligent Transportation Systems Proceedings*, 2004.

[67] Bahareh Kalantar; Shattri Bin Mansor; Alfian Abdul Halin; Helmi Zulhaidi Mohd Shafri ; Mohsen Zand. Multiple moving object detection from uav videos using trajectories of matched regional adjacency graphs. *IEEE Transactions on Geoscience and Remote Sensing*, 2017.