Design of control system for UAV-based video recording and tracking

Ludvig Langebro

Baldomero Puche Moreno

Lund University

Department of Automatic Control

MSc Thesis
TFRT-6105
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

The core objective of this project was to design a controller for a drone (or quadcopter) that would allow the drone to keep a target in the center of a video filmed by a camera mounted on the drone. This was done by controlling the pitch of the gimbal, on which the camera was placed, and the yaw of the entire drone combined with using an image recognition program that could identify the target. For this project the DJI Mavic 2 Zoom was used. Although designing a PID controller was an important part of the project it was relatively easy and emphasis was instead put on system identification and minimizing the impact of a large delay. The large delay was present in sending images from the drone to the controller and could be as long as 1 second. A Smith predictor was used in order to reduce the influence of the measurement delay in the control while a Kalman filter was used in order to estimate the position of the target based on the old measurements. The system consisted of a computer connected wirelessly to a mobile phone, and the drone connected wirelessly to the remote controller which in its turn was connected by wire to the mobile phone. The end result was a system that could keep a moving target, with constant velocity, well in the center of the video but suffered from overshoots for a target making quick turns.

Sammanfattning

Huvudsyftet med detta projekt var att designa en regulator för en drönare som skulle tillåta drönaren att hålla ett mål i centrum av en video, filmad av en kamera monterad på drönaren. Detta gjordes genom att reglera kardanupphängningens vinkel i vertikalled samt reglera hela drönares vridning i horisontalled, kombinerat med ett bildigenkänningsprogram som kunde identifiera målet. I detta projektet användes drönaren DJI Mavic 2 Zoom. Även om designen av en PID-regulator var en viktig del av projektet så var den relativt enkel och vikten av projektet placeras i systemidentifiering samt förmågan att minimera påverkan av en stor tidsfördröjning i systemet. Den stora tidsfördröjningen uppstod mellan drönaren och regulatorn i datorn, där tidsfördröjningen kunde bli så stor som en hel sekund. En Smith-prediktor användes för att reducera påverkan av tidsfördröjda mätvärden i regleringen medan ett Kalman-filter användes för att estimera positionen av målet baserat på fördröjda mätvärden. Systemet bestod av en dator trådlöst kopplad till en mobil telefon, och en drönare trådlöst kopplad till en handkontroll trådat kopplad till mobilen. Slutresultatet var ett system som kunde följa ett mål med konstant hastighet, och behålla målet väl i centrum av videon. Däremot, som resultat av Kalman-filtret kunde kameran få översläng vid snabba stopp och svängar av målet.

Preface

This Master Thesis was carried out during the second semester of the academic year 2019-2020 in Lund, Sweden. The Company Sony Nordic (Sweden) has a division called Flying Camera Solutions, where the intention was to have a quadcopter drone that uses a camera to film and follow a skier or snowboarder. A remote computer was used to control the drone. Help was needed in keeping the target centered in the screen and so a Thesis Project was offered to help to solve this issue.

Acknowledgements

To our families for being a constant support our whole lives and especially during these university years.

To Anders Robertsson for all the help, patience, late-night zoom meetings and time he spent with us on this project.

To the whole Sony team: Johan Hergertz, Micke Berglund, Fredrik Persson, Mathias Franzén, Jonas Berglund and Kristian Dreher for their pleasant treatment as we could feel part of the team and their availability to answer any single question to us.

Contents

1

Introduction

1.1 Objective

The main objective of the project was to design a controller that could keep a moving target centered in the video filmed by the drone. The problems that were thought to be required to be solved were handling varying speeds of the skier, handling of strong wind, handling the large delay of the system and therefore also creating a prediction model. Considering how all objectives would rely on the model of the system a good model would be required. The success of the project would be determined by the ability to keep a target with constant speed centered on the screen.

1.2 Drone

The drone that was used for this project was the Mavic 2 Zoom shown in Figure 1.1, a quadcopter developed and released by DJI in August 2018 [1]. It has a weight of 905 grams, and a flight time of 31 minutes with no wind flying at 15.5mph (25km/h) speed. The maximum forward flight speed is 45mph (72km/h) in Sport mode although for this project the drone had to be set in Positioning mode, which allows a maximum forward speed of 31mph (50km/h) and a maximum backward flight speed of 27mph (43km/h). The drone had a maximum tilt angle of 35° and a maximum angular velocity of 200°/s in horizontal direction.

The Mavic 2 has a flight distance of 18km (11 miles) and a maximum height of 5km (3.7 miles) above the sea. The navigation satellite system that it uses is GPS and GLONASS, which allows the drone to connect to multiple satellites. This dual satellite connectivity is used to assist with the precision flying, return to home functionality, obstacle avoidance or waypoints, for example. This model is not waterproof, can withstand wind speeds up to 24mph (38km/h) and temperatures from -10ºC to 40ºC.

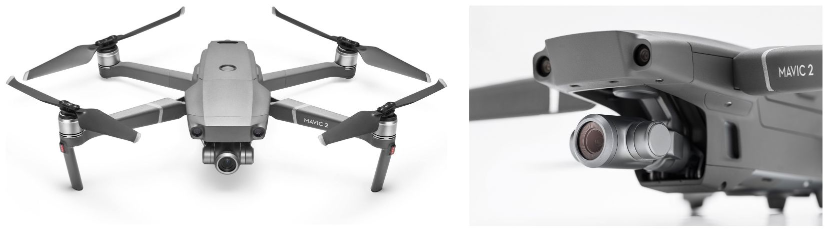

Figure 1.1 DJI Mavic 2 Zoom [2] (left) and close-up of controllable gimbal [3] (right)

Camera

The lens of the Mavic 2 Zoom has a field of view at approximately 83º horizontally and 48º vertically. The video resolution can be in 4K (3840x2160p), 2.7K (2688x1512p) or FHD (1920x1080p). In addition, the electronic shutter speed can be set from 8s to 1/8000s and the still image resolution is 4000x3000p, which means 12 million pixels.

Gimbal

The 3-axis gimbal stabilizes the camera when its mount (i.e. the drone) is moving. The motor controllers receive gyroscope information from the gimbal to compensate for rotational movements of the mount. It has a controllable tilt range from -90º to +30º, although mechanically the tilt range is from -135º to 45º and the roll range is from -45º to 45º. The maximum speed control of the tilt is 120º/s and it has an angular vibration range of $\pm$ 0.005º, which assures that when the drone is still it will not divert more than that.

1.3 Remote Controller and Mobile SDK

The remote controller shown in Figure 1.2 has a transmission frequency between 2.4-2.483GHz and an operating temperature range from 0ºC to 40ºC. A mobile phone is connected to the remote controller by a wired link. It can be micro-USB or USB to micro-USB, USB-C or Lightning, depending on the phone which will be connected. The schema is explained in Figure 1.3.

DJI has an SDK (Software Development Kit), designed to give developers access to the capability of DJI's aircraft and handheld camera products, for different devices. DJI developed SDKs for being used in different platforms in mobile phones, User Experience (UX), onboard the drone, payload the drone carries and Windows [5]. The best solution would be onboard so that no connection with another device would be needed. The problem was that DJI only has this SDK for certain model drones, the Matrice Series, and these drones are too big for this pur-

Figure 1.2 Remote Controller of DJI Mavic 2 Zoom with a Mobile Phone [4]

pose along with the camera quality not being good enough. Then the next solution would be the Windows SDK, but this has too few features for what was needed. Therefore, in the project the Mobile SDK was used because of all the available features. Although this requires a connection to the drone followed by a connection to a Mobile phone and finally a connection to the computer.

There are many features that can be used, for example, flight control that can be done manually with virtual stick commands and by missions, the reception of aircraft's state through telemetry and sensors such as GPS position, compass, barometer, IMU, flight velocity and altitude. It has readings available at up to 10 Hz through the Mobile SDK, obstacle avoidance, camera and gimbal control, the exposure and parameters of the image and video and the direction using the gimbal, live camera video feed, remote access to media stored (SD card or solid state drive), and predefined missions such as Waypoint, HotPoint and FollowMe [6]. This following mission only allows a person to be followed from a certain position, for example laterally or from the back.

The Mobile SDK includes a library that can be imported into an Android or iOS app that gives access to the DJI product. In this case the app was developed (by the Sony team before the project was started) for Android systems within Android Studio. Furthermore, an aircraft simulator and visualization tools exist for the chosen drone.

Flight Control

In order to better understand how the flight of the drone is controlled, the coordinate systems and the attitude of the drone are explained.

Coordinate systems In the DJI Mobile SDK there exist two coordinate systems, one relative to the aircraft body (body frame) and the other relative to the ground (world frame), see Figure 1.4.

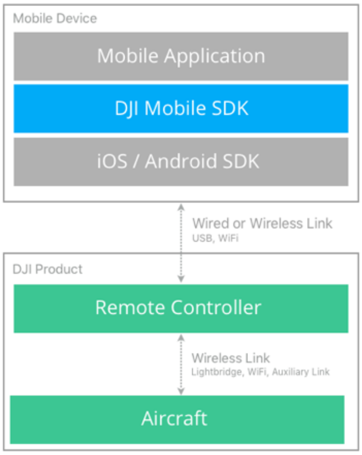

Figure 1.3 Connection from the Mobile Device to the Aircraft [6]

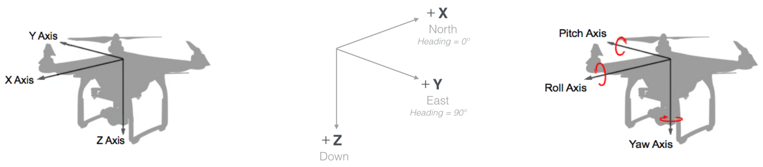

Figure 1.4 Body Coordinate System translation axes (left), Ground Coordinate System axes (center) and Body Coordinate System rotation axes (right) [6]

The body coordinate system is defined by the three axes so that the origin is located in the center of mass, the X axis is directed through the front of the aircraft, the Y axis through the right of the aircraft and the Z axis through the bottom of the aircraft, using the coordinate right hand rule. Therefore, the positive translation movements are considered forward, right and downward for the X, Y and Z axes, respectively. The aircraft rotation is also described with the same axes, defining the direction positive rotation using the coordinate right hand rule. When describing rotational movements, the X, Y and Z axes are renamed Roll, Pitch and Yaw, respectively.

The ground or world coordinate system aligns the positive X, Y and Z axes with the directions of North, East and down respectively. This convention is called North-East-Down or NED. The axes movements follow the right hand rule, so for instance a heading angle of 0º will point toward the North and +90º toward the East.

In this thesis work the yaw is set in the ground coordinate system and the roll and pitch are set in the body coordinate system, although as a default it is set in the ground coordinate system.

Attitude and Flight Attitude is the orientation of the aircraft and its camera gimbal and it is defined by the rotation of the pitch, roll and yaw axes. Combined with the throttle, which controls the aircraft's average thrust from its propulsion system, rotating the aircraft in pitch, roll and yaw orients the aircraft and moves it in space. The DJI Mobile SDK provides APIs to adjust the pitch, roll and yaw angle and velocity to achieve movement.

Pitch measures an object's rotation about the lateral (Y, pitch) axis. Adjusting the pitch will tilt the drone forwards or backwards. If the back propellers spin faster and have more thrust than the front propellers, the pitch will be forward. The thrust on each propeller is automatically balanced by the flight controller.

Roll measures an object's rotation about the longitudinal (X, roll) axis. Adjusting the roll will tilt the drone left or right. If the right propellers spin faster and have more thrust than the left propellers, the roll will be to the left.

Yaw measures an object's rotation about the vertical (Z, yaw) axis. Adjusting the yaw will change the heading of the drone. Half of the propellers spin clockwise, while the other half anti-clockwise. If one half spins faster than the other half, the drone will rotate around the yaw axis.

Gimbal and Camera

The DJI Mobile SDK has different capabilities for runtime use, such as mechanical stops of each gimbal axis, response speeds, and manual control can be customized and which axis are controllable. The work modes of the drone depend on the number of axes that are available for control:

- Free Mode: The gimbal can move independently of the aircraft's yaw, which means that pitch, roll and yaw are controllable.
- Yaw Follow Mode: Pitch and roll can be controlled so that the yaw will follow the aircraft's heading.
- FPV (First Person View) Mode: where yaw and roll are fixed relative to the drone while pitch remains controllable. This is the mode set for this project.

State information is pushed back to the mobile device at up to 10 Hz by the gimbal components. This information includes current attitude, calibration state and offsets, work mode and whether the gimbal is at a mechanical stop.

The gimbal can be moved in two ways through the DJI Mobile SDK: the first one is moving to an angle over a duration, where the angle rotation of the gimbal can be defined as either relative to the aircraft heading (Absolute) or relative to its current angle (Relative). The second way is to move at a certain speed in a direction where the direction can either be set to clockwise or counter-clockwise.

The camera can be set with DJI Mobile SDK to different operations modes, resolutions (in the project, it will be 640x380px for the image processing), frame rates, exposure, picture setting and file types

Missions

Many different missions, which allow automated flight, can be uploaded to and managed by the drone or from the mobile device. To assure that the mission run without problems the wireless link between the drone and the remote controller should always be good enough not to lose the connection. The missions that are available are:

- Hot Point Mission: the aircraft will repeatedly fly circles of a constant radius around a specified point called Hot Point.
- Follow-Me Mission: the drone will follow GPS coordinates continually sent to the aircraft maintaining separation and constant altitude.
- ActiveTrack Mission: allows tracking of a moving subject using the vision system and without a GPS tracker on the subject. It can be done in three different modes: trace mode (follows behind or in front of the subject keeping a constant distance), profile mode (moves in parallel withe the subject) and spotlight mode (the remote controller can be used manually to control the aircraft move around the subject).
- TapFly Mission: flies the drone in the direction of a point on the live video stream that the user chooses.
- Waypoint Mission: the aircraft will fly to a series of pre-defined locations (waypoints). A location is a latitude, longitude and altitude. Aircraft heading and altitude between waypoints can change either gradually or at the waypoint itself. At each waypoint a series of actions can be executed, such as taking a photo. The drone can execute 99 waypoints per mission.

1.4 Drone Controller

As the purpose of the project was to film a skier from different positions and that is something that DJI Mobile SDK does not support, another program was required for tracking and control, written in C++ on the control-PC running Ubuntu/Linux, called `DroneController`. The program is divided in many files, each one with it own header file, which allows for a better structure of the program for future modifications. The flowchart of the program is shown in Figure 1.5.

To run the app it is necessary to first execute the DroneController program in order to establish the connection between the mobile phone and computer. When both devices are connected, the information from and to the drone will start to be sent, but any of the missions programmed will not be executed until the user introduces the code in the session website.

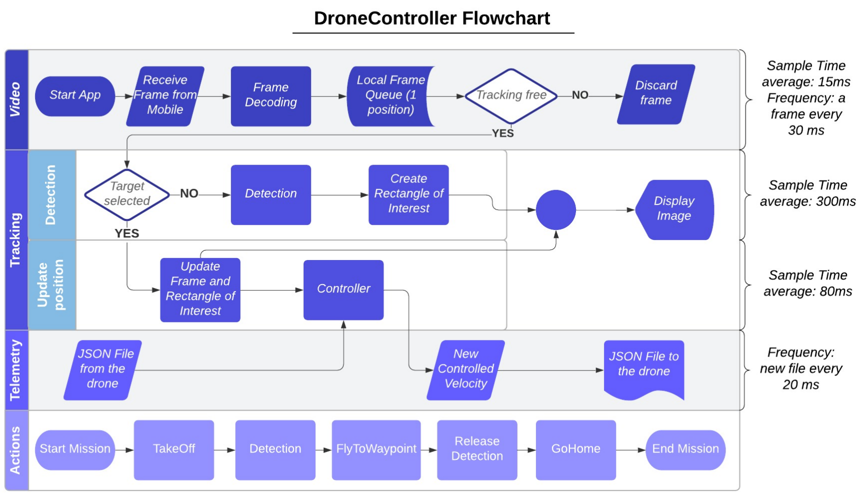

Figure 1.5 Flowchart of DroneController program

Video

The decoding and processing of every video frame is done using the OpenCV library [7] and for the image detection of people the real-time object detection system YOLOv3 is used [8]. Although the drone is filming in 4K resolution, in the computer the video will be rescaled to the lower resolution of 640x380 pixels (in order to send and use images faster).

When the app has been started video frames are stored in a buffer and when it is full, the video is started to be sent to the computer from the phone through that data buffer. The computer receives a frame every 30ms, in average, and that image is decoded and stored in a queue of size one waiting to be processed, shown in video part of Figure1.5. If the image recognition process is working and a new frame arrives, it will discard the previous image as a way to always have the newest frame. This will happen until a frame is required by the processing section and the vector will be empty until the next frame arrives to the computer.

Tracking

Once a frame arrives from the drone, it will go to the program module called `PersonTracker`, where the image recognition process is done. If the target hasn't been selected, the program will try to detect people in the image until the person at the computer selects the person that the drone should follow. Then when the person is chosen, the program will from now on use the subsequent frames to update the position of the target. In order to help the programmer debug, in every frame two lines that form a cross are drawn to show the center of the screen and when the program has a target tracked bounding box, another cross will be drawn for the center of this bounding box. This rectangle of interest, which is being updated in every frame, is not constant and will change the size slightly trying to find the complete size of the person.

The way to control the gimbal's pitch and the drone's yaw for tracking a person that was implemented before the start of this thesis work consisted solely of a proportional controller. Firstly, it measured the error in pixels of the center of the rectangle of interest on the target, to the center of the screen both vertically and horizontally. Then the proportional controller was implemented having as input the pixel error of the image from the center of the target to the center of the screen for each axis. These simple P-controllers are shown in Equations 1.1 and 1.2.

$$\mathrm{u_{yaw}} = \frac{\mathrm{pixelError_{yaw}}}{7} \tag{1.1}$$

$$\mathrm{u_{pitch}} = \frac{\mathrm{pixelError_{pitch}}}{10} \tag{1.2}$$

To reduce flickering, a dead-zone for the gimbal motion was introduced. If the velocity calculated for the gimbal pitch is between -5°/s and 5°/s a null velocity is returned. Otherwise the calculated velocity reference is returned. In the case of the drone yaw, the velocity calculated will always be sent.

Telemetry

All the information received from the drone and transmitted to it, and is not part of the video, is sent in another thread, all included in a JSON file. For example, if the drone altitude is wanted to be read or to set a new velocity reference for the yaw, it will be received and sent by a JSON file. The commands available depend on the DJI Mobile SDK, see Section 1.3. The frequency of receiving and sending these JSON files is one file every 20ms.

Actions

As it was said in the beginning of Section 1.4, the drone will not fly until the mission actions have started. These missions, predefined in the program, are called and ordered from a mission file.

The first action needed to activate the flight of the drone is `TakeOff`, for which the drone will be elevated 1.2m from the ground. Then the drone will ascend and go to the detection position. In this point, it will point to a certain area where the person must be located and wait until a target in the computer is selected to be followed. Then the tracking and recording will start.

After that, `FlyToWaypoint` action is called. In this action the drone will go to different waypoints that were previously defined, corresponding to a nominal path along the ski slope, while the drone will track the skier with the camera. In these waypoints, the index of the waypoints, latitude, longitude, altitude and yaw (only at that position) are set. Moreover, the minimum speed of the drone is set and if the drone must keep that speed constant or not until it gets to the next desired point. During the movement of the drone to these waypoints, the gimbal pitch and the drone yaw will be moving according to the skier, trying always to keep it centered in the screen.

Finally, when the drone gets to the final waypoint, it will release the detection of the target, stop the recording and call `GoHome` action, where the drone will return to the place from where it took off. Once the drone lands, the mission will have finished.

1.5 Transmission of information

There are two different ways of receiving information from the drone. One is the stream of the video images shown in green color in Figure 1.6, and other is the telemetry information data in blue. As explained in Subsection Telemetry, the commands sent to the drone are sent in a JSON file, so they are part of the telemetry data, and can be transmitted to the drone with a delay of around 70ms. This delay includes, being first sent to the app and then, forwarded from the app to the drone.

Every piece of information that is transmitted from and to the drone, both image and data, goes firstly to the remote controller, which is connected with a cable to the Android mobile phone, where the information is decoded and shown through the app. For the image, it arrives to the app around 300ms delayed and then, it is compressed again and sent to the computer. This transmission takes around 450ms and it is longer because it is done by WiFi connection. There is therefore a big dependency on the quality of the WiFi connection because if it is bad, the image can be display on the computer screen even 2s delayed. However, with good connection the delay is typically below 1s.

As this is the main cause of time delay in the transmission and it can differ hugely, it is important to explain in detail every single step that the information and image go through.

When the main program is started in the computer, all the sockets and connections are not activated until the mobile app has been started. Then, the telemetry data arrives pretty fast (every 20ms), but with a delay of 200ms from the drone. The first video frame takes 1.5 seconds fixed delay to be received in the computer. This delay is due to the buffer, that needs to have a determinate fixed size to be sent. For this reason, although it takes 1.5s since the program has been loaded, the first frame received might only be 800ms delayed (without counting the processing time of the frame, just when it has been decoded). This reception delay during the execution of

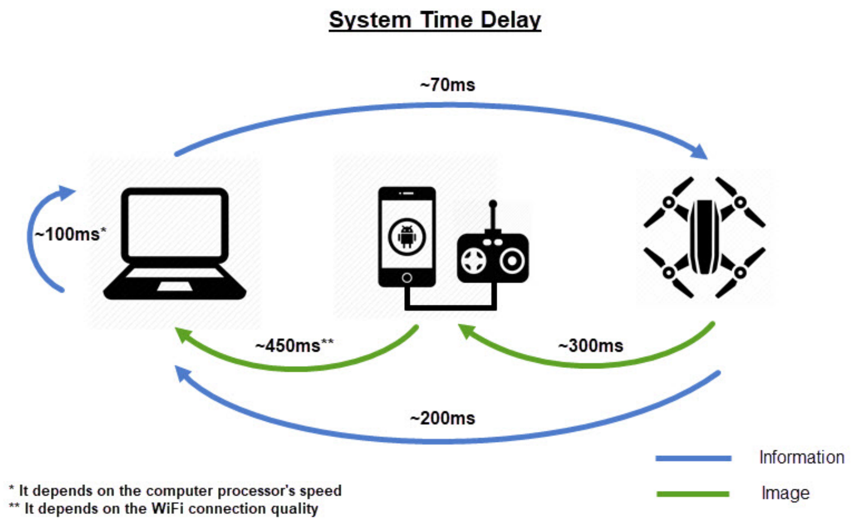
System Time Delay
~70ms
~100ms*
~450ms**
~300ms
~200ms
Information
Image
* It depends on the computer processor's speed
** It depends on the WiFi connection quality

Figure 1.6 Time delay of every system part

the program is around 700ms on average and the decoding just 10ms on average.

Recognizing and selecting a target takes a long time, in part because it will detect all possible targets. After the detection it takes noticeably less time to update the position of the target and it depends very much on the processor's speed. With the computer used for this project, an HP EliteBook 840 G3 with a processor Intel Core i5 with vPro [9], the sample time for the detection was around 300ms and for the target update, around 80ms, as Figure 1.5 shows. Using a computer with four cores for instance the sample time can be halved.

In addition, it is important to comment that at least for the computer used, every time that an action is started, the sample time is bigger. After some runs, it decreases to a shorter sample time.

1.6 Skier motion

The purpose of the project was to design a controller for the drone while it was tracking someone performing an outdoors sport such as skiing. Therefore, it was important to know the movement of a skier.

Figure 1.7 shows that the movement of these skiers follows an approximate zig-zag or sinusoidal motion. They descend in a straight line and turn to reduce their speed. Depending on how a skier takes the turn, the path can be straighter and more aggressive or have larger curves with a smoother turn. So while the skier is going

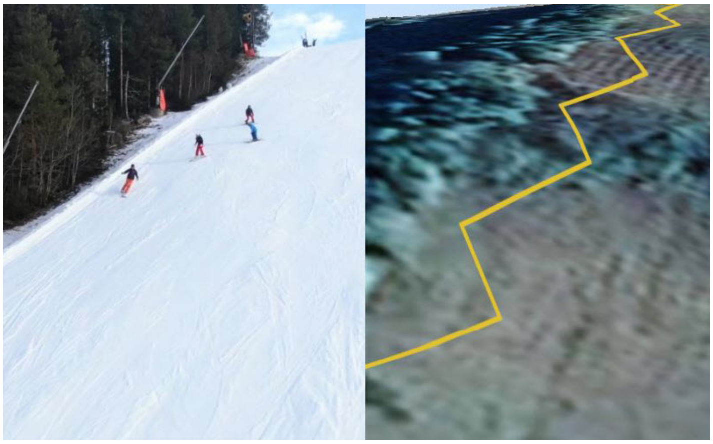

Figure 1.7 Clip from a video filmed by the drone and the approximate, reconstructed path of the skiers

approximately straight the movement can be predicted and the break point will be in the turns.

2

Methodology

2.1 System Identification

The first step to designing a controller for the system was to identify what kind of system it was and determine its parameters. Due to the nature of the project, in which an externally produced DJI drone was being used with internal proprietary code and data sheets unavailable, the system had to be determined through measurements.

The functions that could be used to control the drone that were available included setting the angular velocity, setting the angle, and reading the angle. However, reading the angular velocity was not possible. This went for both the pitch of the gimbal, on which the camera was mounted, and the yaw for the entire drone. With the given constraints measurements were taken by making multiple test runs where a desired angle was set, i.e. the control signal of the system, and measurements of the angle were made. Likewise, test runs where the angular velocity was set with measurements of the angle were also made. This was done for both controlling the gimbal pitch and the drone yaw.

Attempts to model the system included the System Identification Toolbox in Matlab [10] and Simulink. However, what proved to work best was choosing appropriate transfer functions and using a conversion table to get the same transfer function in the time domain. Choosing suitable transfer function were done by looking at a time plot of the measurements. The *fminsearch* [11] function was then used in Matlab to minimize the squared error between the model and all the measurements.

2.2 Measuring the Angle Error

The controller that had been previously used was obtained through trial and error and had been using the error between the target and camera (center of the screen) in pixels. However in order to provide more accurate results it was necessary to measure the error as an angle.

It is important to emphasize that the coordinates origin of the screen is in the top left side of the screen. There is a function for each axis, although both have the same structure. For the horizontal axis the function takes as input values the pixel error from the center of the screen to the center of the bounding box of the person, variable a in Figure 2.1. Similarly, variable b is used in the vertical axis. The field of view is 83º horizontally and 48º vertically. The width of the received image is 640 pixels (p) and the height is 380 pixels.

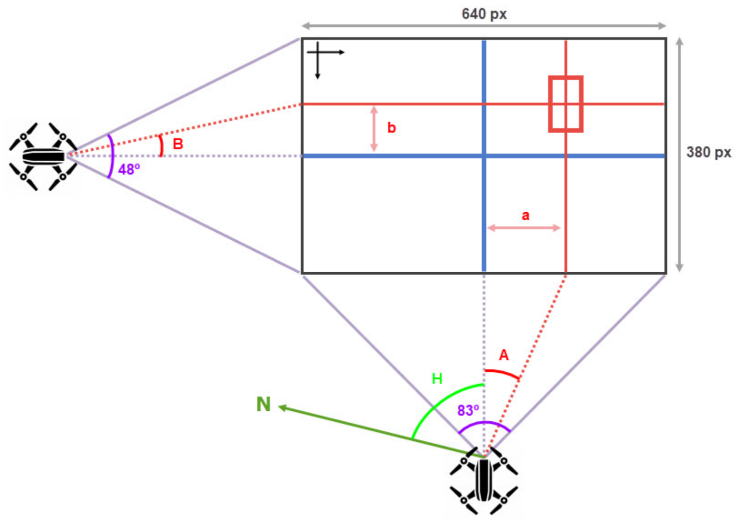

Figure 2.1 Field of view of the drone and variables to calculate the conversion of pixel error to angle error

The objective is to calculate the angles A and B in Figure 2.1, following the principles of trigonometry. First the "distance" d in pixels between the drone and screen is calculated (which is merely an arbitrary constant used to better understand how the pixels are converted to an angle), yielding that

$$\tan\left(\frac{83^\circ}{2}\right) = \frac{640/2}{d_a} \Rightarrow d_a = \frac{320}{\tan(41.5^\circ)} = 361.69$$

$$\tan\left(\frac{48^\circ}{2}\right) = \frac{380/2}{d_b} \Rightarrow d_b = \frac{190}{\tan(24^\circ)} = 426.75$$

This is then used to calculate angles A and B as given by Equations 2.1 and 2.2.

$$\tan(A) = \frac{a}{d_a} \Rightarrow A = \arctan\left(\frac{a}{361.69}\right) \tag{2.1}$$

$$\tan(B) = \frac{b}{d_b} \Rightarrow B = \arctan\left(\frac{b}{426.75}\right) \tag{2.2}$$

These functions can be seen implemented in lines 134-160 of Appendix A.

It is relevant to mention that the units of the angular measurement used in the program for the calculations are in radians as a baseline and that all calculations were made using degrees, so a conversion to radians in the code is needed.

It is also very important to clarify that due to the coordinate system and how the drone moves, some things change compared to how it is in theory. Let's say for instance that the image detects a person up to the right like Figure 2.1 shows. In this case to decrease the error in the vertical axis, the pitch gimbal has to move upwards. But the error will be negative (due to the coordinate system of screen) which will tell the controller to move the gimbal down, and then the error increases. Therefore, to move the camera in the right direction, the sign must be switched.

2.3 Time delay and sample time

One crucial part to get a great controller and predictor is to measure the sampling time of the process and the time delay of the image. The sample time was calculated as the time it took between signals sent from the controller.

Getting the time delay of the image is more complicated. As the telemetry data is not coordinated to the image, it is not known when received images of the video were taken. The first way to measure it was taking a picture of a timer running and the live video in the computer of the same timer the drone was filming. The problem was that the time delay of the image depends on the WiFi connection.

So that, the way the time delay was measured was placing a dark object in front of the camera and a white object below the camera, on the ground, before the drone takes off. The pixel color of the center of the screen is measured using the RGB color model [12]. This model is the composition of color in terms of the intensity of the primary colors of the light: red, green and blue (RGB). The scale of this model goes from 0 to 255 for each color e.g. the color white is [255,255,255] and the color black is [0,0,0]. However, in OpenCV the order of the primary colors is BGR instead of RGB.

Once the pixel color is measured, a fast movement of the gimbal pitch is sent and when the program detects that the image change from a dark value (close to 0 in RGB scale) to a light color (close to 255 in RGB scale) the complete time delay can be measured, including the time taken from sending action to the drone.

2.4 PID Controller

Measurements were made to determine the various delays of the system, and the sampling time of the controller. The system was then implemented in Matlab and

PID Tuner [13] was used to get an approximate controller. Afterwards, the closed loop system is plotted and discretized as well as fine tuned. Then the preliminary controllers could be tested on the real process.

With a derivative filter in place for noise reduction the PID controller is given by Equation 2.3.

$$C(s) = K_p + K_i \cdot \frac{1}{s} + K_d \cdot \frac{N}{1 + \frac{N}{s}} \tag{2.3}$$

Afterwards the controller could be discretized [14] using backward difference, as shown in Equation 2.4. Combined with Equation 2.3 it yields the discretized controller in Equation 2.5 where h denotes the sampling time of the controller.

$$s' = \frac{z-1}{zh} \tag{2.4}$$

$$C(z) = K_p + K_i \cdot \frac{zh}{z-1} + K_d \cdot \frac{N(z-1)}{(1+Nh)z-1} \tag{2.5}$$

This gives Equations 2.6 and 2.7 that are used to implement the controller.

$$\begin{cases} b_0 = K_p(1+Nh) + K_i h(1+Nh) + K_d N \\ b_1 = -(K_p(2+Nh)) + K_i h + 2K_d h \\ b_2 = K_p + K_d N \\ a_0 = (1+Nh) \\ a_1 = -(2+Nh) \\ a_2 = 1 \end{cases} \tag{2.6}$$

$$u(k) = -\frac{a_1}{a_0}u(k-1) - \frac{a_2}{a_0}u(k-2) + \frac{b_0}{a_0}e(k) + \frac{b_1}{a_0}e(k-1) + \frac{b_2}{a_0}e(k-2) \tag{2.7}$$

Finally, it is also important to implement some form of anti-windup [15] function for the controller since the real process has limitations on its input signal.

2.5 Smith Predictor

Due to the large delay in the system, a Smith predictor [16] could allow for a faster controller. The basics of a Smith controller are to use a model of the system, rather than the real system, and feed back the response without the delay. The difference between the model (with the delay) and the real process is then compared and fed back into the controller. The principle is shown in Figure 2.2.

It can easily be seen that if the process model is perfect, then the delayed Smith measurement and delayed real measurement cancel each other out. This means the

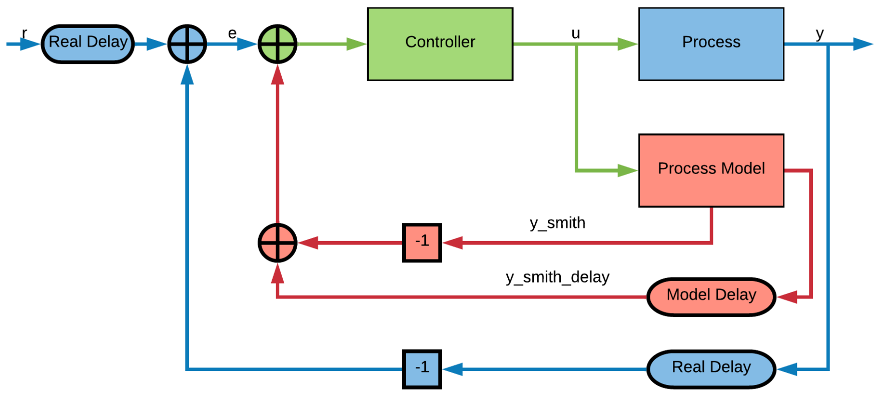

Figure 2.2 Smith predictor block diagram

only thing entering the controller is the non-delayed Smith measurement (which should be equal to the real measurement). If the model of the process is not perfect, then the error between the delayed Smith measurement and the real delayed measurement will be returned into the controller and adjusted for. This still does not remove any delay in the reference, and only allows the controller to act faster.

Assuming a perfect model, it is then possible to design a controller completely as if the delay in the measurement did not exist.

Implementation

In order to implement the Smith Predictor in the code, it was required to store the previous values of the model in a vector to get the corresponding real value of the drone due to the time delay.

Therefore, three vectors were implemented using the `std::vector` library: one for the Pitch values, another for the Yaw values and the third one is for the sample times of each iteration. In such iterations, the newest value is stored at the end of each vector. Then with a `while` function the sample time of the previous steps were summed until the time delay is achieved. Finally linear interpolation was used to approximate the delayed value of the desired Smith predictor value.

To avoid the collapse of the vector's size, at the end of each iteration, the first value of each vector is removed. Therefore, there is always the same amount of values in the vector (once the full size of the vector has been reached). Although different in implementation, this effectively becomes a ring buffer.

It later turned out that an integrator was part of the process model, so trapezoidal integration was used in the code given by Equation 2.8.

$$\int_a^b f(x)dx \approx (b-a) \cdot \frac{f(a)+f(b)}{2} \tag{2.8}$$

2.6 Kalman Filter

In the process that was being controlled there was a very large time delay in receiving measurements. Which meant that as long as the target was moving the controller would never be able to make the camera look directly at the target since it was constantly trying to move the camera to the target's old position.

In order to remedy this a simple prediction model was introduced where the idea was to take the current position of the target and add the velocity of the target multiplied by the time delay. If the target would be keeping constant speed then the camera would be able to look at where the target actually is rather than where the target was the time delay ago. However, in the process it is only possible to read the position of the target (in the form of pixels on an image) and not the velocity. For this reason a Kalman filter [17] is introduced to get an estimation of the velocity.

The model for the Kalman filter is based on the assumption of double-integrator dynamics

$$\begin{cases} x = x_1 \\ \dot{x} = x_2 \\ \ddot{x} = 0 \end{cases}$$

where x is an angle, which yields that in state-space representation

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

The B-matrix is considered to be 0 since the Kalman filter was following the the target, i.e., the reference, and the controller signal had no affect on the target's position. It only had an effect on the camera's position. Since only the position of the target could be measured, and not the target's velocity, it was given that

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

Additionally, the K-matrix was given by

$$K = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix}$$

which gave Equations 2.9 to 2.12. This is visualized in Figure 2.3.

$$\dot{\hat{x}} = A\hat{x} + K(y - \hat{y}) \tag{2.9}$$

$$\hat{y} = C\hat{x} \tag{2.10}$$

$$\dot{\hat{x}} = A\hat{x} + K(y - C\hat{x}) \tag{2.11}$$

$$\dot{\hat{x}} = (A - KC)\hat{x} + Ky \tag{2.12}$$

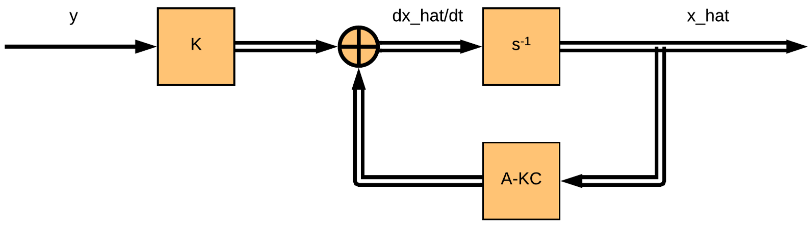

Figure 2.3 Kalman filter block diagram

In order to discretize [18] Equation 2.12, and still have it be dependant on a varying sampling time h, Equations 2.13 and 2.14 (the ordinary *c2d* [19] function in Matlab) cannot be used without creating needlessly complicated calculations.

$$\Phi(h) = e^{(A-KC)h} \tag{2.13}$$

$$\Gamma(h) = \int_0^h K e^{(A-KC)s} ds \tag{2.14}$$

Instead a simplified way of calculating the matrices was used, shown by Equations 2.15 and 2.16.

$$\Phi_t(h) \triangleq I + (A - KC)h + \frac{(A-KC)^2}{2!}h^2 + \frac{(A-KC)^3}{3!}h^3 \tag{2.15}$$

$$\Gamma_t(h) \triangleq \int_0^h K\Phi_t(s)ds \tag{2.16}$$

The results were then inserted into Equation 2.17. Since the Kalman filter was used to approximate the velocity of the target in order to approximate the position of the target, the approximate reference is given by Equation 2.18, where t_d is the time delay from when an image is sent from the drone to its effect being noticed in a received image.

$$\hat{x}(kh+h) = \Phi_t \hat{x}(kh) + \Gamma_t y(kh) \tag{2.17}$$

$$\hat{r}(t) = \hat{x}_1(t-t_d) + \hat{x}_2(t-t_d) \cdot t_d \tag{2.18}$$

2.7 Combining Smith Predictor with Kalman Filter

A core issue in implementing the Kalman filter was rooted in the fact that it was only possible to measure the angle error (more precisely, the pixel error and convert it into an angle error) between the reference (i.e., the target or skier) and the measurement (i.e. the angle of the camera). This meant that there was no direct access to measuring the reference and how its velocity was being changed.

However, by the time the Kalman filter was being worked on it had been made clear that the Smith predictor could very accurately predict the angle of the camera. The solution was to apply the Kalman filter on the sum of the error and the Smith predictor's estimation of the camera position at that point in time, as clarified in Equation 2.19. To be clear, r_s becomes the input value y for the Kalman filter as seen in Equation 2.12 and 2.17 while the y (or rather y_{smith}) relates to the camera angle y in Figure 2.2 and 2.4.

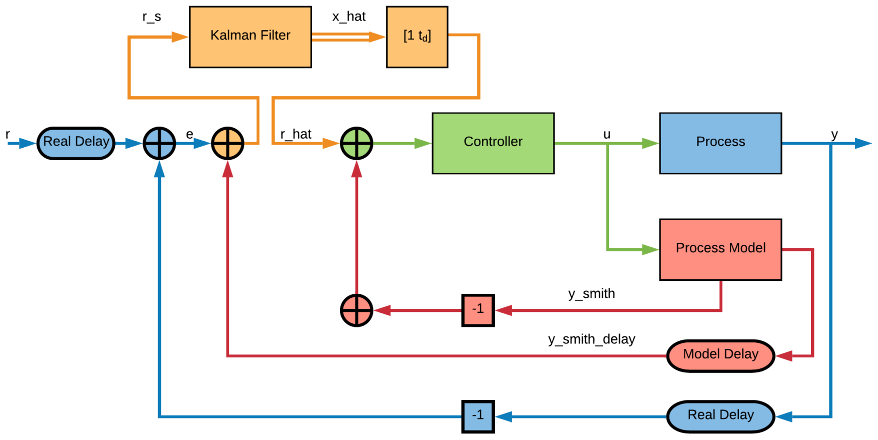

Figure 2.4 Full control system, consisting of controller (green), Smith predictor (red), process (blue), and Kalman filter (orange)

$$r_s(t-t_d) = e(t-t_d) + y_{smith}(t-t_d) \tag{2.19}$$

$$y_{smith}(t-t_d) = y_{smith\_delay}(t) \tag{2.20}$$

Figure 2.4 shows a block diagram of the full control system and how each part is connected. Note that although the reference r and measurement of camera position y are shown individually it was only possible to measure the error between them.

3

Results

3.1 System Identification

For all sets of measurements the input signals were sent as constant steps. The measurements were then normalized by dividing them by the input signal and plotted out. The results can be seen in Figures 3.1 and 3.2. From Figure 3.1 it can be seen that a transfer function according to Equation 3.1 became suitable, where the input is an angle and the output is an angle. Since Figure 3.2 showed an input signal that is an angular velocity, and the measurements showed an angle, it became reasonable to just add an integrator to Equation 3.1. However, the Matlab code used to estimate the transfer function then completely ignored the time delay (for some unknown reason). Since the curve of the slope was so small, and the measurements were not fully accurate anyway, a simple delay and integrator shown in Equation 3.2 was instead used.

$$G_a(s) = b \cdot \frac{a}{as+1} \cdot e^{-sL} \tag{3.1}$$

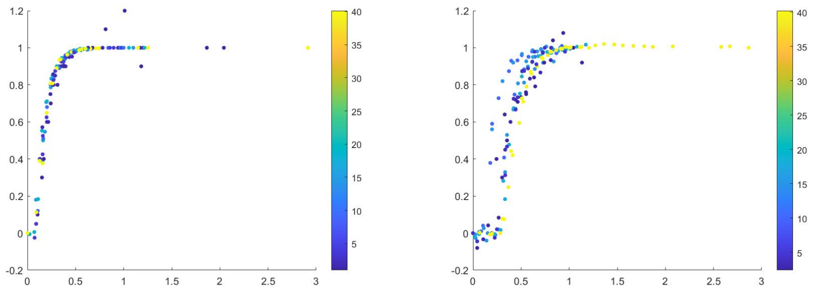

Figure 3.1 Plot of normalized gimbal pitch (left) and drone yaw (right) angle measurements over time, color coded to input signal

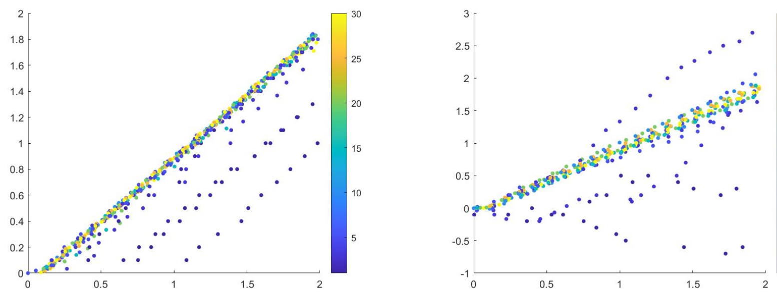

Figure 3.2 Plot of normalized gimbal pitch (left) and drone yaw (right) angle measurements over time for a certain angular velocity, color coded to input signal

$$G_v(s) = \frac{b}{s} \cdot e^{-sL} \tag{3.2}$$

When converted from the Laplace domain to the time domain, these transfer functions become Equations 3.3 and 3.4 respectively.

$$g_a(t) = b \cdot (1 - e^{-at}) \qquad \begin{cases} g_a(t-L) & t-L>0 \\ 0 & t-L<0 \end{cases} \tag{3.3}$$

$$g_v(t) = b \cdot t \qquad \begin{cases} g_v(t-L) & t-L>0 \\ 0 & t-L<0 \end{cases} \tag{3.4}$$

After the *fminsearch* function was used in Matlab using the transfer functions in the time domain, a line was fitted to each set of measurements in order to get the transfer functions. The results are shown in Figures 3.3 to 3.6. The parameters for the transfer functions are provided in Table 3.1.

Table 3.1 Parameter values for the transfer function

Function	Measurement	a	b	L
Angle	Pitch	8.82	1.01	0.08
Angle	Yaw	5.55	1.02	0.23
Velocity	Pitch	-	0.93	0.10
Velocity	Yaw	-	0.97	0.13

However, as can be seen there were plenty of outliers. This was in part due inaccurate measurements and it also turned out that sometimes the first run in a set of measurements had a much larger delay. Moving forward, since these transfer functions are results of built-in functions made to set the angle or speed in the drone,

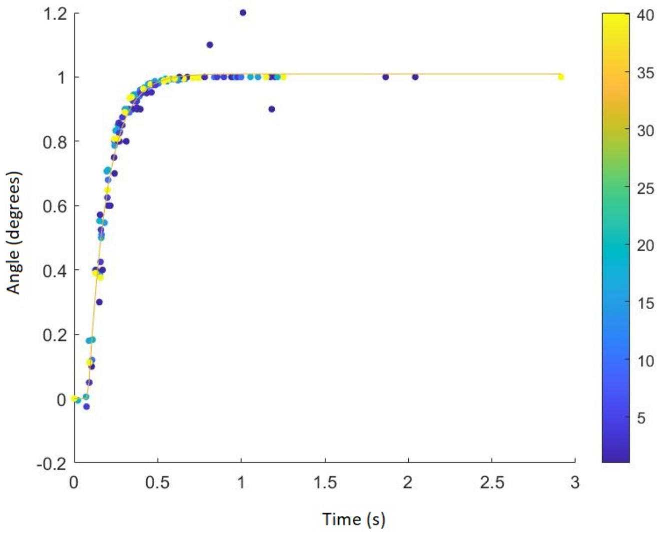

Figure 3.3 Plot of transfer function next to the gimbal pitch angle measurements

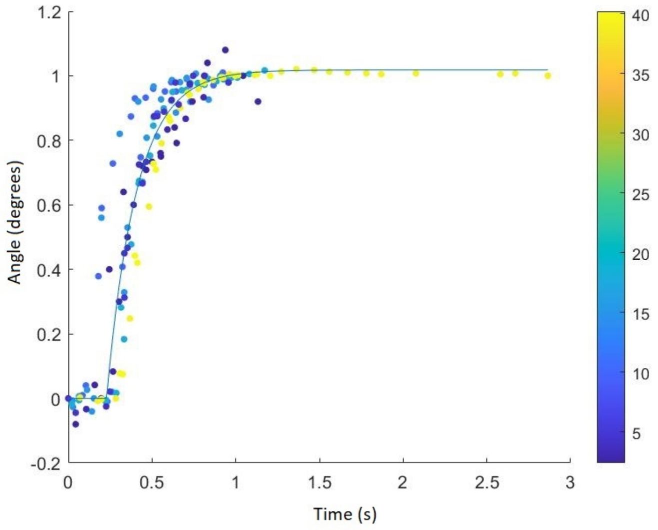

Figure 3.4 Plot of transfer function next to the drone yaw angle measurements

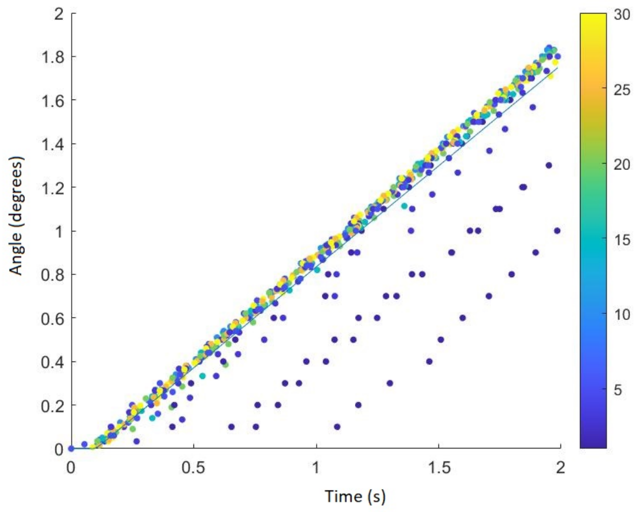

Figure 3.5 Plot of transfer function next to the gimbal pitch angular velocity measurements

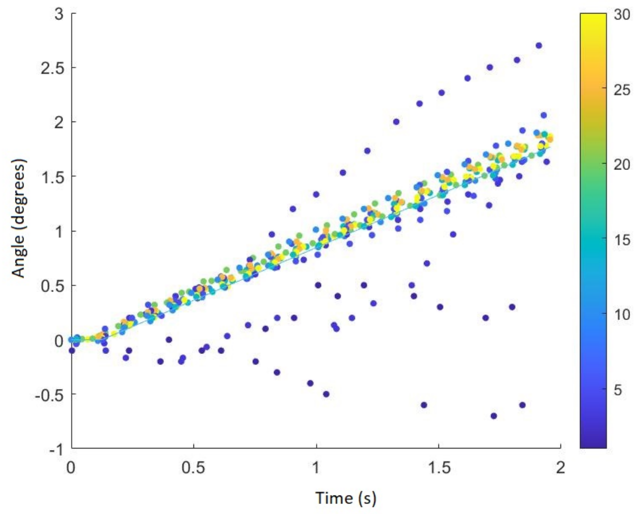

Figure 3.6 Plot of transfer function next to the drone yaw angular velocity measurements

and all b values are so close to 1, it was assumed that b=1 for all four transfer functions.

3.2 Sample time and Time delay

The sampling time of the controller was greatly dependant on the speed of the computer's processor. The sampling time was constantly changing, shown in Figures 3.7 and 3.8, where the results were centered around 70 to 80ms. However, this variation in sampling time did not have to represent a standard run and can vary a lot between runs. This variation was a core reason in not assuming a constant sampling time in the discretization of the controller, Smith predictor or Kalman filter.

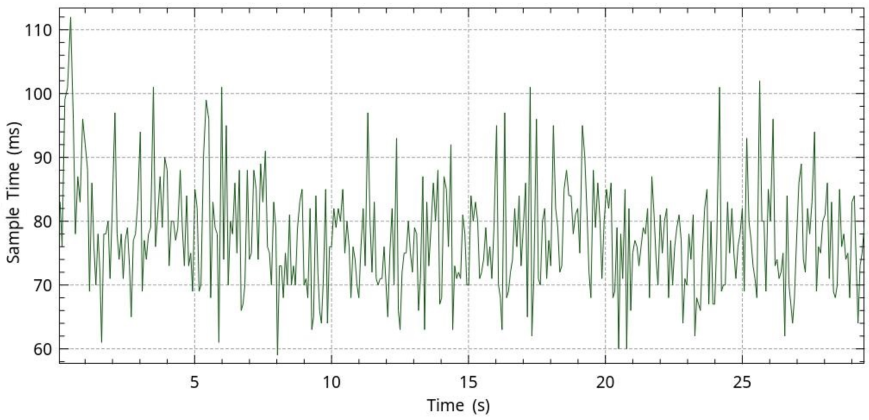

Figure 3.7 Sample times during a program's execution.

Regarding the full delay from computer to drone and back to the computer, it became evident that the delay was far from constant. Attempting to use the proposed method of measuring the delay at the beginning of a run, comparing a white and black background, turned out to give worse results than an estimated full delay of 0.7 seconds. This is a result of the quality of the WiFi connection which can change during the run of the program.

3.3 PID Controller

Even though controllers were made for the angle processes G_a, it turned out to be a problem with the angle process. The issue was that it was not possible to increase or decrease the angle without reading the current angle (without altering other parts of the project which would create new problems). Note that this angle is the angle

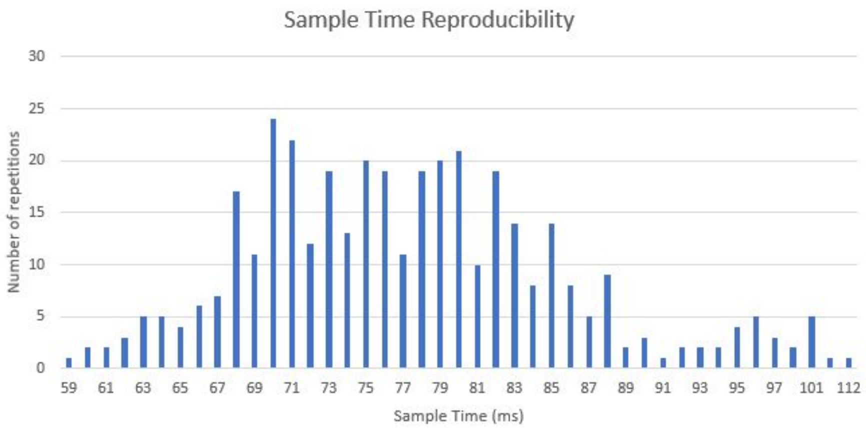

Figure 3.8 Histogram of Sample time.

of the gimbal or drone in reference to north, and not the angle from the image. The main issue in reading this angle was that it had a different delay (blue in Figure 1.6) than the images from the camera and could not be synchronized to the other parts of the process. For this reason the project was continued by designing a controller for the velocity processes G_v. The velocity processes are differentiated by calling the angular velocity process G_{vp} for the gimbal pitch and G_{vy} for the drone yaw.

The controller for the gimbal pitch C_{vp} and the controller for the drone yaw C_{vy} are shown in Equation 3.5 with parameters in Table 3.2. Since the models of the processes were almost identical the same controller could be used for them both. The closed loop of the process and controller can be seen in Figure 3.9.

$$C_v(s) = K_p + K_i \cdot \frac{1}{s} + K_d \cdot \frac{N}{1+\frac{N}{s}} \tag{3.5}$$

Table 3.2 Parameter values for the angular velocity controllers

K_p	K_i	K_d	N
0.5	0	0.15	3

The controller was then promptly discretized and implemented. Since $K_i = 0$ there was no need for an anti-windup functionality.

Implementation

The implementation of the PID discretized controller and its parameters was done following the Equations 2.6 and 2.7. In lines 126-145 of Appendix B the implementation of the PID for the gimbal can be seen and lines 165-182 for the yaw. All the

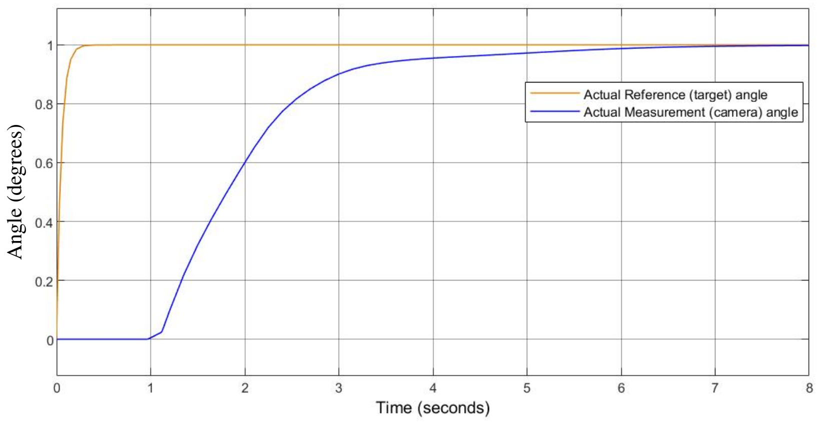

Figure 3.9 Closed-loop step response of C_{vp} and G_{vp} (with a full delay [computer-to-drone and drone-to-computer] of 960ms and a constant sampling time of 150ms) in simulation

values were updated in every iteration due to the variation of the program execution sample time for improved accuracy.

Measurements were taken from the stationary target using the same controller values as shown in Table 3.2. The measured angle and the control signal are shown in Figure 3.10 for both the pitch and yaw.

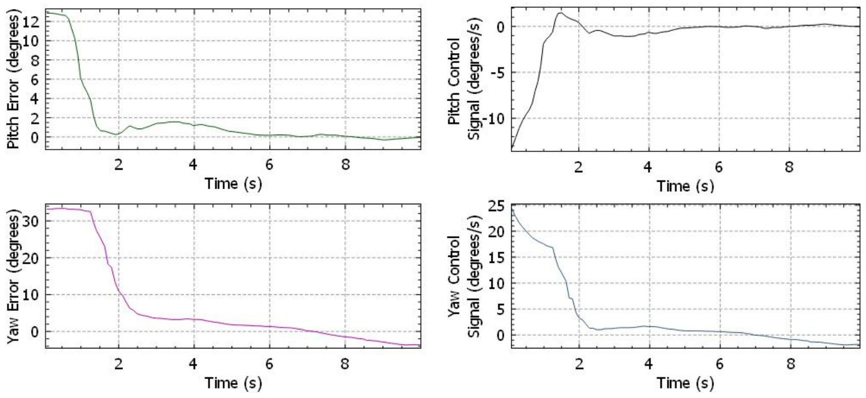

Figure 3.10 Response of yaw and gimbal controlled with a PD controller following a target without movement

3.4 Smith Predictor

Since the delay can be ignored by a Smith predictor with a perfect model, the delay from drone to computer was ignored when designing a faster controller. However, the delay L in Table 3.1 was still kept. That is because the delay L is the actual delay in which the system starts moving. By using this system's new controller parameters, given in Table 3.3, the closed loop step response could be evaluated. As seen in Figure 3.11, the new controller C_{smith} gives a much faster response.

Table 3.3 Parameter values for the angular velocity controllers in simulations

K_p	K_i	K_d	N
2.9	0	0.1	3

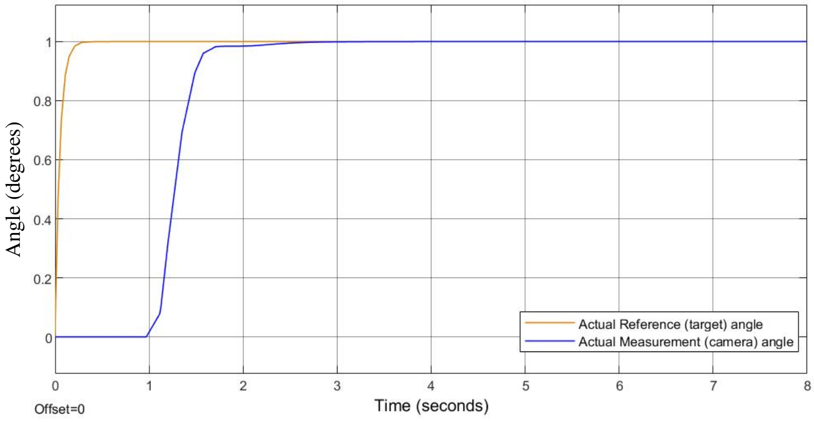

Figure 3.11 Closed-loop step response of C_{smith} and G_{vp} (with a full delay [computer-to-drone and drone-to-computer] of 960ms and a constant sampling time of 150ms) in simulation

Implementation

The Smith Predictor was implemented in the program using trapezoidal integration for the system model, as is shown in lines 137 to 146 of Appendix B for the gimbal pitch and in lines 176 to 184 for the drone yaw, and using the ring buffer to store and select the proper model value delayed, in lines 82 to 108 of Appendix B. For the measurements of following a person standing still and camera initially facing wrong direction, using the parameters values of Table 3.4, the results for the error and controller signal can be seen in Figure 3.12 for both the pitch and yaw.

Table 3.4 Parameter values for the angular velocity controllers in practice

K_p	K_i	K_d	N
2.5	0	0.2	3

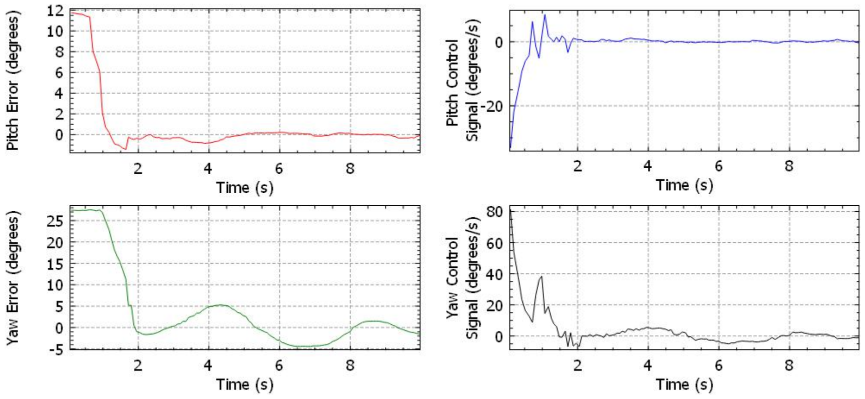

Figure 3.12 Response of yaw and gimbal controlled with Smith predictor following a target without movement

3.5 Kalman Filter

The dynamics of the Kalman filter were given by the A-KC-matrix in Equation 2.12. With poles placed in

$$\begin{cases} p_1 = -0.85 + 0.45i \\ p_2 = -0.85 - 0.45i \end{cases}$$

where the characteristic polynomial was given by Equation 3.6. This could easily be compared to the determinant of A-KC to acquire a suitable K-matrix, seen in Equation 3.7.

$$(s - p_1)(s - p_2) = s^2 + 1.7s + 0.925 \tag{3.6}$$

$$|A - KC| = s^2 + k_1 s + k_2 \tag{3.7}$$

This gave

$$K_1 = \begin{bmatrix} 1.7 \\ 0.925 \end{bmatrix}$$

and served as a decent starting point for the Kalman filter. The Kalman filter parameters were then manually tuned to

$$K_2 = \begin{bmatrix} 4 \\ 10 \end{bmatrix}$$

where the respective results in simulation can be seen in Figures 3.13 and 3.14 for an arbitrary measurement. The pole placement for the discretized Kalman filter of K_2, assuming a sampling time of h=0.200s, was given by Equation 2.13 and yielded that

$$\text{eig}(\Phi) \Rightarrow \begin{cases} p_1 = 0.590 + 0.316i \\ p_2 = 0.590 - 0.316i \end{cases}$$

which are within the unit circle and corresponds to continuous poles with negative real values.

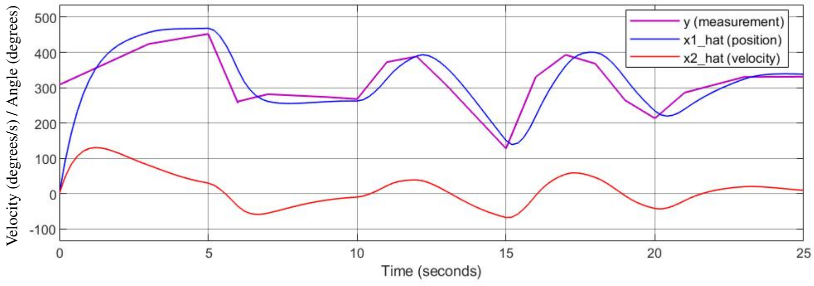

Figure 3.13 Kalman filter states given by K_1

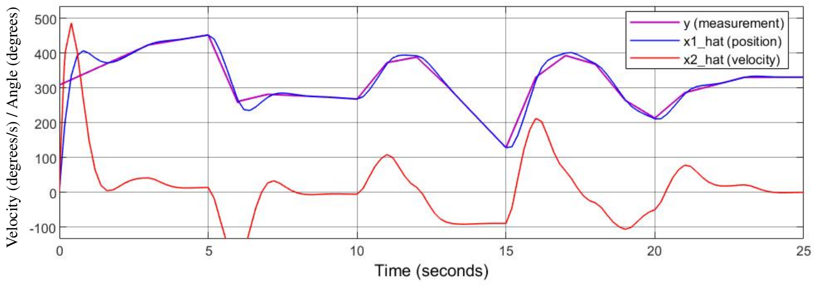

Figure 3.14 Kalman filter states given by K_2

By using Equations 2.17 to 2.20 it was possible to test the Kalman filter's ability to predict where a target was heading. In Figures 3.15 and 3.16 the Kalman filter prediction $\hat{r}$ can be seen for two different curves, both using a time delay from drone to computer at t_d=890ms and a constant sampling time of h=150ms.

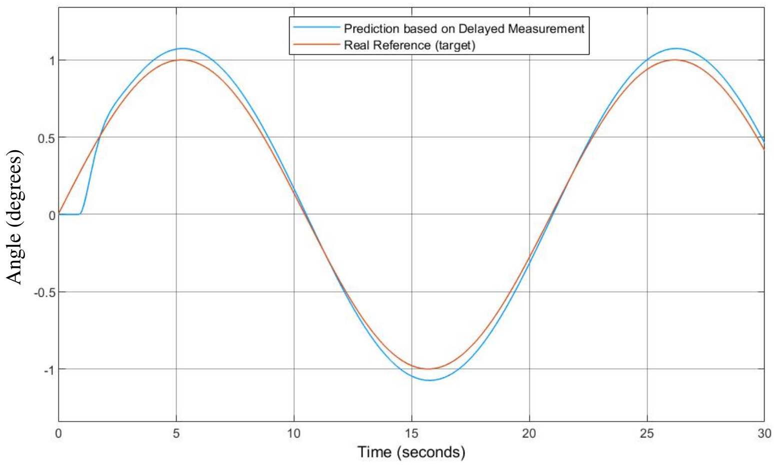

Figure 3.15 Kalman filter prediction given by K_2 for a target moving in a sinusoidal shape

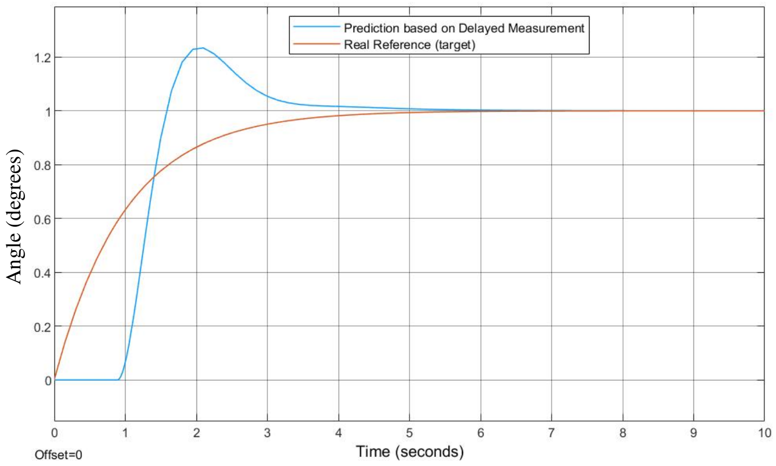

Figure 3.16 Kalman filter prediction given by K_2 for a target making a quick turn

Note that the two curves are used in simulation for two arbitrary curves and time frames, that intentionally showcase the strength and weakness of Kalman filter K_2. It became evident that the the ability to predict the position of a target that quickly changes velocity is difficult while a target that turns slowly (or not at all) is easy to follow.

Implementation

The Kalman filter was discretized and implemented using Equation 2.15. The Kalman filter did however have to be tuned further for the real process, giving the Kalman filter

$$K_3 = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$$

for the real implementation. The states of the filter can be seen in Figure 3.17 and the ability to follow a measurement and approximate the velocity is shown in Figure 3.18.

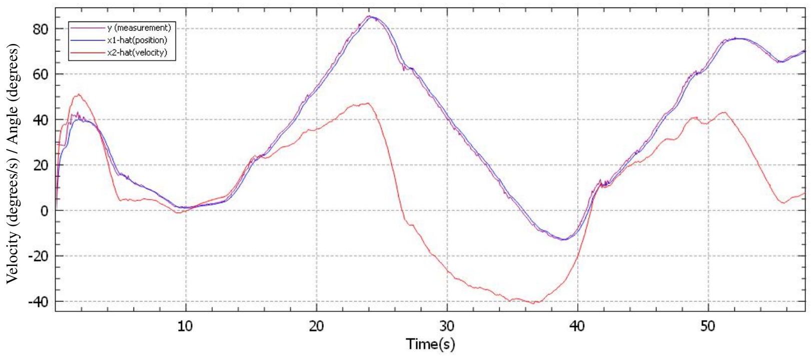

Figure 3.17 Kalman filter states given by K_3

3.6 Combining Smith Predictor with Kalman Filter

When everything was finally combined into one system, according to Figure 2.4, it was possible to get a fast controller that could get the camera into almost the same position as the target due to the prediction.

In Figures 3.19 and 3.20 the results can be seen of how well in simulation the actual position of the camera can follow the actual position the target. It should come as no surprise that the results are very similar to those in Figures 3.15 and

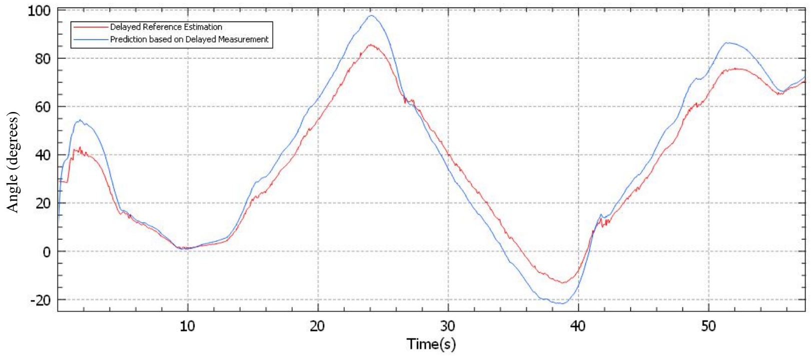

Figure 3.18 Kalman filter K_3 input and prediction for 0.7s later

3.16, respectively. The results are also using the K_2 Kalman filter, combined with the system consisting of C_{smith} and G_{vp}.

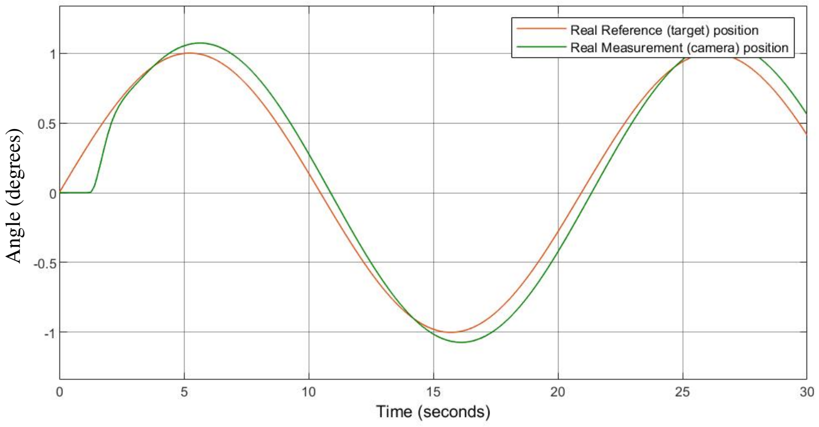

Figure 3.19 Actual position of camera and target (not the delayed measurement) compared in simulation for sinusoidal movement of target in system consisting of K_2, C_{smith} and G_{vp}

Implementation

When running full system it was not uncommon for the camera to start looking ahead of the target. For this reason Equation 2.18 was modified into Equation 3.8 where the trust factor is added. The trust factor T_{fac} can be seen as how much into the future it is desired to look, as a percentage.

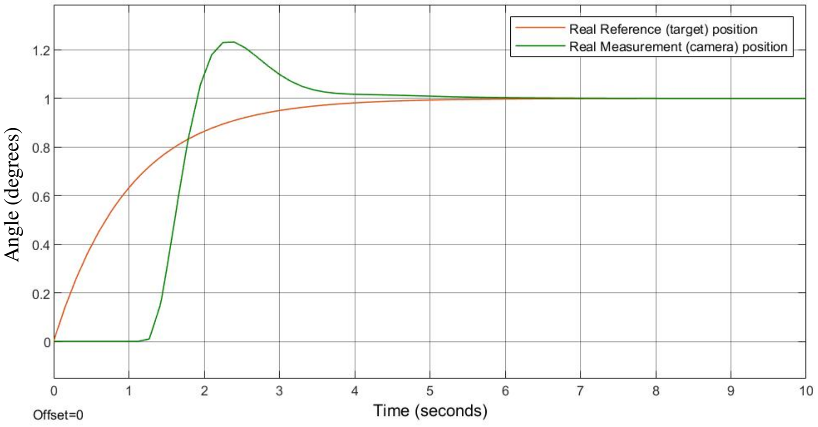

Figure 3.20 Actual position of camera and target (not the delayed measurement) compared in simulation for a fast turn in system consisting of K_2, C_{smith} and G_{vp}

$$\hat{r}(t) = \hat{x_1}(t - t_d) + \hat{x_2}(t - t_d) \cdot t_d \cdot T_{fac} \tag{3.8}$$

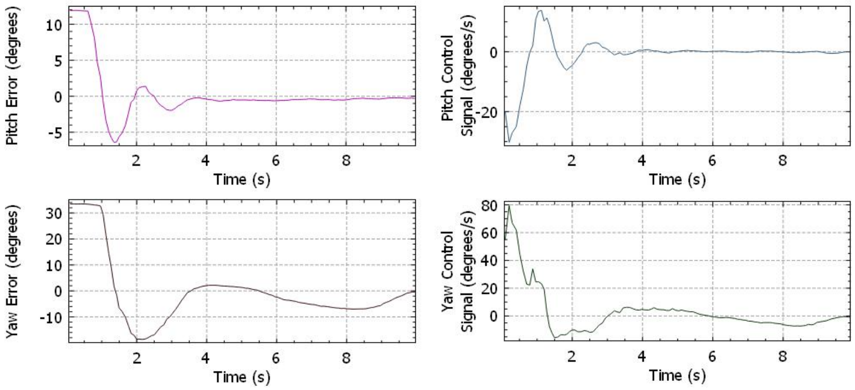

Figure 3.21 Response of yaw and gimbal controlled with Smith predictor and Kalman filter following a target without movement

When running the full implementation with Smith predictor and Kalman filter used to predict position of target a trust factor of 0.8 worked well for the pitch process. However, for the yaw a trust factor as low as 0.4 had to be used. Otherwise the system would become unstable and start to oscillate. The discretized implementation is seen in lines 112-124 for the pitch and in lines 152-163 for the yaw in Appendix B, as well as lines 138 and 176 for how they were connected to the controllers.

Still using K_3, the results of following a stationary target with an initial error are shown in Figure 3.21. There it can clearly be seen how there is an overshoot, which is due to the Kalman filter "thinking" the target moved that distance very quickly and believing the target will continue at that speed.

4

Discussion

4.1 Improved System Identification

A mistake that was made in modelling the system was to only model how the camera moves from being stationary to starting to move. This was enough for modelling the gimbal pitch and creating a very fast and functional Smith predictor. However, when controlling the drone's yaw the process seemed to be different when the velocity was decreased compared to increased, and was not as quick to stop as it was to start moving. Although not detrimental to the project this effect did decrease performance. The issue was realized too far into the project and at that point there was not enough time to model this effect. But if this project were to be redone then modelling the camera movement as the angular velocity is decreased, rather than just increased, should have been done as well.

Alternatively, if the gimbal on which the camera was mounted could be turned in the yaw direction then more control and accuracy could be achieved over the system. This way a fast controller could be used for the gimbal yaw, and a slow controller for the drone yaw so that no turn limit is reached for the gimbal. This could be handled with midranging control [20].

4.2 Measuring the Time delay

The idea to measure the delay at the start of a run by measuring the time it takes for the first image to be received from when the program is told to start did not work. The issue was that before the image could be sent an image buffer had to be filled and this made it impossible to know the actual delay. There was also an effect of the program running more slowly at the start of a run this kind of time delay measurement method was unlikely to work. Especially since the delay was not constant.

An alternative way of measuring the delay as it is right now is using the ability to change certain settings for the image or drone, such as modifying the frequency of the camera shutter, in order to measure the time for this effect to be detected, i.e., the time delay. The benefit of this would be the ability to measure the time delay mid flight but the downside would be that it would affect the video.

Instead it would be recommended to find a way to continuously measure the time delay similar to how the sampling time is measured. The best solution would be to simply add timestamps to the images that are sent from the drone to the computer, but that was not possible in this case.

If it is not possible to get timestamps for the images that are sent, the ability to mount a clock in front of the camera should be investigated. This could however be in the form of very tiny LEDs in the corner of the screen that form a binary clock. This would give every image a timestamp and would provide the option of using a non-static time delay in the Smith predictor and Kalman filter which would improve performance. The main downside of this option is that the image would have to be cropped in order to remove the LEDs. The reasons this option was not pursued in this project was in part the time constraint but also the physical limitations of this drone model.

4.3 Controlling the Velocity instead of the Angle

The choice to control the velocity instead of controlling the angle was a practical decision. It was done because it was not possible to increase the angle without also measuring the current drone heading angle, in relation to the north. If the drone had a current heading angle at 70°, and it was desired to increase it by 30°, then the input to the function would have to be 100°. This caused issues because not only were these heading angles received from the telemetry data (which was different from the images) with a different frequency, they were also sent with a different delay and could therefore not be synchronized to the images with the tools that were at hand.

The main downside of controlling the velocity is that if the computer (or any other part of the process) gets stuck just as it is sending a high velocity signal then the camera might turn too fast and lose the target. If this project would be done on another drone then investigating the option of controlling the angle instead of the velocity would be advised.

4.4 Identifying inaccuracies in the Smith Predictor

Previously in the report simulated results have only been shown in the case of a perfect model of the system. However, it is very important to understand how a wrong model and delay estimation affects the full system, in order to recognize what might be wrong with the model when adjusting and fine tuning.

In order to more easily explain the effects of an inaccurate model, a couple parameters are introduced as well as Figure 4.1 to easier understand them. In the process there are two delays: the computer-to-drone delay L and the drone-to-computer delay t_d, where the full delay T_d is given by 4.1.

$$T_d = L + t_d \tag{4.1}$$

The delay L_e denotes the error in approximating the delay L, where a perfect approximation has L_e=0. The delay t_e denotes the error in approximating the delay t_d, where a perfect approximation has t_e=0. The process and model are in this case just an integrator, and the factor K is used to represent an erroneous approximation of the model where K=1 represents a perfect approximation.

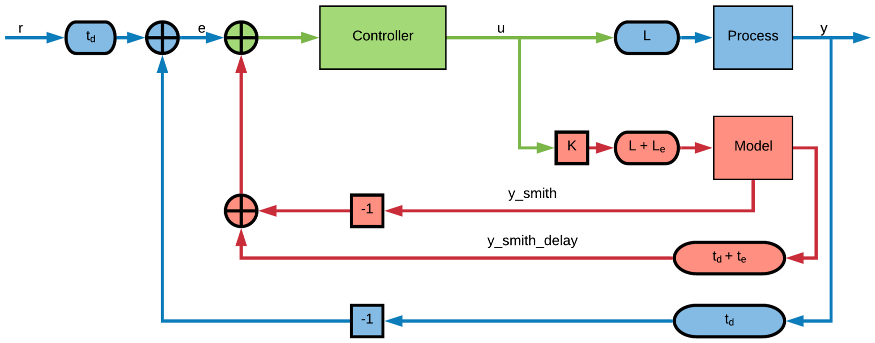

Figure 4.1 The Smith predictor shown in more detail for this system

When K>1 then the controller will give a smaller control signal u than what is actually needed, which can be seen in Figure 4.2. After the delay the controller receives the difference between the real process and Smith predictor and adjusts. When K<1 then the controller acts similarly, as seen in Figure 4.3, however this time the controller will send a larger signal than what is actually needed and will cause the real process to overshoot.

If instead, the model of the system is approximated correctly but the the full delay T_d have been approximated poorly, then a different issue arises. What happens is that at first the Smith predictor will correctly predict how the (in this case) camera will move. However, it will then receive a mismatched comparison between the real measurement and Smith predictor, because the delay approximation is wrong, and cause a "bump" to be sent into the controller which is then acted upon. This process of following the reference perfectly at first, followed by a "bump," is shown in Figures 4.4 and 4.5.

Additionally, if the full delay T_d has been successfully approximated but been allocated wrongfully so that $L_e = -t_e$, and $L_e \neq 0$, then another issue can be seen in Figure 4.6. In this case the comparison between the delayed real measurement and delayed Smith predictor value are equal and cancel each other out, which means that

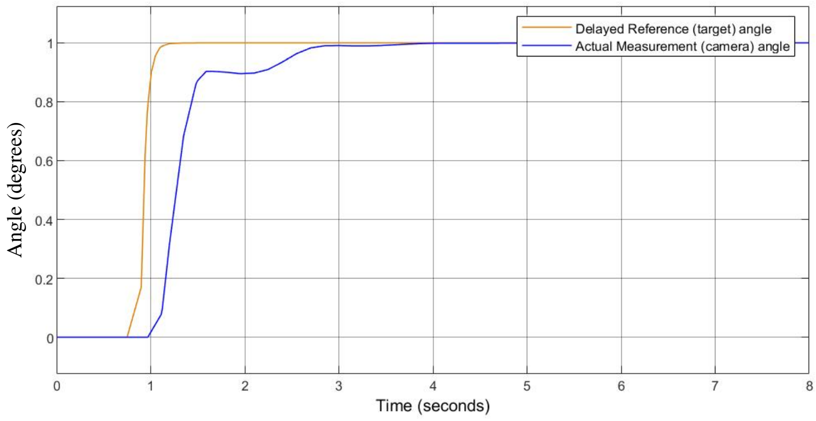

Figure 4.2 The delayed reference at the time it is received compared to the real camera movement for K=1.1

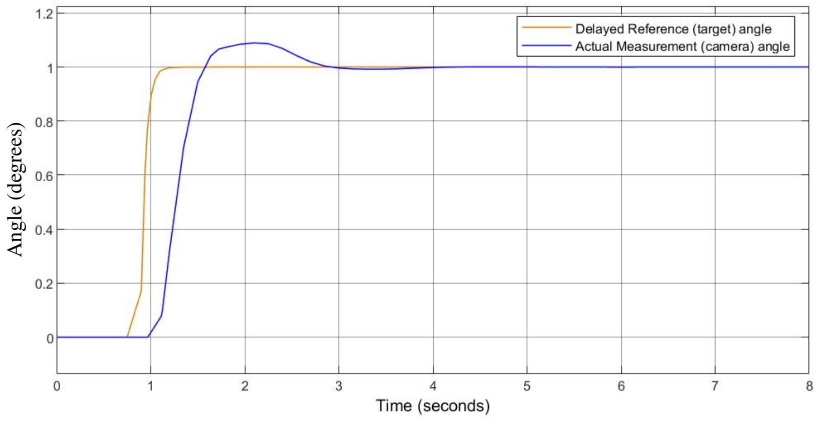

Figure 4.3 The delayed reference at the time it is received compared to the real camera movement for K=0.9

all that remains is the innermost loop with the Smith predictor with only the delay L+L_e. This causes a too fast controller to act on a too slow system and the result is an overshoot. This is the reason a Smith predictor is used so that the delay can be artificially "removed" and allow a much faster controller. If instead L_e is negative then a mismatch between the real process and process model is caused (since L is considered to be part of the process) and although it does not result in an overshoot it does result in a slower system as seen in Figure 4.7.

Understanding Figures 4.2 to 4.7 is important in order to understand not only

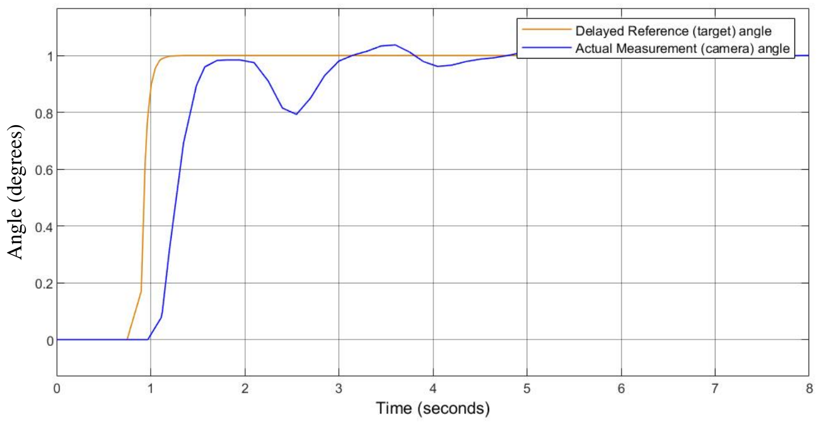

Figure 4.4 The delayed reference at the time it is received compared to the real camera movement, for a real process T_d=960ms and t_e=100ms

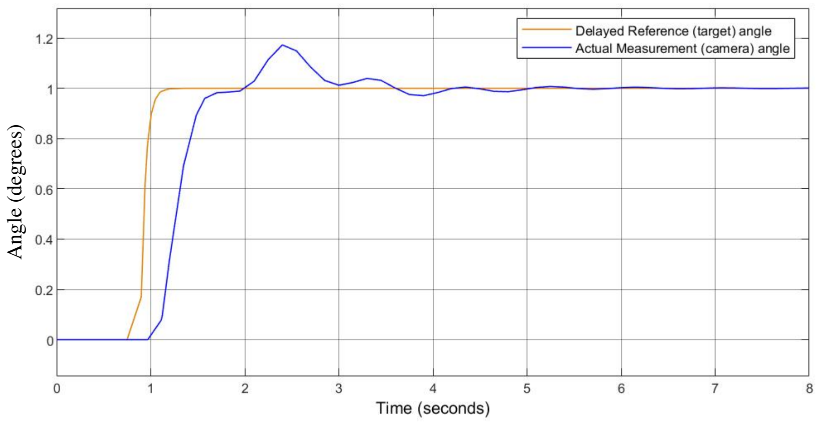

Figure 4.5 The delayed reference at the time it is received compared to the real camera movement, for a real process T_d=960ms and t_e=-100ms

how the Smith predictor works but also to identify where the issue is, if there is one, so that it can be fixed.

4.5 Effects of altering the Kalman filter

When designing a Kalman filter it is important to keep in mind that there is a trade-off between speed and sensitivity. If Figures 3.13 and 3.14 are taken into account,

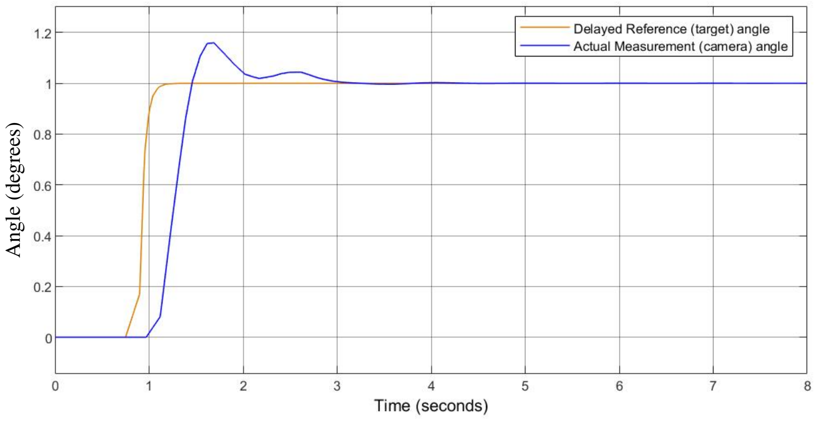

Figure 4.6 The delayed reference at the time it is received compared to the real camera movement, for a real process T_d=960ms where L=70ms and L_e=70ms

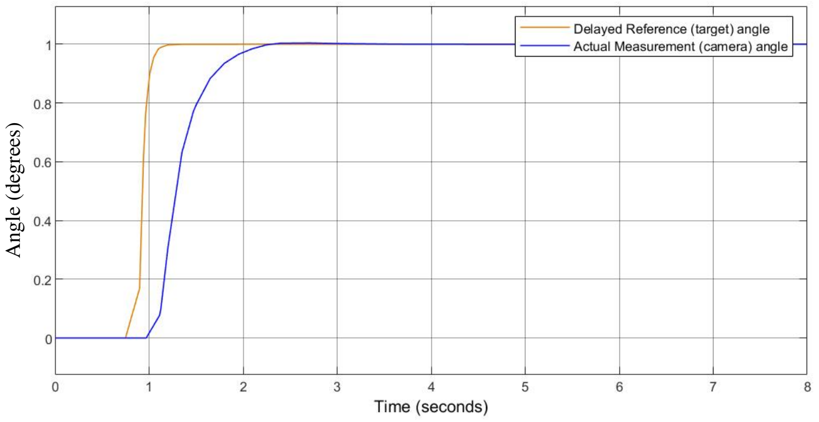

Figure 4.7 The delayed reference at the time it is received compared to the real camera movement, for a real process T_d=960ms where L=70ms and L_e=-70ms

where K_1 and K_2 are compared, then it can be seen that K_2 is much faster and better at following the arbitrary measurement. However, this makes it more sensitive to noise and small changes that are better to be ignored. Also, it can clearly be seen how fast the velocity state $\hat{x}_2$ reacts in K_2. This is not inherently a bad thing, but in this application where there is a large delay it can cause overshoot as seen in Figure 3.16. What happens there is that the Kalman filter prediction will think that whatever velocity it is detecting will continue to have that velocity for as long as the delay is. So when the target makes a quick turn the Kalman filter will cause an overshoot. While K_2 was shown before, a new Kalman filter K_4 can be shown where

$$K_2 = \begin{bmatrix} 4 \\ 10 \end{bmatrix}$$

$$K_4 = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

What this shows in Figure 4.8 (using the same arbitrary measurement as in Figure 3.14) is that it is fairly effective at following the measurement with the position $\hat{x}_1$ while the velocity $\hat{x}_2$ changes much slower than when K_2 is used.

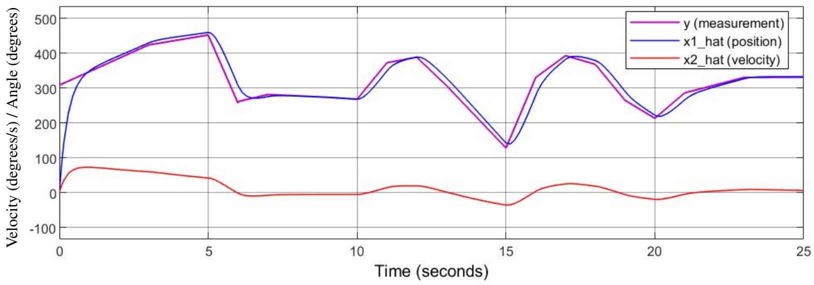

Figure 4.8 Kalman filter states given by K_4

The benefit of this Kalman filter, that is significantly slower at detecting a change in velocity, is that when it is used to predict the position of the target it can provide much smoother camera movement for a target that suddenly stops, as is shown in Figure 4.10. However, since it is so slow at detecting a change in velocity this creates a greater overshoot for a target turning back and forth as well as taking longer time at "catching up" with the target as seen in Figure 4.9. These figures follow the same simulated target (and the same delay and sampling time) as Figures 3.15 and 3.16, and the difference between them showcase the trade-off that must be made when choosing parameters for the Kalman filter.

Had more time been available then a suggested way of tuning the Kalman filter would be to use a video run of the drone and use that same run for multiple tests. Then use the Kalman filter to predict where the target will be based on images as old as the time delay t_d and use it to draw a visible dot where it thinks the target should be. By doing this the dot will hopefully always be on top of the target and it becomes easier to fine tune the Kalman filter. Since the Kalman filter uses the Smith predictor values, these would likely have to be saved along with the images of the video with corresponding time stamps.

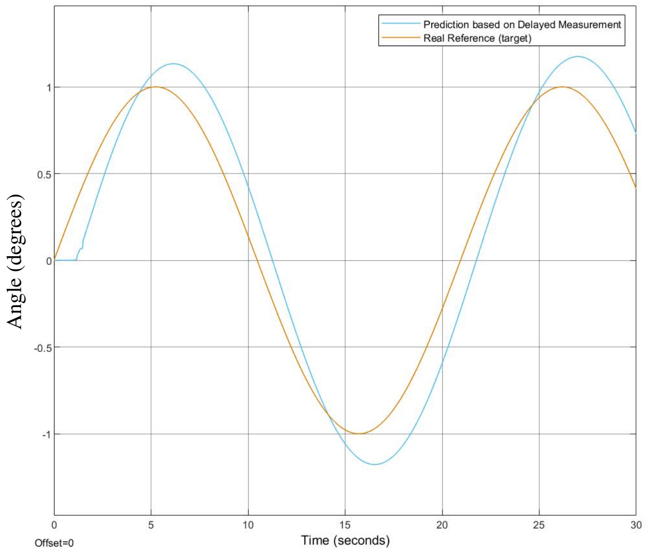

Figure 4.9 Kalman filter prediction given by K_4 for a target moving in a sinusoidal shape

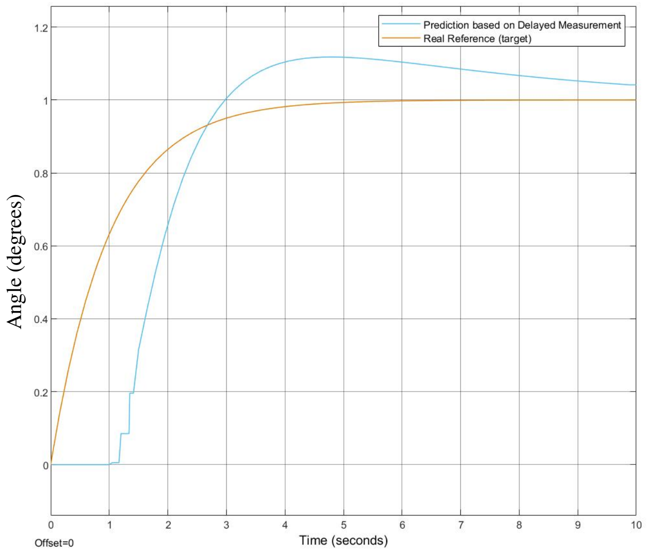

Figure 4.10 Kalman filter prediction given by K_4 for a target making a quick turn

4.6 The Kalman filter's reliance on the Smith predictor

Something interesting in this project was the reliance the Kalman filter had on the Smith predictor. Normally in an application such as this (and this is actually what was done in the project) someone would want to track where the reference is heading and attempt to predict where the target actually is at that moment based on the delayed measurement.

Since the thing being measured is the difference in angle between the target r and camera y, the only thing being received is the error. There is no information in the received images about the angle of the target or the angle of the camera. Due to this it was initially believed that it would be possible to apply the Kalman filter on the error $e(t-t_d)$ and use it to predict the error $\hat{e}(t)$. At first this seemed reasonable because it would mean that if the error is increasing the Kalman prediction will believe that it will continue to increase and apply a control signal accordingly. The same principle goes for a decreasing error, in which if old images are seen were the error is decreasing then it is likely that the real error will be gone before the received delayed error shows this, where once again the Kalman filter can be used to predict this and the controller can act accordingly.

However, when the Kalman filter was applied on the error in simulation the results looked the same as before the filter was introduced. This can be explained by the fact that neither the target angle r or the camera angle y remains static, so the Kalman filter also introduces an unwanted derivative term of the camera angle y. The following equations use the same color coding as is used in Figure 2.4 for easier understanding, where e_{tot} will be used to denote the input signal to the controller.

By looking at the process without the Kalman filter but with the Smith predictor, it can clearly be seen in Equation 4.3 that the real time camera position is following the delayed position of the target. This allows the camera to move quickly but it will never catch up with a moving target.

$$e_{tot}(t) = r(t-t_d) - y(t-t_d) + y_{smith}(t-t_d) - y_{smith}(t) \tag{4.2}$$

$$e_{tot}(t) \approx r(t-t_d) - y_{smith}(t) \tag{4.3}$$

If instead the Kalman filter is introduced then an unwanted term appears. In Equation 4.9 it becomes easy to see that the only thing that should be sent into the controller is the error between the target and camera without any delay, but the unwanted term $\dot{y}(t-t_d)\cdot t_d$ is subtracted.

$$e_{tot}(t) = \hat{x_1}(t-t_d) + \hat{x_2}(t-t_d)\cdot t_d + y_{smith}(t-t_d) - y_{smith}(t) \tag{4.4}$$

$$e_{tot}(t) = \hat{e}(t-t_d) + \dot{\hat{e}}(t-t_d)\cdot t_d + y_{smith}(t-t_d) - y_{smith}(t) \tag{4.5}$$

$$e_{tot}(t) = \hat{r}(t-t_d) - \hat{y}(t-t_d) + (\dot{\hat{r}}(t-t_d) - \dot{\hat{y}}(t-t_d))\cdot t_d + y_{smith}(t-t_d) - y_{smith}(t) \tag{4.6}$$

$$e_{tot}(t) \approx \hat{r}(t-t_d) + (\dot{\hat{r}}(t-t_d) - \dot{\hat{y}}(t-t_d))\cdot t_d - y_{smith}(t) \tag{4.7}$$

$$e_{tot}(t) \approx \hat{r}(t) - y_{smith}(t) - \dot{\hat{y}}(t-t_d)\cdot t_d \tag{4.8}$$

$$e_{tot}(t) \approx e(t) - \dot{\hat{y}}(t-t_d)\cdot t_d \tag{4.9}$$

What this shows is that if the Kalman filter is applied on the error signal then the unwanted term negates any ability it might have to predict the future. Therefore this shows that in this system where only the error can be measured the Kalman filter becomes reliant on a functional Smith predictor. This puts further emphasis on the importance of an accurate model of the system and accurate measurements of the time delay.

4.7 The Trust Factor

The trust factor for the pitch process was very high ($T_{fac} = 0.8$) and can easily be explained by a slightly imperfect model or poor estimation of the time delay. However, for the yaw process it was very low ($T_{fac} = 0.4$) and this circles back to previous problems. It puts emphasis on the importance of an accurate model and also that a proper method of continuously measuring the time delay is something very integral to all aspects of this project and the quality of the video. A hypothesis for why the trust factor was much lower for the yaw process than the pitch process is that the way the drone stops or decreases its angular velocity in the yaw direction was not modelled properly and that it might move further than the Smith predictor estimates, which matters to the Kalman filter since only the error could be measured and the position of the target is reliant on the Smith predictor. It should be noted though that since the process worked as well as it did in the pitch direction, it can be seen as a proof of concept and that improvements to the yaw process should be possible.

4.8 Comparing the Control Methods

In this section graphs are shown for the old controller (a P-controller), the PD controller, the PD controller with Smith predictor, and the PD controller with Smith predictor and Kalman filter. They are compared for a step response, where the target is standing still and the camera starts looking in wrong direction, and for a moving target, respectively.

Tracking a stationary target

Figure 4.11 shows the different controllers in comparison for the gimbal pitch process when the target is standing still. It can clearly be seen how by only adding a small derivative part to the controller much was not changed. The main issue was still the large time delay which slowed down the system. The addition of the Smith predictor did however speed up the process and that can clearly be seen in this comparison. However, the main downside of the Kalman filter can also be seen here. Since the controller is practically following a step response it thinks that the target has moved very far away very quickly and will continue to do so, which is why the overshoot happens. The Kalman filter works best when following a target with a constant velocity.

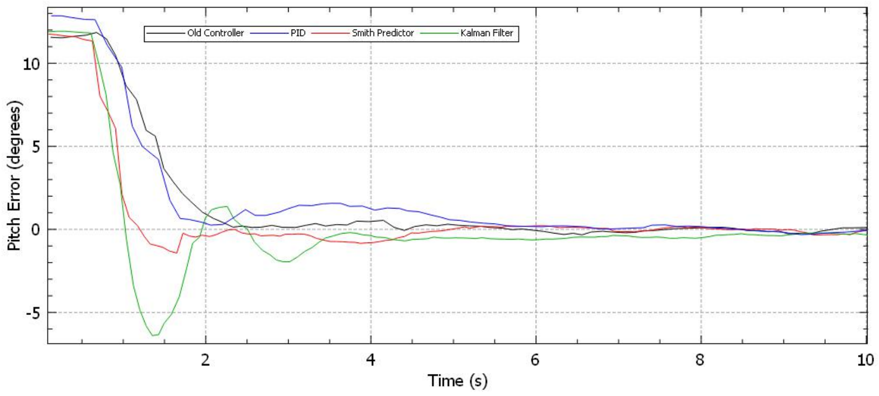

Figure 4.11 Error comparison in pitch direction for various control methods for stationary target

When the same experiment was carried out for the drone yaw process similar results were obtained as can be seen in Figure 4.12. With the Kalman filter implemented there was still an overshoot as this test showcases the weakness of the Kalman filter prediction. Although the Smith predictor was not faster for this process it was still an improvement as the old controller and the PD controller had larger overshoot. It should be noted that the target was in the same position for all trials, however the image detection program is not perfect and the target was therefore perceived to be in slightly different positions.

Tracking a moving target

The controllers were compared for a target moving back and forth. The PD controller was not used in this case since it was so similar to the old controller. The trials had some variation since there were issues in attempting to standardize the

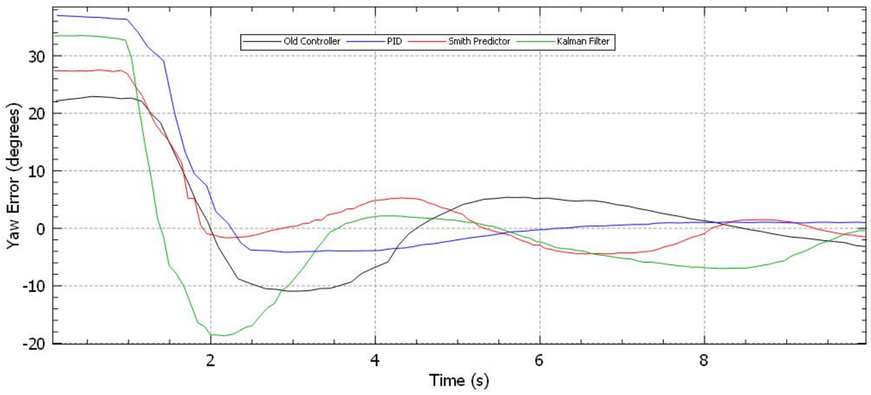

Figure 4.12 Error comparison in yaw direction for various control methods for stationary target

tests, such as the image recognition program being "too smart" and not recognizing an image of skier on a projector screen or a printed photograph of a skier.

The tests for the gimbal pitch process can be seen in Figure 4.13 where the benefit of the Kalman filter is clearly show. The ability to predict the velocity and look ahead of received information allowed it to minimize the error. The Smith predictor works well enough, and can be seen to be an improvement over the old controller that was not as well designed.

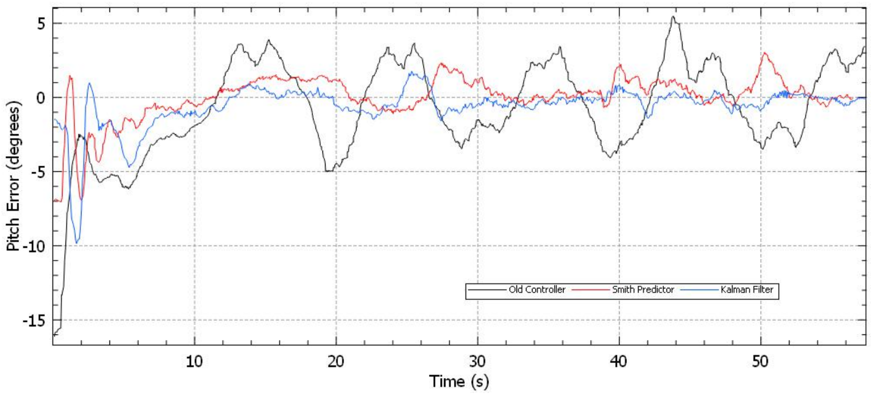

Figure 4.13 Error comparison in pitch direction for various control methods for target moving back and forth

Although the drone yaw process is less accurate, the same conclusion could be drawn from the corresponding tests. In Figure 4.14 it can be seen how there was always an error for the old controller and the new controller with a Smith predictor. That is because the target was moving and the controllers were using delayed measurements. When the Kalman filter was applied in order to estimate the position of the target there was a smaller error. However, there were also larger overshoots.

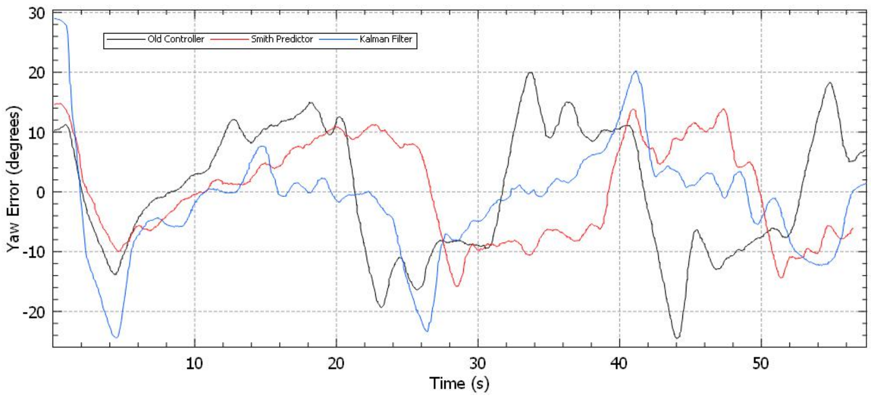

Figure 4.14 Error comparison in yaw direction for various control methods for target moving back and forth

It should be noted how the yaw process is less still than the pitch, and that the yaw is controlled with four motors as well as constantly affected by wind. The Kalman filter could possibly be adapted to react faster or slower to the shakiness. However as discussed previously this both changes come with their own pros and cons. Furthermore, when the process starts to stabilize it is not actually changing very fast or very much even though the graphs might at a quick glance suggest otherwise.

5
Conclusion

5.1 Completion of Objectives

Due to certain parts of the project taking longer than anticipated, there was no time to secure robustness to strong winds or implement a program to take into account the varying velocity of the target. However, the project was a success in terms of minimizing the impact of the large delay, implementing a prediction program, modelling the system, and keeping a target with constant velocity centered in the screen.

The impact of the delay was successfully mitigated through the implementation of a Smith predictor to allow a faster controller and the implementation of a Kalman filter that was used to predict the position of the target. Although improvements could be made to the model of how the drone turns in the yaw direction it was still good enough to keep the target centered in the video. The model of how the camera moves in the pitch direction worked very well, and provided excellent results.

Overall the project was a success but with room for improvement.

5.2 Future Research

To improve the results of this project, there are some very clear goals that must be achieved. The best method of improving the performance is to simply eliminate the time delay. The setup used in this project was far from optimal and the Sony team is already working on removing steps in the transfer of information by using a more accommodating drone, where a computer can communicate with the drone directly. This is likely to have a huge impact in minimizing the time delay (currently hypothesized by the team that the full delay will be decreased from around 1s to 0.3s). Minimizing the time delay is important because not only will the controller start receiving information earlier but the Kalman filter prediction will not have to "look" as far into the future, making the prediction more accurate.

The second best method of improving performance is to make sure it is possible to measure the time delay. As have been shown, the time delay delay estimation only has to be wrong by 100ms for the end result to give unsatisfactory results (and that is just taking the Smith predictor into account).

Additionally, the Smith predictor will only work well with an accurate model of the system. For that reason it is important that moving forward the system is modeled both for an increase and decrease of velocity. Furthermore, since the gimbal pitch process was easier to control accurately than the drone yaw process, the ability to implement a fast controller for the gimbal yaw (accompanied by a slow controller for the full drone yaw) should be looked into.

In conclusion, these are the areas that are the most important to investigate if this project is to be continued with improved performance.

6

References

[1] DJI. *Mavic 2 - Specifications, FAQs, Videos, Tutorials, Manuals*. URL: https://www.dji.com/se/mavic-2/info#specs. (accessed: 23.04.2020).

[2] DJI Store. *Mavic 2 Zoom Aircraft (Excludes Remote Controller and Battery Charger)*. URL: https://store.dji.com/product/mavic-2-zoom-aircraft?gclid=Cj0KCQjw0YD4BRD2ARIsAHwmKVlxTZb33mvEpDPQ6-ayEQiZHxvmAz3VwAajzHE4XybTp2d0pdXwUtMaAo-1EALw_wcB. (accessed: 04.07.2020).

[3] D. A. Willis. *DJI Mavic 2 Pro and Mavic 2 Zoom Camera Drones*. URL: https://ascmag.com/articles/the-dji-mavic-2-pro-and-mavic-2-zoom-camera-drones. (06.09.2018).

[4] DJI Store Arizona. *Mavic 2 Remote Controller*. URL: https://djistoreaz.com/product/mavic-2-replacement-remote-controller/. (accessed: 04.07.2020).

[5] DJI. *DJI Developer*. URL: https://developer.dji.com/. (accessed: 24.04.2020).

[6] DJI. *Mobile SDK Documentation*. URL: https://developer.dji.com/mobile-sdk/documentation/introduction/index.html. (accessed: 24.04.2020).

[7] OpenCV Team. *OpenCV*. URL: https://opencv.org/. (accessed: 22.05.2020).

[8] J. C. Redmon. *Yolov3*. URL: https://pjreddie.com/darknet/yolo/. (accessed: 22.05.2020).

[9] HP Development Company, L.P. *HP EliteBook 840 G3 Notebook PC Specifications*. URL: https://support.hp.com/us-en/document/c05259054#AbT1. (accessed: 23.05.2020).

[10] The MathWorks, Inc. *System Identification Toolbox*. URL: https://www.mathworks.com/products/sysid.html. (accessed: 24.05.2020).

[11] The MathWorks, Inc. *fminsearch*. URL: `https://se.mathworks.com/help/matlab/ref/fminsearch.html`. (accessed: 01.06.2020).

[12] Wikipedia. *RGB color model*. URL: `https://en.wikipedia.org/wiki/RGB_color_model`. (accessed: 30.05.2020).

[13] The MathWorks, Inc. *PID Tuner*. URL: `https://se.mathworks.com/help/control/ref/pidtuner.html`. (accessed: 01.06.2020).

[14] K.-E. Årzén. *Real-Time Control Systems*. Department of Automatic Control, Lund University, Lund, Sweden, 2014. Chap. 10.

[15] T. Glad, L. Ljung. *Reglerteori: Flervariabla och olinjära metoder*. Studentlitteratur, Lund, Sweden, 2017. Chap. 15.

[16] A. Cervin. *Smith Predictor*. URL: `http://archive.control.lth.se/media/Education/EngineeringProgram/FRTN10/2018/L12_notes.pdf`. (accessed: 01.06.2020).

[17] T. Hägglund. *Reglerteknik AK Föreläsningar*. Department of Automatic Control, Lund University, Lund, Sweden, 2017. Chap. 9.

[18] B. Wittenmark, K. J. Åström, K.-E. Årzén. *Computer Control: An Overview*. Department of Automatic Control, Lund University, Lund, Sweden, 2016. Chap. 3.

[19] The MathWorks, Inc. *Continuous to Discrete*. URL: `https://se.mathworks.com/help/control/ref/c2d.html`. (accessed: 01.06.2020).

[20] A. Cervin. *Example of Midranging control*. URL: `http://archive.control.lth.se/media/Education/EngineeringProgram/FRTN10/2018/L05_notes.pdf`. (accessed: 22.06.2020).

A

PersonTracker.h

This file is the header file `PersonTracker.h` where the variables and the static parameters values are defined.

```
#ifndef DRONECONTROLLER_PERSONTRACKER_H
#define DRONECONTROLLER_PERSONTRACKER_H

#include <opencv2/opencv.hpp>
#include <opencv2/tracking.hpp>
#include <command/MessageContainer.h>
#include <command/SpeedController.h>
#include "YoloDetector.h"
#include <command/Actions/IAction.h>
#include <command/Actions/TakeOff.h>
#include <memory>
#include <vector>
#include <iostream>

namespace Tracking {


    class PersonTracker {

    public:
        PersonTracker(int width, int height, std::shared_ptr<
    Command::MessageContainer> message);

        bool processFrame(cv::InputOutputArray frame);

        int resetTracker();

        virtual ~PersonTracker();

    private:
        cv::Ptr<cv::Tracker> tracker;
        cv::Vec3b color;
        bool trackingPerson = false;
        cv::Rect2d roi;
        std::vector<cv::Rect> boxes;
        YoloDetector detector;
```

```cpp
        int width; //640 px
        int height; //380 px

        int blue = 0;
        int green = 0;
        int red = 0;
        std::chrono::milliseconds endDelay;

        void drawDebugLines(const cv::_InputOutputArray &frame, double offset_x, double offset_y) const;

        //Smith Predictor
        double smithPitch = 0;
        double smithPitchDelayed = 0;
        double smithYaw = 0;
        double smithYawDelayed = 0;
        std::vector<double> smithPitchV;
        std::vector<double> smithYawV;
        std::vector<double> smithTime;
        // Read the item of a position in the vector
        double readVector(std::vector<double> vector, int prev){
            // get previous item
            if(prev < 0){return 0;}
            else{
                double item = vector[prev];
                return item;
            }
        }

        //Kalman filter
        double inputKalmanGimbal = 0;
        double inputKalmanYaw = 0;

        double yGimbal = 0;
        double yGimbalNew = 0;
        double x1P = 0;
        double x2P = 0;
        double x1PNew = 0;
        double x2PNew = 0;
        double k1P = 5;
        double k2P = 8;

        double yYaw = 0;
        double yYawNew = 0;
        double x1Y = 0;
        double x2Y = 0;
        double x1YNew = 0;
        double x2YNew = 0;
        double k1Y = 5;
        double k2Y = 8;

        int printstate =0;
        std::shared_ptr<Command::MessageContainer> mMessage;
```

```
    std::unique_ptr<Command::SpeedController>
mSpeedController;
    double Td = 0; //Time image delay
    double sampleTime = 0; //Everytime a new measurement is
received and processed
    double Tex = 0; //Total program execution time

    //PID pitch vel control parameters
    double Ts = sampleTime;
    double Ppv = 2; //Proportional coefficient 0.5 Smith: 2.5
 Kalman: 2
    double Ipv = 0; //Integral coefficient
    double Dpv = 0.2; //Derivative coefficient 0.1 Smith: 0.2
 Kalman:0.2
    double Npv = 3; //Filter coefficient derivative zoom:
    //Coefficients for the discretization
    double b0pv = Ppv*(1+Npv*Ts) + Ipv*Ts*(1+Npv*Ts) + Dpv*
Npv;
    double b1pv = -(Ppv*(2+Npv*Ts) + Ipv*Ts + 2*Dpv*Npv);
    double b2pv = Ppv + Dpv*Npv;
    double a0pv = (1 + Npv*Ts);
    double a1pv = -(2 + Npv*Ts);
    double a2pv = 1;

    //PID yaw velocity control parameters
    double Pyv = 2.5; //Proportional coefficient 0.5 Smith
:2.5  Kalman:2.5
    double Iyv = 0; //Integral coefficient zoom:
    double Dyv = 0.2; //Derivative coefficient 0.1 Smith:0.2
 Kalman:0.2
    double Nyv = 3; //Filter coefficient derivative zoom:
    //Coefficients for the discretization
    double b0yv = Pyv*(1+Nyv*Ts) + Iyv*Ts*(1+Nyv*Ts) + Dyv*
Nyv;
    double b1yv = -(Pyv*(2+Nyv*Ts) + Iyv*Ts + 2*Dyv*Nyv);
    double b2yv = Pyv + Dyv*Nyv;
    double a0yv = (1 + Nyv*Ts);
    double a1yv = -(2 + Nyv*Ts);
    double a2yv = 1;

    double velPitchControl = 0;
    double velYawControl = 0;

    //Velocity Pitch
    double pepv = 0; //previous error PID pitch
    double peppv = 0; //previous previous error PID pitch
    double pupv = 0; //previous pitch velocity send to the
process
    double puppv = 0; //previous previous pitch velocity send
 to the process
    //Velocity yaw
    double yepv = 0; //previous error PID yaw
    double yeppv = 0; //previous previous error PID yaw
```

```
        double yupv = 0; //previous yaw velocity send to the process
        double yuppv = 0; //previous previous yaw velocity send to the process

        /**
        * Get the error angle of the gimbal pitch between the reference in the screen and the target
        * boxVerticalCenter - The amount of pixels between the target and the screen center (targetPixelCenter-screenPixelCenter).
        * fieldOfView - The FOV for the drone. For Mavic 2 Zoom it is 83 degrees and for Mavic 2 Pro it is 77 degrees.
        * screenPixelHeight - The max amount of pixels vertically on the screen. We use 380.
        */
        double getPitchAngleError(double boxVerticalCenter, double fieldOfView, double screenPixelHeight) {
        double oppositeKathetus = screenPixelHeight / 2;
        double halfFieldOfView = fieldOfView / 2;
        double adjacentKathetus = oppositeKathetus / tan(Utils::deg2rad(halfFieldOfView));
        double result = Utils::rad2deg(atan(boxVerticalCenter/adjacentKathetus));
        return result;
        }

        /**
        * Get the error angle of the yaw between reference in the screen and the target
        * boxHorizontalCenter - The amount of pixels between the target and the screen center (targetPixelCenter-screenPixelCenter).
        * fieldOfView - The FOV for the drone. For Mavic 2 Zoom it is 83 degrees and for Mavic 2 Pro it is 77 degrees.
        * screenPixelWidth - The max amount of pixels horizontally on the screen. We use 640.
        */
        double getYawAngleError(double boxHorizontalCenter, double fieldOfView, double screenPixelWidth) {
        double oppositeKathetus = screenPixelWidth / 2;
        double halfFieldOfView = fieldOfView / 2;
        double adjacentKathetus = oppositeKathetus / tan(Utils::deg2rad(halfFieldOfView));
        double result = Utils::rad2deg(atan(boxHorizontalCenter/adjacentKathetus));
        return result;
        }
    };
}//


#endif //DRONECONTROLLER_PERSONTRACKER_H
```

B

PersonTracker.cpp

`PersonTracker.cpp` is included to show how the controller is implemented.

```cpp
#include <utils/Utils.h>
#include "utils/Functions.h"
#include "PersonTracker.h"
#include <Networking/Telemetry.h>
#include <core/SimpleStateMachine.h>
#include <core/GlobalVariables.h>
#include "command/Actions/ThesisTesting.h"
#include "video/FeedServer.h"


namespace Tracking {

    PersonTracker::PersonTracker(int width, int height, std::shared_ptr<Command::MessageContainer> message) :
            width(width), height(height), mMessage(std::move(message)) {
        detector.init(width, height);
        mSpeedController = std::make_unique<Command::SpeedController>(width);
    }

    PersonTracker::~PersonTracker() = default;

    int PersonTracker::resetTracker() {
        trackingPerson = false;
        return 0;
    }

    bool PersonTracker::processFrame(cv::InputOutputArray frame) {
        Utils::print("Frame");
        //Take color BGR of the frame
        color = frame.getMat().at<cv::Vec3b>(height/2,width/2); //center bottom pixel (y,x)
        blue = color.val[0];
        green = color.val[1];
        red = color.val[2];
```

```
  //check if the image has change
        if(blue > 50 && green > 50 && red > 50 && Td==0){ //Check if the image is darker
            endDelay = std::chrono::duration_cast< std::chrono::milliseconds >(
            std::chrono::system_clock::now().time_since_epoch());
            Td = (endDelay.count() - Core::GlobalVariables::getInstance().getTime() - 200)/1000; //Time delay image in seconds
        }
        Td=0.7; //Estimated complete time delay
        Utils::print("Colour: " + std::to_string(blue) + "," + std::to_string(green) + "," + std::to_string(red) + " Time delay: " + std::to_string(Td));

        std::chrono::milliseconds start = std::chrono::duration_cast< std::chrono::milliseconds >(
            std::chrono::system_clock::now().time_since_epoch());
        if (!trackingPerson) {
            detector.processFrame(frame, boxes); //Detect people in the frame
            Utils::print("Frame processed");
            std::chrono::milliseconds end = std::chrono::duration_cast< std::chrono::milliseconds >(
                std::chrono::system_clock::now().time_since_epoch());
            Utils::print("Time to do detection: " + std::to_string(end.count()-start.count()) + " ms");
        } else {
            trackingPerson = tracker->update(frame, roi); //Update the position of the target
            Utils::print("Frame and target processed");

            std::chrono::milliseconds end = std::chrono::duration_cast< std::chrono::milliseconds >(
                std::chrono::system_clock::now().time_since_epoch());
            double Tp = end.count()-start.count();
            Utils::print("Time to update detection and control: " + std::to_string(end.count()-start.count()) + " ms");
            Ts=Tp/1000; //Sample time in seconds
            Tex = Tex + Ts; //Total program execution time

            //Print in the Measurements Log file
            //Utils::MeasurementsLog::getInstance().log("Time to update detection and control: " + std::to_string(end.count()-start.count()) + " ms");

            if (!trackingPerson) { //If target lost, find and detect it again
                tracker.reset();
                return trackingPerson;
            }
```

```cpp
    cv::rectangle(frame, roi, cv::Scalar(255, 0, 0), 2, 1);

    //debugline, follow roi
    auto offset_x = roi.x + (roi.width / 2); //get the x center of roi from the left of the screen.
    auto offset_y = roi.y + (roi.height / 2); //get the y center of roi from the top of the screen.
    drawDebugLines(frame, offset_x, offset_y);
    auto diff_yaw = offset_x - (width / 2); //the amount of pixels from the center of the screen. Width = 640px
    auto diff_gimbal = offset_y - (height / 2); //height = 380px


    auto controllerData = mMessage.get()->getData();
    //GIMBAL
 double pitchAngleError = PersonTracker::getPitchAngleError(diff_gimbal, 48, height); //Error pitch center screen - center reference

    //Smith predictor (for both PITCH and YAW)
    smithTime.push_back(sampleTime); //Add a value at the end of the vector
    smithPitchV.push_back(smithPitch);
    smithYawV.push_back(smithYaw);
    int pos = 0, prev = 0;
    double sumTs = 0;
    //increse the previous sample time position until the sum achieves the time delay
    while(prev >= 0){
        prev = (smithTime.size() - pos);//vector.size print the size of the vector
        sumTs = sumTs + readVector(smithTime, prev);
        pos++;
        if(sumTs >= Td){break;}
    }
    if(readVector(smithPitchV, prev) == 0){//Take the previous values corresponding to the time delay
        smithPitchDelayed = 0;
        smithYawDelayed = 0;
    }else{ //interpolate to get the accurate value
        smithPitchDelayed = readVector(smithPitchV, prev-1) + (readVector(smithPitchV, prev) - readVector(smithPitchV, prev-1))/
              readVector(smithTime, prev) * (Td - (sumTs - readVector(smithTime, prev)));
        smithYawDelayed = readVector(smithYawV, prev-1) + (readVector(smithYawV, prev) - readVector(smithYawV, prev-1))/
              readVector(smithTime, prev) * (Td - (sumTs - readVector(smithTime, prev)));
    }
```

```cpp
        if(smithTime.size() == 30){ //Erase the first value of the vector to avoid it colapse
        smithTime.erase(smithTime.begin());
        smithPitchV.erase(smithPitchV.begin());
        smithYawV.erase(smithYawV.begin());
        }

        //Kalman filter Pitch
        inputKalmanGimbal = -pitchAngleError*0.95 + smithPitchDelayed; //0.95 because of the constant change of the ROI
        //Taylor
        yGimbal = inputKalmanGimbal;
        x1P = x1PNew;
        x2P = x2PNew;
        x1PNew = x1P*((k1P*k1P*Ts*Ts)/2 - (k2P*Ts*Ts)/2 - k1P*Ts + (k1P*k2P*Ts*Ts*Ts)/6 + (k1P*Ts*(- k1P*k1P*Ts*Ts + k2P*Ts*Ts))/6 + 1) +
                yGimbal*(k1P*(Ts + (Ts*((k1P*k2P*Ts*Ts*Ts)/6 + (k1P*Ts*(- k1P*k1P*Ts*Ts + k2P*Ts*Ts))/6))/4 - (k1P*Ts*Ts)/2 - (Ts*((k2P*Ts*Ts)/2 - (k1P*k1P*Ts*Ts)/2))/3) -
                k2P*((k1P*Ts*Ts*Ts)/6 + (Ts*((k2P*Ts*Ts*Ts)/6 - (k1P*k1P*Ts*Ts*Ts)/6))/4 - Ts*Ts/2)) + x2P*((k1P*k1P*Ts*Ts*Ts)/6 - (k1P*Ts*Ts)/2 - (k2P*Ts*Ts*Ts)/6 + Ts);

        x2PNew = x1P*((k1P*k2P*Ts*Ts)/2 - k2P*Ts + (k2P*Ts*(- k1P*k1P*Ts*Ts + k2P*Ts*Ts))/6) + x2P*((k1P*k2P*Ts*Ts*Ts)/6 - (k2P*Ts*Ts)/2 + 1) +
                yGimbal*(k2P*((k1P*k2P*Ts*Ts*Ts*Ts)/24 - (k2P*Ts*Ts*Ts)/6 + Ts) + k1P*((k1P*k2P*Ts*Ts*Ts)/6 - (k2P*Ts*Ts)/2 + (k2P*Ts*Ts*(- k1P*k1P*Ts*Ts + k2P*Ts*Ts))/24));
        yGimbalNew = x1PNew + x2PNew*Td*0.8;

        Utils::print("diff_gimbal: " + std::to_string(diff_gimbal) + " x1PNew: " + std::to_string(x1PNew));

        //PID for the pitch velocity discretized
        //Coefficients for the discretization
        b0pv = Ppv*(1+Npv*Ts) + Ipv*Ts*(1+Npv*Ts) + Dpv*Npv;
        b1pv = -(Ppv*(2+Npv*Ts) + Ipv*Ts + 2*Dpv*Npv);
        b2pv = Ppv + Dpv*Npv;
        a0pv = (1 + Npv*Ts);
        a1pv = -(2 + Npv*Ts);
        a2pv = 1;
        sampleTime=Ts;

        //double pev = -pitchAngleError*0.95; //pitch error
        //double pev = -pitchAngleError*0.95 + smithPitchDelayed - smithPitch; //pitch error with Smith Predictor
        double pev = yGimbalNew - smithPitch; //pitch error with Smith Predictor
        double puv = (-a1pv*pupv - a2pv*puppv + b0pv*pev + b1pv*pepv + b2pv*peppv)/a0pv; //pitch gimbal velocity send to
```

```
 the drone
        velPitchControl = puv;
        peppv = pepv; //Update values
        pepv = pev;
        puppv = pupv;
        pupv = puv;

        smithPitch = (pupv+puv)/2*sampleTime + smithPitch; //
trapezoidal integration


      //YAW
        double yawAngleError = PersonTracker::
getYawAngleError(diff_yaw, 83, width);

        //Kalman filter Yaw
        inputKalmanYaw = yawAngleError + smithYawDelayed;
        yYaw = inputKalmanYaw;
        x1Y = x1YNew;
        x2Y = x2YNew;
        x1YNew = x1Y*((k1Y*k1Y*Ts*Ts)/2 - (k2Y*Ts*Ts)/2 - k1Y
*Ts + (k1Y*k2Y*Ts*Ts*Ts)/6 + (k1Y*Ts*(- k1Y*k1Y*Ts*Ts + k2Y*
Ts*Ts))/6 + 1) +
                yYaw*(k1Y*(Ts + (Ts*((k1Y*k2Y*Ts*Ts*Ts)/6 + (
k1Y*Ts*(- k1Y*k1Y*Ts*Ts + k2Y*Ts*Ts))/6))/4 - (k1Y*Ts*Ts)/2 -
 (Ts*((k2Y*Ts*Ts)/2 - (k1Y*k1Y*Ts*Ts)/2))/3) -
                k2Y*((k1Y*Ts*Ts*Ts)/6 + (Ts*((k2Y*Ts*Ts*Ts)/6
 - (k1Y*k1Y*Ts*Ts*Ts)/6))/4 - Ts*Ts/2)) + x2P*((k1Y*k1Y*Ts*Ts
*Ts)/6 - (k1Y*Ts*Ts)/2 - (k2Y*Ts*Ts*Ts)/6 + Ts);

        x2YNew = x1Y*((k1Y*k2Y*Ts*Ts)/2 - k2Y*Ts + (k2Y*Ts*(-
 k1Y*k1Y*Ts*Ts + k2Y*Ts*Ts))/6) + x2Y*((k1Y*k2Y*Ts*Ts*Ts)/6 -
 (k2Y*Ts*Ts)/2 + 1) +
                yYaw*(k2Y*((k1Y*k2Y*Ts*Ts*Ts*Ts)/24 - (k2Y*Ts
*Ts*Ts)/6 + Ts) + k1Y*((k1Y*k2Y*Ts*Ts*Ts)/6 - (k2Y*Ts*Ts)/2 +
 (k2Y*Ts*Ts*(- k1Y*k1Y*Ts*Ts + k2Y*Ts*Ts))/24));
        yYawNew = x1YNew + x2YNew*Td*0.4;

        //PID for the yaw velocity discretized
        //Coefficients for the discretization
        b0yv = Pyv*(1+Nyv*Ts) + Iyv*Ts*(1+Nyv*Ts) + Dyv*Nyv;
        b1yv = -(Pyv*(2+Nyv*Ts) + Iyv*Ts + 2*Dyv*Nyv);
        b2yv = Pyv + Dyv*Nyv;
        a0yv = (1 + Nyv*Ts);
        a1yv = -(2 + Nyv*Ts);
        a2yv = 1;

        //double yev = yawAngleError;//yaw error
        //double yev = yawAngleError + smithYawDelayed -
smithYaw; //yaw error with Smith Predictor
        double yev = yYawNew - smithYaw; //yaw error with
Smith Predictor and Kalman Filter
        double yuv = (-a1yv*yupv - a2yv*yuppv + b0yv*yev +
b1yv*yepv + b2yv*yeppv)/a0yv; //yaw velocity send to the
```

```cpp
drone
        velYawControl = yuv;
        yeppv = yepv; //Update values
        yepv = yev;
        yuppv = yupv;
        yupv = yuv;

        smithYaw = (yupv+yuv)/2*sampleTime + smithYaw; // trapezoidal integration

        //Send values to be saved in the JSON file
  controllerData.gimbal = velPitchControl;
        controllerData.velocityYaw = velYawControl;


        //Old controller//

        /*auto velocity_yaw = diff_yaw/7; ///@note "Taget ur luften... divide by 7 worked fine in Idre"
        auto velocity_gimbal = - diff_gimbal/10; ///@note " Taget ur luften"

        //controllerData.velocityYaw = 0; //@todo - Use this to have a constant yaw rotation. 0 means no rotation. 90 means rotate clockwise with velocity 90.
        controllerData.velocityYaw = velocity_yaw; //@todo - Use this in FlyToWaypoint to let the targets pixel offset control the yaw with velocity.

        if(Utils::IsBetween<double>(-2, 2)(velocity_gimbal)){

            controllerData.gimbal = 0;
        }else {

            controllerData.gimbal = velocity_gimbal;
        }*/


        mMessage.get()->setData(controllerData);
    }
    return trackingPerson;
}

void PersonTracker::drawDebugLines(const cv::_InputOutputArray &frame, double offset_x, double offset_y) const {

    cv::line(frame, cv::Point((int)offset_x, 0), cv::Point((int)offset_x, height), cv::Scalar(0, 0, 255), 2, 1);
    cv::line(frame, cv::Point(0, (int)offset_y), cv::Point(width, (int)offset_y), cv::Scalar(0, 0, 255), 2, 1);

    //debugline for image center
    auto line1_x = width / 2;
```

```
        cv::line(frame, cv::Point(line1_x, 0), cv::Point(line1_x,
     height), cv::Scalar(255, 0, 0), 2, 1);

        //debugline horizontal
        auto line1_y = height / 2;
        cv::line(frame, cv::Point(0, line1_y), cv::Point(width,
    line1_y), cv::Scalar(255, 0, 0), 2, 1);
    }
} //namespace
```

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2020

Document Number
TFRT-6105

Author(s)
Ludvig Langebro
Baldomero Puche Moreno

Supervisor
Johan Helgertz, Sony Mobile Communications, Sweden
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Rolf Johansson, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Design of control system for UAV-based video recording and tracking

Abstract

The core objective of this project was to design a controller for a drone (or quadcopter) that would allow the drone to keep a target in the center of a video filmed by a camera mounted on the drone. This was done by controlling the pitch of the gimbal, on which the camera was placed, and the yaw of the entire drone combined with using an image recognition program that could identify the target. For this project the DJI Mavic 2 Zoom was used. Although designing a PID controller was an important part of the project it was relatively easy and emphasis was instead put on system identification and minimizing the impact of a large delay. The large delay was present in sending images from the drone to the controller and could be as long as 1 second. A Smith predictor was used in order to reduce the influence of the measurement delay in the control while a Kalman filter was used in order to estimate the position of the target based on the old measurements. The system consisted of a computer connected wirelessly to a mobile phone, and the drone connected wirelessly to the remote controller which in its turn was connected by wire to the mobile phone. The end result was a system that could keep a moving target, with constant velocity, well in the center of the video but suffered from overshoots for a target making quick turns.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title	*ISBN*
0280-5316	

Language	*Number of pages*	*Recipient's notes*
English	1-63	
Security classification		