Computer Vision Based UAV Stabilisation Technique

Kostiantyn Perehuda

Supervised by Dr Tim Morris

The University of Manchester

Department of Computer Science

June 2022

Abstract

Enabling an easy, intuitive, high-level user control over Unmanned Aerial Vehicles (UAVs) has been an important research topic for years. Solutions to this problem involve designing UAV control systems which are capable of abstracting all the intricacies of piloting the UAVs (such as controlling the power level of each individual turbine) into high-level commands. One type of such control systems are the drone stabilizing systems. Their purpose is to ensure that the controlled drone correctly beys its high-level commands.

This report analyses existing drone stabilizing techniques and proposes a basic implementation of a computer vision based system for stabilising the drone. Performance of the developed system is then analyzed and compared to an existing stabilisation system.

Despite being worse than existing state-of-the-art solutions, the developed algorithm is capable of stabilising the drone and, with sufficient tuning and potential future improvements, will be able to compete with existing solutions.

Acknowledgements

I would like to thank my supervisor, Dr Tim Morris for guiding me and providing feedback during the course of this project. I would also like to thank my friends for helping with testing the application.

Contents

List of Figures

Chapter 1

Introduction

1.1 UAVs and their Applications

Unmanned Aerial Vehicles (UAVs) have become a significant part of modern technology. They started as an expensive specialized equipment used for military purposes. However, due to massive technological advancements, UAVs have become extremely affordable to produce and found their way to a mass market. Modern UAVs vary significantly in their form-factor and are used in a plethora of different applications.

The most common form-factor of the UAVs is a **Quadcopter** - a four rotor UAV (*Devopedia* 2022). Quadcopters have a massive variety of applications in a civilian sector. Here are the most popular (*RiseAbove Blog* 2022), (*MyDearDrone Blog* 2022):

- Aerial Photography and Videography
- Filming and Movies
- Agriculture and Crop Monitoring
- Mapping and Surveying
- Product Delivery
- Drone Racing and Hobby

Some of these applications require the drones to be operated by people e.g. for Drone Racing. Others require the drones to be able to fly autonomously without explicit user control, e.g. for Product Delivery. The rest requires a mixture of those. For example, when filming movies, one sometimes wants the drone to be able to follow the moving object automatically while manually controlling the camera parameters (such as zoom or focus) to achieve a good clip. But when filming aerial scenes, the cameraman would want to have a full precise control over the motion of the drone.

1.2 Introducing Drone Stabilisation Algorithms

To enable a successful use of quadcopters in the aforementioned applications, advanced drone controlling algorithms have been developed. Such algorithms are responsible for making it easy for an end user to control the motion of the drone and/or enabling autonomous drone flights. They abstract low level control commands (usually controlling individual rotors) into high level commands (such as "move forward with a speed of 5 m/s") and ensure that the drone correctly obeys those high level commands.

One type of these algorithms is so called **drone stabilisation algorithms**. The main purpose of such algorithms is to ensure that drone does not deviate from its motion setpoint due to external factors which affect the drone's position in the air, such as side winds. Such algorithms are one of the key factors improving the quality of experience of using quadcopters. Nowadays almost every publicly available quadcopter utilizes at least one of existing stabilisation algorithms or their combination. An obvious example is a drone hovering mode. With hovering mode enabled, the drone will maintain its spatial position in the air and counteract to factors that could affect the position without manual user intervention. As an example, imagine releasing an air balloon into the air. It will most probably drift away. The stabilisation system, however, will not allow the drift to happen. It will adjust the UAV's rotor speeds to eliminate the drift.

1.3 Project Aims

This project focuses on a research and implementation of a specific type of drone stabilisation algorithms - **computer vision based stabilisation algorithms**. An idea behind these is to have a bottom looking camera attached to the drone. This camera is then used to observe the area underneath the drone and use computer vision techniques such as feature detection and tracking to estimate the motion of the drone relative to the ground. The motion (drift) estimate is then fed back into the drone controller and the controller adjusts the rotor output signals to counteract to the encountered drift. A common implementation of such controllers are Proportional Integral Derivative (PID) Controllers (*Kong Wai Weng* 2012). They are designed to adjust the controlled signal based on the feedback provided from the environment (drone's drift in this case).

Hence, the main aims of the project are to:

1. Understand the theory behind UAV stabilisation algorithms.
2. Explore how computer vision can be used in the aforementioned algorithms.
3. Deliver an application which will:
    - Provide an easy-to-use an understandable flying drone controller user interface.
    - Assist the user in controlling the drone by implementing a drone hovering mode via a vision-based drone stabilisation system.

1.4 Report Structure

- Chapter 2 will provide an overview of hardware components used in this project. Based on them, the underlying application requirements and project success criteria will be defined.
- Chapter 3 will analyze existing stabilisation algorithms and underlying theory behind them.
- Chapter 4 will provide in-depth description of how Parrot A.R.Drone 2.0 implements its hovering mode. This will be the main point of reference for the rest of the project.
- In Chapter 5 the reader will be presented with an underlying architecture and implementation of the stabilisation algorithm developed during the course of the project.
- Chapter 6 will introduce an architecture of the Main Application used to control the drone, describe the main application components, and discuss how to stabiliser algorithm fits into these. In this chapter will also talk about APIs and Libraries used by the application and the reasons why they have been selected.
- Chapter 7 will discuss how the code was tested during development and how the good set of stabilisation algorithm parameters was found.

- Chapter 8 will evaluate the performance of the developed stabilisation algorithm through a number of defined performance criteria and compare the algorithm to the one used by the Parrot A.R.Drone 2.0.
- Chapter 9 will conclude the report by identifying the vector of the future development of the Application.

Chapter 2

Hardware and Application Requirements

Although the aims of the project are fixed, the actual implementation and design of the underlying algorithm and application in general are heavily dependent on the hardware components used. This chapter will provide a detailed description of hardware components used in the project and derive the key application requirements based on the hardware used. Project success criteria will also be defined in this chapter.

2.1 UAV

The UAV used in this project is a Parrot A.R.Drone 2.0. Its full technical specifications and a client API description is provided by *S. Piskorski, N. Brulez, P. Eline, F. D'Haeyer* (2012).

Below is the summary of key takeaway points about the drone:

- The drone has a plethora of sensors, including a 720p front-faced a 360p bottom-faced cameras. Only the bottom-faced camera will be used for the stabilisation algorithm.
- The drone has on-board processing capacity provided by ARM Cortex A8 1GHz processor and it has 1GB of DDR2 RAM. However the user cannot access/modify the firmware of the UAV. Thus an external computing device (A Windows Computer) will be used to host the stabilisation algorithm.
- The drone does not have a dedicated Remote Controller device (e.g. an RC transmitter).

Instead, the drone exposes its interface by utilising an Open Systems Interconnection (OSI) model for wireless communication. Hence, there must be a computer program responsible for communicating with and controlling the drone. A use of OSI model implies that drone can be controlled pretty much by any computer (PC, or a phone), because most of the computers support full OSI protocol stack.

- The UAV doesn't utilize any high level application layer protocols. The communication happens by exchanging AT commands and data directly at a transport layer. Most of the communication and video streaming happens via a User Datagram Protocol (UDP), while critical data (e.g. Drone Calibration Info) is exchanged using Transmission Control Protocol (TCP). A WiFi protocol is used at the MAC layer.
- Due to security reasons, only the first client which connects to the drone, can communicate with it. This implies that stabilisation algorithm cannot be a separate application, independent of the application controlling the drone. They must be the parts of a single application.

2.2 Derived Application Requirements

The application must (key requirements):

1. Run on a windows machine.
2. Provide a user with an interface to control the motion of the drone in a user-friendly manner.
   (a) A video from the UAV camera should be displayed to the user to improve quality of experience.
   (b) In input from the user must be acquired via a convenient input device which supports a precise control over the UAV. An example of such input device is a wireless gamepad (e.g. wireless xbox controller).
3. Implement a drone hovering mode. The hovering mode must:
   (a) Correctly function outdoors in good lighting conditions at a height ranging from 1 meter to 5 meters above the ground.
   (b) Keep a drone stable in a specified position in the 3D world without user intervention.

(c) Long term spatial deviation from the specified position (drift) must be minimised.

(d) The maximum short term spatial deviation from the specified position due to rapid wind changes of at most 1 meter in any direction.

The application should (additional requirements):

1. Run on Windows/Linux/MacOS computers and on Android/iOS mobile devices (cross-platform).

2.3 Success Criteria

A project will be considered to be successful if **all the key application requirements** are achieved.

Chapter 3

Existing Stabilisation Technologies

This chapter focuses on existing solutions to a UAV stabilisation problem. Stabilisation problem can be formulated as a problem of controlling the drone's speed and/or position i.e. to make sure that those do not deviate from desired values. In the case of hovering mode, for example, a desired value for the drone speed is 0.

3.1 Closed Loop Control Systems

Typically controlling a process or an action happens through setting some control variables to a needed value. In the case of drones, controlling the position/speed happens through setting the power level of the rotors. However, in real life scenarios, the actual behaviour of the system depends not only on the control variables that one can set, but also on external factors which can't be controlled, and what's more important, are usually unknown and unpredictable. These external factors can cause disturbances to the behaviour of the system and even cause a failure. An obvious example of such external factors in case of UAVs is gravity, wind, and drag. To deal with this issues caused by external factors, closed loop control systems were introduced. The main idea behind such systems is that the system's output is fed back into controller together with the system's setpoint (desired output) to correct for errors caused by any external factors. The feedback is generally provided by a sensor in the system.

Figure 3.1 shows a typical architecture of a Closed Loop Control System. A common practice for such systems is to subtract the feedback from the setpoint to compute the system's error and feed the error into the controller. These systems are called **negative feedback control**

systems. A controller then uses the error to produce a control signal, which in turn affects the system's behaviour and decreases (we hope) the system's error. The process is continuously repeated. More information can be found here: *Electronics Coach* (2022).

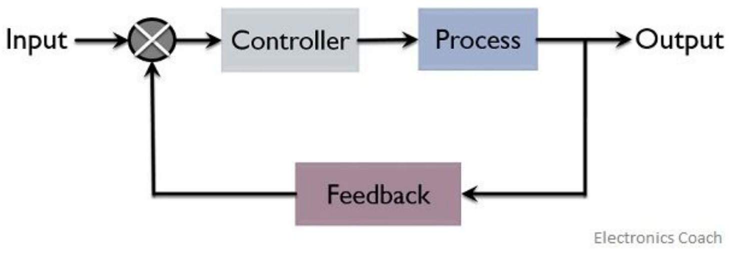


Figure 3.1: Closed Loop Control System

3.2 A Pipeline of Drone Stabilisation Algorithms

All the drone stabilising algorithms utilise the closed loop control architecture. It is an iterative process, and each iteration consists of the following three phases:

1. **Sensing phase**. The drone senses the changes in its position and orientation. This functionality is usually provided by the UAV's Inertial Measurement Unit (IMU). The IMU usually encompasses a range of sensors such as a gyroscope, an accelerometer, a pressure sensor, an ultrasonic sensor, cameras, and other sensors. Alongside with IMU, other modules exist to provide additional data to the drone (e.g. a GPS module).

2. **Motion Estimation phase**. The sensor data is processed by a Drone Flight Controller (a chip processing the data) and an estimate of UAV's prosition and orientation is produced.

3. **Reaction phase**. Rotor driving signals are adjusted to minimise any errors in the desired speed and orientation of the UAV. This job is usually done by a Drone Flight Controller together with an Electronic Speed Controller (ESC).

Different types of stabilizers exist due to different implementations of each of the phases.

3.3 Categorisation of UAV Stabilisation Techniques based on the Motion Estimation Techniques

3.3.1 Accelerometer and Gyroscope Stabilisation

It is the most common and the cheapest system to implement. Every drone (except racing drones with "3d flight mode") has this system enabled.

Fintan Corrigan (2020) provides a good overview of how the algorithm works:

1. IMU uses one or more gyroscopes to detect changes in the UAV's rotation in all three directions (yaw, pitch, and roll).
2. IMU also uses one or more accelerometers to measure the UAV's rate of acceleration.
3. An on-board processor calculates the UAV's current spatial and angular velocity from obtained sensor readings.
4. The velocity is integrated to get an estimate of the UAV's position.
5. The Flight Controller adjusts the motion signals based on gathered data and sends the adjusted signals to the ESC.
6. ESC converts motion signals to level of thrust for each individual motor and sends the level signals to the motors.

The exact implementation of steps 3-5 varies from drone to drone. It is also worth mentioning that sometimes only gyroscope stabilisation is used. The implication of this is that only drone's orientation in 3d space is kept stable. Note that a gyroscope stabilisation doesn't cope with positional drift.

For example, consider a specific implementation of a gyro stabilisation algorithm from *Mr. G. Pradeep Kumar M.E., B. Praveen, U. Tarun Nisanth, M. Hemanth* (2017). It uses a gyroscope together with a calibrated magnetometer to detect the rotational changes of the UAV. Since the signal produced by the sensors is noisy, it is filtered using a low-pass filter. An output signal is the voltage which is proportional to the angular velocity. Flight controller then calculates the deviation of the drone's from the setpoint and signals to the rotors are adjusted.

Benefits:

- Cheap to implement.
- Low power consumption due to absence of expensive calculations.
- Gyroscope stabilisation is very effective at keeping the drone's orientation stable.

Drawbacks:

- Noisy sensor outputs can introduce error in estimations and lead to eventual drift (especially when it comes to positional stabilisation using accelerometers).

3.3.2 GPS Stabilisation

More expensive and advanced drones (e.g. DJI Mavic drones) are equipped with Global Positioning System (GPS) modules. These provide a range of functionalities, including:

- Altitude and Position Hold (Hovering).
- Return to Home.
- Flight Reporting.
- Waypoint Navigation.
- Localisation and Mapping.

More information about functionalities enabled by GPS modules can be found here: *DroneBlog* (2022)

GPS based hovering algorithms are pretty similar to gyro algorithms. The difference is in the means by which the positional data is collected. When the drone locks on to a GPS signal, it is able to identify its position and detect a positional drift. The drift is then minimized by the flight controller.

Benefits:

- Efficiently copes with long-term drift. The position signal is obtained in a global coordinate system. This eliminates the possibility of small local errors to add up to a big error over time (unlike in accelerometer based stabilizers).

Drawbacks:

- Increased battery consumption due to more processing required as well as power required by GPS module itself.

3.3.3 Computer Vision Based Stabilisation

This approach uses computer vision methods to estimate the motion of the drone. The rest of the logic is the same as in the other methods. There are plenty of algorithms and sensors that can be used to estimate the UAV's motion. Below are different types of sensors:

- Single camera to get an image of the environment.
- Calibrated stereo camera system to get a depth map of the environment.
- Lidar camera to get spatial environment information.
- Ultrasonic sensors to measure distance to nearby objects.
- And others.

The state of the art in this field are Visual Simultaneous Localization and Mapping (SLAM) Techniques. And examples of these techniques are implemented by *J. Skoda, R. Bartak* (2015) and *Dávid Dezső and Kornél Sarvajcz* (2018).

SLAM works by tracking the features over the sequence of images. Having found enough corresponding points between frames, the camera motion vector can be recovered by triangulation using 8-point algorithm for example.

SLAM techniques also involve "loop detection" realized by detecting known landmarks in the image and correcting for any accumulated drift. The drift correction is possible since the location of the landmarks is known. Initializing/inserting the landmarks into the map is an ongoing research topic and often involves user intervention. For example, *J. Skoda, R. Bartak* (2015) requires the user to initialize the map before the localisation can happen. It requires the user to capture some key frames and moving the drone by a fixed known distance between the successive key frames.

J. Skoda, R. Bartak (2015) mentions one problem with recovering pose using triangulation. If the transition between successive frames is too small, then the accuracy of triangulation can become poor. That is why the algorithm doesn't use every frame to update its estimates.

However when implementing my algorithm, I found it very hard to track the needed amount of points over several frames, and I needed to use a simpler solution.

Simpler solutions assume negligible UAV rotation between frames and only estimate the motion parallel to the image frame. In the case of drones they only estimate horizontal motion. It is done by tracking the features between successive images and estimating the average (or least squares fitted) displacement vector of those features in the image plane. Later in Section 4.1.1 we will see that this is what the Parrot A.R. Drone 2.0 is doing. This approach is utilized by the algorithm I've implemented and is described in Section 5.2.

Benefits:

- Produces a reliable estimation of the UAV speed.

Drawbacks:

- Local solutions can sometimes be noisy. Noise filtering is required.
- Extremely power hungry.

3.3.4 Summary and Common Practicalities

There is a variety of different techniques for motion estimation, which vary in their complexity, accuracy and power consumption. A choice of the method depends heavily on the hardware and estimation accuracy requirements. In practical scenarios it is common to combine different approaches and utilize redundancy (e.g. multiple gyroscopes) to achieve a more precise and reliable output.

3.4 Control Techniques

So far we have only discussed how to obtain an estimate of the UAV's position, orientation, and velocity. To complete a closed loop control system, a controller algorithm responsible for processing the feedback is needed.

H. Teimourian, K. Dimililer, F. Al-Turjman (2020) gives a summary of the most popular controllers that are used by quadrotors:

1. Proportional Integral Derivative (PID) Control (*Kong Wai Weng* 2012).

2. Linear Quadratic Regulator (LQR) Control (*Heri Purnawan, Mardlijah and Eko Budi Purwanto* 2017).

3. H-Infinity and Mu Synthesis Control (*Matlab* 2020).

For this particular project the PID control was selected as a control algorithm. It has several advantages over the others:

1. It is relatively easy to understand and does not require advanced signal processing and control theory knowledge.

2. It is simple to implement.

3. Despite its simplicity it is robust and effective and is heavily used for UAV control. In fact, as we will see later in Chapter 4, PID algorithm is used to control the Parrot A.R.Drone 2.0. It is also the one used by *J. Skoda, R. Bartak* (2015) and *Dávid Dezső and Kornél Sarvajcz* (2018).

Section 3.4.1 will provide an explanation of how a general PID control works.

3.4.1 Proportional Integral Derivative (PID) Control

Proportional Integral Derivative (PID) Control is a typical model used in Closed Loop Control Systems. It takes as an input the system's output setpoint and the measurement of the actual system's output value and computes the difference between the two - the system error, usually marked as $e(t)$. The error is then processed by three components: Proportional, Integral, and Derivative. The results are added together to form the output value.

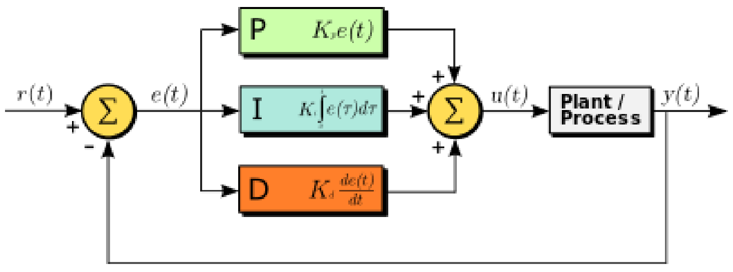


Figure 3.2: PID Control Diagram

Figure 3.2 illustrates how a typical PID controller is implemented. Each component represents a mathematical function over the error multiplied by some constant parameter called Gain.

Proportional Component is just an identity function multiplied by the proportional gain (Equation 3.1). It is the main driving force of the algorithm and it tries to correct for **current error**. The bigger the error is, the bigger the output signal that is produced. However, this term cannot accomplish the task on its own when external forces are present. Imagine lifting a weight and keeping it at a steady level above the ground. You obviously need to apply force to lift it, however, when the weight reaches the required position, you still need to apply some force to overcome the gravity and make it stable. But in this case Proportional controller will output 0 and lead to dropping the weight.

$$P = K_p * e(t) \tag{3.1}$$

In order to overcome this problem, integral component is introduced. Equation 3.2 shows how the component is computed. The purpose of the component is to correct for the **previous errors**. Since P controller will yield some error, this error will be accumulated in the integrator over time and cause the output signal to raise/fall, eventually eliminating the error completely. PI controllers are capable of stabilising the systems and are commonly used in electronics.

$$I = K_i * \int_0^t e(t)dt \tag{3.2}$$

Derivative component is optional and is given by the term(Equation 3.3). They are used to avoid overshooting that can be introduced by integral component and thus correct for **future errors**. The principle is that they slow down the rate of change of error. And avoid overshooting by not allowing the error to decrease too quickly. The problem with derivative component is that derivative of a function highly amplifies high frequency noise and can cause disturbances in the controller's function. Thus derivative component is avoided if possible, and if not, it is coupled with a low pass filter to decrease the effects of the noise.

$$D = K_d * \frac{\partial}{\partial t}e(t) \tag{3.3}$$

Chapter 4

An Overview of The Parrot A.R.Drone 2.0 Stabilisation System and Hovering Mode

Chapter 3 presented the theory and algorithms behind a general UAV Stabilisation System. This chapter takes the discussion further, by looking at a particular example of such a system. It provides an in depth description of how The Parrot A.R.Drone 2.0 Stabilisation System and Hovering Mode work. The chapter is based on the *Pierre-Jean Bristeau, François Callou, David Vissière, Nicolas Petit* (2011) paper.

As the Parrot A.R.Drone 2.0 is the drone used in the project, it makes sense to compare the algorithm developed during the course of the project to the algorithm used by the drone. Moreover, the developed algorithm is inspired by the drone's implementation of the Hovering mode. Section 4.1 focuses on how the UAV's motion is estimated. Section 4.2 focuses on how the UAV's Control Logic works and how the motion signal is processed to enable hovering. Section 4.3 looks at the performance of the algorithm and concludes the chapter.

4.1 Motion Estimation Algorithm

As mentioned in Section 3.3.4, to achieve the best results, the UAV combines different motion estimation techniques.

- An Internal Gyroscope is used to estimate the drone's Angular Velocity. To deal with the noise of the Gyroscope readings, the readings are fused with the readings produced by an accelerometer using a complementary filter.
- A combination of gyroscope, accelerometer and is used to get an initial estimate of the drone's attitude.
- Pressure sensor readings are used to estimate the drone's altitude.
- Horizontal velocity is estimated using computer vision techniques which are discussed in section 4.1.1).
- The horizontal velocity estimate produced by the vision system is used to de-bias and de-noise the accelerometer readings. This done is by fusing the estimates using a complementary filter.
- Eventually a precise velocity estimate is produced from the de-biased accelerometer readings by utilising the UAV's aerodynamics model.

All these techniques form a tightly integrated vision and inertial navigation filter.

4.1.1 Horizontal Velocity Estimation Using Computer Vision

A bottom-looking camera is used to provide the speed estimation. Two complementary algorithms are used. The UAV switches between the two depending on the quality of the results they produce.

- The first method uses a sparse optical flow algorithm over the whole picture range to estimate the horizontal velocity. It implements the Lucas-Kanade methodology for optical flow estimation. The method ignores vertical and angular displacements (those are provided by the gyroscope and accelerometer readings). The induced error is canceled in most cases by subtracting the displacement of the optical center derived from the attitude change which is produced by the gyroscope and accelerometer readings.
- The second method uses a feature tracking approach. A FAST corner detector is used to find corner-like features. The features are matched to those detected in a nearby region in the next image. The camera motion is computed using an iteratively weighted least-squares (IRLS) minimization procedure. The motion vector produced at each iteration is

the one the minimizes the sum of square differences between itself and the displacements between the corresponding tracked features in the image. The result of each iteration is used to filter out erroneous tracker candidates positions. Then, the trackers updated positions are searched in the image, within a frame whose radius is adapted to the expected trackers screen speed. This approach assumes that the depth of the scene is uniform. That means that the ground level underneath the drone is uniform and that the drone's orientation is approximately horizontal.

The paper concludes that the second approach provides more accurate results, however it is more demanding to the quality of the image. The first algorithm can handle scenes that have a very low contrast, and, due to coarse-to-fine optical flow estimation, it can handle both small and large speed motions. The A.R.Drone 2.0 uses the first algorithm by default and switches to the second for the sake of increased accuracy when the speed of the UAV is low and the quality of the image is good. The UAV switches back to the first algorithm if the speed becomes higher that the predefined threshold, or when the number of detected features to track becomes too low.

4.2 Control Architecture

The Control Logic is quite complex and is implemented using a finite state machine (FSM).

The three most important FSM states are:

- Flight mode. When this mode is enabled, the motion of the drone is controlled by the user.
- Hovering mode.
- Gotofix. Used to stop the drone when transitioning between the Flight mode and the Hovering mode.

It is important to note that stabilisation works in both Hovering and Flight modes. The objective of the stabilisation is to ensure that drone's actual attitude and velocity match those that are set by the user (called setpoints). The main difference is that in the flight mode the velocity setpoint is provided by the user (through the UI), and in hovering mode it is set to 0.

The "Hovering mode" (zero speed and zero attitude) is implemented by the Hovering Control

Loop which consists of a Proportional Integral (PI) controller on the speed estimate. Note that a PI controller is a specific type of a PID controller discussed in section 3.4.1. In a PI controller the Derivative gain is set to zero, meaning that the derivative component is ignored.

The "Flight mode" implementation is a little bit more complicated, but under the hood it uses a PI controller to track the drone's attitude based on the velocity setpoint provided by the user and a Proportional (P) Controller to track the the rotors' speed based on attitude. Here the term "track" means bring it as close to the desired value as possible.

"Gotofix" mode uses a motion planning together with a transient trajectory generation technique to quickly stop the drone from moving. It is an important feature of the drone, however it is irrelevant to the current project.

4.3 Performance and Conclusions

The practical performance of the algorithm is shown in the screencast. Below are the key facts obtained from the testing.

1. The algorithm works extremely well in the outdoor setting but struggles indoors and in poor lighting conditions.
2. At the height of 1.5 meters above the ground in the daylight the UAV is kept perfectly stable with only up to 20cm of instantaneous drift from the set position, however the drift is eliminated over time due to integral component of the PI controller used.
3. The algorithm is robust to sudden wind blows and works in a windy weather too.
4. However, when hovering over uniform low-textured ground (e.g. indoors) or in poor lighting conditions the UAV relies only on accelerometers and gyroscopes and can lead to a rapid drift if those are not properly calibrated.

The optical flow based motion estimation and PID control techniques have shown to be pretty robust and reliable when applied in UAV stabilisation algorithms. These techniques are also relatively easy to implement compared to more advanced options (e.g. SLAM based motion estimation and LQR control). Due to these reasons Optical Flow and PID control have been selected as the main drivers of the proposed stabilisation algorithm discussed in Chapter 5.

Chapter 5

Developed Stabilisation Algorithm

Having considered all the underlying theory behind stabilising algorithms in Chapter 3 and reviewed an existing implementation in Chapter 4, this chapter focuses on the description and implementation of the algorithm developed during the course the project and the underlying design decisions.

Section 5.1 will provide an overview of the algorithms pipeline and the rest of the sections will focus on each individual stage of the pipeline.

5.1 The Algorithm's Pipeline

The proposed system follows a Closed Loop Architecture discussed in Section 3.1. And each iteration of the algorithm follows a pipeline discussed in Section 3.2.

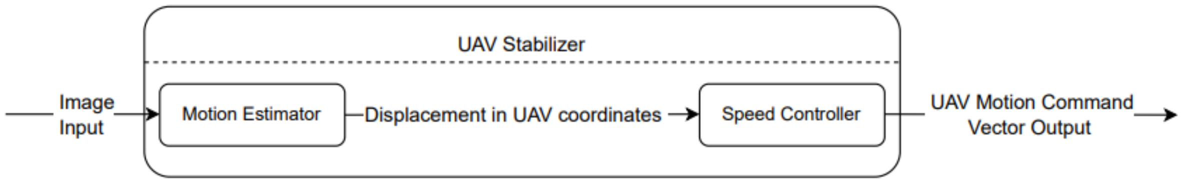

UAV Stabilizer
Image Input
Motion Estimator
Displacement in UAV coordinates
Speed Controller
UAV Motion Command Vector Output

Figure 5.1: UAV Stabilizer Pipeline

Figure 5.1 illustrates a high level architecture of the stabilizer:

- As soon as the application receives a new image from the drone's bottom camera, it is passed to a "Motion Estimator" component.

- The "Motion Estimator" uses an optical flow technique to estimate the displacement between the current image and the previous image. It applies noise filtering techniques to suppress noise and converts the displacement vector from the image coordinate frame to the drone coordinate frame.
- The estimated displacement is then passed to a "Speed Controller".
- "Speed Controller" in turn uses a PID Control to track the velocity or position of the drone and produces as an output a motion command vector to be sent to the drone. Here the term "track" means bring it as close zero as possible.
- The rest of the application takes care of passing the output of the system to the drone, as well as feeding images received from the drone into the system. More about this in Chapter 6

Now let's discuss each of the components in detail.

5.2 The Motion Estimator

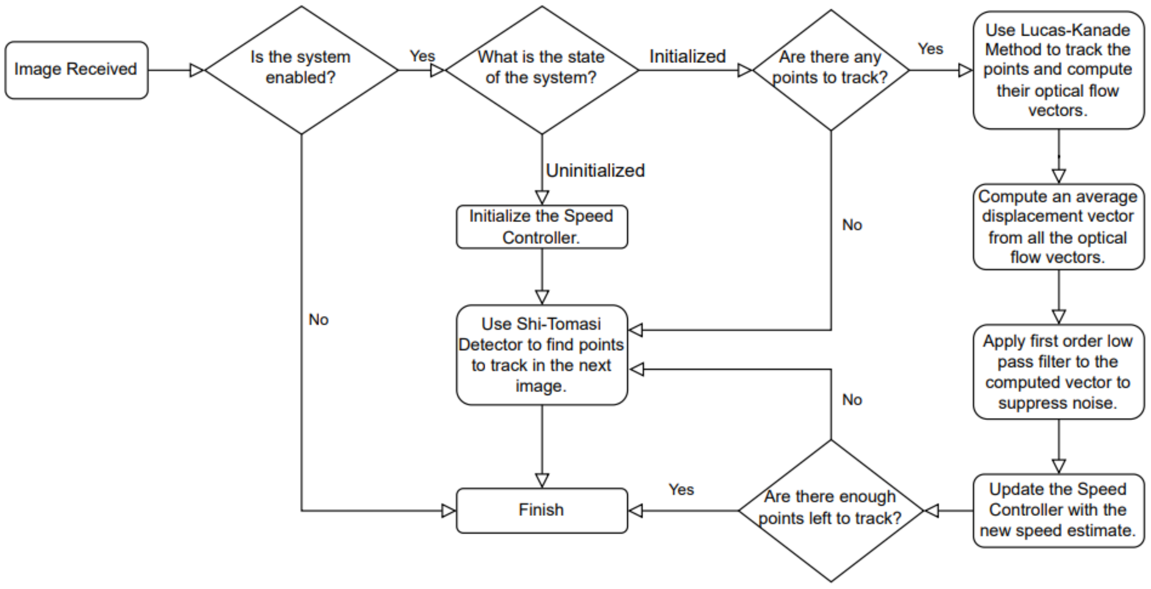


Image Received
Is the system enabled?
Yes
No
What is the state of the system?
Initialized
Uninitialized
Initialize the Speed Controller.
Are there any points to track?
Yes
No
Use Lucas-Kanade Method to track the points and compute their optical flow vectors.
Compute an average displacement vector from all the optical flow vectors.
Apply first order low pass filter to the computed vector to suppress noise.
Update the Speed Controller with the new speed estimate.
Are there enough points left to track?
Yes
No
Use Shi-Tomasi Detector to find points to track in the next image.
Finish

Figure 5.2: Motion Estimator Component Architecture

Figure 5.2 illustrates the sequence of actions that happen when the Motion Estimator component receives a new image. The diagram should be self-explanatory so let's not focus on the structure of the component itself, but discuss the key functions inside it.

5.2.1 Lucas-Kanade Method to Track Keypoints

A specific keypoint found in an image is tracked in another image by finding the same (or very similar) point in that second image and computing the displacement vector between the two.

One method to do this is to use an optical flow technique. Optical flow estimates how intensity (light) patterns move between images. Finding an optical flow for a point (pixel) on an image involves solving an optical flow constraint equation. This equation can be derived by assuming that an optical flow vector for the pixel is very small and that the intensity of the pixel is the same in the two images, and then applying first order Taylor approximation to the pixel's motion. However, the derived equation is under-constrained. The solution is a line in a 2D space, and the true optical flow vector lies somewhere on the line, but we don't know where. We can compute a component of the optical flow vector that is perpendicular to the found line (called normal optical flow), but the parallel component cannot be computed. This is known as the aperture problem.

$$\frac{dI}{dx}u + \frac{dI}{dy}v + \frac{dI}{dt} = 0 \tag{5.1}$$

Equation 5.1 is an optical flow equation for a pixel in an image. u and v are horizontal and vertical components of the optical flow. $\frac{dI}{dx}$ is a horizontal image derivative, $\frac{dI}{dy}$ is a vertical image derivative and $\frac{dI}{dt}$ is the image derivative in the time frame (with respect to the previous image).

Lucas-Kanade Method (*First Principles of Computer Vision* 2021) is one of the approaches to find an approximate solution to the optical flow problem. It is a sparse approach, which means it only computes an optical flow for a sparse set of locations in the image. To deal with the constraint issue, Lucas-Kanade Method assumes that in a tiny local patch of the image all the pixels have the same optical flow vector. Due to this assumption, we can now form a system of simultaneous optical flow equations (number of equations is equal to the number of pixels in the patch) and solve them using a least-squares method, thus obtaining the optical flow vector.

5.2.2 Shi-Tomasi Corner Detector

To be able to solve the system of equations in a Lucas-Kanade Method, the equations in the system have to be independent of each other (meaning that one equation cannot be derived by using a linear combination of others). That means that the derivatives ($\frac{dI}{dx}$, $\frac{dI}{dy}$) should be unique, sparse and uncorrelated. I other words, the image patch should have a well-defined texture. A uniform region is bad and an edge is bad.

In technical terms it means that the Eigenvalues of the distribution of gradients in the region should be big (meaning not a uniform patch) and it must not be the case that one is much bigger than the other (must not be an edge). These regions are known as corners. The three most famous corner detection algorithms are:

1. Harris Corner Detector
2. FAST Corner Detector
3. Shi-Tomasi Corner Detector

Haydar A.Kadhima, Waleed A.Araheemah (2019) summarizes that Shi-Tomasi detector is more robust to noise than Harris and FAST detectors, but FAST is the best in terms of keypoint localisation. Since the images coming from the drone are inherently very noisy, Shi-Tomasi has been chosen. Also in practice Shi-Tomasi gives slightly better results than Harris due to using a different corner response function: $R = min(\lambda_1, \lambda_2)$, as compared to Harris original response function: $R = (\lambda_1 * \lambda_2) - k(\lambda_1 + \lambda_2)$.

Note that other feature detectors exist that do not detect just corners, but also blob-like features. For example the most famous blob-like feature detector is a SIFT detector. But the problem with blob detectors is that they detect blobs, which can be edges. But, as mentioned previously, edges tend to produce wrong results and are very bad locations for optical flow estimation due to the aperture problem. That is why blob detectors are not used in the project.

5.2.3 Motion Vector Estimation And Noise Filtering

This stage assumes that the drone is only moving parallel to the image plane/ground (i.e. horizontally) and does not tilt sideways. This assumption is made possible because the images from the camera are coming at a high rate (15 fps) and we also assume that the drone is hovering, keeping constant altitude, without moving fast. Due to these reasons the tilt between frames is negligible and can be ignored.

With these assumptions in place, the image motion vector is estimated to be an average of all the optical flow vectors computed from the image. More advanced techniques could be used instead of just averaging the value. For example (as described in Section 4.1.1) Parrot A.R.Drone 2.0 uses Least Squares fitting approach. But averaging still produces good results.

After getting the motion vector, it is passed through a digital first-order low-pass filter to suppress any high frequency noise. The filtered vector is then negated to convert from image motion vector to camera motion vector and is mapped to a 4D vector representing the UAV motion in the UAV coordinate frame: 3D spatial displacement vector plus rotational displacement. Note that rotational displacement and vertical component of the spatial displacement are always 0 in this particular implementation because only horizontal motion is estimated.

5.3 The Speed Controller

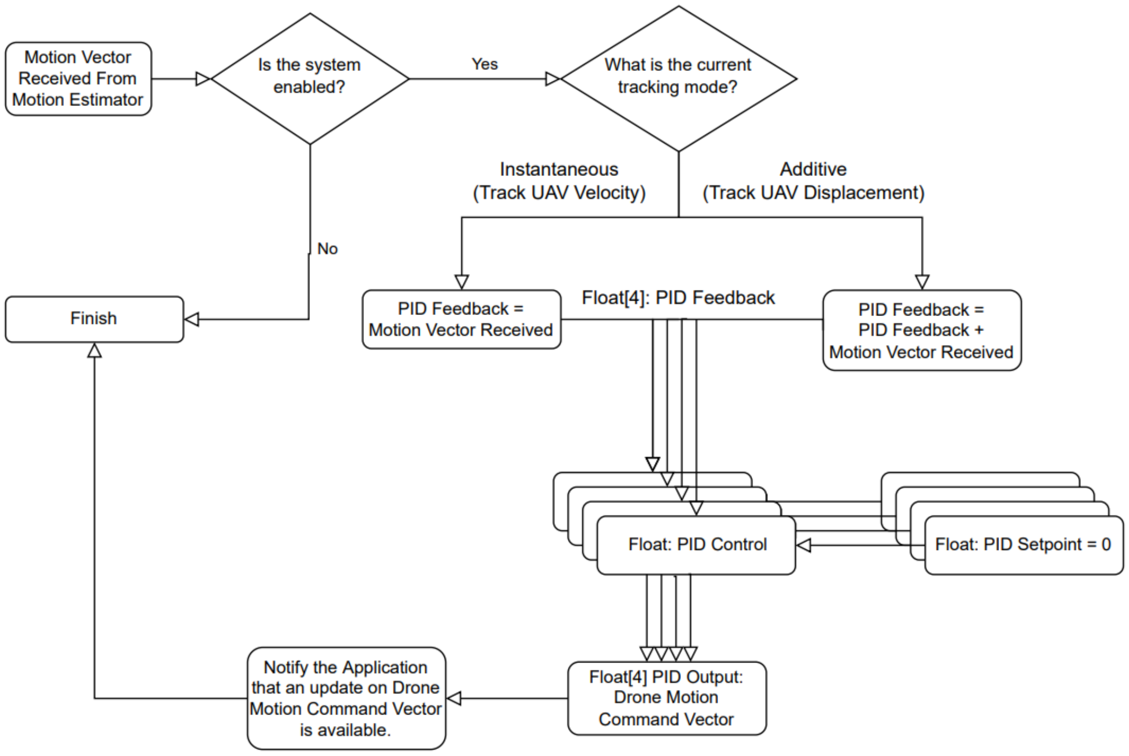

Motion Vector Received From Motion Estimator
Is the system enabled?
Yes
What is the current tracking mode?
Instantaneous (Track UAV Velocity)
Additive (Track UAV Displacement)
No
Finish
PID Feedback = Motion Vector Received
Float[4]: PID Feedback
PID Feedback = PID Feedback + Motion Vector Received
Float: PID Control
Float: PID Setpoint = 0
Notify the Application that an update on Drone Motion Command Vector is available.
Float[4] PID Output: Drone Motion Command Vector

Figure 5.3: Speed Controller Component Architecture

Figure 5.3 shows the functionality of the Speed Controller. It is a simple wrapper over four (float) PID Controllers, working independently and joined together to produce a 4D Motion

Vector Command that is then sent to the drone. One of its key features is a support of two different methods of producing feedback - **tracking modes**. Section 5.3.1 will explain the difference between the tracking modes and how they work. And Section 5.3.2 will focus on implementation of the PID control in software by utilizing a PID updating equation and discuss some practical considerations that need to be taken into account when implementing it.

5.3.1 Instantaneous Feedback vs Additive Feedback

To fully understand how the modes operate and why they work at all, it is important to understand the difference between the **value tracked (controlled) by the controller** (i.e. the system state or system output) and the **control signal output by the controller** (controller output).

The PID controller does not assume anything about the scale, range, or the units of the tracked variable. It also doesn't assume anything about the control signal (its output). The only thing it knows is that increasing the control signal (controller output) will either increase the tracked value (known as **direct acting controller**), or decrease the tracked value (known as **reverse acting controller**).

For this reason, there is no tight connection between the value we are trying to track (control) and the control signal produced in order to track the value. They do not have to represent the same thing, or have the same range of possible values. As a simple example, imagine tracking the room temperature by regulating an electrical voltage applied to the heater. In this case the tracked variable is the temperature in the room, and the control signal is a number ranging from 0V to 12V.

In the case of the drone, the control signal is a 4D vector telling the drone where and how fast it should move. The value on each axis has a range $[-1, +1]$ where 0 means no motion, $+1$ means forward motion with a maximum speed along that axis and -1 means backward motion with a maximum speed along that axis. The 4th component is used for clockwise/anticlockwise rotation around the vertical axis.

And the tracked variable in this case can be one of the following:

- The drone's speed at the current moment in time. Keeping it as close to 0 as possible means minimizing the current speed of the drone, which is a sensible thing to do, but can cause the drone to fail to return to initial position after the rapid wind starts blowing and instead start hovering over a different position.

- The drone's displacement from the point where the hovering mode was enabled. Keeping it as close to 0 as possible can be a good idea, but might fail if the Motion Estimator produces noisy results, and lead to a potential drift accumulating over time.

Since the images from the camera (hence the drone's motion estimation vector) come at a fixed frequency of 15 times per second, we can say that a displacement per image is a displacement per a unit of time (one fifteenth of a second). In that way we can treat the motion estimation vector as the drone's velocity vector, and we can treat the sum of all different motion vectors as the drone's displacement vector.

This leads to a two different tracking modes:

- **Instantaneous** - bring the displacement per image as close to zero as possible, hence track the drone's velocity.
- **Additive** - bring the sum of all the displacements as close to zero as possible, hence track the drone's displacement.

5.3.2 PID Implementation in Software

The algorithm uses a classic forward-acting PID clontrol with a low pass filter on a derivative term. It's transfer function (ratio of output to input) in a Continuous Laplace Domain is given by the Equation 5.2.

$$H(s) = (K_p) + (K_i * \frac{1}{s}) + (K_d * s * \frac{1}{s\mathcal{T} + 1}) \tag{5.2}$$

The continuous transfer function can be converted to the discrete transfer function in a Z-Domain by utilizing a Tustin Transform shown in Equation 5.3. Note that T in the formula is a "sampling time of the discrete controller in seconds", or in simple terms, the time between updates of the PID output (dt).

$$s = \frac{2}{T}\frac{(z-1)}{(z+1)} \tag{5.3}$$

The terms in discrete transfer function can then be rearranged to get an updating equation. The key point to remember here is that multiplying a value by z means a time shift forward by one sample. The final updating formula is given by Equations 5.4-5.7. Here ”[n]” stands for current sample and ”[n-1]” stands for previous sample, that is ”e[n]” means current error and ”i[n-1]” means previous output of the integral term.

$$p[n] = K_p * e[n] \tag{5.4}$$

$$i[n] = \frac{K_i T}{2} * (e[n] + e[n-1]) + i[n-1] \tag{5.5}$$

$$d[n] = \frac{2K_d}{2\mathcal{T} + T} * (e[n] - e[n-1]) + \frac{2\mathcal{T} - T}{2\mathcal{T} + T} * d[n-1] \tag{5.6}$$

$$output[n] = p[n] + i[n] + d[n] \tag{5.7}$$

Now let’s talk about the common practical issues of the PID controllers and how the implemented algorithm deals with them.

Derivative on Measurement

When a user changes the value of the setpoint to track by the controller (e.g. changing the temperature in the room), the change happens instantaneously. Since the setpoint is changed, the system’s error (error = setpoint - system value) is also changed. The derivative component amplifies this change significantly. This results in a huge rise in a control variable, but only for one iteration of the updating loop of the controller. On the next update, the setpoint is stable and the derivative term drops in magnitude, compared to the iteration when the setpoint was changed. This in turn leads to the control variable (output) to drop. The phenomenon is known as an ”output kick on a setpoint change”, and can lead to a nasty behaviour. To deal with it, PID controllers employ a technique called a ”Derivative on Measurement”. Equation 5.8 shows how the derivative term is computed.

$$\frac{\partial}{\partial t}e = \frac{e[n]-e[n-1]}{dt} = \frac{(s[n]-m[n])-(s[n-1]-m[n-1])}{dt} = \frac{(s[n]-s[n-1])}{dt} + \frac{(m[n-1]-m[n])}{dt} \tag{5.8}$$

Here e, s, m stand for "error", "setpoint" and "measurement" respectively. The kick is caused by the term $\frac{(s[n]-s[n-1])}{dt}$. It is 0 when the setpoint is constant and is high when the setpoint is changed. When the setpoint is constant, this term does not contribute to the value of the derivative. Since the setpoint is constant most of the time, the term $\frac{(s[n]-s[n-1])}{dt}$ is eliminated from calculations to avoid the kick.

The implemented PID algorithm uses the "Derivative on Measurement". However, since in the stabiliser, the PID setpoint is always fixed to be 0 by the algorithm, the kick can't happen. But the derivative on measurement is still employed, because it is commonly a good practice.

Integrator Anti-Windup via Clamping

Another common issue that can happen in PID controllers is an over-saturation of the Integral Term. Suppose the system (drone) is under such a heavy load (wind), that setting the control outputs to the highest values still cannot eliminate error (the drone still drifts away). The integral term will increase over time as it accumulates the error. When the heavy load is removed (wind stops), the system needs to react accordingly by lowering its output. But because the integral is over-saturated, the output will continue to be at a maximum value, until the system overshoots the setpoint so that error becomes negative (opposite direction) and reduces the integral term. This is known as a "Reaction Lag". The time of the lag is proportional to the time the system was under the heavy load and can be quite long. Techniques that deal with the issue are known as "Integrator Anti-Windup Methods". The simplest one is "Clamping". It the integral term from being higher than some threshold value. Other methods exist. One example is a "Back-Calculation Method". The implemented PID algorithm uses clamping as an integrator anti-windup technique.

Phil's Lab (2020) was used as a main reference point for implementing a PID control in the project.

Chapter 6

Application Design And Architecture

This chapter provides an overview of the design and implementation of the actual software application that was developed during the course of the project. The main purpose of the application is to host the stabilisation algorithm and provide a communication link between the drone, algorithm, and the user of the application.

6.1 A Choice of APIs and Libraries

While application requirements define its architecture, the used libraries can sometimes define its implementation. And this project is not an exception. This section provides an overview of the libraries used in the project and their influence on the implementation of the project.

6.1.1 Image Processing

For image processing and computer vision related tasks OpenCV library was used. It is an open source library that has bindings to all the popular programming languages, such as C++, Java and Python.

6.1.2 Communication With the Parrot UAV

I was able to find two open source libraries for communication with the Parrot drone:

1. CVDrone. It is a C++ library providing a two-way communication interface with the

drone. It allows to send a predefined set of motion, calibration and configuration commands to the drone and retrieve sensor and image data from the drone. The library includes OpenCV for image processing. The retrieved drone images come in OpenCV compatible format.

2. YADrone. It is a Java library/framework for creating applications interfacing with the drone. It is more complex than CVDrone and implements Publish-Subscribe model for retrieving data from the drone instead of polling model utilised by CVDrone. It does not include OpenCV though, which means that it should be included separately.

Although YADrone is more developer-friendly, CVDrone was used for the project due to lower video streaming latency compared to the YADrone. Minimizing feedback latency is crucial for control systems, high latency can lead to high delays in the controller reaction, or even a failure of the controller.

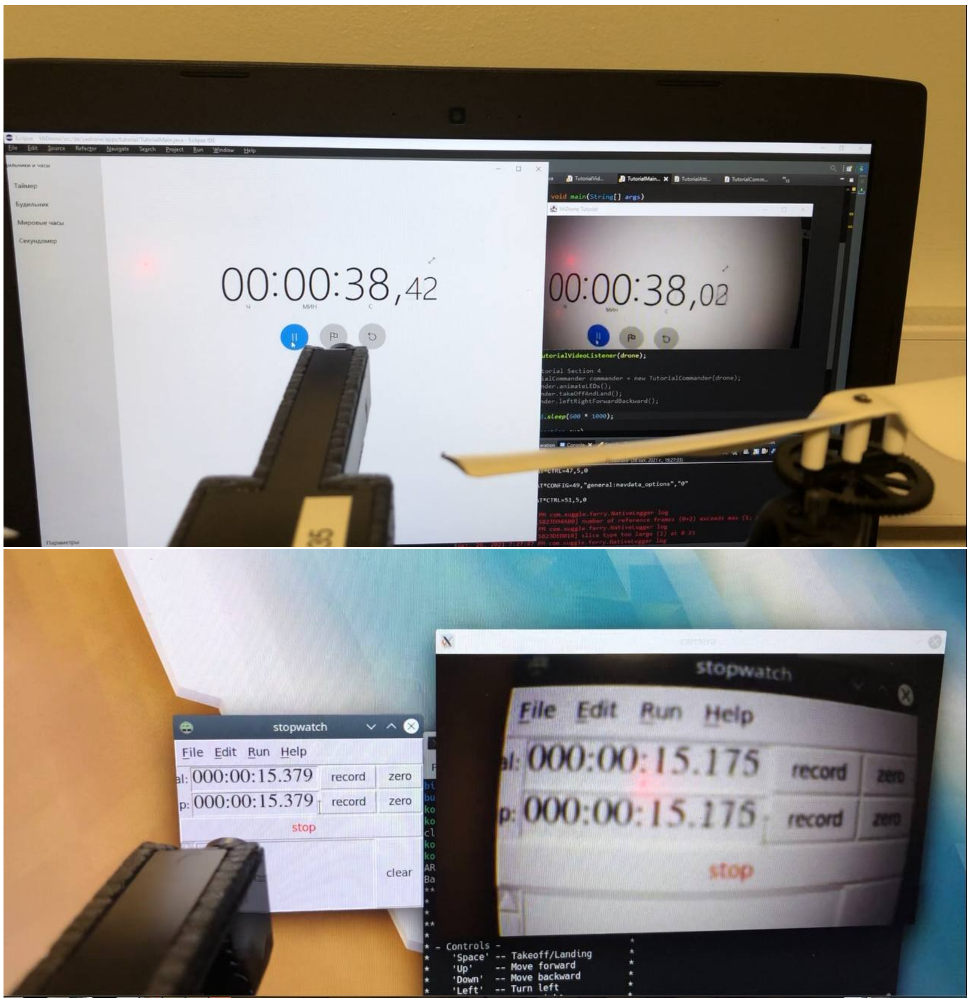


Figure 6.1: YADrone (top) and CVDrone (bottom) Latency Test

To estimate the video streaming latency a following setup was used. An on-screen timer displayed a reference time. The drone's camera was looking at that timer and the video coming from the drone was displayed on screen next to the timer. The difference in times shown is the estimated latency. Figure 6.1 shows that for YADrone library the latency is 0.4 seconds, while for CVDrone it is 0.2 seconds. It is a significant difference in favour of CVDrone.

Using CVDrone means that the project is implemented in C++.

6.1.3 Handling user input via a Gamepad

The input from the gamepad is acquired with a help of XInput C++ library provided as a part of Windows C++ libraries package. It is a low-level, polling based library with a single-function interface for polling the gamepad for its state.

6.2 Application Architecture and Components

6.2.1 Component Based Architecture

The application utilizes a **Component-based Architecture**. The functional units are separated into different components. The components are initialized and interconnected during a configuration stage at runtime. This allows for different instances of the application to provide different functionalities, when run. For example, one configuration may have the stabilisation algorithm linked, while another may not have it. More about configurations in Section 6.3.

Some of the components are **runnable**. These components are called Managers. They extend from the AbstractManager class and are required to override a method *poll()*. AbstractManager has a method *run()*, which repeatedly calls the *poll()* method (polls the manager), until a call to a *stop()* method on that component is made. The class also provides a *spawn()* method which creates a separate thread and calls the *run()* method on that thread.

The application is run by instantiating the Managers (at least one of them) and calling *run()* method on those. For simplicity and thread safety, a special class called UnionManager was created. It is a manager that holds a collection of all the managers in the application (linked at configuration stage). Its *poll()* method polls each of the linked managers in turn. Calling the *run()* method on this class allows to run all the application Managers in parallel using a single thread, thus avoiding thread unsafety issues. Examples of Managers are DroneVideoManager, AviVideoManager, GamepadInputManager, and DroneCommander. Other components are passive and cannot be run directly. Their methods are invoked by the Managers.

6.2.2 Component Linking

The application utilizes two main approaches for the component linking.

1. **Dependency Injection**. With this approach one component holds a reference to another. These components are said to be tightly coupled. The dependencies are injected at a configuration stage. An example of such relationship is a tight linking between the SpeedController class and the PID class discussed in Section 5.3.

2. **Publisher-Listener Model**. With this approach, a component can notify all its listeners when some event occurs and provide data associated with the event. This is a form of loose coupling. The publisher can have an arbitrary amount of listeners, starting from zero. For a component to become a listener, it should implement a **Listener** interface, which includes implementing the method that is called when the listener is notified. Publishers have the code to notify the listeners. The linking (subscription) process happens at the configuration stage.

6.2.3 Components

The application has a lot of components. Most of those are utility components, e.g. for visualizing data or recording the video from the drone. Only the key components, which are crucial for functioning of the application, are described in the report.

DroneVideoManager

This component is responsible for polling the drone for new images produced by the camera. It is a Manager, which means it implements a function polling the drone, which is repeatedly called when the application is running. As its dependency, it requires a handle to the drone, in order to be able to poll it, but it does not require an exclusive ownership of that handle. It acts a publisher of images and any ImageListener component can subscribe to the stream of images produced by it. Examples of such listeneres are the MotionEstimator (discussed in Section 5.2) and a DisplayAndRecordVideo component. The latter is a simple component that displays the video on the screen and can record it and store in the local file system of the computer if needed.

AviVideoManager

AviVideoManager is very similar to DroneVideoManager. But it reads the image stream from a video in the file system. The rest of the funtionality is the same as in the DroneVideoManager. The aim of this component is to allow offline operation of the components that require the video stream without being connected to the drone. An obvious example of that is testing and parameter tuning of the MotionEstimator class using recorded reference videos.

GamepadInputManager

This component is a wrapper over a low level XInput library, providing user input via an Xbox Controller. The code that uses XInput library, needs to poll the controller for any updates. Hence, this component is made to be a Manager. It is responsible for polling the connected controllers and detecting any updates (e.g. key presses). GamepadInputManager also uses Publisher-Listener model to pass the detected controller updates to other components. Any component extending from GamepadListener interface can subscribe to automatically receive any updates from the Xbox gamepad. This is how the user input is passed to the components. One example of such listeners is the DisplayAndRecordVideo component mentioned above. When it detects a specific trigger press, it starts/finishes a video recording and saves it into the file system. Another example is the DroneCommander discussed below.

DroneCommander

It is a central component representing the state machine for controlling the drone. While the DroneVideoManager is responsible for getting video data from the drone, DroneCommander is responsible for sending all the commands to the drone. To avoid race conditions and synchronization problems, DroneCommander is the only component that is allowed to send the commands. If any component wants to send some data to the drone, it should ask the DroneCommander to send that data to the drone on its behalf. The DroneCommander has two main purposes:

1. It acts a state machine, governing which component has the permission to request to send any command to the drone.
2. It acts as one of those components, which can request to talk to the drone. And this component is responsible for providing the user a manual control over the drone. It takes

input from the Xbox Gamepad and sends the corresponding commands to the drone.

MotionEstimator and SpeedController

These are the two components that make up the stabilisation algorithm. Chapter 5 provides a detailed overview of the components.

6.2.4 Linking the Stabilisation Algorithm

Now let's talk about how the Stabilisation Algorithm is linked to the application. As described in Section 5.1, the algorithm's implementation consists of the two components: MotionEstimator to estimate the UAV's speed and SpeedController to track its speed.

From the high level point of view, the whole stabilisation can be viewed a stream processing task. A stream of images from the drone goes into the stabiliser, and a stream of drone motion commands is produced by the stabiliser. The Publisher-Listener Model works ideally for this kind of linking. The link chain is described below:

- The MotionEstimator component listens to the stream of drone images produced by the VideoManager component (DroneVideoManager or AviVideoManager depending on the configuration). The component processes the images and, in turn, produces a stream of motion estimates.
- The SpeedController listens to the motion estimates published by the MotionEstimator. Based on the received data, it produces the motion commands, and stores the most recent command in a container.
- The SpeedController cannot however send the motion commands directly to the drone. Since the only component that can send commands to the drone is the DroneCommander, the SpeedController is linked to the to it via an optional dependency injection. The DroneCommander can ask the SpeedController of a new motion command, when it needs to. Note that the stabilisation algorithm can act without the DroneCommander. But in that case, it will produce the commands, which will never be sent to the drone.

6.3 Application Configurations

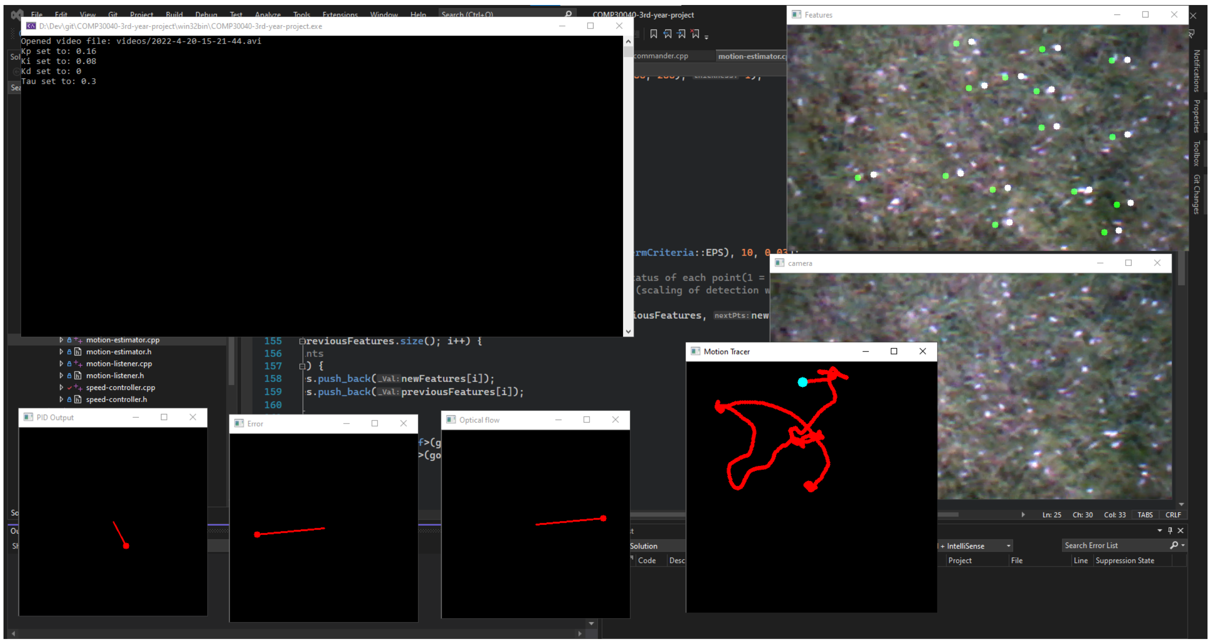
Since almost every component is independent of others, the application can be configured to be pretty much anything the user wants. And by saying "independent" components I do really mean they are independent. Even if one component requires data from another component in order for some of its functions to work, the link between the two is not compulsory. An example is the stabilisation system and its linking process described above. It can exist while only being linked to the VideoManager. In that case it will produce a stream of drone commands, but they will never be executed. In can also be only connected to the DroneCommander. In that case you will be able to enable it via the DroneCommander, but it will do nothing. Or it can exist in the application without being connected to anything at all. But this is obviously not a good idea to do.

For convenience, three different application configurations were created:

1. **Drone Flight Configuration**. It includes all the major application components. It provides functionality to control the drone via an Xbox Controller, displays (and records on demand) the video from the drone on screen, and includes a hovering mode utilizing the stabilisation algorithm.
2. **Minimal Drone Flight Configuration**. Same as the Drone Flight Configuration, but with stabilisation algorithm stripped out.
3. **File System Video Processing Configuration**. It provides a means of observing the behaviour of stabilisation algorithm without being connected to the drone. It replays the drone camera footage that is saved on the computer. More about it in chapter 7.

6.4 Application UI

User Interface is not the strongest side of the application. The application does not have a centralized UI system. Instead, the components are allowed to use data visualisation functions provided by OpenCV library. OpenCV provides means of visualizing image data and vector data by means of creating windows and drawing images inside them. OpenCV also provides means of getting input from the keyboard, which can be utilized independently by each of the components.


D:\Dev\git\COMP30040-3rd-year-project\win32bin\COMP30040-3rd-year-project.exe
Features
camera
Motion Tracer
PID Output
Error
Optical flow

```
Opened video file: videos/2022-4-20-15-21-44.avi
Kp set to: 0.16
Ki set to: 0.08
Kd set to: 0
Tau set to: 0.3
```

Figure 6.2: An Example Application UI

Figure 6.2 demonstrates the UI layout. It is typically a collection of windows presenting the data under processing and results. The windows shown in the figure include output from the drone's camera with features being tracked, current estimate of the drone's velocity, PID controller output, trace of the drone's motion path and a console with some logging information.

In general the application utilizes a decentralized UI architecture, where every component defines its own UI inputs and outputs. Its a good approach considering the flexibility of the application and its configurations and independence of its components. However, as the project growth more, it would be beneficial to switch to more advanced UI libraries. A good choice is an ImGui library, which also supports a decentralized UI architecture.

Chapter 7

Algorithm Parameter Tuning and Application Testing

This chapter focuses on how the correctness of application and implemented algorithms were tested during the development phase. It also describes how the algorithm parameters were tuned in order to maximize the accuracy and robustness of the implemented algorithms. The chapter also covers some of the failed attempts of implementing the stabilisation system.

7.1 Testing the Code

Unfortunately, the project does not employ any advanced testing techniques, such as automated unit/integration testing. The code was tested manually by building and launching the required configuration and observing performance of each of the components. This approach was manageable due do a relatively small size of the project and the fact that only a single person was working on it. This approach was generally efficient and helped fixing lots of the errors and bugs.

7.2 Parameter Tuning Framework

In order for the stabilisation algorithm to work, it requires a lot of parameters to tune. Its accuracy, robustness and response delay depend on the values of the parameters. Whilst there are guidelines to tuning some of the parameters used in the algorithm, the actual performance and the best parameter values highly depend on the application at hand. And the best tuning process for this specific algorithm, that I'm aware of, is a **guided interactive tuning**. Guided means that changing the values is based on some commonly known and established theory in the field. Interactive means ability to change parameters in real time and observe any effects caused by the change in real time.

To create this interactive framework, I gave the individual components an ability to listen to user input and regulate the parameters based on detected user key presses.

At this stage I've implemented an AviVideoManager and DroneVideoRecorder (described in Section 6.2.3). They allow to record videos from the camera of the drone and replay that same footage later during fine-tuning of the motion estimator. Using the same footage repeatedly ensures reliable and accurate evaluation and comparison of performance of each of the parameter configurations. And that also meant that I didn't require to connect the drone every time I wanted to test out the image processing algorithms.

I've also implemented some utility components which allowed for better observation of the behaviour of the algorithms. These components include interactive vector visualizers, motion estimates tracers, and others.

7.3 Tuning a Motion Estimator

Tuning the Motion Estimator happened by utilizing a video recorded by the drone bottom camera during flight. The video was replayed and the parameters were tuned interactively.

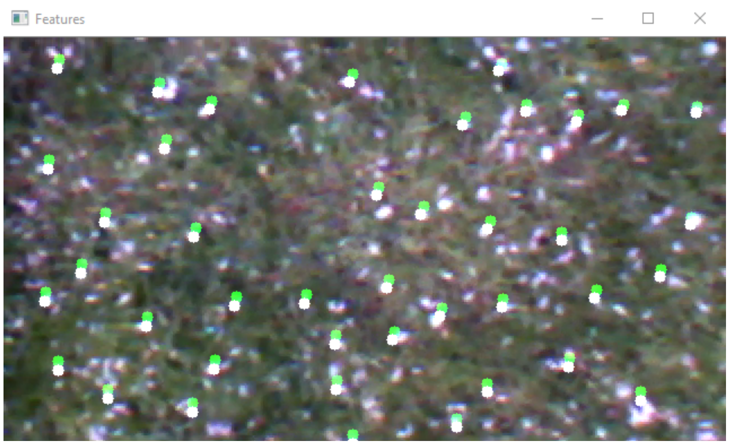


Figure 7.1: An example image used for tuning parameters

Figure 7.1 shows an image from the video used to optimize the parameters and also the features detected and tracked in the image (white and green circles). Note that this image has quite a good texture, which is not the case for some of the areas where the drone flies.

A general strategy for optimizing the parameters was to create a noise-free (or noise-minimized) local feature tracking system, that is able to precisely detect motion at small scale. That however means that the algorithm is not able to track high speed motion.

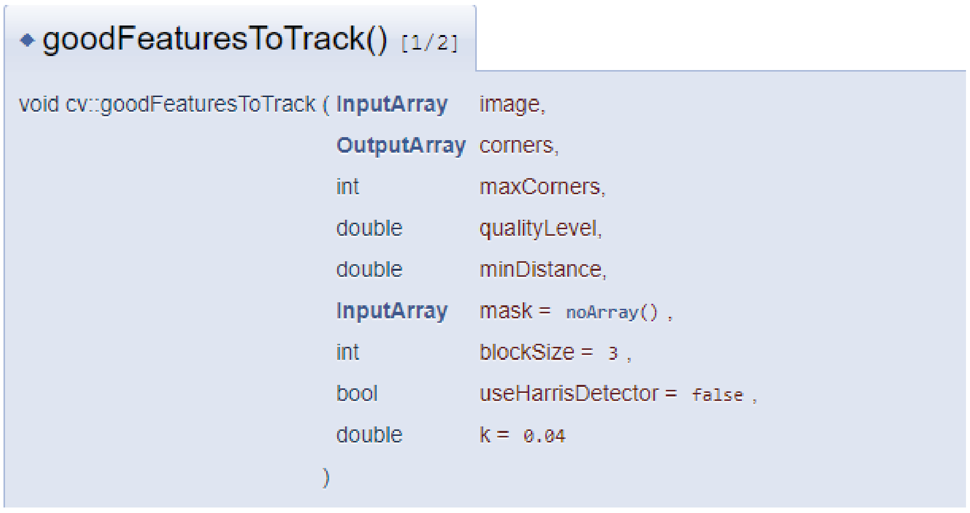
As described in Section 5.2, Motion Estimator uses two main image processing algorithms:

- Shi-Tomasi Corner Detector.
- Lucas-Kanade Optical Flow Algorithm.

OpenCV library provides implementations for both of these. The following subsections will provide an explanation of each of the parameters they require, and how the corresponding values were found.

7.3.1 Tuning a Shi-Tomasi Detector

OpenCV provides an implementation of Shi-Tomasi corner detector via a function called *goodFeaturesToTrack()*.



```
◆ goodFeaturesToTrack() [1/2]

void cv::goodFeaturesToTrack ( InputArray   image,
                               OutputArray  corners,
                               int          maxCorners,
                               double       qualityLevel,
                               double       minDistance,
                               InputArray   mask = noArray() ,
                               int          blockSize = 3 ,
                               bool         useHarrisDetector = false ,
                               double       k = 0.04
                             )
```

Figure 7.2: Declaration of *goodFeaturesToTrack()* function

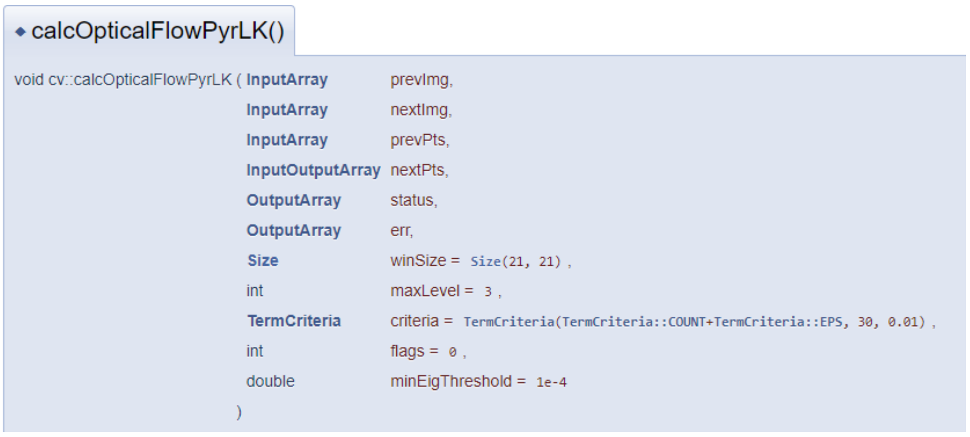
Figure 7.2 shows a declaration of the *goodFeaturesToTrack()* function. The parameters to be tuned are:

- **maxCorners**: a maximum amount of corner locations to return. Value used: 0 (meaning unlimited).

- **qualityLevel**: defines a lower threshold on the corner response function for the corresponding location as a ratio of the maximum found corner response function. The lower threshold is equal to the product of the maximum found corner response value and the quality level. Value used: 0.4.
- **minDistance**: a minimum euclidean distance between detected corner locations. Value used: 40.
- **blockSize**: a size of block around the feature location that is used to compute a corner response function. Value used: 7.

Since the ground under the drone is mostly uniform, the corner detector should detect only really distinct/unique features, which are also far enough from one another to eliminate ambiguous matches during optical flow estimation. Due to this reason both qualityLevel and minDistance were set to relatively high values.

7.3.2 Tuning a Lucas-Kanade Optical Flow Algorithm



◆ calcOpticalFlowPyrLK()

```
void cv::calcOpticalFlowPyrLK ( InputArray          prevImg,
                                InputArray          nextImg,
                                InputArray          prevPts,
                                InputOutputArray    nextPts,
                                OutputArray         status,
                                OutputArray         err,
                                Size                winSize = Size(21, 21) ,
                                int                 maxLevel = 3 ,
                                TermCriteria        criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01) ,
                                int                 flags = 0 ,
                                double              minEigThreshold = 1e-4
                              )
```

Figure 7.3: Declaration of Lucas-Kanade Optical Flow function

Figure 7.3 provides a declaration of the function implementing Lucas-Kanade optical flow. The parameters to be tuned here are:

- **winSize**: size of the window around the tracked point in which the algorithm will look for the new position of the point. Chosen value: 9x9.
- **maxLevel**: size of the scaling pyramid. Chosen value: 4

Lucas-Kanade method implements a coarse-to-fine technique to track the points. That means it will subsample the image to lower scale and find the tracked point in the winSize region around the previous location in that subsampled image. The found optical flow vector is then used to reduce the range of search at higher scale. MaxLevel parameter defines a number of times the image will be subsampled. So the product of winSize and maxLevel (36x36 in this case) is an effective size of local region around the keypoint where the algorithm will look for a new position of that keypoint. 36x36 is a small region compared to a size of the image (640x360). This implies that the algorithm cannot detect high speed motion. However the algorithm copes well with small-speed motions.

7.4 Tuning a PID Controller

Tuning a PID controller involves tuning 3 parameters:

1. Proportional Gain.
2. Integral Gain.
3. Derivative Gain.

Each of these parameters defines an amount of contribution of each of the respective components of the PID controller to the final controller output.

Unlike motion estimation, PID is a closed loop reactive system. Its next output depends on how the environment reacts to its previous output. Due to this reason, the component cannot be tuned in an offline manner (like the motion estimator was tuned). It must be tuned while being a part of the whole and acting system. A common solution to this problem is to design a simulator of the system. There are online simulators for simple electrical systems, but there is no such simulator for A.R.Drone 2.0. Since writing a whole drone and wind physics simulator is unfeasible and is not part of the project, I had to tune the system in real life environment while flying actual drone. Unfortunately due to time and resource constraints I was not able to find a good enough set of parameters to make it quick, stable and robust at the same time. But more about this in Chapter 8.

Although PID controller is a very well studied controller, there is no strict, science-based methodology of deriving the best parameters for its operation. There are however guidelines to how to tweak the parameters in order to achieve good results. The methodology used in this project is described by *PID Explained Team* (2018). The methodology involves the following steps:

1. Start by setting all the gains to 0.
2. Repeatedly increase the proportional gain until the system becomes just about unstable. An unstable system is the system, which value fluctuates around some value, but never converges to that value.
3. When the system becomes unstable, record the value of the proportional gain. Divide the value by 2. The result is the value of proportional gain that you should use.
4. Repeat the same process with an integral gain (but now the proportional gain is not 0).
5. The next stage is to fine tune the derivative gain and also fine tune the combination of the all three gains. At this point you should observe the results and try to identify the best one.

7.5 Failed Attempts

As mentioned in Section 5.2, after tracking the features and their optical flow vectors, the drone's motion is estimated by simply averaging all the optical flow vectors. I tried to implement more complex and more accurate (in theory) solutions, but they all failed in real life environment. One of the approaches that I've tried, was a 3D pose and motion recovery via computing an Essential Matrix. An Essential Matrix is a transformation matrix telling how the camera moved between frames. It is computed from a number of point correspondences in the two images. However, as mentioned in Section 3.3.3, in order for it to work properly, there should be a reasonable displacement (motion) between the pair of images. When called on pair of images with a tiny displacement (like 15 pixels displacement between 640x360 pixel images), it produces a degenerate result. A solution was to track the same point over several frames and call the algorithm with a period of say 5 frames. However this approach also failed, because after fine tuning of the feature tracking parameters, it turned out that the algorithm cannot track even the minimum amount of points required for the pose recovery over such a big number of frames. Another approach was to assume that the detected image features are located on the same plane (since the ground under the drone is horizontal) and use a planar homography transform in order to figure out a motion between the two images. The approach also failed due to exactly same reasons as the 3D pose recovery approach using essential matrix.

Chapter 8

Stabilisation Algorithm Evaluation

This chapter provides an analysis of the stabilisation algorithm performance. The evaluation of the algorithm will be conducted by utilizing a set of performance metrics defined in Section 8.1. Section 8.2 describes the performance of the developed algorithm in terms of these metrics and compares it to the performance of the algorithm used natively by Parrot A.R.Drone. And finally, Section 8.3 will analyse the accuracy of the motion estimation technique used.

8.1 Evaluation and Performance Metrics

The project defines three performance metrics that are used to assess the quality of the stabilisation algorithm:

1. **Convergence of motion**. An ability of the drone governed by the algorithm to eventually converge to a desired static state. The desired static state can be defined as a fixed 3D position of the drone in the world that is equal to its position at the moment when the stabilisation system was enabled. The drone should arrive to that position and eventually stay there without drifting away.
2. **Speed of reaction**. A time it takes for the algorithm to adjust its output as a fact of reaction to a significant change in the external forces acting on the drone. Changes in external forces affect the velocity of the drone and it drift. Ability to react quickly minimizes the drift. Measured in seconds.
3. **Maximum Displacement Deviation**. A maximum allowed magnitude of displacement

from the desired static positional location of the drone at any particular point in time. Measured in meters.

8.2 Stabilisation Performance Analysis

8.2.1 PID Configuration

Chapter 7 lists the values of parameters used for motion estimation, however it omits listing parameters used by the PID contoller. Due to the lack of time, I was unable to find a universally good set of parameters which provided a robust and quick control under both windy and non-windy weather conditions. This is still an ongoing process.

The algorithm was tested using a PI controller with velocity tracking mode (described in Section 5.3.1). Note that due to the mathematical relationship of velocity and displacement (displacement is an integral of the velocity), PI controller on velocity is the same as PD controller on displacement. This is why I've tuned parameters mostly using the velocity tracking mode.

The parameters used for PID controller of the system under test are:

- Proportional Gain: 0.160
- Integral Gain: 0.005
- Derivative Gain: 0.0

8.2.2 Performance

Table 8.1 summarizes the performance of the developed stabilisation algorithm in terms of metrics defined in Section 8.1. It also provides a comparison to the Parrot A.R.Drone 2.0 native stabilisation algorithm. Note that the speed of wind during the test was roughly 15km/h.

Table 8.1: Stabilizers Performance Comparison.

Performance Metric	Developed Algorithm	Native Algorithm
Convergence of motion	Converges	Converges
Speed of reaction (seconds)	1 - 2	< 1
Maximum Displacement Deviation (meters)	≈ 2	≈ 0.3

The data suggests that the developed algorithm reacts too slowly to the changes in environment compared to the native stabilizer. The native stabilizer is quite reliable do to its highly accurate motion estimation system and also precisely tuned PID parameters.

Now let's break down what actually happens inside the drone. The proportional controller keeps the speed of the drone close to zero, but not zero. This implies that drone starts drifting away, and during the tests it actually drifted in the direction of the wind. This implies that the proportional gain is a bit low and should ideally be increased by some amount. After the drone drifts away from the tracked position, the integral value (accumulated displacement) increases and the drone then flies back. This happens after a relatively long drift (1 meter, and in worst cases up to 2 meters), which means that the value of the integral gain is too small and should also be increased. But it still remains unknown by how much the values should be changed. This is why tuning a PID controller is a hard task and can take a lot of time.

However, the developed algorithm has achieved its main high level goal of being able to maintain roughly the same position in the air.

8.3 Evaluation of the Motion Estimation Algorithm

The success of the stabilizer depends heavily on the accuracy of the estimation of the drone's motion. If the drone cannot correctly estimate how it moves, it will not be able to correctly stabilize itself. This is why Parrot A.R.Drone 2.0 has a complex motion estimating system (described in Section 4.1), fusing the results produced by different sensors and components.

Due to the lack of illumination the implemented motion estimation algorithm fails in indoor spaces, and only work in well illuminated, outdoor spaces. This is one of the reasons why one cannot always rely only on computer vision for the motion estimation.

In order to analyse accuracy of the implemented motion estimation technique when used in outdoor setting, the following experiment was conducted:

- Key points (objects) were laid out on the ground in a predefined arrangement.
- The drone was moved between the key points in a predefined path, eventually returning to its starting position.
- Drone's motion estimation was traced, and final error recorded.

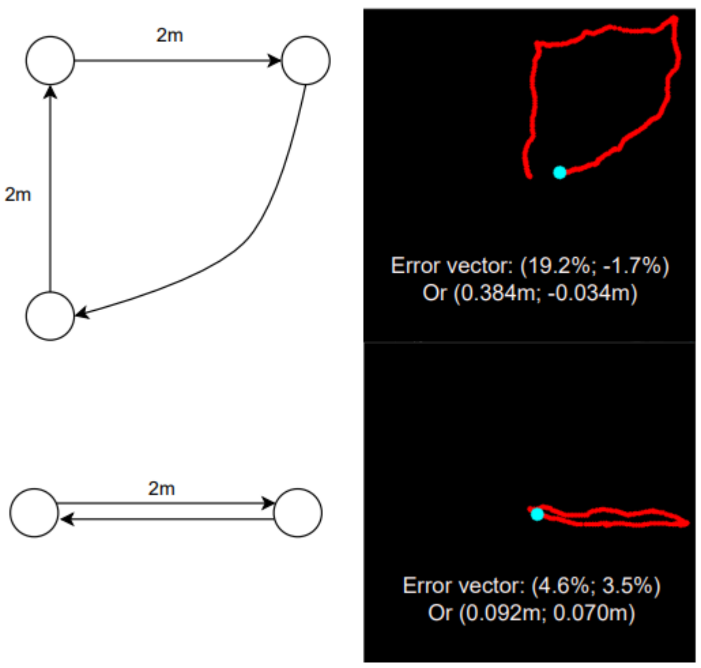


Figure 8.1: Results of Motion Estimation Tests

Figure 8.1 shows the number and layout of the key points on the ground. On the left hand side of the image you can see a sketch of the true path that the drone took. On the right hand side, the estimated drone's motion together with the error in the estimates are shown.

The figure shows that the error accumulates pretty quickly, which can be a potential problem for the stabilizer. In the worst case (shown at the top of Figure 8.1), the accumulated error was equal to 0.4m only after the motion for about 7 meters. This can cause a significant drift of the drone and means that more accurate motion estimation techniques should ideally be employed.

Chapter 9

Conclusions and Future Work

9.1 Was the Project Successful?

To answer this question, let's refer to the key project requirements and projects success criteria described in Chapter 2. **Application must (key requirements):**

1. Run on a windows machine. - **Satisfied.**
2. Provide a user with an interface to control the motion of the drone in a user-friendly manner. - The user friendliness of the UI is debatable. **Satisfied to an extent.**
   (a) A video from the UAV camera should be displayed to the user to improve quality of experience. - **Satisfied.**
   (b) In input from the user must be acquired via a convenient input device which supports a precise control over the UAV. An example of such input device is a wireless gamepad (e.g. wireless xbox controller). - **Satisfied.**
3. Implement a drone hovering mode. - **Satisfied to an extent.**

   The hovering mode must:

   (a) Correctly function outdoors in good lighting conditions at a height ranging from 1 meter to 5 meters above the ground. - The algorithm passed functional tests at an altitude range from 1m to 3m. Higher altitudes were not tested. **Satisfied to an extent.**

(b) Keep a drone stable in a specified position in the 3D world without user intervention. - The algorithm works completely without user intervention. **Satisfied.**

(c) Long term spatial deviation from the specified position must be minimised. - With occasional spatial deviation, the drone managed to track a stable position. **Satisfied.**

(d) The maximum short term spatial deviation from the specified position due to rapid wind changes of at most 1 meter in any direction. - Short term spatial deviation can sometimes be higher than 1 meter. **Failed.**

Although some of the application requirements were not fully achieved, I still consider the project to be successful. The developed application is a good proof of concept and demonstrates the usefulness of computer vision in stabilising the UAV. With more time spent tweaking parameters and optimising the algorithm it can be made even better and more robust. Section 9.2 provides a list of potential future improvements of the stabilisation algorithm.

However, the project wasn't only about the stabiliser. It was also about creating an environment in which the stabiliser can function. Having started as an augmented application, with only purpose to enable operation of the stabiliser, it has grown to a powerful, independent application/framework, enabling numerous high level drone applications to be developed. Section 9.3 talks about a potential future of the developed application and steps that should be taken in order for that future to be possible.

9.2 Potential Improvements of the Stabiliser

Improving the stabiliser can be achieved by:

1. Finding better parameters for the PID controller. This is a tedious process, and in order to properly do it, a simulator needs to be utilized. However I would say that by continuing real-life tuning, it is possible to find better parameters. In fact, with current parameters, the drone is a little bit slow in terms of reaction to changes in its displacement, when tracking speed. This suggests that increasing the integral gain of the PID controller in speed tracking mode can make the performance better.

2. Improving the accuracy of the motion estimation algorithm. Section 8.3 mentions that motion estimation algorithm is not 100% accurate and accumulates error over time. In

order to eliminate the error, SLAM based techniques, such as loop detection can be utilized. This is definitely an area worth looking into.

9.3 Future of the Main Application

The developed application features a highly configurable environment of independent components and a central drone controller. It is easily extensible with new functionality and serves as a good framework for creating an advanced component based drone management system. It allows for creating an ecosystem of independent components, each of which can access drone's data and request an exclusive control over it if needed. However there are flaws in some of its design elements, resulting in the application being not very developer-friendly. To eliminate these flaws, the code needs to undergo a significant refactoring stage. Below is the list of refactoring changes that should be considered.

1. Make the Publish-Listener Dependency Injection implementation generic. Currently the model is not generic. This means the developer needs to overload the publish/listen code for every different type of data it wants the system to be able to handle. This results in code repetition and can lead to bugs later on. Using C++ templates is required in order to fix the problem and avoid code duplication.

2. Make DroneCommander state machine to be configurable. Section 6.2.3 mentions that the DroneCommander component acts as a state machine determining which component can request to send commands to the drone. Currently the configuration of this state machine (states and transitions between them) is hardcoded into this component. This implies that it is tricky to add a new component to the state machine. However the application is intended to be easily extensible. So the DroneCommander needs to be refactored to accommodate the extensibility of the application.

3. Switch to a proper UI library, ImGui. As the application continues to grow, the demand for advanced and powerful UI features will emerge. Using a proper UI library is a key in enabling good user experience, as well as good developer experience.

4. Add an automated testing system. As the application continues to grow, it will become hard to test manually. Automated testing is a key for delivering a reliable application.

References

Dávid Dezső and Kornél Sarvajcz (2018). Measuring drone flight-stability using computer vision. URL: `https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiU5Me_io_4AhVNQ0EAHS_FDa0QFnoECAUQAQ&url=https%3A%2F%2Fwww.aaai.org%2Focs%2Findex.php%2FFLAIRS%2FFLAIRS15%2Fpaper%2Fdownload%2F10398%2F10299&usg=AOvVaw3ZkcvTSAWnOZ6kkBwifaJV` (visited on 05/20/2022).

Devopedia (2022). Quadcopter. URL: `https://devopedia.org/quadcopter` (visited on 05/20/2022).

DroneBlog (2022). What Are GPS Drones, and Why Does It Matter. URL: `https://www.droneblog.com/what-are-gps-drones-and-why-does-it-matter` (visited on 05/20/2022).

Electronics Coach (2022). Closed-Loop Control System. URL: `https://electronicscoach.com/closed-loop-control-system.html` (visited on 05/20/2022).

Fintan Corrigan (2020). Drone Gyro Stabilization, IMU and Flight Controllers Explained. URL: `https://www.dronezon.com/learn-about-drones-quadcopters/three-and-six-axis-gyro-stabilized-drones` (visited on 05/20/2022).

First Principles of Computer Vision (2021). Lucas-Kanade Method — Optical Flow. URL: `https://www.youtube.com/watch?v=6wMoHgpVUn8` (visited on 05/20/2022).

H. Teimourian, K. Dimililer, F. Al-Turjman (2020). Drones in Smart-Cities: Chapter 10 - Physics of stabilization and control for the Drone's quadrotors. URL: `https://reader.elsevier.com/reader/sd/pii/B9780128199725000100?token=C0D5CCE86E1024E8E356622B8DD4E316CCC2`
`originRegion=eu-west-1&originCreation=20220601113722` (visited on 05/20/2022).

Haydar A.Kadhima, Waleed A.Araheemah (2019). A Comparative Between Corner-Detectors (Harris, Shi-Tomasi and FAST) in Images Noisy Using Non-Local Means Filter. URL: `https://iasj.net/iasj/download/a1ac58d172a482a9#:~:text=In%20addition%2C%20Shi%2DTomasi%20corner,to%20noise%20than%20Harris%20%26%20FAST%20.` (visited on 05/20/2022).

Heri Purnawan, Mardlijah and Eko Budi Purwanto (2017). Design of linear quadratic regulator (LQR) control system for flight stability of LSU-05. URL: `https://iopscience.iop.org/article/10.1088/1742-6596/890/1/012056/pdf` (visited on 05/20/2022).

J. Skoda, R. Bartak (2015). Camera-Based Localization and Stabilization of a Flying Drone. URL: `https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiU5Me_io_4AhVNQ0EAHS_FDa0QFnoECAUQAQ&url=https%3A%2F%2Fwww.aaai.org%2Focs%2Findex.php%2FFLAIRS%2FFLAIRS15%2Fpaper%2Fdownload%2F10398%2F10299&usg=AOvVaw3ZkcvTSAWnOZ6kkBwifaJV` (visited on 05/20/2022).

Kong Wai Weng (2012). PID for Embedded Design. URL: `https://tutorial.cytron.io/2012/06/22/pid-for-embedded-desig` (visited on 05/20/2022).

Matlab (2020). H Infinity and Mu Synthesis — Robust Control. URL: `https://www.youtube.com/watch?v=kRt7H0k8A4k` (visited on 05/20/2022).

Mr. G. Pradeep Kumar M.E., B. Praveen, U. Tarun Nisanth, M. Hemanth (2017). Development of Auto Stabilization Algorithm for Uav Using Gyro Sensor. URL: `https://www.ijert.org/research/development-of-auto-stabilization-algorithm-for-uav-using-gyro-sensor-IJERTCONV5IS09057.pdf` (visited on 05/20/2022).

MyDearDrone Blog (2022). Drone Uses and Applications. URL: `https://mydeardrone.com/uses` (visited on 05/20/2022).

Phil's Lab (2020). PID Controller Implementation in Software. URL: `https://www.youtube.com/watch?v=zOByx3Izf5U&t=1126s` (visited on 05/20/2022).

PID Explained Team (2018). How to tune a PID Controller. URL: `https://www.youtube.com/watch?v=dZ8lzDi3cXY` (visited on 05/20/2022).

Pierre-Jean Bristeau, François Callou, David Vissière, Nicolas Petit (2011). The Navigation and Control technology inside the AR.Drone micro UAV. URL: `https://www.asprom.com/drone/PJB.pdf` (visited on 05/20/2022).

RiseAbove Blog (2022). UAV Applications and Uses. URL: `https://riseabove.com.au/pages/uav-applications-and-uses` (visited on 05/20/2022).

S. Piskorski, N. Brulez, P. Eline, F. D'Haeyer (2012). A.R.Drone Developer Guide. URL: `https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf` (visited on 05/20/2022).