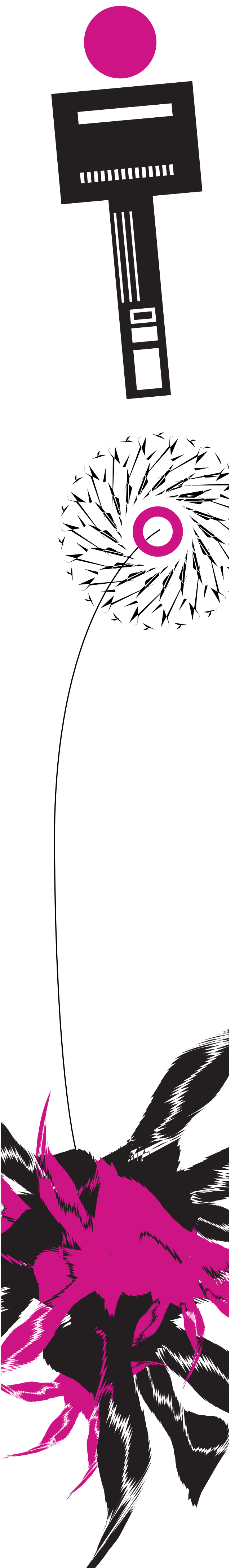


ROBOTICS
AND
MECHATRONICS

STUDY THE OUTDOOR PERFORMANCE OF AN OPEN-HARDWARE/SOURCE RACE-LEVEL QUADROTOR

R.F. (Remmelt) Fopma

MSC ASSIGNMENT

Committee:
prof. dr. ir. A.L. Varbanescu
prof. dr. ir. A. Franchi
dr. S Sun
C. Gabellieri, Ph.D

June, 2023

018RaM2023
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE
UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

1 Abstract

This thesis studies feasibility of outdoor agile autonomous flight of a quadrotor, that is equipped with an RTK-GPS, IMU and magnetometer. The novelty of this study is the autonomous flight of agile trajectories using low sample rate pose measurements. The open-source autonomous agile flight platform agilicious is used as starting point, where the bridges and state estimator are altered. An extended Kalman filter is used as state estimation algorithm and model predictive control is used as control algorithm. GPS samples at a low sample rate of 5 Hz are fused with IMU samples at a high sample rate of 400Hz in the extended Kalman filter. The feasibility is tested using a closed-loop simulation, where sensor data is generated using noise characteristics based on the sensors. The results are compared to a benchmark that uses perfect state estimation. From the results, it is found that the implementation has the same performance as the benchmark flying trajectories with accelerations of over 3g, angular rates of 700 deg/s and velocities of over 60 km/h. Furthermore, there is leeway in the maximum noise-levels of the sensor messages. These results indicate a strong possibility that this implementation will work in real-life flight.

Contents

2 Introduction

Quadrotors itself is an old research filed, with first quadrotors dating back to the sixties. In the beginning, these quadrotors had large helicopter-like sizes with combustion engines, so had a large mass. Over the years, the sizes of these quadrotors have reduced and electromotors have replaced combustion engines, reducing the weight drastically. This reduced size and weight allows the quadrotor to fly more agile, which means the quadrotor can fly with higher accelerations and traverse faster through sharper corners. This opens up a lot of new ways to use quadrotors. For example, a new field of use for quadrotors are drone racing. In this field, small quadrotors are equipped with a camera on the front and drone pilots see these images using virtual reality glasses. The pilot has to fly the drone as fast as possible through an agile parkour using a controller.

Inside the quadrotor research field, there is also another research field that treats autonomous flight of quadrotors. A general overview of an autonomous flight operation consists of two parts, a state estimator and controller. The state estimator tries to predict the current position and attitude of the quadrotor, while the controller compares this state estimate with the to-be flown trajectory and generates motor commands to follow this trajectory. The autonomous quadrotor field is already very advanced and results can be found in off-the-shelf drones, from major drone suppliers. Many new drones come with hovering or return-to-base capabilities. Furthermore, the vendor *DJI* offers an application to select waypoints and let the drone fly from point to point. These implementations fly basic trajectories, which are not agile and GPS and inertial measurement data is used for the state estimation. The *Robotics and Perception* research group of the University of Zurich have developed a project *agilicious*, which is a software and hardware platform that provides autonomous flight of agile trajectories. This platform contains multiple use cases, among others flying in a motion capture hall and using these measurements as state estimates, or flying outdoors using external vision hardware that provide a state estimate. These solutions have several drawbacks that limit the freedom of flying. The question this thesis tries to answer is if it is possible, using state-of-the-art technology, to autonomously fly agile trajectories using GPS and inertial measurement data.

The main research question of this thesis is defined as follows:

- With the current state of technology, is it possible to obtain a state estimate, using GPS and inertial measurements, that is accurate enough to perform autonomous flight of agile trajectories in outdoor environments with a quadrotor?

The following assumptions yield for this main research question:

- An agile trajectory is defined as follow: A trajectory consisting of accelerations of over 1.5g, angular rates of over360 $^{\circ}/s$ and velocities of over 10 m/s. This specification is based on literature study, which can be found in the next section. Multiple sources treated agile/aggressive autonomous flight, using different definitions. These definitions have been examined and based on the age and similarity of the source to this thesis, a definition of agile trajectories for this thesis has been formulated. Angular acceleration is not taken into account, because no sources mentioned this in their definition.
- Autonomous flight is defined as flight during which the soft- and hardware on a quadrotor performs all the control and a human pilot cannot exert any influence on the quadrotor, apart from start- and stop procedures and safety measures.
- A state estimate is accurate enough to perform autonomous flight when the position tracking root mean square error (RMSE) between a trajectory and actual position of the quadrotor is less than 0.2 meter. The controller is assumed to be state-of-the-art and not a bottleneck.

Next to the main research question, the following research questions are investigated:

- Is an extended Kalman filter sufficient for accurate state estimation for agile flight, or is a more complex (Kalman) filter necessary?
- Is it feasible to run a state estimation solution accurate enough for agile autonomous flight real-time, without timing violations, on compact hardware?

3 Literature Study

The research field of autonomous quadrotors is already very advanced, though there is still a lot of research into new topics, like more complex and agile trajectories, using different sensors for more freedom in flying and artificial intelligence. In this section, I present the current state of the autonomous quadrotors research field and present a couple of papers about the new topics of this research field.

3.1 Current state of research field

Currently, autonomous quadrotor flight is performed in various manners. Firstly, different sizes of drones of drones fly with different purposes, facing different problems. For example, some quadrotors are developed for speed and are designed with this in mind. This means that the controller should react very quick when it faces obstacles or the trajectory changes. Another large quadrotor topic is transportation. These quadrotors are equipped with much larger motors, so they can fly with heavy cargo. Flying with cargo causes the quadrotor dynamics to change drastically, so the controller needs a much more detailed physics model. In this thesis, the focus is on speed.

Autonomous quadrotor flight is implemented in various ways. In [1], a visual inertial odometry (VIO) method is used to fly agile trajectories with a small quadrotor. An extended Kalman filter is used to fuse the data of the visual sensors with the data of the inertial sensors. Furthermore, an unscented Kalman filter is used to estimate the full state of the quadrotor at 500 Hz. The calculations are performed using hardware similar to hardware in a high-end smartphone from 2018. Using this method, the states can be estimated accurately when the quadrotor if flying agile trajectories with accelerations of 1.5g, speeds up to 4.5 m/s and angular rates of 800°/s.

The paper [2] presents a full open-source software- and hardware platform, called agilicious, designed especially for autonomous agile flight. This platform uses state-of-the-art methods to perform autonomous flight. The state estimation is performed using on-board or off-board capabilities. When flying in a motion-capture hall, the state estimate of the motion-caption system can be used as state estimate. Furthermore, cameras can be attached to the quadrotor which can be used by VIO methods to obtain a state estimate. When using on-board capabilities, an extended Kalman filter is used to perform the visual and inertial sensor data fusing and state estimation. Model predictive control (MPC) is a fairly new method for control and used dynamic quadrotor and physics models to predict how the quadrotor behaves and uses this info for control commands. This is beneficiary when flying agile trajectories, because this allows the controller to respond faster. A wide variety of hardware is supported for the computations. When using the off-board capabilities in a motion-capture hall, the states can be estimated accurately when the quadrotor is flying agile trajectories with accelerations of 5g and speeds of 70 km/h. A more detailed explanation about the working of the agilicious project can be found in Section 3.3.

In the paper [3], a state estimation solution is presented that is based on GPS- and inertial sensor data. The GPS data is obtained using a RTK-GPS system, which contains of a base module that is fixed on the ground and a rover module that is attached to the quadrotor. The base- and rover module communicate with each other over a communication channel and exchange information about atmospheric environments. This system allows for centimeter-level accuracy of position measurements, which is significantly better than ordinary GPS-system, which offer meter-level accuracy. The communication channel between base and rover imposes limits on the freedom of flying. According to this paper, the communication between base- and rover module causes a delay and together with the computational delay of calculating the position, this system is not suitable for autonomous flight when it is not compensated. This paper uses an extended Kalman filter to fuse the inertial- and GPS data and make a state estimate. This extended Kalman filter is adjusted to compensate for the RTK time delay, using a fixed time delay. This is performed by saving the calculated states and when a GPS sample is arrived, correct the concerning state in the past and propagate to the present time using saved IMU samples. The computations are performed on a large computation board. The method has been tested in a real-life experiment, where a basic trajectory is flown successfully.

3.2 Expanding research field

Currently, the research field of autonomous quadrotor flight is expanding in numerous topics. In [4] and [5], multiple Kalman filters are used in serial and parallel respectively. In the serial example, 3 different Kalman filters are used to fuse different sensor data. This way, you can tune the noise coefficients more accurately, increasing state estimation accuracy. According to [4], this is beneficiary for vehicles moving with 2-3g acceleration. In the parallel example, 2 extended Kalman filters have been implemented and are used alternately based on the current situation. According to [5], this method performs more accurate state estimates during turning manoeuvres.

Furthermore, other, more complex, versions of Kalman filters are used to increase state estimate accuracy. In [6], [7] and [8], more complex Kalman filters are presented which handle non-linear systems better than an extended Kalman filter. This is especially beneficiary when flying agile flights, because the high accelerations and turns cause large non-linear effects.

Finally, the research field of artificial intelligence has entered the autonomous flight domain. In [9] and [2], neural networks are used to increase the autonomous flying performance. In [9], neural networks are trained while flying about the bias and noise of the inertial measurements. This increases the accuracy of the state estimate. In [2], neural-network based controllers are supported.

3.3 Agilicious project

An overview of the agilicious platform structure can be found in Figure 1. For this thesis, the *Pipeline* is the most interesting block. The functionality of the different blocks are straight-forward. The estimator receives a measured state of the *Real World* or simulation using the bridge module. It uses these state measurements together with some other measurements to calculate a state estimate of the drone. This state estimate is compared with the trajectory that is supposed to be flown by the sampler and the setpoint is passed to the controller. The controller sends 4 individual rotor-speed commands for the 4 motors using the bridge to the low-level controller, which drives the motor.

The agilicious project provides a couple of options for the estimator block [2]. The most important are the *ekf* and *ekf_imu*. The *ekf* estimator exists of an extended Kalman filter, where the prediction step is performed using an constant acceleration model and the update step using 6-DoF pose and IMU measurements. The *ekf_imu* differs slightly, the prediction step is performed using IMU measurements and the update step using only 6-DoF pose measurements. The sample rate of an IMU module is much higher than a GPS module, so the *ekf* estimator is not suitable, because the estimator only performs the prediction and update step when both measurements are received. The *ekf_imu* uses the IMU data in the prediction step and 6-DoF pose data in the update step, so the state estimate is predicted at the high IMU sample rate and updated at a lower GPS sample rate. The GPS is very suitable to fill the position information in this 6-DoF pose data, but not for the attitude information, so other sensors are needed for this.

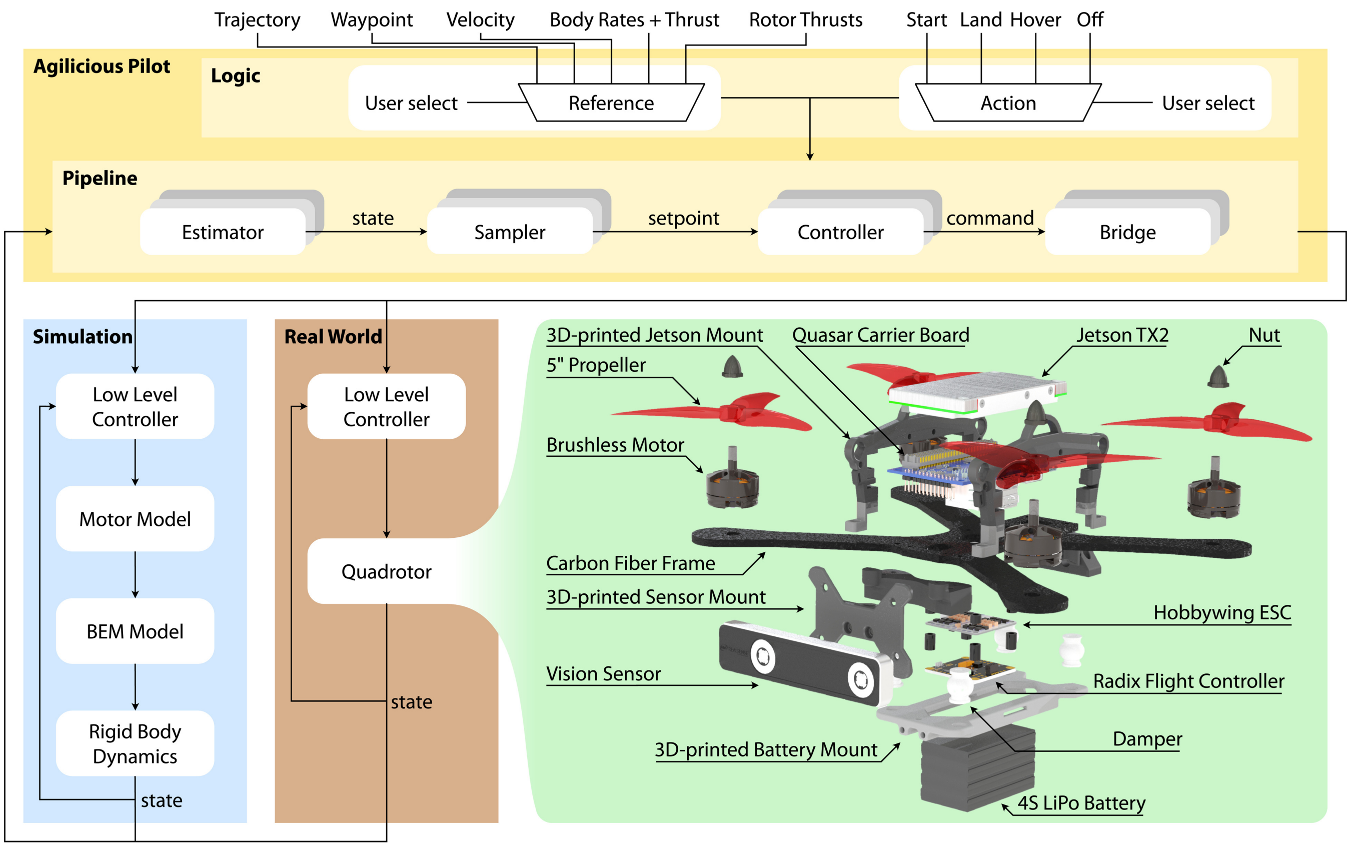


Figure 1: Diagram showing structure of agilicious project. [2]

4 Method

In this chapter, the final design is presented. The final design is based on the agilicious platform, but 2 bridges have been added and the state estimation module has been altered. A bridge for communication with the IMU was already present in the agilicious project, but bridges for communication with the GPS and magnetometer modules have been implemented myself. For the estimator module, the *ekf_imu* estimator from the agilicious project has been taken as starting point, but has been altered to work with the GPS, IMU and magnetometer sensor data and sample rates. Furthermore, support for delay compensation has been implemented, as was suggested in literature.

The final design can be subdivided in two parts, the hardware- and the software architecture, and will be treated individually.

4.1 Hardware Architecture

This section presents the final hardware architecture. First, a global overview is shown, followed by more detailed description of separate modules.

4.1.1 Overview

Figure 2 shows an abstract overview of the total hardware architecture. This overview mentions the names of the used ports and whether some conversion is used.

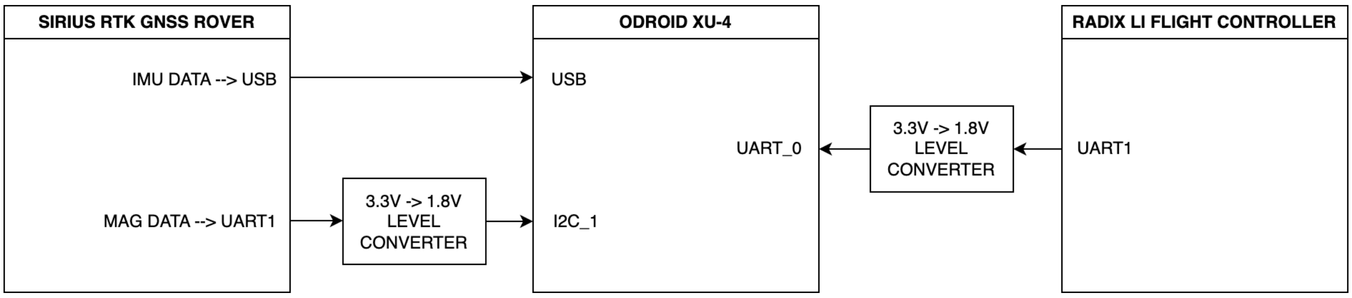


Figure 2: Overview of hardware architecture.

4.1.2 Final realization of test setup

Figure 3 shows the realized test setup that is used to test the state estimation algorithm. The connections between the components have been removed to make the overview more clear. The components are fixed on a plastic frame, consisting of custom plates and standard spacers and screws, which can be easily altered. A KISS 4-in-1 ESC is indicated in the picture, which is used to drive the motors.

4.1.3 Inertial measurement

The inertial measurement solution is similar to the Agilicious platform solution, where they use a RADIX flight controller from *brainfpv* [10], which has an inertial measurement unit (IMU) placed on it. This board is usually used as low-level flight controller, but in the agilicious project, it is also used to fetch IMU data. The agilicious project provides custom firmware that makes the chip send the IMU samples using UART communication. This firmware is based on the NuttxOS, which is a real-time embedded operating system.

In this implementation we use the RADIX LI flight controller from *brainfpv* [11], because the original version is discontinued. The firmware is altered to make it compatible with this version. The IMU sample rate is programmable in the firmware up until 3.2 kHz and in this implementation it is fixed at 400 Hz. The IMU has 6 degrees-of-freedom and consists of an accelerometer and a gyroscope. Furthermore, the firmware contains a Madgwick orientation filter, that estimates the pitch- and roll angles of the quadrotor and sends these together with the IMU data in quaternion form. In this implementation, these pitch- and roll estimations are used as part of the attitude measurement.

The most important specifications can be found in Table 1.

4.1.4 GPS

The SIRIUS RTK GNSS Rover/Base modules are used as GPS solution. This solution uses an RTK system consisting of 2 GPS modules that exchange information about the atmospheric environment with each other, to increase the position accuracy to one centimeter. An RTK system is used instead of an ordinary GPS solution, because the increased accuracy is very beneficiary to the accuracy of the state estimation. In this implementation, one GPS

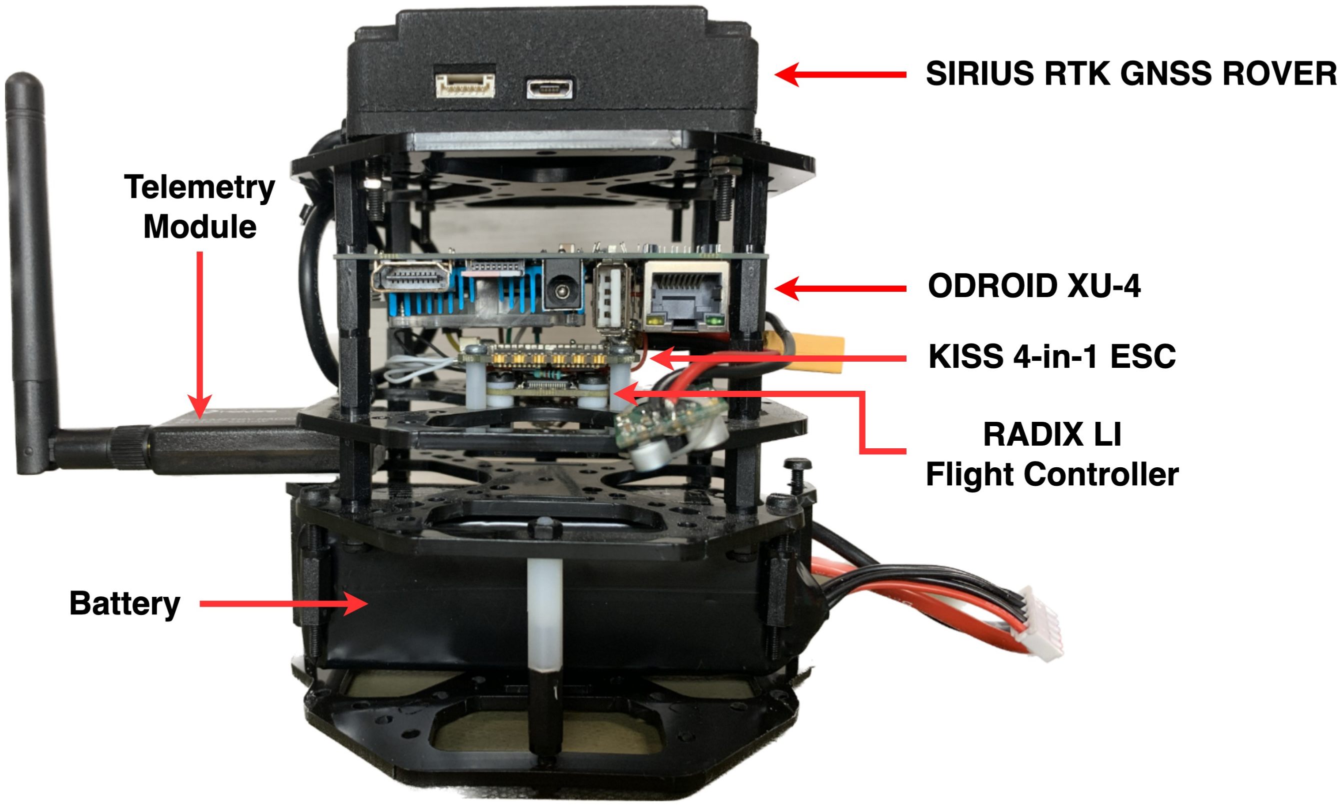


Figure 3: Overview of realized test setup.

Processor	STM32F446RET6
IMU	Bosch Sensortec BMI160
Power	3S-6S
Dimensions	26.5mm x 26.5mm

Table 1: Table containing RADIX LI flight controller specifications.

module is placed at a fixed position and acts as a base and the other module is attached to the drone and acts as a rover module. A telemetry kit is used for communication between the base and rover modules. The only downside of this implementation is that the drone should stay near the base module, in order to receive the base information. With a telemetry kit, the range is about 300 metres, but there are several solutions to increase it even further. The telemetry kit used in this implementation is the Holybro Telemetry Radio V3.

Specifications of GPS Rover/Base modules [12] [13] and telemetry kit[14] can be found in Table 2 and 3. In our implementation, the rover module is connected to the computing board by a USB connection. Over this connection, NMEA messages are send at a rate of 5 Hz. NMEA messages provide information about position, altitude, satelite connection, etc.

GPS chip	u-Blox ZED-F9P
RTK position accuracy	Down to 0.01 m
RTK update rate	Up to 20 Hz
Interface modi	U-blox or NMEA messages over I2C, UART or SPI

Table 2: Table containing SIRIUS RTK GNSS Rover/Base specifications.

4.1.5 Magnetometer

Magnetometer measurements are used to enhance the accuracy of the attitude state estimate. The magnetometer functions as a compass and the magnetometer measurements are used to determine the yaw angle of the quadrotor. The SIRIUS RTK GNSS Rover module mentioned in Section 4.1.4 contains a RM3100 magnetic sensor from PNI. This sensor is used as magnetometer in this implementation. The specifications of this module [15] can be found in Table 4. This module is configured at a sample rate of 9 Hz.

Carrier frequency	433 Hz
Maximum output power	100mW
Range	>300m

Table 3: Table containing Holybro Telemetry Radio V3 specifications.

Magnetic field measurement range	+-800µT
Maximum sample rate	533 Hz

Table 4: Table containing RM3100 magnetic sensor specifications.

4.1.6 Computing board

The ODROID XU-4 is used as computing board. The design principle is similar to the Raspberry Pi, it has significant computing power, an ARM chip capable of running Linux, lots of I/O and a small form factor. The specifications of the ODROID XU-4 board can be found in Table 5.

Hardware	
Processor	Samsung Exynos5422 ARM
Cores	Quad-core Cortex™-A15 at 2.0GHz and Quad-core Cortex™-A7 at 1.4GHz
GPU	Mali™-T628 MP6
Memory	2 GB LPDDR3 RAM
IO	12- and 30 pin headers including support for UART, SPI, I2C, I2S
Software	
OS	Ubuntu 20.04.5 LTS (Focal Fossa)
Kernel	Linux 5.4.224-246

Table 5: Table containing ODROID XU-4 computing board specifications.

4.2 Software Architecture

This section presents the final software architecture. First, a global overview is shown, followed by a more detailed description of separate modules.

4.2.1 Global overview of software architecture

Figure 4 contains a global overview of the software architecture. In this overview, 3 different channels are used to receive information from the IMU, GPS and magnetometer modules. The dataflow of all these modules are similar. The messages of the different sensor are received at a *SerialPort* object, which makes sure a complete message is received and passes that to a *Bridge* object. This *Bridge* decodes the messages, saves it and possibly, passes it to the estimator, with an *addIMU/Pose()* function call. This function adds the measurement to a queue and if the state estimator is free, executes the state estimator with a *process()* call.

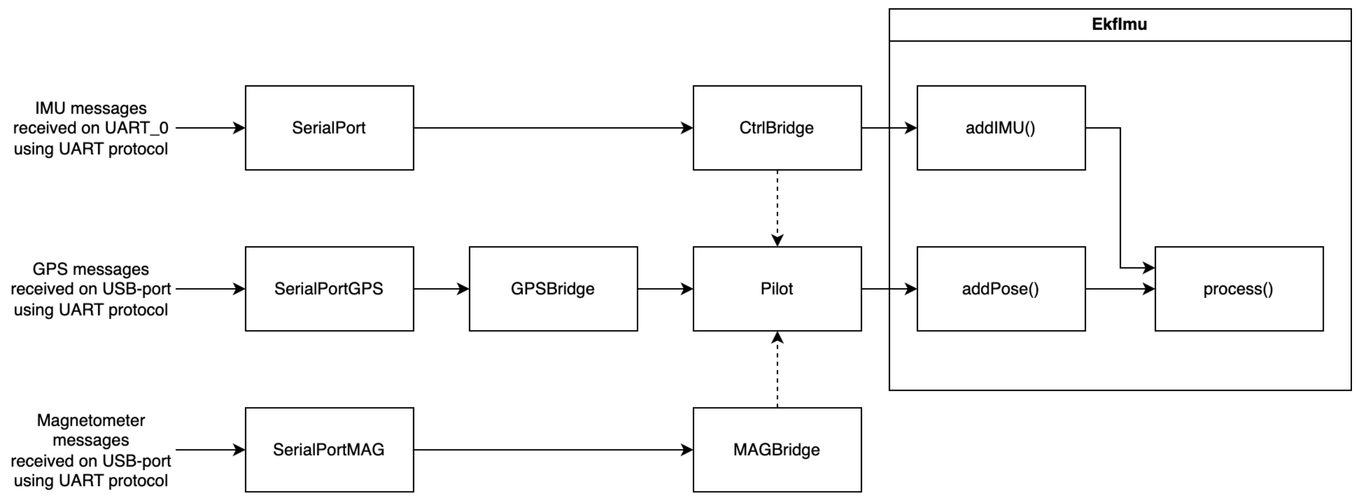

EkfImu
IMU messages received on UART_0 using UART protocol
GPS messages received on USB-port using UART protocol
Magnetometer messages received on USB-port using UART protocol
SerialPort
SerialPortGPS
SerialPortMAG
GPSBridge
CtrlBridge
Pilot
MAGBridge
addIMU()
addPose()
process()

Figure 4: Block-diagram showing global overview of software architecture.

4.2.2 Parsing of measurement data.

IMU data

The IMU and attitude data is received at a baudrate of 115200 bits/s, is not human readable and COBS encoded. After receiving and decoding the data in the CtrlBridge, the data is filled in a feedback object and callback functions are executed. One of these functions is *EkfImu::addImu()*, which starts the state estimation algorithm using the *EkfImu::process()* call. A visual overview of this procedure can be found in Figure 5.

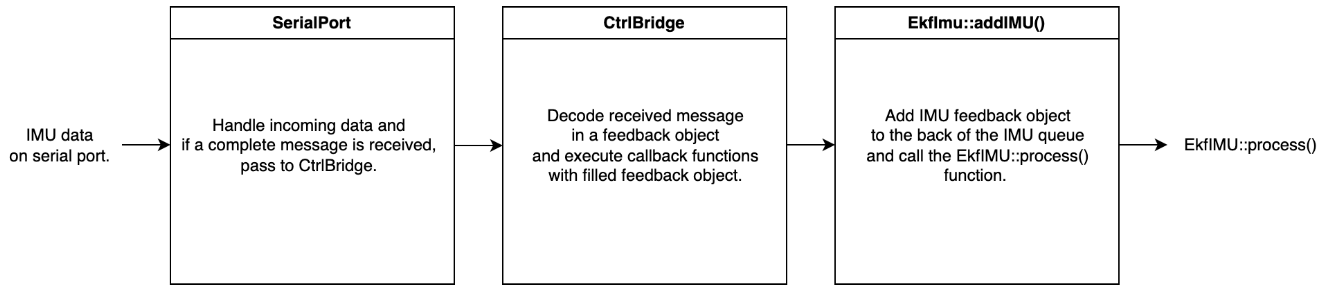

SerialPort
CtrlBridge
EkfImu::addIMU()
IMU data on serial port.
Handle incoming data and if a complete message is received, pass to CtrlBridge.
Decode received message in a feedback object and execute callback functions with filled feedback object.
Add IMU feedback object to the back of the IMU queue and call the EkfIMU::process() function.
EkfIMU::process()

Figure 5: Block-diagram showing order of parsing IMU data.

Magnetometer data

The communication with the magnetometer is based on the I2C protocol. First, the magnetometer has to be configured, by setting the appropriate registers. The SerialPortMAG object polls the magnetometer constantly if a new measurement if available, and if so, the measurement registers are read. In the MAGBridge, the magnetic field measurement is converted into a yaw angle, using simple trigonometry. First an initial angle is calculated using Equation 1. The heading is calculated by rotating this initial angle 90/180/270 degrees, depending on the sign of the magnetic field measurements. This resulting heading angle is saved and can be fetched with a function call.

$$alpha = tan^{-1}(\frac{|y\ sample|}{|x\ sample|}) \tag{1}$$

GPS data

The GPS rover module sends NMEA messages over the USB port using the UART protocol. These NMEA messages are in human readable format and send at a baudrate of 115200 bits per second. In the GPSBridge, the NMEA messages are parsed and all info is stored in GPSfeedback objects. In the *Pilot::registerGPSFeedbackCallbacks()* function, the obtained GPS coordinates are converted to a local coordinate system. The first time this function is called, the GPS coordinates are saved as reference coordinates. After this, the local coordinates are derived by calculating the distance between the received- and reference coordinates in meters. By using the difference between coordinates, radius of the earth and basic trigonometry, the distance is calculated in north and west direction. Because the earth has an ellipsoid shape, the radius of the earth differs at various places. A longitude and latitude correction algorithm is used to compensate this. [16]

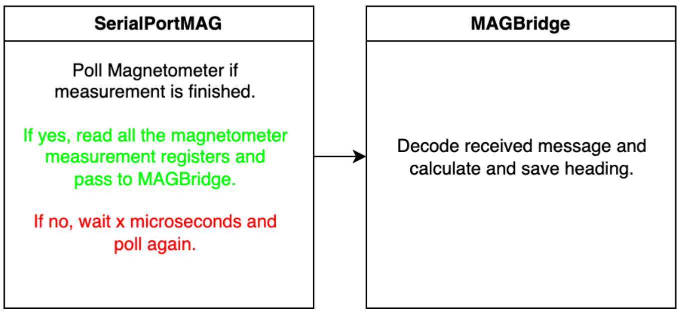

SerialPortMAG
Poll Magnetometer if measurement is finished.
If yes, read all the magnetometer measurement registers and pass to MAGBridge.
If no, wait x microseconds and poll again.
MAGBridge
Decode received message and calculate and save heading.

Figure 6: Block-diagram showing order of parsing MAG data.

Furthermore, the last received attitude from IMU and heading from the last magnetometer sample are fetched. In order to formulate a full attitude measurement, the pitch- and roll angles from the IMU and yaw angle from the magnetometer have to be combined. Because the attitude from the IMU is formulated as a quaternion, this is done using quaternion multiplication. First, a quaternion is composed of only the heading info, thus pitch and roll angles are zero. This quaternion is than multiplied with the IMU quaternion, resulting in a quaternion that represents the complete attitude.

The final position and attitude measurements are filled in a pose object and is handed to the *EkfImu::addPose()* function, which starts the state estimate update step by a *EkfImu::process()* call.

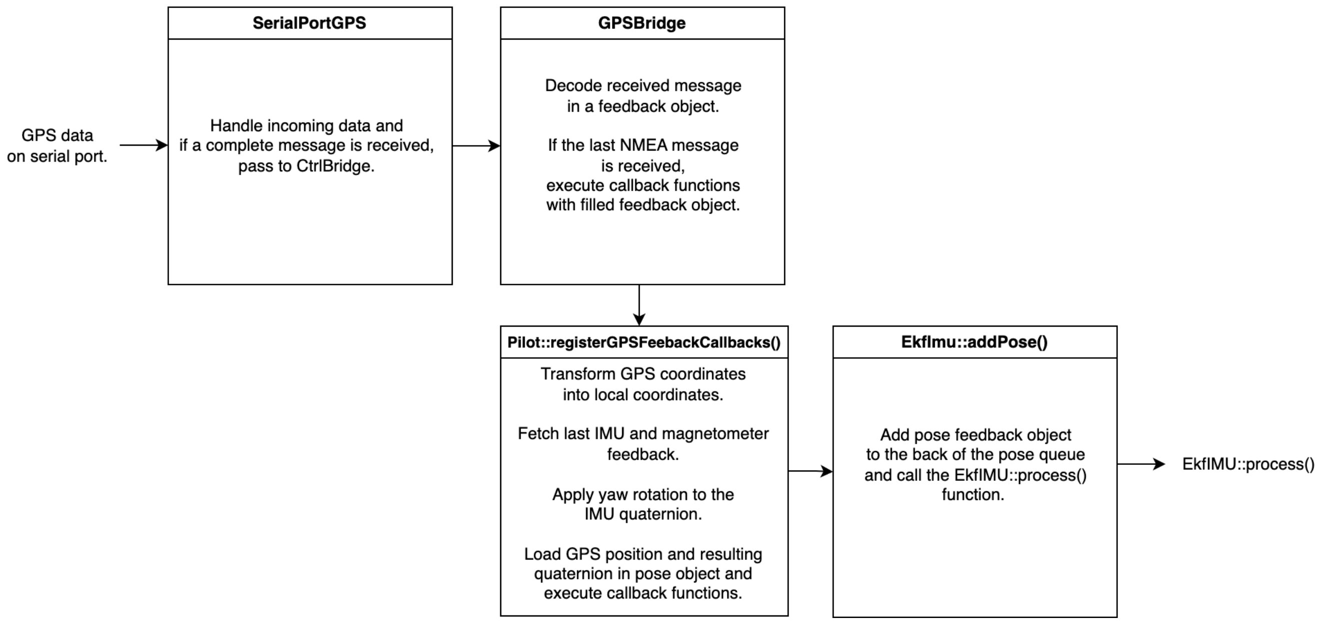

GPS data on serial port.
SerialPortGPS
Handle incoming data and if a complete message is received, pass to CtrlBridge.
GPSBridge
Decode received message in a feedback object.
If the last NMEA message is received, execute callback functions with filled feedback object.
Pilot::registerGPSFeebackCallbacks()
Transform GPS coordinates into local coordinates.
Fetch last IMU and magnetometer feedback.
Apply yaw rotation to the IMU quaternion.
Load GPS position and resulting quaternion in pose object and execute callback functions.
EkfImu::addPose()
Add pose feedback object to the back of the pose queue and call the EkfIMU::process() function.
EkfIMU::process()

Figure 7: Block-diagram showing order of parsing GPS data.

4.2.3 Implemented Kalman filter.

The final Kalman filter is based on the *ekf_imu* estimator in the agilicious platform. This estimator performs the prediction step using IMU measurements and the update step using only 6-DoF pose measurements. The implemented Kalman filter equations can be found in Table 6. A summary on the theory of Kalman filters can be found in Appendix 8.1. The states of the final Kalman filter can be found in Equation 2, where x denotes the position in meters, q the attitude in quaternion notation, v the velocity in m/s, ω the rotational frequency measurements in rad/s, a the acceleration measurements in m/s^2, ω_b the bias on rotational frequency measurements in rad/s and a_b the bias on acceleration measurements in m/s^2.

Linearization:

Linearize process model	$F_{k-1} = \frac{\partial f}{\partial \mathbf{x}}\|_{\hat{\mathbf{x}}^+_{k-1},\mathbf{u}_{k-1}}$
Linearize measurement model	$H_k = \frac{\partial h}{\partial \mathbf{x}}\|_{\hat{\mathbf{x}}^+_k}$

Prediction:	
Predicted state estimate	$\hat{\mathbf{x}}_k^- = \boldsymbol{f}(\hat{\mathbf{x}}_{k-1}^+, \mathbf{u}_{k-1})$
Predicted error covariance	$P_k^- = FP_{k-1}^+F^T + GR_{IMU}G^T + Q$
Update:	
Measurement residual	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \boldsymbol{h}(\hat{\mathbf{x}}_k^-)$
Kalman gain	$K_k = P_k^- H_k^T(R + H_k P_k^- H_k^T)^{-1}$
Updated state estimate	$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k\tilde{\mathbf{y}}_k$
Updated error covariance	$P_k^+ = (I - K_kH_k)P_k^-$

Table 6: Table containing the implemented Kalman filter equations.

$$\mathbf{x} = \begin{bmatrix} x_x \\ x_y \\ x_z \\ q_w \\ q_i \\ q_j \\ q_k \\ v_x \\ v_y \\ v_z \\ \omega\_b_x \\ \omega\_b_y \\ \omega\_b_z \\ a\_b_x \\ a\_b_y \\ a\_b_z \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ a_x \\ a_y \\ a_z \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} x_x \\ x_y \\ x_z \\ q_w \\ q_i \\ q_j \\ q_k \end{bmatrix} \tag{2}$$

Composition of F matrix.

The F matrix describes the relation of the previous state vector to the new state vector. We use the extended Kalman filter, so the F matrix is calculated by linearizing the process model. This is done using the calculations in equations 3, 4, 5 and 6.

Equation 3 specifies the relation between the derivative of the position and the velocity, which is equal. So this is equal to the identity matrix.

Equation 4 specifies the relation between the derivative of the attitude and the attitude, so the angular rate and the attitude, which is an integral and in quaternion, represented by quaternion multiplication. [17]

Equation 5 specifies the relation between the derivative of the attitude and the bias on the rotational frequency measurements. This is the negative integration of the bias on the previous attitude.

Equation 6 specifies the relation between the derivative of the velocity and the bias on the acceleration measurements. These are equal, but velocity is given in the world-frame and the acceleration measurements in the body-frame, so the rotational matrix is the relation. The relation has a negative sign, because this bias should be subtracted from the velocity after integration.

Furthermore, there is a relation of the attitude on the velocity, because the thrust of the motors is given in the body-frame, while the velocity is in the world frame.

The final F matrix, *IdtF*, is calculated by taking a 16x16 identity matrix and adding the aforementioned relations multiplied with *dt*. An instance of an *IdtF* matrix, can be found in Appendix 8.2. Note that the *IdtF* matrix is not constant and changes constantly.

$$\frac{d}{dv}\dot{p} = I \tag{3}$$

$$\frac{d}{dq}\dot{q} = 0.5 \cdot \bar{Q}(\omega) \tag{4}$$

$$\frac{d}{d\omega\_b}\dot{q} = -0.5 \cdot Q(q_{k-1}) \tag{5}$$

$$\frac{d}{da\_b}\dot{v} = -R \tag{6}$$

Composition of G matrix.

The *ekf_imu* has an additional transformation matrix G, that describes the relation between the new states and the measurements. This G matrix is calculated by linearizing the process model. This is done using the equations 7 and 8.

Equation 7 specifies the relation between the derivative of the attitude and the rotational frequency measurement. This represents an integration of the previous attitude with the received rotational frequency measurement.

Equation 8 specifies the relation between the derivative of the velocity and the acceleration, which are the same. The acceleration is given in the body-frame and the velocity in the world-frame, so the rotational matrix is the relation.

The final G matrix, *dtG*, is calculated by taking the aforementioned relations multiplied with *dt*. Thus in this final G matrix, the new states are integrations on the aforementioned relationships. An instance of a *dtG* matrix, can be found in Appendix 8.3. Note that the *dtG* matrix is not constant and changes constantly.

$$\frac{d}{dw}\dot{q} = 0.5 * Q(q_{k-1}) \tag{7}$$

$$\frac{d}{da}\dot{v} = R \tag{8}$$

Prediction step

The position state is predicted by integrating it with the previous velocity state with only half the timestep. Next, the velocity state vector is updated by integrating it with the acceleration measurement minus bias. Finally, the position is predicted again by integrating it with the new velocity state with again half the timestep. This way, the position state is is predicted by integrating with the mean velocity in the past timestep.

The attitude state is predicted by integrating it with the angular rate measurement minus the bias.

Composition of H matrix.

The H matrix describes the relation between the measurement and the new state vector. The measurement model is linear, so their is no linearization needed in this part.

The relation between measured position and new position state is the identity matrix, because they are equal and measured in the same frame.

The relation between measured attitude and new attitude state is also the identity matrix, because they are equal and measured in the same frame.

Update step

The update step is performed as indicated in Table 6

Process- and measurement noise coefficients.

The used noise coefficients will be specified in the Results section at each experiment, as different experiments required different tuning.

4.2.4 Delay compensation implementation.

Figure 8 contains an overview of the implemented Kalman filter using delay compensation. Whenever a *process()* call is received, the state estimation algorithm is executed. First, the IMU sample queue is fetched and all the entries are processed, from the oldest to newest sample. This causes the state estimate to propagate until the time stamp of the most recent IMU sample. For the delay compensation, all the IMU samples are saved even if they have been processed. Next, the pose sample queue is fetched and first it is checked whether it is empty or not. When the pose queue is empty, the *process()* function is exited. When it is not empty, the pose samples are processed, from oldest to newest sample. If a pose sample is processed, the delay compensation algorithm is used. Instead of the current state, the state at time $t_{rec} - t_{del}$, time when the pose sample is received minus the delay time, is updated. The saved IMU samples are used to propagate the state estimate to the current time. When another pose sample is present in the queue, this is processed at the same manner, otherwise the *process()* function is exited. When a pose sample has been processed, this pose sample and IMU samples with timestep lower than $t_{rec} - t_{del}$ are deleted from their queues. The delay time, t_{del}, is not known and should be tuned during real-life testing. As an initial value, 0.4 seconds is used as delay time, because it is used in [3]. The state estimator acts as an ordinary extended Kalman filter when t_{del} is set to zero.

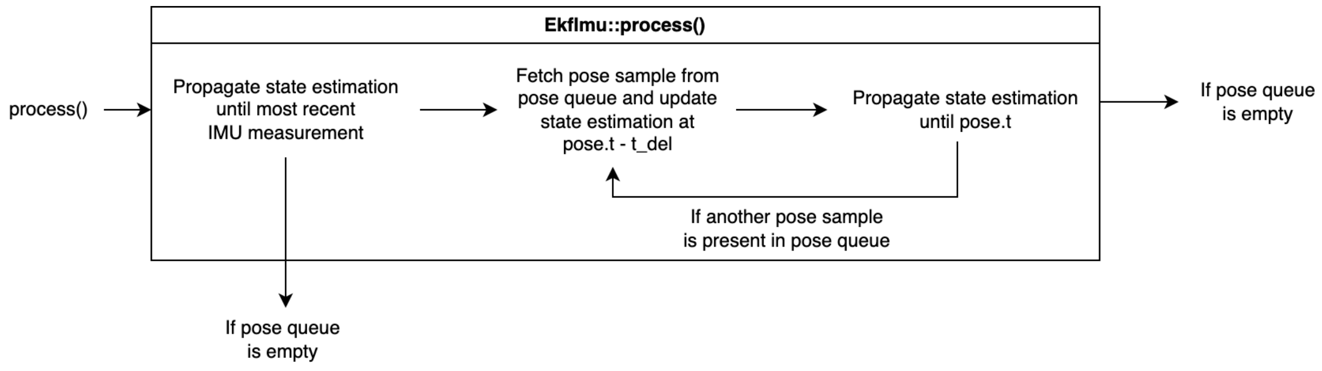

EkfImu::process()
process()
Propagate state estimation until most recent IMU measurement
Fetch pose sample from pose queue and update state estimation at pose.t - t_del
Propagate state estimation until pose.t
If pose queue is empty
If another pose sample is present in pose queue
If pose queue is empty

Figure 8: Block-diagram showing structure of implemented Kalman filter.

4.2.5 Threading of implementation.

Figure 9 contains an overview of the threading structure of this implementation, focused on the bridges and estimator. The main functionality is performed by Thread #4, where the *Pilot::pipelineThread()* function is executed. This function periodically runs the entire pipeline, which consists of fetching the most recent state estimate, get setpoints from the sampler, run the controller and send motor commands to the low-level controller. In Figure 9, the fetching of the most recent state estimate is visible, by performing a *getAt()* function call of the estimator. This function activates the state estimator and returns the most recent state estimate.

Because the *Pilot::pipelineThread()* function is quite extensive, the IMU and pose queues in the estimator become quite big in between *getAt()* function calls. This causes the calculation time of estimator to increase. In order to reduce this, parallel computing is used by adding 3 threads that update the state estimate in parallel with the pipeline thread. The 3 extra threads run the *SerialPort(xxx)::serialThread()* function of the 3 different *SerialPort* objects. This function periodically checks if data has been received from the concerned sensor at a time period of some microseconds. It is important that each *SerialPort(xxx)* object is running in its own thread, because additional delays in measurements can cause inaccuracies in the state estimate and causes the quadrotor to crash. When the *SerialPort(xxx)* objects receive a full message, the message is passed through and the measurement is added to the IMU or pose queues and the state estimator is activated. The estimator consists of a Mutex object that is used to avoid deadlock and shared resource problems. Whenever a *addIMU()*, *addPose()* or *getAt()* function call is performed by a thread, the Mutex object is blocked, causing safe access to memory. When this thread is done, the Mutex is released again. While the Mutex object is blocked, other threads that want to access to the estimator are blocked and have to wait until the Mutex object is released.

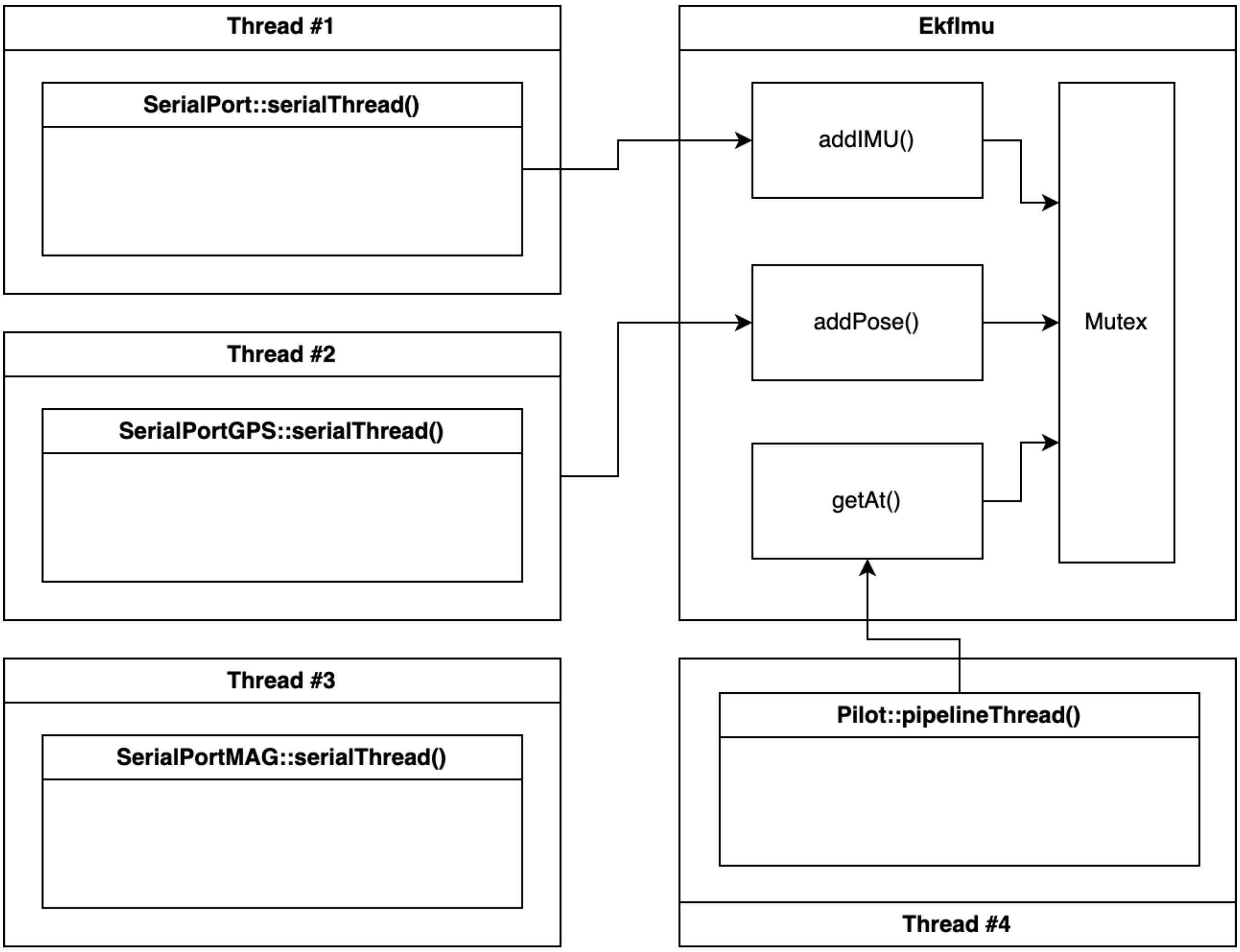


Figure 9: Block-diagram showing structure of threading of implementation.

5 Results

The implemented extended Kalman filter has to be tested, such that the research questions can be answered. In order to answer if the resulting state estimate is accurate enough for autonomous flight, the main research question is subdivided in different tests, whose results together can answer the research question. These different tests are as follows and explained in the following subsections.

- Validate the hardware architecture and extended Kalman filter using captured measurement data.
- Validate the functionality of the extended Kalman filter using simulation data.
- Test accuracy for autonomous flight in closed loop simulation.
- Test maximum noise level of measurements for autonomous flight.

5.1 Evaluation criteria

In order to evaluate the performance of the state estimation, you need some criteria that measures this. The Root Mean Square Error (RMSE) is a criteria that says how accurate something is tracked. The tracking error is the RMSE between the trajectory and position/attitude of the quadrotor and describes how accurate the trajectory is tracked. The estimation error is the RMSE between the ground truth and state estimate of the quadrotor and describes how accurate the ground truth is tracked. The ground truth is the actual position and attitude of the quadrotor. The RMSE algorithm can be found in Equation 9, where x_i is the to-be tracked variable, $\hat{x}_i$ the tracking variable and n the total amount of samples.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_n)^2} \tag{9}$$

5.2 Quadrotor terminology

Figure 10 shows the definition of the yaw, pitch and roll axis. Figure 11 shows the used axis-system in this thesis.

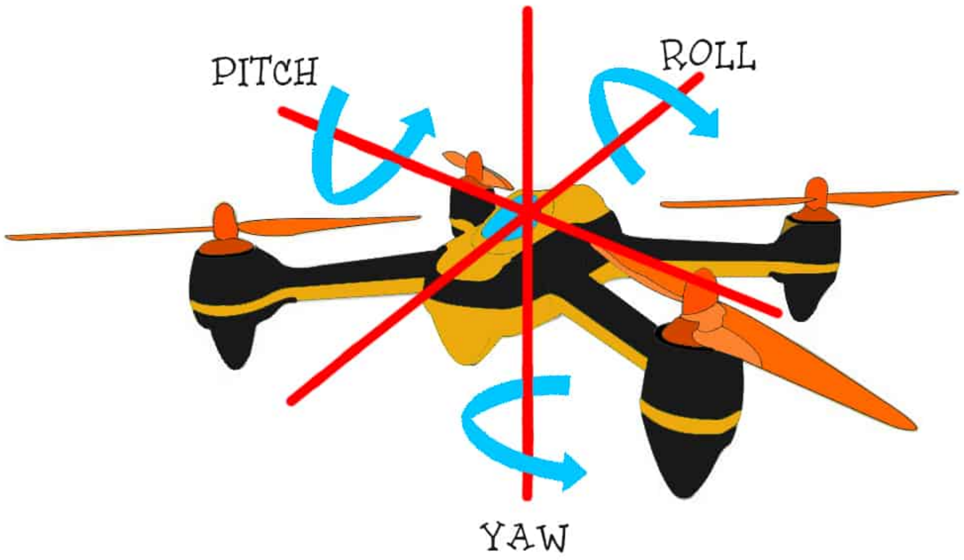


Figure 10: Figure showing definition of yaw, pitch and roll angles.

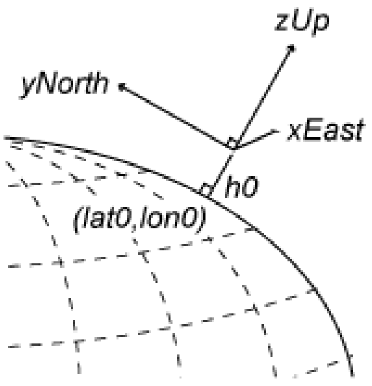


Figure 11: Figure showing definition of used axis-system.

5.3 Validate the hardware architecture and extended Kalman filter using captured measurement data

5.3.1 Method

In this step, measurements of the sensors are captured while traversing certain trajectories and later used to validate the Kalman filter. On the hardware side, the test setup is used that is shown in Section 4.1.2. On the software side, two special versions of the agilicious project are made. One version for capturing of sensor measurements, which only contains the serial ports of the original implementation and some additional functionality for saving these measurements to file. The other version is used for reading the measurement data from file, so the serial ports of this version is altered.

The measurement data is captured while traversing several trajectories, testing different aspects of the estimator. The different trajectories are as follows:

Trajectory 1: Square trajectory

This trajectory tests the position and yaw angle estimates. In this trajectory, the test setup is traversed over a square trajectory of 10x10. The test setup is always facing the direction of traversing, so yaw angle rotates 360 degrees while traversing the trajectory. The pitch and roll angles are not varied. The traversing of the trajectory is filmed, so a ground truth can be established.

Trajectory 2: Pitch and roll rotations

This trajectory tests the pitch and roll angle estimates. In this trajectory, the setup is fixed at a certain position and rotated 180 degrees clock-wise and counter-clock-wise around the pitch- and roll axis, with increasing angular rate. This increasing angular rate can be used to test how the state estimator react to agile flight. The pitch- and roll axis are tested separately, so only one axis is treated per test. These experiments are filmed, so a ground truth can be established.

Trajectory 3: Height variation

This trajectory tests the position estimate. In this trajectory, the test setup is traversed over a single line of 10 meters walking north and after passing the 5 meter point, the height of the test setup is varied. This test is filmed, so a ground truth can be established.

After capturing the sensor measurement data of the above trajectories, the state estimator is executed with the captured sensor measurements and the resulting state estimates are saved. A ground truth is established from the recording of each trajectory. The state estimate and the ground truth are compared and the estimation error is calculated. This process is repeated multiple times, while tuning the noise coefficients. The noise coefficients are tuned such that the overall estimation error of the 3 trajectories is as low as possible.

5.3.2 Results

The used Kalman coefficients for all 3 trajectories can be found in Table 7. Unfortunately, Trajectory 3 has not been tested, due to problems in Trajectory 2. Furthermore, the magnetometer failed and could not be used in the tests. This is further explained in the Discussion. Instead of the magnetometer, the Course over Ground data from the GPS module is used to determine the yaw-angle. The Course over Ground indicates in which direction the quadrotor is moving instead of pointing. When the quadrotor is moving with the front facing forward, the Course over Ground is the same as the yaw angle.

R_pos	1.0e-5
R_att	1.0e-5
R_acc	1.0e+2
R_omega	1.0e-4
Q_pos	1.0e-5
Q_att	1.0e+1
Q_vel	1.0e+1
Q_bome	1.0e-1
Q_bacc	1.0e-1

Table 7: Used Kalman coefficients for validating Kalman filter using captured measurement data.

Trajectory 1

Figure 12, 13 and 14 show measurements and state estimates of the validation of the Kalman filter using captured measurement data of Trajectory 1. Unfortunately, a ground truth could not be established, so the measured position is used in comparison with the position estimate. Another important note, is that the magnetometer failed and could not be used. Instead, the Course over Ground data from the GPS module is used to calculate the yaw-angle. In Figure 12 and 13, the trajectory is clearly visible. In Figure 12, a lot of altitude noise is visible in the estimate, but also in the measurement. In Figure 14, it can be seen that the attitude estimate is correct. In the beginning, the Y position is increased, thus the quadrotor is moving north and yaw angle should be zero, which is the case. Next, the X-position is increasing, thus the quadrotor is moving east, so yaw angle should be 90 degrees, which is the case. The results moving south and east are also valid.

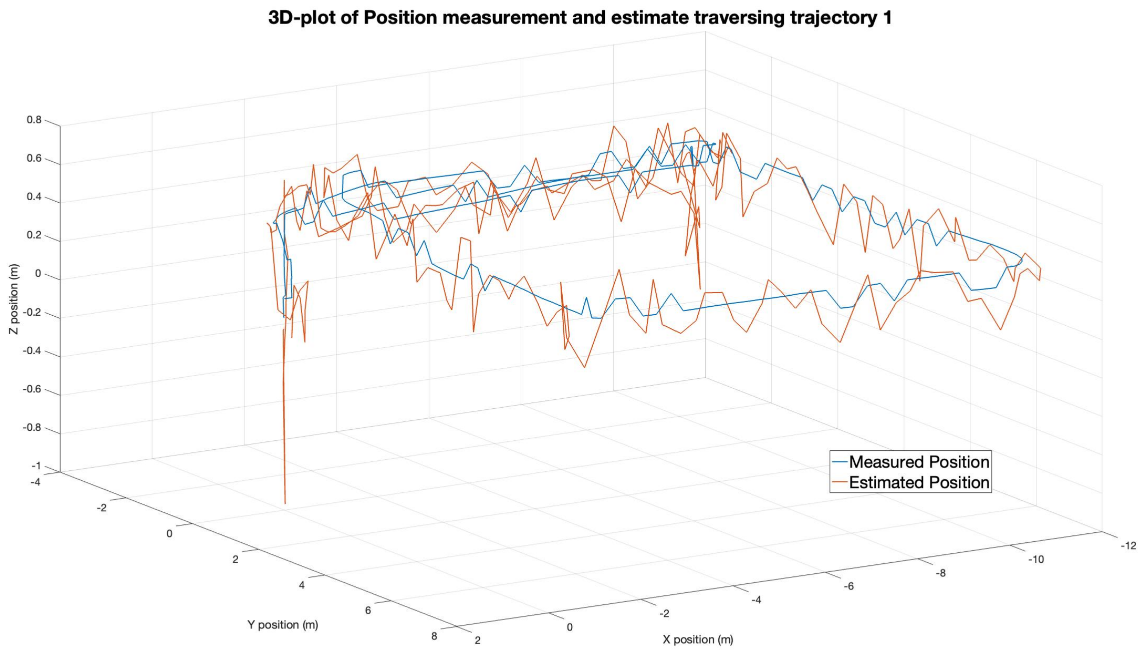


Figure 12: 3D-plot showing the result of position state estimation versus measurement.

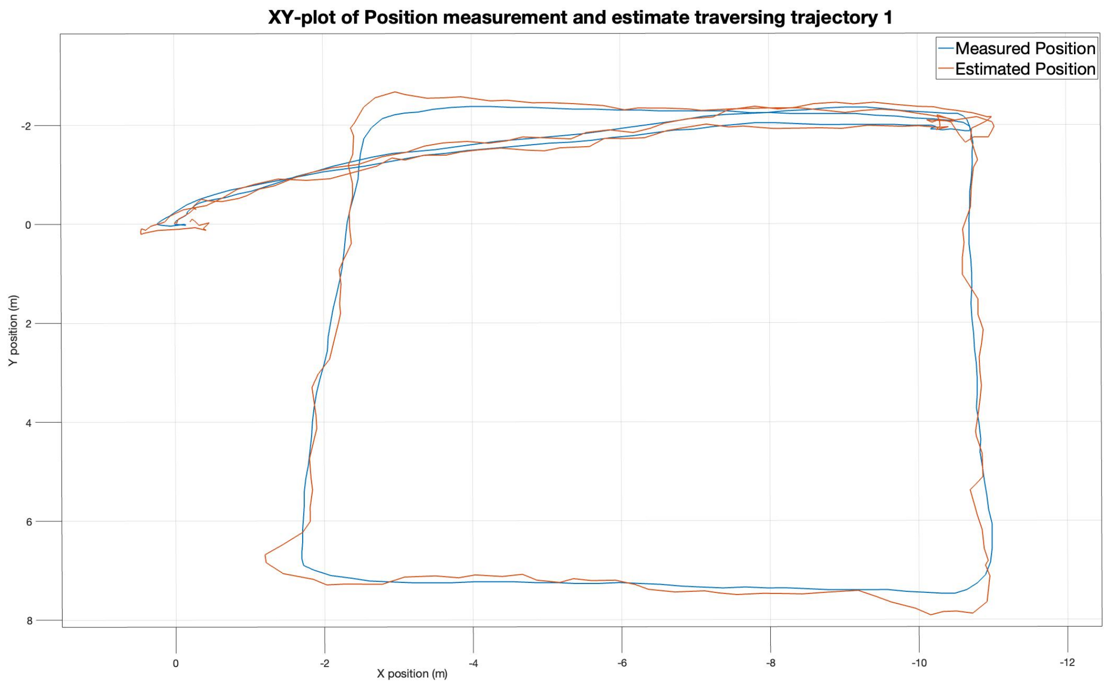


Figure 13: XY-plot showing the result of position state estimation versus measurement.

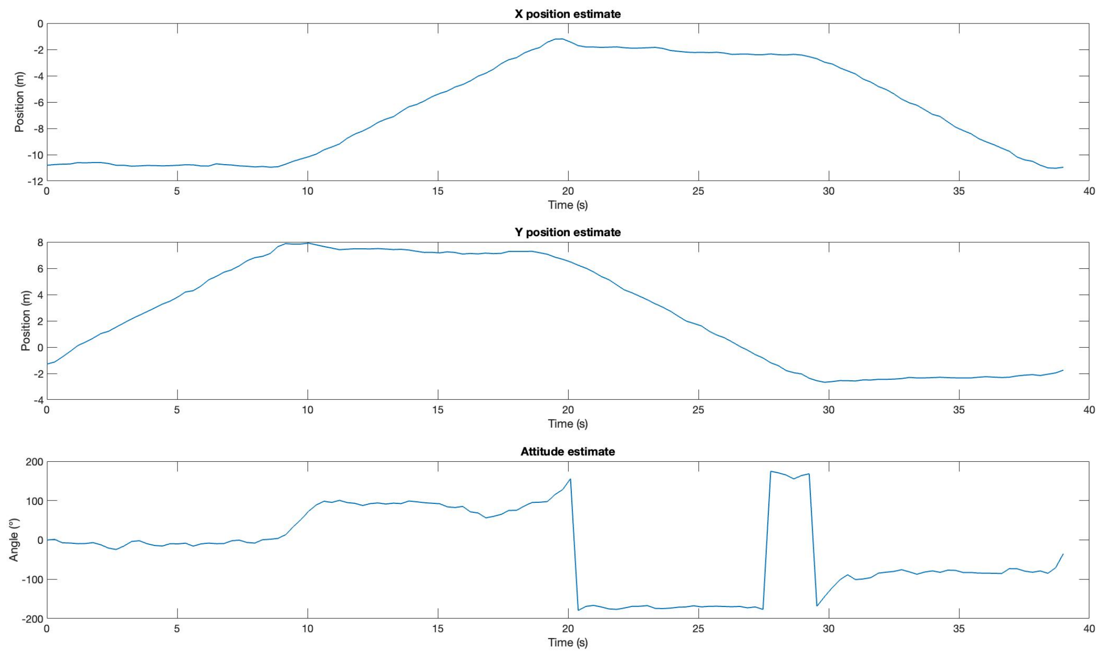


Figure 14: X- and Y axis position and attitude estimation results.

Trajectory 2

Figure 15 contains the result of the attitude state estimation of Trajectory 2. In this case, the roll angle is rotated. In Figure 15, it is visible that the quadrotor indeed is rotated around the roll-axis, but yaw- and pitch angles also vary, which is invalid. Further analysis can be found in the Discussion.

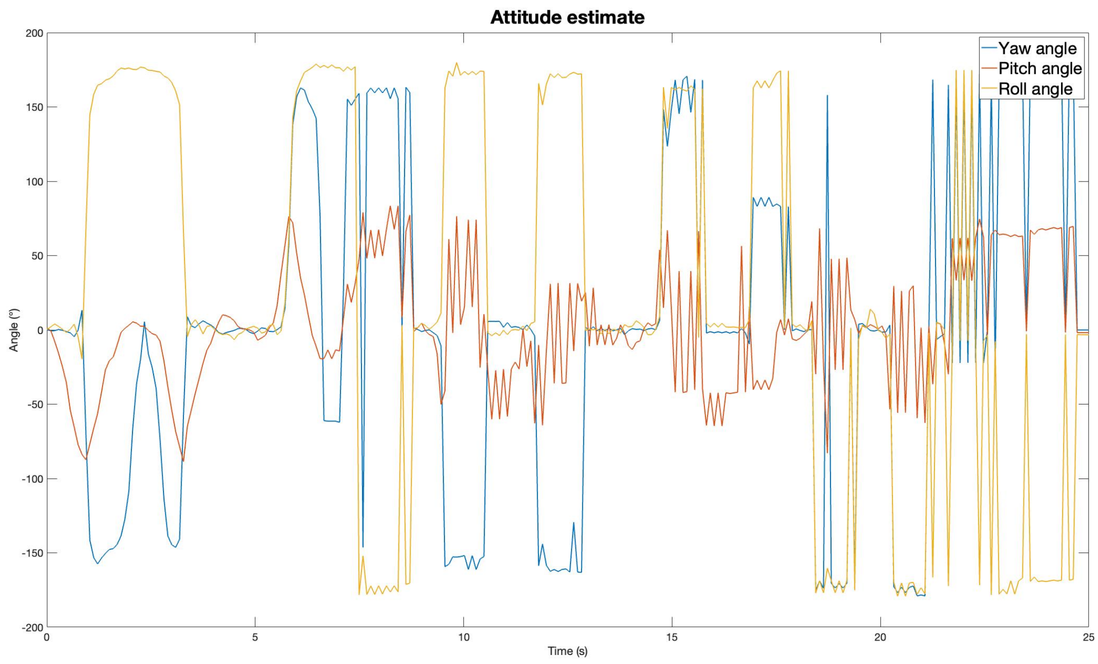


Figure 15: Plot showing the result of attitude state estimation.

5.4 Validate the functionality of the extended Kalman filter using simulation data.

5.4.1 Method

For validation, a detailed quadrotor simulation is used to generate GPS, IMU and magnetometer data. In this simulation, a dynamic quadrotor model is used to let a quadrotor fly a semi-agile trajectory. In this dynamic model, if the quadrotor turns, it actually has to change the attitude to create a force in the wanted direction. Position, attitude and magnetometer data is saved at an interval of 1 *ms*. The position is saved in meters distance with respect to the starting point, attitude in Euler angles and magnetometer measurement as magnetic field strength between 1 and -1.

A script has been made to add zero-mean Gaussian noise to these measurements, in order to simulate the real-life system. Using the data-sheets of the different sensors in the hardware architecture and experience, estimations have been made about the standard deviation and sample rate of these measurements. The noise coefficients and sample rates in Table 5.4.1 and 5.4.1 have been used to generate the noisy sensor data.

Data	**Standard Deviation**
Position	0.01 m
Acceleration	0.1 m/s
Angular Rate	1 deg/s
Attitude	0.5 deg
Magnetic Field	0.05

Table 8: Table containing estimated noise coefficients of sensors.

Message Type	Sample Time (ms)
IMU	3
GPS	200
Magnetometer	440

Table 9: Table containing sample rates of sensor messages.

The resulting noisy sensor data is bundled in GPS, IMU or magnetometer messages and saved to file. In the agilicious platform, the serial ports have been altered to read from file instead from the actual sensors. Some further alterations have been made to make up for the different compositions than the real-life sensor messages.

The implementation is executed and the resulting state estimates are written to file. The state estimates are compared to the ground truth and the estimation error is calculated. This process is repeated multiple times to tune the noise coefficients such that the estimation error becomes as low as possible.

This process is done with and without delayed data. In the test without delayed data, the *t_del* is also set to zero in the implementation. In the test with delayed data, during the bundling of GPS, IMU and magnetometer messages, the position, attitude and magnetometer samples are taken with a delay of 0.4 seconds, so at time x the sample of time $x - 0.4$ are written to file. The *t_del* is set to 0.4 seconds in the implementation.

5.4.2 Results

Without delay.
The tuned Kalman filter coefficients can be found in Table 5.4.2. The state estimation results can be found in Figures 16, 17, 18, 19, 20 and 21. The resulting estimation error can be found in Table 5.4.2. Analysis of the results can be found in the Discussion.

R_pos	1.0e-5
R_att	7.6e-5
R_acc	1.0e+2
R_omega	1.0e-5
Q_pos	1.0e-5
Q_att	1.0e-1
Q_vel	1.0e+2
Q_bome	3.0e-4
Q_bacc	1.0e-1

Table 10: Tuned Kalman coefficients for validating Kalman filter using simulation data.

Total RMSE	0.02
Total attitude RMSE	1.81 °
Yaw angle RMSE	3.09 °
Pitch angle RMSE	0.49 °
Roll angle RMSE	0.44 °
Total position RMSE	1.45 cm
X-axis RMSE	1.61 cm
Y-axis RMSE	1.38 cm
Z-axis RMSE	1.33 cm

Table 11: Table containing estimation error using simulation data.

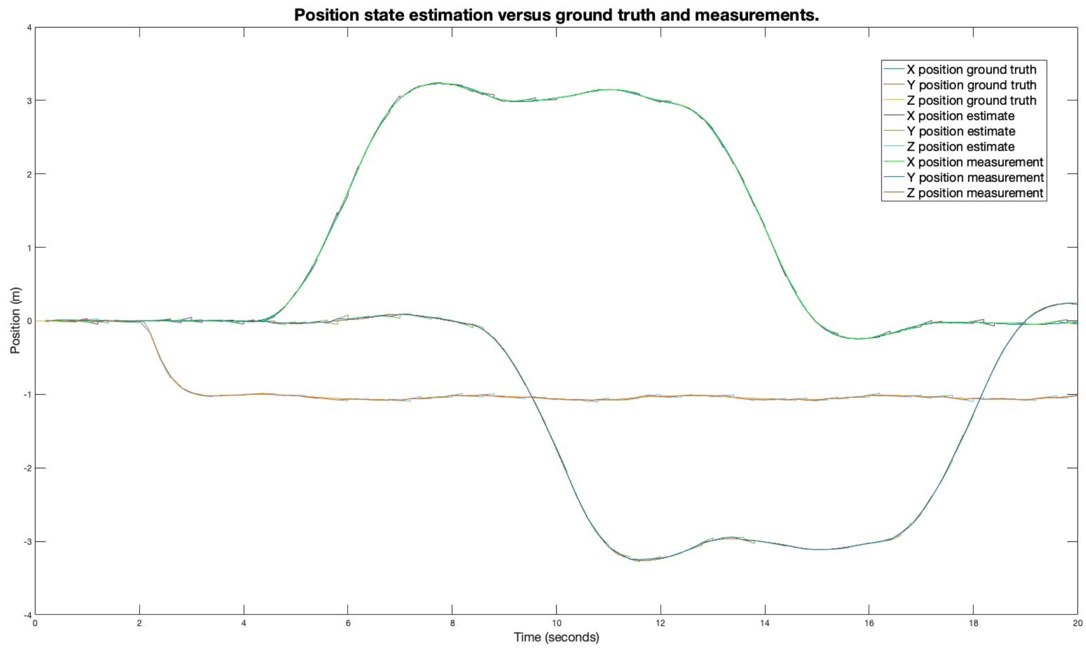


Figure 16: Plot showing the result of the position state estimation versus ground truth and measurement.

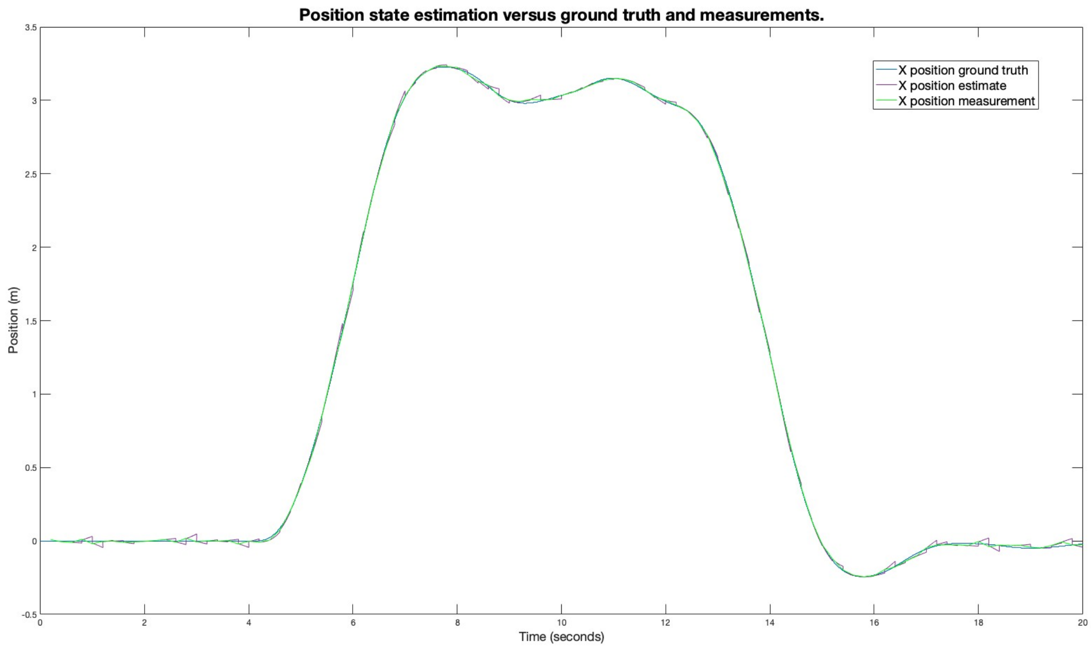


Figure 17: Plot showing the result of the x-axis position estimation versus ground truth and measurement.

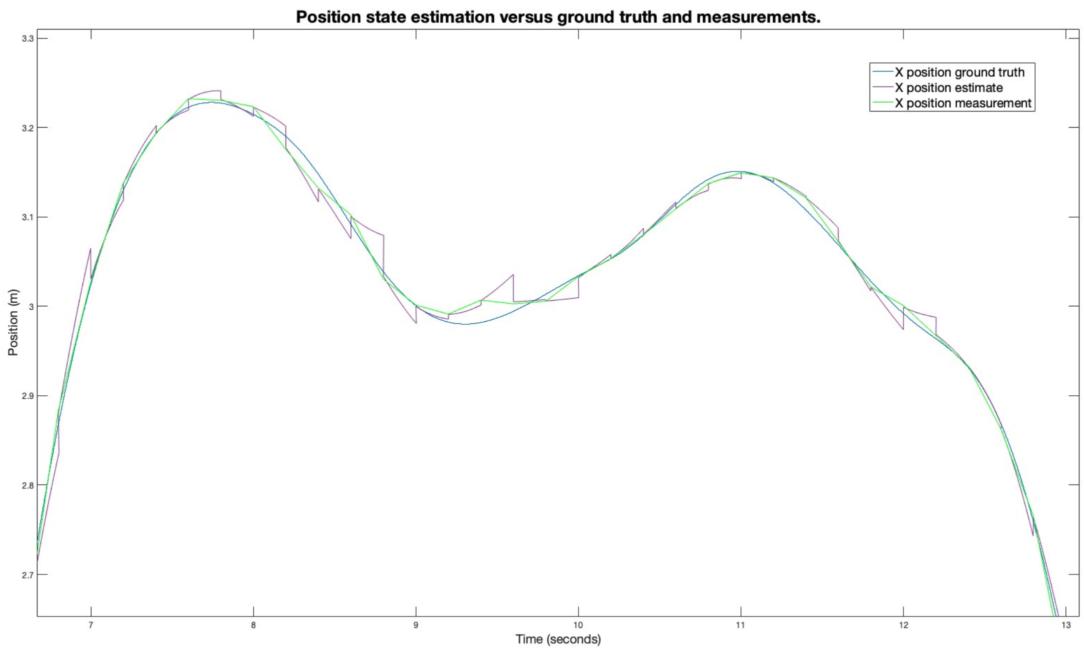


Figure 18: Zoomed in plot showing the result of the x-axis position estimation versus ground truth and measurement.

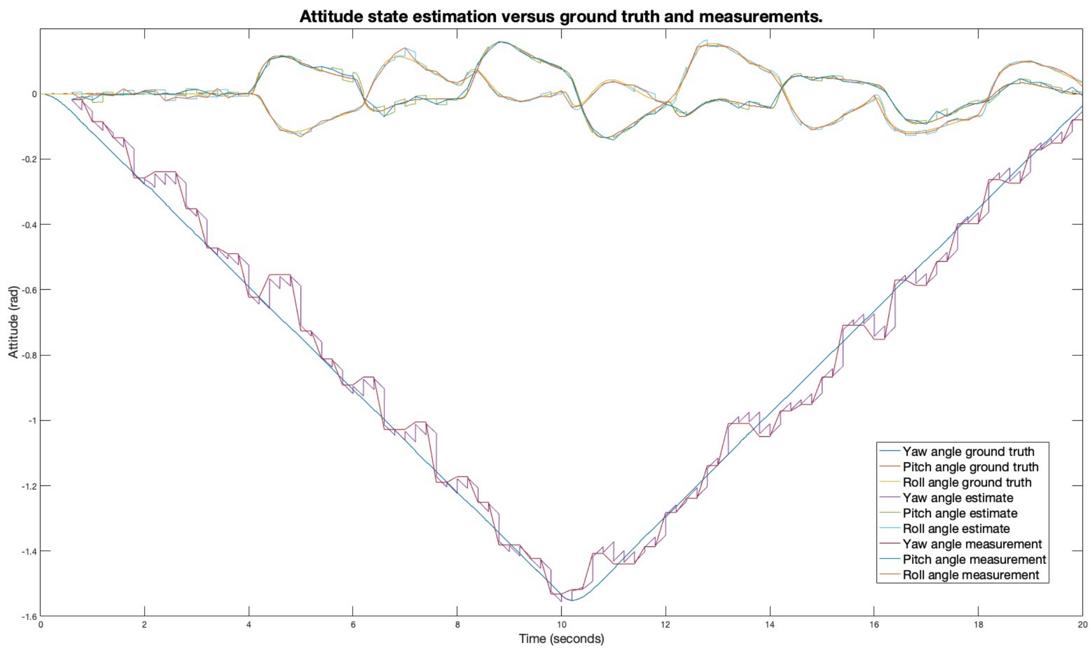


Figure 19: Plot showing the result of the attitude state estimation versus ground truth and measurement.

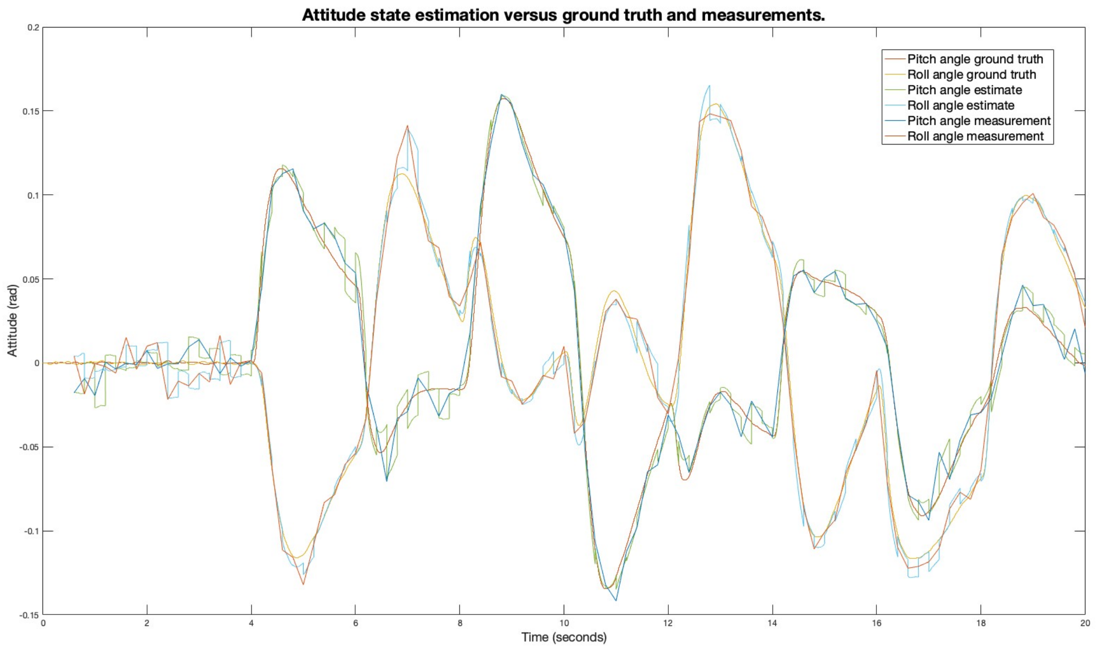


Figure 20: Plot showing the result of the pitch and roll angle estimation versus ground truth and measurement.

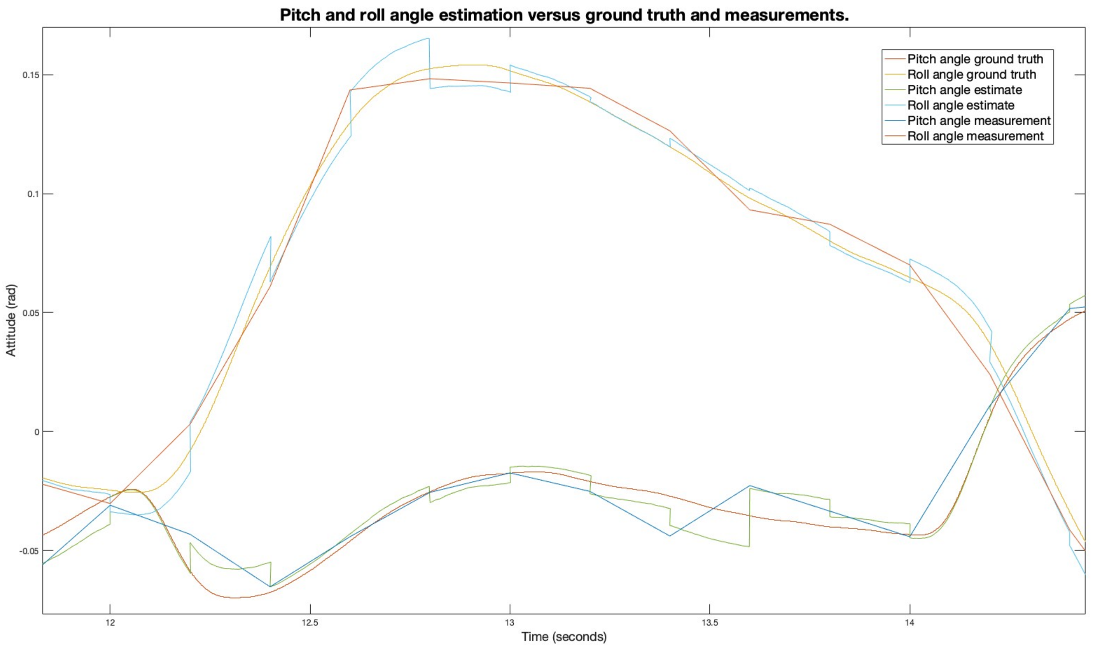


Figure 21: Zoomed in plot showing the result of the pitch and roll angle estimation versus ground truth and measurement.

With delay and delay compensation.
The tuned Kalman filter coefficients can be found in Table 5.4.2. The state estimation results can be found in Figures 22, 23, 24, 25, 26 and 27. The resulting estimation error can be found in Table 5.4.2. Analysis of the results can be found in the Discussion.

R_pos	1.0e-5
R_att	7.6e-5
R_acc	1.0e+2
R_omega	1.0e-5
Q_pos	1.0e-5
Q_att	1.0e-1
Q_vel	1.0e+3
Q_bome	3.0e-4
Q_bacc	1.0e-1

Table 12: Tuned Kalman coefficients for validating Kalman filter using simulation data and delay compensation.

Total MSE	0.04
Total attitude MSE	1.95 °
Yaw angle MSE	3.32 °
Pitch angle MSE	0.52 °
Roll angle MSE	0.49 °
Total position MSE	5.78 cm
X-axis MSE	5.78 cm
Y-axis MSE	6.30 cm
Z-axis MSE	5.43 cm

Table 13: Table containing estimation error using simulation data with delay compensation.

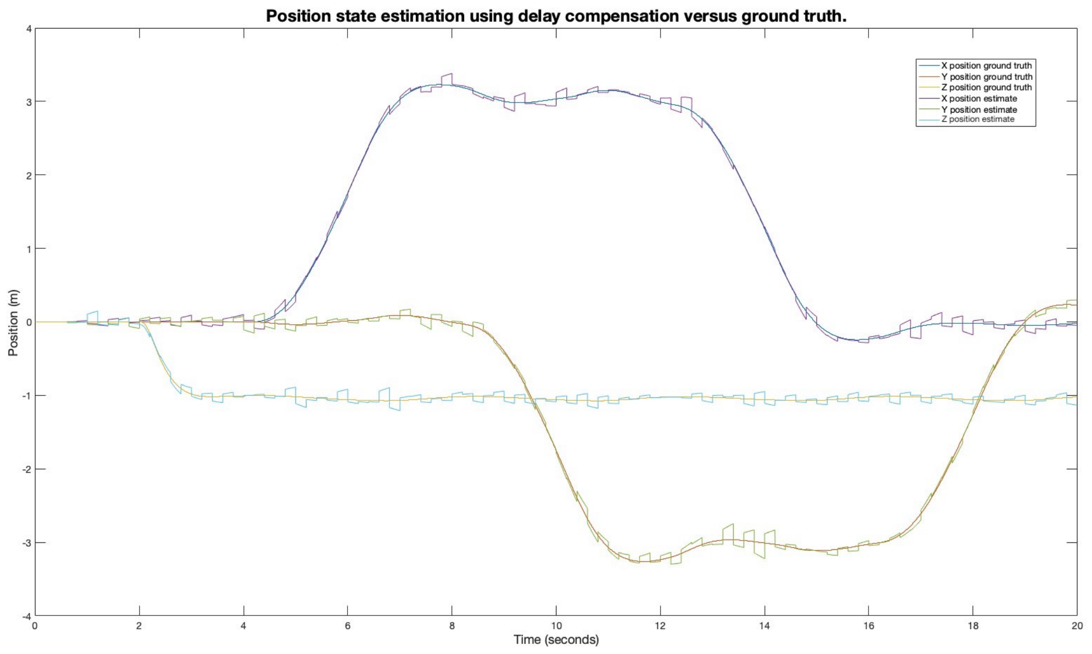


Figure 22: Plot showing the result of the position state estimation using delay compensation versus ground truth.

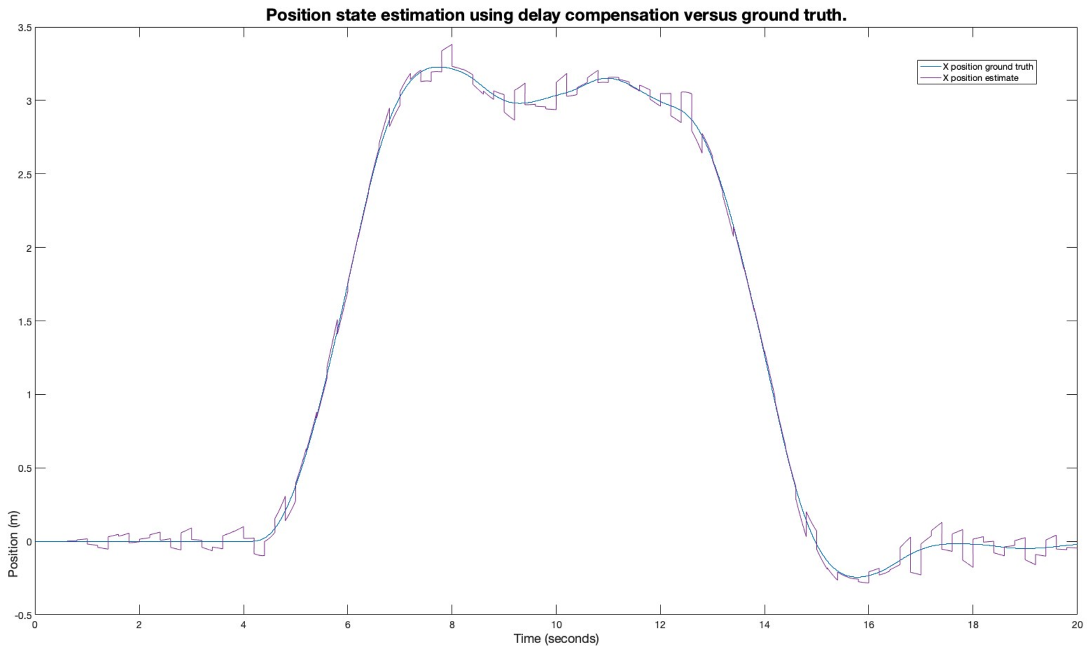


Figure 23: Plot showing the result of the x-axis position estimation with delay compensation versus ground truth.

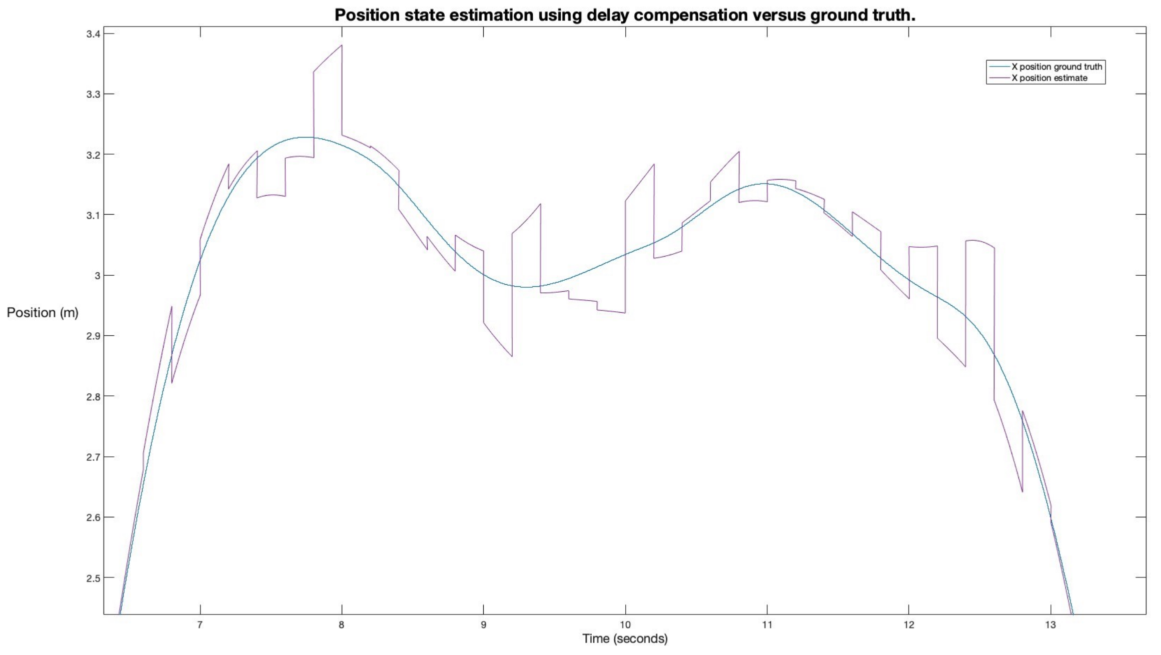


Figure 24: Zoomed in plot showing the result of the x-axis position estimation with delay estimation versus ground truth.

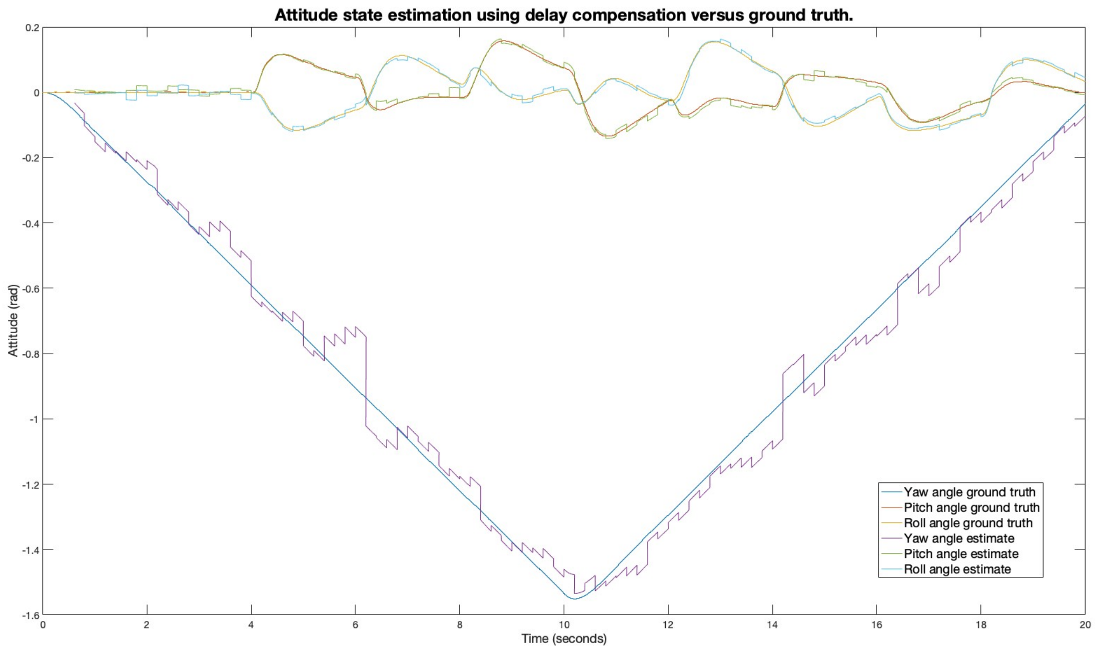


Figure 25: Plot showing the result of the attitude state estimation with delay compensation versus ground truth.

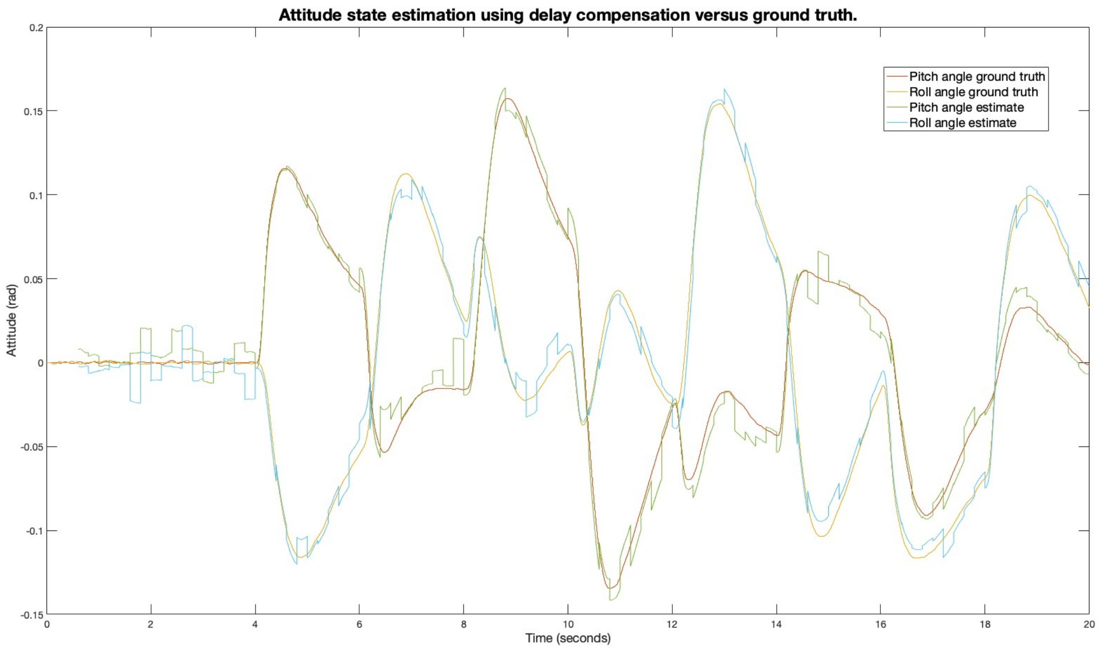


Figure 26: Plot showing the result of the pitch and roll angle estimation with delay compensation versus ground truth.

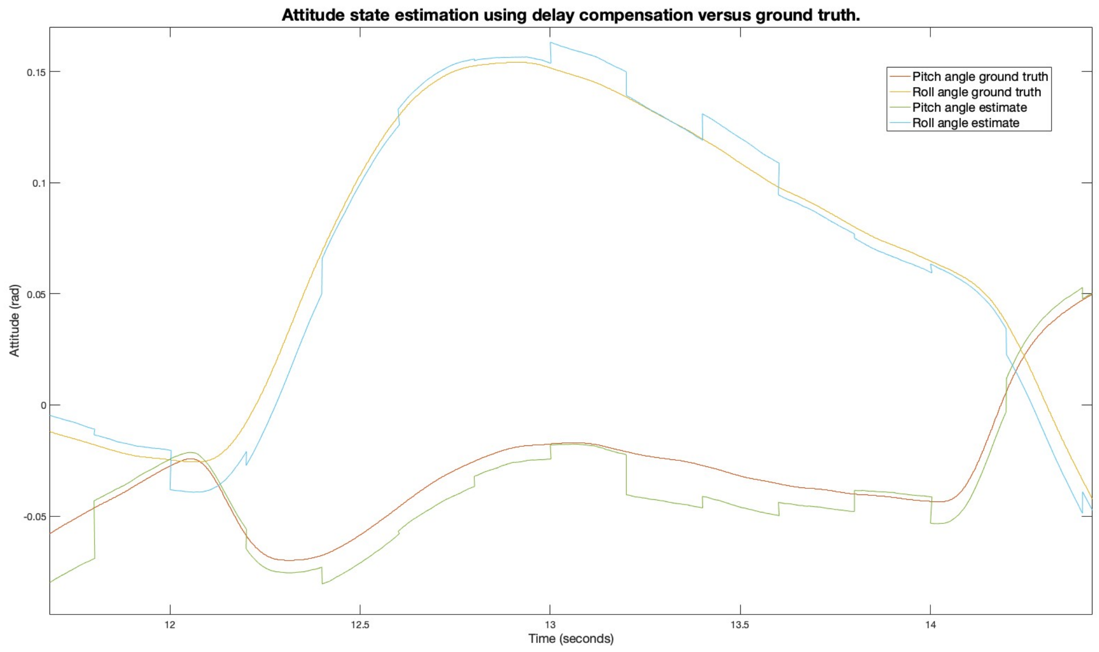


Figure 27: Zoomed in plot showing the result of the pitch and roll angle estimation with delay compensation versus ground truth.

5.5 Test accuracy for autonomous flight in closed loop simulation.

5.5.1 Method

This test gives us an answer whether the state estimate is accurate enough for performing autonomous agile flights. The agilicious platform contains support for closed-loop simulation that exists of a a low-level controller and a physics simulator.[2] The simulation allows to access the current quadrotor states real-time, so you can compose GPS, IMU and magnetometer messages by fetching the current state, adding noise and possibly some transformations. The messages are fed to the estimator, creating a closed-loop simulation. The same noise characteristics as in Section 5.4 are used to add noise for the GPS, IMU and magnetometer messages. Furthermore, the agilicious platform contains a sampler module, that takes a trajectory and calculates setpoints for the controller. Finally, the Model Predictive Controller (MPC) from the agilicious platform is used in this test, because the use of state-of-the-art technology is assumed in this thesis and this controller performed the best while testing. In this test, the complete agilicious platform is ran on a personal computer using the same software configuration as the computing board. The hardware of this personal computer is old, it is an 8-year old mid-range laptop. Two agile trajectories, fetched from [18] are used to test the accuracy of the state estimate. These trajectories can be flown at different speeds and will be explained later in this section.

In this test, it is examined whether the state estimator returns an accurate enough state estimate, such that the 2 trajectories can be flown autonomously. In this thesis, we specify that the state estimate is accurate enough if the position trajectory error is less than 0.2 meter. This is tested by flying the 2 trajectories, logging all the data and calculate the trajectory error using a Matlab script. Furthermore, the performance of the estimator is also tested using a benchmark. This benchmark is the same as the implementation, but uses perfect state estimates. By comparing the trajectory error of our implementation with the benchmark, the performance decrease can be formulated. Initially, the final Kalman coefficients of Section 5.4 without delay are used, but tuned to obtain an as low as possible trajectory error. When the 2 trajectories can be flown accurately, the trajectories are flown at higher speeds, increasing the agility of the trajectory.

Unfortunately, there was no option to introduce time delays in the simulation, so the delay compensation has not been tested.

Hovering and straight line

First, 2 simple trajectories are tested: hovering and straight line. The hovering is straight forward and consists of a fixed trajectory at the point (0, 0, 1). The straight line trajectory is a by agilicious generated trajectory that goes from (5, 0, 1) to (-5, 0, 1).

Trajectory 1

The lay-out of this trajectory can be found in Figures 28 and 29. This trajectory is characterized by 3D-turns and sharp corners. These 3D-turns require accurate control of the attitude of the quadrotor, so attitude state estimate it tested most noticeably in this trajectory. When flown at higher speed, the sharp corners require high rotational velocities and the quadrotor is subjected to high forces.

This trajectory can be flown in 3 different modes, increasing in speed and agility. The specifications of the trajectory under these modes can be found in Table 14.

	Mode 1	**Mode 2**	**Mode 3**
Average velocity (m/s)	7.18	9.90	11.41
Maximum velocity (m/s)	11.72	16.47	19.14
Maximum acceleration (m/sˆ2)	21.52	30.23	37.97
Maximum angular rate (deg/s)	324.59	522.39	700.30

Table 14: Table containing specification of trajectory 1 under 3 different modes.

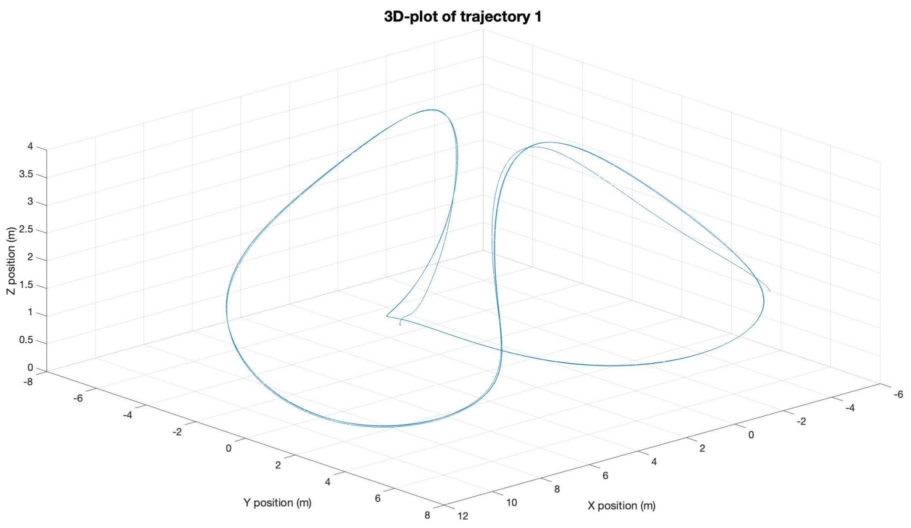


Figure 28: 3D-plot of trajectory 1.

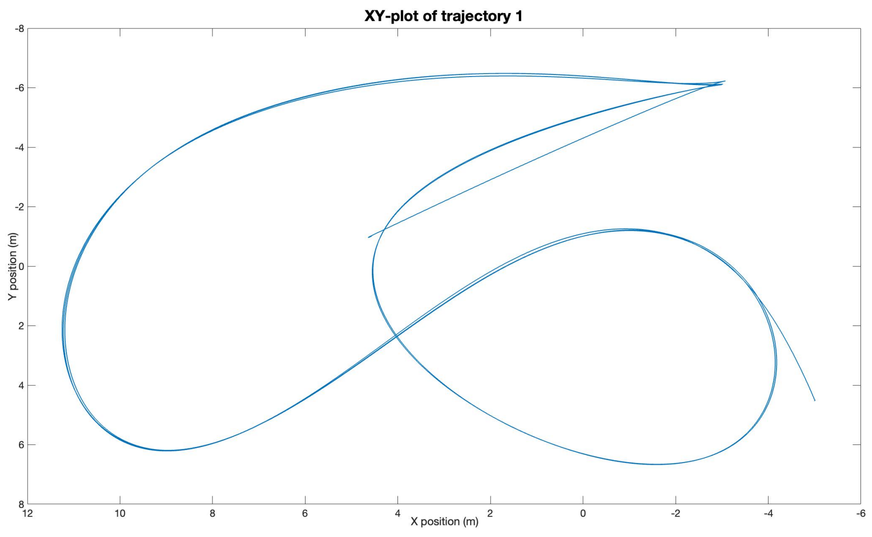


Figure 29: XY-plot of trajectory 1.

Trajectory 2

The lay-out of this trajectory can be found in Figures 30 and 31. This trajectory is characterized by its circular form. When flying at high speed, the drone is subjected to high centripetal force.

This trajectory can be flown in 2 different modes, increasing in speed and agility. The specifications of the trajectory under these modes can be found in Table 15.

	Mode 1	**Mode 2**
Average velocity (m/s)	6.57	8.70
Maximum velocity (m/s)	9.99	13.39
Maximum acceleration (m/sˆ2)	20.57	35.98
Maximum angular rate (deg/s)	120.81	173.84

Table 15: Table containing specification of trajectory 2 under 3 different modes.

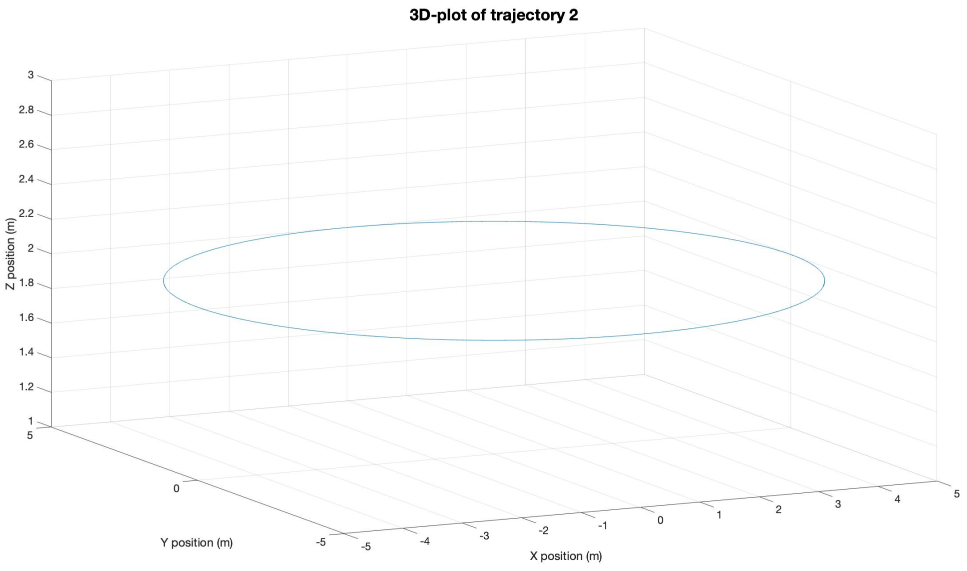


Figure 30: 3D-plot of trajectory 1.

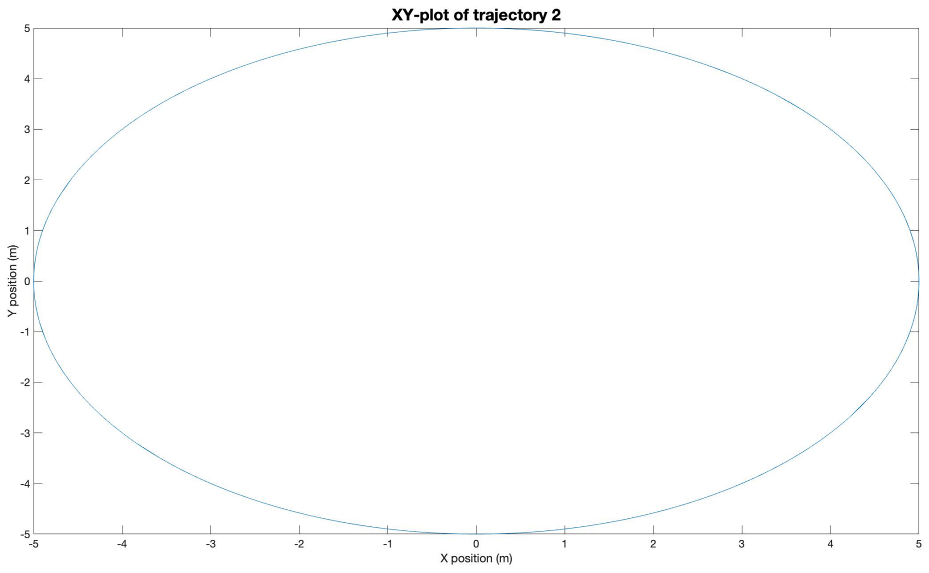


Figure 31: XY-plot of trajectory 1.

5.5.2 Results

Test accuracy of state estimate for autonomous flight with open-loop coefficients.
This section contains the results of testing accuracy of state estimate using tuned Kalman coefficients from the simulation data test without delay. These coefficients can be found in Section 5.4.2. Only mode 1 of trajectory 1 and 2 is tested, because this mode failed the test in both cases.
Trajectory 1 - Mode 1

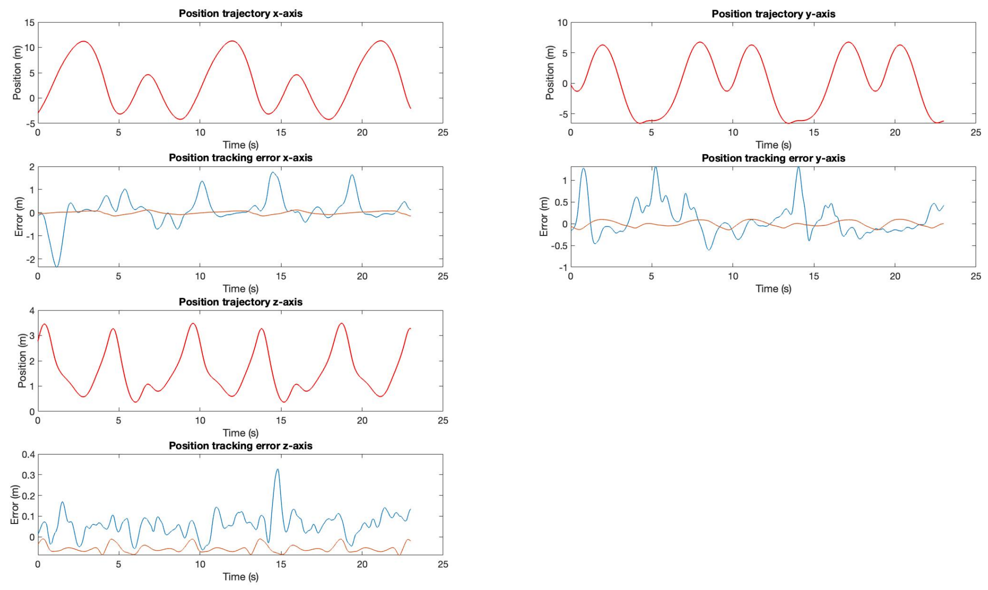


Figure 32: Position tracking results of trajectory 1 at mode 1 with open-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

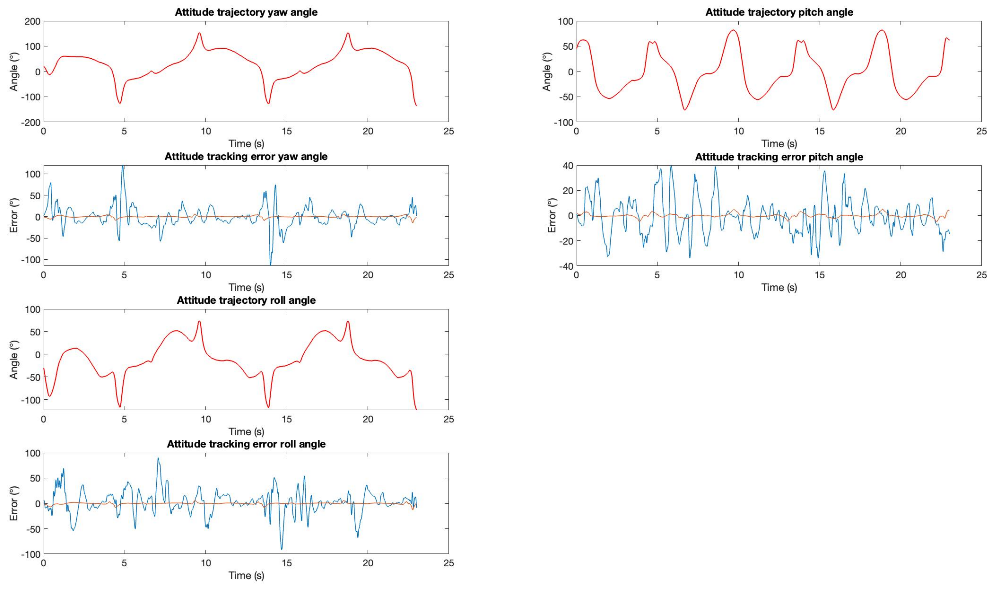


Figure 33: Attitude tracking results of trajectory 1 at mode 1 with open-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	14.46	1.30	+1010.62
Total attitude RMSE	20.44 °	1.84°	+1011.05
Yaw angle RMSE	22.64 °	2.13 °	+960.69
Pitch angle RMSE	14.40 °	1.47 °	+877.00
Roll angle RMSE	23.10 °	1.85 °	+1148.01
Total position RMSE	42.58 cm	6.38 cm	+567.58
X-axis RMSE	62.11 cm	6.34 cm	+880.16
Y-axis RMSE	38.88 cm	6.78 cm	+473.15
Z-axis RMSE	8.27 cm	5.99 cm	+38.15

Table 16: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 1, mode 1, open-loop coefficients.

Trajectory 2 - Mode 1

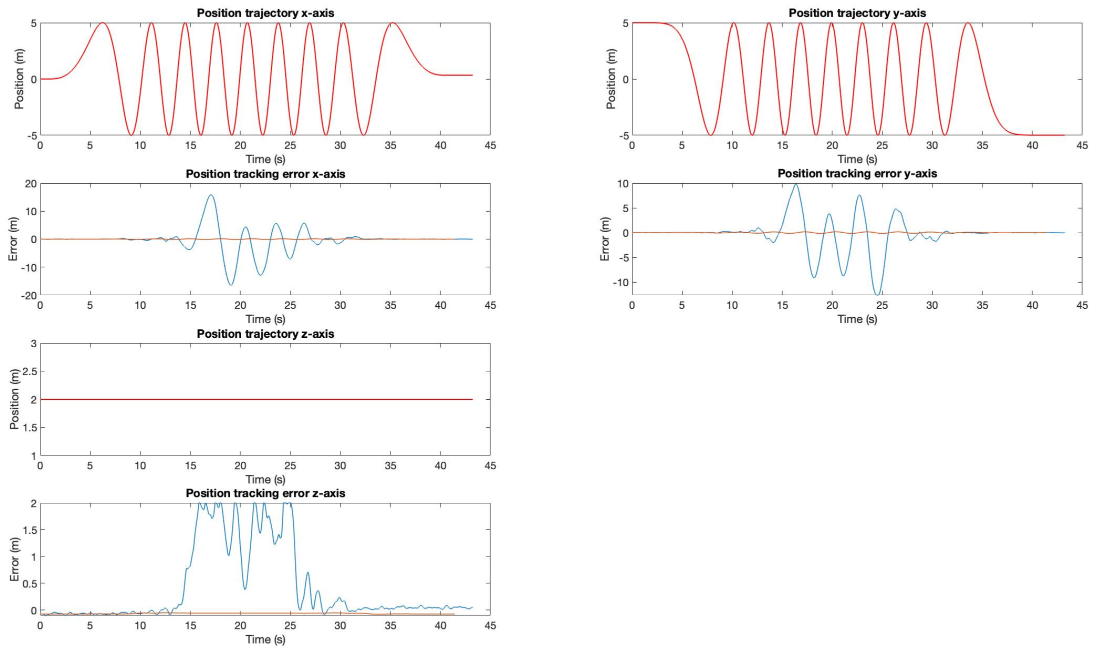


Figure 34: Position tracking results of trajectory 2 at mode 1 with open-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

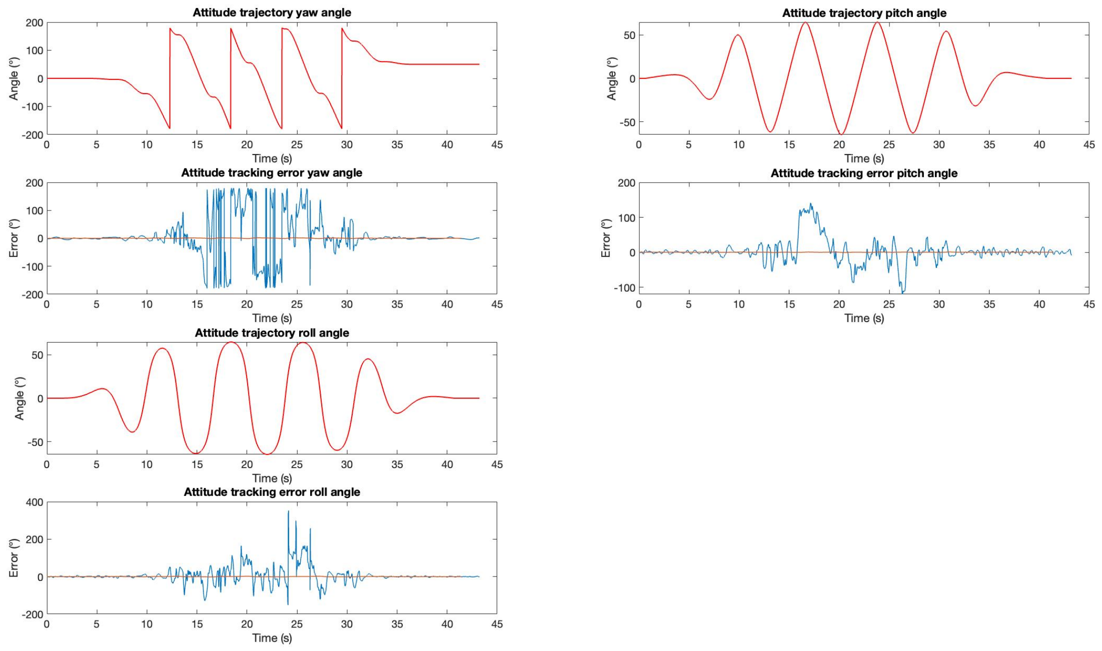


Figure 35: Attitude tracking results of trajectory 2 at mode 1 with open-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	30.44	0.42	+7100.20
Total attitude RMSE	42.94 °	0.59 °	+7154.39
Yaw angle RMSE	52.17 °	0.72 °	+7185.89
Pitch angle RMSE	33.63 °	0.26 °	+12677.11
Roll angle RMSE	40.99 °	0.69 °	+5882.88
Total position RMSE	319.47 cm	7.31 cm	+4268.98
X-axis RMSE	431.84 cm	7.71 cm	+5500.90
Y-axis RMSE	336.30 cm	7.75 cm	+4239.01
Z-axis RMSE	81.22 cm	6.39 cm	+1170.15

Table 17: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 2, mode 1, open-loop coefficients.

Test accuracy of state estimate for autonomous flight with tuned coefficients.
The final Kalman coefficients after tuning can be found in Table 18. The Kalman coefficients have been tuned by first minimizing the estimation error while hovering and then minimizing the tracking error of Trajectory 1 at different modes. *Q_att* has been tuned to zero, because the attitude update result was disturbed by influence of position measurements. This way, this influence is minimized. An overview of the results can be found in Table 19. The individual results of the different trajectories at different modes can be found below.

R_pos	1.0e-3
R_att	7.6e-1
R_acc	1.0e-1
R_omega	1.0e-5
Q_pos	1.0e-5
Q_att	0
Q_vel	1.0e-5
Q_bome	3.0e-4
Q_bacc	1.0e-1

Table 18: Resulting Kalman coefficients after closed-loop tuning.

	Trajector 1 Mode 1	**Trajectory 1 Mode 2**	**Trajectory 1 Mode 3**	**Trajectory 2 Mode 1**	**Trajectory 2 Mode 2**
Total RMSE	-6.72	-4.58	-2.30	+26.82	+15.50
Total attitude RMSE	-6.72	-4.58	-2.30	+27.18	+15.59
Yaw angle RMSE	-7.40	-8.00	-2.59	+17.10	+6.51
Pitch angle RMSE	-3.50	-2.76	+2.45	+108.59	+34.29
Roll angle RMSE	-7.90	-3.04	-2.56	+21.77	+33.70
Total position RMSE	-4.83	-5.23	-9.49	+1.02	-0.12
X-axis RMSE	-5.31	-3.64	-9.88	+4.67	-4.57
Y-axis RMSE	-9.51	-7.86	-12.07	-1.47	-0.76
Z-axis RMSE	+1.35	-1.76	-1.44	-0.80	+6.29

Table 19: Table containing overview of relative change in percentage of tracking error of implementation, in comparison with the benchmark, on both trajectories and all modes.

Hovering

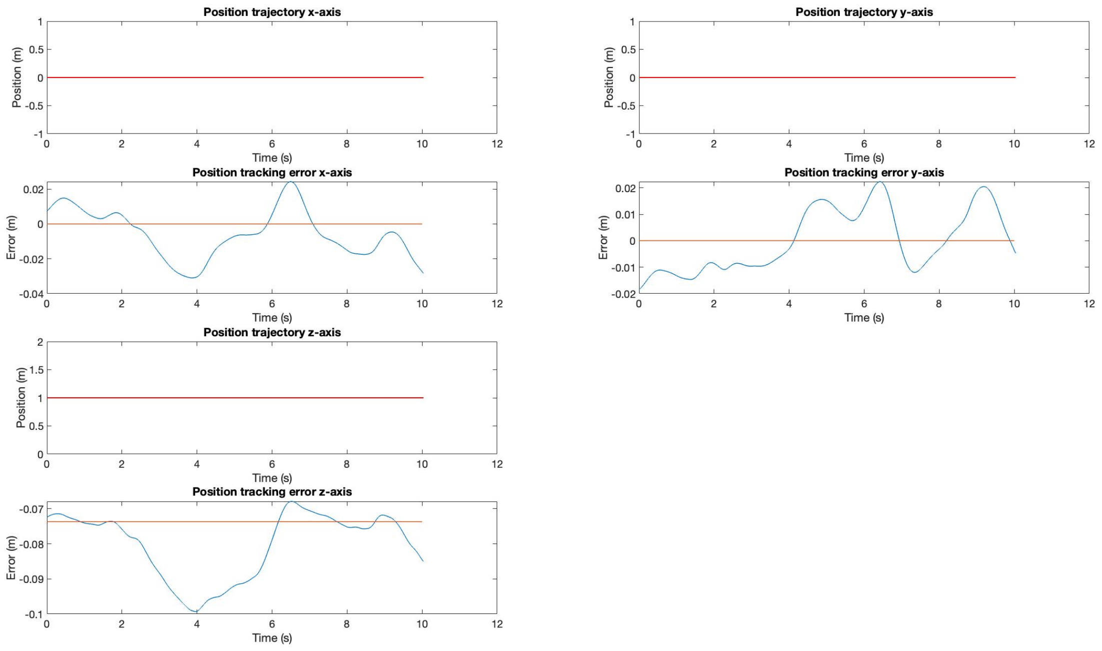


Figure 36: Position tracking results of hovering with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

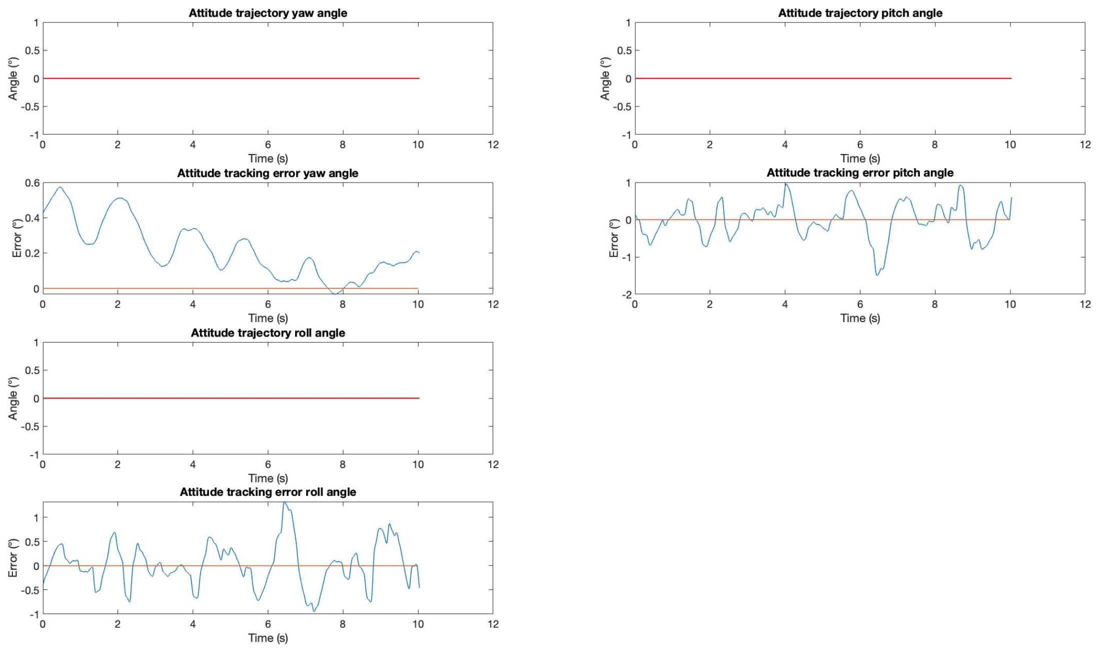


Figure 37: Attitude tracking results of hovering with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	0.28	0.03	+843.51
Total attitude RMSE	0.40 °	0 °	+Inf
Yaw angle RMSE	0.27 °	0 °	+Inf
Pitch angle RMSE	0.46 °	0 °	+Inf
Roll angle RMSE	0.44 °	0 °	+Inf
Total position RMSE	4.79 cm	4.26 cm	+12.53
X-axis RMSE	1.49 cm	0 cm	+Inf
Y-axis RMSE	1.16 cm	0 cm	+Inf
Z-axis RMSE	8.08 cm	7.37 cm	+9.57

Table 20: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Hovering, closed-loop coefficients.

Straight line trajectory

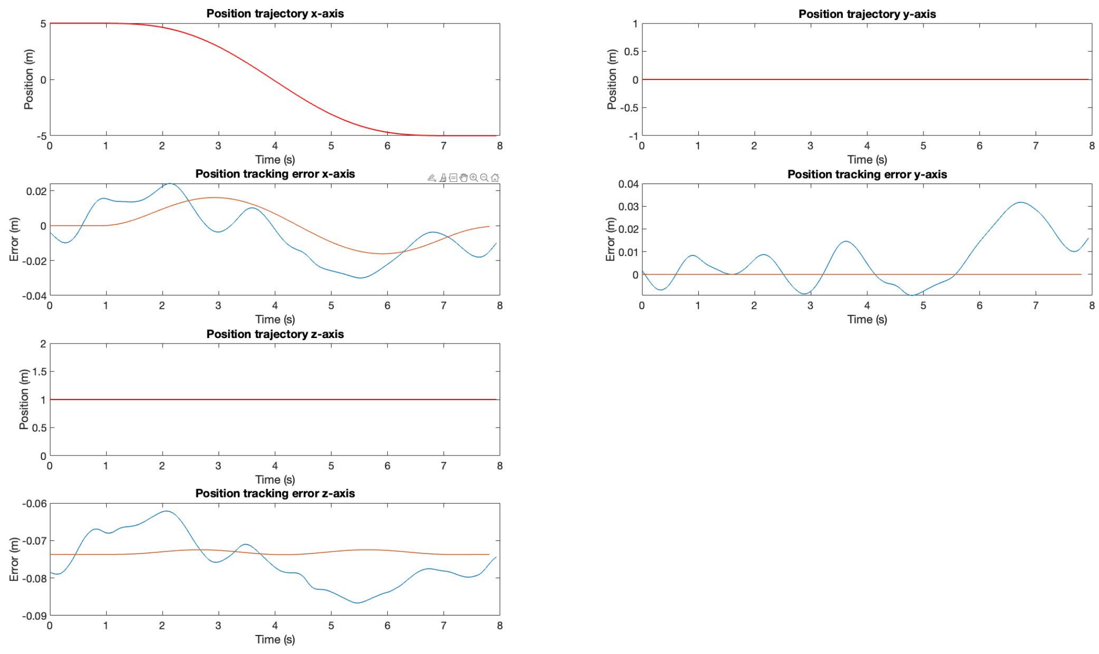


Figure 38: Position tracking results of straight line trajectory with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

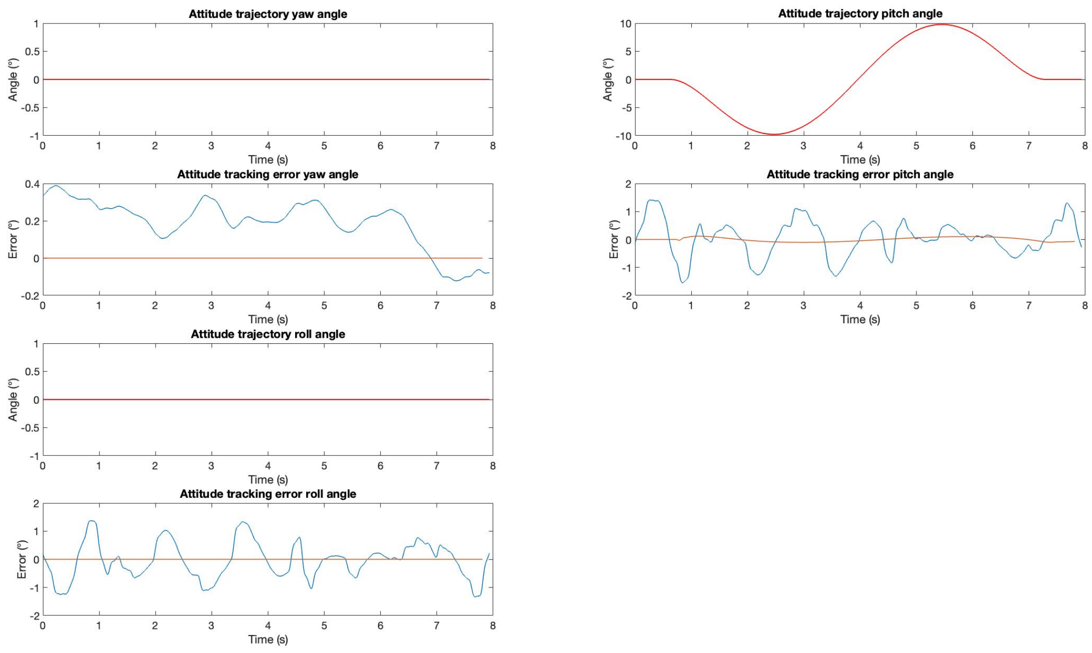


Figure 39: Attitude tracking results of straight line trajectory with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	0.39	0.04	+815.42
Total attitude RMSE	0.54 °	0.04 °	+1205.32
Yaw angle RMSE	0.23 °	0 °	+Inf
Pitch angle RMSE	0.65 °	0.07 °	+804.65
Roll angle RMSE	0.64 °	0 °	+Inf
Total position RMSE	4.54 cm	4.27 cm	+6.41
X-axis RMSE	1.58 cm	1.02 cm	+55.30
Y-axis RMSE	1.26 cm	0 cm	+Inf
Z-axis RMSE	7.60 cm	7.32 cm	+3.82

Table 21: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Straight line trajectory, closed-loop coefficients.

Trajectory 1 - Mode 1

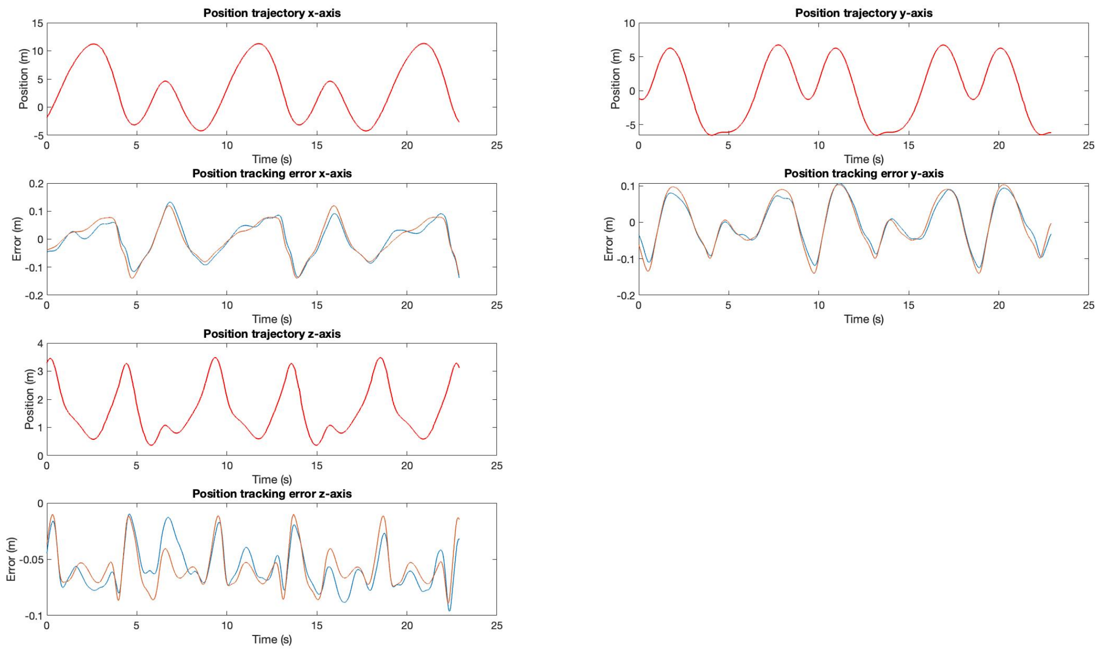


Figure 40: Position tracking results of trajectory 1 at speed 16 with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

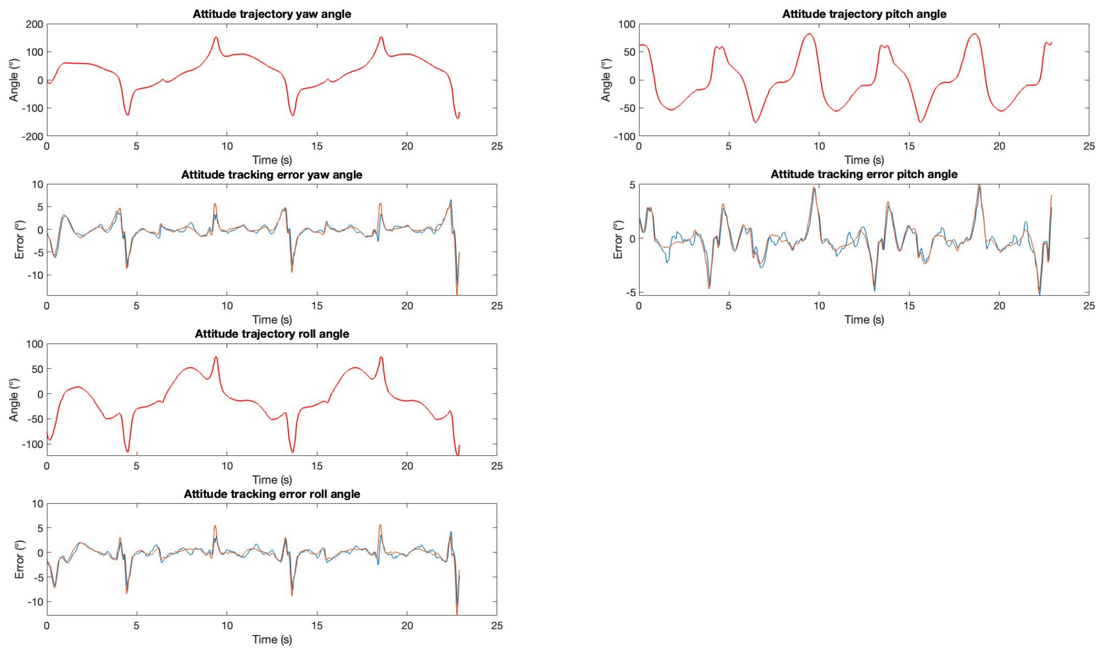


Figure 41: Attitude tracking results of trajectory 1 at speed 16 with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	1.21	1.30	-6.72
Total attitude RMSE	1.71 °	1.84°	-6.72
Yaw angle RMSE	1.98 °	2.13 °	-7.40
Pitch angle RMSE	1.42 °	1.47 °	-3.50
Roll angle RMSE	1.70 °	1.85 °	-7.90
Total position RMSE	6.07 cm	6.38 cm	-4.83
X-axis RMSE	6.00 cm	6.34 cm	-5.31
Y-axis RMSE	6.14 cm	6.78 cm	-9.51
Z-axis RMSE	6.07 cm	5.99 cm	+1.35

Table 22: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 1, mode 1, closed-loop coefficients.

Trajectory 1 - Mode 2

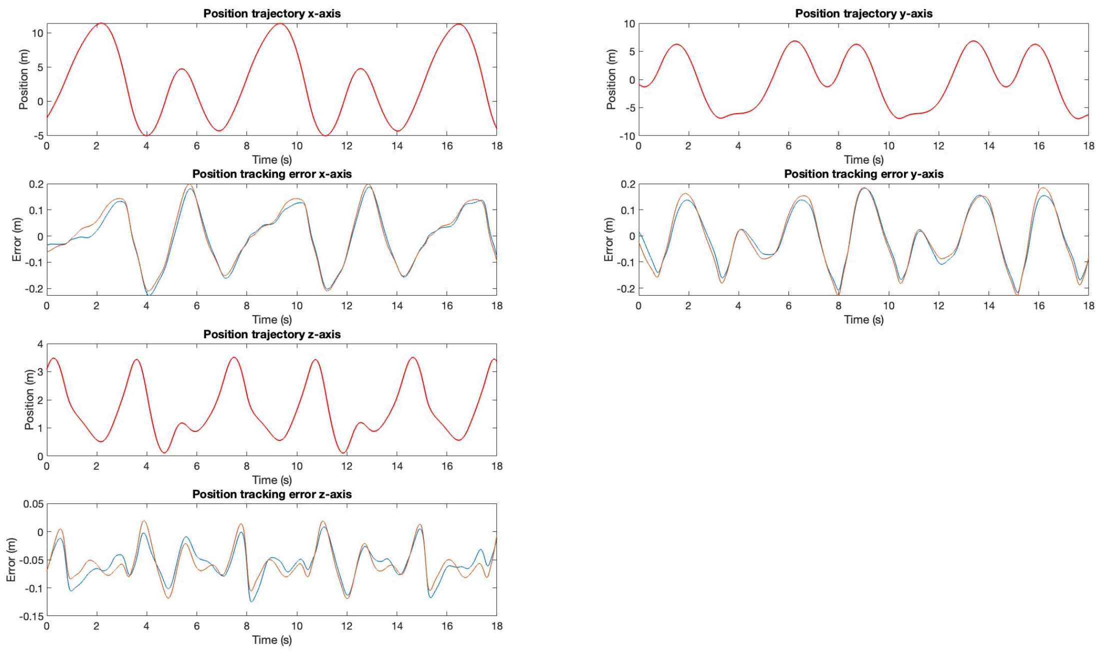


Figure 42: Position tracking results of trajectory 1 at speed 25 with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

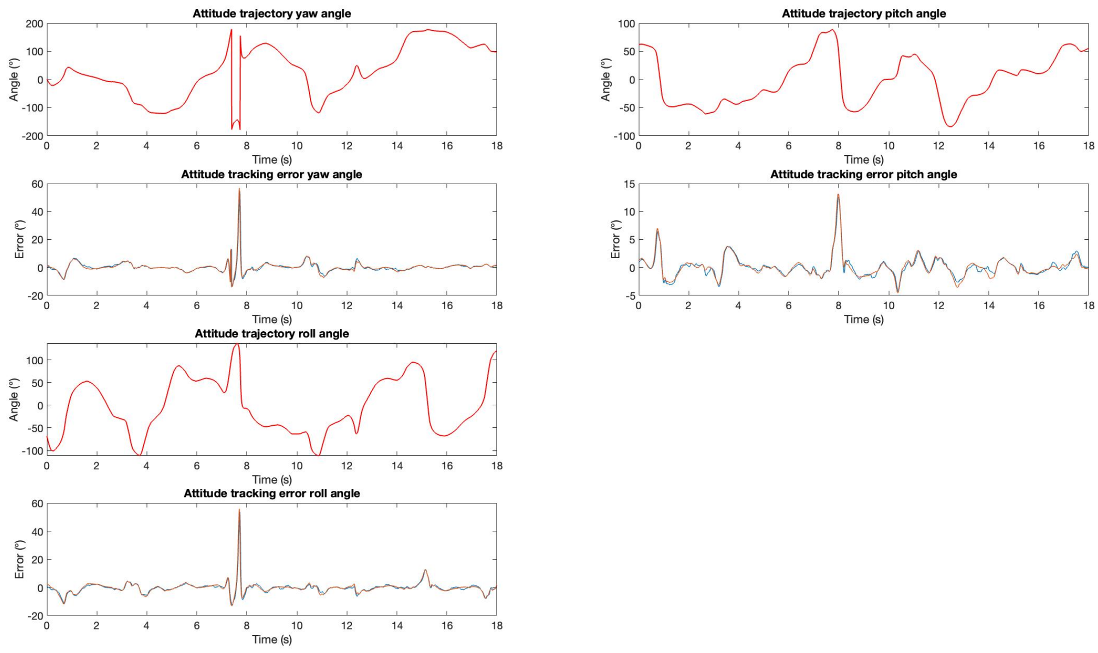


Figure 43: Attitude tracking results of trajectory 1 at speed 25 with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	2.32	2.43	-4.58
Total attitude RMSE	3.28 °	3.44 °	-4.58
Yaw angle RMSE	3.12 °	3.39 °	-8.00
Pitch angle RMSE	1.88 °	1.92 °	-2.76
Roll angle RMSE	4.37 °	4.51 °	-3.04
Total position RMSE	9.22 cm	9.73 cm	-5.23
X-axis RMSE	10.35 cm	10.74 cm	-3.64
Y-axis RMSE	10.41 cm	11.30 cm	-7.86
Z-axis RMSE	6.29 cm	6.41 cm	-1.76

Table 23: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 1, mode 2, closed-loop coefficients.

Trajectory 1 - Mode 3

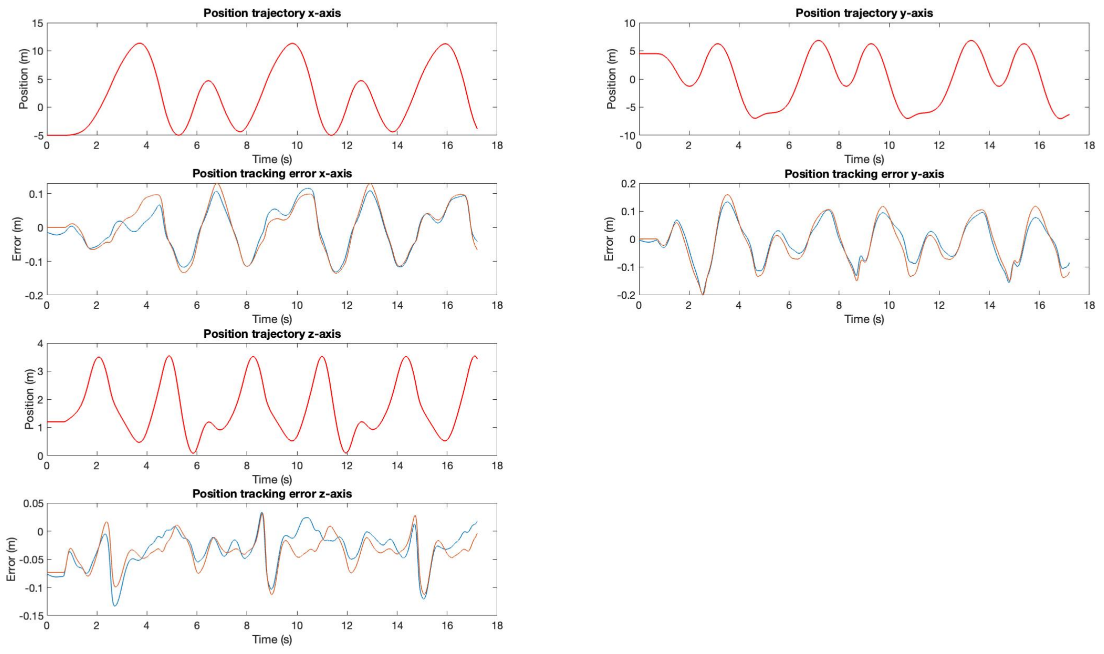


Figure 44: Position tracking results of trajectory 1 at speed 33 with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

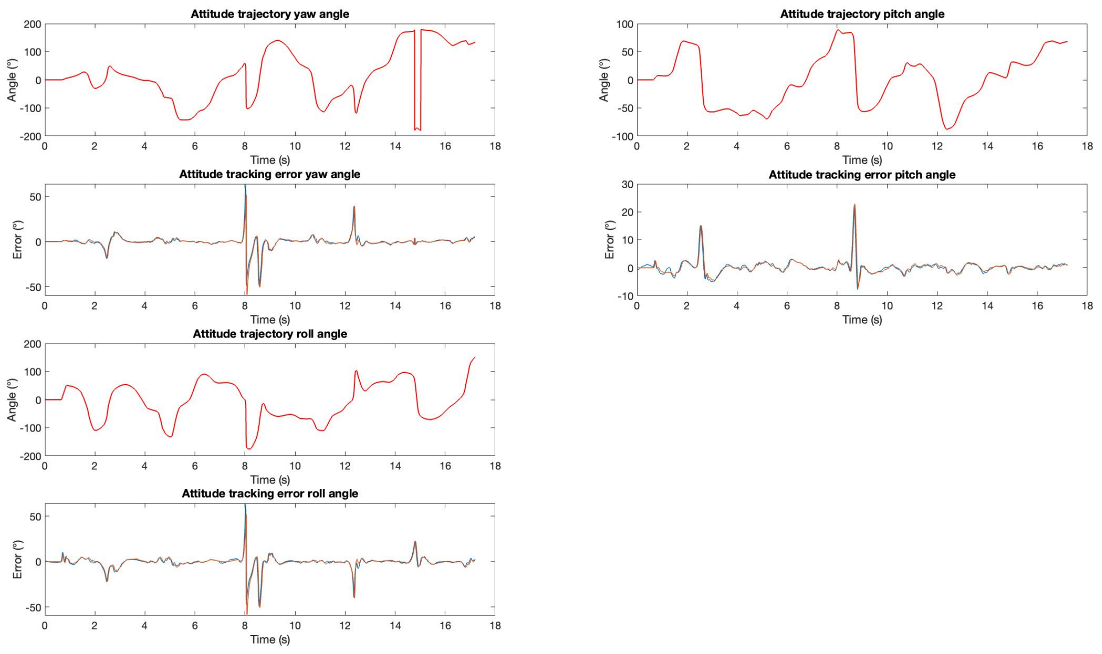


Figure 45: Attitude tracking results of trajectory 1 at speed 33 with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	4.17	4.26	-2.30
Total attitude RMSE	5.90 °	6.03°	-2.30
Yaw angle RMSE	6.90 °	7.08 °	-2.59
Pitch angle RMSE	2.49 °	2.43 °	+2.45
Roll angle RMSE	7.11 °	7.30 °	-2.56
Total position RMSE	6.14 cm	6.78 cm	-9.49
X-axis RMSE	6.47 cm	7.18 cm	-9.88
Y-axis RMSE	7.07 cm	8.04 cm	-12.07
Z-axis RMSE	4.62 cm	4.69 cm	-1.44

Table 24: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 1, mode 3, closed-loop coefficients.

Trajectory 2 - Mode 1

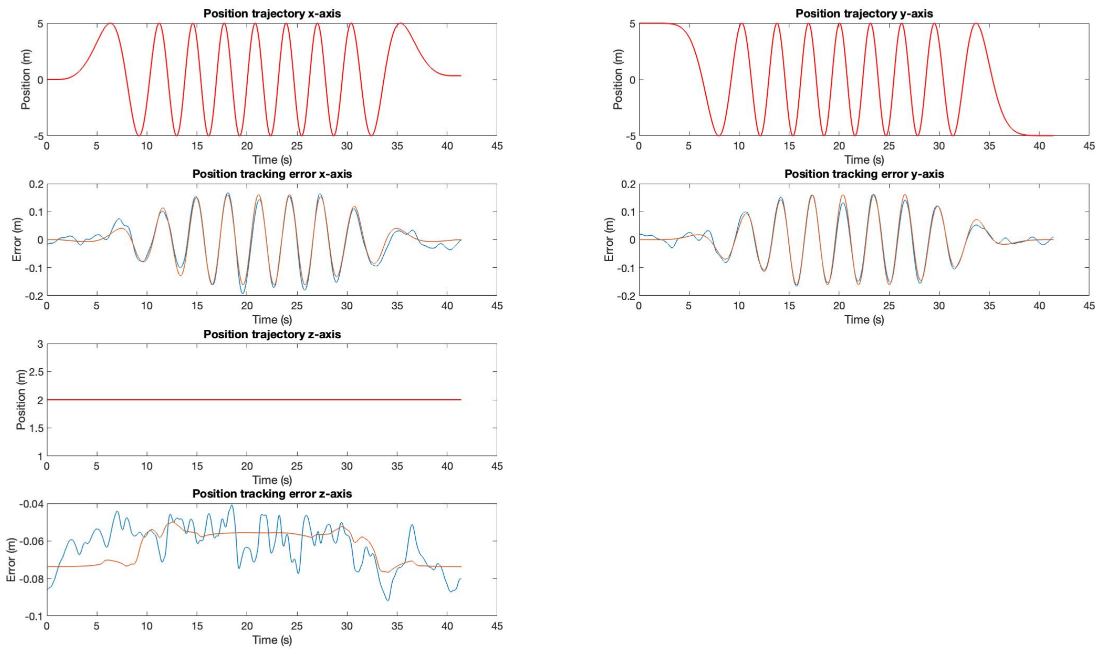


Figure 46: Position tracking results of trajectory 2 at speed 10 with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

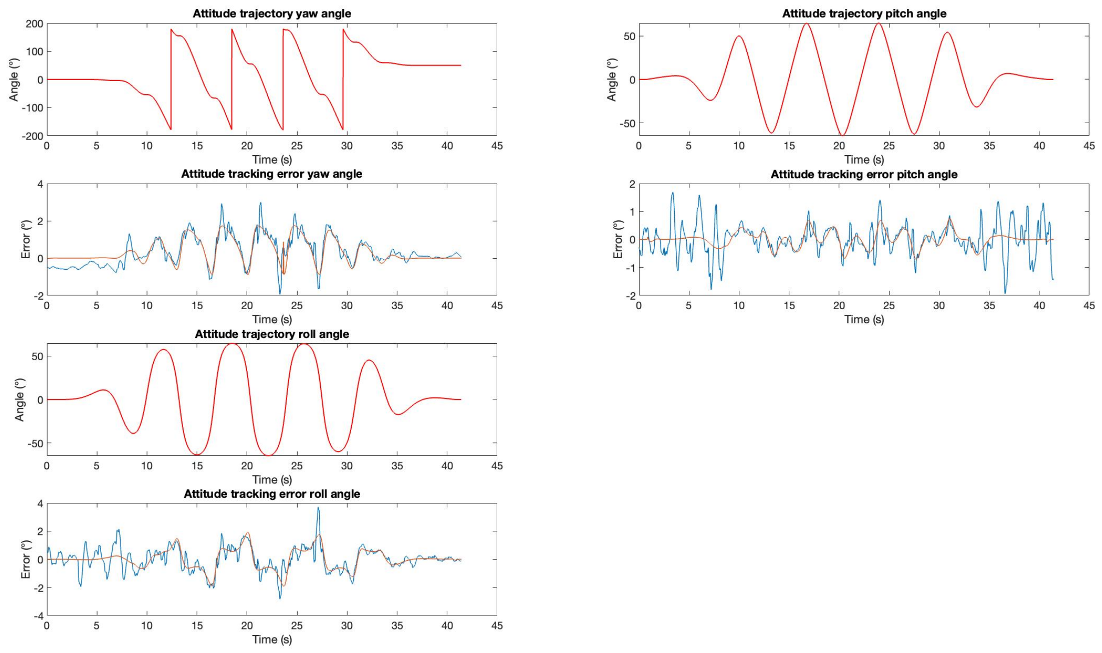


Figure 47: Attitude tracking results of trajectory 2 at speed 10 with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	0.53	0.42	+26.82
Total attitude RMSE	0.75 °	0.59 °	+27.18
Yaw angle RMSE	0.84 °	0.72 °	+17.10
Pitch angle RMSE	0.55 °	0.26 °	+108.59
Roll angle RMSE	0.83 °	0.69 °	+21.77
Total position RMSE	7.39 cm	7.31 cm	+1.02
X-axis RMSE	8.07 cm	7.71 cm	+4.67
Y-axis RMSE	7.64 cm	7.75 cm	-1.47
Z-axis RMSE	6.34 cm	6.39 cm	-0.80

Table 25: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 2, mode 1, closed-loop coefficients.

Trajectory 2 - Mode 2

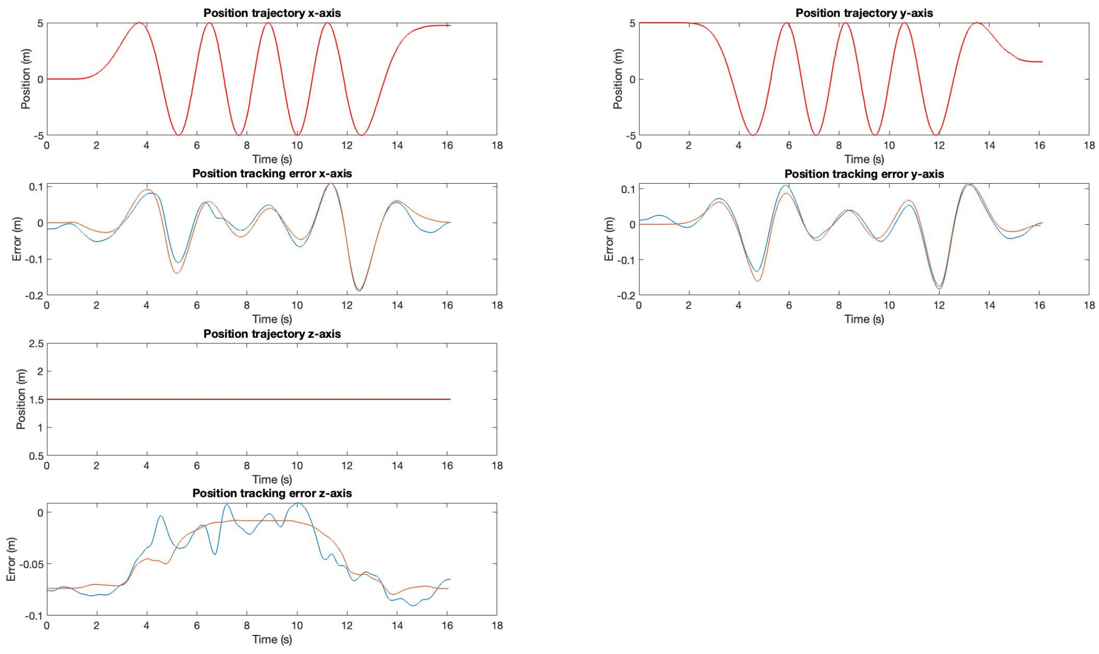


Figure 48: Position tracking results of trajectory 2 at speed 14 with closed-loop coefficients. Red: Position trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

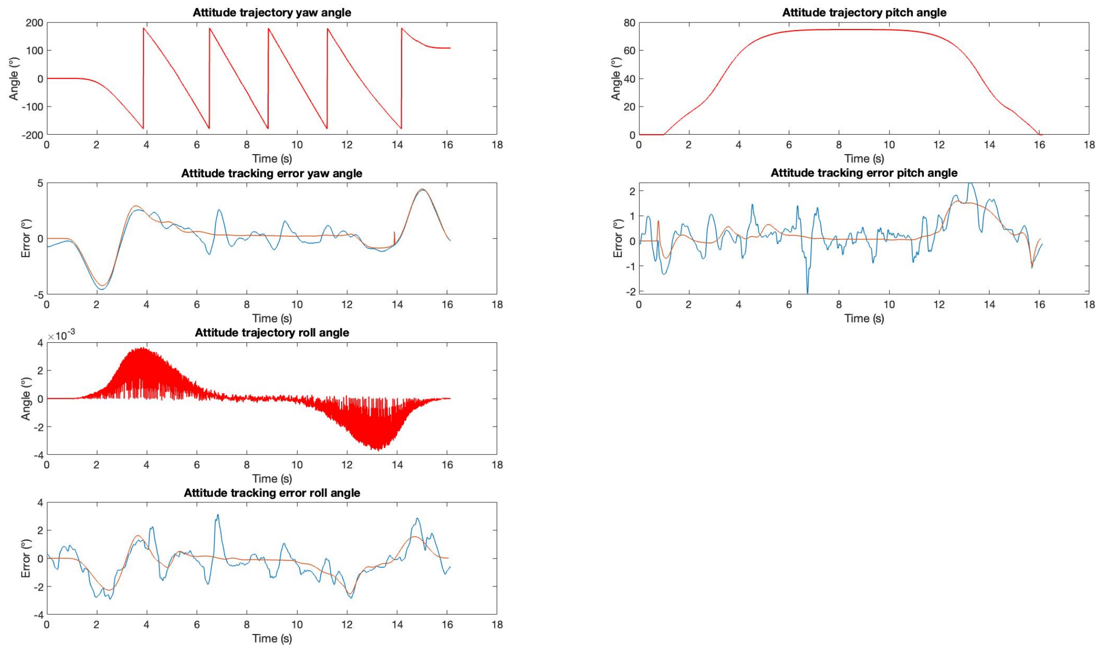


Figure 49: Attitude tracking results of trajectory 2 at speed 14 with closed-loop coefficients. Red: Attitude trajectory, Blue: Implementation tracking error, Orange: Benchmark tracking error.

	Implementation	**Benchmark**	**Relative change (%)**
Total RMSE	0.90	0.78	+15.5
Total attitude RMSE	1.27 °	1.10 °	+15.59
Yaw angle RMSE	1.69 °	1.59 °	+6.51
Pitch angle RMSE	0.75 °	0.56 °	+34.29
Roll angle RMSE	1.20 °	0.90 °	+33.70
Total position RMSE	5.76 cm	5.76 cm	-0.12
X-axis RMSE	5.72 cm	6.00 cm	-4.57
Y-axis RMSE	6.00 cm	6.05 cm	-0.76
Z-axis RMSE	5.54 cm	5.21 cm	+6.29

Table 26: Root mean square trajectory error (RMSE) of implementation, benchmark and comparison. Trajectory 2, mode 2, closed-loop coefficients.

5.6 Test maximum noise level of measurements for autonomous flight.

5.6.1 Method

In this test, it is examined at which measurement noise levels the state estimate becomes so inaccurate, the specification for autonomous flight fails. This is done by considering 3 scenarios, which are explained below. In these 3 scenarios, the targeted noise level is iteratively increased and it is checked whether the hovering is stable and the state estimation is still accurate enough according to the specification. When the test fails at a given noise level, the Kalman filter noise parameters are tuned to try improving the results. The same setup as the closed-loop test in Section 5.5 is used and trajectory 1 at mode 3 is used for this test.

- Scenario 1: Initial noise level on IMU and position measurements, increased noise level on attitude measurements.
- Scenario 2: Initial noise level on IMU and attitude measurements, increased noise level on position measurements.
- Scenario 3: Initial noise level on attitude and position measurements, increased noise level on IMU measurements.

5.6.2 Results

Scenario 1

The maximum noise levels for this scenario can be found in Table 5.6.2. Next to the attitude noise, the angular rate measurement noise has also been increased, because the test kept succeeding at very high noise levels. Eventually, it was deemed more insightful to fix the attitude measurement noise level and increase the angular rate measurement noise level. The Kalman coefficients have not been altered in this scenario. The final total position tracking error was 0.42 meter.

Data	**Original Standard Deviation**	**Maximum Standard Deviation**
Position	0.01 m	0.01 m
Acceleration	0.1 m/s	0.1 m/s
Angular Rate	1 deg/s	57.3 deg/s
Attitude	0.5 deg	20 deg
Magnetic Field	0.05	0.05

Table 27: Table containing maximum noise levels for scenario 1.

Scenario 2

The maximum noise levels for this scenario can be found in Table 5.6.2. The tuned Kalman coefficients can be found in 5.6.2. The final total position tracking error was 0.24 meter.

Data	**Original Standard Deviation**	**Maximum Standard Deviation**
Position	0.01 m	0.5 m
Acceleration	0.1 m/s	0.1 m/s
Angular Rate	1 deg/s	1 deg/s
Attitude	0.5 deg	0.5 deg
Magnetic Field	0.05	0.05

Table 28: Table containing maximum noise levels for scenario 2.

R_pos	1.0e-1
R_att	7.6e-1
R_acc	1.0e-1
R_omega	1.0e-5
Q_pos	1.0e-5
Q_att	0
Q_vel	1.0e-5
Q_bome	3.0e-4
Q_bacc	1.0e-1

Table 29: Table containing tuned Kalman coefficients for scenario 2.

Scenario 3

The maximum noise levels for this scenario can be found in Table 5.6.2. The tuned Kalman coefficients can be found in 5.6.2. The final total position tracking error was 0.17 meter. The test was stopped, because the hovering became unstable.

Data	**Original Standard Deviation**	**Maximum Standard Deviation**
Position	0.01 m	0.01 m
Acceleration	0.1 m/s	1.25 m/s
Angular Rate	1 deg/s	42.97 deg/s
Attitude	0.5 deg	0.5 deg
Magnetic Field	0.05	0.05

Table 30: Table containing maximum noise levels for scenario 3.

R_pos	1.0e-3
R_att	7.6e-1
R_acc	1.0e+1
R_omega	1.0e-3
Q_pos	1.0e-5
Q_att	0
Q_vel	1.0e-5
Q_bome	3.0e-4
Q_bacc	1.0e-1

Table 31: Table containing tuned Kalman coefficients for scenario 3.

6 Conclusion

This thesis aimed to answer the question whether it is possible with the current state of technology, to obtain a state estimate based on GPS and inertial measurements, that is accurate enough to perform autonomous flight of agile trajectories in outdoor environments. Based on the results of the closed-loop simulation in Section 5.5, it can be concluded that the state estimates from the designed state estimation solution are accurate enough such that the tracking error is smaller than 0.2 m, in a closed-loop simulation using artificial measurements with noise-levels based on the real sensors. Furthermore, is has been shown that these noise-levels can be increased even further, indicating there is leeway for when performance of sensors turn out to be worse than indicated.

These 2 results give a strong indication that it is possible with some improvements, to use the implementation presented in this thesis in real-life flight and perform autonomous flight of agile trajectories.

Another question this thesis aimed to answer is if an extended Kalman filter is sufficient to obtain an accurate state estimation. From literature research and results from this thesis, it has been proven that you can obtain state estimations accurate enough for agile autonomous flight, using an extended Kalman filter.

The final research question is if it is feasible to run a state estimation solution accurate enough for agile autonomous flight, real-time, without timing violations. This research question cannot be answered, because the closed-loop simulation ran on a personal computer instead of compact hardware due to lack of computing power. The designed state estimation algorithm is feasible to run on the compact hardware, but it is unknown if the resulting state estimation is accurate enough for less performing controllers.

7 Discussion

The results of the position state estimation with delay compensation in Section 5.4.2 are worse with respect to the position state estimation without delay compensation. The update step introduces square-wave like noise in the state estimate and the prediction step does not follow the ground truth. This indicates that something fails integrating the acceleration. In the prediction step, the acceleration is multiplied with the rotational matrix to convert measurements from body-frame to world-frame. When the attitude has an error, the acceleration will be converted faulty, introducing errors in the measurements. In the test without delay compensation, this is visible when zooming in, but the position measurement is very accurate, directing the estimate back to the ground truth very quickly, causing this acceleration error not to have a lot of influence. In the test with delay compensation, after the update step, there is another prediction/propagation step to the current time, integrating the saved acceleration measurements over the delay time. This causes the influence of the acceleration measurement to grow, so the error becomes more visible. Furthermore, the prediction of the position is calculated by integrating the velocity and the velocity is calculated by integrating the acceleration and updated in the update step. The velocity and attitude states are connected in the Kalman filter matrices, causing the velocity to be dependent of the attitude measurements. This also causes some error in the velocity estimate, thus position estimate. This can be solved by tuning the Kalman coefficients, though this tuning can also increase estimation error of other states. This has been tried and the best results have been presented in the Results section.

In Results Section 5.3.2, the results of the state estimation with captured measurements on trajectory 1 are accurate. On trajectory 1, a square is traversed clock-wise, first facing north. There is no noise present of the estimate and the X- and Y estimate come quite close to the 10 meters. It can be explained that the estimate does not actually reach the 10 meters, by the reason that in the traversed square, you walk not perfectly in north, east, south and west direction, causing a reduced measurement.

In Results Section 5.3.2, the results of the state estimation with captured measurements on trajectory 2 are hard to interpret. In this test, the test setup is rotated 180 degrees around the roll axis with increasing rotational frequency. It is visible that the roll axis varies between zero and 180 degrees and that the angular rate also increases. But, the pitch and roll angle also vary, which are fixed in the measurement, so this is invalid behaviour. Most likely, the angles get locked in the measurement or state estimate and get dependent on each other. This can also be seen as a 180 degree roll rotation can also be given as a 180 degree pitch rotation and a 180 degree yaw rotation. More research is needed, to find a solution for this.

Unfortunately, due to problems with the magnetometer and attitude estimation in the validation test with captured measurement data in Section 5.3, it was decided to focus on simulation results, instead of real-life results. When testing the magnetometer, it was found that the results were not valid. Weird magnetometer data samples were received and I was not able to correct them using some form of calibration. For example, during one test, when facing the magnetometer north, a large magnetic field was measured along the x-axis, which is a valid measurement. When you rotate the magnetometer 180 degrees, you would expect to measure a large negative magnetic field along the x-axis, but still a positive magnetic field was measured. This caused a lot of problems during the real-life tests. Furthermore, it was hard to improve the attitude estimation results using captured measurement data, because all the measurements were coded, thus not easy to play around with or make comparisons. Using simulation data, this was much easier.

In Results Section 5.5.2, Table 19, the results of the closed-loop tests indicated that the tracking error using the implementation was lower than the tracking error of the benchmark on Trajectory 1. This is surprising, because the benchmark contains a perfect estimator, so should give the minimum tracking error. Controllers can be very complex and sometimes it is possible that input data with small errors actually decrease the tracking error. Most likely, this is also the case in this situation. When tuning the Kalman coefficients, the trajectory error of trajectory 1 was minimized. Presumably, this did not minimize the estimation error, but formed an optimal state estimate for the controller to track this specific trajectory. The benchmark does perform better than the implementation while hovering, on the straight line trajectory and trajectory 2, which is logical, because the Kalman filter was not tuned to these trajectories. This is an interesting result however, because you can tune your state estimator to specifications of a trajectory and perhaps change Kalman coefficients on-the-fly, based on your quadrotors trajectory planning.

I have several recommendations for future research. The next step of this research would be to test the state estimator in real-life by letting it fly a trajectory autonomously. Unfortunately, the computing board used in this thesis is not powerful enough to run MPC real-time, without timing violations. The use of other controllers has to be investigated whether they can run on compact hardware and have sufficient performance to fly autonomous flight with the implemented state estimator. Another option is to examine other compact computing boards whether they have the computing power to run MPC. When one of these options is successful, the implementation can be tested in a real-life flying experiment. Furthermore, a calibration algorithm for the magnetometer has to be developed, or another magnetometer module has to be chosen, before real-life flying experiments can commence. Another interesting option is to perform the magnetometer calibration in the Kalman filter, by adding an magnetometer bias state to the state estimator.

When the tracking error is found to be too high, a suggestion to decrease it is to decouple position and attitude update step. For simplicity, in this thesis, it is assumed that position and attitude measurements contain the same delay, but in real-life, this is not true. In order to decouple this, a serious overhaul of the estimator is needed and this was not viable in this thesis due to time.

8 Appendix

8.1 Summary on Kalman filter theory.

The Kalman filters are the best known state estimation algorithms. They use sensor data-fusion to calculate an accurate state estimate. This means that data at different sample rates from different sensors can be combined into one state estimate. There are several versions of a Kalman filter, increasing in complexity:

8.1.1 Kalman Filter Introduction

A Kalman filter uses a model and noisy measurements of inputs outputs of the to-be predicted system and combines them to calculate a state estimation. An abstract overview of an application of a Kalman filter can be found in Figure 50.

In this overview, the $\mathbf{u}$ vector is the input state vector and the $\mathbf{z}$ vector is the output state vector. Both these vectors contain states that are measured. Furthermore, the $\mathbf{x}$ vector contains all the to-be estimated states, which you can or cannot measure. The, $\mathbf{w}$ and $\mathbf{v}$ vectors are noise vectors, representing respectively the zero-mean Gaussian process noise with covariance Q and measurement noise with covariance R. Finally, the F, B and H are transformation matrices containing information about the system.

In the *Real System*, the vector $\mathbf{x}_k$ consists of the previous state of the vector $\mathbf{x}$ increased with the input $\mathbf{u}_{k-1}$ and the process noise. Furthermore, the output state vector, $\mathbf{z}$, is based on the $\mathbf{x}$ state vector and the measurement noise. In the *Model of System*, the state vectors are similar, only the process and measurement noise are neglected as this is a pure mathematical model.

As the *Model of System* is not perfect, the $\mathbf{x}_k$ and $\hat{\mathbf{x}}_k$ state vector will differ. This is also the case for the $\mathbf{z}_k$ and $\hat{\mathbf{z}}_k$ state vector, but because this is measurable, we can calculate the error state vector, $\tilde{\mathbf{y}}_k = \mathbf{z}_k$ - $\hat{\mathbf{z}}_k$, and tune the model to behave as the *Real System* as much as possible. This tuning can be done by creating a feedback loop of the error state vector multiplied by some gain K back to the model. Using this, we can obtain an accurate state estimation for $\mathbf{x}$. This is the basic working of the Kalman filter. In a Kalman filter, a gain K is calculated using a algorithm that uses the noise characteristics of the process and the different sensors, such that this gain ensures the most optimal state estimate.

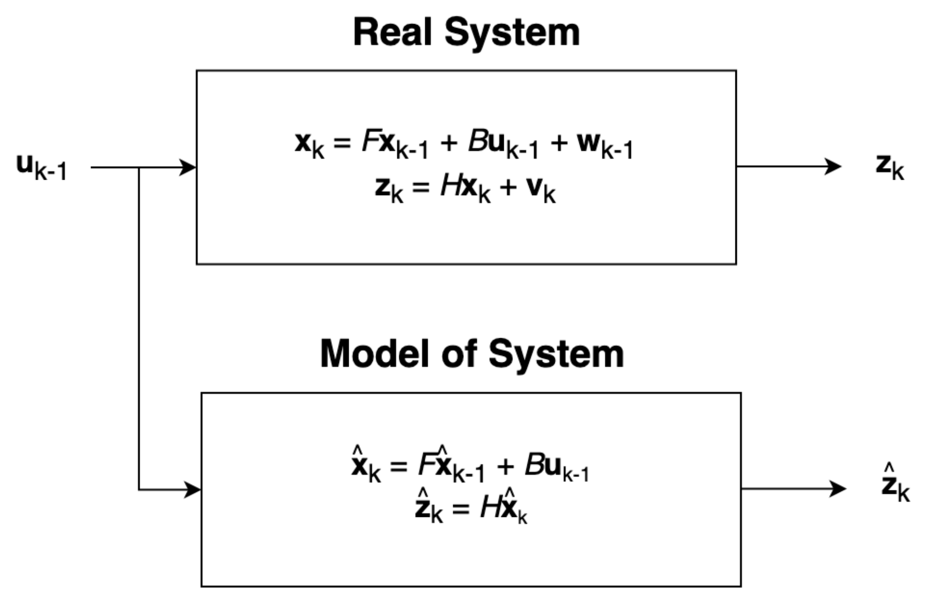

Real System
$\mathbf{x}_k = F\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{w}_{k-1}$
$\mathbf{z}_k = H\mathbf{x}_k + \mathbf{v}_k$
$\mathbf{u}_{k-1}$
$\mathbf{z}_k$
Model of System
$\hat{\mathbf{x}}_k = F\hat{\mathbf{x}}_{k-1} + B\mathbf{u}_{k-1}$
$\hat{\mathbf{z}}_k = H\hat{\mathbf{x}}_k$
$\hat{\mathbf{z}}_k$

Figure 50: Abstract graphical overview of an application of a Kalman filter.

8.1.2 Kalman filter algorithm

The Kalman filter algorithm consists of two parts, the *prediction* and *update* parts. In the *prediction* part, a state estimate, $\hat{\mathbf{x}}_k^-$ is calculated using the input state vector and the mathematical model of the system. Furthermore, an predicted error covariance matrix, P_k^- is calculated using the previous error covariance matrix and the process noise.

In the *update* part, the Kalman gain, K, is calculated using the predicted error covariance matrix and the measurement noise. The $\hat{\mathbf{x}}_k^-$ state vector is updated using the error state vector, $\tilde{\mathbf{y}}_k$, and the Kalman gain, K, resulting in the final state estimate, $\hat{\mathbf{x}}_k^+$.

A summary of the formulas of the Kalman filter algorithm can be found below [19]:

Prediction:

Predicted state estimate	$\hat{\mathbf{x}}_k^- = F\hat{\mathbf{x}}_{k-1}^+ + B\mathbf{u}_{k-1}$
Predicted error covariance	$P_k^- = FP_{k-1}^+F^T + Q$

Update:

Measurement residual	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - H\hat{\mathbf{x}}_k^-$
Kalman gain	$K_k = P_k^- H^T(R + HP_k^- H^T)^{-1}$
Updated state estimate	$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k\tilde{\mathbf{y}}_k$
Updated error covariance	$P_k^+ = (I - K_k H)P_k^-$

8.1.3 Extended Kalman filter

The drawback of the Kalman filter is that it only exists of linear algebra, so it is not possible to make accurate state estimations of non-linear systems. In an extended Kalman filter, non-linear process or measurement models are linearized such that they can be used in the original Kalman filter algorithm. In each prediction step, the process model is linearized around the last computed state estimate. In each update step, the measurement model is linearized around the last predicted state estimate. This results in a most optimal linear estimation of the process and measurement models. There are a few drawbacks to the extended Kalman filter, the calculations have a high computational cost and it only gives accurate linear estimates at low non-linearity, because only one point is considered. The extended Kalman filter is described by the following formulas. [19]

Linearization:

Linearize process model	$F_{k-1} = \frac{\partial \boldsymbol{f}}{\partial \mathbf{x}}\vert_{\hat{\mathbf{x}}_{k-1}^+,\mathbf{u}_{k-1}}$
Linearize measurement model	$H_k = \frac{\partial \boldsymbol{h}}{\partial \mathbf{x}}\vert_{\hat{\mathbf{x}}_k^+}$

Prediction:

Predicted state estimate	$\hat{\mathbf{x}}_k^- = \boldsymbol{f}(\hat{\mathbf{x}}_{k-1}^+, \mathbf{u}_{k-1})$
Predicted error covariance	$P_k^- = F_{k-1}P_{k-1}^+F_{k-1}^T + Q$

Update:

Measurement residual	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \boldsymbol{h}(\hat{\mathbf{x}}_k^-)$
Kalman gain	$K_k = P_k^- H_k^T(R + H_kP_k^- H_k^T)^{-1}$
Updated state estimate	$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k\tilde{\mathbf{y}}_k$
Updated error covariance	$P_k^+ = (I - K_k H_k)P_k^-$

8.1.4 Others

Next to the extended Kalman filter, there are more types of Kalman filters improving the performance of working with high non-linear processes. An unscented Kalman filter is a variant of a Kalman filter that does not make an approximation of the non-linear process, but approximates the normal probability density distribution using the original non-linear process. This causes better tracking of high non-linear processes. Finally, a particle filter is similar to the unscented Kalman filter, but is not limited to normal distributions, but can approximate any arbitrary distribution, improving the tracking of non-linear processes even further. [20]

8.1.5 Principals of position and attitude state estimation

In order to perform autonomous flight with a drone, the drone needs to know what its position is. To fly to the next point in the trajectory, the controller compares the current position to the trajectory and commands the drone to fly to the reference position. To calculate your position, you need to know what your velocity is. In order to fly from point A to B, the drone rotates around the pitch- or roll axis, to create a force in the opposite direction you want to fly. To do this, the controller also needs to know the current attitude of the drone. This results in the state vector in Equation 10 used for position and attitude state estimation, where x is the position in meters, q is the attitude in quaternion and v is the velocity in meters per second.

$$\mathbf{x} = \begin{bmatrix} x_x \\ x_y \\ x_z \\ q_w \\ q_i \\ q_j \\ q_k \\ v_x \\ v_y \\ v_z \end{bmatrix} \tag{10}$$

In order to make an estimate of this state, we need to establish our inputs and measurements. In this thesis, we focus on position and attitude state estimation using GPS and IMU. From the GPS and IMU, we get position and attitude measurements and from the IMU we also get the angular frequency and acceleration that is used as input. This results in the input and measurement vectors in Equation 11.

$$\mathbf{u} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ a_x \\ a_y \\ a_z \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} x_x \\ x_y \\ x_z \\ q_w \\ q_i \\ q_j \\ q_k \end{bmatrix} \tag{11}$$

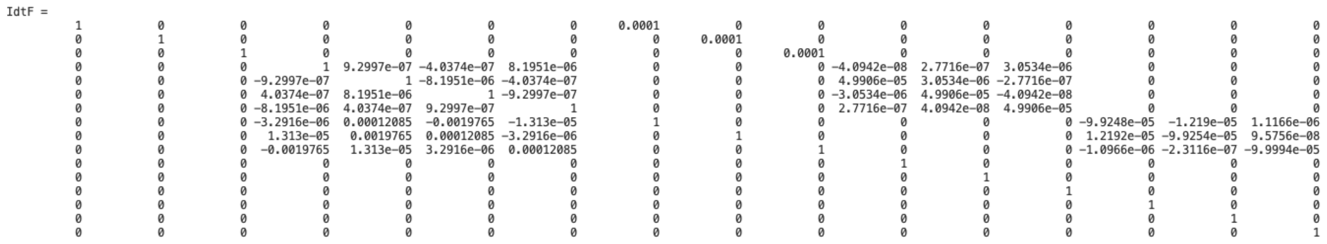
The elements of the input vector are measured in the body frame, while the elements in the state vector are specified in the world frame. This is a problem especially for the velocity prediction from acceleration input, as simple integration needs both in the same frame. To solve this, a rotational matrix from body frame to world frame is calculated from the last attitude estimation. Using this rotational matrix, the acceleration is calculated in the world frame and can be integrated to calculate the velocity.

The prediction of the attitude is a bit different, because quotations cannot be simply integrated. Quaternions are basically the same as a complex number, but with more elements, 1 real and 3 imaginary. With this you can specify a vector in 3D-space indicating the attitude of your drone. A basic complex number can represent a 2D-vector and you can rotate it ϕ radians around the origin by multiplying it with $cos(\phi) + sin(\phi)i$. A similar thing can be done using quaternions, which is called quaternion rotation. You can specify the axis and angle of rotation and rotate the entire quaternion around this. Using this, we can "integrate" the quaternion using the angular frequency. First, the last attitude is fetched and the yaw, pitch or roll axis in the body frame is selected as axis of rotation. Secondly, the angular frequency of the same axis is fetched and multiplied with the timestep, resulting in the angle of rotation. Using quaternion multiplication, the attitude is rotated around the axis of rotation with the calculated angle, resulting in the integration of the angular frequency on the attitude. If you do this for all 3 axis, you calculate the integrated attitude.

Equation 12 shows an example of a prediction step. In this equation, QM stands for quaternion multiplication, which is the integration variant for quaternions. The position is calculated by taking the position of the previous state and integrating the velocity over the timestep. The velocity state is updated by taking the velocity of the previous state and integrating the measured acceleration. The attitude is calculated similarly, take the attitude of the previous state and integrate the measured rotational frequency.

$$\mathbf{x}_k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & dt \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x}_{k-1} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ QM & QM & QM & 0 & 0 & 0 \\ QM & QM & QM & 0 & 0 & 0 \\ QM & QM & QM & 0 & 0 & 0 \\ QM & QM & QM & 0 & 0 & 0 \\ 0 & 0 & 0 & R_x \cdot dt & R_y \cdot dt & R_z \cdot dt \\ 0 & 0 & 0 & R_x \cdot dt & R_y \cdot dt & R_z \cdot dt \\ 0 & 0 & 0 & R_x \cdot dt & R_y \cdot dt & R_z \cdot dt \end{bmatrix} \mathbf{u}_k \qquad (12)$$

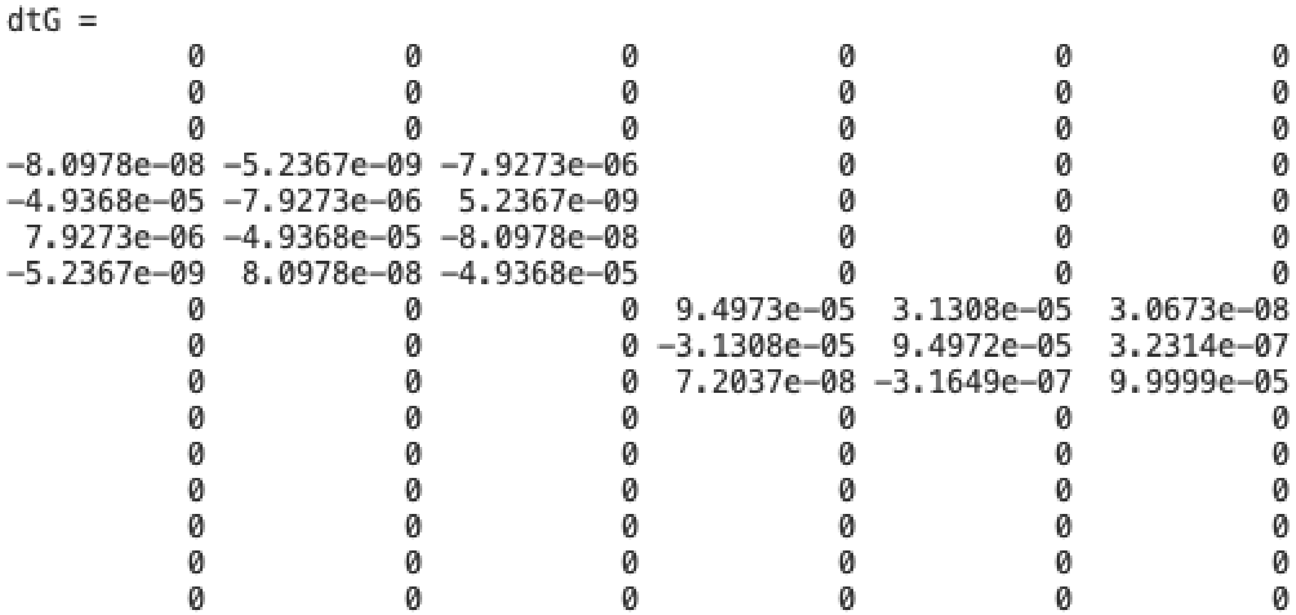
8.2 Appendix A: ***IdtF*** matrix



```
IdtF =
           1           0           0           0           0           0           0      0.0001           0           0           0           0           0           0           0           0
           0           1           0           0           0           0           0           0      0.0001           0           0           0           0           0           0           0
           0           0           1           0           0           0           0           0           0      0.0001           0           0           0           0           0           0
           0           0           0           1  9.2997e-07 -4.0374e-07  8.1951e-06           0           0           0 -4.0942e-08  2.7716e-07  3.0534e-06           0           0           0
           0           0           0 -9.2997e-07           1 -8.1951e-06 -4.0374e-07           0           0           0  4.9906e-05  3.0534e-06 -2.7716e-07           0           0           0
           0           0           0  4.0374e-07  8.1951e-06           1 -9.2997e-07           0           0           0 -3.0534e-06  4.9906e-05 -4.0942e-08           0           0           0
           0           0           0 -8.1951e-06  4.0374e-07  9.2997e-07           1           0           0           0  2.7716e-07  4.0942e-08  4.9906e-05           0           0           0
           0           0           0 -3.2916e-06  0.00012085  -0.0019765   -1.313e-05           1           0           0           0           0           0 -9.9248e-05  -1.219e-05  1.1166e-06
           0           0           0   1.313e-05   0.0019765  0.00012085 -3.2916e-06           0           1           0           0           0           0  1.2192e-05 -9.9254e-05  9.5756e-08
           0           0           0  -0.0019765   1.313e-05  3.2916e-06  0.00012085           0           0           1           0           0           0 -1.0966e-06 -2.3116e-07 -9.9994e-05
           0           0           0           0           0           0           0           0           0           0           1           0           0           0           0           0
           0           0           0           0           0           0           0           0           0           0           0           1           0           0           0           0
           0           0           0           0           0           0           0           0           0           0           0           0           1           0           0           0
           0           0           0           0           0           0           0           0           0           0           0           0           0           1           0           0
           0           0           0           0           0           0           0           0           0           0           0           0           0           0           1           0
           0           0           0           0           0           0           0           0           0           0           0           0           0           0           0           1
```

Figure 51: Instance of *IdtF* matrix.

8.3 Appendix B: ***dtG*** matrix



```
dtG =
           0           0           0           0           0           0
           0           0           0           0           0           0
           0           0           0           0           0           0
 -8.0978e-08 -5.2367e-09 -7.9273e-06           0           0           0
 -4.9368e-05 -7.9273e-06  5.2367e-09           0           0           0
  7.9273e-06 -4.9368e-05 -8.0978e-08           0           0           0
 -5.2367e-09  8.0978e-08 -4.9368e-05           0           0           0
           0           0           0  9.4973e-05  3.1308e-05  3.0673e-08
           0           0           0 -3.1308e-05  9.4972e-05  3.2314e-07
           0           0           0  7.2037e-08 -3.1649e-07  9.9999e-05
           0           0           0           0           0           0
           0           0           0           0           0           0
           0           0           0           0           0           0
           0           0           0           0           0           0
           0           0           0           0           0           0
           0           0           0           0           0           0
```

Figure 52: Instance of *dtG* matrix.

References

[1] G. Loianno, C. Brunner, G. McGrath, and V. Kumar. “Estimation, Control, and Planning for Aggressive Flight With a Small Quadrotor With a Single Camera and IMU”. In: *IEEE Robotics and Automation Letters* (2016).

[2] Philipp Foehn, Elia Kaufmann, Angel Romero, Robert Penicka, Sihao Sun, Leonard Bauersfeld, Thomas Laengle, Giovanni Cioffi, Yunlong Song, Antonio Loquercio, and Davide Scaramuzza. “Agilicious: Open-Source and Open-Hardware Agile Quadrotor for Vision-Based Flight”. In: *AAAS Science Robotics* (2022).

[3] K. Gamagedara, T. Lee, and M. Snyder. “Quadrotor State Estimation With IMU and Delayed Real-Time Kinematic GPS”. In: *IEEE Transactions On Aerospace And Electronic Systems* (2021).

[4] Z. Bodó and Béla Lantos. “State estimation for UAVs using sensor fusion”. In: *IEEE International Symposium on Intelligent Systems and Informatics* (2017).

[5] P. J. Glavine, O. De Silva, G. Mann, and R. Gosine. “GPS Integrated Inertial Navigation System Using Interactive Multiple Model Extended Kalman Filtering”. In: *Moratuwa Engineering Research Conference* (2018).

[6] H. Sahin, B. Gurkan, and V.E. Omurlu. “Sensor Fusion Design by Extended and Unscented Kalman Filter Approaches for Position and Attitude Estimation”. In: *International Congress on Human-Computer Interaction, Optimization and Robotic Applications* (2022).

[7] H. Benzerrouk, A. Nebylov, and H. Salhi. “Quadrotor UAV state estimation based on High-Degree Cubature Kalman filter”. In: *IFAC Symposium on Automatic Control in AerospaceACA* (2016).

[8] R. van der Merwe, E. A. Wan, and S. I. Julier. “Sigma-Point Kalman Filters for Nonlinear Estimation and Sensor-Fusion Applications to Integrated Navigation”. In: (2004).

[9] M. Malleswaran, V. Vaidehi, and M. Mohankumar. “A hybrid approach for GPS/INS integration using Kalman filter and IDNN”. In: *International Conference on Advanced Computing* (2011).

[10] brainfpv. *RADIX product page.* URL: `https://www.brainfpv.com/radix`. (accessed on 24/04/23).

[11] brainfpv. *RADIX LI product page.* URL: `https://www.brainfpv.com/radix-li/`. (accessed on 24/04/23).

[12] drotek. *Sirius RTK GNSS Rover datasheet Rev2.* URL: `https://store-drotek.com/911-sirius-rtk-gnss-rover-f9p.html`.

[13] drotek. *Sirius RTK GNSS Base datasheet Rev1.* URL: `https://store-drotek.com/912-sirius-rtk-gnss-base-f9p.html`.

[14] Holybro. *SiK Telemetry Radio V3 product page.* URL: `https://holybro.com/collections/telemetry-radios/products/sik-telemetry-radio-v3`. (accessed on 25/04/23).

[15] PNI. *RM3100 & RM2100 User Manual R09.* URL: `https://www.pnicorp.com/rm3100/`.

[16] G. Cai, B. M. Chen, and T. H. Lee. *Unmanned Rotorcraft Systems.* Springer, 2011. Chap. 2.

[17] E. Fresk and G. Nikolakopoulos. “Full Quaternion Based Attitude Control for a Quadrotor”. In: (2013).

[18] P. Foehn, A. Romero, and D. Scaramuzza. “Time-optimal planning for quadrotor waypoint flight”. In: *Science Robotics* (2021).

[19] Y. Kim, H. Bang, and F. Govaers (Ed.) “Introduction and Implementations of the Kalman Filter”. In: IntechOpen, 2019. Chap. Introduction to Kalman Filter and Its Applications.

[20] D. Simon. *Optimal State Estimation: Kalman, H∞, and Nonlinear Approaches.* Wiley, 2006.