

Adaptive Extended Kalman Filtering for Navigation State Estimation of Drones and Sounding Rockets

By Sahithyen Kanaganayagam, Luca Moor, Florian Eugster

Bachelor's Thesis

Department of Computer Science

Eastern Switzerland University of Applied Sciences

Advisor OST: Prof. Dr. Andreas Breitenmoser

Co-Examiner: Dr. Arno Wagner

Expert: Amir Shimon Melzer

Advisor ARIS: Carl Fabian Ceccucci

Date: 24.06.2026

eprints version

Abstract

Reliable state estimation is essential for the robust navigation of aerial vehicles. At the Academic Space Initiative Switzerland (ARIS), the "ASTERIA" bi-liquid sounding rocket currently relies primarily on raw GNSS data for state estimation, but this becomes unreliable during high-dynamic flight phases. The Bitcraze Crazyflie 2.1 indoor quadcopter presents a contrasting application case, featuring different flight dynamics and onboard sensors. This thesis examines whether a unified estimation framework can provide reliable state estimates for both systems while adapting to platform-specific characteristics.

We designed and implemented an Extended Kalman Filter (EKF) that operates on both platforms. It is built as a Multiplicative EKF with a nine-dimensional state comprising position, velocity and attitude error. An IMU-driven process model propagates the state. Corrections are provided by interchangeable measurement models for GNSS position, ultra-wideband ranging and time-of-flight distance. The filter is written in Rust and integrated into the firmware of both the ARIS sensor carrier and the Crazyflie. Alongside a static baseline configuration with fixed measurement noise parameters, two adaptive variants were developed that estimate measurement noise online during operation. To validate the filter without physical flight testing, we developed a Python simulation environment based on RocketPy and RotorPy. This environment generates realistic flight trajectories and provides full control over sensor characteristics and data dropouts.

We evaluated the filter using a set of simulated flights with varying trajectories, noise levels and dropout conditions. Across all simulation runs, the Crazyflie achieved an average position RMSE of 0.030 m, while the rocket achieved 15.1 m over the full flight profile. Under nominal conditions with constant sensor noise, the manually tuned static configuration performed best. However, when sensor quality degraded during a flight, the adaptive variants were able to account for this. The modular architecture also facilitated the integration of additional sensors. Adding a time-of-flight sensor reduced the Crazyflie's vertical RMSE from 0.026 m to 0.010 m in simulation and improved performance in real flight tests without requiring changes to the core filter structure. This demonstrates that our architecture enables new sensors to be integrated into both the Rust filter and the simulation. A consistency check revealed that the filter is currently overconfident, primarily due to the absence of IMU bias states in the estimator. To validate deployment on real hardware, the filter was tested on Crazyflie flight experiments and during a ground-driving test using the ARIS avionics stack, confirming successful integration and real-time operation.

Despite their substantially different dynamics and sensor suites, the same EKF architecture can be used for both vehicles. While the adaptive variants showed clear benefits under changing measurement conditions, their performance was confined to the simulation and did not fully transfer to the real hardware. The main limitations that remain are the lack of bias estimation and the absence of an absolute heading reference. Furthermore, the filter has not yet been deployed on ASTERIA and validation during an actual rocket flight remains future work.

Table of Contents

Product Documentation	13
1 Introduction and Requirements	14
1.1 Context	14
1.2 Motivation	14
1.3 Requirements Overview	14
1.4 Stakeholders	17
2 System Context	18
2.1 ARIS Sounding Rocket “ASTERIA”	18
2.2 Crazyflie 2.1 Quadcopter	24
2.3 Platform Comparison	30
3 EKF Design	32
3.1 Design Rationale	32
3.2 Extended Kalman Filter Introduction	32
3.3 Multiplicative Extended Kalman Filter	33
3.4 State Representation and Frame Conventions	34
3.5 Process Models	35
3.6 Measurement Models	35
3.7 Adaptive Measurement Noise	37
3.8 Limitations	37
4 Simulation	40
4.1 Rocket Simulation	40
4.2 Drone Simulation	50
4.3 Simulation Engines Credibility Analysis	56
4.4 Synthetic Sensors	57
4.5 Noise Modeling	59
4.6 Dropout Modeling	60
5 Architecture and Software Design	62
5.1 EKF Rust Library Architecture	62
5.2 Simulation Architecture	68
5.3 Evaluation Architecture	76
6 Software Integration	83
6.1 ARIS Avionics Firmware Integration	83
6.2 Crazyflie Integration	89
6.3 Evaluation Integration	95
7 Results and Evaluation	97
7.1 Evaluation Methodology	97
7.2 Parameter Optimization	100
7.3 Simulation Validation	101
7.4 Adaptive Measurement Noise	109
7.5 Validation with Real Data	114
7.6 Real-Time Performance	123
7.7 Discussion	125
8 Conclusion and Outlook	130
8.1 Conclusion	130
8.2 Outlook	132

Project Documentation	134
9 Assignment	135
9.1 Context	135
9.2 Formalities	135
10 Project Management	136
10.1 Project Resources	136
10.2 Project Plan	136
10.3 Processes, Meetings and Roles	140
10.4 Planning Tools	141
11 Quality Assurance	143
11.1 Organisational Measures	143
11.2 Requirements and Story Definition	143
11.3 Guidelines	143
11.4 Testing Strategy	145
12 Risk Management	146
12.1 Risk Register	146
12.2 Risk Review at Midpoint	148
12.3 Retrospective Risk Review at Project End	150
13 Time Tracking Overview	153
14 Retrospective	155
14.1 Reflection on Organisational Management	155
14.2 Reflection on the Results	156
15. Bibliography	157
 Appendix	 160
I Assignment	161
II Directory of Tools Used	167
III Hardware and Versions	168
III.A Crazyflie	168
III.B ARIS ASTERIA	168
IV Lab Journals	170
IV.A Drone Test Session #1 - Loco Positioning Test	170
IV.B Drone Test Session #2 - Loco Positioning with Flow Deck v2	172
IV.C Drone Test with Rust Library stationary #1	173
IV.D Drone Test with Rust Library stationary #2	175
IV.E Drone Test with Rust Library stationary #3	175
IV.F Drone Test with Rust Library stationary #4	176
IV.G Flight Test with our EKF #1	177
IV.H Flight Test with our EKF #2	180
IV.I Flight Test with our EKF #3	182
IV.J Flight Test with our EKF #4 - ToF Sensor	184
IV.K Flight Test with our EKF #5 - Attitude Fix and ToF Re-Test	187
IV.L Evaluation Flight Session	189
V Architectural Decision Records	190
V.A EKF Rust Library Architecture	190
V.B Simulation Architecture	196
V.C Evaluation Architecture	202
VI Sensor Specifications	206
VI.A ARIS Rocket "Asteria"	206

VI.B Crazyflie 2.1	211
VII Detailed Analysis of the Existing Crazyflie EKF Implementation	214
VII.A EKF System Model for the Crazyflie	214
VII.B EKF flow diagram	216
VII.C EKF functions	216
VII.D Sensor Dropout Handling	220

List of Figures

Figure 1	ASTERIA rocket overview. [1]	18
Figure 2	ARIS avionics stack. [1]	18
Figure 3	C4 system landscape of ASTERIA.	20
Figure 4	GNSS position data recorded during a drop test of the rocket HERMES. The color indicates the phase of the flight.	23
Figure 5	Velocity data provided by the GNSS during the HERMES drop test.	23
Figure 6	g-load data recorded by the accelerometer during the HERMES drop test.	24
Figure 7	Crazyflie 2.1 equipped with a Flow Deck v2 and a Loco Positioning Deck.	24
Figure 8	C4 container diagram of the Crazyflie system.	25
Figure 9	C4 component diagram of the Crazyflie firmware.	26
Figure 10	Reconstructed 3D flight path of a Crazyflie test flight.	28
Figure 11	IMU sensor data recorded during a Crazyflie test flight.	29
Figure 12	Distance to four UWB anchors recorded during a Crazyflie test flight.	29
Figure 13	Distance measured by the ToF sensor during a Crazyflie test flight.	30
Figure 14	General Kalman filter predict-correct cycle.	33
Figure 15	MEKF predict-correct-finalize cycle.	34
Figure 16	RocketPy plot of the configured ASTERIA rocket.	43
Figure 17	Library diagram of the EKF Rust library.	64
Figure 18	C4 system landscape of simulation and evaluation architecture.	68
Figure 19	C4 component diagram of the simulation architecture.	70
Figure 20	UML class diagram of the simulation core domain model.	71
Figure 21	Flow chart showing data flow through the simulation pipeline.	73
Figure 22	Visual validation plots of a representative drone trajectory (right) and a rocket trajectory (left), both showing plausible flight paths.	75
Figure 23	C4 component diagram of the evaluation architecture.	77
Figure 24	UML class diagram of the evaluation domain model.	78
Figure 25	Flow chart showing intended evaluation process.	79
Figure 26	Task diagram in fw-sensor-carrier-v3, from sensor hardware through calibration and state estimation to the SD card log.	85
Figure 27	Flowchart of the rkfTask loop, Rust EKF functions are highlighted in blue.	93
Figure 28	Ground-truth trajectories of the nine simulated rocket flights generated with RocketPy.	99
Figure 29	Ground-truth trajectories of the seven simulated Crazyflie flights generated with RotorPy.	99
Figure 30	True vs. estimated 3D trajectory of the rocket evaluation run, including the GNSS-outage segment.	102
Figure 31	Per-axis position error of the rocket evaluation run.	103
Figure 32	Attitude error of the rocket evaluation run.	103
Figure 33	Per-axis velocity error of the rocket evaluation run.	103
Figure 34	Grouped NEES of the rocket position, velocity and attitude over time, with the expected value for dof = 3.	104
Figure 35	Position error during the ascent phase of the baseline evaluation run.	105
Figure 36	Attitude error during the ascent phase of the baseline evaluation run.	105
Figure 37	Velocity error during the ascent phase of the baseline evaluation run.	105
Figure 38	Position error during the descent phase of the baseline evaluation run.	105
Figure 39	Velocity error during the descent phase of the baseline evaluation run.	105
Figure 40	True vs. estimated 3D trajectory of the Crazyflie evaluation run.	107

Figure 41	Per-axis position error of the Crazyflie evaluation run.	107
Figure 42	Attitude error of the Crazyflie evaluation run.	107
Figure 43	Per-axis velocity error of the Crazyflie evaluation run.	108
Figure 44	Grouped NEES of the crazyflie position, velocity and attitude over time, with the expected value for $\text{dof} = 3$	108
Figure 45	Altitude error of the Crazyflie evaluation run with and without the ToF sensor.	109
Figure 46	Position error over time for the static, sliding window and EMA variants on the ARIS rocket evaluation run.	111
Figure 47	Position error over time for the static, sliding window and EMA variants on the Crazyflie evaluation run.	111
Figure 48	Position error over time for the static parameters and the sliding window and EMA variants (with and without bias compensation) on the Crazyflie evaluation run.	111
Figure 49	Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the changing-variance Crazyflie scenario.	112
Figure 50	Estimated measurement noise standard deviation $\sqrt{R}$ over time for the EMA variant on the changing-variance Crazyflie scenario.	112
Figure 51	Position error over time for the static, sliding window and EMA variants on the changing-variance Crazyflie scenario.	113
Figure 52	Four of the eight UWB anchors mounted on the wall of the test room.	114
Figure 53	3D flight paths of the four different trajectories flown for the Crazyflie evaluation.	115
Figure 54	3D flight path of the figure-8 trajectory. Left: Crazyflie EKF active, our EKF as shadow. Right: our EKF active, Crazyflie EKF as shadow.	116
Figure 55	Per-axis position estimates of our EKF and the Crazyflie EKF over time for the figure-8 trajectory with NIS-tuned parameters, Crazyflie EKF active and our EKF as shadow.	117
Figure 56	3D flight path of the figure-8 trajectory with NIS-tuned parameters. Left: Crazyflie EKF active, our EKF as shadow. Right: Our EKF active, Crazyflie EKF as shadow.	118
Figure 57	Z-axis position estimates of our EKF and the Crazyflie EKF over time for the figure-8 trajectory, with and without the ToF sensor enabled in our EKF.	118
Figure 58	Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the Crazyflie.	119
Figure 59	Housing used to hold the ASTERIA boards in place during the car drives.	120
Figure 60	3D plot of the GNSS path and our EKF path recorded on the sensor carrier board.	121
Figure 61	NED axis position comparison between the GNSS path and our EKF path recorded on the sensor carrier board.	121
Figure 62	Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the ASTERIA sensor carrier board.	123
Figure 63	Measured execution times of our EKF functions on the Crazyflie during a 10-second flight test.	123
Figure 64	Measured execution times of the EKF functions on the ASTERIA sensor carrier board.	124
Figure 65	Project plan showing iterations, milestones and deadlines.	136
Figure 66	Burndown chart showing the remaining estimated effort over the duration of the thesis.	153
Figure 67	Distribution of the total tracked hours across the different activity tags.	153

Figure 68	Overview of the tracked hours broken down by team member.	154
Figure 69	Number of Jira issues per label, distinguishing work attributed to the OST part, the ARIS part and issues contributed to both.	154
Figure 70	UWB anchor A0	170
Figure 71	UWB anchor A1	170
Figure 72	UWB anchor A2	170
Figure 73	UWB anchor A3	170
Figure 74	Visualized flight path from one of the CSV logs	171
Figure 75	Visualized flight path from one of the CSV logs	173
Figure 76	UWB anchor A0--A3	178
Figure 77	Visualized flight path showing both the Crazyflie EKF and our EKF estimates .	179
Figure 78	Visualized flight path of the first flight	181
Figure 79	Visualized flight path of the last flight	182
Figure 80	x position: state vs. shadow	183
Figure 81	y position: state vs. shadow	184
Figure 82	z position: state vs. shadow	184
Figure 83	Flight path with ToF sensor active	185
Figure 84	Pitch: state vs. shadow	186
Figure 85	Roll: state vs. shadow	186
Figure 86	Yaw: state vs. shadow	186
Figure 87	Pitch: state vs. shadow	188
Figure 88	Roll: state vs. shadow	188
Figure 89	Yaw: state vs. shadow	188
Figure 90	z position: state vs. shadow (with ToF)	189
Figure 91	Control flow of the Crazyflie EKF.	216

List of Tables

Table 1	The abbreviations used in this document.	11
Table 2	List of all must-have requirements.	15
Table 3	List of all should-have requirements.	16
Table 4	List of stakeholders, their roles and expectations.	17
Table 5	Sensor overview of the ARIS rocket ASTERIA.	21
Table 6	Sensor overview of the Bitcraze Crazyflie 2.1.	27
Table 7	General platform comparison between ASTERIA and the Crazyflie 2.1.	30
Table 8	Sensor comparison between ASTERIA and the Crazyflie 2.1.	31
Table 9	Required simulation outputs for ground-truth data.	41
Table 10	Analyzed simulation tools for rocket flight.	41
Table 11	Simulation tool comparison for the ARIS rocket.	42
Table 12	ASTERIA airframe parameters used in the RocketPy model.	43
Table 13	ASTERIA motor parameters used in the <code>GenericMotor</code> model.	44
Table 14	EuroC environment variants used across the rocket simulation scenarios.	45
Table 15	Sensors configured in the rocket simulation, including sampling rates and the origin of each signal.	46
Table 16	Noise parameters applied to the rocket simulation sensors.	46
Table 17	Simulation scenarios used in the rocket evaluation.	47
Table 18	Required simulation outputs for Crazyflie ground-truth data.	50
Table 19	Analyzed simulation tools for the Crazyflie.	51
Table 20	Simulation tool comparison for the Crazyflie.	51
Table 21	Key Crazyflie parameters used in the RotorPy model, taken from RotorPy’s built-in <code>crazyflie_params</code> [2].	52
Table 22	Reference trajectories used across the Crazyflie simulation scenarios.	53
Table 23	Sensors configured in the Crazyflie simulation, including sampling rates and the origin of each signal.	54
Table 24	Noise parameters applied to the Crazyflie simulation sensors.	54
Table 25	Simulation scenarios used in the Crazyflie evaluation.	55
Table 26	Parameters of the noise profile that is created per sensor.	60
Table 27	Dropout modes supported by the dropout implementation.	61
Table 28	Overview of the design decisions for the EKF Rust library architecture.	63
Table 29	Measurement implementations provided by the measurements package.	65
Table 30	Overview of the design decisions for the simulation architecture.	69
Table 31	Description of the three core domain model dataclasses.	71
Table 32	Overview of the design decisions for the evaluation architecture.	76
Table 33	Description of the two core output type dataclasses <code>KalmanEstimate</code> and <code>EvaluationResult</code>	78
Table 34	Constructor arguments of the <code>BayesianSearch</code> class.	81
Table 35	The contents of the <code>asteria-firmware</code> repository.	83
Table 36	Binary log record types written to the SD card.	87
Table 37	Available sensor types for the estimator.	90
Table 38	Core C API of the Rust EKF library.	92
Table 39	Search parameters and bounds for the ARIS rocket (50 trials).	100
Table 40	Search parameters and bounds for the Crazyflie (150 trials).	100
Table 41	Selected EKF parameters for the ARIS rocket.	101
Table 42	Selected EKF parameters for the Crazyflie.	101

Table 43	Rocket position, velocity and attitude RMSE per axis and as a combined value, averaged over all simulated runs ($\pm$ standard deviation across runs).	102
Table 44	Rocket position, velocity and attitude RMSE per flight phase, averaged over all simulated runs ($\pm$ standard deviation across runs).	104
Table 45	Crazyflie position, velocity and attitude RMSE per axis and as a combined value, averaged over all simulated runs ($\pm$ standard deviation across runs).	106
Table 46	Adaptive window size per sensor used for the evaluation.	110
Table 47	Rocket RMSE for the static variant and the two adaptive variants (SW and EMA), averaged over all simulated runs.	110
Table 48	Crazyflie RMSE for the static variant and the two adaptive variants (SW and EMA), averaged over all simulated runs.	110
Table 49	RMSE of the static, sliding window and EMA variants on the changing-variance Crazyflie scenario.	113
Table 50	Deviation between the two estimators for the Bayesian-optimized configuration, combined across all four trajectories. Position and velocity are Euclidean distances, attitude is the Euclidean distance of the Euler-angle vector.	116
Table 51	Mean NIS per trajectory type with a UWB standard deviation of 0.18 m and a ToF standard deviation of 0.0055 m.	117
Table 52	Deviation between our EKF and the Crazyflie EKF with the NIS-tuned parameters, combined across all four trajectories.	117
Table 53	Deviation between our EKF and the Crazyflie EKF for the adaptive sliding window and EMA configurations, combined across all four trajectories.	119
Table 54	Deviation between the GNSS measurements and our EKF running the Bayesian-optimized configuration. Position and velocity are Euclidean distances.	122
Table 55	Deviation between the GNSS measurements and the state estimator for the adaptive noise configurations. Position and velocity are Euclidean distances.	122
Table 56	Execution times of our EKF functions on the Crazyflie during a 10-second flight test.	124
Table 57	Execution times of the EKF functions on the ASTERIA sensor carrier board.	125
Table 58	Project milestones and their respective goals and reviews.	137
Table 59	Overview of project deliverables.	140
Table 60	Overview of recurring project meetings.	140
Table 61	Assignment of technical leads to the three core areas.	141
Table 62	Used Jira ticket-types.	141
Table 63	Risk register along with the mitigation measures.	146
Table 64	Updated risk ratings and status at the project midpoint, relative to the initial assessment in Table 63	148
Table 65	Final risk ratings and status at project end, relative to the initial assessment in Table 63 and the midpoint review in Table 64	150
Table 66	Directory of Tools Used in this project	167
Table 67	Crazyflie hardware components	168
Table 68	Crazyflie software versions	168
Table 69	ARIS ASTERIA hardware components	168
Table 70	ARIS ASTERIA software versions	169
Table 71	Key specifications of the LSM6DSO32 (measured at VDD = 1.8 V, T = 25 °C) [3]	206
Table 72	Key specifications of the MS5607 (measured at VDD = 3 V, T = 25°C) [4]	207
Table 73	Key specifications of the LSM303AGR (measured at VDD = 2.5 V, T = 25 °C) [5]	208

Table 74	Key specifications of the SHT4 [6]	209
Table 75	Key specifications of the ZED-F9P-05B [7]	210
Table 76	Key specifications of the BMI088 [8]	211
Table 77	Key specifications of the BMP388 [9]	212
Table 78	Key specifications of the PMW3901 [10]	212
Table 79	Key specifications of the VL53L1x [11]	213
Table 80	Key specifications of the DWM1000 [12]	213
Table 81	Measurement model files and their calls to <code>kalmanCoreScalarUpdate()</code>	220

List of Code-Blocks

Code Snippet 1	Synthetic ToF sensor generation.	58
Code Snippet 2	Synthetic UWB transceiver sensor generation.	59
Code Snippet 3	Implementation of an adaptive measurement type.	65
Code Snippet 4	Usage example of the SyntheticSensorGenerator class.	73
Code Snippet 5	Usage example of the NoiseInjector class.	74
Code Snippet 6	Usage example of the DropoutInjector class.	74
Code Snippet 7	Usage example of the NearestTimeIndexAligner class.	80
Code Snippet 8	Usage example of class BayesianSearch, optimizing for position RMSE. .	82
Code Snippet 9	StateEstimate struct published to STATE_ESTIMATE_WATCH after each EKF predict step.	86
Code Snippet 10	Cargo configuration for the ARM Cortex-M7 with double-precision FPU. .	88
Code Snippet 11	Crazyflie Makefile additions for the Rust EKF library.	91
Code Snippet 12	RkfState struct exposed across the C boundary	92
Code Snippet 13	Simplified PyO3 binding exposing a Rust function to Python.	95
Code Snippet 14	Usage of exported Rust function in Python.	95
Code Snippet 15	Code showing an example of an IMU predict EKF call made from an evaluation runner.	96
Code Snippet 16	Crazyflie EKF state vector.	214
Code Snippet 17	Crazyflie EKF velocity update equations for the airborne case.	215
Code Snippet 18	Crazyflie EKF position update rotating body-frame displacement into the global frame.	218
Code Snippet 19	Crazyflie EKF process noise terms added to the covariance diagonal. .	218
Code Snippet 20	Crazyflie EKF Kalman gain computation.	219
Code Snippet 21	Crazyflie EKF state update applying the Kalman gain.	219
Code Snippet 22	Crazyflie EKF covariance update using the Joseph form.	219

Glossary

Abbr.	Stands for
ADR	Architectural Decision Record
CEP	Circular Error Probable
DOF	Degrees of Freedom
DPS	Degrees Per Second
DWNA	Discrete White-Noise Acceleration
DWT	Data Watchpoint and Trace
EKF	Extended Kalman Filter
EMA	Exponential Moving Average
ENU	East North Up
EuRoC	European Rocketry Challenge
FAT	File Allocation Table
FIFO	First In, First Out
FPU	Floating Point Unit
GNSS	Global Navigation Satellite System
HIL	hardware-in-the-loop
I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
MEKF	Multiplicative Extended Kalman Filter
NED	North East Down
NEES	Normalized Estimation Error Squared
NIS	Normalized Innovation Squared
NVIC	Nested Vectored Interrupt Controller
ODR	Output data rate
RH	Relative humidity
RKF	Rust Extended Kalman Filter
RMS	Root mean square
RMSE	Root Mean Squared Error

Abbr.	Stands for
SITL	Software-In-The-Loop
SPI	Serial Peripheral Interface
TPE	Tree-structured Parzen Estimator
TWR	Two-Way Ranging
ToF	Time-of-Flight
UART	Universal Asynchronous Receiver Transmitter
UKF	Unscented Kalman Filter
UWB	Ultra-Wideband

Table 1: The abbreviations used in this document.

Product Documentation

1 Introduction and Requirements

This chapter introduces the problem this thesis addresses and defines its objectives. [Section 1.1](#) outlines the setting and the two flying platforms. [Section 1.2](#) explains why a dedicated navigation state estimator is needed and states the central research question. [Section 1.3](#) derives the concrete project requirements based on the thesis task. Finally, [Section 1.4](#) lists the stakeholders we identified.

1.1 Context

This thesis develops an Extended Kalman Filter (EKF) for navigation state estimation on two different aerial platforms: The ARIS sounding rocket “ASTERIA”, a student-built bi-liquid sounding rocket competing in the European Rocketry Challenge (EuRoC) and the Bitcraze Crazyflie 2.1, an open-source miniature quadcopter widely used in research. Both platforms are described in detail in [Section 2](#).

Both platforms require a continuous, robust estimate of their navigation state but are equipped with different sensor sets and operate under different flight dynamics. Rather than designing a separate estimator for each vehicle, this thesis treats the EKF as a software architecture problem. The goal is a single, modular estimator that can be adapted to separate platforms by configuring its sensors and parameters, without rewriting the filter itself.

This thesis focuses on designing, implementing and validating a concrete EKF implementation in Rust. The complete set of requirements is listed in [Section 1.3](#) and the original task definition is provided in [Section 9](#).

1.2 Motivation

On the ARIS rocket, navigation state estimation currently relies on raw GNSS data. During high-dynamic flight phases, GNSS data becomes unreliable, which results in a poor state estimate. Guided descent and post-flight analysis depend on accurate, continuous state estimates that the current approach cannot always reliably provide.

An EKF addresses these limitations by fusing data from all available sensors. Combining different measurements lets the filter bridge GNSS outages and produce a more reliable state estimate. The EKF is a well-established method for real-time state estimation [\[13\]](#), which makes it a suitable choice for this thesis.

The Crazyflie quadcopter serves as a second platform with different dynamics, used to test whether a shared EKF design generalizes across vehicles. This leads to the central research question of this thesis: Is it possible to create a single EKF-based estimator that can adapt, potentially automatically, to inertial measurement unit (IMU)-driven aerial platforms operating under different flight dynamics? This investigation is supported by a simulation environment that provides ground-truth data against which the filter can be validated.

1.3 Requirements Overview

This section defines the estimated project scope by categorizing requirements into must-have and should-have items. We derived the requirements from the task assignment in [Section 9](#).

1.3.1 Must-Have Requirements

[Table 2](#) describes all must-have requirements we identified for this thesis. The status indicates whether it was achieved (✓), partially achieved (~) or not achieved (✗).

ID	Status	Title	Description
MHR-01	✓	Multi-Sensor Fusion	Data from different sensors shall be fused into a combined navigation state estimate (position and velocity) using an Extended Kalman Filter.
MHR-02	✓	Coordinate Frame Transformations	The EKF shall define and implement all required coordinate frame transformations.
MHR-03	✓	Sensor Dropout Handling	The EKF shall remain operational when individual sensors are unavailable and the topic of invalid sensor data is addressed, using the remaining functional sensors for state estimation.
MHR-04	✓	Real-Time State Estimation	Navigation state estimates shall be produced in real time.
MHR-05	✓	Asynchronous Sensor Handling	A strategy shall be developed to handle sensor measurements arriving at different sampling rates.
MHR-06	✓	Simulation Environment	A simulation environment shall be used to generate realistic sensor data for offline testing.
MHR-07	✓	Filter Behavior Analysis	An analysis of the filter behavior during the different flight phases and maneuvers shall be performed.
MHR-08	~	Offline Validation	The EKF shall be validated offline using recorded real flight data as well as simulated data.
MHR-09	✓	Cross-Platform Performance Comparison	The filter behavior shall be evaluated and compared across all implemented platforms using consistent metrics such as estimation error, innovation statistics and robustness during sensor outages.
MHR-10	✓	ARIS Firmware Integration	The EKF implementation shall integrate into the existing ARIS sensor carrier firmware without requiring modifications to low-level drivers.
MHR-11	✓	Crazyflie Firmware Integration	The EKF shall be integrated into the Crazyflie firmware.
MHR-12	✓	Implementation Language (Rocket)	The EKF implementation for the ARIS rocket shall be written in Rust.

ID	Status	Title	Description
MHR-13	✓	Implementation Language (Crazyflie)	The EKF adaptation for the Crazyflie platform shall be implemented in Python and/or Rust or C/C++.
MHR-14	✓	Code Quality	All implemented code shall follow established language-specific conventions, pass static code analysis without errors and include appropriate test cases (e.g. numerical stability).
MHR-15	✓	Numerical Stability	The EKF implementation shall remain numerically stable under high accelerations, sensor noise and temporary measurement loss throughout all flight phases.

Table 2: List of all must-have requirements.

1.3.2 Should-Have Requirements

[Table 3](#) describes all should-have requirements we identified for this thesis. The status indicates whether it was achieved (✓), partially achieved (⚡) or not achieved (✗).

ID	Status	Title	Description
SHR-01	⚡	Adaptive Noise Tuning	The EKF should automatically adapt its measurement and process noise parameters to improve estimation quality across different platforms.
SHR-02	✗	Third Platform Support	The EKF should be validated on a third simulated platform (e.g. a balloon) to demonstrate cross-platform generalizability.
SHR-03	✗	Hardware-in-the-Loop Validation	A Hardware-in-the-Loop (HIL) setup should be established in which the EKF runs on the embedded target hardware while the rest of the system is simulated, to validate real-time behavior on actual hardware.

Table 3: List of all should-have requirements.

1.4 Stakeholders

[Table 4](#) lists the stakeholders that we identified for this project, along with their roles and expectations.

Stakeholder	Role and Expectations
Prof. Dr. Andreas Breitenmoser	Thesis advisor at OST. Expects a well-structured project with regular status reviews as well as a clear and well-written document. Provides guidance on documentation and implementation, as well as feedback on the written thesis.
ARIS Team	Provides existing hard- and software for the ASTERIA rocket. Expects a functional EKF implementation that is integrated into the existing ARIS firmware. Provides support and guidance for integration and testing on the ASTERIA rocket.
Dr. Arno Wagner	Proofreader of the thesis. Provides feedback on the thesis, will attend the final presentation and the bachelor examination.
Amir Shimon Melzer	Expert of the thesis. Will conduct an additional evaluation of the thesis and provide feedback. Will also attend the final presentation and the bachelor examination.

Table 4: List of stakeholders, their roles and expectations.

2 System Context

This chapter describes the two primary target platforms of the Extended Kalman Filter (EKF) developed during this thesis. [Section 2.1](#) introduces the ARIS (Akademische Raumfahrt Initiative Schweiz) sounding rocket ASTERIA and its operational context. [Section 2.2](#) introduces the second platform, the Bitcraze Crazyflie 2.1 quadcopter. Finally, [Section 2.3](#) concludes with a comparison between the two systems and highlights the key differences in dynamics and sensor characteristics.

2.1 ARIS Sounding Rocket “ASTERIA”

ASTERIA is a student-built bi-liquid sounding rocket developed by ARIS (Akademische Raumfahrt Initiative Schweiz), a student-led aerospace association founded in 2017 at ETH Zürich [\[1\]](#). Its primary mission is to compete at the European Rocketry Challenge (EuRoC) in the Liquid 30000 ft category, targeting an apogee of approximately 9 km [\[1\]](#). During the boost phase, the rocket reaches a maximum velocity of approximately 353 m/s ($\sim$ Mach 1.1) and a maximum acceleration of approximately 40 m/s² [\[14\]](#).

[Figure 1](#) shows an overview of the components present on the ASTERIA rocket.

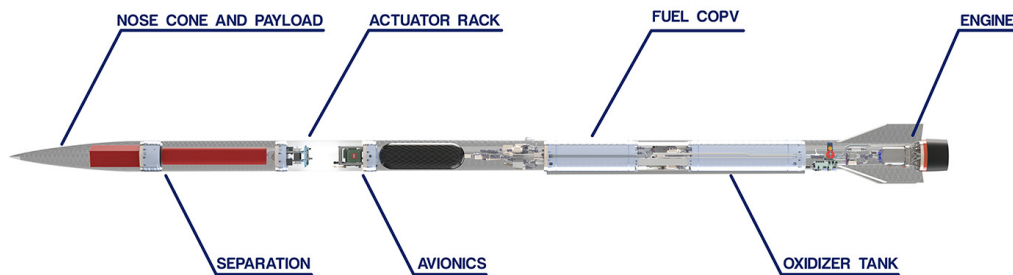


Figure 1: ASTERIA rocket overview. [\[1\]](#)

The rocket is powered by a regeneratively-cooled bi-liquid engine burning liquid oxygen (LOX) and Ethanol [\[15\]](#). ASTERIA follows a modular structural design, divided into distinct sections including the nosecone, recovery, avionics, pressurant bay, midbay, Ethanol tank, LOX tank and engine section [\[14\]](#).

The avionics section displayed in [Figure 2](#) houses a modular sensor carrier board and a flight computer responsible for all mission-critical software. The sensor suite includes redundant inertial sensors, magnetometers, GNSS receivers and barometric sensors operating at different sampling rates [\[14\]](#).

The recovery system features a two-stage parachute deployment: A drogue chute at apogee to stabilize descent, followed by a steerable ram-air main parachute controlled by two lines, which guides the rocket to a predetermined landing zone [\[15\]](#).

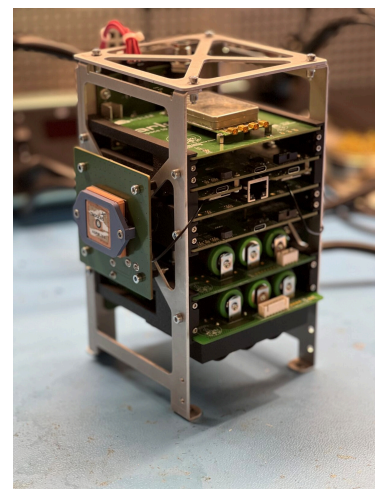


Figure 2: ARIS avionics stack. [\[1\]](#)

2.1.1 Avionics Architecture

[Figure 3](#) illustrates the ASTERIA rocket system landscape. The rocket consists of multiple subsystem boards that communicate with each other via CAN and RS-485. The `Sensor Carrier Board` continuously publishes its state estimation to the `Flight Computer` via the CAN bus. The flight computer relies on this estimate for all mission-critical decisions, including state transitions such as apogee detection and recovery system deployment, as well as control commands directed at the `Auxiliary Boards`. Inaccurate or delayed state estimates can cause premature or missed parachute deployment, or prevent the steerable main parachute from guiding the rocket to its predetermined landing zone. This makes the quality of the state estimation critical to safety.

The `Communication Board` connects the internal CAN bus to the ground station. It relays live flight data from the `Flight Computer` to `Mission Control` via MAVLink, including the current state estimation. The `Controller` monitors this stream in real time to assess mission progress and execute manual mission steps when required.

After the mission, `Mission Control` uploads the recorded flight data to the `Log database`. The `Mission analyst` then retrieves this data to reconstruct the flight trajectory and compile the post-flight report. Therefore, the fidelity of this reconstruction and the completeness of the report depend directly on the quality of the state estimation produced during flight.

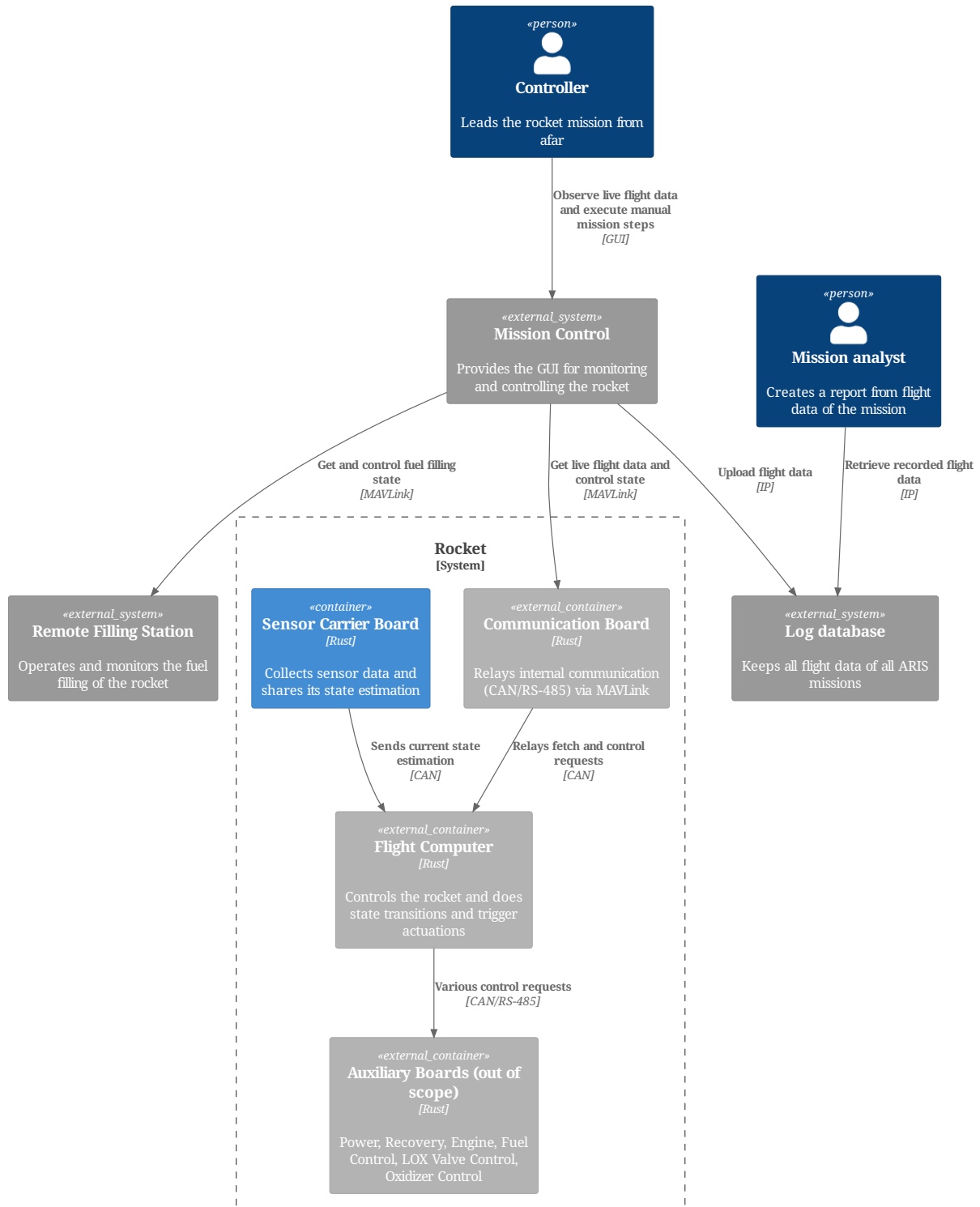


Figure 3: C4 system landscape of ASTERIA.

2.1.2 Sensor Overview ARIS Rocket “ASTERIA”

The avionic stack of the ARIS rocket “ASTERIA” holds the sensors listed in [Table 5](#). Each sensor listed is available twice on the sensor carrier board for redundancy.

Sensor	Signal	Max ODR	Unit	Accuracy
LSM6DSO32	Acceleration	6664 Hz	mg	± 20 mg
	Angular rate	6664 Hz	dps	± 0.5 dps
MS5607	Pressure	122 Hz	mbar	± 1.5 mbar
LSM303AGR	Acceleration	5376 Hz	mg	± 3 mg RMS
	Magnetic field	100 Hz	mgauss	± 3 mgauss RMS
SHT4	Humidity	770 Hz	%RH	± 1.8 %RH
	Temperature	770 Hz	°C	± 0.2 °C
ZED-F9P-05B	Position	25 Hz	m	1.5 m CEP
	Velocity	25 Hz	m/s	0.05 m/s

Table 5: Sensor overview of the ARIS rocket ASTERIA.

The following sections briefly describe each sensor. Full specifications are provided in [Section VI.A](#).

2.1.2.1 LSM6DSO32 - IMU (Accelerometer + Gyroscope)

The LSM6DSO32 is a system-in-package by STMicroelectronics featuring a 3D digital accelerometer and a 3D digital gyroscope. The accelerometer outputs a three-axis linear acceleration vector, while the gyroscope outputs a three-axis angular rate vector in degrees per second. The sensor also includes an embedded temperature sensor and a 9 KB FIFO buffer for data batching.

Two LSM6DSO32 units are installed on the sensor carrier board, each connected to a separate SPI bus for redundancy. The sensor carrier board currently uses a sampling rate of 833 Hz. Full specifications that we derived from the datasheet [\[3\]](#) are listed in [Table 71](#).

2.1.2.2 MS5607 - Altimeter Module

The MS5607 is a high-resolution altimeter sensor by Measurement Specialties (TE Connectivity). The sensor provides a 24-bit pressure and temperature value. From the measured pressure, altitude can be derived using the barometric formula.

Two MS5607 units are installed on the sensor carrier board, each connected to a separate I2C bus for redundancy. The sensor carrier board currently uses a sampling rate of 40 Hz. Full specifications that we derived from the datasheet [\[4\]](#) are listed in [Table 72](#).

2.1.2.3 LSM303AGR - eCompass (Magnetometer and Accelerometer)

The LSM303AGR is a system-in-package by STMicroelectronics containing a 3-axis digital linear acceleration sensor and a 3-axis digital magnetic sensor. The magnetometer outputs a magnetic field vector and the accelerometer outputs a linear acceleration vector.

Two LSM303AGR units are installed on the sensor carrier board, each connected to a separate I2C bus for redundancy. The sensor carrier board currently uses a sampling rate of 10 Hz. Full specifications that we derived from the datasheet [\[5\]](#) are listed in [Table 73](#).

2.1.2.4 SHT4 - Humidity and Temperature

The SHT4 is a digital sensor platform for measuring relative humidity and temperature by Sensirion. It provides humidity and temperature readings via an I2C interface.

Two SHT4 units are installed on the sensor carrier board, each connected to a separate I2C bus for redundancy. Full specifications that we derived from the datasheet [6] are listed in [Table 74](#).

2.1.2.5 ZED-F9P-05B - GNSS

The ZED-F9P-05B is a high precision multi-band GNSS module by u-blox. The sensor offers a high update rate as well as fast convergence. It provides reception of GPS, GLONASS, Galileo and BeiDou constellations.

Two ZED-F9P-05B units are installed on the sensor carrier board, each connected to a separate UART bus for redundancy. Full specifications that we derived from the datasheet [7] are listed in [Table 75](#).

The ZED-F9P-05B supports different dynamic platform models to adjust the navigation engine to the expected application environment. These models offer a trade-off between maximum altitude, maximum velocity and the position deviation. The operational limits listed above are specified using the Airborne <4g dynamic platform model [7]. By default, the module ships configured in Portable mode [16], which has lower altitude and velocity limits but improves the maximum position deviation.

2.1.3 Past Flight Data Analysis

By the time of this thesis, the ASTERIA rocket has not yet completed a full flight. Its predecessor, the ARIS rocket “HERMES”, also did not complete a full flight but was part of several drop tests during its development phase. The earlier rocket “NICOLIER” was successfully launched twice in 2024.

The flight data recorded during the two NICOLIER launches is not directly accessible. The data is stored in a binary format and the configuration of NICOLIER differs substantially from the current ASTERIA stack. Reconstructing the data would require considerable effort and we therefore did not pursue it.

The drop test data from HERMES is accessible and was reviewed as part of this thesis. However, drop tests only cover a portion of the descent phase under a parachute and therefore do not capture the full dynamics of the flight. The HERMES avionics configuration also differs from ASTERIA, which limits how directly the recorded sensor characteristics transfer.

For these reasons, we treated the available flight data primarily as reference and to develop an intuition for the sensor characteristics and flight dynamics, rather than as a direct source for validation or simulation.

2.1.3.1 HERMES Drop Test

[Figure 4](#) shows the GNSS position data recorded during a drop test of the HERMES rocket, including the ascent by helicopter. It was recorded on June 17, 2025 in Bern. The rocket was dropped from a helicopter at an altitude of approximately 1 km. The flight path is colored according to the phase of the flight. The trajectory confirms that the GNSS data is available during the entire descent phase under the parachute.

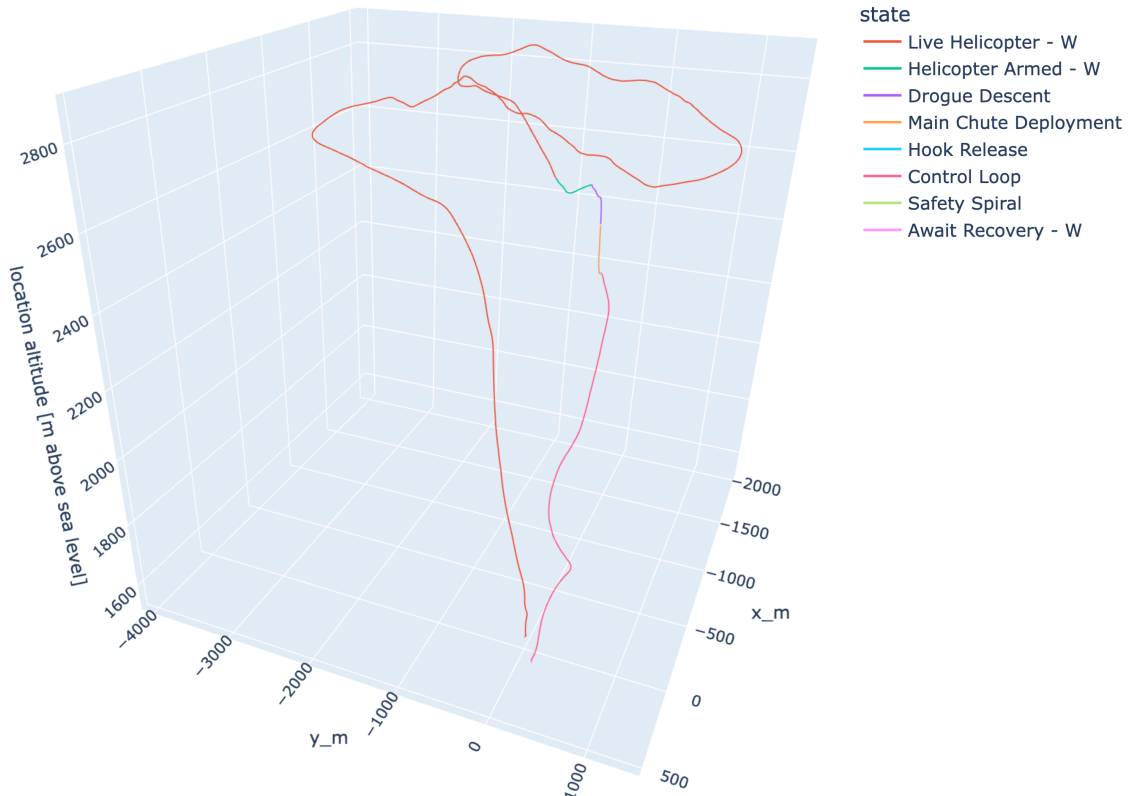


Figure 4: GNSS position data recorded during a drop test of the rocket HERMES. The color indicates the phase of the flight.

[Figure 5](#) shows the velocity data provided by the GNSS during the same drop test. The data shows that during free fall, the descent velocity increases to approximately 34 m/s before the main parachute deploys. After the deployment, the velocity stabilizes at around 12 m/s until touchdown.

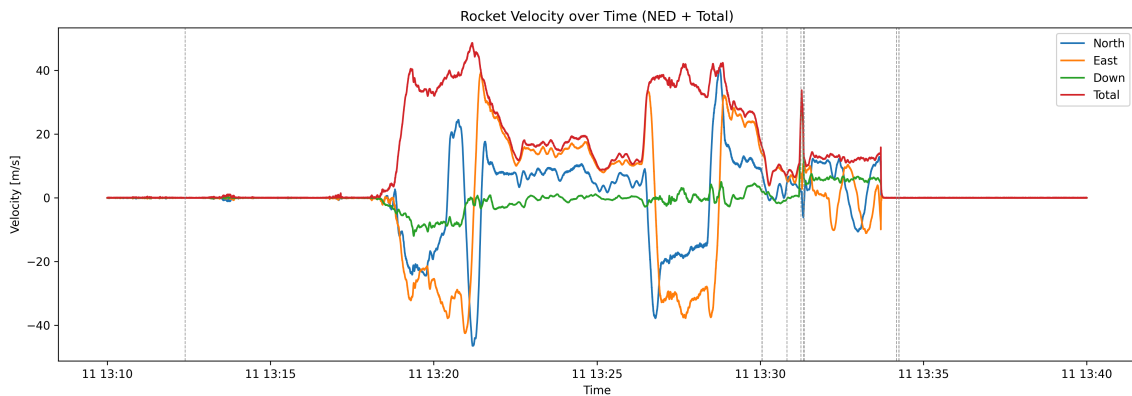


Figure 5: Velocity data provided by the GNSS during the HERMES drop test.

[Figure 6](#) shows the g-load data recorded by the accelerometer during the same drop test. Two large spikes can be observed. The first spike occurs right at the main parachute deployment, where the rocket experiences a g-load of approximately 7.82 g. The second spike occurs at touchdown with a g-load of approximately 6.78 g.

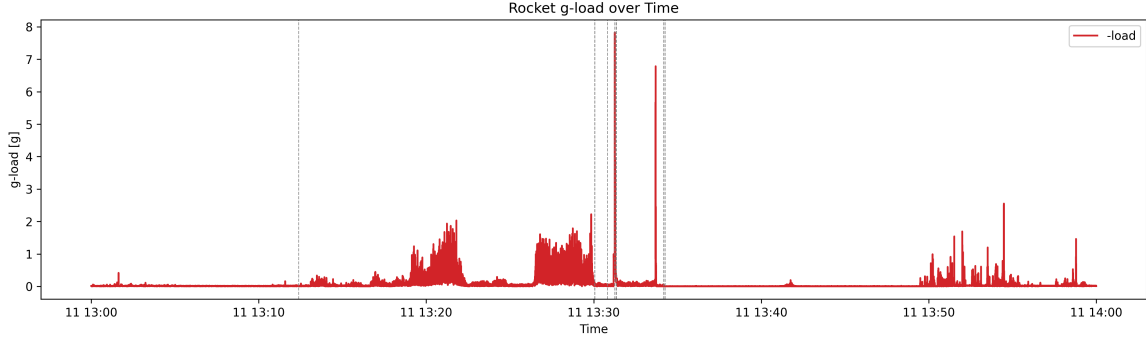


Figure 6: g-load data recorded by the accelerometer during the HERMES drop test.

The HERMES drop test illustrates the behavior of the sensor suite during the descent phase. These observations are limited to the descent phase and to the HERMES hardware configuration, but they confirm that the sensors produce usable data during descent dynamics. Because ASTERIA mainly uses the same sensor types, we used these observations as a baseline expectation for the rocket simulation in [Section 4.1](#).

2.2 Crazyflie 2.1 Quadcopter

The Crazyflie 2.1 is an open-source flying development platform by Bitcraze [\[17\]](#). Both the firmware source code and the hardware schematics are publicly available which makes the platform well-documented and widely adopted in academic research [\[18\]](#). [Figure 7](#) shows the Crazyflie 2.1 equipped with a Flow Deck v2 and a Loco Positioning Deck.



Figure 7: Crazyflie 2.1 equipped with a Flow Deck v2 and a Loco Positioning Deck.

With a weight of 29 g and dimensions of $92 \times 92 \times 29$ mm, it is one of the smallest commercially available quadcopters [\[19\]](#). According to the datasheet, the platform achieves a flight time of approximately seven minutes on a single battery charge [\[19\]](#).

The Crazyflie follows a modular hardware design. Through the use of decks, which are small circuit boards that attach to the main board, the capabilities of the quadcopter can be extended. Examples include the Flow Deck v2 for optical flow and height sensing, the Loco Positioning Deck for Ultra-Wideband-based positioning and the AI Deck for onboard neural network computation capabilities [\[20\]](#). This modularity allows researchers to equip the sensors required for their specific application.

2.2.1 System Architecture

This section describes the architecture of the Crazyflie system using the C4 model, progressing from a high-level system overview down to the internal firmware structure.

2.2.1.1 System Context and Containers

[Figure 8](#) shows the Crazyflie system at the container level. All PC-side communication with the drone runs through cflib, the official Bitcraze Python library, which sends and receives CRTP packets over a 2.4 GHz radio link via a Crazyradio PA USB dongle. cfclient is a graphical

application built on top of cflib for manual control, while autopilot programmers use cflib directly in Python scripts. On the drone side, the nRF51 coprocessor handles the radio stack, battery management and LED control, forwarding CRTP packets to the main STM32 firmware over an internal Syslink bus. The Loco Positioning Deck communicates with a set of Ultra-Wideband (UWB) anchors fixed in the room, which emit ranging signals that the firmware uses to estimate the position. Our EKF is integrated directly into the Crazyflie Firmware container.

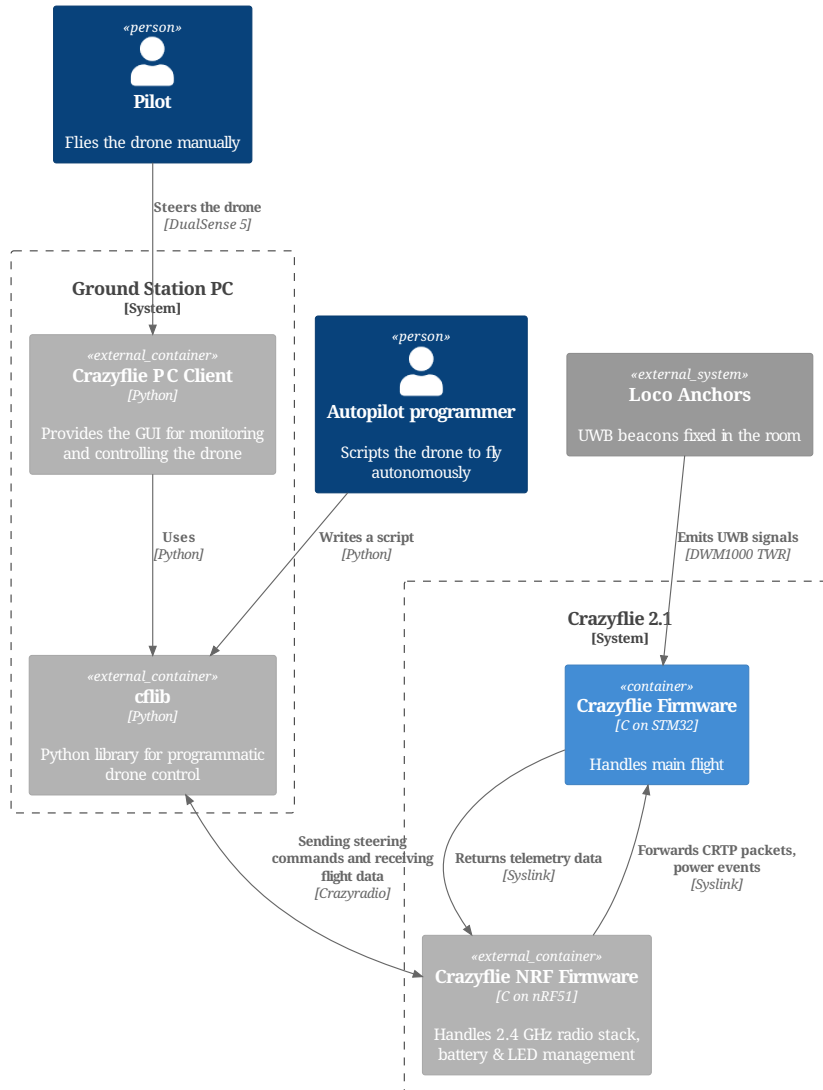


Figure 8: C4 container diagram of the Crazyflie system.

2.2.1.2 Crazyflie Firmware Components

[Figure 9](#) illustrates the internal structure of the Crazyflie firmware at the component level. The `Stabilizer` orchestrates the 1 kHz flight loop, triggering in sequence: `Sensors`, `State Estimator`, `Supervisor`, `Commander`, `Controller` and `Power Distribution` each tick.

Our EKF is embedded inside the `State Estimator`, which acts as its sole interface to the rest of the firmware. The `Flow Deck driver` and `Loco Deck driver` each push their measurements into the `State Estimator`, which then feeds the aggregated sensor data into the EKF and retrieves the updated pose estimate. Telemetry is streamed back to the ground station via the `Log/Param` component.

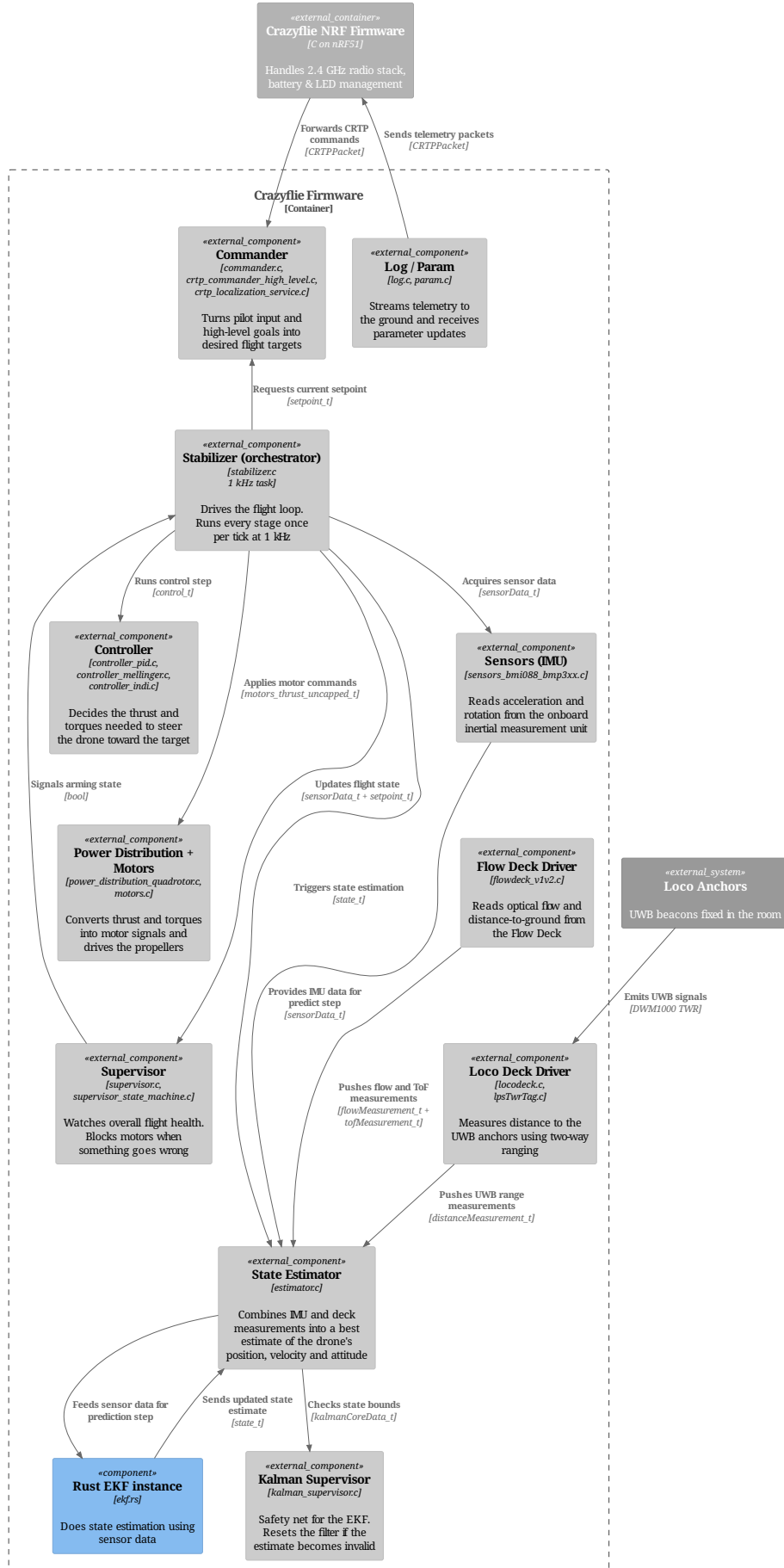


Figure 9: C4 component diagram of the Crazyflie firmware.

2.2.2 Sensor Overview Crazyflie

The Crazyflie 2.1 is equipped with onboard sensors on its main board and can be extended with the Flow Deck v2 and the Loco Positioning Deck expansion decks. [Table 6](#) lists all sensors on the Crazyflie's main board and both decks.

Sensor	Signal	Max ODR	Unit	Accuracy	Location
BMI088	Acceleration	1600 Hz	mg	± 20 mg	Main board
	Angular rate	2000 Hz	dps	± 1 dps	Main board
BMP388	Pressure	200 Hz	hPa	± 0.50 hPa	Main board
PMW3901	Optical flow	n.g.	counts/frame	n.g.	Flow Deck v2
VL53L1x	Distance	50 Hz	mm	± 20 mm	Flow Deck v2
DWM1000	UWB range	n.g.	cm	± 10 cm	Loco Positioning Deck

Table 6: Sensor overview of the Bitcraze Crazyflie 2.1.

The following sections briefly describe each sensor. Full specifications are provided in [Section VI.B](#).

2.2.2.1 BMI088 - IMU (Accelerometer and Gyroscope)

The BMI088 is a 6-axis inertial measurement unit by Bosch Sensortec featuring a 3-axis accelerometer and a 3-axis gyroscope. It is mounted directly on the Crazyflie 2.1 main board. Full specifications that we derived from the datasheet [\[8\]](#) are listed in [Table 76](#).

2.2.2.2 BMP388 - Barometric Pressure Sensor

The BMP388 is a digital barometric pressure and temperature sensor by Bosch Sensortec. It is mounted directly on the Crazyflie 2.1 main board and is used to derive altitude via the barometric formula. Full specifications that we derived from the datasheet [\[9\]](#) are listed in [Table 77](#).

2.2.2.3 PMW3901 - Optical Flow Sensor

The PMW3901 is a low-power optical flow sensor by PixArt Imaging, mounted on the Flow Deck v2. It tracks surface features below the platform and outputs a two-dimensional motion vector in counts per frame, which the flight controller integrates to estimate horizontal velocity. Full specifications that we derived from the datasheet [\[10\]](#) are listed in [Table 78](#).

2.2.2.4 VL53L1x - Time-of-Flight Distance Sensor

The VL53L1x is a Time-of-Flight (ToF) sensor by STMicroelectronics based on a 940 nm VCSEL laser, mounted on the Flow Deck v2. It measures the distance to the surface directly below the platform and provides the height estimate used to scale the optical flow output. Full specifications that we derived from the datasheet [\[11\]](#) are listed in [Table 79](#).

2.2.2.5 DWM1000 - UWB Ranging Sensor (Loco Positioning Deck)

The DWM1000 is a Ultra-Wideband (UWB) sensor by Qorvo, mounted on the Loco Positioning Deck. The deck communicates with a set of fixed anchor nodes placed in the environment. The system supports Two-Way Ranging (TWR) and Time Difference of Arrival (TDoA) operating modes. Full specifications that we derived from the datasheet [\[12\]](#) are listed in [Table 80](#).

2.2.3 Initial Flight Data Analysis

To build an understanding of the sensor characteristics of the Crazyflie platform, initial test flights were conducted using the standard Crazyflie firmware without any modifications. The drone was equipped with the Flow Deck v2 and the Loco Positioning Deck and flown manually in an indoor environment with four UWB anchors placed around the room. The primary goal of these flights was to collect raw sensor data from the onboard IMU, the UWB sensor and the ToF sensor.

2.2.3.1 3D Flight Path

[Figure 10](#) shows the reconstructed 3D flight path of one test flight, using the position estimates computed by the Crazyflie EKF.

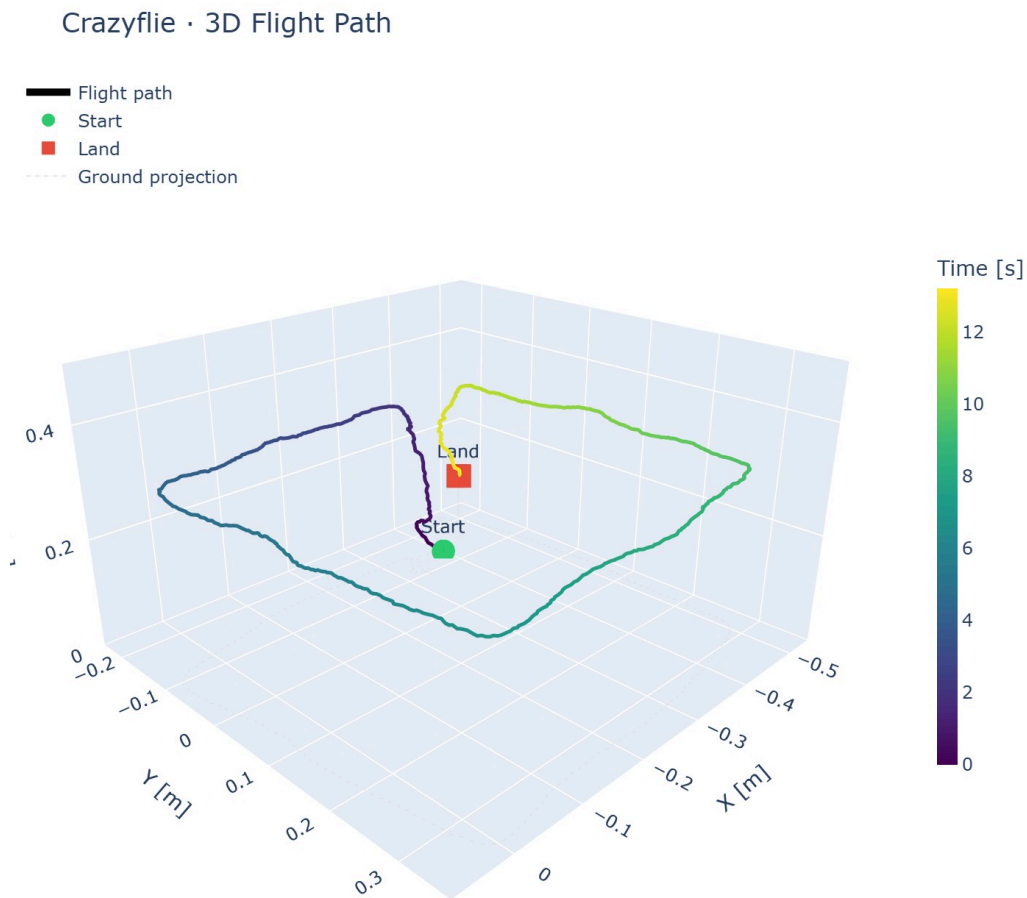


Figure 10: Reconstructed 3D flight path of a Crazyflie test flight.

2.2.3.2 IMU Sensor Data

[Figure 11](#) shows the BMI088 accelerometer and gyroscope readings during the flight. The Z-axis acceleration stays near 1 g throughout, as the accelerometer measures the reaction force of gravity. The X and Y axes remain close to zero because the drone flies at low speed, requiring only a small tilt angle. The gyroscope shows short spikes during attitude corrections at the start, end and direction changes of the flight and settles near zero between maneuvers.

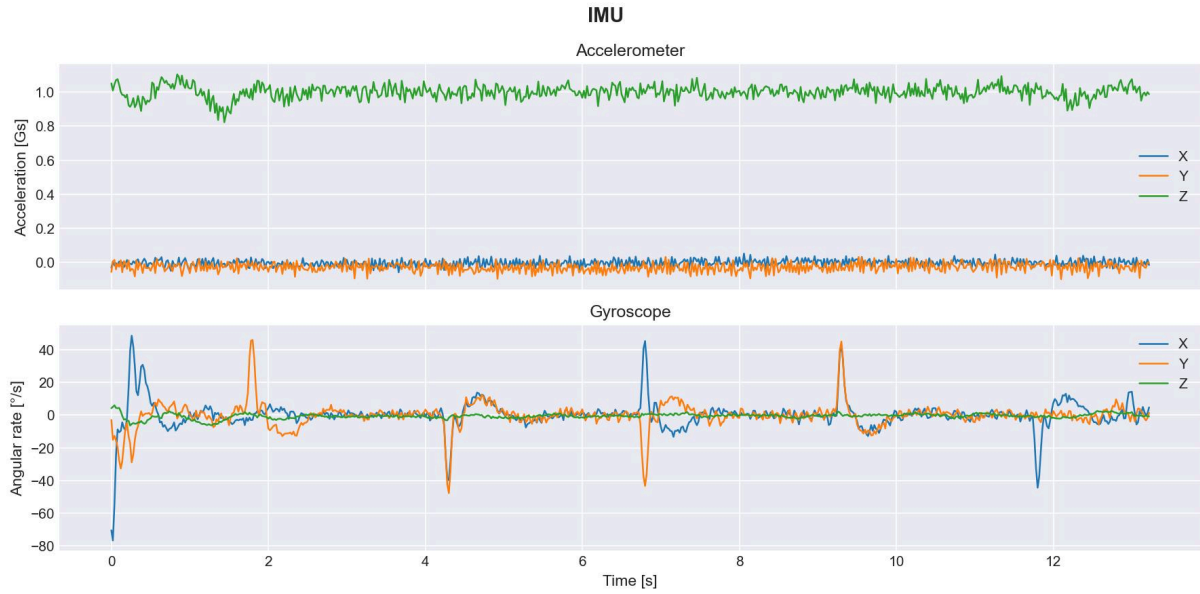


Figure 11: IMU sensor data recorded during a Crazyflie test flight.

2.2.3.3 UWB Sensor Data

[Figure 12](#) shows the TWR distances to the four UWB anchors during the flight. All anchors remained in the 2 – 3.5 m range and the traces vary smoothly as the drone moves, confirming reliable range measurements from the Loco Positioning Deck.

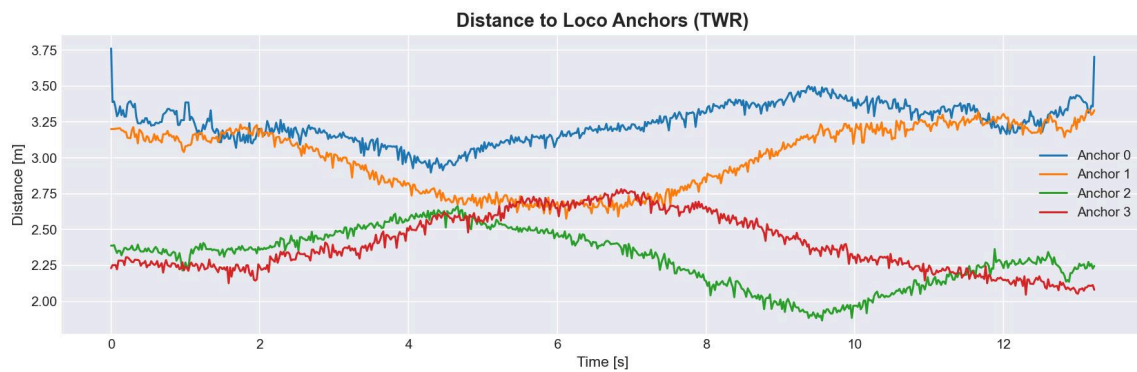


Figure 12: Distance to four UWB anchors recorded during a Crazyflie test flight.

2.2.3.4 ToF Sensor Data

[Figure 13](#) shows the ToF distance readings during the flight. The drone holds a stable cruising height of approximately 30 cm throughout and descends smoothly toward the ground. The recording ends at roughly 10 cm rather than 0 cm, as the logging stopped before the drone completed its landing.

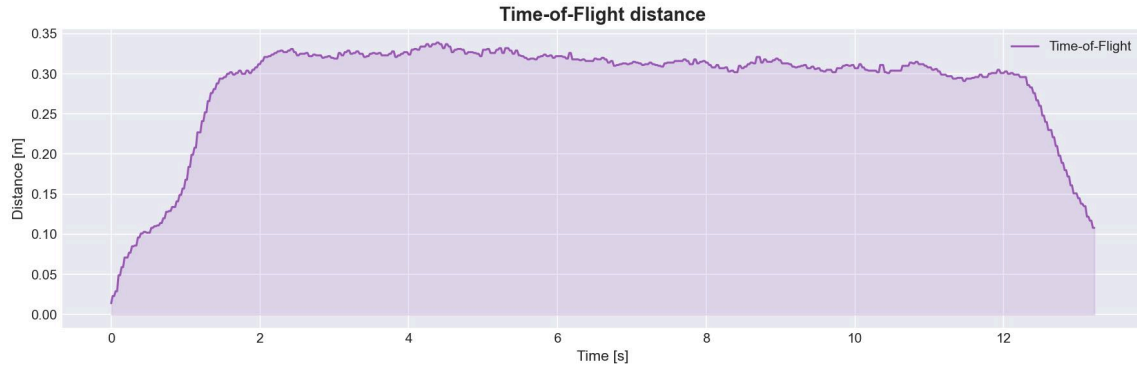


Figure 13: Distance measured by the ToF sensor during a Crazyflie test flight.

These test flights gave us a solid understanding of the data each sensor produces during real flight conditions.

2.3 Platform Comparison

This section compares ASTERIA and the Crazyflie 2.1 across the dimensions most relevant to the EKF design: Platform dynamics, sensor characteristics and operational constraints.

2.3.1 Platform Dynamics

The two platforms operate under fundamentally different dynamic conditions. ASTERIA is a high-speed rocket that reaches supersonic velocities, whereas the Crazyflie 2.1 operates in near-hover conditions at low speed. [Table 7](#) summarizes the key differences.

Property	ASTERIA Rocket	Crazyflie 2.1
Mass	49 kg (dry)	29 g
Dimensions	0.0895 m (radius), 5 m (height)	92 × 92 × 29 mm
Peak Velocity	≈ 353 m/s (Mach 1.1)	≈ 2.5 m/s
Flight Duration	≈ 10 s burn time, ≈ 500 s flight time	max. 7 min
Operating Environment	Outdoor	Indoor
GNSS Availability	Yes	No
Absolute Position Source	GNSS (satellite)	UWB (fixed anchors)
Sensor Redundancy	Dual sensors	Single sensor

Table 7: General platform comparison between ASTERIA and the Crazyflie 2.1.

2.3.2 Sensor Characteristics

Both platforms share an IMU and a barometric pressure sensor, but differ in the technology used for absolute position and in which additional sensors are available. [Table 8](#) gives a combined overview.

Signal	ASTERIA	Max ODR	Crazyflie 2.1	Max ODR
Acceleration	LSM6DSO32	6664 Hz	BMI088	1600 Hz
Angular Rate	LSM6DSO32	6664 Hz	BMI088	2000 Hz
Pressure	MS5607	—	BMP388	200 Hz
Position	ZED-F9P-05B (GNSS)	25 Hz	DWM1000 (UWB)	—
Magnetic Field	LSM303AGR	100 Hz	—	—
Humidity	SHT4	770 Hz	—	—
Temperature	SHT4	770 Hz	—	—
Optical Flow	—	—	PMW3901	—
ToF	—	—	VL53L1x	50 Hz

Table 8: Sensor comparison between ASTERIA and the Crazyflie 2.1.

3 EKF Design

This chapter describes the design of the Multiplicative Extended Kalman Filter (MEKF) implemented as part of this thesis. [Section 3.1](#) motivates the choice of the multiplicative variant for attitude estimation. [Section 3.2](#) introduces the Extended Kalman Filter and its predict-correct cycle and [Section 3.3](#) extends it to the multiplicative formulation used for attitude. [Section 3.4](#) defines the state vector and frame conventions. [Section 3.5](#) presents the constant velocity and IMU-driven process models, while [Section 3.6](#) describes the GNSS, UWB and ToF measurement models. [Section 3.7](#) presents the adaptive measurement noise estimation strategy. [Section 3.8](#) concludes with the known limitations of the design.

3.1 Design Rationale

The primary references used throughout this thesis, namely [\[21\]](#) and [\[13\]](#), consistently adopt the MEKF formulation for attitude estimation, providing a well-established theoretical foundation and concrete derivations to build upon. Furthermore, the Crazyflie firmware already employs an MEKF with scalar updates [\[22\]](#), which serves as a useful point of orientation throughout the design process, reduces integration risk and allows direct comparison of filter behavior across the two target platforms. We conducted an analysis of the Crazyflie EKF implementation, the results of which are documented in [Section VII](#). Based on these considerations, we decided to use the MEKF approach for attitude estimation in this thesis. This decision comes with trade-offs that are discussed in [Section 3.8](#).

3.2 Extended Kalman Filter Introduction

The Extended Kalman Filter (EKF) is a recursive state estimation algorithm for nonlinear systems [\[23\]](#). It extends the classical Kalman filter by linearizing the nonlinear process and measurement functions around the current state estimate at each time step using first-order Taylor expansion, yielding the Jacobian matrices F (state transition) and H (measurement).

[Figure 14](#) illustrates the alternating filter cycle between two steps:

- **Prediction:** Propagate the state estimate $\hat{x}$ and covariance P forward in time using the process model.
- **Correction:** Incorporate a measurement z by computing the measurement innovation¹ $y = z - h(\hat{x})$, the Kalman gain K and updating the state and covariance.

¹There is ambiguity with the terms residual and measurement innovation. In our documentation, we adhere to the definition in [\[13, Section 3.2.1\]](#). In our code, the measurement innovation is called residual.

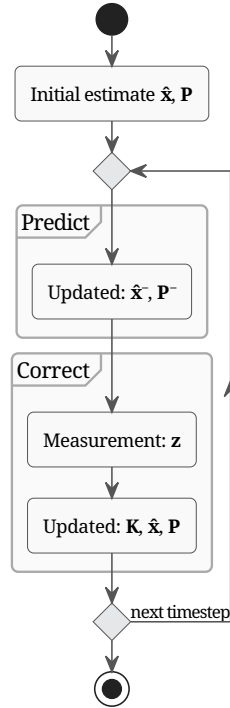


Figure 14: General Kalman filter predict-correct cycle.

3.3 Multiplicative Extended Kalman Filter

A unit quaternion represents 3D attitude without singularities but carries the normalization constraint $\|q\| = 1$, which a standard additive EKF cannot preserve [24].

The Multiplicative Extended Kalman Filter (MEKF) resolves this by decomposing attitude into two parts: a **nominal** quaternion q that is always kept normalized and a small-angle **attitude error** $\delta\theta \in \mathbb{R}^3$ that is part of the filter's state. Because the error is reset to zero after every update (see Figure 15), it remains small and the small-angle linearization stays valid.

Figure 15 shows the resulting predict-correct-finalize cycle [25].

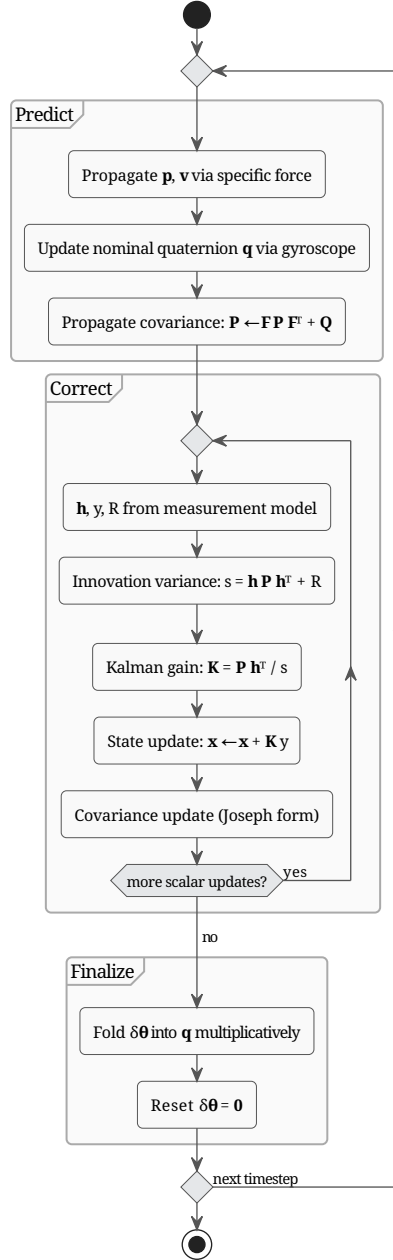


Figure 15: MEKF predict-correct-finalize cycle.

3.4 State Representation and Frame Conventions

The filter tracks a 9-dimensional state vector:

$$x = [N, E, D, V_N, V_E, V_D, \delta\theta_x, \delta\theta_y, \delta\theta_z]^T \quad (1)$$

The first six components are position m and velocity m/s in a local North-East-Down (NED) frame. The last three are the small-angle attitude error in body-frame axes rad. The nominal attitude is maintained as a unit quaternion outside the state vector and describes the rotation from the body frame to the NED frame. Measurement updates in WGS84 coordinates are converted to a local NED frame relative to a fixed reference origin before the updates are applied. All frame definitions and transformations between these frames follow [13].

The covariance $P \in \mathbb{R}^{9 \times 9}$ is initialized as a diagonal matrix reflecting high initial uncertainty.

3.5 Process Models

The filter supports multiple interchangeable process models. For this thesis, two models have been implemented: A constant velocity model and an IMU-driven model.

3.5.1 Constant Velocity Model

The constant velocity model assumes the vehicle moves at constant velocity between updates:

$$p_{k+1} = p_k + v_k \cdot \Delta t, \quad v_{k+1} = v_k \quad (2)$$

Process noise follows the Discrete White Noise Acceleration (DWNA) model [26], which treats unmodeled accelerations as zero-mean white noise with standard deviation σ_a . The resulting process noise matrix couples position and velocity with entries proportional to $\sigma_a^2 \cdot \Delta t^4/4$, $\sigma_a^2 \cdot \Delta t^3/2$ and $\sigma_a^2 \cdot \Delta t^2$.

This model serves as a fallback and is used during early testing. It does not propagate attitude.

3.5.2 IMU-Driven Model

The IMU predictor integrates accelerometer and gyroscope readings to propagate the full state [13]. Position and velocity are updated using the specific force a_b (gravity-compensated accelerometer output in body frame) rotated into NED using the mid-point rotation matrix R :

$$p_{k+1} = p_k + v_k \Delta t + \frac{1}{2} R a_b \Delta t^2, \quad v_{k+1} = v_k + R a_b \Delta t \quad (3)$$

The nominal attitude is updated via the quaternion exponential map from the gyroscope angular rate ω :

$$q_{k+1} = q_k \otimes \exp(\omega \cdot \Delta t/2) \quad (4)$$

The error-state covariance is propagated as $P \leftarrow F P F^\top + Q$. Key off-diagonal blocks of the state transition Jacobian F capture how attitude errors couple into position and velocity through the specific force [13], [21]:

$$F_{\text{vel, att}} = -R[a_b]_\times \Delta t, \quad F_{\text{pos, att}} = -\frac{1}{2} R[a_b]_\times \Delta t^2 \quad (5)$$

where $[a_b]_\times$ denotes the skew-symmetric matrix of a_b . Accelerometer noise uses the DWNA model. Gyroscope noise enters as white noise on the attitude error ($Q_{\text{att}} = \sigma_\omega^2 \Delta t^2 I$).

3.6 Measurement Models

Three measurement sources are integrated: GNSS position, UWB range and ToF altitude.

3.6.1 Scalar Updates

All measurement types, regardless of dimension, are decomposed into a sequence of **scalar updates** [13]. Each scalar update supplies the measurement innovation y , the Jacobian row $h \in \mathbb{R}^{1 \times 9}$ and the measurement variance R . The update proceeds in four steps:

1. Compute the innovation variance: $s = h P h^\top + R$
2. Compute the Kalman gain: $K = P h^\top / s$
3. Update the state: $x \leftarrow x + K y$
4. Update the covariance using the **Joseph form** [13]:

$$P \leftarrow (I - K h) P (I - K h)^\top + K R K^\top \quad (6)$$

The Joseph form is valid for any gain and better preserves the symmetry and positive-definiteness of P under floating-point rounding than the standard form, at the cost of more than twice the operations. Additionally, after every prediction and correction step, symmetry is enforced via $P \leftarrow (P + P^\top)/2$ to eliminate asymmetry accumulated through floating-point rounding.

After all scalar updates, the finalize step folds the accumulated attitude error into the nominal quaternion and resets it to zero.

3.6.2 GNSS Position Model

The GNSS measurement produces three scalar updates, one per NED axis. For each axis $i \in \{N, E, D\}$, the measurement function directly observes the corresponding position state:

$$h_i(x) = x[i], \quad h_i = e_i^\top \quad (7)$$

where $e_i^\top$ is a 1×9 row vector with 1 at position i and 0 elsewhere. The measurement innovation is $y_i = z_i - x[i]$ and the variance is $R_i = \sigma_i^2$.

Geodetic WGS84 coordinates are converted to a local NED frame relative to a fixed reference origin before the updates are applied.

3.6.3 UWB Range Model

An UWB anchor at known NED position a provides a single scalar update. The nonlinear measurement function is:

$$h(x) = \|p - a\|_2 \quad (8)$$

where $p = [N, E, D]^\top$ is the estimated position extracted from x and a is the anchor position. The Jacobian row is the normalized direction vector from anchor to estimated position:

$$h = \frac{(p - a)^\top}{\|p - a\|_2} \quad (9)$$

The measurement innovation is $y = d - \|p - a\|_2$ where d is the measured distance and the variance is $R = \sigma^2$. A singularity guard applies when $\|p - a\|_2 \approx 0$ to avoid division by zero, falling back to $e_N^\top$, the 1×9 row vector with 1 at the North position and 0 elsewhere.

The measurement function follows the model used in the Crazyflie firmware [22].

3.6.4 ToF Distance Model

The ToF sensor measures distance to the ground below the vehicle. Assuming flat ground at $D = 0$ in NED, let $\varphi = \arccos(R_{zz})$ be the vehicle's tilt angle from vertical. The measurement function and its Jacobian row are:

$$h(x) = -\frac{D}{c_\varphi}, \quad h_{[D]} = -\frac{1}{c_\varphi} \quad (10)$$

where $c_\varphi = \cos(\max(\varphi - \varphi_{\text{cone}}, 0))$, $\varphi_{\text{cone}} = 7.5^\circ$ is the sensor's half field-of-view cone angle and all other Jacobian entries are zero. The clamping ensures the cone correction only applies when the tilt exceeds the half-angle. The measurement innovation is $y = d - h(x)$ where d is the measured distance and the variance is $R = \sigma^2$.

The update is gated: if $R_{zz} \leq 0.1$ (tilt beyond approximately 84° from vertical), the measurement is discarded to prevent invalid height estimates.

The measurement function follows the model used in the Crazyflie firmware [27].

3.7 Adaptive Measurement Noise

The measurement variance R controls how strongly the filter trusts each incoming measurement relative to its own prediction. A fixed R assumes the sensor noise is known and constant, but in practice it may be poorly calibrated or drift during operation.

To test whether R can be estimated online from the measurement innovations y and letting the filter adapt to its measurements automatically, we implement a sliding window and an exponential moving average (EMA) variant, both exploiting the relation [28]:

$$\hat{R} = \overline{y^2} - \mathbf{h}P\mathbf{h}^\top \quad (11)$$

We adopt only this innovation-based estimate of the measurement noise from [28]. The process-noise estimation, dead-reckoning swapping and optimality checks of the full method are not implemented here.

The term $\mathbf{h}P\mathbf{h}^\top$ subtracts the contribution of state uncertainty from the squared innovation, isolating the measurement noise. We clamp the estimate $\hat{R}$ to $[10^{-6}, 200]$, following the recommendation to impose limits for numerical stability [13, Section 3.4.4]. The lower bound keeps the estimate strictly positive, while the upper bound prevents divergence.

3.7.1 Sliding Window Variant

The sliding window variant computes $\hat{R}$ directly from the windowed innovation estimate of [28]. A buffer of the last N measurement innovations is kept. The estimate of R is then calculated as

$$\hat{R} = \frac{1}{N} \sum_{i=k-N+1}^k y_i^2 - \mathbf{h}P\mathbf{h}^\top, \quad (12)$$

where N is the chosen window size. The estimated $\hat{R}$ is only used once the buffer is full. Until then, the filter falls back to the fixed R .

3.7.2 Exponential Moving Average Variant

For the exponential moving average (EMA) variant, we replace the window with an EMA to test whether the same adaptation can be achieved without storing the last N samples, which would drastically reduce the memory allocated on our target platform. The estimate is updated at every step with a smoothing factor $\alpha \in (0, 1)$:

$$\hat{R} \leftarrow (1 - \alpha)\hat{R} + \alpha \cdot \max(y^2 - \mathbf{h}P\mathbf{h}^\top, 10^{-6}). \quad (13)$$

A higher α adapts faster at the cost of more variance in the estimate. Unlike the sliding window, the EMA keeps only the running estimate $\hat{R}$ in memory and updates from the first sample onward.

3.8 Limitations

The design presented in this chapter makes several simplifying assumptions that bound its accuracy and robustness. The following subsections describe each known limitation. The implication of these limitations is further discussed in Section 7.7.

3.8.1 IMU Bias Terms

The current design omits **accelerometer and gyroscope bias terms**. The LSM6DSO32 (see Section 2.1.2) and BMI088 (see Section 2.2.2) exhibit slowly varying biases (see Section 4.5). If unmodeled, these accumulate as growing position and attitude errors. This is particularly

relevant when GNSS is absent and the filter relies solely on IMU integration. Incorporating bias states would extend the state vector to 15 components (adding 3 accelerometer and 3 gyroscope bias terms) and require corresponding process noise modeling [13]. We have implemented a prototype, the details of which can be found in the outlook (see [Section 8.2.2](#)). Proper evaluation of this extension is identified as future work. Notably, the Crazyflie firmware also operates with a 9-state filter and still achieves adequate positioning performance in practice, as the evaluation in [Section 7](#) also shows.

3.8.2 Yaw Observability

As the current design does not have a direct heading measurement, it cannot directly observe yaw. The current design relies on cross-coupling attitude into position and velocity via the specific force in order to correct attitude errors. This is sufficient to constrain roll and pitch, since gravity provides a constant specific force signature perpendicular to the horizontal plane. However, as yaw errors leave the gravity vector unchanged, they cannot be observed using gravity alone. Yaw can only be observed during horizontal accelerations, such as changes in speed or turns, which produce a specific force signature that changes with yaw error. During constant velocity motion, yaw is effectively unobservable, drifting with gyro bias [13, [Section 14.2.1](#)].

3.8.3 Innovation Filtering

Our design applies no innovation filtering to GNSS or UWB measurements. Both are accepted, meaning that a faulty GNSS fix or a spurious UWB range can directly corrupt the state estimate. Innovation filtering compares the normalized measurement innovation against a threshold before computing the Kalman gain and rejects the measurement if the threshold is exceeded [13, [Section 17.3.1](#)]. This limitation applies equally to the adaptive measurement noise variant, which adjusts the noise covariance but still accepts every measurement without gating.

3.8.4 IMU Vibration

We currently do not apply any mitigation strategies against IMU vibrations. During high-acceleration phases such as motor burn, propulsion vibrations can corrupt IMU readings through vibration rectification errors and coning or sculling errors, which propagate directly into the navigation states [13, [Section 4.4.5](#)]. Examples of mitigation in software include low-pass filtering the IMU signal, using coning and sculling compensation, or increasing the process noise covariance to account for the elevated sensor uncertainty [13, [Section 5.5.4](#)], [29].

3.8.5 GNSS Time Delay

The current design does not account for the delay between the valid time of a GNSS measurement and its arrival at the filter. If this delay is not accounted for, the filter incorrectly attributes the resulting discrepancy to the navigation states. This error increases with vehicle dynamics [13, [Section 3.3.4](#)]. Possible mitigations are buffering and interpolating the IMU output to the GNSS time of validity, propagating the measurement matrix to account for state propagation past that time or increasing the measurement noise covariance to absorb the timing error [13, [Section 3.3.4](#)].

3.8.6 Adaptive Measurement Noise Invalidation

The adaptive variants presented in [Section 3.7](#) do not account for long periods without measurement updates. The estimated measurement noise $\hat{R}$ persists unchanged across such

gaps, so a value learned under favorable conditions is reused once updates resume, even if the sensor characteristics have changed in the meantime.

A representative case is GNSS during a rocket flight: A low $\hat{R}$ is learned on the launch pad under good reception, followed by a long outage during ascent. When GNSS returns, the signal is often still degraded, but the stale low $\hat{R}$ causes the filter to treat these measurements as overly accurate.

A possible mitigation is to invalidate $\hat{R}$ after a defined time without updates and fall back to the fixed R , until enough new measurement innovations have been collected to re-estimate it. This is identified as future work.

3.8.7 Linearization Error

The first-order Taylor linearization can introduce truncation errors during aggressive maneuvers with high angular rates, potentially causing the filter to diverge [13]. An Unscented Kalman Filter (UKF) would handle such nonlinearities more robustly by propagating sigma points through the nonlinear functions directly, at the cost of higher computational complexity and implementation effort [13].

4 Simulation

A central requirement of this thesis is that the EKF must be validated without physical flight tests ([MHR-06](#)). The ASTERIA rocket has not yet completed a full flight and the available drop-test data from its predecessor HERMES covers only the descent phase under a parachute (see [Section 2.1.3](#)). Indoor flights with the Crazyflie are technically possible, but tuning and validating the filter on a physical drone risks damage to the hardware and does not provide the ground-truth reference that filter evaluation requires.

A simulation environment provides two assets that recorded flight data cannot: Ground-truth state information against which estimation errors can be measured directly and full control over sensor characteristics such as noise, bias and dropout. These properties make simulation the foundation for tuning the EKF and for the evaluation requirements of this thesis which include analyzing filter behavior across distinct flight phases ([MHR-07](#)), performing offline validation against a known reference ([MHR-08](#)) and comparing filter performance across platforms ([MHR-09](#)).

This chapter describes the simulation pipeline used to generate data for the ASTERIA rocket and the Crazyflie drone. [Section 4.1](#) analyzes simulation tools for the ARIS rocket, selects RocketPy as the primary tool and documents the simulation configuration. [Section 4.2](#) performs the same analysis for the Crazyflie and selects RotorPy. [Section 4.3](#) discusses the credibility of both simulation engines. [Section 4.4](#) describes how sensor signals not provided natively by the chosen simulators are derived from the simulated ground truth. [Section 4.5](#) defines the noise models applied on top of the clean signals. [Section 4.6](#) describes the dropout strategies used to model temporary sensor outages.

The software architecture of the simulation pipeline itself is described in [Section 5.2](#).

4.1 Rocket Simulation

To validate and optimize the EKF without physical flight tests, a simulation environment is needed to generate ground-truth sensor data for the full duration of a rocket flight.

4.1.1 Rocket Simulation Tool Requirements

The simulation must produce ground-truth data that corresponds to the physical quantities measured by the on-board sensors described in [Section 2.1.2](#).

[Table 9](#) lists the required simulation outputs derived from the sensors on the sensor carrier.

Output	Description	Required by
3-axis linear acceleration	Acceleration vector in the rocket body frame	LSM6DSO32, LSM303AGR
3-axis angular rate	Angular velocity vector in the rocket body frame	LSM6DSO32
Atmospheric pressure	Pressure along the flight trajectory	MS5607
Temperature	Temperature along the flight trajectory	LSM6DSO32, MS5607, SHT4
3-axis magnetic field	Earth's magnetic field vector in the rocket body frame	LSM303AGR
Position (latitude, longitude, altitude)	Global position of the rocket	ZED-F9P-05B
Velocity	3-axis velocity vector in the NED frame	ZED-F9P-05B

Table 9: Required simulation outputs for ground-truth data.

We omitted humidity as a required output because its contribution to the state estimation accuracy is negligible. Beyond the outputs above, the tool should ideally also provide a programmatic interface that would enable hardware-in-the-loop (HIL) tests.

4.1.2 Rocket Simulation Tool Evaluation

[Table 10](#) lists the candidate tools that we evaluated.

Software	Description	Source
OpenRocket	A free, open-source rocket simulation software used for modeling and simulating rocket flights, including flight dynamics, aerodynamics and parachute deployment.	OpenRocket
RocketPy	An open-source Python-based framework for simulating and designing high-performance rockets, primarily used in academic and professional contexts.	RocketPy
ARIS gnc-simulation	A Rust-based simulator developed by ARIS to model the ASTERIA avionics stack and the guided descent. The tool is not open-source and access to the ARIS GitHub is required to access it.	GitHub

Table 10: Analyzed simulation tools for rocket flight.

We evaluated the tools against the criteria listed in [Table 11](#). The criteria focus on whether a tool can produce the required ground-truth outputs from [Table 9](#) as well as some additional criteria needed to generate data.

The symbols used in [Table 11](#) are defined as follows: ✓ indicates full support, ~ indicates partial or limited support and ✗ indicates no support.

Criterion	OpenRocket	RocketPy	ARIS gnc-sim
6-DOF flight dynamics	✓	✓	✓
Atmospheric model (pressure, temperature)	✓	✓	~
Magnetic field model	✗	✗	✗
Programmatic API access	~	✓ (Python)	✓ (Rust)
Open-source and accessible	✓	✓	✗
Actively maintained and documented	✓	✓	✓
Simulation of launch, powered flight, coasting and recovery phases	✓	✓	✗
Simulation of guided descent used by ASTERIA	✗	✗	✓

Table 11: Simulation tool comparison for the ARIS rocket.

As seen in [Table 11](#), no tool currently supports the output of a magnetic field vector. The Earth’s magnetic field can be computed offline given the simulated position. Thus, the absence of a built-in magnetic field model in the simulation is not a blocking issue.

4.1.2.1 Tool Selection

OpenRocket is primarily GUI-based. While external scripting is possible through Python and JPyype [\[30\]](#), using OpenRocket programmatically requires considerable effort, as OpenRocket does not offer a native Python API.

The ARIS gnc-simulation only covers the guided descent, not the powered flight and coast phases. Since we also require simulated data from these phases to evaluate our EKF, the gnc-simulation cannot be used as the primary tool.

RocketPy offers a full six-degree-of-freedom simulation with a well-documented Python API, enabling programmatic access to all relevant flight data [\[31\]](#). The ASTERIA Flight Dynamics team already created a RocketPy setup prior to this project. Currently, no descent simulation exists within this setup but RocketPy provides a well-documented interface for configuring a parachute phase. The only limitation is that the ASTERIA guided descent cannot be modeled, as RocketPy only supports unguided descent using round parachutes.

Therefore, we selected **RocketPy** as the primary simulation tool for this thesis. The ARIS gnc-simulation remains available as a cross-validation environment for guided-descent conditions.

4.1.3 Rocket Simulation Configuration

This section documents the configurations used for the ASTERIA rocket simulation.

4.1.3.1 ASTERIA Rocket Model

The ASTERIA rocket model is configured to match the physical airframe and propulsion system as documented by the ARIS Flight Dynamics team. As the ASTERIA design is still being optimized, the configuration described here reflects the state of the rocket at the time of writing. [Figure 16](#) shows the resulting RocketPy representation.

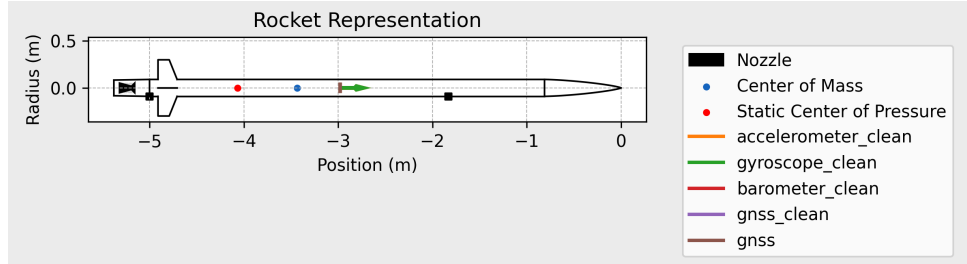


Figure 16: RocketPy plot of the configured ASTERIA rocket.

The airframe configuration uses RocketPy's `tail_to_nose` coordinate system. We chose this convention as a workaround for an issue we encountered with the alternative `nose_to_tail` orientation, which produced unexpected sensor outputs.

RocketPy supports the addition of multiple sensors to a rocket to simulate their output. We added an accelerometer, a gyroscope, a barometer and two GNSS receivers to the ASTERIA rocket model. The two GNSS instances differ only in their noise configuration. One is configured with zero noise and used as reference, while the other produces the noisy measurement stream consumed by the EKF. The noise configuration for each sensor is described in [Section](#).

The simulated sensor carrier is mounted at -2.978 m, which corresponds to the center of mass of the rocket during the majority of the flight. Placing the sensor at the center of mass minimises lever-arm contributions to the simulated accelerometer and gyroscope data (see [Section 4.1.4.5](#)).

The remaining airframe parameters are taken from the ARIS Flight Dynamics team's design data. The most relevant values are summarised in [Table 12](#).

Parameter	Value	Unit
Radius	0.0895	m
Dry mass (without motor)	46.01	kg
Dry mass (with motor)	49.00	kg
Total mass (with motor and propellant)	76.34	kg
Inertia	(94.51, 94.52, 0.2107)	kg·m ²
Nose cone length	0.81	m
Number of fins	4	—
Tail length	0.38	m

Table 12: ASTERIA airframe parameters used in the RocketPy model.

The motor is modelled with RocketPy's `GenericMotor` class, whereas the ARIS Flight Dynamics team uses a `LiquidMotor` instead. The `GenericMotor` is a simple motor defined mainly

by its thrust curve and without any distinction between propellant types [32]. We decided to use the `GenericMotor` due to its simpler configuration.

The parameters used for the `GenericMotor` are listed in Table 13. The dry motor mass, tank volumes, chamber radius and nozzle radius are taken directly from the ARIS Flight Dynamics team’s design data. The chamber height was computed by summing the heights of the three propellant tanks present in the ASTERIA rocket. The propellant masses are derived from the tank volumes and the densities of LOX (1141 kg/m³), Ethanol (789 kg/m³) and the pressurant (303 kg/m³ at approximately 300 bar), which yield a total propellant mass of 27.36 kg.

The thrust curve was provided by the ARIS Flight Dynamics team as a CSV file and is loaded into RocketPy with linear interpolation between sample points. An initial comparison of our simulation’s trajectory with the reference simulation provided by the ARIS Flight Dynamics team showed that our simulation’s apogee altitude was lower. We assume this is because the thrust curve was originally tuned for the `LiquidMotor`. As a correction, we uniformly scaled the thrust curve by a factor of 1.15. This resulted in the simulated apogee aligning more closely with the expected ASTERIA performance. The implications of this correction are documented in Section 4.1.4.2.

Parameter	Value	Unit
Dry motor mass	2.99	kg
Liquid oxygen volume	0.0131	m ³
Ethanol volume	0.0131	m ³
Pressurant volume	0.0068	m ³
Combustion chamber radius	0.0845	m
Combustion chamber height	1.64	m
Nozzle radius	0.0511	m

Table 13: ASTERIA motor parameters used in the `GenericMotor` model.

The rocket is equipped with two parachutes that deploy sequentially. The drogue parachute deploys at apogee and is configured with a drag area of 0.7 m² and an inflation lag of 1.5 s. The main parachute is configured with a drag area of 24.2 m² and it deploys when the rocket descends below 2000 m above ground level and uses the same 1.5 s inflation lag. In reality, the ASTERIA main parachute is a parafoil that is used for performing a guided descent. RocketPy does not support guided descent and only models circular parachutes. The implications of this are documented in Section 4.1.4.6.

4.1.3.2 Launch Environments

All rocket simulations use the EuroC launch site as their reference location, as ASTERIA will launch there in October 2026. The site is located in Portugal at latitude 39.389700°N and longitude 8.288964°W, at an elevation of 113 m.

The baseline environment uses real atmospheric data from the ECMWF ERA5 reanalysis dataset [33]. The data was provided as a NetCDF file from the ARIS Flight Dynamics team.

In addition to the reanalysis baseline, we configured a set of synthetic environment variants in which the atmospheric model is replaced by RocketPy’s custom atmosphere. The variants are summarised in Table 14.

Variant	Atmosphere	Wind (east)	Wind (north)
Reanalysis	ERA5 (NetCDF)	from data	from data
NoWind	Custom	0 m/s	0 m/s
SlightWind	Custom	2 – 6 m/s	0.5 – 2 m/s
Windy	Custom	8 – 20 m/s	3 – 8 m/s
Gust	Custom	gust to 25 m/s at 1.6 – 1.9 km	gust to 10 m/s at 1.6 – 1.9 km
HotDay	Custom, 40 °C	3 m/s (constant)	0 m/s (constant)
ColdDay	Custom, 0 °C	0 m/s (constant)	3 m/s (constant)

Table 14: EuroC environment variants used across the rocket simulation scenarios.

These variants enable parameter optimisation on different conditions and they are used to verify that the filter generalises well.

4.1.3.3 Simulation Parameters and Frame Conventions

The simulation outputs data at a fixed sampling rate of 800 Hz. A single simulated flight takes approximately four minutes to compute on a standard laptop.

The simulator-internal state is stored in the `GroundTruth` contract defined in [Section 5.2.3](#), which fixes the frames and units for every field. RocketPy’s native output already uses the ENU frame and the expected units, so no rotation or conversion is applied. The only exception is the GNSS position. RocketPy assumes a spherical Earth, so coordinates are manually recomputed from ground-truth data to the WGS84 datum (see [Section 4.1.4.3](#)).

4.1.3.4 Sensor Configuration

The ASTERIA sensor carrier contains each sensor at least twice for redundancy. The simulation models only one instance of each sensor. This choice reduces the complexity of the simulation configuration.

The magnetometer is an exception to the sensor configuration described in this section because it is generated but not currently consumed by the EKF (see [Section 4.1.4.4](#)).

Sensor Rates and Sources

[Table 15](#) lists the sensors configured in the rocket simulation together with their rates and origin. The origin distinguishes between sensors that RocketPy provides natively and sensors that are generated from the ground truth through our synthetic-sensor stage (see [Section 4.4](#)).

The IMU is specified at 833 Hz in hardware, but we slightly reduced the simulated rate to 800 Hz so that the lower-rate sensors can be sampled directly from the same time grid without requiring interpolation. The same approach is applied to the barometer.

Sensor	Hardware rate	Simulation rate	Source
Accelerometer	833 Hz	800 Hz	RocketPy native
Gyroscope	833 Hz	800 Hz	RocketPy native
GNSS receiver	25 Hz	25 Hz	RocketPy native
Barometer	110 Hz	100 Hz	RocketPy native
Magnetometer	100 Hz	100 Hz	Synthetic

Table 15: Sensors configured in the rocket simulation, including sampling rates and the origin of each signal.

Sensor Noise Parameters

[Table 16](#) lists the noise parameters applied to each configured sensor. Where available, the values are derived from the corresponding manufacturer datasheets. For the remaining parameters, we made and documented reasonable assumptions. The limitations of this approach are discussed in [Section 4.1.4.7](#).

For the accelerometer, gyroscope, barometer and magnetometer, the noise is applied using our own noise model (see [Section 4.5](#)). The GNSS receiver is the exception. RocketPy’s built-in GNSS noise model is used, since it accepts horizontal and vertical accuracy values directly in meters which matches the parameters provided in the u-blox ZED-F9P datasheet [\[7\]](#).

The LSM6DSO32 datasheet [\[3\]](#) does not contain any random walk values. An analysis of the closely related LSM6DSR reports an accelerometer random walk of roughly 70 – 170 μg and gyroscope random walk of 5 – 13 mdps [\[34\]](#). We therefore set the IMU random walk values to conservative values of 0.2 mg and 0.02 dps. These values are estimates rather than measured quantities for the specific parts used.

Sensor	Noise density	Fixed bias	Random Walk	Scale factor	Range
Accelerometer	220 $\mu\text{g} / \sqrt{\text{Hz}}$	± 20 mg	0.2 mg	0.005	± 32 g
Gyroscope	3.8 mdps $/ \sqrt{\text{Hz}}$	± 0.5 dps	0.02 dps	0.005	± 2000 dps
GNSS receiver ²	horizontal ³ : 1.5 m, vertical ³ : 2.25 m	$\times$	$\times$	$\times$	$\times$
Barometer ⁴	0.024 mbar ⁵	± 2 mbar	0.2 mbar	0	300 – 1100 mbar
Magnetometer ⁴	3 mgauss ⁵	± 60 mgauss	6 mgauss	0	± 49.15 gauss

Table 16: Noise parameters applied to the rocket simulation sensors.

4.1.3.5 Simulation Scenarios

The configurations described in the previous sections define all aspects of a rocket simulation. A simulation scenario now defines a concrete combination of these configurations that is run end-to-end and can be used for evaluation.

²Noise applied via RocketPy’s built-in GNSS noise model, not our noise-injection stage.

³GNSS accuracy is given as CEP (circular error probable).

⁴Generated but not included in the EKF configuration.

⁵The barometer and magnetometer noise values are the RMS noise figures stated in the datasheets.

[Table 17](#) summarizes the scenarios used in this thesis.

Nr.	Environment	Dropouts	Noise	Purpose
1	Reanalysis	None	$0.5 \times$ on all sensors	Low-noise baseline
2	Reanalysis, no parachute	GNSS dropout above $4g$	Nominal on all sensors	Tests estimation during ballistic descent after parachute failure
3	NoWind	None	$1.5 \times$ on IMU, nominal on GNSS	Evaluates position estimation under elevated noise
4	SlightWind	GNSS dropout between 2 and 40 s and 50% GNSS per-sample-dropout probability afterwards	Nominal on IMU, $1.5 \times$ on GNSS	Evaluates position estimation under faulty GNSS
5	Windy	None	Nominal on all sensors	Tests estimation under strong wind
6	Gust	GNSS dropout above $4g$	Nominal on all sensors	Tests estimation under wind gusts combined with GNSS loss at launch
7	HotDay	GNSS dropout above $4g$	Nominal on IMU, $1.5 \times$ on GNSS	Evaluates estimation under a hot atmosphere and degraded GNSS
8	ColdDay	GNSS dropout above $4g$	$1.2 \times$ on IMU, nominal on GNSS	Evaluates estimation under a cold atmosphere and elevated sensor noise
9	Reanalysis	GNSS dropout between 2 and 40 s	nominal on all sensors	Evaluation flight excluded from optimization data

Table 17: Simulation scenarios used in the rocket evaluation.

These scenarios cover the flight dynamics and sensor noise conditions used for evaluation and optimization. The final scenario is excluded from the optimization dataset (see [Section 7.2](#)) and serves as a validation flight.

4.1.4 Limitations and Workarounds

RocketPy is a general-purpose rocket simulator and does not reproduce every aspect of the ASTERIA rocket. This section documents the limitations we encountered and the workarounds we applied.

4.1.4.1 Parachute Phase 3-DOF Model

RocketPy switches from a 6-DOF model to a 3-DOF model at the moment the drogue parachute deploys [35]. From that point on, the simulator no longer tracks the rocket's attitude and reports no angular velocity to the gyroscope. Thus, attitude data from the moment of drogue deployment onwards can no longer be used as ground truth for the evaluation.

We apply two mitigations to prevent the 3-DOF artifacts from corrupting the evaluation. First, we freeze the ground-truth quaternion at the moment of drogue inflation. Second, the EKF runner zeroes the gyroscope input past the same timestamp, freezes the attitude inside the EKF state and zeroes the attitude covariances.

As a consequence, attitude estimation cannot be evaluated during the descent phase. The rest of the rocket state can still be evaluated but the simplified 3-DOF descent dynamics may deviate from the real rocket's behavior and should be kept in mind when interpreting descent-phase results.

4.1.4.2 Motor Configuration

As described in [Section 4.1.3.1](#), the ASTERIA motor is modelled with RocketPy's `GenericMotor` and the thrust curve is empirically scaled by 1.15 to compensate for the underestimated apogee.

A more accurate model would replace the `GenericMotor` with a `LiquidMotor` configured with the actual propellant properties. This would remove the need for the empirical scaling factor.

4.1.4.3 GNSS Data Produced by RocketPy

RocketPy's built-in GNSS outputs calculate latitude and longitude values by using the Haversine formula, which assumes a spherical Earth [36]. This approximation is not accurate enough for our use case, as we observed discrepancies of up to 10 meters in the location data compared to the WGS84 data.

To keep the RocketPy output consistent with the WGS84 datum used in the EKF, we recompute the GNSS coordinates from the ENU ground truth using the `pymap3d` library, which performs a proper WGS84 projection. To preserve the noise that Rocketpy generated, we compute the difference between the noisy and the clean Haversine outputs of RocketPy, which isolates the sensor noise component from the projection error. We then add this noise back on top of the WGS84 ground truth. The result is a noisy GNSS measurement stream that is consistent with the WGS84 datum while retaining the noise produced by RocketPy.

4.1.4.4 Magnetometer Output

RocketPy does not provide a magnetometer sensor and does not output a magnetic field vector along the trajectory. The LSM303AGR magnetometer on the sensor carrier can therefore not be simulated with RocketPy.

We generate a synthetic magnetometer signal by using the simulated position and attitude in combination with the Python library `geomag`.

The EKF does not consume magnetometer data in its current configuration, so the synthetic magnetometer signal has not been further tested or validated.

4.1.4.5 Lever Arm Corrections

In a real rocket, the IMU is rarely mounted exactly at the center of mass. This offset between the IMU position and the center of mass introduces lever-arm effects in the measured acceleration [13]. The center of mass is also not fixed: As propellant is burnt, it shifts along the rocket's axis.

Our EKF does not model the lever arm (see [Section 5.1.8.2](#)) and the simulation is configured to match this assumption. As described in [Section 4.1.3.1](#), we placed the simulated sensor carrier at the center of mass after the burn phase to keep the lever arm minimal. A residual offset therefore remains during the powered flight phase.

4.1.4.6 Circular Parachute

The real ASTERIA main parachute is a parafoil. During the descent, this parachute produces a horizontal velocity that is used to steer and to achieve a guided descent.

RocketPy only supports circular parachutes and no guided descent [\[37\]](#). For the simulation, we approximated the main parachute by using the same drag area as the ASTERIA parafoil. The descent trajectory produced by this configuration is purely vertical apart from wind effects. The drogue parachute of ASTERIA is circular in reality and is therefore reproduced accurately by RocketPy's model.

As a result, the descent phase in the simulation models an unguided descent rather than the guided descent that the real rocket performs. Modelling the guided descent would require switching to a parafoil-capable simulator that supports the guided descent, such as the ARIS gnc-simulation (see [Section 4.1.2](#)).

4.1.4.7 Noise Model and Values

The noise parameters used in the rocket and drone simulations are derived from manufacturer datasheets wherever possible. This approach has different limitations that we accept for this thesis.

First, manufacturer datasheets do not provide every parameter that our sensor error model requires. Most datasheets specify a noise density, a fixed bias range and a measurement range. Values such as random walk or scale factor error are often missing or reported in a different form. For the parameters that are not specified in the datasheet, we applied and documented reasonable assumptions based on additional research.

Second, the noise model itself is a simplified representation of real sensor behavior. As described in [Section 4.5](#), it omits multiple noise characteristics. It also assumes the noise characteristics are constant across the full measurement range and the full duration of a flight, which is not true for sensors such as the DWM1000 UWB transceiver.

The same applies to the used GNSS receiver, whose real error characteristics are more complex than the horizontal and vertical accuracy figures RocketPy's built-in model uses. We also do not model measurement latency. A real GNSS receiver outputs a position fix with a delay relative to the IMU samples, whereas in our simulation every sensor stream is aligned to the same time grid and treated as instantaneous.

The most accurate approach would be to determine the noise parameters experimentally on the actual ASTERIA sensor carrier and the Crazyflie. A typical procedure would record several minutes of stationary sensor data and compute noise density, fixed bias and random walk from real data [\[13\]](#). Performing this characterisation is outside the time budget of this thesis.

As a consequence, we accept that our sensor model does not capture every effect present on the real sensors and that the selected noise parameters may deviate from the true values. The evaluation results presented in this thesis should be interpreted with this limitation in mind.

4.2 Drone Simulation

To validate and optimize the EKF for the Crazyflie drone without physical flight tests, a simulation environment is needed to generate ground-truth sensor data representative of indoor quadrotor flight.

4.2.1 Drone Simulation Tool Requirements

The simulation must produce ground-truth data that corresponds to the physical quantities measured by the on-board sensors described in [Section 2.2.2](#).

[Table 18](#) lists the required simulation outputs derived from the sensors and decks on the Crazyflie platform.

Output	Description	Required by
3-axis linear acceleration	Acceleration vector in the body frame	BMI088
3-axis angular rate	Angular velocity vector in the body frame	BMI088
Optical flow	Translational flow measurements (delta x, delta y in pixels)	Flow Deck v2 (PMW3901)
Height	Distance to ground measured by ToF sensor	Flow Deck v2 (VL53L1x)
UWB ranging	Distances from the Crazyflie to the Loco Positioning Nodes	Loco Positioning Deck (DWM1000)

Table 18: Required simulation outputs for Crazyflie ground-truth data.

4.2.2 Drone Simulation Tool Evaluation

[Table 19](#) lists the candidate tools that we evaluated.

Software	Description	Source
Webots	A free and open-source robot simulator maintained by Cyberbotics. It supports a wide range of robot models including multirotor UAVs and provides a rich physics engine based on ODE/Bullet as well as a Python API for programmatic control.	Webots
RotorPy	A lightweight Python-based quadrotor simulation framework focused on flight dynamics and sensor modelling. It is designed for rapid prototyping of estimation and control algorithms with minimal dependencies.	RotorPy
CrazySim	A Gazebo-based simulator specifically designed for the Crazyflie platform. It integrates with the official Crazyflie firmware and supports software-in-the-loop (SITL) testing by running the actual Crazyflie firmware against simulated sensor inputs.	CrazySim

Table 19: Analyzed simulation tools for the Crazyflie.

We evaluated the tools against the criteria listed in [Table 20](#). The criteria focus on whether a tool can produce the required outputs from [Table 18](#), as well as support for the decks specific to the Crazyflie platform.

The symbols used in [Table 20](#) are defined as follows: ✓ indicates full support, ~ indicates partial or limited support and ✗ indicates no support.

Criterion	Webots	RotorPy	CrazySim
6-DOF flight dynamics	✗	✓	✓
Gyroscope model (angular rate)	✓	✓	✓
Accelerometer model	✗	✓	✓
Optical flow simulation (Flow Deck v2)	~	~	~
ToF sensor simulation (Flow Deck v2)	~	~	~
UWB sensor simulation (Loco Positioning Deck)	~	~	~
Lightweight, no GUI simulator required	✗	✓	✗
Easy integration as Python library	~	✓	✗
Open-source and accessible	✓	✓	✓
Actively maintained and documented	~	~	~
Software-in-the-loop (SITL) with Crazyflie firmware	✗	✗	✓

Table 20: Simulation tool comparison for the Crazyflie.

4.2.2.1 Tool Selection

As seen in [Table 20](#), no tool provides out-of-the-box simulation of both the Flow Deck v2 and the Loco Positioning Deck. Optical flow measurements can be derived from the ground-truth data and UWB ranging distances can be computed from the simulated position and known anchor positions. These quantities therefore do not require native simulator support and can be synthetically generated in a post-processing step if needed.

Webots is excluded even though it supports a wide range of sensors. The official Bitcraze documentation explicitly notes that the Webots Crazyflie model is not realistic, not feature-rich and not actively maintained [\[38\]](#). In particular, the Webots model lacks an accelerometer, which is a required output for IMU-based state estimation and disqualifies it for this use case.

CrazySim supports software-in-the-loop testing by running the actual Crazyflie firmware inside the simulation [\[39\]](#). Its sensor models are however tightly coupled to the firmware, which makes it difficult to inject custom noise models or modify sensor behavior for isolated EKF evaluation.

RotorPy offers a six-degree-of-freedom multicopter simulation with a documented Python API and exposes all relevant state data programmatically [\[2\]](#). The authors validated the simulated IMU sensor model against real Crazyflie flight data and report good agreement for both accelerometer and gyroscope measurements [\[2\]](#). RotorPy does not provide dedicated documentation beyond the repository README and the publication, so usage relies on reading the source code and the included examples.

We selected **RotorPy** as the primary simulation tool for the Crazyflie simulation. CrazySim remains available as a cross-validation environment for tests that require realistic firmware behavior.

4.2.3 Drone Simulation Configuration

This section documents the configurations used for the Crazyflie simulation.

4.2.3.1 Crazyflie Model

The Crazyflie is represented in RotorPy by a built-in parameter set named `crazyflie_params`, which the RotorPy authors validated against real flight data of a Crazyflie platform [\[2\]](#). We use this parameter set unchanged in our simulations. The key physical parameters used in the model are summarised in [Table 21](#).

Parameter	Value	Unit
Mass	0.03	kg
Inertia (I_{xx}, I_{yy}, I_{zz})	$1.43 \times 10^{-5}, 1.43 \times 10^{-5}, 2.89 \times 10^{-5}$	kg·m ²
Arm length	0.043	m
Rotor count	4	--
Hover rotor speed	1788.53	RPM

Table 21: Key Crazyflie parameters used in the RotorPy model, taken from RotorPy’s built-in `crazyflie_params` [\[2\]](#).

The simulation also requires a flight controller that consumes the current state and returns the command inputs to the rotors. We use RotorPy’s built-in `SE3Control` controller.

4.2.3.2 Reference Trajectories

The Crazyflie is flown along a predefined reference trajectory in every simulation run. The simulated flight volume is bounded by x, y in $[-3, 3]$ m and z in $[0, 3]$ m.

RotorPy supports configurable wind profiles, but we disable wind for all runs by overwriting the profile with zero.

At the start of every simulation, the Crazyflie is placed at the origin with zero velocity and pointing upwards. The controller then drives the Crazyflie onto the configured reference trajectory.

We use the same flight volume across all simulations and only vary the trajectory between scenarios. The trajectories used are listed in [Table 22](#).

Trajectory	Description	Parameters
Circular	Horizontal circle at constant altitude	center at $[0, 0, 1.5]$ m, radius of 1.5 m
Circular (varying height)	Horizontal circle with altitude variation	center at $[0, 0, 1.5]$ m, horizontal radius of 1.5 m, vertical variation of 0.5 m
Circular (fast)	Horizontal circle at twice the normal speed	center at $[0, 0, 0.8]$ m, radius 1.5 m
Figure-eight	Two-dimensional Lissajous figure at constant altitude	amplitudes $A = B = 2$ m, frequencies $a = 0.5, b = 1.0$, phase $\delta = \pi/2$, altitude at 1.0 m
Vertical oscillation	Stationary in the horizontal plane with vertical oscillation	center at $[0, 0, 1.5]$ m, vertical variation of 1.0 m

Table 22: Reference trajectories used across the Crazyflie simulation scenarios.

4.2.3.3 Simulation Parameters and Frame Conventions

The simulation outputs data at a fixed sampling rate of 1000 Hz. A single simulated flight takes approximately one minute to compute on a standard laptop.

The simulator-internal state is stored in the `GroundTruth` contract defined in [Section 5.2.3](#), which fixes the frames and units for every field. RotorPy’s native output already uses the ENU frame and the expected units, with the only exception being the attitude. RotorPy outputs the attitude quaternion in the scalar-last convention $[x, y, z, w]$, so it is reordered to the scalar-first convention $[w, x, y, z]$. RotorPy does not model a GNSS receiver or a barometer, so these fields are filled with `NaN`.

4.2.3.4 Sensor Configuration

The Crazyflie carries a single instance of each sensor. The sensor rates, sources and their noise parameters are listed in the following two sections.

Sensor Rates and Sources

[Table 23](#) lists the sensors configured in the Crazyflie simulation together with their rates and origin. The origin distinguishes between sensors that RotorPy provides natively and sensors that are generated from the ground truth through our synthetic-sensor stage (see [Section 4.4](#)).

The simulation sampling rate of 1000 Hz is driven by the BMI088 IMU, which is the highest-rate sensor in the configuration. Aligning the simulation rate with the IMU rate allows the lower-rate sensors to be sampled directly from the same time grid without requiring interpolation.

Sensor	Hardware rate	Simulation rate	Source
Accelerometer	1000 Hz	1000 Hz	RotorPy native
Gyroscope	1000 Hz	1000 Hz	RotorPy native
ToF sensor	40 Hz	40 Hz	Synthetic
UWB transceiver	62.5 Hz	50 Hz	Synthetic

Table 23: Sensors configured in the Crazyflie simulation, including sampling rates and the origin of each signal.

Sensor Noise Parameters

[Table 24](#) lists the noise parameters applied to each configured sensor. Where available, the values are derived from the corresponding manufacturer datasheets. For the remaining parameters, we made and documented reasonable assumptions. The limitations of this approach are discussed in [Section 4.1.4.7](#).

For the synthetic ToF and UWB sensors, the noise is applied using our own noise model (see [Section 4.5](#)) after the synthetic signal has been generated from the ground truth. The IMU is configured to use RotorPy’s built-in noise rather than our injector, because the `SE3Control` controller must see the noisy signal directly (see [Section 4.2.4.5](#)).

The datasheet of the BMI088 IMU does not contain any random walk values [\[8\]](#). Therefore, we set the random walk to the same values as the LSM6DSO32 IMU on the sensor carrier, which are 0.2 mg for the accelerometer and 0.02 dps for the gyroscope.

Sensor	Noise density	Fixed bias	Random walk	Scale factor	Range
Accelerometer ⁶	160 $\mu\text{g} / \sqrt{\text{Hz}}$ (X/Y), 190 $\mu\text{g} / \sqrt{\text{Hz}}$ (Z)	± 20 mg	0.2 mg	$\times$	$\times$
Gyroscope ⁶	0.014 dps $/ \sqrt{\text{Hz}}$	± 1 dps	0.02 dps	$\times$	$\times$
ToF sensor	5 mm ⁷	± 20 mm	0	0	40 – 3600 mm
UWB transceiver	10 cm ⁷	0	0	0	0 – 10 m

Table 24: Noise parameters applied to the Crazyflie simulation sensors.

For the ToF and UWB sensors, we apply only white noise, together with a fixed bias for the ToF sensor. We omit the random walk and scale factor terms for two reasons. First, the corresponding datasheets do not specify values for these parameters. Second, our sensor error model is primarily aimed at inertial sensors, as it is inspired by the formulation in [\[13\]](#). As these sensors operate in a fundamentally different way to an IMU, these concepts cannot

⁶Noise is applied via RotorPy’s built-in noise model, which does not offer a scale factor or range parameter.

⁷The ToF sensor and UWB transceiver datasheets do not specify a noise density. The values listed in that column for these two sensors are accuracy figures rather than noise densities.

be easily transferred to them. We therefore made the simplification of modeling them with white noise only, rather than developing a separate noise model.

4.2.3.5 Simulation Scenarios

The configurations described in the previous sections define all aspects of a Crazyflie simulation. A simulation scenario now defines a concrete combination of these configurations that is run end-to-end and can be used for evaluation.

[Table 25](#) summarizes the scenarios used in this thesis.

Nr.	Trajectory	Dropouts	Noise	Purpose
1	Circular	None	$0.5 \times$ on all sensors	Low-noise baseline
2	Circular (varying height)	None	Nominal on all sensors	Tests position and altitude estimation under nominal flight
3	Circular (varying height)	None	Nominal on IMU and UWB, $1.5 \times$ on ToF	Evaluates altitude estimation under bad ToF
4	Circular (fast)	10% on UWB	$1.2 \times$ on all sensors	Stresses IMU under fast flights, elevated noise and some UWB dropout
5	Figure-eight	50% on UWB	Nominal on IMU and ToF, $1.5 \times$ on UWB	Evaluates horizontal estimation under bad UWB sensors
6	Vertical oscillation	None	$1.2 \times$ on IMU and ToF, nominal on UWB	Tests vertical estimation using higher IMU and ToF noise
7	Circular	None	Nominal on all sensors	Evaluation flight excluded from optimization data

Table 25: Simulation scenarios used in the Crazyflie evaluation.

The final scenario is excluded from the optimization dataset (see [Section 7.2](#)) and serves as a validation flight.

4.2.4 Limitations and Workarounds

RotorPy is a general-purpose multirotor simulator and does not reproduce every aspect of the Crazyflie platform used in this thesis. This section documents the limitations we encountered and the workarounds we applied.

4.2.4.1 Missing ToF Sensor

RotorPy does not provide a model for the ToF sensor on the Flow Deck v2. We therefore generate the ToF measurement synthetically from the simulated position and attitude. The mechanism is documented in [Section 4.4.1](#).

4.2.4.2 Missing Optical Flow Sensor

The PMW3901 optical flow sensor on the Flow Deck v2 is not modelled by RotorPy and is not synthetically generated in our pipeline. We do not consume optical flow sensor data in the EKF and therefore did not invest time for adding it to the simulation.

4.2.4.3 Missing UWB Sensor

RotorPy does not model the DWM1000 UWB sensor on the Loco Positioning Deck (see [Section 4.2.2](#)). We synthetically generate the UWB ranging measurements from the simulated position and the known anchor positions. The mechanism is documented in [Section 4.4.2](#).

The noise model used for the synthetic UWB ranges is configured with a single, fixed level of white noise, which we derived from the ranging accuracy reported in the DWM1000 datasheet [\[12\]](#). However, this is a simplification, as real UWB ranging is subject to additional error sources that our model does not capture. Our noise model is therefore not optimal for UWB ranging, but it provides a reasonable approximation. A more detailed noise model could be incorporated into the synthetic-sensor pipeline without major changes.

4.2.4.4 Missing Barometer

RotorPy does not provide a barometer sensor model. The Crazyflie does carry a barometer in hardware, but the Crazyflie EKF and our EKF do not currently use pressure for state estimation. We therefore did not generate a synthetic pressure output.

4.2.4.5 IMU Sensor Noise Injection

For the IMU, we use RotorPy's built-in noise injection rather than our own noise injector. This restricts us to the noise parameters that RotorPy exposes. The available parameters per channel are initial bias, noise density and random walk.

We accept this limitation because RotorPy's controller consumes the IMU measurements internally to stabilise the drone, so the noise must already be present in the signal the controller sees. Routing the IMU output through an external noise injector would mean that the controller operates on clean data while the EKF receives noisy data, which would not represent the behavior of the real Crazyflie.

The noise parameters are configured to match the BMI088 IMU on the Crazyflie. RotorPy outputs both the clean ground-truth signal and the noisy IMU measurement at each simulation step, which allows us to use the clean signal as ground truth for evaluation while feeding the noisy signal to the EKF. For the synthesised ToF and UWB sensors, the controller does not consume their output, so we apply noise externally through our noise injector.

4.3 Simulation Engines Credibility Analysis

We selected RocketPy and RotorPy as the simulation tools for the ASTERIA rocket and the Crazyflie (see [Section 4.1.2](#) and [Section 4.2.2](#)). Both tools are actively maintained open-source projects.

Because our EKF is evaluated against simulated data, the credibility of the evaluation depends directly on how well the simulation engines reproduce real flight behavior. This section summarizes the validation work published by the authors of each simulator and discusses what it implies for our use case.

4.3.1 RocketPy Credibility

The original RocketPy publication [\[35\]](#) compares RocketPy's predictions against measured flight data from two real rocket missions: The Bella Lui Kaltbrunn mission and the 2020 NASA Student Launch mission of the Notre Dame Rocketry Team (NDRT).

The authors report relative errors of 0.45% for the Bella Lui Kaltbrunn apogee altitude and -0.75% for the NDRT apogee altitude. Maximum-velocity errors are 4.24% and 2.31% and apogee-time errors stay within two percent. Throughout the ascent phase, the simulated altitude and velocity profiles closely match the measured data. Deviations during the descent phase are attributed to parachute oscillations that the simplified descent model does not capture.

The validation rockets are smaller than the ASTERIA rocket. Bella Lui Kaltbrunn reached an apogee of around 460 m and NDRT around 1320 m, compared to the approximately 9000 m apogee expected for ASTERIA. The authors note that future work should include validation against larger sounding rockets [35].

We therefore consider RocketPy’s ascent-phase predictions sufficiently credible for filter evaluation, while noting that the trajectory accuracy at the ASTERIA scale has not been evaluated yet.

The descent phase is only modeled with a 3-DOF model and thus less accurate, which we accept and also document in [Section 4.1.4.1](#).

4.3.2 RotorPy Credibility

The original RotorPy publication [2] validates the simulator against IMU data collected from a real Crazyflie performing tight circular maneuvers at speeds up to 2.5 m/s. The authors compare the simulated and measured accelerometer and angular-rate data and report good agreement, although no numerical error metrics are provided.

The validation covers the same hardware platform, a comparable trajectory type and the same speed range as our simulations, so it applies directly to our use case.

4.4 Synthetic Sensors

The simulators selected in [Section 4.1.2](#) and [Section 4.2.2](#) do not natively model every sensor required for this thesis. Some signals must therefore be generated outside the simulation engines.

These signals are generated in a dedicated stage of the simulation pipeline that takes the simulated ground truth as input and produces a clean synthetic sensor stream (see [Section 5.2.5](#)).

The remainder of this section describes the synthetic sensors produced during this thesis.

4.4.1 Synthetic ToF Sensor

The synthetic ToF sensor models the VL53L1x on the Flow Deck v2 of the Crazyflie. It produces an altitude signal by utilizing a laser measurement. The physical sensor has a 15° downward-pointing measurement cone and reports the smallest distance within that field of view.

The method we used to replicate this data can be seen in [Code Snippet 1](#). From the attitude quaternion, we extract the drone’s tilt angle. As long as the tilt stays within the half-cone of 7.5° , the distance equals the reported altitude of the drone. Beyond that threshold, the distance is calculated as $h / \cos(\theta - 7.5^\circ)$, where h is the altitude and θ is the tilt angle. The synthetic sensor then outputs the tilt-compensated altitude measurement.

The method used to generate this data is based on the Crazyflie EKF-implementation of the sensor [27].


```

1  ...
2  time = ground_truth.time[::step].copy()
3  altitudes = ground_truth.position[::step, 2].copy()
4  quaternions = ground_truth.attitude[::step].copy()
5
6  rotations = R.from_quat(quaternions, scalar_first=True)
7  r_zz = rotations.as_matrix()[2, 2] # Z-Z component of the rotation
8  r_zz_safe = np.clip(r_zz, -1.0, 1.0) # fix for floating point errors
9  drone_tilt_angle = np.arccos(r_zz_safe) # actual angle of the drone
10 tilt_raw = np.abs(drone_tilt_angle) - math.radians(7.5)
11 tilt = np.maximum(0.0, tilt_raw)
12 distances = altitudes / np.cos(tilt)
13 ...
14 return SensorStream(name=self.name,
15                     time=time,
16                     data=distances,
17                     channels=channels,
18                     unit=unit,
19                     sampling_rate=self.sampling_rate,
20                     origin=origin,
21                     processing_history=processing_history)

```

Code Snippet 1: Synthetic ToF sensor generation.

The implementation requires the simulation sampling rate to be a multiple of the sensor sampling rate. This holds for our configuration, which uses a 1000 Hz simulation rate and a 40 Hz sensor rate. Adding interpolation logic would be straightforward but is omitted during this thesis.

4.4.2 Synthetic UWB Sensor

The synthetic UWB sensor models a single DWM1000 transceiver on the Loco Positioning Deck of the Crazyflie. Each instance of the synthetic sensor represents one transceiver communicating with one fixed anchor at a known position. Multiple instances are configured per simulation to cover the four to eight anchors used in the indoor flight setup.

The sensor produces a five-channel stream at the configured sampling rate. The primary channel carries the Euclidean distance between the simulated Crazyflie position and the configured anchor position. The remaining four channels carry the anchor identifier and the anchor coordinates. We include the anchor coordinates in the stream to keep the EKF interface stateless and to allow anchors to be added or removed between simulation runs without changing the EKF configuration.

At each sub-sampled time step, the distance is computed as the L2 norm of the difference between the ground-truth position and the anchor position. The corresponding implementation is shown in [Code Snippet 2](#).


```

1  ...
2  step = int(ground_truth.simulation_sampling_rate / self.sampling_rate)
3  time = ground_truth.time[::step].copy()
4  channels = ["distance", "anchor_id", "anchor_x", "anchor_y", "anchor_z"]
5  positions = ground_truth.position[::step].copy() # (N, 3)
6  distances = np.linalg.norm(anchor_position - positions, axis=1) # (N,)
7  ...
8  return SensorStream(name=self.name,
9                      time=time,
10                     data=np.column_stack(
11                         [distances, anchor_ids,
12                          anchor_positions]),
13                     channels=channels,
14                     unit=unit,
15                     sampling_rate=self.sampling_rate,
16                     origin=origin,
17                     processing_history=processing_history)

```

Code Snippet 2: Synthetic UWB transceiver sensor generation.

As with the synthetic ToF sensor, the implementation requires the simulation sampling rate to be an integer multiple of the sensor sampling rate and omits interpolation. This is achieved by setting the simulated sensor rate to 50 Hz instead of the actual 62.5 Hz.

4.5 Noise Modeling

To evaluate the filter under realistic conditions, we developed a noise injector to apply a configurable noise model to the clean simulated sensor signals before they are passed to the EKF. We used our own noise injector wherever possible to have as much control over the data as possible.

4.5.1 Noise Profile

We describe the noise characteristics of each sensor through a noise profile. This profile is a small yaml-file that holds the parameters listed in [Table 26](#).

Parameter	Description
<code>noise_density</code>	White noise density of the sensor, derived from the sensor's datasheet.
<code>bias_fixed</code>	3σ bound for the fixed bias. The actual offset is sampled once per sensor and remains constant for the duration of the run.
<code>random_walk</code>	Standard deviation of the random walk applied on top of the fixed bias to model slow bias drift over time.
<code>scale_factor_error</code>	Multiplicative error that stretches or compresses the signal.
<code>range_min</code> , <code>range_max</code>	Hard clipping bounds applied to the signal, representing the physical measurement range of the sensor.
<code>resolution</code>	Optional step, representing the smallest signal change the sensor can output.

Table 26: Parameters of the noise profile that is created per sensor.

4.5.2 Sensor Error Model

Our sensor error model is inspired by the error characteristics described in [13], though it is not a direct implementation of the model presented there. Our sensor model applies the parameters of a noise profile to a clean signal in a fixed sequence of steps:

1. The signal is multiplied by $1 + \text{scale_factor_error}$ to introduce the scale factor error.
2. The fixed bias (sampled from a zero-mean Gaussian with 3σ bound `bias_fixed`) is added as a constant offset.
3. A random walk with per-step standard deviation $\text{random_walk} \cdot \sqrt{dt}$ is added to model the in-run bias variation, where dt is the sample interval.
4. White Gaussian noise with standard deviation $\text{noise_density} \cdot \sqrt{\text{ODR}/2}$ is added to each sample, where ODR is the sample rate of the sensor.
5. The signal is clipped to the configured range to model the physical measurement range of the sensor.
6. If a resolution is configured, the signal is rounded to the nearest value the sensor can output.

We do not attempt to reproduce the full set of error effects that Groves describes, such as cross-axis coupling or vibration-induced effects. This reduces the complexity of the noise model and keeps the model general enough to apply to all simulated sensors at the cost of omitting effects present on real sensors.

All random number generation in the noise pipeline uses NumPy's `default_rng` with explicit seeds, which makes the noise injection reproducible.

4.6 Dropout Modeling

Sensors like GNSS might drop out for an extended period of a rocket flight. The EKF must handle these conditions, which means the simulation pipeline has to be able to reproduce them in a controlled way. We model dropouts as a separate stage in our simulation pipeline (see [Section 5.2.5](#)).

4.6.1 Dropout Strategies

We model three strategies that represent the most common dropout patterns encountered in practice. They can be applied to any sensor stream and can also be combined.

The first strategy is **random dropout**, in which each sample is independently dropped with a fixed probability. This represents situations like occasional packet loss on a communication channel or sensor faults that are not tied to any specific time window.

The second strategy is **window dropout**, in which all samples within one or more configured time windows are dropped. This represents predictable loss patterns such as a GNSS outage caused by high acceleration or a deliberate fault injection to evaluate the filter's recovery behavior.

The third strategy is **external signal threshold dropout** in which a sample is dropped when an external reference signal exceeds a threshold at that timestamp. The reference signal is sampled at a different rate than the sensor and is therefore linearly interpolated onto the sensor's time grid before the mask is applied. This strategy is useful if for example GNSS samples should be dropped if the acceleration exceeds certain values.

4.6.2 Dropout Modes

Each dropout configuration also specifies what happens to a sample once the strategy decides to drop it. We implemented the three modes summarised in [Table 27](#).

Mode	Represented condition
NAN	The sensor explicitly signals an invalid measurement. The dropped sample is replaced by NaN. In this case, the EKF will get no update of the current sensor.
HOLD_LAST	The sample is replaced by the last valid measurement and reports it as if it were current. The EKF cannot distinguish this value from a correct one.
DELETE	The dropped sample is removed from the stream entirely. This is mainly the same as the NaN method and primarily an implementation topic.

Table 27: Dropout modes supported by the dropout implementation.

5 Architecture and Software Design

This chapter describes the software architecture of the components developed in this thesis and the design decisions made during their development.

[Section 5.1](#) describes the architecture of the EKF Rust library, which implements the filter. [Section 5.2](#) describes the simulation system that produces synthetic sensor data and ground truth for both platforms and [Section 5.3](#) describes the evaluation system that consumes this data, drives the EKF Rust library and computes performance metrics. It also includes the parameter search infrastructure used to tune the filter.

5.1 EKF Rust Library Architecture

This section describes the software architecture of the EKF Rust library and the design decisions made during its development. The mathematical model that the library implements is described separately in [Section 3](#).

5.1.1 Design Goals and Constraints

The use of Rust is a hard requirement from ARIS, as all software components in the ASTERIA firmware ecosystem are written in Rust ([MHR-12](#)). Beyond this constraint, Rust is well suited for the task: Its ownership model and type system eliminate whole classes of memory errors at compile time, it achieves C-comparable runtime performance and its ecosystem provides mature, `no_std`-compatible libraries for linear algebra (nalgebra) and language interoperability (PyO3, cbindgen).

The most constraining requirement is **`no_std` compatibility**: The library must run inside the ASTERIA rocket firmware on a bare-metal microcontroller without an operating system or heap allocator ([MHR-10](#)). This rules out dynamic memory allocation, dynamic dispatch via trait objects and all standard library facilities beyond the language core. Every buffer is sized at compile time and lives on the stack or in static memory.

The library targets three execution surfaces from a single crate:

- **Embedded (`no_std`)**: integrated as a `no_std` Rust crate directly into the ASTERIA firmware.
- **C bindings**: compiled as a static archive exposing a C API generated by cbindgen, activated by the `c_api` Cargo feature. Used by the Crazyflie C firmware ([MHR-11](#)).
- **Python**: compiled as a native extension module via PyO3, activated by the `python` Cargo feature. Consumed by the simulation and evaluation pipeline ([MHR-08](#)).

The filter struct `Ekf<T>` and all predictor and measurement types are generic over `T: RealField + Copy`. This allows the Crazyflie drone to use `f32` and the ASTERIA rocket and the simulation pipeline to use `f64`, without maintaining separate implementations.

5.1.2 Design Decisions and Trade-Offs

This section summarizes the key design decisions made during development of the EKF Rust library. Each decision is recorded as an Architectural Decision Record (ADR) and documented in [Section V](#). [Table 28](#) lists the decisions and links to the corresponding ADRs.

Decision	Summary	ADR
PyO3 for Rust-Python integration	The Python bindings are built with PyO3 and exposed as a native extension module. Runners in the evaluation pipeline are the only consumers of the PyO3-bound API.	ADR 1
Scalar updates over batched measurement matrix	All measurements are decomposed into a sequence of scalar updates. Each update uses a 1×9 Jacobian row and scalar division rather than a matrix inversion.	ADR 2
Trait-based static dispatch	All extension points use generic type parameters. No <code>dyn Trait</code> or heap allocation appears in the crate, making it compatible with <code>no_std</code> .	ADR 3
Macro-generated adaptive measurement wrappers	The <code>impl_adaptive_measurement!</code> macro generates newtype wrappers that add online noise estimation to any <code>Measurement</code> type, eliminating per-type boilerplate.	ADR 4
Stateful global C API facade	The C API holds a single library-owned <code>EkfState</code> initialized by <code>rkf_init</code> , bridging Rust's ownership model to C's stateless function interface without heap allocation on the caller side.	ADR 5
Feature-gated compilation for multi-target delivery	Cargo feature flags select the active build surface: <code>no_std</code> for the embedded target, <code>c_api</code> for the C bindings and <code>python</code> for the PyO3 module. A single crate produces all three outputs.	ADR 6

Table 28: Overview of the design decisions for the EKF Rust library architecture.

5.1.3 Component Overview

Figure 17 shows the structure of the EKF library. The crate is organized into six packages.

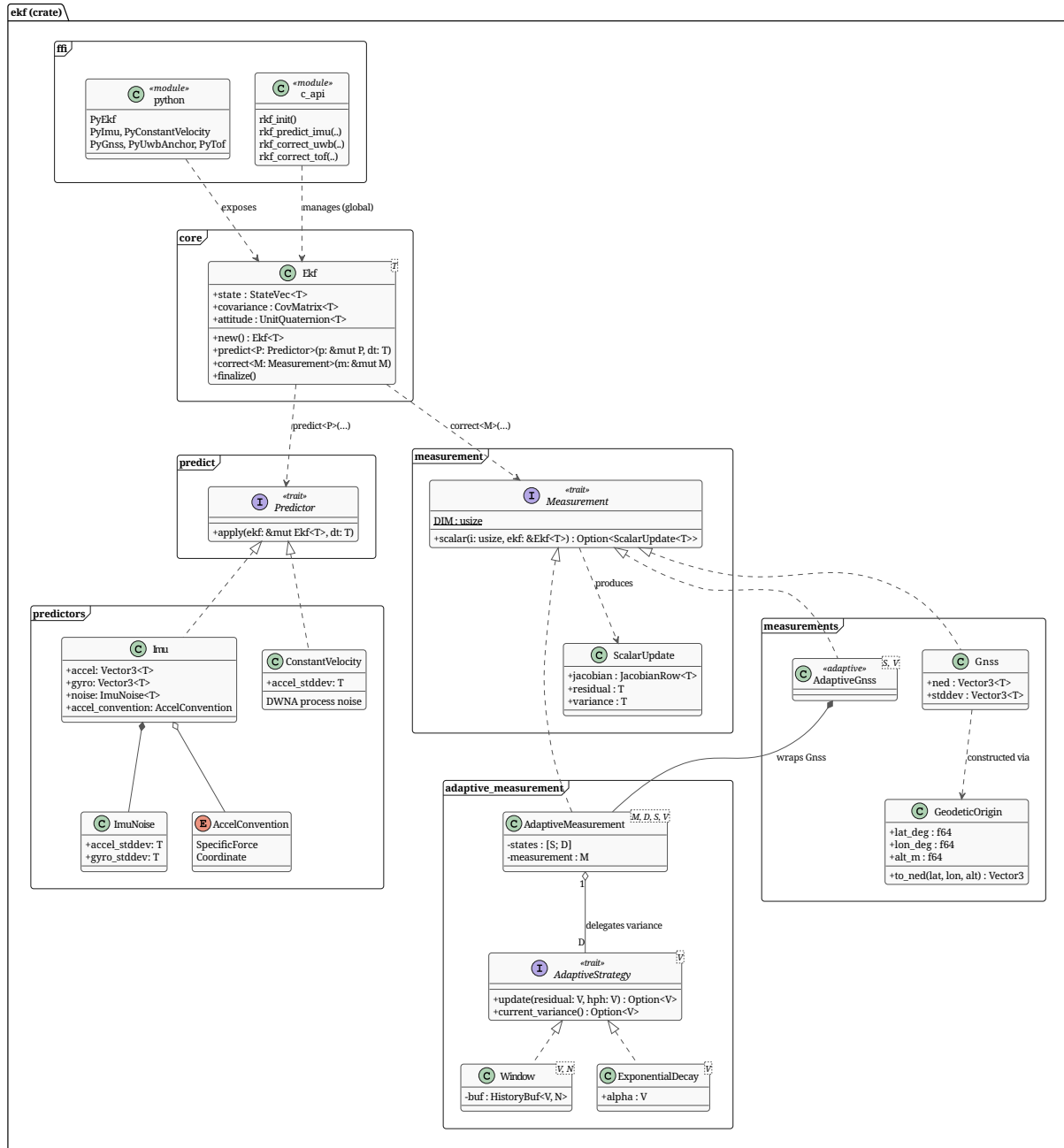


Figure 17: Library diagram of the EKF Rust library.

The **core** package contains the `Ekf<T>` struct which contains the EKF state and core algorithms. The two primary extension points are the `Predictor<T>` trait (in the **predict** package) and the `Measurement<T>` trait (in the **measurement** package). Concrete predictors live in the **predictors** package. Concrete measurement types live in the **measurements** package. The **adaptive_measurement** package provides a generic wrapper that adds online noise estimation to any `Measurement<T>` implementation. The **ffi** modules expose the Rust API to the Crazyflie C firmware (`c_api`) and the Python evaluation pipeline (`python`).

The library's integration boundary is narrow by design: Consumer code calls `predict`, `correct` and `finalize` on an `Ekf<T>` instance. The ASTERIA firmware consumes the library

directly as a `no_std` Rust crate. In the Crazyflie firmware the C API (`rkf_*` functions) is linked into the C firmware, which owns the global filter state (see [ADR 5](#)). In the simulation and evaluation pipeline the Python bindings expose the full API to the evaluation runners (see [ADR 1](#)).

The diagram shows `Gnss` as a representative example of a measurement implementation. [Table 29](#) lists all measurement types provided by the `measurements` package.

Type	DIM	Description	Adaptive variant
<code>Gnss</code>	3	NED position from a GNSS receiver (three scalar updates, one per axis)	<code>AdaptiveGnss</code>
<code>UwbAnchor</code>	1	Range to a UWB anchor at a known NED position	<code>AdaptiveUwbAnchor</code>
<code>Tof</code>	1	Height above ground from a ToF sensor, with tilt correction	<code>AdaptiveTof</code>

Table 29: Measurement implementations provided by the `measurements` package.

5.1.4 Core Abstractions and Type Design

`Ekf<T>` is the central struct holding the state vector, the covariance matrix and the nominal attitude quaternion. Named index constants (`IDX_N`, `IDX_E`, `IDX_D`, ...) identify the individual state slots across predictors and measurement types.

The two extension points are the `Predictor` and `Measurement` traits. `Predictor` exposes a single `apply` method called by `Ekf::predict`. `Measurement` requires a compile-time constant `DIM` and a `scalar(i, ekf)` method that returns one `ScalarUpdate` per dimension. Returning `Option` lets a measurement self-gate when data is invalid ([MHR-03](#)); the correction loop skips `None` returns. `ScalarUpdate` carries the Jacobian row, the measurement innovation and the measurement variance.

5.1.5 Adaptive Measurement

The adaptive measurement layer adds online noise estimation on top of any `Measurement<T>` implementation. The mathematical strategies, sliding window and EMA, are described in [Section 3](#). On the implementation side, `AdaptiveMeasurement<M, D, S, T>` wraps an inner `M: Measurement<T>` and holds `D` independent `AdaptiveStrategy<T>` instances, one per scalar dimension. Two concrete strategies implement `AdaptiveStrategy<T>: Window<T, N>` (stack-allocated ring buffer via `heapless::HistoryBuf`) and `ExponentialDecay<T>` (EMA with configurable `alpha`).

Because wrapping a measurement type requires identical boilerplate, the crate provides the `impl_adaptive_measurement!` macro (see [ADR 4](#)). A single invocation generates the newtype struct, `Window` and EMA constructors, named accessors and the `Measurement` delegation. Adding an adaptive variant for a new measurement type therefore reduces to the code shown in [Code Snippet 3](#).

```

RUST
1  impl_adaptive_measurement!(AdaptiveTof, Tof<V>, 1, tof, tof_mut);

```

Code Snippet 3: Implementation of an adaptive measurement type.

The arguments are, in order: The generated wrapper name, the inner measurement type, `DIM` and the names of the immutable and mutable accessor methods. The three adaptive wrapper types in [Table 29](#) are each produced by one such invocation.

5.1.6 Language Bindings

The library exposes its Rust API to two external consumers through feature-gated language bindings: A Python extension module built with PyO3 and a C static archive generated with `cbindgen`.

5.1.6.1 Python

The `python` feature (which implies `std`) activates a PyO3 extension module exposing all public filter types. Vectors and matrices cross the language boundary as NumPy `float64` arrays. Attitude is exposed as a `[w, x, y, z]` quaternion array. The Python binding is used by the evaluation runners. The design rationale for this integration approach is documented in [ADR 1](#) and the implementation of the bindings is further described in [Section 6.3.2](#).

5.1.6.2 C API

The `c_api` feature activates `c_api.rs`, which declares a C-compatible `RkfState` struct and `extern "C"` functions prefixed `rkf_`. The function set is intentionally narrow: `rkf_init`, a predict function and one correct function per sensor type. `cbindgen` generates the matching C header at build time via `build.rs`. The API runs the filter as `Ekf<f32>` to match the embedded Crazyflie's M4 with a single-precision FPU. A compile-time flag (`c_api_sliding_window`) selects between the EMA and sliding-window adaptive strategy. The stateful facade design is documented in [ADR 5](#).

5.1.7 no_std Support

Enabling the `no_std` feature adds `#![no_std]` at the crate root and activates the `libm` feature on both `nalgebra` and `map_3d`, replacing standard-library floating-point functions with implementations from the `libm` crate. The `heapless` crate provides `HistoryBuf<V, N>`, a fixed-capacity ring buffer with a const-generic size that lives entirely on the stack, used by the sliding-window adaptive strategy. No `dyn Trait` appears anywhere in the library: All polymorphism over predictors, measurements and adaptive strategies is resolved monomorphically at compile time, meaning no vtable overhead and no heap allocation for trait objects.

When building for the ASTERIA firmware, the crate is compiled with `--no-default-features --features no_std`. For the Crazyflie C API the additional `c_api` feature and `--crate-type=staticlib` are required.

5.1.8 Limitations

This section documents the known limitations and deliberate out-of-scope decisions of the EKF Rust library.

5.1.8.1 Runtime Configurability

Because all predictor and measurement types are resolved at compile time, the active filter configuration cannot be changed at runtime. Switching from constant-velocity to IMU-driven prediction or enabling adaptive noise on a particular measurement, requires recompilation. This is acceptable for static configurations where the hardware platform and sensors are known before compilation.

5.1.8.2 Sensor Calibration and Lever Arm Correction

The library operates on pre-processed sensor measurements and does not include sensor calibration or spatial corrections.

Sensor calibration, such as removing biases, scale-factor errors and misalignment from raw measurements, must be applied by the consumer before passing data to `correct`. Lever arm correction must similarly be applied externally. The lever arm is the fixed offset vector between the IMU's measurement center and the navigation reference point. Because the IMU is not co-located with that reference point, the angular velocity of the vehicle induces an additional linear velocity and acceleration at the IMU that is not present at the reference point. Correcting for this requires subtracting the cross-product and angular-acceleration terms from the raw IMU output before integration. The full derivation is given in [\[13, Section 2.5.5\]](#).

5.1.8.3 15-State Bias Extension

The 15-state bias extension identified as future work in [Section 3](#) would require additions throughout the crate: New state-index constants, updated Jacobians in both predictors and corresponding changes to the C and Python APIs.

5.2 Simulation Architecture

This section describes the software architecture of the simulation system and the design decisions that we made during development. The conceptual pipeline behind this architecture, the choice of simulation tools, the synthetic sensor models as well as the noise and dropout strategies are described separately in [Section 4](#).

[Figure 18](#) shows the system landscape. The simulation container produces synthetic sensor data and ground truth for both platforms. The evaluation container consumes this data, drives the EKF Rust library and computes performance metrics and plots. Treating simulation and evaluation as separate containers connected through a data contract allows the two subsystems to evolve independently and makes simulation runs reusable across multiple evaluation experiments.

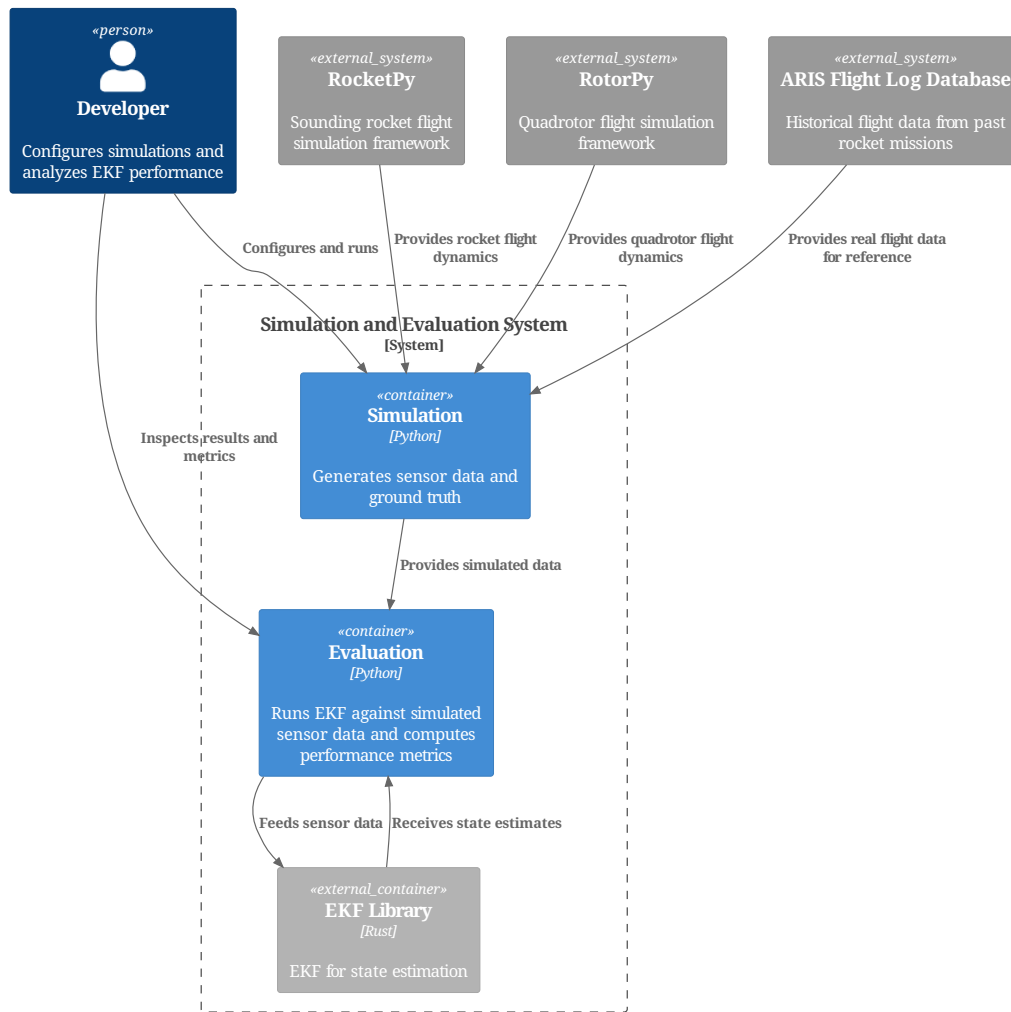


Figure 18: C4 system landscape of simulation and evaluation architecture.

This chapter focuses on the simulation container. The architecture and design decisions of the evaluation container are described in [Section 5.3](#).

5.2.1 Design Decisions and Trade-Offs

This section summarizes the design decisions that we made during the development of the simulation architecture described in the remainder of this chapter. Each decision is recorded

as an Architecture Decision Record (ADR) and documented in [Section V. Table 30](#) lists the decisions and links to the corresponding ADRs.

Decision	Summary	ADR
Unified pipeline with per-platform drivers	A single simulation pipeline with per-platform drivers that adapt simulator-specific outputs to a shared <code>SimulationResult</code> contract.	ADR 7
Builder pattern for simulation scenario object construction	Each platform exposes a builder dataclass that assembles the simulator-specific scenario objects from a small set of explicit parameters with meaningful defaults where possible.	ADR 8
Core Domain Model as simulation-evaluation contract	A set of simulator-agnostic dataclasses (<code>GroundTruth</code> , <code>SensorStream</code> , <code>SimulationResult</code>) that form the data contract between the simulation and evaluation containers.	ADR 9
Jupyter notebooks as the simulation and evaluation entry point	Simulations and evaluations are configured, executed and inspected from Jupyter notebooks, with shared logic moved into regular Python modules.	ADR 10
Pickle as primary serialization format for simulation runs	<code>SimulationResult</code> instances are persisted via pickle. The CSV+YAML export remains available but is no longer used in the main pipeline.	ADR 11
Strategy pattern for simulation pipeline modules	The synthetic sensor, noise and dropout stages each delegate to interchangeable strategy classes deriving from a stage-specific abstract base.	ADR 12

Table 30: Overview of the design decisions for the simulation architecture.

5.2.2 Component Overview

[Figure 19](#) shows the components of the simulation container and their dependencies on the external simulation engines. The architecture is built around a single data contract, the **Core Domain Model** (see [ADR 9](#)). Every component either produces, transforms, or consumes entities defined in the domain model.

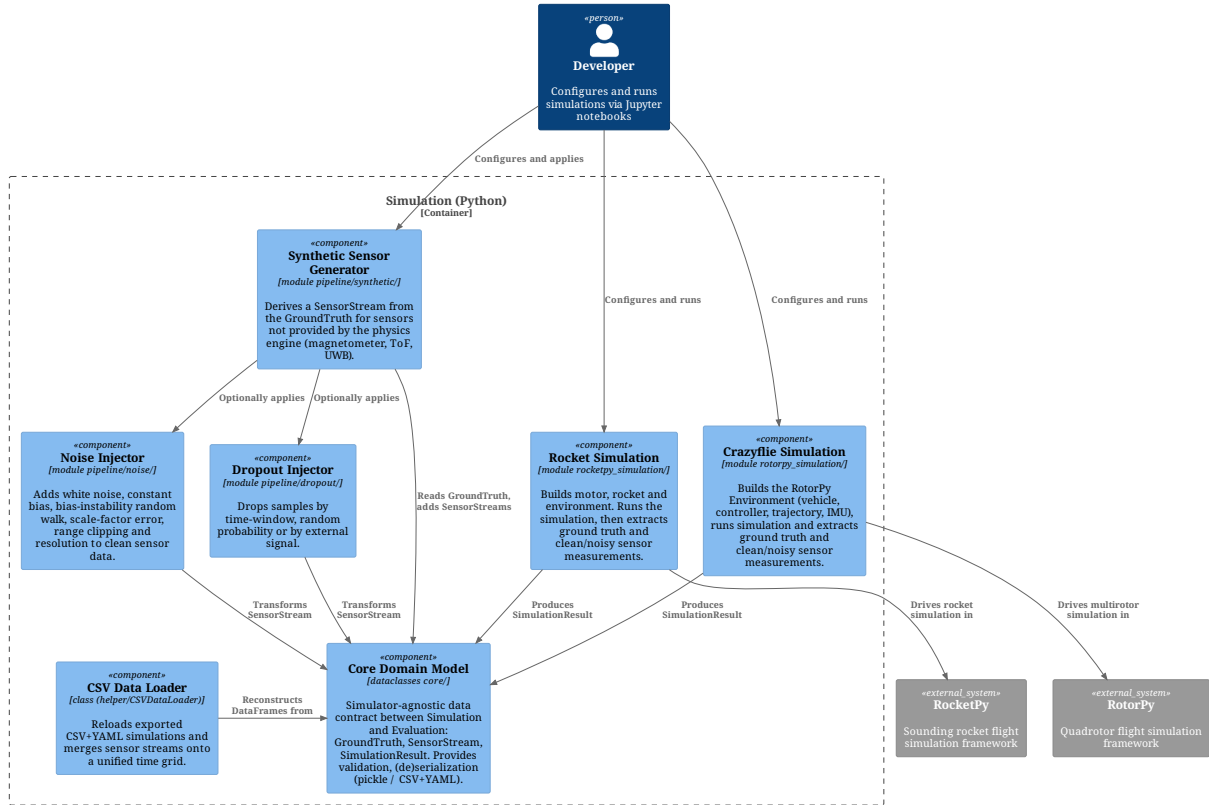


Figure 19: C4 component diagram of the simulation architecture.

The drivers (`RocketSimulation`, `CrazyflieSimulation`) wrap the external simulation engines and produce a `SimulationResult`. The pipeline stages (`SyntheticSensorGenerator`, `NoiseInjector`, `DropoutInjector`) post-process this result. A developer configures and drives the whole simulation from Jupyter notebooks (see [ADR 10](#)). All simulation components are described in the following sections.

5.2.3 Core Domain Model

The Core Domain Model is the data contract between the simulation driver and any downstream consumer (see [ADR 9](#)). It is implemented as Python dataclasses in the `core/` module. Nothing in the type definitions references `RocketPy`, `RotorPy`, or any platform-specific concept. This separation is what allows the same evaluation pipeline to consume data from either simulation engine. This also enables new simulation engines to be added at a later date without affecting the consumer side.

[Figure 20](#) shows the UML class diagram of the core dataclasses and how they work together.

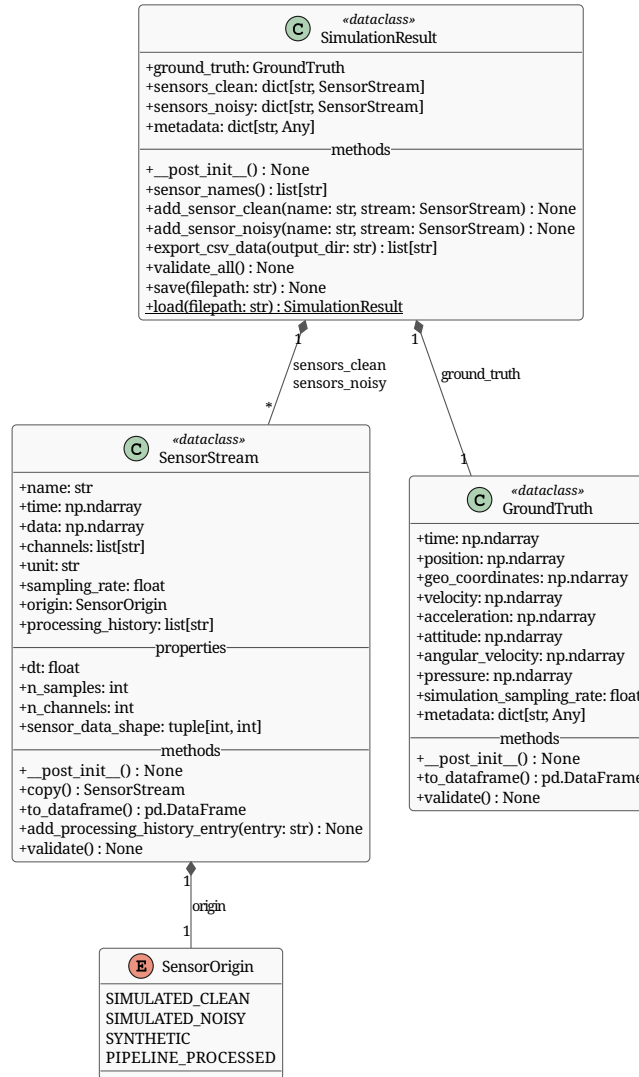


Figure 20: UML class diagram of the simulation core domain model.

The contract consists of the three primary dataclasses outlined in [Table 31](#).

Dataclass	Description
GroundTruth	Holds the time-indexed ground-truth state (ENU position, coordinates, velocity, acceleration, attitude, angular velocity and pressure) produced by the simulation engine. This is the data against which estimations are evaluated.
SensorStream	Holds the time-indexed measurements produced by a single sensor. Each stream carries its own sample timestamps, payload and metadata identifying the sensor. This architecture therefore allows different sensors to have different sampling rates.
SimulationResult	The top-level container returned by every simulation run. It holds a single GroundTruth with the set of SensorStream instances produced by the run. The metadata holds information on how the exact simulation run can be reproduced.

Table 31: Description of the three core domain model dataclasses.

`GroundTruth` defines the following shapes, units and frames. The `time` field is a one-dimensional array of timestamps in seconds and every other field shares the same first dimension. Position, velocity and acceleration are given in the East-North-Up (ENU) frame. `position` and `velocity` are stored in meter and meter per second. `acceleration` is stored in meter per second squared. `geo_coordinates` carries the position expressed as latitude and longitude in degrees (WGS84) together with altitude above sea level in meter. `attitude` is a scalar-first quaternion $[w, x, y, z]$ representing the rotation from the body frame to ENU and `angular_velocity` is given in the body frame in radians per second. `pressure` carries the barometric pressure in Pascals.

The domain model also contains the serialization logic. The primary persistence format of a simulation run is pickle (see [ADR 11](#)). Export to CSV+YAML is also supported but no longer used in the main pipeline.

Finally, each of the three dataclasses exposes a `validate` method that runs after producing a new instance. The validation logic is described in [Section 5.2.6](#).

5.2.4 Per-Platform Drivers

The two simulation drivers (`RocketSimulation` and `CrazyflieSimulation`) wrap the external simulation engines and adapt their outputs to produce an instance of the simulator-agnostic `SimulationResult` (see [ADR 7](#)). Their purpose is to be the only place in the codebase where simulator-specific APIs appear. Once a `SimulationResult` has been returned, the simulator that produced it is no longer visible to the rest of the pipeline.

Each driver is responsible for three things:

1. **Scenario construction:** The driver builds the simulator-specific object required to initialize the simulation. We utilize a builder pattern for instantiation of these objects (see [ADR 8](#)). This ensures loose coupling and allows for a fast change of individual components. For RocketPy this means building motor, rocket and environment objects and assembling them into a flight. For RotorPy this means instantiating the vehicle, the controller, the reference trajectory and the IMU model.
2. **Simulation execution:** The driver runs the simulation and incorporates all configurations that have been made.
3. **Output extraction:** The driver pulls ground-truth state out of the simulator, applies unit or coordinate frame transformations if needed and packages it into a `GroundTruth` instance. If the simulator has options to directly simulate sensors (RocketPy and RotorPy do have these features), this data is also pulled out into `SensorStream` instances. The result is wrapped in a `SimulationResult`.

Drivers only output sensor signals that the simulation engine produces natively. If the simulator has the possibility of generating sensor noise, the driver is allowed to capture it. However, drivers should not manually apply noise and dropout or generate synthetic sensors that are missing in the simulation. These tasks are delegated to the downstream pipeline stages (see [Section 5.2.5](#)).

The combination of separated concerns and a unified output contract makes the architecture well suited for adding new simulation platforms. Implementing a different system simply requires the creation of a new simulation driver.

5.2.5 Pipeline Stages

The pipeline stages are optional post-processing steps that transform a `SimulationResult` produced by a simulation driver. The stages are independent of one another and independent

of the driver that produced the data. Each stage takes a `SimulationResult` or a single `SensorStream` as input, transforms it and returns a new instance of the object.

The available stages and their intended order are shown in [Figure 21](#). The rationale for the strategy-pattern-based design of the different stages is documented in [ADR 12](#).

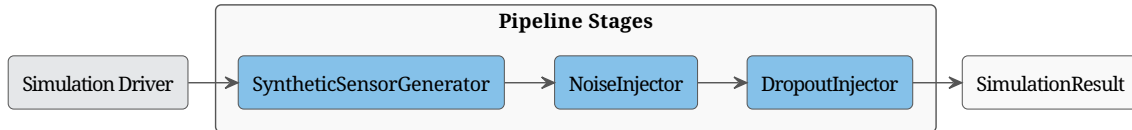


Figure 21: Flow chart showing data flow through the simulation pipeline.

5.2.5.1 SyntheticSensorGenerator

The `SyntheticSensorGenerator` can be used to generate sensor signals that the underlying simulation engines do not provide natively. Each synthetic sensor is computed using the values inside the `GroundTruth` object of the current run. The resulting `SensorStream` is added to the input `SimulationResult`.

A configuration for adding multiple sensors to an existing `SimulationResult` is shown in [Code Snippet 4](#)

```
PYTHON
1  synthetic_sensors = [
2      (SyntheticTimeOfFlightDistance(name="tof_distance"), None, None),
3      (SyntheticUWBTransceiver(name="uwb_transceiver_1"), None, None),
4      (SyntheticUWBTransceiver(name="uwb_transceiver_2"), None, None),
5      (SyntheticUWBTransceiver(name="uwb_transceiver_3"), None, None),
6      (SyntheticUWBTransceiver(name="uwb_transceiver_4"), None, None),
7  ]
8
9  generator = SyntheticSensorGenerator(sensors=synthetic_sensors)
10 generator.apply(result)
```

Code Snippet 4: Usage example of the `SyntheticSensorGenerator` class.

The logic for calculating the actual `SensorStream` values is written inside sensor-specific classes that all inherit from a common superclass. The `SyntheticSensorGenerator` optionally accepts a per-sensor noise and dropout config. This is functionally equivalent to chaining the `NoiseInjector` and `DropoutInjector` stages afterwards, but keeps per-sensor configuration in one place when many sensors with different profiles are configured at once. The implemented synthetic sensors can be found in [Section 4.4](#).

5.2.5.2 NoiseInjector

The `NoiseInjector` takes a sensor stream and applies the noise (as defined in [Section 4.5](#)) to it. The injector is parameterized per sensor, so each stream can receive its own noise profile derived from the corresponding datasheet or from testing.

Noise profiles are stored as YAML files, with one file per sensor. Adding a profile for a new sensor requires only adding a new YAML file in the noise profile directory. At runtime, the profile is loaded into a `NoiseProfile` instance, which is passed to the `NoiseInjector`. The injector then applies the configured noise components to the input stream and returns a new `SensorStream` containing the noisy measurements. [Code Snippet 5](#) shows a typical configuration.

PYTHON

```
1 noise_profile = NoiseProfile.from_yaml("noise.yaml")
2 noise_injector = NoiseInjector(noise_profile)
3
4 noisy_sensor = noise_injector.apply(clean_sensor)
```

Code Snippet 5: Usage example of the NoiseInjector class.

5.2.5.3 DropoutInjector

The `DropoutInjector` simulates temporary sensor unavailability. It supports three selection strategies (fixed time windows, random-probability per-sample dropout and external signal thresholds) and three replacement modes: NaN substitution, hold-last-value and deletion of the affected samples. The dropout strategies are further described in [Section 4.6](#) and an example of the usage is shown in [Code Snippet 6](#).

PYTHON

```
1 strategies = [WindowDropout([(0, 1), (2, 3)]),
2               RandomDropout(probability=0.1)]
3
4 dropout_injector = DropoutInjector(strategies, DropoutMode.NAN, 42)
5 sensor_faulty = dropout_injector.apply(sensor_clean)
```

Code Snippet 6: Usage example of the DropoutInjector class.

5.2.6 Data Validation

Each simulation run is validated at two levels: A programmatic check that runs automatically as part of the pipeline and a set of plots and summaries produced by the simulation engine that the developer inspects before exporting a run.

The programmatic check is the `validate` method implemented on the three core domain model dataclasses introduced in [Section 5.2.3](#). Each dataclass invokes its own `validate` method after initialization and `SimulationResult` additionally invokes `validate` on all of its contained objects before serialization. The method asserts that timestamps are monotonically increasing, that the dimensionality and shape of each field match the expected schema, that no field carries NaN values unless they were explicitly produced by the dropout injector and that recorded values lie within the expected range.

The additional visual validation covers properties that are difficult to express as assertions, such as whether the overall shape of a rocket trajectory is plausible. The simulation engines used by the drivers (RocketPy and RotorPy) provide diagnostic plots and summaries at the end of each run which can be inspected by a developer. Visual validation examples for both the rocket and the drone trajectory are shown in [Figure 22](#).

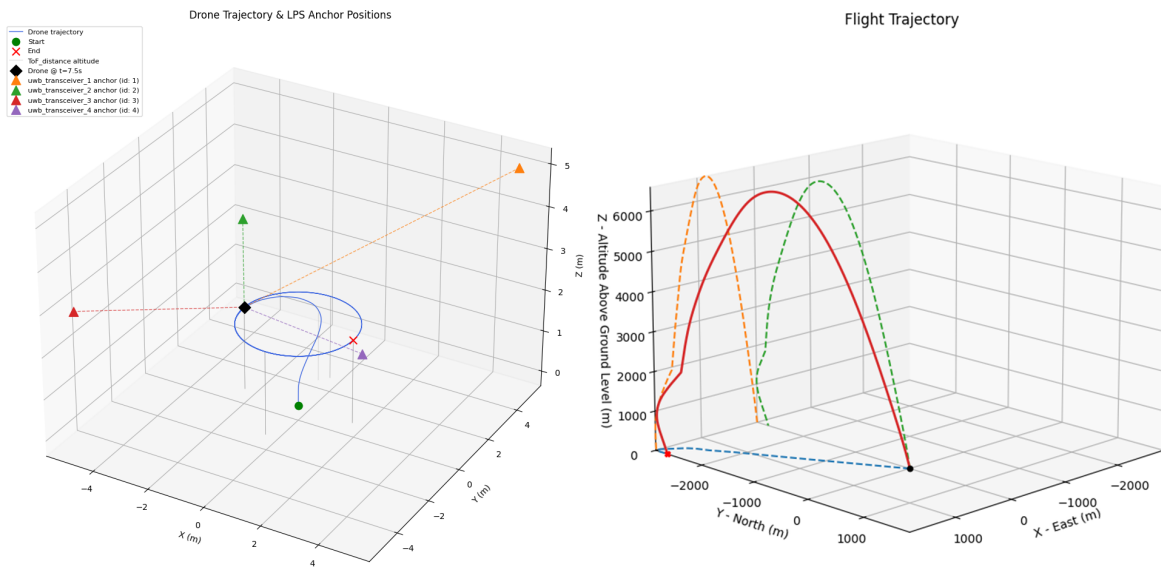


Figure 22: Visual validation plots of a representative drone trajectory (right) and a rocket trajectory (left), both showing plausible flight paths.

Additionally, unit tests cover the pipeline stages where applicable.

5.3 Evaluation Architecture

This section describes the software architecture of the evaluation system and the design decisions that we made during development.

The evaluation container consumes a `SimulationResult`, drives the EKF Rust library and evaluates the filter performance. It also provides functionality to search for optimal parameters. The two containers share the Core Domain Model introduced in [Section 5.2.3](#) as their primary data contract.

5.3.1 Design Decisions and Trade-Offs

This section summarizes the design decisions that we made during the development of the evaluation architecture described in the remainder of this chapter. Each decision is recorded as an Architecture Decision Record (ADR) and documented in [Section V.C](#). [Table 32](#) lists the decisions and links to the corresponding ADRs.

Decision	Summary	ADR
<code>KalmanEstimate</code> as the runner output contract	A runner-agnostic dataclass that decouples the EKF implementation from the metric, aligner and visualizer stages. Every runner produces an instance of this contract and downstream stages use it.	ADR 13
Strategy pattern for runners, metrics and aligners	Runners, metrics and time aligners each derive from a stage-specific abstract base class. Concrete strategies are passed into the <code>Evaluator</code> as constructor arguments and the orchestration code references only the abstract bases.	ADR 14
Separate time alignment stage with metrics on aligned data	A dedicated <code>TimeAligner</code> stage resamples estimate and ground truth onto a common time grid before the metric computations. Metrics operate on the aligned pair and perform no resampling themselves.	ADR 15
Bayesian parameter search with Optuna	The parameter search infrastructure uses Bayesian optimization via the Optuna framework.	ADR 16

Table 32: Overview of the design decisions for the evaluation architecture.

5.3.2 Component Overview

[Figure 23](#) shows the components of the evaluation container and their dependencies. The architecture is built around two data contracts: The Core Domain Model introduced in [Section 5.2.3](#) on the input side and a set of evaluation-specific dataclasses (`KalmanEstimate`, `MetricResult`, `EvaluationResult`) on the output side. Every component produces or consumes entities defined in one of these two contracts.

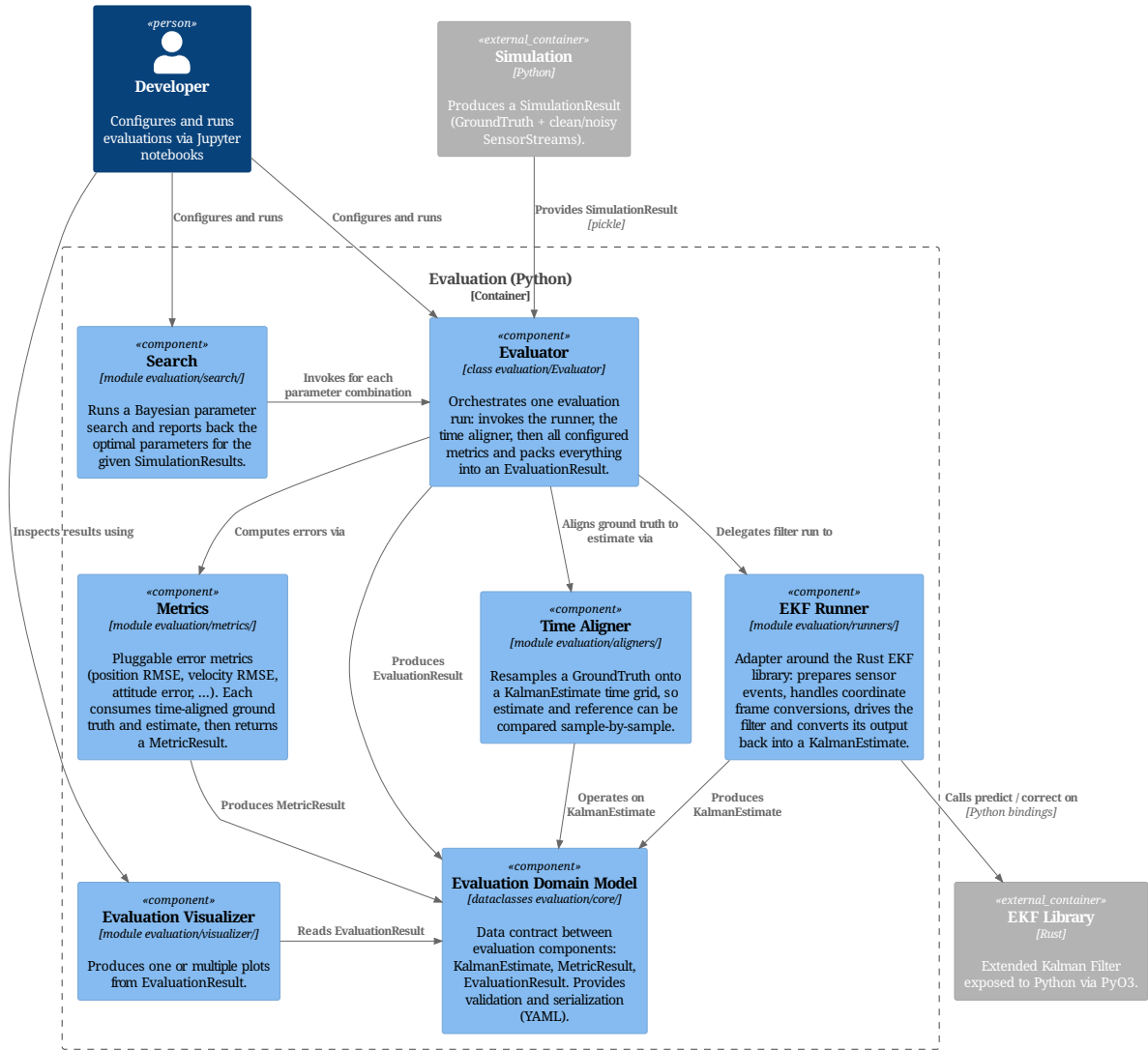


Figure 23: C4 component diagram of the evaluation architecture.

The `Evaluator` orchestrates a run by driving a `KalmanRunner`, a `TimeAligner` (Section 5.3.6) and a list of metrics. The `BayesianSearch` tunes parameters on top of it and the `EvaluationVisualizer` provides plots to visualize the results. As on the simulation side, a developer drives evaluation and parameter search from a Jupyter notebook (see ADR 10). All evaluation components are described in the following sections.

5.3.3 Evaluation Domain Model

This section documents the evaluation domain model and the persistence behavior of `EvaluationResult`.

Both `KalmanEstimate` and `EvaluationResult` are implemented in the `evaluation/core/` module. Like the Core Domain Model on the simulation side, they are simulator-agnostic and runner-agnostic. Nothing in their definitions references the EKF Rust library, RocketPy, RotorPy or any platform-specific concept. This is what allows the same metric and visualization stages to consume the output of any runner and what allows new runners to be added later without needing additional code changes.

Figure 24 shows the UML class diagram of the evaluation core dataclasses and their relationships. The two primary dataclasses are described in Table 33.

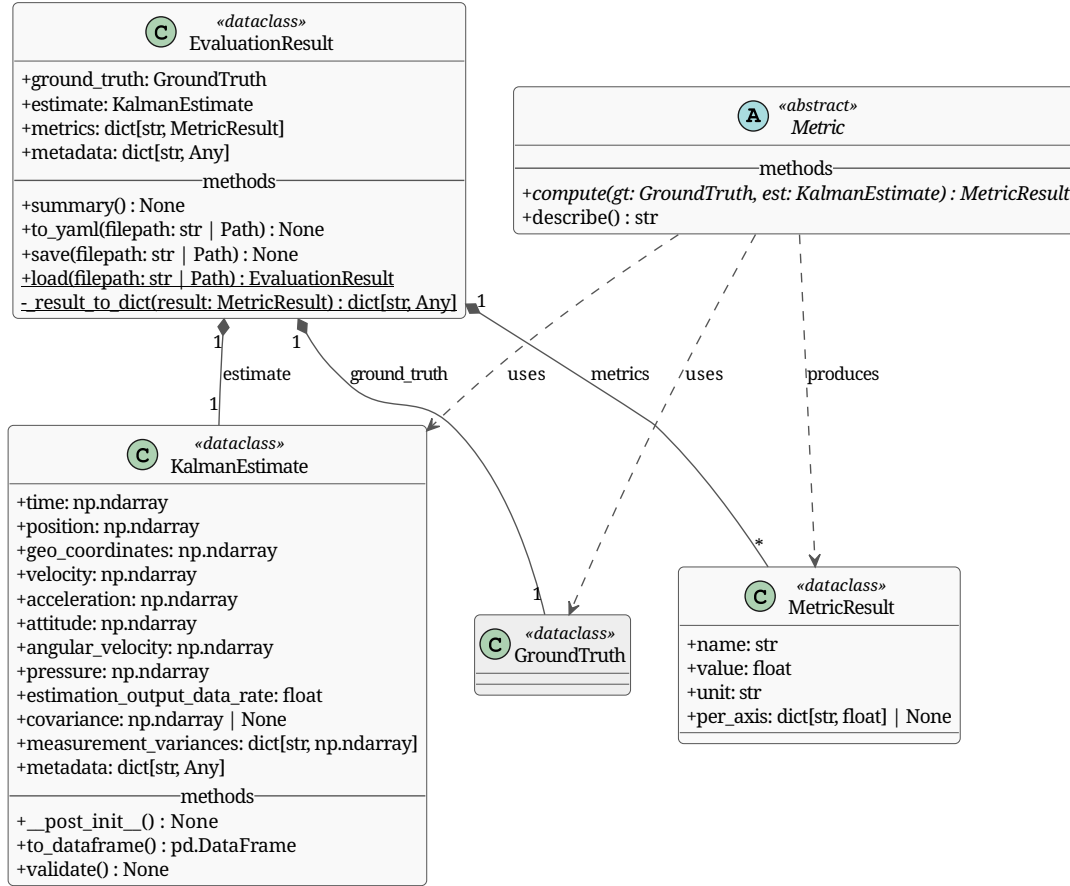


Figure 24: UML class diagram of the evaluation domain model.

Dataclass	Description
EvaluationResult	The top-level container returned by every evaluation run. It holds the <code>GroundTruth</code> from the simulation run, the <code>KalmanEstimate</code> produced by the runner and the dictionary of <code>MetricResult</code> instances produced by the configured metrics. The metadata records the runner, aligner, metric configuration and a timestamp so that an evaluation run can be reconstructed.
KalmanEstimate	Holds the time-indexed state estimate produced by a single EKF run (ENU position, coordinates, velocity, acceleration, attitude, angular velocity and pressure), together with an optional state covariance and metadata. All fields share the same first dimension. This is the data against which the ground truth is evaluated.

Table 33: Description of the two core output type dataclasses `KalmanEstimate` and `EvaluationResult`.

`KalmanEstimate` defines the same shapes, units and frames for its fields as the `GroundTruth` (see [Section 5.2.3](#)). Additionally, an optional `covariance` is present and has shape (N, d, d) , where d is the dimension of the EKF state vector. Like the core domain model dataclasses, `KalmanEstimate` exposes a `validate` method following the same validation checks described in [Section 5.2.6](#).

`EvaluationResult` is the persistable boundary of the evaluation container. This allows for an evaluation run to be inspected or visualized later without rerunning the EKF.

`MetricResult` is the third dataclass shown in [Figure 24](#) and is described together with the metric architecture in [Section 5.3.7](#).

5.3.4 Evaluator

The `Evaluator` is the orchestration class that ties a runner, an optional time aligner and a list of metrics into a single evaluation run. Its `evaluate` method is invoked with a `SimulationResult` and returns an `EvaluationResult`.

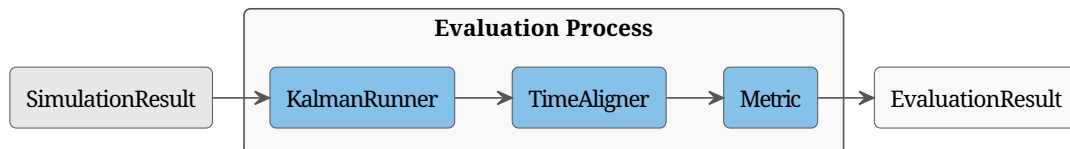


Figure 25: Flow chart showing intended evaluation process.

The `evaluate` method drives the different evaluation stages shown in [Figure 25](#):

1. The `KalmanRunner` is invoked on the simulation result to produce a `KalmanEstimate`.
2. The `TimeAligner` resamples the estimate and the ground truth onto a common time grid.
3. Each configured metric is invoked on the aligned pair to produce a `MetricResult`.
4. The estimate, the ground truth, the metrics and all metadata are packed into an `EvaluationResult` and returned.

5.3.5 Runners and EKF Integration

The runners wrap the EKF Rust library and adapt its API to produce an instance of the runner-agnostic `KalmanEstimate` (see [ADR 13](#)). Their purpose is to be the only place in the codebase where the EKF Rust library API appears.

All platform-specific code, all unit or frame conversions and all calls to the EKF Rust library live in the runners. Adding a new runner (for a different platform, a different filter implementation or a different sensor configuration) means writing a new subclass. No additional changes to the rest of the evaluation container are needed.

Two runners are implemented: `RustRunnerAsteria` for the rocket and `RustRunnerCrazyflie` for the drone. Both call the same EKF Rust library through PyO3 bindings (see [Section 6.3.1](#) and [ADR 1](#)). They differ in the sensors they feed into the filter and in the conversions applied to the sensor data.

Each runner is responsible for three things:

1. **Sensor extraction:** The runner pulls the relevant `SensorStream` instances out of the `SimulationResult`.
2. **Filter execution:** The runner drives the EKF using an event-loop. Sensor samples are merged into a single time-sorted event queue and at each event the runner either calls `predict` or `correct`.
3. **Output assembly:** The runner reads the filter state at each step, applies a coordinate frame and unit conversion if needed and creates a `KalmanEstimate`.

The noise parameters that the EKF requires to be tuned per platform are passed to the runner constructor. These parameters are primarily used by the search module described in [Section 5.3.8](#). Runners may expose additional constructor parameters that support quick configurations of the estimation.

In addition to the two production runners, a `PassthroughRunner` is provided for testing. It returns the ground truth of the `SimulationResult` as the `KalmanEstimate`, with optional Gaussian jitter and downsampling applied. Because no Rust code is involved, the passthrough runner allows the metric and aligner stages to be tested in isolation.

5.3.6 Time Alignment

The time alignment stage resamples a `KalmanEstimate` and a `GroundTruth` onto a common time grid before metric computation.

The EKF output rate and the ground-truth simulation rate generally differ. The EKF runs at the IMU rate, while the ground truth is sampled at the rate of the underlying simulation engine.

To minimize complexity during the metric calculation, the metric stage compares the two arrays element-wise. This requires the estimate and the ground truth to share a common time grid. A dedicated time alignment stage exists so that the resampling logic is decoupled from the metric computation (see [ADR 15](#)).

The abstract base class `TimeAligner` exposes a single method `align` that returns a new aligned pair that can be used for further evaluations. Concrete subclasses can implement different resampling strategies.

The strategy used for this work is implemented as a `NearestTimeIndexAligner`. It resamples the ground truth onto the time grid of the estimate using nearest-neighbour lookup based on the time column. The aligner computes the average time difference between paired samples and emits a warning if it exceeds one millisecond. This aligner works best when the IMU rate that determines the rate of the `KalmanEstimate` is a divisor of the simulation sampling rate. The `NearestTimeIndexAligner` is unit-tested in isolation using small synthetic estimate and ground-truth objects.

[Code Snippet 7](#) shows the usage of the implemented `NearestTimeIndexAligner`.

PYTHON

```
1 gt = simulation_result.ground_truth
2 aligner = NearestTimeIndexAligner()
3
4 estimate_aligned, gt_aligned = aligner.align(estimate, gt)
```

Code Snippet 7: Usage example of the `NearestTimeIndexAligner` class.

5.3.7 Metrics

The metrics compute performance numbers from a time-aligned estimate and ground truth. The abstract base class `Metric` exposes a single method `compute`. Concrete metrics assume that the inputs already share a common time grid. This separation keeps the metrics short and side-effect-free.

The output of every metric is a `MetricResult`. It carries the metric `name`, the scalar `value`, the `unit` in which the value is expressed and an optional `per_axis` dictionary that breaks the value down by axis where applicable.

The metrics implemented and used during this thesis are described in [Section 7.1.1](#).

5.3.8 Parameter Search

The search module provides functionality to optimize the EKF process and measurement noise parameters. The methodology behind the actual tuning is described in [Section 7.2](#). This section describes the architecture that enables the programmatic optimization.

Manual grid search across this many parameters is expensive and quickly becomes impractical. Thus, we used the hyperparameter optimization framework “optuna” [\[40\]](#). Optuna uses a TPE (Tree-structured Parzen Estimator) by default, which is a Bayesian optimization method. Parameters get sampled based on their observed performance, which allows for a more efficient exploration than grid search (see [ADR 16](#)).

The class `BayesianSearch` wraps the Optuna optimization framework [\[40\]](#). The required constructor arguments are listed in [Table 34](#).

Argument	Description
<code>runner_factory</code>	A function that constructs a runner from the parameter dictionary suggested by Optuna. This logic keeps the search module compatible with runner-specific construction logic.
<code>search_params</code>	List the parameters that should be searched. These are built from <code>FloatParam</code> , <code>IntParam</code> and <code>CategoricalParam</code> dataclasses.
<code>metrics</code>	The metrics to compute on every trial. All metrics are evaluated even when only some of them appear in the objective.
<code>objective</code>	Function that combines the metric results of a trial into the optimization target.
<code>aligner</code>	The time aligner used in each trial. If omitted, the estimate and ground truth must already share a common time grid.
<code>n_trials</code>	Number of Optuna trials to run. Each trial samples one parameter combination and evaluates it against every simulation in the input list.
<code>directions</code>	Either “minimize” or “maximize” for single-objective runs, or a list of these strings for multi-objective runs.

Table 34: Constructor arguments of the `BayesianSearch` class.

The actual search proceeds trial by trial. At each trial, `SearchParams.suggest` produces a parameter dictionary, the `runner_factory` constructs a runner from it and the runner is evaluated against every simulation in the input list using an instance of `Evaluator`. The metric results across all simulations are averaged per metric and the resulting averaged dictionary is passed to the `objective` to produce the value that Optuna uses to drive the next trial. Averaging per trial across simulations is what makes the tuned parameters generalize rather than overfit to a single trajectory. Failed evaluations (for example, an EKF that diverges on a particular configuration) return infinity for the objective and are logged but do not abort the search.

The result of the optimization is captured in a `SearchResult` object. Internally it stores all trials, their parameters and the recorded metrics. The `SearchResult` object provides methods to create a dataframe from the results as well as to find the optimal parameter combination for a given metric.

Multi-objective runs are also supported. For multi-objective runs, the `SearchResult` only supports exposing the parameter combinations and their metric results as dataframe, since the developer must resolve the trade-off between the different objectives manually.

[Code Snippet 8](#) shows a minimal configuration of `BayesianSearch`.

```
PYTHON
1 runner_factory = lambda p: RustRunnerAsteria(
2     accel_stddev=p["accel_stddev"],
3     gyro_stddev=p["gyro_stddev"])
4 search_params = SearchParams(
5     accel_stddev=FloatParam(0.01, 1.0, log=True),
6     gyro_stddev=FloatParam(0.001, 0.1, log=True),
7 )
8 metrics = [PositionRMSE()]
9 objective = lambda m: m["position_rmse"].value
10 aligner = NearestTimeIndexAligner()
11 n_trials = 100
12 directions = "minimize"
13 study_name = "ASTERIA position optimization"
14
15 search = BayesianSearch(runner_factory, search_params, metrics,
16     objective, aligner, n_trials, directions, study_name)
17 result = search.run(simulations)
```

Code Snippet 8: Usage example of class `BayesianSearch`, optimizing for position RMSE.

5.3.9 Visualizer

The `EvaluationVisualizer` takes an `EvaluationResult` in its constructor and its methods produce matplotlib plots from it. It is used to unify the appearance of plots produced during evaluation.

The visualizer is decoupled from the rest of the pipeline. An `EvaluationResult` can be persisted to disk, reloaded later and visualized without rerunning the EKE.

6 Software Integration

This chapter describes how the EKF Rust library is integrated into the two embedded firmware targets and the Python evaluation pipeline.

[Section 6.1](#) describes the integration into the ASTERIA firmware. [Section 6.2](#) describes the integration into the Crazyflie firmware, which consumes the library through its C API. [Section 6.3](#) describes the integration into the evaluation pipeline, which drives the library from Python through PyO3 bindings.

6.1 ARIS Avionics Firmware Integration

This section describes how the EKF library is integrated into the ASTERIA sensor carrier firmware as a native Rust crate ([MHR-10](#)).

6.1.1 ASTERIA Firmware Repository

The firmware for all ASTERIA embedded boards is collected in the repository `asteria-firmware`. All firmware targets are written in `#![no_std]` Rust and use the embassy framework for hardware abstraction and async task management. `just` is the primary build orchestrator.

The repository consists of several independent Cargo workspaces, as listed in [Table 35](#).

Directory	Contents
<code>fw-*/</code>	Per-board firmware workspace with the binary in the <code>app/</code> folder
<code>ba-implementation</code>	Git submodule containing the EKF library
<code>crates/</code>	Shared embedded libraries and device drivers
<code>tools/</code>	Host-side CLI tooling (asteria-tool)
<code>just/</code>	Shared automation scripts

Table 35: The contents of the asteria-firmware repository.

We introduced the EKF library as a Git submodule in the `ba-implementation` folder and integrated it into the `fw-sensor-carrier-v3` folder for the sensor carrier firmware. This firmware is the planned successor to the ARIS sensor carrier board and is currently under active development by the ASTERIA team. The integration work described in the following sections is being carried out on the `feature/integrate-ekf` branch of the firmware repository.

6.1.2 Sensor Carrier Firmware Architecture

`fw-sensor-carrier-v3` runs on the Embassy async runtime on an STM32H723 microcontroller. The STM32H723 contains an ARM Cortex-M7 core with a double-precision floating-point unit.

Embassy supports two types of executor: An interrupt executor and a thread executor. An interrupt executor is driven by a hardware timer interrupt and runs at a configurable NVIC priority level. Since the microcontroller preempts lower-priority interrupt handlers when a higher-priority interrupt occurs, a higher-priority interrupt executor will always take over

from a lower-priority one mid-task if necessary. The thread executor runs in the main thread at the lowest possible priority and can be interrupted by all interrupt executors.

This hierarchy is used by the firmware to prevent long-running EKF prediction steps from delaying sensor readout. Time-critical sensor readout tasks and the inertial processing task run in a high-priority interrupt executor (TIM2, P6). We introduced a lower-priority interrupt executor (TIM3, P7) for our state estimation task, which the sensor executor can preempt at any point. Non-time-critical tasks, such as the buzzer, storage, SD writer and SD data producers, run in the thread executor.

Inter-task communication uses two Embassy primitives. High-throughput sensor streams use the `PubSubChannel`, which buffers a configurable number of slots and supports multiple concurrent subscribers without blocking producers. Time-critical or state-only data uses the `Watch` primitive, which stores only the most recent value and discards older ones, making it suitable for outputs like the EKF state, where only the latest estimate matters.

6.1.3 Sensor Data Pipeline

Each sensor operates within a dedicated readout task. Raw samples are published to typed channels and then consumed by the inertial processing and state estimation tasks downstream.

The IMU readout task communicates with the LSM6DSO32 over SPI and uses the sensor's hardware FIFO with a watermark of 26 entries. An EXTI interrupt fires when the watermark is reached and the task drains all accumulated entries in a single burst. Sample timestamps within each burst are computed by linear interpolation between consecutive drain edges.

The GNSS readout task receives UBX protocol frames from the GNSS module over UART, parsed with the `ublox` crate. Each fix is backdated by approximately 100 ms to compensate for UART transmission and parser latency before it is forwarded downstream.

The inertial processing task involves applying per-axis bias correction and frame alignment between the raw IMU readout and the EKF. Currently, the firmware is unable to store calibration data in non-volatile memory, meaning that the calibration must be repeated each time the system starts up. For this reason, we have introduced a stationary calibration pass for each IMU at startup, as described in [Section 6.1.4](#).

[Figure 26](#) shows the task diagram of the sensor carrier firmware, from sensor readout through calibration and state estimation to the SD card log. It shows the executor levels with the high priority at the bottom and the low priority at the top.

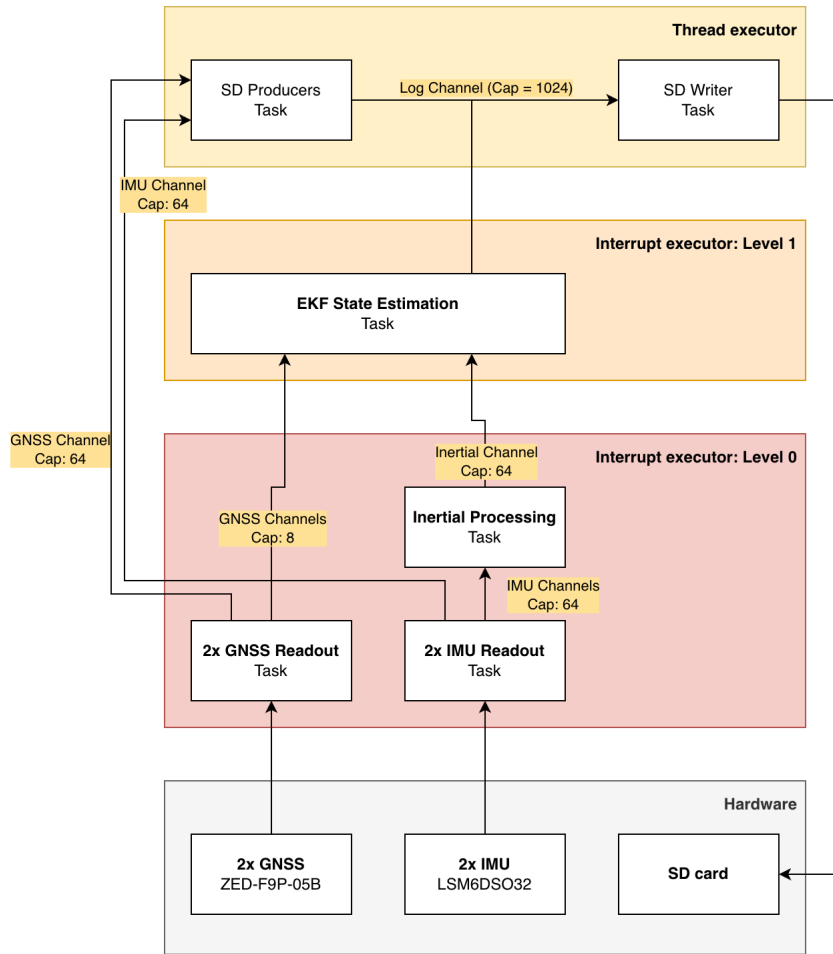


Figure 26: Task diagram in fw-sensor-carrier-v3, from sensor hardware through calibration and state estimation to the SD card log.

6.1.4 IMU Calibration and Initialization

We have implemented an IMU calibration procedure, which is described in more detail in this section. Before the EKF can produce valid state estimates, the IMU must complete a stationary calibration procedure. The inertial processing task runs this procedure independently for both IMUs at startup.

Following a 30-second warm-up period, the task collects 32,768 consecutive samples from each IMU. The mean of the accelerometer and gyroscope readings on each axis during this period is used to estimate the bias and is stored for subtraction from all subsequent measurements. Once the estimation of the bias is complete, the task applies the mount rotation matrix loaded from the already existing `params/mount/` to align the sensor frame with the board frame. This is followed by any stored fine-rotation corrections.

Once the `IMU_0` calibration process is complete, the firmware raises the `CALIBRATION_DONE` signal. The buzzer task then responds with two 2.5 kHz beeps to confirm this. The EKF cannot run until this step has been completed, since prediction requires calibrated inertial data.

The EKF attitude state is initialised using the identity quaternion. It would be better to initialise using the specific force measured during standstill to infer initial pitch and roll, but we had to leave this out due to time constraints.

6.1.5 EKF State Estimation

We introduced a state estimation task, which serves as our primary integration point for the EKF library in `fw-sensor-carrier-v3`.

NED frame anchoring: Before the first GNSS correction can be applied, the EKF must establish its local NED coordinate origin. The task waits for the first valid GNSS fix with at least five satellites. On this event it initialises a `GeodeticOrigin` from the reported geodetic coordinates, raises the `FIRST_GNSS_FIX` signal and the buzzer emits three beeps at 3.5 kHz. GNSS corrections are suppressed until the NED origin is established.

Prediction: The task subscribes to `INERTIAL_CHANNELS[IMU_0]` and processes every incoming calibrated sample at 833 Hz with an IMU predict step.

After each predict step, the current `StateEstimate` is published to `STATE_ESTIMATE_WATCH`. The predict-step duration is measured via the DWT cycle counter and written to the SD card log.

GNSS correction: The task subscribes to `GNSS_CHANNELS[GNSS_0]` and applies a 3D position correction using the `Gnss` measurement type on each incoming fix. The measurement noise variance is selected at compile time from three modes: fixed, sliding window adaptive and EMA adaptive.

The correct-step duration, measurement innovations and innovation covariances are logged to the SD card on each correction.

The `StateEstimate` struct carrying the filter output is defined as shown in [Code Snippet 9](#).

```

RUST                                                                 measurements.rs
1  pub struct StateEstimate {
2      pub ts: Instant,
3      pub pos_ned_m: [f64; 3],
4      pub vel_ned_mps: [f64; 3],
5      pub attitude_quat_wxyz: [f64; 4],
6      pub cov_diag: [f64; 9],
7  }
```

Code Snippet 9: `StateEstimate` struct published to `STATE_ESTIMATE_WATCH` after each EKF predict step.

6.1.6 Dual-IMU Fusion

The sensor carrier board has two IMUs integrated into the firmware. Currently, we only use `IMU_0` due to the relative offset of the two IMUs on the board. `IMU_1` is calibrated at startup and its data flows through the inertial processing task but is not used any further.

A dedicated IMU fusion task could subscribe to both `INERTIAL_CHANNELS[IMU_0]` and `INERTIAL_CHANNELS[IMU_1]`, apply a consistency gate on each incoming pair of samples and publish a single fused sample to a new `FUSED_INERTIAL_CHANNEL`. The state estimation task would then subscribe to `FUSED_INERTIAL_CHANNEL` instead of `INERTIAL_CHANNELS[IMU_0]`, with no other changes required downstream.

6.1.7 Dual-GNSS Selection

The sensor carrier board has two GNSS receivers, but the current firmware only forwards fixes from `GNSS_0` to the EKF. Data from `GNSS_1` is logged to the SD card, but is not used for correction.

Each GNSS fix already includes receiver-reported horizontal and vertical accuracy figures. These can drive a simple selection policy: On each incoming fix from either receiver, a GNSS arbiter task compares the horizontal accuracy figures and forwards only the fix from the receiver reporting better accuracy to a new `SELECTED_GNSS_CHANNEL`. The state estimation task would then subscribe to `SELECTED_GNSS_CHANNEL` instead of `GNSS_CHANNELS[GNSS_0]`, with no other changes required downstream. This provides passive redundancy when one receiver loses signal quality without requiring any additional filtering.

6.1.8 SD Card Binary Logging

For our evaluation purposes, we implemented a task that logs raw sensor data and EKF outputs to an SD card in a compact binary format designed to minimise write latency and prevent stalling of the sensor data pipeline. We also enabled a DWT cycle counter at startup to enable microsecond-resolution profiling of individual EKF steps.

A single SD writer task manages a FAT filesystem on the first partition of the SD card using the `embedded-fatfs` library. Log files are named sequentially (`LOG00000.BIN`). Each file begins with an 8-byte magic header `b"SCV3LOG\x01"`.

Individual records are written in little-endian byte order and each starts with a two-byte sync sequence (`0xA5 0x5A`) followed by a one-byte tag. [Table 36](#) lists the four record types.

Tag	Name	Key fields	Size
0x01	IMU	<code>sensor_id</code> , <code>ts_us</code> , <code>accel_xyz</code> ($g \times 3$), <code>gyro_xyz</code> ($dps \times 3$)	24 B
0x02	GNSS	<code>sensor_id</code> , <code>ts_us</code> , lat, lon, fix type, satellite count, heights, velocities, accuracies	77 B
0x03	EKF state	<code>ts_us</code> , pos NED $\times 3$, vel NED $\times 3$, quaternion $\times 4$, covariance diagonal $\times 9$, <code>predict_us</code>	88 B
0x04	GNSS correction	<code>ts_us</code> , residual $\times 3$ (m), innovation covariance $\times 3$ (m^2), adapted variance $\times 3$ (m^2), <code>correct_us</code>	56 B

Table 36: Binary log record types written to the SD card.

High-rate sensor data is fed to the SD writer via dedicated producer tasks, which use non-blocking ‘try_send’ calls to access a shared queue with a capacity of 1024 entries. Entries that are dropped are counted in a `DROPPED` counter. EKF state and GNSS correction records are written directly from the state estimation task at every prediction and correction stage.

The writer accumulates records in a 16 KB buffer. A flush to disk is triggered when the buffer exceeds 8 KB, after one second of continuous writing or after 500 ms of inactivity. Writes are aligned to 512 byte sectors to avoid read-modify-write cycles and keep write latency predictable.

Log files can be decoded offline with the Python script `scripts/decode_log.py`, which produces four CSV files: `imu.csv`, `gnss.csv`, `ekf.csv` and `gnss_corr.csv`. To produce plots and statistics, the `ba-implementation` repository contains the Jupyter notebook `asteria-evaluation/evaluation.ipynb`, which decodes the binaries and generates figures similar to those in [Section 7](#).

6.1.9 Enabling the Double-Precision FPU

The double-precision FPU is required to produce state estimates in real time when using the EKF with f64 ([MHR-04](#)). To target the ARM Cortex-M7 with a double-precision FPU using Rust, the `thumbv7em-none-eabihf` target triple must be installed via `rustup` and configured in the workspace's `.cargo/config.toml` alongside the `target-cpu=cortex-m7` compiler flag:

```
TOML                                                                    .cargo/config.toml
1  [target.thumbv7em-none-eabihf]
2  rustflags = [
3      "-C",
4      "target-cpu=cortex-m7",
5  ]
6
7  [build]
8  target = "thumbv7em-none-eabihf"
```

Code Snippet 10: Cargo configuration for the ARM Cortex-M7 with double-precision FPU.

We identified that the ARIS team had omitted the required `target-cpu=cortex-m7` parameter and supplied the correct value.

6.1.10 CAN Bus

At the time of writing, CAN communication has not been implemented. Once the ASTERIA team has implemented a CAN task, the team can subscribe to the EKF output via `STATE_ESTIMATE_WATCH` and forward each estimate over the bus.

6.2 Crazyflie Integration

This section describes how the Rust EKF library is integrated into the Crazyflie firmware (MHR-11) via a C API exposed by the Rust library (MHR-13). We describe the firmware interfaces an estimator must implement and cover the build system setup, the C API and the estimator wrapper that connects the filter to the firmware.

6.2.1 Crazyflie Estimator Interface

The Crazyflie firmware defines two contracts that every estimator must fulfil: The estimator function table, which the firmware calls to drive the estimator lifecycle and to tick it on every stabilizer step and the sensor measurement queue, through which sensor drivers deliver measurements to the active estimator.

6.2.1.1 Estimator Function Table

The `EstimatorFcns` struct defines the interface that each estimator backend must implement. The following sections describe each field and its role in the integration.

`init`

The `init` function takes no parameters and returns nothing. It is called once when the estimator is selected and is responsible for initializing the EKF state vector, covariance matrices and any internal buffers to their default values.

`deinit`

The `deinit` function takes no parameters and returns nothing. It is called when switching away from the estimator.

`test`

The `test` function takes no parameters and returns a `bool`. It is called once during the firmware self-test phase to verify that the estimator is fit for use. `true` clears the estimator for flight, while `false` signals a fault that prevents operation.

`update`

The `update` function receives a pointer to `state_t` and the current stabilizer tick count as a `stabilizerStep_t`, which increments at 1000 Hz. It is called once per stabilizer tick and is responsible for processing all pending sensor measurements and writing the resulting estimates back into the `state_t` struct. The `state_t` struct contains the following fields:

- `attitude`: roll, pitch, yaw in degrees
- `attitudeQuaternion`: orientation as a quaternion with components x, y, z, w
- `position`: x, y, z position in meters
- `velocity`: x, y, z velocity in m/s
- `acc`: x, y, z acceleration in g, excluding gravity on the z-axis

`name`

The `name` field is a `const char*` string identifier used by the firmware for logging and debugging. It is accessible at runtime via `stateEstimatorGetName()` and appears in the debug output when the estimator is selected.

6.2.1.2 Sensor Measurement Queue

The firmware decouples sensor drivers from estimators through a shared measurement queue. Sensor drivers call `estimatorEnqueue()` to push a `measurement_t` struct into the queue. The estimator can call `estimatorDequeue()` to consume all pending measurements. The queue must be fully drained each tick to prevent overflow and measurement loss.

The `measurement_t` struct uses a tagged union pattern: A `MeasurementType` enum field identifies the active sensor and the `data` union holds the corresponding payload. The available sensor types are listed in [Table 37](#).

Type	Fields	Source	Rate
<code>MeasurementTypeAcceleration</code>	<code>acc.x/y/z</code> in g	IMU (always available)	1000 Hz
<code>MeasurementTypeGyroscope</code>	<code>gyro.x/y/z</code> in deg/s	IMU (always available)	1000 Hz
<code>MeasurementTypeBarometer</code>	<code>baro.asl</code> in m, <code>baro.pressure</code> in mbar	Barometer (always available)	50 Hz
<code>MeasurementTypeTOF</code>	<code>distance</code> in m, <code>stdDev</code>	Flow Deck v2	40 Hz
<code>MeasurementTypeFlow</code>	<code>dpixelx/y</code> in pixels, <code>dt</code> in s	Flow Deck v2	100 Hz
<code>MeasurementTypeDistance</code>	<code>distance</code> in m, <code>anchorId</code> , <code>stdDev</code>	Loco Positioning Deck	500 Hz with 8 anchors (~62 Hz per anchor)

Table 37: Available sensor types for the estimator.

6.2.1.3 Sensor Data Preprocessing

The `measurement_t` structs in the queue do not carry raw sensor readings. The Crazyflie IMU driver applies three preprocessing steps before pushing the sensor data into the queue.

Gyroscope bias calibration

On startup the sensor task accumulates raw gyroscope readings into a circular buffer of 512 samples. Once the buffer is full the driver evaluates the sample variance on each axis. If all three variances fall below a threshold and at least one second has elapsed since startup, the mean of the last 512 samples is stored as the bias estimate and locked in for the remainder of the flight. The calibration relies on the platform being stationary during this initial phase. This removes the constant offset from the gyroscope readings so that a stationary sensor reports zero angular velocity.

Accelerometer scale normalisation

Once the gyroscope bias has been found, the sensor task collects 200 accelerometer samples. For each sample the Euclidean magnitude across all three axes is computed. The average of these 200 magnitudes serves as a scale factor. When the platform is stationary this magnitude should equal exactly 1 g. Each axis component is then divided by this factor. This corrects the overall scale of the accelerometer output so that gravity reads as exactly 1 g.

Low-Pass filtering

After bias removal and scale normalisation, the driver applies a second-order low-pass filter on each axis of both sensors. On each sample the filter computes the new output as a weighted sum of the current input, the two previous inputs and the two previous outputs. The cutoff

frequency determines above which frequency signals are attenuated. The gyroscope cutoff is 80 Hz and the accelerometer cutoff is 30 Hz. This suppresses high-frequency noise and vibration before the measurements reach the estimator.

6.2.2 Embedding the Library into the Crazyflie Build System

Since the Crazyflie firmware is written in C, the Rust EKF library must be compiled to a static archive and a corresponding C header before it can be used in the firmware. Within the Crazyflie firmware, our EKF is named RKF, short for Rust EKF. We use `cbindgen` [41] to automatically generate the header `rkf.h` from the Rust source. The Rust toolchain simultaneously produces the static archive `libekf.a`. Both artefacts are placed under `libs/rkf/` inside the Crazyflie firmware tree.

To integrate the library the Crazyflie `Makefile` needs two additions, shown in [Code Snippet 11](#):

MAKEFILE	Makefile
1	<code>EXTRA_FIRMWARE_LIBS += -L\$(LIB)/rkf -lekf</code>
2	<code>export EXTRA_FIRMWARE_LIBS</code>
3	
4	<code>INCLUDES += -I\$(LIB)/rkf</code>

Code Snippet 11: Crazyflie Makefile additions for the Rust EKF library.

`EXTRA_FIRMWARE_LIBS` sets the library search path and the link flag `-lekf`, which instructs the linker to resolve symbols from `libekf.a`. The variable is exported so that it is visible in the sub-make that runs the final link step, where it is appended in the linker command. `INCLUDES` adds `libs/rkf/` to the compiler include path so that `rkf.h` is found by any C translation unit that includes it.

6.2.3 C API

The `c_api.rs` module exposes all filter operations across the C boundary and is only compiled when the `c_api` feature flag is enabled. All entry points are declared `pub extern "C"` with `#[unsafe(no_mangle)]` so the C compiler sees stable, predictable symbol names that match the prototypes in `rkf.h`.

The filter runs internally as `Ekf<f32>`. Since the library targets a `#![no_std]` environment without a heap, the filter and IMU state are stored in two static cells. `rkf_init` must be called exactly once before any other function, it zero-initialises both cells and then sets the non-zero initial values (identity quaternion, initial covariance diagonal) by calling the internal `init()` method.

All values crossing the C boundary are `f32`. The state is returned as an `RkfState` struct which is shown in [Code Snippet 12](#).

RUST
c_api.rs

```

1  pub struct RkfState {
2      pub position: [f32; 3],
3      pub velocity: [f32; 3],
4      pub attitude: [f32; 4],    // quaternion (w, i, j, k)
5      pub covariance: [f32; COVARIANCE_DIM],
6  }

```

Code Snippet 12: RkfState struct exposed across the C boundary

The generated `rkf.h` contains the `RkfState` struct declaration and the function prototypes listed in [Table 38](#).

Symbol	Purpose
<code>rkf_init()</code>	Initialise filter and IMU scratch state, must be called exactly once
<code>rkf_predict_imu(...)</code>	Run the IMU prediction step with raw accelerometer and gyroscope data and their noise standard deviations
<code>rkf_update_uwb(...)</code>	Scalar UWB ranging correction given anchor coordinates, measured distance and standard deviation
<code>rkf_update_tof(...)</code>	Scalar ToF distance correction given measured distance and standard deviation
<code>rkf_get_rotation_matrix(...)</code>	Write the 3×3 body-to-NED rotation matrix in row-major order into a <code>float[3][3]</code> passed by pointer
<code>rkf_get_state()</code>	Return position, velocity, attitude and the full covariance as a <code>RkfState</code>
<code>rkf_init_uwb_adaptive(...)</code>	Initialise the adaptive UWB state with the given parameters, called automatically by <code>rkf_init</code>
<code>rkf_init_tof_adaptive(...)</code>	Initialise the adaptive ToF state with the given parameters, called automatically by <code>rkf_init</code>
<code>rkf_update_uwb_adaptive(...)</code>	Adaptive UWB ranging correction, noise variance is estimated online
<code>rkf_update_tof_adaptive(...)</code>	Adaptive ToF distance correction; noise variance is estimated by EMA

Table 38: Core C API of the Rust EKF library.

Beyond these core functions, the library additionally exposes a set of getter functions for internal filter quantities such as measurement innovations, innovation covariances and the current adaptive noise estimates for each sensor. These are not required to run the filter but allow the Crazyflie logging subsystem to record them for analysis and tuning.

6.2.4 Estimator Wrapper

The estimator wrapper `estimator_rkf.c` connects the Crazyflie firmware infrastructure to the Rust filter. It implements the `EstimatorFcns` interface [Section 6.2.1.1](#) and runs the filter in a dedicated FreeRTOS task.

6.2.4.1 Task Architecture

The wrapper follows the same task-based pattern used by the Kalman estimator in the stock firmware. A binary semaphore `runTaskSemaphore` is given by `estimatorRkf` on each stabilizer tick and taken by `rkfTask` to pace execution. A mutex `dataMutex` protects `taskEstimatorState`, the buffer that `estimatorRkf` copies back into the stabilizer's `state_t` on every tick.

After startup, `rkfTask` calls `rkf_init` once to initialize the filter and then enters its main loop. Each iteration begins with a prediction step that runs at 100 Hz: `Axis3fSubSampler` accumulates all IMU samples from the 1000 Hz interrupt and finalizes a weighted average, which is then passed to `rkf_predict_imu` together with the elapsed time in seconds. Next, all pending measurements are dequeued: UWB distance measurements are forwarded to `rkf_update_uwb` and ToF distance measurements to `rkf_update_tof`, all other measurement types are ignored. Finally, `rkf_get_state` retrieves the current filter state and `rkf_get_rotation_matrix` retrieves the body-to-NED rotation matrix for the acceleration computation, both of which are written into `taskEstimatorState` under the mutex.

[Figure 27](#) shows the flowchart of the `rkfTask`.

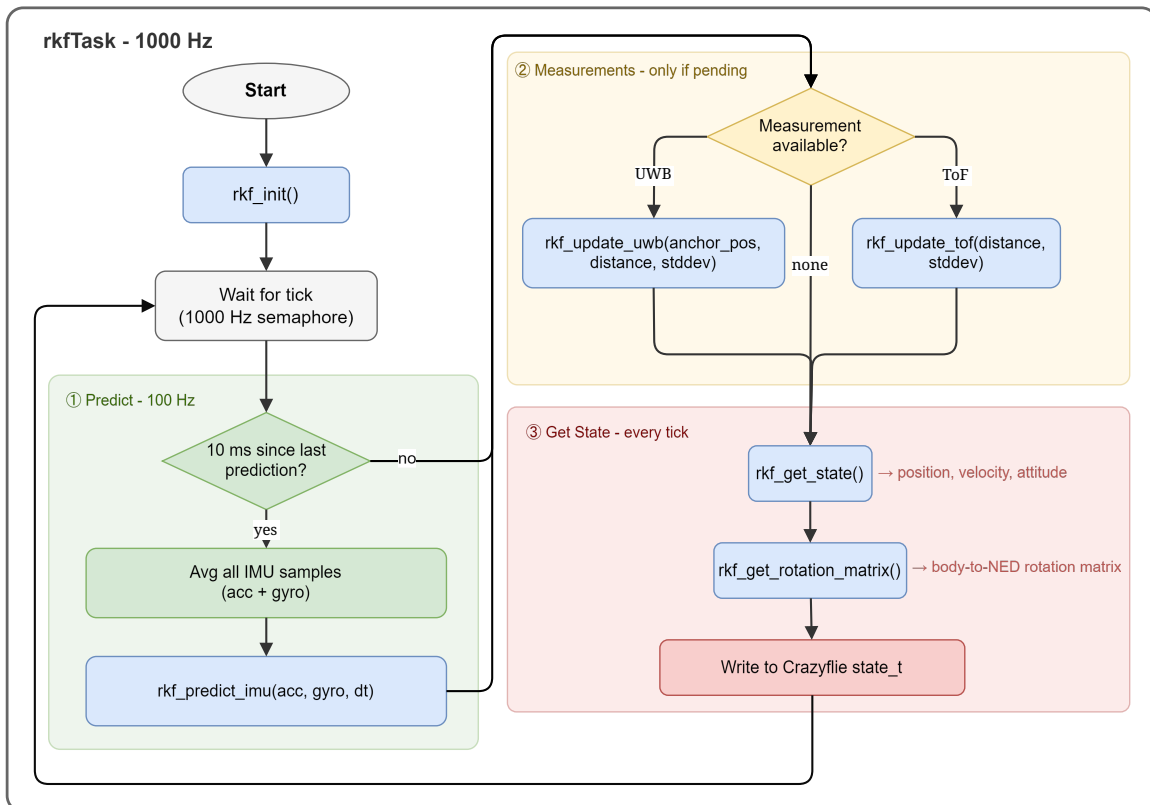


Figure 27: Flowchart of the `rkfTask` loop, Rust EKF functions are highlighted in blue.

6.2.4.2 Coordinate Frame Conversion

The Rust filter works in NED (North-East-Down). The Crazyflie firmware uses a right-handed frame where x points north, y points west and z points up, which differs from NED in the sign

of the y and z axes. The conversion is applied consistently at every boundary: sensor inputs to `rkf_predict_imu` and `rkf_update_uwb` negate the y and z components before passing them to the filter and `rkfExternalizeState` applies the inverse negation when writing position, velocity and attitude back into `taskEstimatorState`.

6.2.4.3 Measurement Dispatch

`updateQueuedMeasurements` drains the firmware measurement queue each tick. Gyroscope and accelerometer samples are forwarded to their respective sub-samplers. UWB distance measurements call `rkf_update_uwb` with the anchor coordinates converted to NED. ToF distance measurements call `rkf_update_tof` directly. All other measurement types are ignored.

6.2.4.4 Adaptive Noise Estimation

The wrapper supports an adaptive noise estimation mode. When enabled, `updateQueuedMeasurements` calls `rkf_update_uwb_adaptive` and `rkf_update_tof_adaptive` instead of their fixed-noise counterparts. The measurement noise variance is then estimated online by the filter rather than requiring a manually tuned standard deviation at call time. Whether adaptive noise estimation is active is selected via compile-time configuration.

6.2.5 Shadow Estimator

To compare two estimators on the same flight, we extended the Crazyflie firmware with a shadow estimator mechanism. The shadow estimator runs in parallel with the active estimator and receives the same sensor measurement. It operates as a fully functional estimator, executing its own predict and update steps, but its output state is never written back into the stabilizer's `state_t` and therefore has no influence on flight control. This allows both estimators to be driven by identical sensor data so that their state estimates can be logged and compared directly without any interaction between them. Which estimator acts as the shadow and whether the mechanism is enabled at all is selected via compile-time configuration.

6.2.6 Limitations

The current integration does not implement a measurement model for the optical flow sensor (PMW3901) available on the Flow Deck v2. `MeasurementTypeFlow` packets are therefore discarded by the estimator wrapper. Adding an optical flow model to the filter and integrating it into the wrapper is left as future work.

6.3 Evaluation Integration

The evaluation container drives the EKF Rust library from Python. This section documents the mechanism by which the evaluation runners (see [Section 5.3.5](#)) drive the filter.

6.3.1 Python Integration via PyO3

To evaluate the developed Rust EKF implementation we exposed its interface to Python using PyO3 [\[42\]](#). The decision to expose the filter to Python with PyO3 is recorded in [ADR 1](#). PyO3 is a library that enables Rust functions to be called directly from Python. It works with macros such as `#[pyfunction]`, `#[pyclass]` and `#[pymodule]` that mark Rust functions, classes and modules for export to Python. To compile the Rust crates we utilized the build tool Maturin [\[43\]](#).

[Code Snippet 13](#) shows a simple example of a PyO3 binding. The `#[pyfunction]` attribute is used to mark a Rust function for export to Python, while the `#[pymodule]` attribute defines the module that registers all exported functions. The usage in Python is straightforward and can be seen in [Code Snippet 14](#).

RUST	PyO3 example
<pre>1 use pyo3::prelude::*; 2 3 #[pyfunction] 4 fn add(left: i32, right: i32) -> i32 { 5 left + right 6 } 7 8 #[pymodule] 9 fn rust_module(m: &Bound<'_, PyModule>) -> PyResult<> { 10 m.add_function(wrap_pyfunction!(add, m)?); 11 Ok(()) 12 }</pre>	

Code Snippet 13: Simplified PyO3 binding exposing a Rust function to Python.

PYTHON	PyO3 usage example
<pre>1 import rust_module 2 result = rust_module.add(10, 20)</pre>	

Code Snippet 14: Usage of exported Rust function in Python.

6.3.2 EKF Python Binding

This section documents the implementation of the bindings used by the evaluation runners to drive the filter.

The binding is implemented in the separate file `python.rs` and compiled into an `ekf` module. The bindings are active if the `python` Cargo feature is used.

The module is designed to mirror the Rust API. The filter type `Ekf` is bound as a Python class and every predictor and measurement type (`ConstantVelocity`, `Imu`, `Gnss`, `UwbAnchor`, `Tof` and their adaptive variants) is bound as its own class. Python therefore drives the filter through the same `predict`, `correct` and `finalize` calls used on the embedded targets. The module additionally exports the state-index constants (`IDX_N`, `IDX_E`, ...) so that evaluation

code can address the state slots. All numeric data from Rust gets passed to Python as NumPy `float64` arrays.

A single `predict` method accepts any predictor instance and a single `correct` method accepts any measurement instance. Internally each method inspects the concrete type of the passed object and dispatches to the matching Rust call. [Code Snippet 15](#) shows a representative IMU prediction step driven from a Python runner.

PYTHON

```
1     ekf = Ekf()
2     predictor = Imu(accel=event["accel"],
3                     gyro=event["gyro"],
4                     noise=imu_noise,
5                     accel_convention=AccelConvention.Coordinate)
6     dt = event["time"] - last_time
7     ekf.predict(predictor, dt)
8     position = ekf.position.copy()
```

Code Snippet 15: Code showing an example of an IMU predict EKF call made from an evaluation runner.

7 Results and Evaluation

This chapter contains the evaluation of our EKF across both target platforms and assesses how well the implementation performs. The evaluation addresses the filter performance across different flight phases ([MHR-07](#)), the offline validation against simulated flight data ([MHR-08](#)), the cross-platform performance comparison ([MHR-09](#)) and the real-time behavior ([MHR-04](#)).

Simulated flight data with known ground truth forms the basis of the analysis. The evaluation focuses on simulation of both platforms and on the real Crazyflie, since the Crazyflie could actually be flown and thus provided real flight data. Testing on the ASTERIA sensor carrier was limited to a driving test, as integration delays restricted the available time and the recorded data offered only limited insight.

7.1 Evaluation Methodology

This section defines how the filter was evaluated, both in simulation and on the real target hardware. We evaluated two aspects of the filter: Accuracy, which measures how close the estimates are to a reference and consistency, which measures whether the filter’s reported uncertainty matches its actual error.

We first tuned the filter in simulation, where a known ground truth is available (see [Section 7.2](#)), then evaluated its accuracy and consistency against that ground truth (see [Section 7.3](#)). We additionally compared the static filter against the adaptive measurement noise variants, both on the simulation scenarios and in a dedicated scenario with changing measurement noise (see [Section 7.4](#)). Finally, we deployed the tuned filter on the real hardware and examined how well the simulation results transferred (see [Section 7.5](#)).

The simulation evaluation forms the core of our quantitative analysis, since ground truth allows accuracy to be measured directly and consistency to be checked with the normalized estimation error squared (NEES). On the Crazyflie the estimates are compared against the Crazyflie EKF as a reference (see [Section 7.5.1](#)) and on the ASTERIA sensor carrier against the recorded GNSS data (see [Section 7.5.2](#)). In both cases consistency is assessed with the normalized innovation squared (NIS), which relies only on quantities the filter computes internally.

The real-hardware evaluation serves primarily as a validation that the filter operates correctly on the target platforms, rather than as a detailed performance analysis. The deployment confirms that the filter runs in real time and produces usable estimates under real conditions, but the configuration was not optimized to the same degree as in simulation due to time limitations.

7.1.1 Performance Metrics

We quantify accuracy using the root mean square error (RMSE) and consistency using the normalized estimation error squared (NEES). Where no ground truth is available, we used the normalized innovation squared (NIS) to check consistency.

7.1.1.1 Root Mean Squared Error (RMSE)

The RMSE measures the average magnitude of the estimation error over a run. For a state component with estimate $\hat{x}_k$, ground truth x_k and N time steps, it is defined as

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{k=1}^N (\hat{x}_k - x_k)^2}. \quad (14)$$

We calculated the RMSE per axis for position and velocity. To obtain the RMSE over all axis we calculated the Euclidean distance between the estimated and the true vector at each time step.

The attitude is stored as quaternions, which cannot be compared component-wise. We therefore first compute the error rotation between the true orientation q_k and the estimated orientation $\hat{q}_k$,

$$q_{\text{err},k} = q_k \otimes \hat{q}_k^{-1}, \quad (15)$$

which describes the rotation that maps the estimate onto the ground truth. The magnitude of the attitude error at step k is the rotation angle of $q_{\text{err},k}$, denoted θ_k . Applying the RMSE formula to this angle gives the total attitude RMSE

$$\text{RMSE}_{\text{att}} = \sqrt{\frac{1}{N} \sum_{k=1}^N \theta_k^2}, \quad (16)$$

which we report in degrees.

For a per-axis breakdown, each error rotation is expressed as a rotation vector. The per-axis attitude RMSE is then calculated as the RMSE of the corresponding rotation-vector component over all time steps. For the attitude calculations, the python package `scipy` is used.

7.1.1.2 Normalized Estimation Error Squared (NEES)

The NEES metric compares the estimation error to the estimated covariance P_k

$$\varepsilon_k = (\hat{x}_k - x_k)^\top P_k^{-1} (\hat{x}_k - x_k). \quad (17)$$

A consistent filter produces NEES values that equal the state dimension, with statistical bounds derived from the chi-squared distribution [23]. Values that lie persistently above the bounds indicate an overconfident filter, values below indicate a conservative one. We use the NEES to judge the filters consistency for position, velocity and attitude.

7.1.1.3 Normalized Innovation Squared (NIS)

The NIS [26] provides an alternative consistency check using only quantities the filter computes internally and is thus applicable even when no ground truth is available. It is defined as

$$\varepsilon_\nu(k) = \nu(k)^\top S(k)^{-1} \nu(k), \quad (18)$$

where $\nu(k)$ is the measurement residual and $S(k)$ is the innovation covariance at time step k . For a consistent filter, the expected value of the NIS equals the measurement dimension m [26]. Values persistently below m indicate a conservative filter, values above m indicate an overconfident filter. In our evaluation this is $m = 1$ for the scalar UWB and ToF measurements on the Crazyflie and $m = 3$ for the three-dimensional GNSS position measurement on the ASTERIA sensor carrier board.

For the real hardware, we report the averaged NIS over N time steps,

$$\bar{\varepsilon}_\nu = \frac{1}{N} \sum_{k=1}^N \varepsilon_\nu(k), \quad (19)$$

whose expected value is likewise m for a consistent filter.

7.1.2 Simulation Test Scenarios and Conditions

The evaluation scenarios are generated with the simulation environments described in [Section 4.1.3.5](#) for the ARIS rocket and in [Section 4.2.3.5](#) for the Crazyflie.

We simulated a total of nine runs for the ARIS rocket and seven for the Crazyflie. The scenarios vary the environment, the flown trajectory and the sensor noise and dropout configurations. Together they cover a broad range of sensor dynamics, which let us verify that the EKF generalizes across different conditions rather than being tuned to a single case. [Figure 28](#) and [Figure 29](#) show the ground-truth trajectories of all simulated runs for the two platforms.

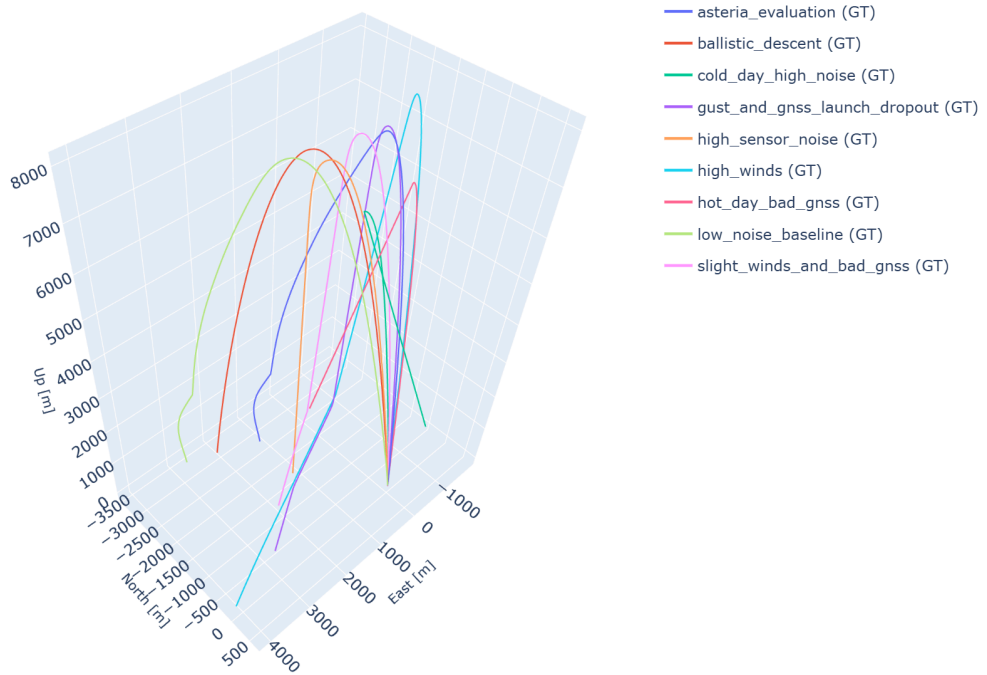


Figure 28: Ground-truth trajectories of the nine simulated rocket flights generated with RocketPy.

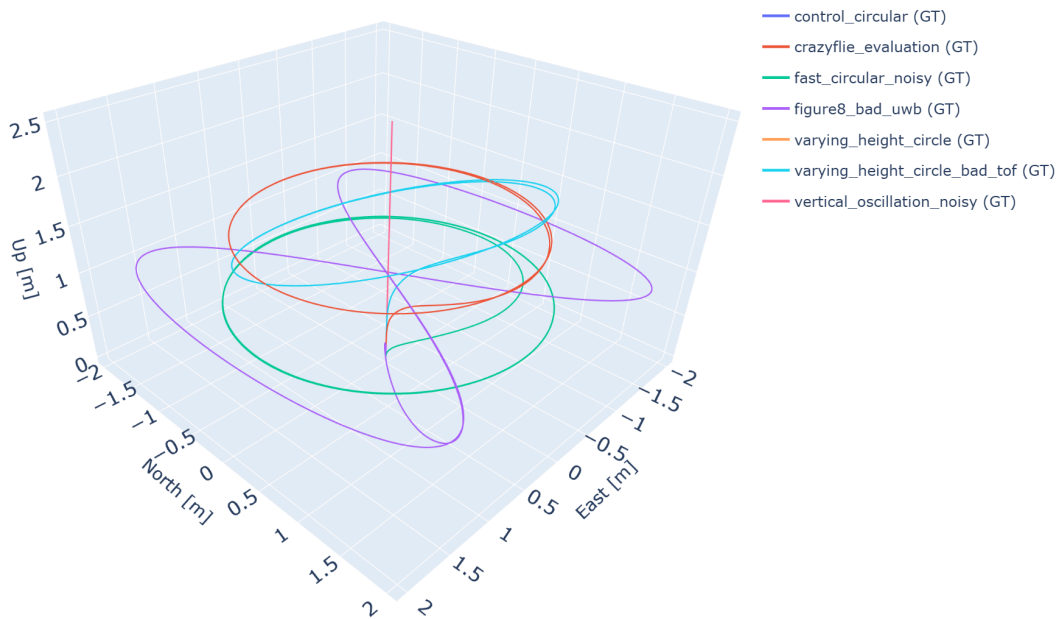


Figure 29: Ground-truth trajectories of the seven simulated Crazyflie flights generated with RotorPy.

Both simulation sets include one dedicated evaluation run that is held out from the optimization set. This run represents a baseline evaluation trajectory with nominal sensor behavior and serves as the primary simulation for the detailed analysis in [Section 7.3](#).

7.2 Parameter Optimization

The EKF behavior depends on a set of parameters that regulate how strongly the filter trusts its process and measurement models. We developed an approach to tune these parameters automatically on top of the existing simulation infrastructure, using a Bayesian search (see [Section 5.3.8](#)).

The search optimizes the parameters against the simulated runs introduced in [Section 7.1.2](#). For each parameter set, the filter is run on all optimization scenarios. The parameter set with the lowest mean position RMSE is selected.

Of the nine rocket runs and seven Crazyflie runs, one run each is considered a dedicated baseline and is held out from the search. This let us analyze the filter behavior on data the parameters were not tuned on.

We optimized each platform separately, since their dynamics and sensor suites differ. The resulting configurations are reported in [Section 7.2.2](#) and [Section 7.2.3](#).

7.2.1 Search Space and Objective Definition

The search space consists of the noise standard deviations that the filter treats as tunable. The position sources differ per platform, so the search spaces differ accordingly. The rocket uses GNSS, with separate horizontal and vertical standard deviations ([Table 39](#)), while the Crazyflie uses UWB and a ToF sensor ([Table 40](#)).

The bounds were chosen to span roughly 0.5 to 10 times the noise applied to each sensor. We performed 50 trials for the rocket and 150 for the Crazyflie. Despite the smaller trial number, the rocket search took about 13 hours, whereas the Crazyflie search completed in roughly 1.5 hours.

Parameter	Scale	Lower bound	Upper bound
Accelerometer std. dev.	log	0.01 m/s ²	0.3 m/s ²
Gyroscope std. dev.	log	0.0005 rad/s	0.03 rad/s
GNSS horizontal std. dev.	linear	0.5 m	10.0 m
GNSS vertical std. dev.	linear	1.0 m	20.0 m

Table 39: Search parameters and bounds for the ARIS rocket (50 trials).

Parameter	Scale	Lower bound	Upper bound
Accelerometer std. dev.	log	0.01 m/s ²	0.5 m/s ²
Gyroscope std. dev.	log	0.001 rad/s	0.05 rad/s
UWB std. dev.	log	0.01 m	0.3 m
ToF std. dev.	log	0.001 m	0.1 m

Table 40: Search parameters and bounds for the Crazyflie (150 trials).

7.2.2 ARIS Rocket Optimization Results

The selected parameters corresponding to the trial with the lowest mean position RMSE are listed in [Table 41](#).

Parameter	Selected value
Accelerometer std. dev.	0.295 m/s ²
Gyroscope std. dev.	0.0285 rad/s
GNSS horizontal std. dev.	2.312 m
GNSS vertical std. dev.	3.847 m

Table 41: Selected EKF parameters for the ARIS rocket.

Both inertial parameters settled close to the upper end of their search ranges, while the GNSS standard deviations were selected well within their bounds, with the vertical value larger than the horizontal one, which was expected. The implications of these values are discussed in [Section 7.7.7](#).

7.2.3 Crazyflie Optimization Results

The selected parameters corresponding to the trial with the lowest mean position RMSE are listed in [Table 42](#).

Parameter	Selected value
Accelerometer std. dev.	0.369 m/s ²
Gyroscope std. dev.	0.0498 rad/s
UWB std. dev.	0.0204 m
ToF std. dev.	0.0068 m

Table 42: Selected EKF parameters for the Crazyflie.

For the Crazyflie, the gyroscope standard deviation was driven almost exactly to its upper bound and the accelerometer standard deviation settled high within its range, mirroring the pattern seen on the rocket. The selected UWB standard deviation of 0.0204 m is notably small for the noise that the real UWB measurements exhibit. The ToF standard deviation was selected substantially smaller than the UWB value, indicating that the search favored the altitude measurement as the more trusted source. The implications of these values are discussed in [Section 7.7.7](#).

7.3 Simulation Validation

This section reports the filter performance on the simulated scenarios with known ground truth, using the static parameters obtained in the parameter search (see [Section 7.2](#)).

Results are reported either as an average over all simulation runs of the respective platform or on the held-out baseline evaluation run.

7.3.1 ARIS Rocket Simulation Results

The following results were obtained on the ARIS rocket with the automatically tuned parameters from [Section 7.2.2](#).

One limitation affects the simulation and the evaluation of these results, especially the descent phase. As described in [Section 4.1.4.1](#), RocketPy switches from a 6-DOF to a 3-DOF model at drogue inflation and no longer tracks attitude or angular velocity from that point on. As a consequence, attitude estimation cannot be evaluated during descent. Additionally, position and velocity values during descent should be interpreted with this in mind.

7.3.1.1 State Estimation Accuracy

[Table 43](#) reports the position, velocity and attitude RMSE of the rocket per axis and as a combined value. The values are averaged over all simulated runs. Position is estimated with an RMSE of 15.12 ± 12.46 m and velocity with 2.12 ± 0.88 m/s. The attitude error is small about the x and y axes (0.74° and 1.14°) but substantially larger about the z axis ($12.84 \pm 9.21^\circ$), which dominates the combined attitude RMSE of 12.93° .

Metric	Unit	x	y	z	3D
Position RMSE	m	8.91 ± 12.54	5.87 ± 6.11	7.46 ± 4.44	15.12 ± 12.46
Velocity RMSE	m/s	1.21 ± 0.95	0.97 ± 0.42	1.07 ± 0.79	2.12 ± 0.88
Attitude RMSE	deg	0.74 ± 0.84	1.14 ± 0.63	12.84 ± 9.21	12.93 ± 9.25

Table 43: Rocket position, velocity and attitude RMSE per axis and as a combined value, averaged over all simulated runs ($\pm$ standard deviation across runs).

[Figure 30](#) shows the estimated and true trajectory of the evaluation run, including the GNSS-outage segment from launch until apogee. The per-axis position, attitude and velocity errors over time for this run are shown in [Figure 31](#), [Figure 32](#) and [Figure 33](#).

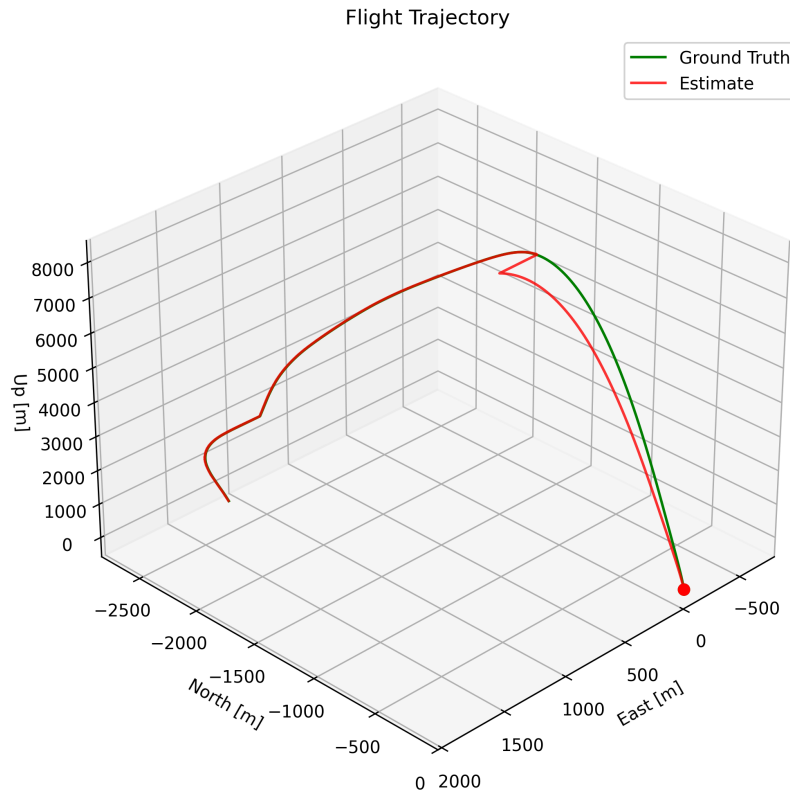


Figure 30: True vs. estimated 3D trajectory of the rocket evaluation run, including the GNSS-outage segment.

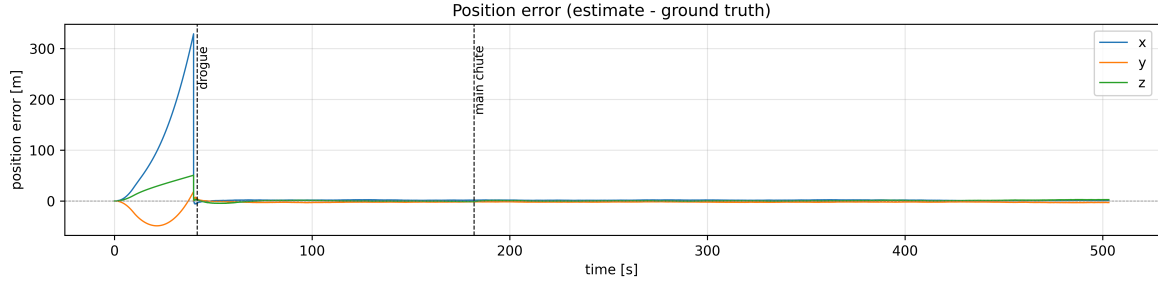


Figure 31: Per-axis position error of the rocket evaluation run.

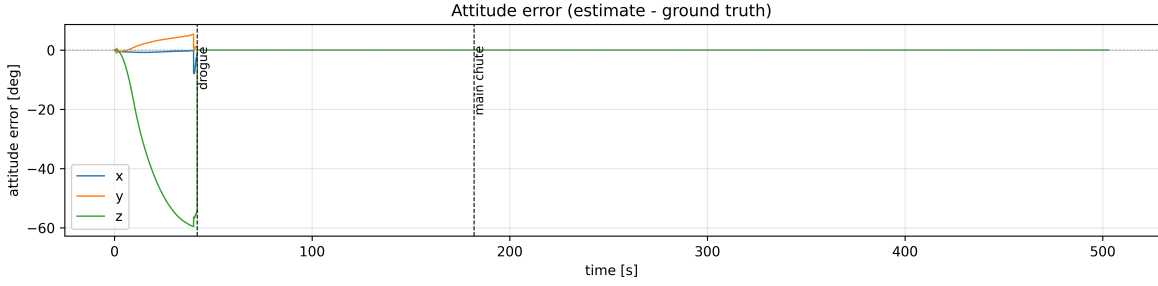


Figure 32: Attitude error of the rocket evaluation run.

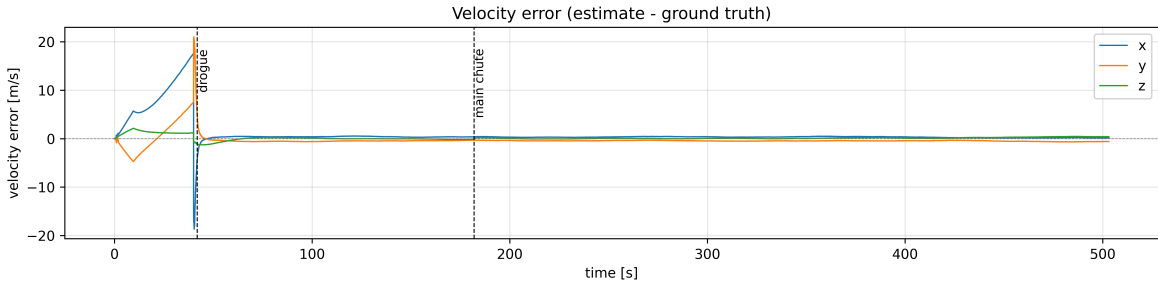


Figure 33: Per-axis velocity error of the rocket evaluation run.

The position error grows throughout the GNSS-outage segment and drops once GNSS corrections resume near apogee. The attitude error is mainly dominated by a growing deviation about the z axis. We interpret these effects in [Section 7.7.3](#) and [Section 7.7.2](#).

7.3.1.2 Filter Consistency

[Figure 34](#) shows the NEES of the position, velocity and attitude groups over time, together with the expected value for a three-dimensional state ($\text{dof} = 3$). For a consistent filter, the NEES should fluctuate around this expected value.

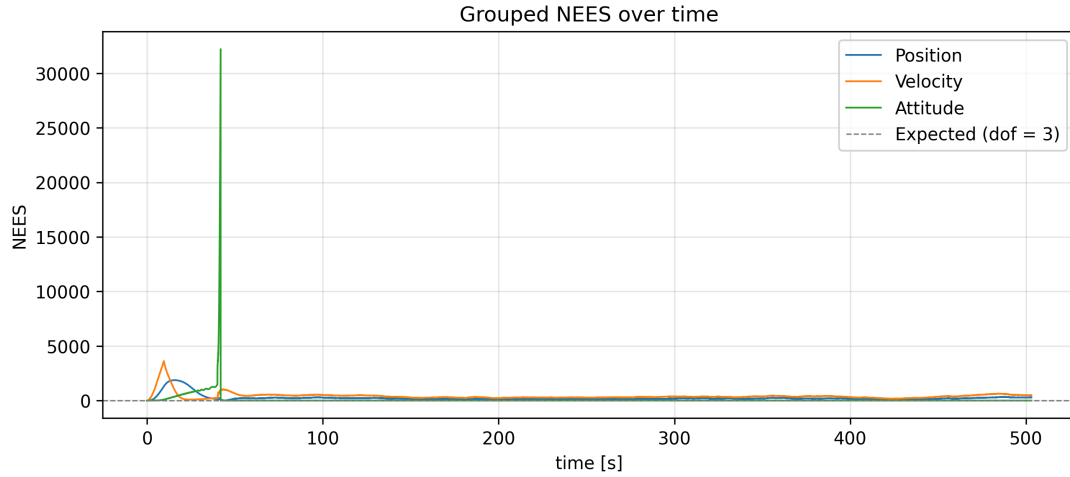


Figure 34: Grouped NEES of the rocket position, velocity and attitude over time, with the expected value for dof = 3.

All three groups lie persistently above the expected value for the entire flight, which indicates that the filter is overconfident. Its reported covariance is smaller than the actual estimation error. The attitude group shows a single sharp spike at around 42 s, which we identified as an artifact of the 3-DOF transition described in [Section 4.1.4.1](#).

The causes of the overconfidence are discussed in [Section 7.7.1](#).

7.3.1.3 Filter Behavior Across Flight Phases

This section breaks down the RMSE by flight phases. [Table 44](#) reports the position, velocity and attitude RMSE per phase, averaged over all simulated runs.

Phase	Position [m]	Velocity [m/s]	Attitude [deg]
Boost	8.73 ± 5.04	2.59 ± 1.25	6.79 ± 0.75
Coast	42.78 ± 59.58	4.94 ± 4.56	40.30 ± 4.48
Drogue	8.35 ± 4.72	1.64 ± 0.84	$\times$
Main chute	7.90 ± 5.24	1.43 ± 0.86	$\times$

Table 44: Rocket position, velocity and attitude RMSE per flight phase, averaged over all simulated runs ($\pm$ standard deviation across runs).

The coast phase shows by far the largest errors across all three groups, with a position RMSE of 42.78 m and an attitude RMSE of 40.30°, both several times higher than in any other phase. Its position error also has a very large standard deviation across runs, indicating that the coast-phase error varies strongly between scenarios. The boost, drogue and main chute phases all remain at a comparable, much lower position RMSE of around 8 m. The attitude RMSE during the drogue and main chute phases is missing due to the state being frozen.

[Figure 35](#), [Figure 36](#) and [Figure 37](#) show the error over time for the ascent phase of the baseline evaluation run.

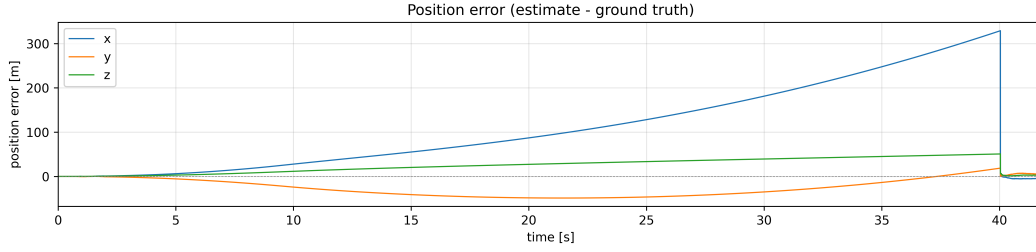


Figure 35: Position error during the ascent phase of the baseline evaluation run.

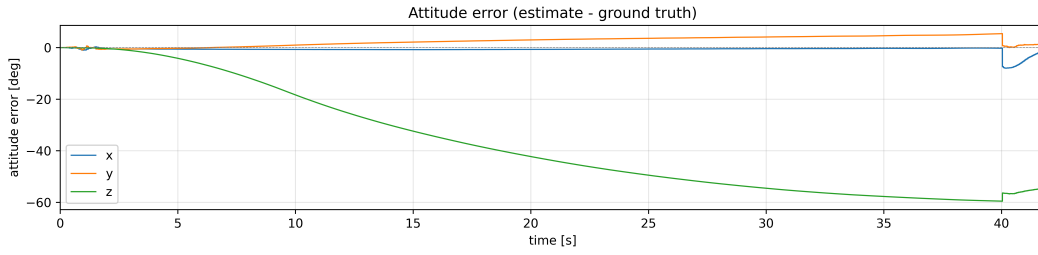


Figure 36: Attitude error during the ascent phase of the baseline evaluation run.

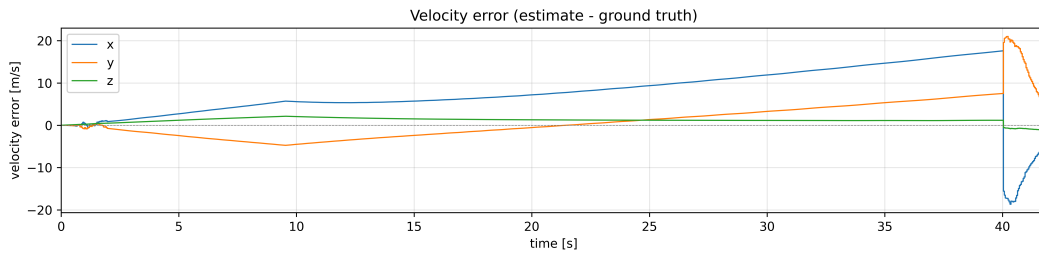


Figure 37: Velocity error during the ascent phase of the baseline evaluation run.

[Figure 38](#) and [Figure 39](#) show the error over time for the descent phase of the baseline evaluation run. For the descent phase, the attitude is omitted as it is frozen.

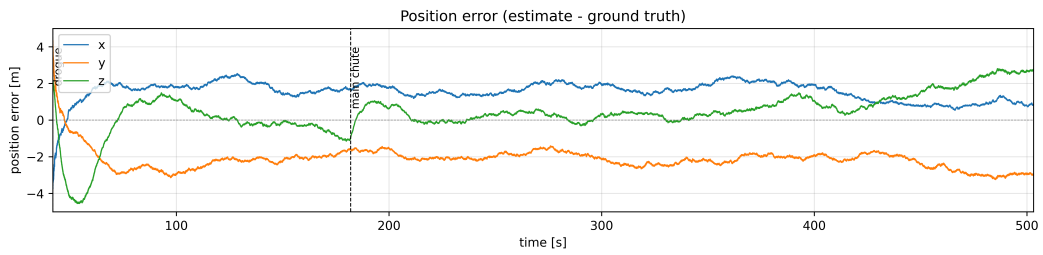


Figure 38: Position error during the descent phase of the baseline evaluation run.

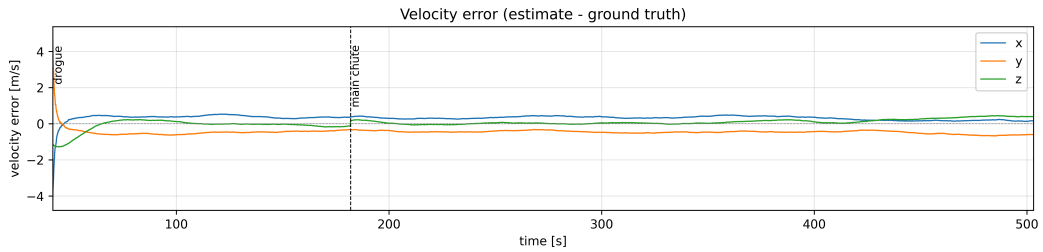


Figure 39: Velocity error during the descent phase of the baseline evaluation run.

These two phases together cover the full range of the flight dynamics. The reasons behind the large coast-phase errors are discussed in [Section 7.7.3](#).

7.3.2 Crazyflie Simulation Results

The following results were obtained on the Crazyflie with the automatically tuned parameters from [Section 7.2.3](#) and including the added ToF sensor.

7.3.2.1 Accuracy

[Table 45](#) lists the average position, velocity and attitude RMSE of all Crazyflie simulation runs. Position is estimated with a total RMSE of 0.0304 ± 0.0087 m and velocity with 0.0966 ± 0.0244 m/s. As for the rocket, the attitude error is small about the x and y axes and larger about the z axis, which dominates the combined attitude RMSE of 1.90° .

Metric	Unit	x	y	z	3D
Position RMSE	m	0.0177 ± 0.0054	0.0223 ± 0.0076	0.0100 ± 0.0021	0.0304 ± 0.0087
Velocity RMSE	m/s	0.0625 ± 0.0138	0.0674 ± 0.0201	0.0276 ± 0.0113	0.0966 ± 0.0244
Attitude RMSE	deg	0.65 ± 0.20	0.66 ± 0.19	1.57 ± 1.20	1.90 ± 1.11

Table 45: Crazyflie position, velocity and attitude RMSE per axis and as a combined value, averaged over all simulated runs ($\pm$ standard deviation across runs).

[Figure 40](#) shows the estimated and true trajectory of the evaluation run. The per-axis position, attitude and velocity errors over time are shown in [Figure 41](#), [Figure 42](#) and [Figure 43](#).

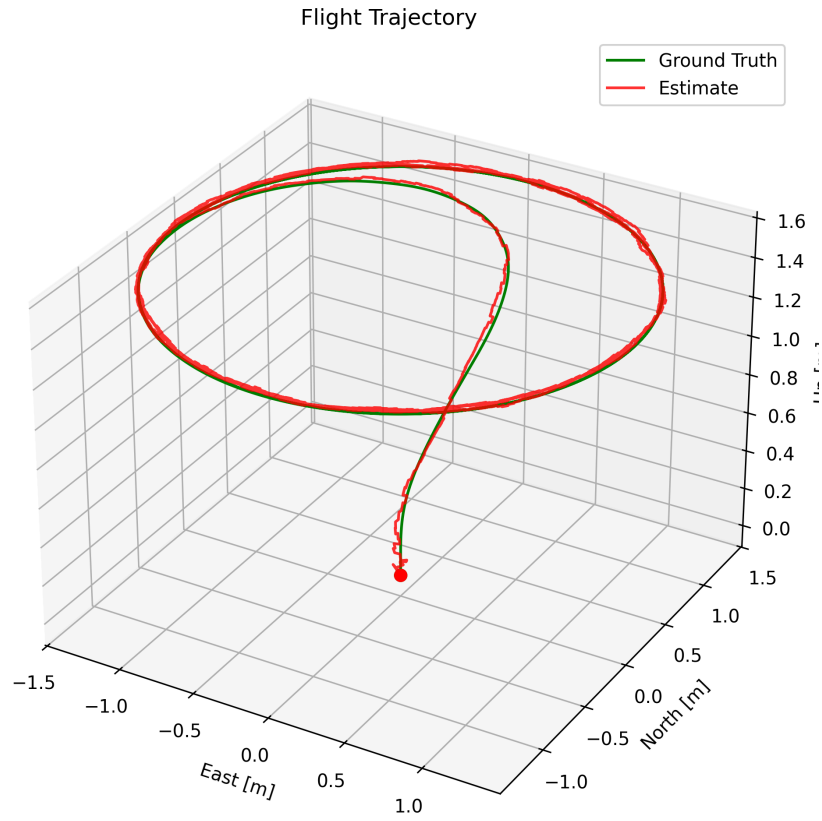


Figure 40: True vs. estimated 3D trajectory of the Crazyflie evaluation run.

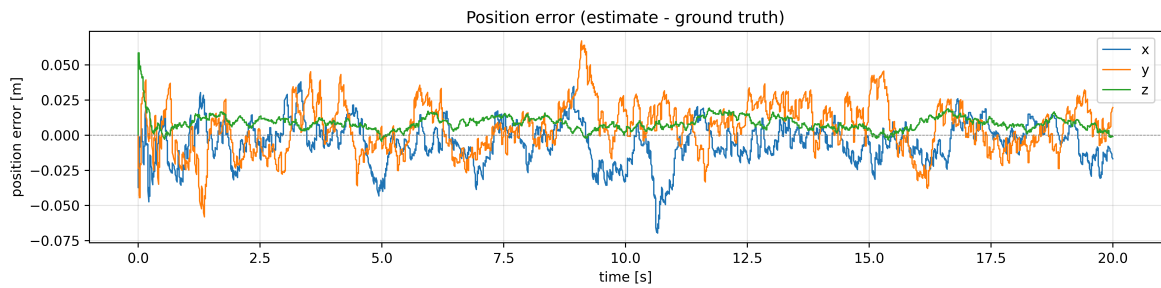


Figure 41: Per-axis position error of the Crazyflie evaluation run.

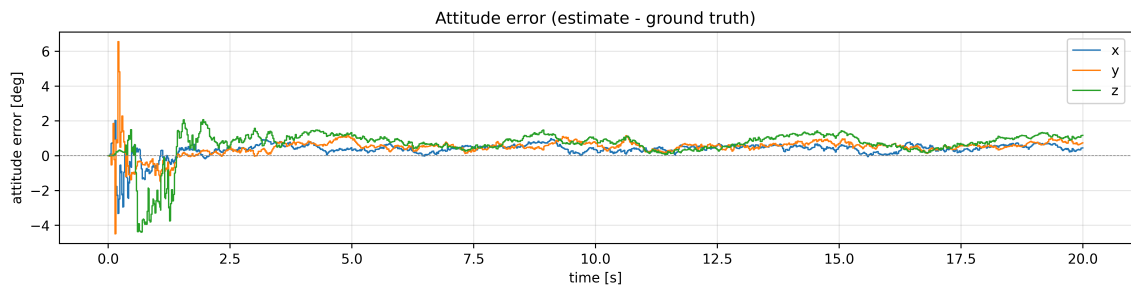


Figure 42: Attitude error of the Crazyflie evaluation run.

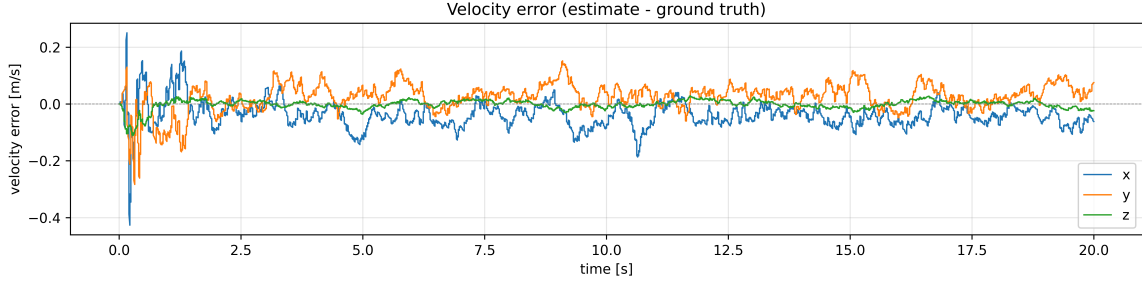


Figure 43: Per-axis velocity error of the Crazyflie evaluation run.

The x - and y -axis position errors vary over time, while the z -axis position error stays relatively consistent throughout. The attitude and velocity errors are largest at the start of the run and settle to their steady-state level after the initial seconds.

7.3.2.2 Consistency

[Figure 44](#) shows the NEES of the position, velocity and attitude groups over time, together with the expected value for a three-dimensional state ($\text{dof} = 3$). For a consistent filter, the NEES should fluctuate around this expected value.

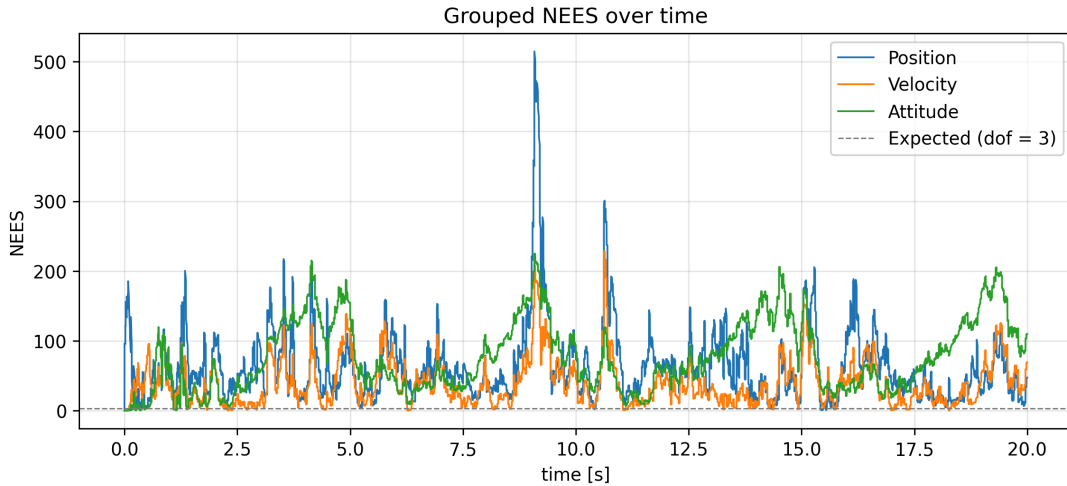


Figure 44: Grouped NEES of the crazyflie position, velocity and attitude over time, with the expected value for $\text{dof} = 3$.

As with the rocket, all three groups lie persistently above the expected value for the entire flight, which indicates that the filter is overconfident. The causes of the overconfidence are discussed in [Section 7.7.1](#).

7.3.2.3 Filter Extensibility

To demonstrate the flexibility of our filter and simulation architecture, we extended the Crazyflie with an additional ToF sensor during development. For the EKF architecture, the sensor was added as a measurement implementation. For the simulation, the sensor was generated synthetically from ground-truth data and then added to the runner.

[Figure 45](#) compares the altitude error of the two configurations (with and without ToF sensor) over time. Adding the ToF sensor reduced the vertical position RMSE from 0.026 m to 0.010 m without any change to the filter beyond enabling the additional measurement.

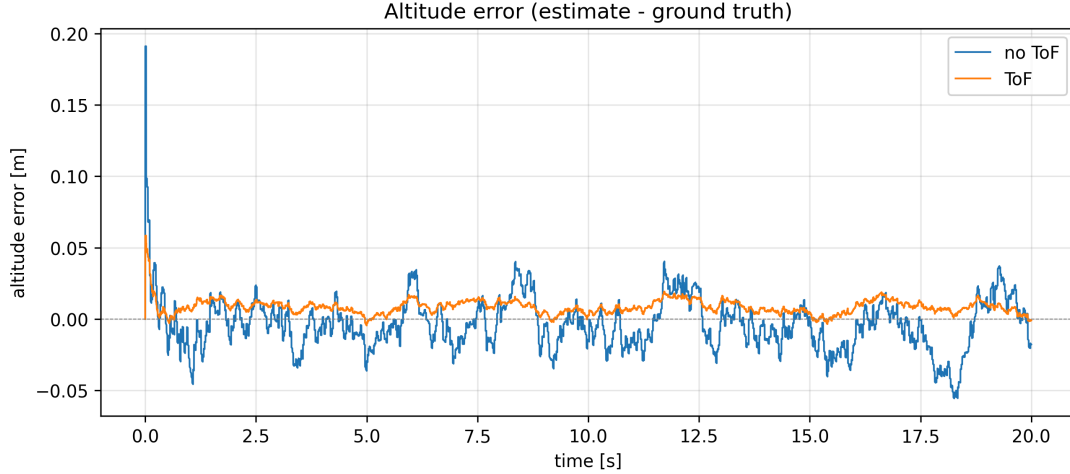


Figure 45: Altitude error of the Crazyflie evaluation run with and without the ToF sensor.

The implications of this are discussed in [Section 7.7.6](#).

7.4 Adaptive Measurement Noise

So far, the filter relied on static noise parameters obtained from the parameter search (see [Section 7.2](#)). This section evaluates the adaptive measurement noise estimation described by [SHR-01](#), in which R is estimated dynamically from the measurement innovations instead of being tuned once and held constant.

We reuse the automatically optimized parameters from [Section 7.2](#) for the static variant and compare them against two adaptive variants. The goal is to assess whether an adaptively estimated R can match the automatically tuned one on the simulation scenarios and to show how it reacts when the measurement noise changes during a run, which a static R cannot account for.

7.4.1 Approach

We compare three variants that differ only in how the measurement noise R is determined:

- Static variant: The static R obtained from the Bayesian search in [Section 7.2](#), held constant throughout the run.
- Sliding window (SW): R is estimated adaptively from the measurement innovations over a fixed-length sliding window (see [Section 3.7.1](#)).
- Exponential moving average (EMA): R is estimated adaptively by exponentially weighting recent measurement innovations with a smoothing factor α (see [Section 3.7.2](#)).

The window size and the smoothing factor α were chosen so that the estimate of R reflects roughly the last few seconds of measurements. We chose the values by hand and they may be tuned further in future work. The same smoothing factor $\alpha = 0.05$ is used for both platforms. The window sizes are set per sensor, as listed in [Table 46](#). Because the sensors run at different sampling rates, the windows correspond to roughly 10 s of GNSS data on the rocket and about 5 s of UWB and ToF data on the Crazyflie.

Platform	Sensor	Window size [samples]	Window [s]
ARIS Rocket	GNSS	250	10
Crazyflie	UWB	250	5
Crazyflie	ToF	200	5

Table 46: Adaptive window size per sensor used for the evaluation.

7.4.2 Accuracy Comparison

We compared the three variants on the baseline evaluation runs of both platforms. [Table 47](#) reports the results for the ARIS rocket and [Table 48](#) for the Crazyflie, each averaged over all simulated runs introduced in [Section 7.1.2](#).

Metric	Unit	Static	Sliding window	EMA
Position RMSE	m	15.12 ± 12.46	35.05 ± 16.55	36.35 ± 15.25
Velocity RMSE	m/s	2.12 ± 0.88	3.25 ± 1.25	3.38 ± 1.18
Attitude RMSE	deg	12.93 ± 9.25	12.93 ± 9.21	12.91 ± 9.18

Table 47: Rocket RMSE for the static variant and the two adaptive variants (SW and EMA), averaged over all simulated runs.

Metric	Unit	Static	Sliding window	EMA
Position RMSE	m	0.0304 ± 0.0087	0.0446 ± 0.0140	0.0477 ± 0.0170
Velocity RMSE	m/s	0.0966 ± 0.0244	0.1170 ± 0.0333	0.1118 ± 0.0411
Attitude RMSE	deg	1.90 ± 1.11	1.71 ± 1.12	1.63 ± 1.15

Table 48: Crazyflie RMSE for the static variant and the two adaptive variants (SW and EMA), averaged over all simulated runs.

On both platforms the static variant achieves the lowest position and velocity RMSE. On the ARIS rocket the difference is quite large. The adaptive variants reach a position RMSE of about 35 m, more than double the 15.12 m of the static variant. On the Crazyflie the gap is smaller but points the same way, with the adaptive variants at around 0.045 – 0.048 m against 0.0304 m for the static variant. The attitude RMSE is practically identical per platform across all three variants. This is expected, since the adaptive estimation only affects the position measurement noise.

[Figure 46](#) and [Figure 47](#) show the position error over time for the three variants on the reference evaluation runs for the ARIS rocket and the Crazyflie.

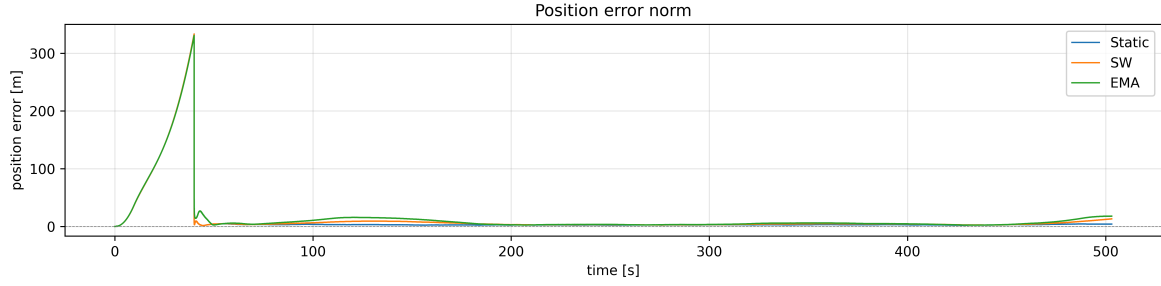


Figure 46: Position error over time for the static, sliding window and EMA variants on the ARIS rocket evaluation run.

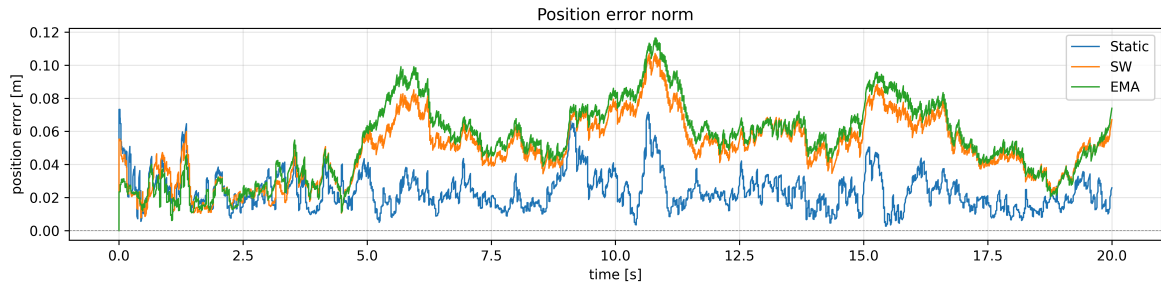


Figure 47: Position error over time for the static, sliding window and EMA variants on the Crazyflie evaluation run.

The reasons why the adaptive variants perform worse than the static variant on the nominal simulation scenarios are discussed in [Section 7.7.4](#).

7.4.2.1 Effect of Bias Compensation on the Crazyflie

The adaptive variants in [Section 7.4.2](#) were evaluated on raw, biased sensor measurements. The real Crazyflie firmware applies a bias compensation step at the initialisation of the IMU that removes part of the inertial bias before the measurements reach the estimator. Our simulation environment does not replicate this. To assess the effect of this step, we repeated the accuracy comparison for the Crazyflie with the fixed bias removed from the simulated IMU data. Because the fixed bias is known exactly in simulation, this removal is idealized.

[Figure 48](#) shows the position error over time for the static variant and the two adaptive variants with and without bias. With the biased measurements, the sliding window and EMA variants drift above the static variant after the first few seconds, reaching position errors of up to roughly 0.12 m. Once the bias is removed, both adaptive variants track closely with the static variant and all three variants remain in the range of about 0.01 – 0.04 m for the whole run.

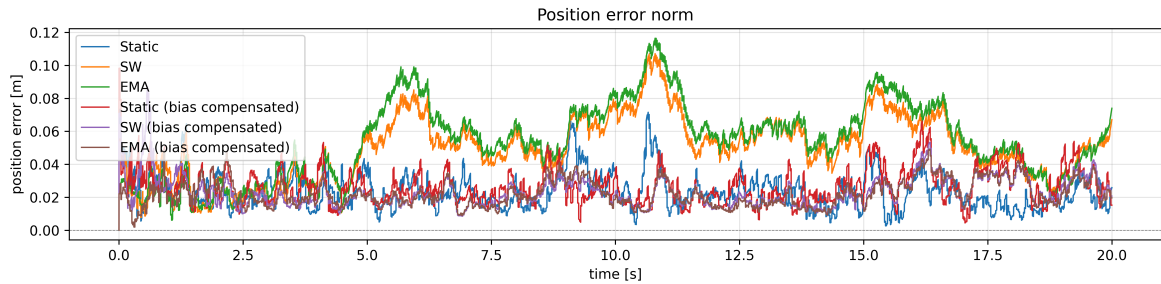


Figure 48: Position error over time for the static parameters and the sliding window and EMA variants (with and without bias compensation) on the Crazyflie evaluation run.

This indicates that the adaptive variants performed worse under unmodeled bias. A bias compensation step on the measurements enables a notable improvement of the adaptive performance. This topic is further discussed in [Section 7.7.4](#).

7.4.3 Robustness with changing Variance

The accuracy comparison in [Section 7.4.2](#) used scenarios with constant measurement noise, on which the static R performed best. However, a static R cannot react when the actual measurement noise changes during a run, whereas the adaptive methods can change R during runtime. This section examines that case.

We evaluated this on the Crazyflie using a dedicated test scenario in which the measurement noise of the UWB transceivers changes during the run. The scenario consists of three consecutive 20-second phases in which the UWB noise is scaled by a factor of 1.0, 2.0 and 0.5 relative to the normal level.

As shown in [Section 7.4.2.1](#), the adaptive variants notably increase their reliability once the inertial bias is compensated for. We therefore evaluated this scenario with the bias compensation step applied to show the potential of the adaptive methods. The process noise Q and all other settings are kept identical to the previous sections, so that only the behavior of R differs between the static and the adaptive variants.

[Figure 49](#) and [Figure 50](#) show the measurement noise standard deviation $\sqrt{R}$ for the sliding window and EMA variants. Both variants react to the change in measurement noise, but with different characteristics. The sliding window variant reacts more slowly, adjusting over roughly the length of its window size (about 5 seconds) but its estimates are more stable. The EMA variant reacts faster but produces a noisier estimate.

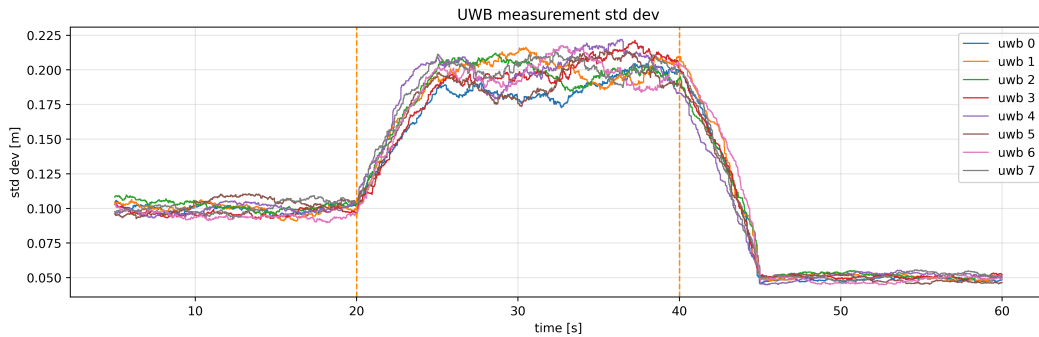


Figure 49: Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the changing-variance Crazyflie scenario.

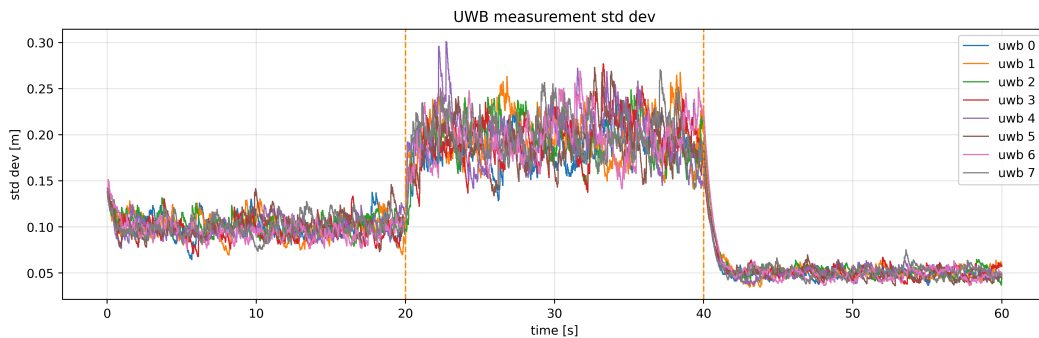


Figure 50: Estimated measurement noise standard deviation $\sqrt{R}$ over time for the EMA variant on the changing-variance Crazyflie scenario.

[Figure 51](#) shows the resulting position error over time for the three variants on the same scenario. During the first 20 seconds (at nominal noise) all three methods perform about the same. In the second phase (with double UWB noise), the advantage of the adaptive variants shows. The adaptive variants increase their measurement noise R and achieve a lower position error than the static variant. In the final phase (at low noise) the position error is low for all three methods. The sliding window and EMA variants behave very similarly throughout the whole run with only a few isolated differences in the position error.

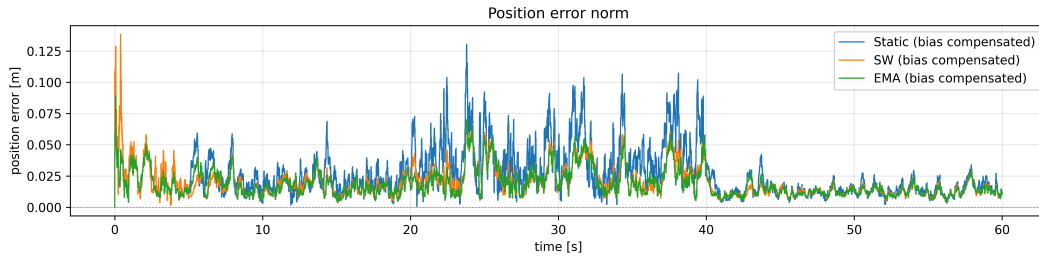


Figure 51: Position error over time for the static, sliding window and EMA variants on the changing-variance Crazyflie scenario.

[Table 49](#) reports the position, velocity and attitude RMSE of the three variants on this scenario. Both adaptive variants clearly outperform the static variant, with the EMA variant achieving the lowest error.

Metric	Unit	Static	Sliding window	EMA
Position RMSE	m	0.0330	0.0234	0.0217
Velocity RMSE	m/s	0.0842	0.0549	0.0259
Attitude RMSE	deg	0.7078	0.6090	0.2182

Table 49: RMSE of the static, sliding window and EMA variants on the changing-variance Crazyflie scenario.

These results are from a single, deliberately constructed scenario and therefore offer only limited insight. They show promising results for the adaptive variants, but further work would be needed to fully verify this. The interpretation of these results and the trade-off between the static and adaptive approaches is discussed in [Section 7.7.4](#).

7.5 Validation with Real Data

This section evaluates our EKF on real hardware. We present results from indoor flights on the Crazyflie ([Section 7.5.1](#)) and car drives with the ASTERIA sensor carrier board ([Section 7.5.2](#)).

7.5.1 Crazyflie Results

We evaluated our EKF on real indoor flights, where it ran on the Crazyflie either as the active estimator controlling the drone or as a passive shadow estimator alongside the Crazyflie EKF. No external ground truth such as a motion-capture system was available, so we compare the output of our EKF against the Crazyflie EKF as a reference. Since the Crazyflie EKF is itself an estimator with its own errors, this evaluation measures agreement between two filter implementations rather than accuracy against an absolute reference.

Unlike the simulation configuration described in [Section 7.3](#), the Crazyflie firmware performs a bias compensation, as detailed in [Section 6.2.1.3](#).

The test environment was an indoor room equipped with eight UWB anchors, one placed in each corner of the flight area. All anchors were placed at least 15 cm from any wall or floor surface to minimize reflections from nearby surfaces. [Figure 52](#) shows four of the eight mounted UWB anchors.



Figure 52: Four of the eight UWB anchors mounted on the wall of the test room.

To cover a range of flight characteristics, the four different trajectories shown in [Figure 53](#) were flown. Neither our EKF nor the Crazyflie EKF followed the programmed paths exactly, as the trajectory definitions could not be configured accurately enough within the available time. Since the two estimators are compared against each other, the actual flight path does not affect the integrity of the results.

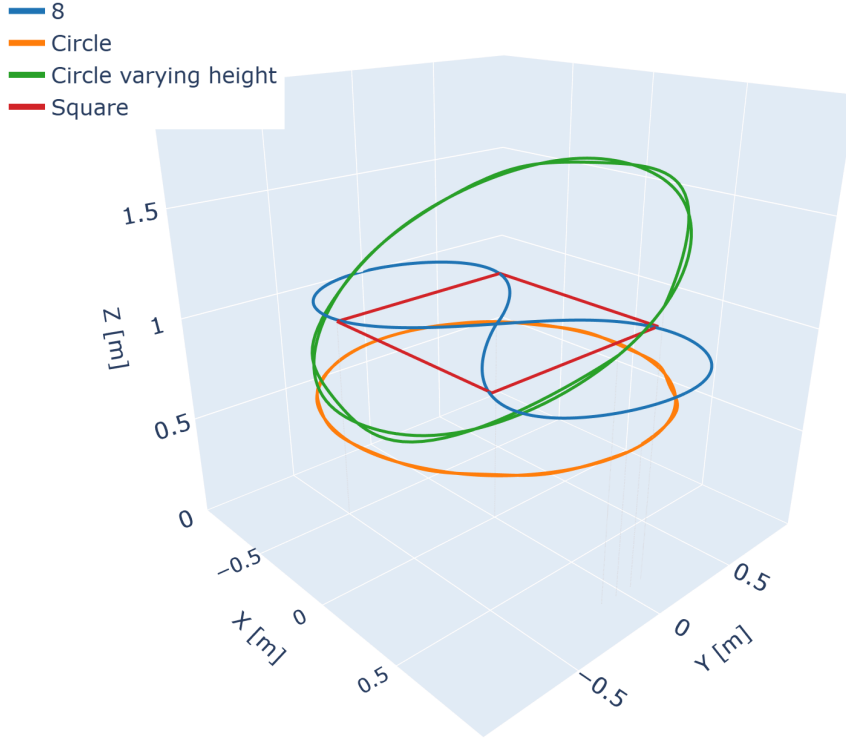


Figure 53: 3D flight paths of the four different trajectories flown for the Crazyflie evaluation.

Repeated flights under identical conditions showed similar results with no notable differences.

7.5.1.1 Evaluation with Bayesian-Optimized Parameters

The parameters used here are taken directly from the Bayesian optimization described in [Section 7.2 \(Table 42\)](#), with UWB sensor and the ToF sensor enabled.

This configuration performed poorly. Our EKF produced very noisy position estimates, whether it was running as the active or shadow estimator. When our EKF was used as active estimator, the noisy estimate caused unstable flight and noticeably worse trajectory tracking compared to the Crazyflie EKF. The attitude deviation is dominated by yaw, which is substantially larger than roll and pitch in both configurations. The root cause of the noisy position estimate is discussed in [Section 7.7.5](#) and the large yaw deviation in [Section 7.7.2](#).

[Table 50](#) shows the deviation between the two estimators for position, velocity and attitude, combined across all four trajectories per configuration and [Figure 54](#) shows the resulting 3D flight paths for the figure-8 trajectory.

Metric	Unit	Crazyflie EKF active			Our EKF active		
		Mean	Std	Max	Mean	Std	Max
Position	m	0.094	0.054	0.393	0.112	0.067	0.406
Velocity	m/s	0.248	0.173	1.123	0.326	0.224	1.383
Attitude	deg	4.30	2.97	21.46	9.11	6.55	25.59

Table 50: Deviation between the two estimators for the Bayesian-optimized configuration, combined across all four trajectories. Position and velocity are Euclidean distances, attitude is the Euclidean distance of the Euler-angle vector.

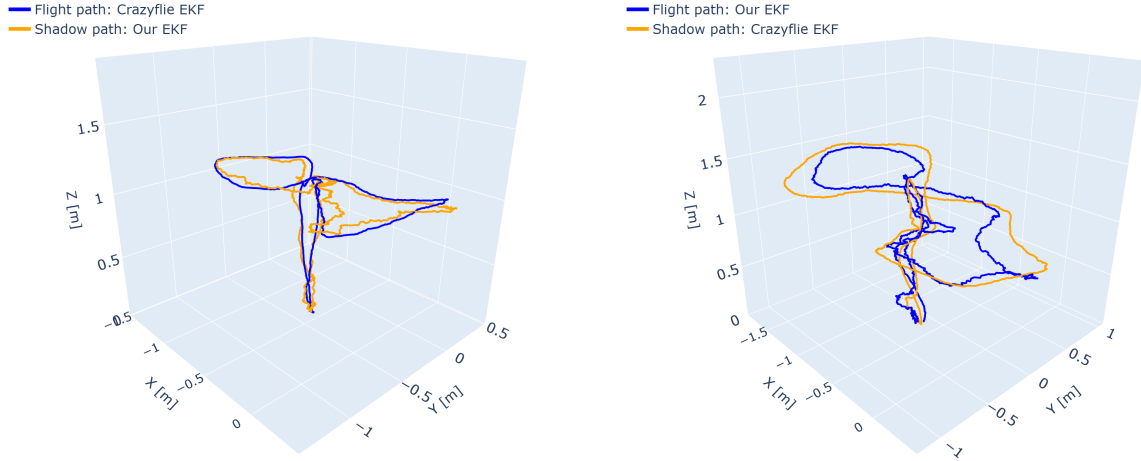


Figure 54: 3D flight path of the figure-8 trajectory. Left: Crazyflie EKF active, our EKF as shadow. Right: our EKF active, Crazyflie EKF as shadow.

7.5.1.2 UWB Measurement Arrival Timing

The poor initial results with the Bayesian-optimized parameters raised the question of how measurements actually arrive on real hardware. The simulation delivers all sensor measurements simultaneously at each time step, whereas the actual behavior on real hardware was unknown. To characterize the timing, we recorded the arrival timestamps of all eight UWB anchors and the ToF sensor during a stationary test. The ToF sensor delivers measurements at approximately 23 Hz and each UWB anchor independently at approximately 20 – 22 Hz. Within a single measurement cycle, the eight anchor readings are spread over up to 41 ms and their arrival order varies between cycles. This is further discussed in [Section 7.7.5](#).

7.5.1.3 Filter Consistency and Parameter Tuning

The Bayesian-optimized standard deviations produced a mean UWB NIS of 54 and a mean ToF NIS of 37, indicating strong overconfidence. We decided to tune the parameters manually on real flights, using the NIS as a consistency metric to guide the tuning process.

We ran targeted flights with several parameter combinations, evaluating the NIS after each run until both sensors reached a mean NIS close to 1, arriving at a UWB standard deviation of 0.18 m and a ToF standard deviation of 0.0055 m. [Table 51](#) reports the mean NIS per used trajectory for this configuration.

Trajectory	UWB NIS	ToF NIS
Figure-8	0.81	0.76
Circle	1.19	1.04
Circle varying height	0.96	1.19
Mean	0.99	1.00

Table 51: Mean NIS per trajectory type with a UWB standard deviation of 0.18 m and a ToF standard deviation of 0.0055 m.

The mean UWB NIS of 0.99 and mean ToF NIS of 1.00 both lie at the expected value of 1, indicating that the chosen parameters produce a better-calibrated filter on real hardware than the Bayesian-optimized parameters. The root cause of this discrepancy is discussed in [Section 7.7.5](#).

7.5.1.4 Evaluation with Tuned Parameters

With the NIS-tuned parameters identified in [Section 7.5.1.3](#), our EKF flies stably and tracks trajectories comparably to the Crazyflie EKF, with all metrics improving substantially over the Bayesian-optimized configuration. [Figure 55](#) shows the per-axis position estimates and [Figure 56](#) the 3D flight paths for the figure-8 trajectory.

Metric	Unit	Crazyflie EKF active			Our EKF active		
		Mean	Std	Max	Mean	Std	Max
Position	m	0.072	0.051	0.193	0.076	0.061	0.228
Velocity	m/s	0.180	0.126	0.492	0.223	0.170	0.609
Attitude	deg	1.60	1.49	7.35	2.70	2.52	10.45

Table 52: Deviation between our EKF and the Crazyflie EKF with the NIS-tuned parameters, combined across all four trajectories.

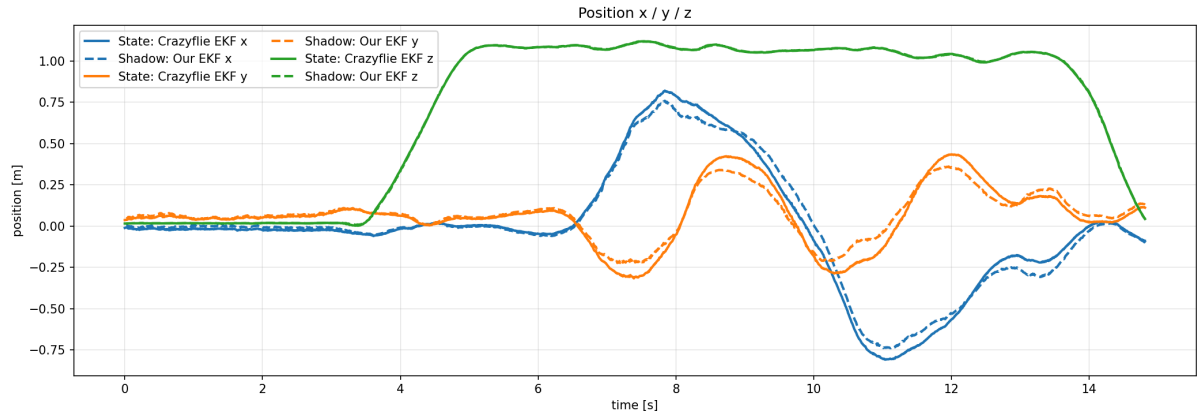


Figure 55: Per-axis position estimates of our EKF and the Crazyflie EKF over time for the figure-8 trajectory with NIS-tuned parameters, Crazyflie EKF active and our EKF as shadow.

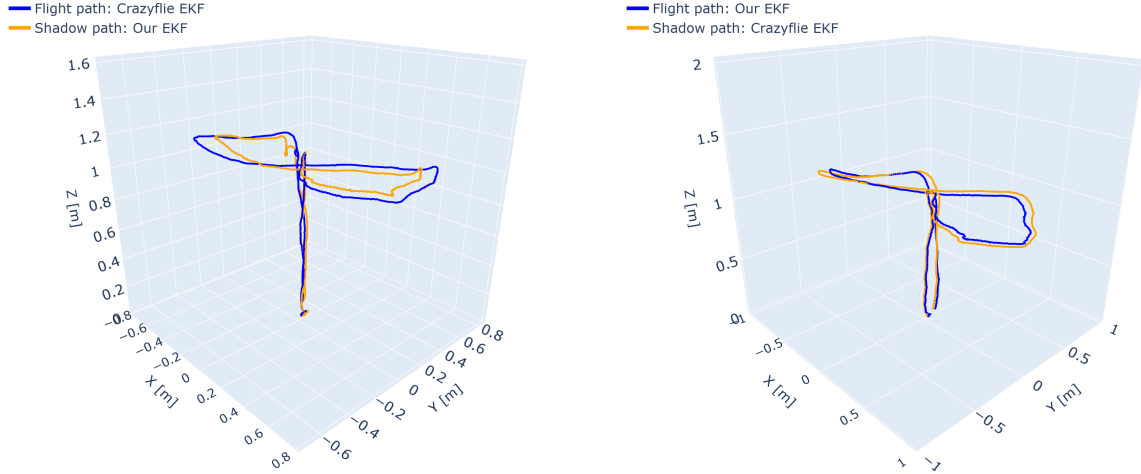


Figure 56: 3D flight path of the figure-8 trajectory with NIS-tuned parameters. Left: Crazyflie EKF active, our EKF as shadow. Right: Our EKF active, Crazyflie EKF as shadow.

7.5.1.5 Contribution of the ToF Sensor to Altitude Estimation

To isolate the contribution of the ToF sensor to altitude estimation, the same NIS-tuned configuration was evaluated with the ToF sensor disabled in our EKF. The Crazyflie EKF continued to use the ToF sensor throughout.

Disabling the ToF sensor causes an increase in z-axis deviation across all trajectories, as shown in [Figure 57](#).

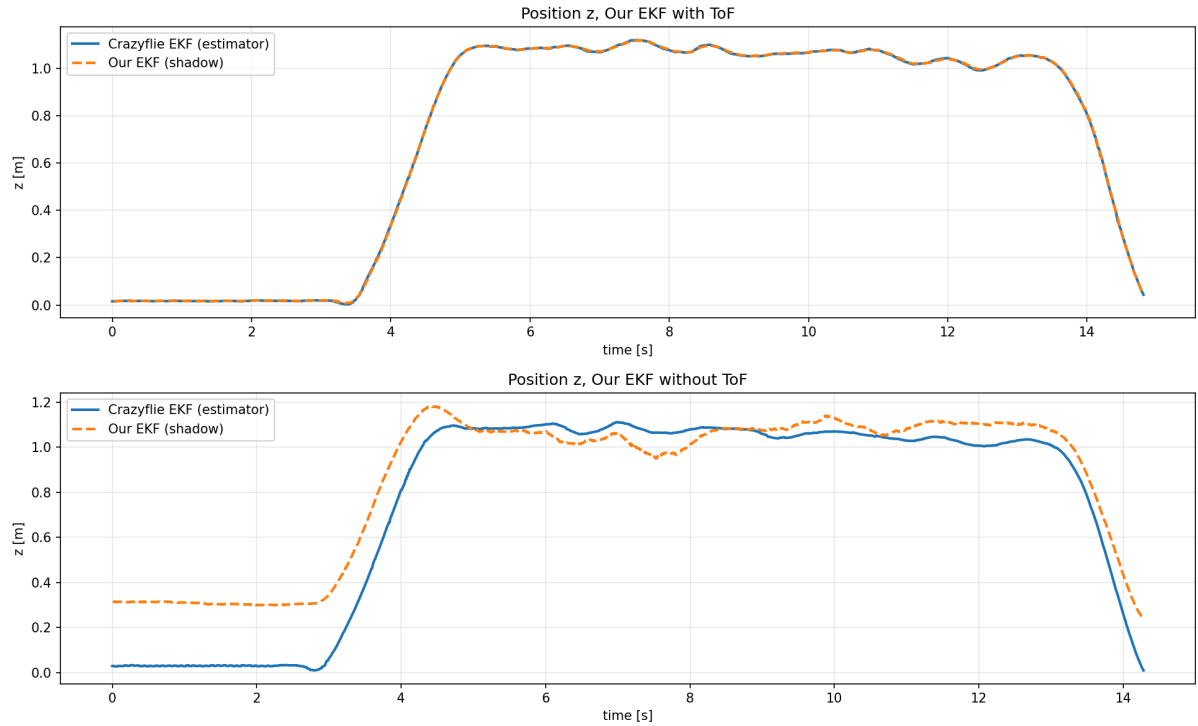


Figure 57: Z-axis position estimates of our EKF and the Crazyflie EKF over time for the figure-8 trajectory, with and without the ToF sensor enabled in our EKF.

This confirms that the ToF sensor is the primary driver of accurate altitude estimation in our EKF. The implications of this are discussed in [Section 7.7.6](#).

7.5.1.6 Evaluation with Adaptive Measurement Noise

The sliding window variant uses a window of 105 samples (approximately 5 seconds) for UWB and 115 samples (approximately 5 seconds) for ToF, based on the measured arrival frequencies reported in [Section 7.5.1.2](#). The EMA variant uses a smoothing factor of $\alpha = 0.05$.

Both adaptive variants perform worse compared to the NIS-tuned parameters found in [Section 7.5.1.3](#). The position deviations to the Crazyflie EKF are roughly three times larger, as shown in [Table 53](#).

Metric	Unit	Crazyflie EKF active			Our EKF active		
		Mean	Std	Max	Mean	Std	Max
Sliding Window							
Position	m	0.211	0.079	0.507	0.221	0.083	0.619
Velocity	m/s	0.255	0.183	0.880	0.299	0.242	1.223
Attitude	deg	2.09	1.48	6.44	3.14	2.23	9.16
EMA							
Position	m	0.217	0.094	0.525	0.217	0.087	0.515
Velocity	m/s	0.194	0.136	0.597	0.196	0.149	0.640
Attitude	deg	2.04	1.56	7.18	1.80	1.35	6.87

Table 53: Deviation between our EKF and the Crazyflie EKF for the adaptive sliding window and EMA configurations, combined across all four trajectories.

The estimated UWB measurement noise standard deviation $\sqrt{R}$ ranges from roughly 0.02 m to 0.5 m, varying across anchors and over time. [Figure 58](#) shows a representative example of the per-anchor measurement noise standard deviation $\sqrt{R}$ over time for a figure-8 trajectory. The cause of this instability is discussed in [Section 7.7.4](#).

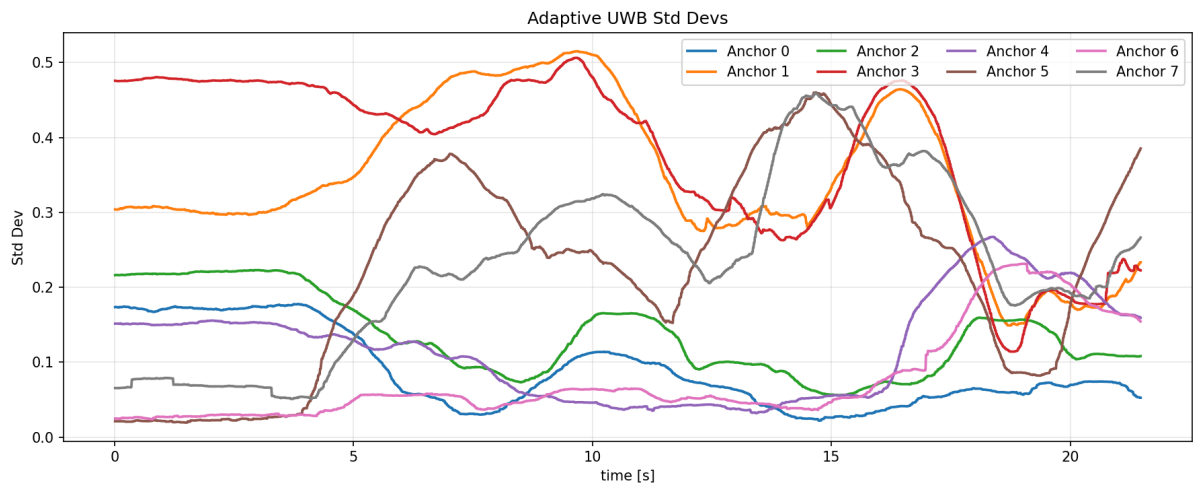


Figure 58: Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the Crazyflie.

7.5.2 Sounding Rocket Results

We evaluated our EKF on the ASTERIA sensor carrier board during a series of car drives. The SD card interface was used to record sensor data during the drives. Since no ground-truth data is available for the car drives, the output of the EKF is compared against the recorded GNSS navigation data. GNSS provides only position and velocity, so the attitude estimate of our filter cannot be validated through this comparison. Additionally, the GNSS navigation data has varying accuracy during the drive due to nearby obstructions such as houses and other vehicles. This evaluation is a check on the general functioning of the EKF on the ASTERIA sensor carrier board and validates its overall performance.

Unlike the simulation configuration described in [Section 7.3](#), the ASTERIA firmware performs a bias compensation, as detailed in [Section 6.1.4](#).

We assembled a housing, shown in [Figure 59](#), to keep the sensor carrier board and the two GNSS antenna boards in place.



Figure 59: Housing used to hold the ASTERIA boards in place during the car drives.

The car drives were conducted in a city environment on a circular route that included both downhill and uphill sections. The drives were affected by stops for traffic and pedestrians and thus vary slightly. We evaluated three scenarios: one with Bayesian-optimized parameters, one using adaptive measurement noise with the sliding-window variant and one using adaptive measurement noise with the EMA variant.

7.5.2.1 Evaluation with Bayesian-Optimized Parameters

The parameters used here are taken directly from the Bayesian optimization described in [Section 7.2](#) ([Table 41](#)).

The EKF closely follows the GNSS path in the horizontal plane, as visible in [Figure 60](#) and [Figure 61](#). The largest deviations occur in the vertical axis, where the EKF shows several jumps that are not present in the GNSS data.

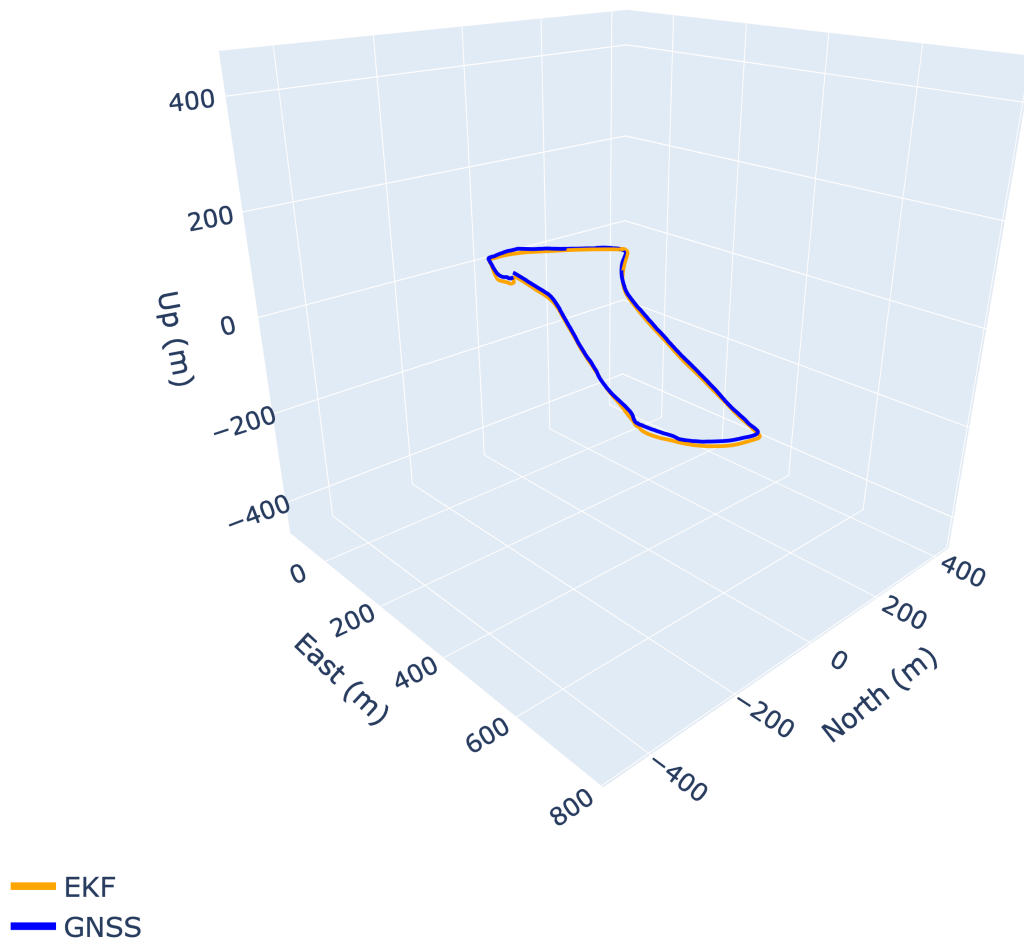


Figure 60: 3D plot of the GNSS path and our EKF path recorded on the sensor carrier board.

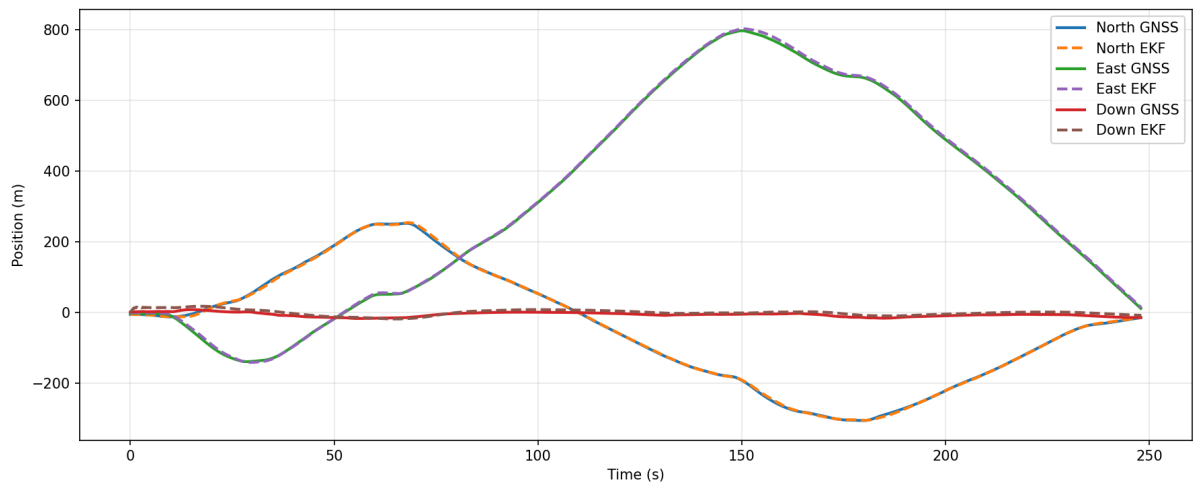


Figure 61: NED axis position comparison between the GNSS path and our EKF path recorded on the sensor carrier board.

[Table 54](#) summarizes the deviations observed between the GNSS measurements and our EKF estimates.

Metric	Unit	EKF		
		Mean	Std	Max
Position	m	7.647	2.415	14.011
Velocity	m/s	1.525	1.361	15.101

Table 54: Deviation between the GNSS measurements and our EKF running the Bayesian-optimized configuration. Position and velocity are Euclidean distances.

A quick analysis yields a mean NIS of 2.30. For a three-dimensional GNSS position measurement the expected value of a consistent filter is 3, indicating that the filter is slightly conservative. No further tuning for consistency was performed.

7.5.2.2 Evaluation with Adaptive Measurement Noise

[Table 55](#) shows the deviations to the GNSS data for both adaptive configurations. Only the measurement noise R is adapted here, the process noise Q remains static and identical to the Bayesian-optimized configuration.

Both variants generally function correctly, tracking the GNSS path without divergence but produced an overall worse position and velocity estimate than the Bayesian-optimized configuration.

Metric	Unit	EKF		
		Mean	Std	Max
Sliding Window				
Position	m	9.437	3.875	26.460
Velocity	m/s	1.154	1.181	6.674
EMA				
Position	m	8.040	9.342	41.927
Velocity	m/s	1.617	1.889	8.398

Table 55: Deviation between the GNSS measurements and the state estimator for the adaptive noise configurations. Position and velocity are Euclidean distances.

[Figure 62](#) shows how the standard deviation of the GNSS measurement noise $\sqrt{R}$ changed over time for the sliding-window variant. During the drive $\sqrt{R}$ varied considerably, falling below and rising above the statically tuned value. Spikes in $\sqrt{R}$ reached the upper bound and coincided with turns where the drive path reversed direction along an axis, as seen in [Figure 61](#).

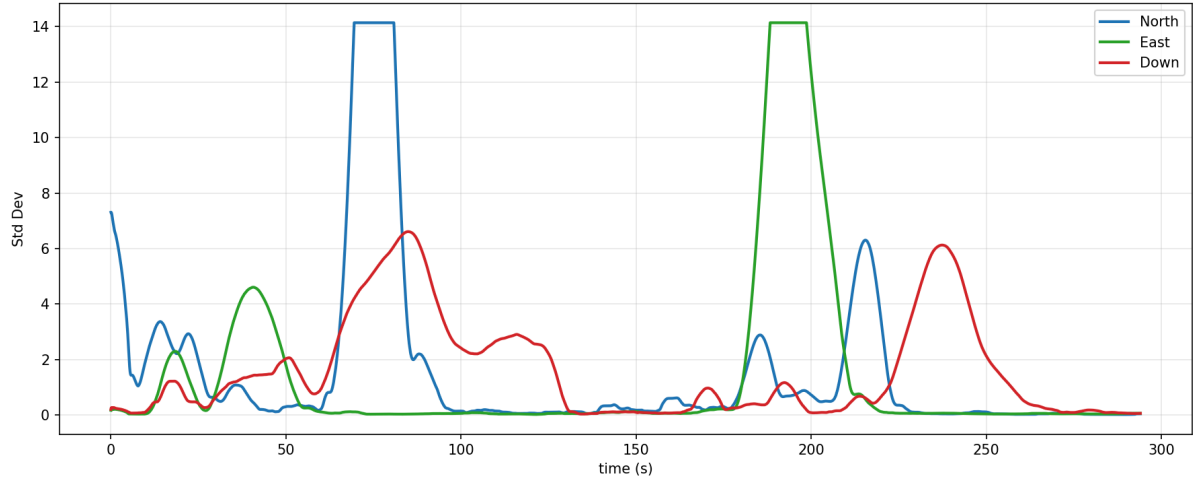


Figure 62: Estimated measurement noise standard deviation $\sqrt{R}$ over time for the sliding window variant on the ASTERIA sensor carrier board.

7.6 Real-Time Performance

This section addresses [MHR-04](#) by verifying that the EKF produces state estimates within the time budget imposed by the sensor rates on each platform.

7.6.1 Crazyflie Execution Time

To verify that our EKF (RKF) meets the real-time requirements of the Crazyflie platform, the execution time of the three core filter functions was measured during a 10-second flight test. [Figure 63](#) shows the measured execution times per function call and [Table 56](#) summarizes the average and peak values.

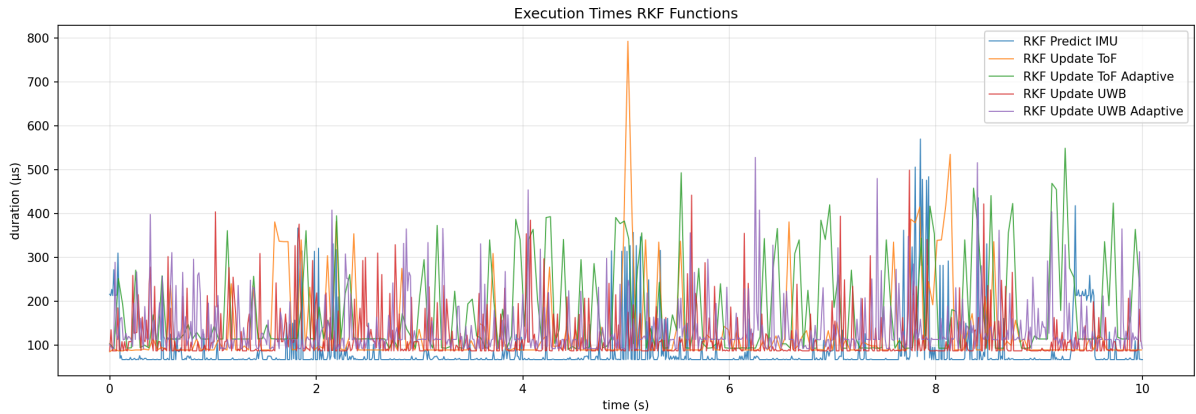


Figure 63: Measured execution times of our EKF functions on the Crazyflie during a 10-second flight test.

Function	n	avg	max	mode
RKF Predict IMU	998	87.45 μ s	570 μ s	67 μ s
RKF Update UWB	763	116.93 μ s	499 μ s	87 μ s
RKF Update UWB Adaptive	762	135.06 μ s	528 μ s	113 μ s
RKF Update ToF	238	131.89 μ s	793 μ s	89 μ s
RKF Update ToF Adaptive	233	176.42 μ s	549 μ s	93 μ s

Table 56: Execution times of our EKF functions on the Crazyflie during a 10-second flight test.

The critical deadline for each function call is 1 ms, because the Crazyflie’s stabilizer loop runs at 1000 Hz. All five functions remain well below that deadline in both average and peak cases, confirming that our EKF can run in real time on the Crazyflie. The mode is a more representative measure of typical performance than the average, as occasional spikes pull the average upward. The correction steps are consistently more expensive than the predict step, because they additionally require computing the Kalman gain and updating the covariance matrix. The adaptive variants carry a small extra cost due to their noise estimation. The occasional peaks are caused by higher-priority interrupts blocking the estimator, the worst observed peak still stays well within the 1 ms budget.

7.6.2 ASTERIA Execution Time

To verify that the EKF implementation meets the real-time requirements of the ASTERIA platform, the execution time of the two core filter functions was measured during a car drive. [Figure 64](#) shows the measured execution times per function call and [Table 57](#) summarizes the average and peak values.

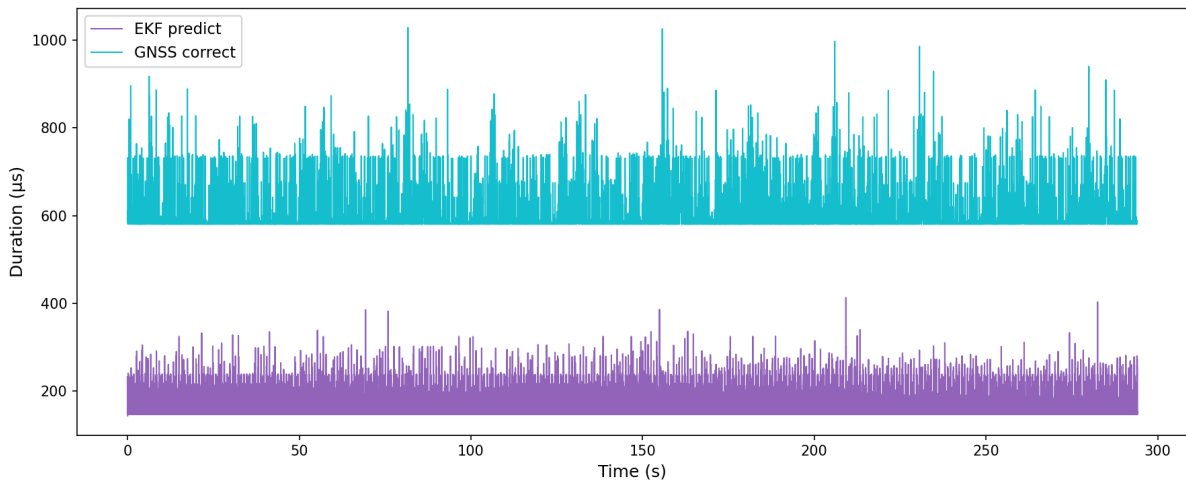


Figure 64: Measured execution times of the EKF functions on the ASTERIA sensor carrier board.

Function	n	avg	max	mode
EKF Predict IMU	195809	153 μ s	409 μ s	148 μ s
EKF Update GNSS	9916	573 μ s	985 μ s	547 μ s
EKF Update GNSS Adaptive	11821	612 μ s	1033 μ s	589 μ s

Table 57: Execution times of the EKF functions on the ASTERIA sensor carrier board.

The predict step was called on every incoming IMU sample at 790 Hz, giving a per-call budget of 1.27 ms. The GNSS update is called at 40 Hz⁸, giving a per-call budget of 25 ms. Both functions remain well below their respective budgets in both average and peak cases, confirming that the EKF can run in real time on the ASTERIA sensor carrier board. Updating the adaptive variants takes about 7.7% longer due to noise estimation.

7.7 Discussion

This section interprets the evaluation results from the previous sections.

7.7.1 Missing Bias Estimation

The parameter search on simulated data converged near the upper bound of the accelerometer and gyroscope ranges on both platforms (see [Table 41](#) and [Table 42](#)). This is a strong sign that the optimal parameters lie outside the search range and that the filter compensates for an error source it does not model, which we attribute to the IMU bias. With no bias states, the filter absorbs the inertial bias into its process noise.

Even at this upper bound, the noise parameters remain too small to account for the true error, so the filter reports a state uncertainty smaller than the real estimation error [\[13, Section 3.1.1\]](#). The estimate stays usable, but the filter becomes overconfident, which is visible in the NEES results ([Figure 34](#), [Figure 44](#)).

The real Crazyflie flights confirm that usable estimates are achievable without dedicated bias states. The Crazyflie firmware already reduces the inertial bias through its sensor preprocessing (see [Section 6.2.1.3](#)) and together with measurement noise parameters tuned for filter consistency, our EKF reached an accuracy comparable to the Crazyflie EKF. This demonstrates that the missing bias states are a limitation of the current filter, but not one that necessarily prevents reliable operation in practice when the bias is handled by a simple preprocessing step.

We implemented dedicated bias states for the filter but could not validate them within the scope of this thesis (see [Section 8.2.2](#)).

7.7.2 Heading Observability and the Missing Magnetometer

The attitude results show a consistent pattern across both platforms in simulation. The error about the x and y axes stays small, while the error about the z axis is several times larger (see [Table 43](#) and [Table 45](#)). The same effect appears on real Crazyflie flights.

This pattern is a direct consequence of the yaw observability limitation described in [Section 3.8.2](#). No sensor in our suite observes an absolute heading, so the z -axis orientation cannot be corrected by a measurement update. Without an absolute reference the heading

⁸The ASTERIA team uses a configuration of the GNSS modules that differs slightly from the one assumed in the simulation [Section 4.1.3.4](#).

drifts due to the unmodeled gyroscope bias (see [Section 7.7.1](#)), which explains the large z -axis attitude error.

A magnetometer would close this gap by providing an absolute heading reference. The Crazyflie does not include one, but the ARIS rocket carries a magnetometer that could be integrated as an additional measurement update. Two challenges should be noted in practice. Calibrating a magnetometer is non-trivial, particularly on a rocket, where surrounding metal components distort the magnetic field. The sensor may also become unreliable during the high accelerations and vibrations of powered flight, so its benefit on the rocket would need to be verified experimentally.

A barometer would similarly improve altitude estimation by adding a direct measurement of vertical position. Integrating these sensors and evaluating their effect remained out of scope for this thesis, but as demonstrated with the ToF sensor (see [Section 7.7.6](#)), the EKF architecture and the simulation we developed support the addition of further sensors without structural changes.

7.7.3 Behavior During GNSS Outage and Across Flight Phases

The phase breakdown in [Table 44](#) shows that the coast phase has the largest errors for the ARIS rocket, with a position RMSE of 42.78 m and an attitude RMSE of 40.30°, both several times higher than in the other phases. This coincides with the simulated GNSS-outage segment that lasts from launch until apogee (see [Figure 30](#)). During this segment no position measurement is available for correction, so the filter relies entirely on integrating the inertial measurements. The unmodeled inertial bias discussed in [Section 7.7.1](#) accumulates without correction, which drives the position error to grow throughout the outage (see [Figure 31](#)). The large standard deviation of the coast-phase position error across runs reflects that this accumulated drift depends strongly on the trajectory and the duration of the outage, which was varied during the different simulation runs. Once GNSS corrections resume near apogee, the position error drops fast and the filter recovers within a short time.

The boost phase shares the same GNSS outage but reports a much lower error. This happens because of two reasons: The filter enters the phase well-initialized from the GNSS position available before launch and the boost phase lasts only about ten seconds which is too short for the inertial drift to accumulate substantially.

As described in [Section 4.1.4.1](#), RocketPy switches from a 6-DOF to a 3-DOF model at drogue inflation and no longer models the attitude of the rocket. To prevent this from corrupting the evaluation, we freeze the attitude state and zero the attitude error from that point on, which is why no attitude RMSE is reported for the drogue and main-chute phases. The sharp NEES spike at around 42 s ([Figure 34](#)) is an artifact of this transition and does not reflect filter behavior. The coast-phase attitude error itself is consistent with the heading-observability limitation discussed in [Section 7.7.2](#).

7.7.4 Static vs. Adaptive Measurement Noise

On the simulation scenarios the static variant achieved the lowest errors and both adaptive variants performed worse without bias compensation (see [Table 47](#) and [Table 48](#)). This outcome is expected, since the static measurement noise R was tuned on exactly these scenarios through the parameter search. When we apply bias compensation to the input data, as done in the Crazyflie experiment, the adaptive variants improve and perform comparably to the static variant.

One notable advantage of the adaptive methods is their ability to represent conditions that a static R cannot, particularly when measurement noise changes during a run. The changing-variance scenario in [Section 7.4.3](#) demonstrates this. When the UWB noise was scaled up and down across the run, both adaptive variants tracked the change and achieved a lower error than the static variant (see [Table 49](#)). This result rests on a single, deliberately constructed scenario and therefore indicates the potential of the adaptive approach rather than proving a general advantage.

The two adaptive variants differ in how they react to such changes. The sliding window reacts more slowly, adjusting over roughly its window length, but produces a stable estimate of R . The EMA variant reacts faster but provides a noisier estimate (see [Figure 49](#) and [Figure 50](#)). The sliding window additionally requires storing the most recent measurement innovations per sensor, whereas the EMA keeps only a running estimate. Their runtime cost is comparable and well within the real-time budget (see [Section 7.7.8](#)).

7.7.4.1 Transfer to Real Hardware

The adaptive variants did not transfer well to the real hardware. On both the ARIS rocket and the Crazyflie the position estimates deviated more strongly from the reference than with the static parameters.

The innovation-based estimation of R rests on the assumption that the measurement innovation sequence behaves as unbiased, Gaussian white noise [\[28\]](#). Under this assumption, the spread of the measurement innovations reflects measurement noise alone. A bias in the measurement innovations breaks the zero-mean part of this assumption and correlated measurement innovations break the white-noise part. Based on our results, we identify two sources of such non-ideal measurement innovations on the real hardware.

The first source, present on both platforms, is bias in the IMU data. In simulation, the adaptive variants improved once the IMU bias was removed through bias compensation, as shown in [Figure 48](#). We apply such a bias compensation step on both the ARIS rocket and the Crazyflie, but on real hardware this compensation may be less effective than the exact bias removal available in simulation, so a measurement innovation bias is likely still present.

The second source, observed only on the Crazyflie, is the structure of the real UWB measurement noise. As described in [Section 7.7.5](#), the real UWB noise does not match the simplified white-noise model applied in simulation. In simulation, the adaptive variants behaved as intended. On the real Crazyflie, the estimated R per anchor varied widely across the anchors and over time and showed visible correlations between certain anchors (see [Figure 58](#)). We therefore attribute this second source to the real UWB noise, which is more complex than the white-noise model used in simulation.

On the ARIS rocket, the estimated GNSS measurement noise $\sqrt{R}$ showed a similar instability. We suspect that analogous mechanisms apply, namely residual IMU bias and a mismatch between the simulated GNSS model and the real measurement behavior, such as timing delays not captured in simulation. We did not verify this further, so it remains a plausible explanation rather than a confirmed result.

We interpret these effects as instances of the same underlying issue. On the real hardware the measurement innovations are no longer the unbiased white noise that the innovation-based estimation assumes. The method we built on addresses this case directly by performing an autocorrelation check on the measurement innovations and suspending the adaptive update once the measurement innovations are no longer consistent with unbiased white noise [\[28\]](#). We did not include such a consistency check in our variants due to time constraints. Imple-

menting such a check could improve the reliability of the adaptive estimate and is left as future work.

7.7.5 Simulation-to-Real Gap in UWB Noise

The UWB standard deviation parameter obtained through the Bayesian optimization was selected too small for the real Crazyflie. In simulation, all UWB anchor measurements arrive simultaneously and share a simple noise model. On real hardware, measurements arrive asynchronously and in varying order across anchors (see [Section 7.5.1.2](#)) and their noise characteristics differ from the simple model used in simulation. With an overly small measurement noise, the filter trusts each incoming measurement too strongly, causing an over-correction toward whichever anchor just reported. The result is a position estimate that jumps repeatedly in different directions, producing the noisy trajectory observed in [Section 7.5.1.1](#).

This gap is the reason the Bayesian-optimized UWB standard deviation produced a mean UWB NIS of 54 on real hardware, indicating extreme overconfidence, whereas the manually NIS-tuned value of 0.18 m brought the mean NIS close to its expected value of 1 (see [Section 7.5.1.3](#)). The same gap provides an explanation on why the adaptive variants did not transfer to the real Crazyflie (see [Section 7.7.4](#)).

Closing this gap requires a more accurate UWB model in simulation that captures the asynchronous arrival of anchor measurements and a more realistic noise behavior. With such a model, both the static parameter search and the adaptive variants would operate on measurement innovations more representative of the real hardware, which we expect would substantially reduce the discrepancy observed here.

7.7.6 Extensibility of the Filter and Simulation Architecture

Adding the ToF sensor to the Crazyflie demonstrates the extensibility of both our EKF and our simulation architecture. On the filter side, the ToF sensor required only adding the new measurement with no further changes to the estimator. On the simulation side, we generated the measurement synthetically from the ground-truth data and added it into the existing runner. This did not require any changes to the surrounding simulation or evaluation architectures. In simulation, this reduced the vertical position RMSE from 0.026 m to 0.010 m (see [Figure 45](#)) and the real flights confirmed the effect.

These results confirm that our implementation acts as a flexible foundation that allows for the simple addition of more sensors. The sensors proposed in [Section 7.7.2](#) would follow the same integration path and can be added without structural changes to the filter.

7.7.7 Tuning Methodology and Objective Design

The parameter search in [Section 7.2](#) optimized a single objective: The mean position RMSE across the optimization runs. This choice was simple to implement and produced a configuration that performs well across all evaluated scenarios, but it optimizes position accuracy alone and does not directly account for velocity, attitude, or filter consistency. The overconfidence reported in [Section 7.7.1](#) illustrates the limitation of this choice. A search driven purely by position RMSE does not need to ensure that the reported covariance remains consistent with the actual error.

Our existing infrastructure supports more advanced objectives without modification. A multi-objective search could optimize for position, velocity and attitude accuracy together. Alternatively, a composite objective could combine position error with a consistency measure

such as the NEES, which would directly penalize the overconfidence observed in our results. Exploring such objectives remained out of scope for this thesis and is left for future work.

A practical constraint on any such search is its computational cost. The rocket search required about 13 hours for 50 trials, since each trial runs the filter over all optimization scenarios. A multi-objective or consistency-aware search does not reduce this per-trial cost and typically requires more trials to explore the parameters. Thus, the available compute budget limits how thoroughly the search space can be explored in practice.

7.7.8 Real-Time Feasibility

As reported in [Section 7.6](#), all filter functions on both platforms remain well within their respective time budgets, including the adaptive variants (see [Table 56](#) and [Table 57](#)). The correction step consistently takes longer than the prediction step because it requires the Kalman gain to be computed and the covariance matrix updated. This poses no problem with the current sensor suites. However, as more measurement sources are added, the number of correction steps per cycle will increase, consuming more of the time budget. At that point, it may be worth considering which measurements to include in each cycle, or combining simultaneous measurements into a single update, in order to limit the per-cycle cost.

8 Conclusion and Outlook

This chapter summarizes the insights of the thesis and compares them against the objectives defined in [Section 1](#). [Section 8.1](#) answers the central research question, assesses the adaptivity and measures goal achievement against the requirements. [Section 8.2](#) then outlines the open points and the directions for future work that follow from our results.

8.1 Conclusion

The central research question of this thesis was whether a single EKF-based estimator can be built that adapts, potentially automatically, to IMU-driven aerial platforms operating under different flight dynamics (see [Section 1.2](#)). The following subsections answer this question, assess the software architecture we developed and measure the outcome against the project requirements.

8.1.1 A Single Cross-Platform Estimator

A single, modular EKF was deployed on both the ASTERIA sensor carrier and the Crazyflie 2.1. The two platforms differ in their dynamics and sensor suites (see [Section 2.3](#)), yet both were supported by configuring the same estimator with platform-specific sensors and parameters.

For each platform we simulated a set of scenarios containing a range of different flight conditions. In simulation, the filter estimated the rocket position with a total RMSE of 15.12 m and the Crazyflie position with a total RMSE of 0.0304 m. These figures are not directly comparable. The rocket RMSE is dominated by the simulated GNSS-outage segment from launch to apogee, during which the filter integrates inertial measurements alone, whereas the Crazyflie operates indoors with continuous UWB and ToF corrections. On both simulated platforms the filter was overconfident.

On real Crazyflie flights, our EKF tracked trajectories comparably to the Crazyflie EKF once tuned for filter consistency. The noise parameters obtained from a simulation-based Bayesian search did not transfer directly. They left the filter strongly overconfident on real hardware, producing a noisy position estimate. However, manually tuning the noise parameters for consistency on the Crazyflie restored a stable flight.

Validation on the ASTERIA sensor carrier was limited to car drives, with GNSS serving as the sole reference due to the absence of an independent ground truth. With Bayesian-optimized parameters, the EKF closely followed the GNSS path horizontally. The largest deviations appeared in the vertical axis. The filter exhibited slightly conservative consistency, with a mean NIS of 2.30 against an expected value of 3. These results confirm the general functioning of the EKF on the ASTERIA sensor carrier board, while also indicating that further improvements and testing would be needed for a more accurate state estimation.

Our EKF therefore generalizes across two platforms with different dynamics and sensor suites, which confirms the cross-platform claim at the center of this work. Its main limitation is the absence of bias states and of an absolute heading reference. Further limitations, such as the absence of innovation filtering, IMU vibration mitigation and GNSS time-delay compensation, are detailed in [Section 3.8](#).

8.1.2 Adaptive Measurement Noise Variants

The second part of the research question, automatic adaptation, was only partially achieved. We automatically adapted the measurement noise R but left the process noise Q fixed, as its automatic adaptation could not be implemented within the available time. For the mea-

surement noise adaptation we evaluated a sliding window approach based on [28] and an exponential moving average (EMA) approach.

In simulation, the two adaptive variants performed comparably to each other across the predefined scenarios. They fell short of the static variant without bias compensation and matched it once bias compensation was applied. Their advantage appears only when the measurement noise changes during a run. In a dedicated changing-variance simulation scenario for the Crazyflie both adaptive variants outperformed the static variant, with EMA achieving the lowest error. This rests on a single, deliberately constructed scenario and therefore indicates the potential of the approach rather than proving a general advantage.

On real hardware, both adaptive variants performed worse than static parameters. The innovation-based estimation of R assumes the measurement innovation sequence is unbiased, Gaussian white noise [28]. We identified that this condition is violated on real hardware by unmodeled IMU bias on both platforms and, on the Crazyflie, the more complex structure of the real UWB noise. A comparable instability appeared in the estimated GNSS measurement noise on the ARIS rocket, though we did not investigate this further. The original estimator guards against these cases with an autocorrelation check that suspends the adaptive update once the measurement innovations are no longer white [28], but our implementation omits this check.

Since the two adaptive variants performed comparably in accuracy, we recommend choosing between them based on platform constraints rather than estimation quality. On memory-constrained targets we favor the EMA variant, as it keeps only a running estimate and avoids storing the measurement window. Where memory permits, the sliding window is the better choice, as it produced a smoother and more stable estimate of R .

We conclude that the adaptive estimators are promising in simulation but do not yet transfer reliably to real hardware. For now, manually tuned parameters remain the more reliable choice, while our results indicate that adaptivity may become advantageous once residual IMU bias is removed within the filter and a consistency check guards the update against corrupted measurement innovations.

8.1.3 EKF Architecture, Simulation and Implementation

Beyond the estimator itself, the thesis delivered a modular filter architecture, a simulation environment and an evaluation pipeline that together proved suitable for the task.

The architecture was designed so that sensors can be added as self-contained measurement updates, independent of the core filter. Adding a ToF sensor to the Crazyflie demonstrated this directly. It required only a new measurement update and the simulation side needed only a synthetic measurement added to the existing runner. The addition immediately improved the vertical estimate in simulation and the effect was confirmed on real Crazyflie flights. The same integration path applies to the further sensors discussed in [Section 7.7.2](#).

The EKF was implemented in Rust and integrated into both the ARIS sensor carrier firmware and the Crazyflie firmware without modifying their low-level drivers. The same estimator therefore ran on two very different embedded targets. On the Crazyflie, all filter functions executed well within the 1 ms stabilizer-loop budget, confirming that the implementation meets the platform's real-time requirements. On the ASTERIA sensor carrier, all filter functions likewise remained well below their respective budgets, processing 790 IMU samples per second without issue (see [Table 57](#)).

The simulation environment provided the ground-truth data needed to tune and validate the filter across a range of scenarios on both platforms and its modular structure allowed new

sensors to be added without reworking the surrounding pipeline. Several limitations remain: The oversimplified UWB model that does not reproduce the asynchronous arrivals seen on real hardware, the missing bias compensation step that the real firmware applies and the simplified single-objective design of the Bayesian parameter search.

8.1.4 Requirements and Goal Achievement

Measured against the requirements defined in [Section 1.3](#), we consider all but one must-have requirement achieved. Two of them require a closer look.

The first is offline validation ([MHR-08](#)), which the requirement specified should cover both simulated and recorded real flight data. Validation against simulated data was completed during the evaluation phase. Validation against real flight data, however, was not possible, as no ASTERIA launch took place during this thesis. We also omitted testing against the recorded drop test data, since these tests were performed too late in the project. We therefore consider this requirement partially achieved.

The second is numerical stability ([MHR-15](#)), whose final confirmation would require running the EKF on a real ASTERIA launch, which was not possible during this thesis. It is covered by unit tests for numerical edge cases and by the filter running through all simulated flights and the real Crazyflie flights without diverging. We therefore consider it achieved, although stability under the high accelerations of a real rocket launch remains to be verified.

Of the should-have requirements, the adaptive noise tuning ([SHR-01](#)) was partially achieved, as discussed in [Section 8.1.2](#). Our implementation adapts the measurement noise automatically, but not the process noise. The third simulated platform ([SHR-02](#)) and the hardware-in-the-loop validation ([SHR-03](#)) were not pursued, as we prioritized the adaptive measurement noise methods and the addition of the ToF sensor within the available time.

The thesis thus delivers a working cross-platform navigation EKF in Rust, implemented on both the ASTERIA firmware and the Crazyflie firmware. Alongside it, we built the simulation, evaluation and tuning infrastructure needed to validate and extend the EKF. It additionally provides two variants for automatic measurement noise adaptation, which function as intended in simulation but require further work before deployment on real hardware.

8.2 Outlook

The results point to several directions for future work, most of which follow directly from the limitations identified in [Section 7.7](#).

8.2.1 Validation on Real Flight Data

Further validation on real flight data remains open for the ASTERIA rocket. The filter was validated against recorded data from a driving test, but a full evaluation on real ASTERIA flight data, ideally against an independent reference, would confirm its behavior under real high-dynamic flight conditions.

8.2.2 Bias Estimation and Adaptive Re-Evaluation

The most immediate next step is the integration of bias estimation. We implemented dedicated bias states for the filter but could not validate them within the scope of this thesis. The implementation is available in the `ba-implementation` repository on the `bias-terms` branch. It follows the EKF design described in [\[13, Section 14.2.2\]](#). It adds three bias states for the accelerometer and three for the gyroscope. The IMU predictor has been extended to utilise the corrected inertial measurements and the state transition Jacobian has been updated to include coupling terms for the bias states. Although initial results from a few

simulated scenarios indicate promising improvements in the RMSE, a comprehensive evaluation is still required. Evaluating their effect on accuracy and consistency is expected to address the overconfidence observed in the NEES results (see [Section 7.7.1](#)).

We apply a startup bias-compensation step to the inertial measurements outside the filter on both platforms and our results showed that removing this bias has a clearly positive effect (see [Section 7.4.2.1](#)). This step is worth looking into further, since better bias compensation before the filter would leave it less error to correct for. Once bias states are in place, the adaptive measurement noise variants should be re-evaluated.

8.2.3 Additional Sensors

The state estimate can be improved by adding the sensors already available on the platforms. On the ARIS rocket, a magnetometer would provide the absolute heading reference and a barometer would add a direct measurement of the altitude. On the Crazyflie, the onboard barometer and the optical flow sensor could be added as further measurement updates. All of these sensors can be integrated through the same path demonstrated with the ToF sensor, without structural changes to the filter.

8.2.4 Filter Stability Improvements

Several limitations identified in the EKF design (see [Section 3.8](#)) remain open and would improve the stability of the filter. An innovation-based outlier rejection could be used for updates to guard the filter against corrupted measurements. A compensation for GNSS measurement latency would align delayed position measurements with the corresponding inertial measurements. Under the high accelerations of powered flight, the IMU may additionally experience vibrations that directly affect the state estimate. Low-pass filtering of the inertial signal could mitigate this effect. Together, these measures would improve the filter's stability and accuracy under real flight conditions.

8.2.5 Simulation Improvements

A more realistic simulation would provide further insight into the filter's behavior during offline validation. In particular, a UWB model that reproduces the asynchronous arrival of anchor measurements and a more realistic per-anchor noise behavior would narrow the simulation-to-real gap identified in [Section 7.7.5](#). Similarly, modeling delayed GNSS measurements would make the simulated rocket data more representative of the real platform. Adding a simulated bias compensation step would also better reflect the behavior of the real firmware. These improvements would make both the static parameter search and the evaluations operate on data closer to the real hardware, allowing observations made in simulation to transfer more directly to the real platform.

8.2.6 Tuning Methodology

Finally, the tuning methodology offers room for refinement. The current parameter search optimizes position RMSE alone. A multi-objective or consistency-aware objective, for example one that combines position error with a NEES consistency term, would directly penalize the overconfidence seen in our results. Both approaches are supported by the existing optimization infrastructure (see [Section 7.7.7](#)).

Project Documentation

9 Assignment

This section describes the context and formalities of the thesis assignment. The original assignment can be found in the appendix under [Section I](#) and the requirements derived from it are listed in the product documentation under [Section 1.3](#). The thesis follows the official guideline for Bachelor and student theses of the OST Computer Science degree program.

9.1 Context

This thesis is conducted as a Bachelor thesis in the Computer Science degree program at OST – Ostschweizer Fachhochschule. It builds on the navigation challenges faced by the ARIS sounding rocket team and extends the investigation to a second aerial platform, the Bitcraze Crazyflie 2.1.

The detailed problem description, motivation and goals are presented in the introduction of the product documentation (see [Section 1](#)).

9.2 Formalities

The total workload for a Bachelor thesis is 360 hours per student, corresponding to 12 ECTS credits at 30 hours per credit. This results in a total of 1080 hours.

The thesis started on 16.02.2026 and is due on 26.06.2026. This submission date includes a two-week extension granted to compensate for a delayed start, as the assignment first had to be specified in detail together with the ARIS team during the initial weeks of the project.

10 Project Management

This chapter describes the project organization, planning and team processes used throughout this thesis. It covers the available resources, the project plan (Section 10.2) with its milestones and deliverables, the development process including meeting structures and roles (Section 10.3) and the planning tools used for issue and time tracking (Section 10.4).

10.1 Project Resources

This section outlines the team composition and the material costs associated with the project.

10.1.1 Project Members

The project team consists of three members. Each member contributes a workload equivalent to 12 ECTS credits, which corresponds to approximately 360 hours per person. This results in a total budget of roughly 1080 hours for the entire project.

10.1.2 Project Costs

The OST covered the hardware procurement costs of three additional sensor carrier boards and five GNSS modules, totalling 1034 CHF.

10.2 Project Plan

We developed the project plan during the first weeks of the semester to outline a broad overview of the available time. As shown in Figure 65, we divided the project into 7 iterations, each having a duration of two weeks.

During the first 4 weeks, the thesis lacked a clearly defined task description, so we dedicated the time during this initial phase to outlining the scope with the advisor and the ARIS team as well as to general project setup, planning and research. We reserved the final week for documentation and cleanup as well as for preparation of the presentation and the bachelor's exam.

Due to the extended initial phase required to establish a well-defined task description together with ARIS, a deadline extension of two weeks was granted.

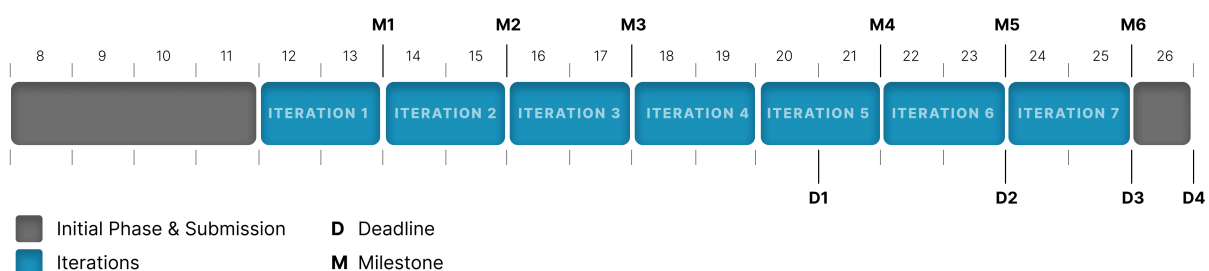


Figure 65: Project plan showing iterations, milestones and deadlines.

10.2.1 Deadlines

The thesis is required to meet the following different deadlines:

- D1** The mid-term presentation (*Fri, 15.05.2026*)
- D2** The submission project A0-poster (*Fri, 12.06.2026*)
- D3** The submission of the abstract (*Mon, 22.06.2026*)
- D4** The final submission of the thesis and all relevant documents (*Fri, 26.06.2026*)

This list does not include the project presentation, as it takes place after the final submission of the thesis.

10.2.2 Milestones

We placed milestones at the end of sprints to validate progress and ensure the project stays on track. They roughly align with the epics we defined in the sprint planning, which in turn reflect the major topics of each sprint.

[Table 58](#) shows the defined milestones for this project. All goals that have been achieved are marked accordingly and each milestone is reviewed once its deadline has passed.

Nr.	Details
M1	Project Setup and Overview 28.03.2026 ☒ Thesis task defined ☒ Project plan, quality assurance and risk analysis documented ☒ Must- and should-have requirements defined ☒ Setup for project tools completed ☒ Basic literature review of EKF theory completed ☒ Available sensors for rocket and Crazyflie analyzed Review <i>review conducted on 27.03.2026</i> • Thesis task is defined and accepted by project members and advisor • Project plan is defined and includes milestones and deadlines. A first draft of the risk analysis was created. The project team decided to do an update of the risk analysis in the middle of the project, once more details about the implementation and potential risks are known. • Must- and should-have requirements are derived from the task description • Project tool setup is completed • A basic literature review of EKF theory was done by every project member • all available sensors for the rocket and the Crazyflie were analyzed and documented
M2	Hardware Bring-up and Validation 11.04.2026 ☐ Sensor carrier board and GNSS modules received and operational ☒ Crazyflie and all needed sensors received and operational ☒ ARIS avionics codebase reviewed and build process understood ☒ Existing Crazyflie EKF implementation and firmware analyzed ☒ Simulation options evaluated and tooling decision made ☒ Previous rocket flight data analyzed Review <i>review conducted on 10.04.2026</i> • GNSS sensors were ordered by ARIS but not yet received. We expect them to arrive in the next weeks. This is currently a non-blocking issue but it is important to have the sensors available as soon as possible to have enough time for testing and integration. • Crazyflie and all needed sensors are operational and have been tested for basic functionality.

Nr.	Details
	• The ARIS codebase has been reviewed and documented. The build process was also analyzed • The Crazyflie firmware and EKF implementation were analyzed and documented. • Simulation tooling decisions for both the rocket and the Crazyflie were made and documented. • Previous rocket flight data was analyzed and evaluated.
M3	Architecture & First Integration 25.04.2026 ✓ First version of EKF model defined for the rocket ✓ First version of EKF model defined for the crazyflie ✓ Architecture for the generic Rust EKF library defined ✓ System architecture for rocket EKF implementation defined ✓ System architecture for Crazyflie EKF implementation defined ✓ Initial rocket simulation environment running ✓ Initial Crazyflie simulation environment running Review <i>review conducted on 24.04.2026</i> • A first version of the EKF model was defined and implemented in code. The model is still work in progress and does not perform well yet. Thus, the documentation part was delayed until the model is well defined to avoid having to redo the documentation. • Architecture of the generic Rust EKF was defined and the current version was documented. • The system architecture for the rocket and the Crazyflie EKF implementations were defined and documented using C4 diagrams. • The initial simulation environments for the rocket and the Crazyflie were set up and can be used to produce sensor data alongside ground-truth data.
M4	Core Application Logic 23.05.2026 ✓ Embedded Rust EKF integrated into the ARIS avionics firmware ✓ Embedded Rust EKF integrated into the Crazyflie firmware ✓ Rocket EKF prototype validated offline against synthetic data ✓ Crazyflie EKF prototype validated offline against synthetic data ✓ Sensor dropout and availability handling implemented ✓ Cross-platform filter behavior comparison started ✓ Report structure and chapter outlines defined Review <i>review conducted on 22.05.2026</i> • The embedded Rust EKF was integrated into the ARIS avionics firmware and into the Crazyflie firmware. The filter is operational and will be evaluated and optimized in the next iteration. • The rocket and Crazyflie EKF prototypes were validated offline against synthetic data. The results match the expected behavior of the current filter implementation.

Nr.	Details
	• Sensor dropout was addressed and the filter implementation can handle sensor dropouts. • The comparison and filter evaluation was started but is not yet completed. The documentation of the evaluation and comparison will be done in the next iteration. • The report structure was defined and the chapter outlines were created.
M5	Evaluation & Optimization 06.06.2026 ☐ Subsystem-level validation completed for both rocket and Crazyflie ☒ Filter performance evaluated with defined metrics ☒ EKF behavior compared across platforms and results documented ☒ Documentation of design rationale, test results and improvements in progress ☒ A0-poster design has started (deadline: Fri, 12.06.2026) Review <i>review conducted on 05.06.2026</i> • The evaluation of the filter performance is still in progress. All flight recordings for the Crazyflie are done but still need to be documented. The evaluation of the filter performance for the ARIS rocket is still in progress. The expected completion date of the evaluation is Friday, 12.06.2026. • The filter performance was successfully evaluated on the simulation data, both for the ARIS rocket and the Crazyflie. • The simulation results were documented and the filter behavior was analyzed across both platforms. The documentation does not include a direct comparison of the two implementations because it would not add much value. The Discussion section of the thesis will include a details about the differences and similarities across both platforms. • The documentation of the design rationale and the test results is completed. Improvements will be included in the discussion and outlook chapter of the thesis and are still work in progress. • The design of the A0-poster has started and a first draft is expected to be completed by Monday, 08.06.2026.
M6	Deliverables & Submission 20.06.2026 ☒ Abstract ready for submission (deadline, Mon, 22.06.2026) ☒ Thesis mostly ready for submission (deadline: Fri, 26.06.2026) Review <i>review conducted on 20.06.2026</i> • Feedback to the abstract was provided by the advisor and incorporated into the current version. The abstract is ready for submission. • The thesis is mostly ready for submission. The core content is completed. What remains to be done is a finalization of the discussion and outlook chapter, a final proofread and a final formatting check.

Table 58: Project milestones and their respective goals and reviews.

10.2.3 Deliverables

[Table 59](#) defines the deliverables to be produced and submitted as part of this thesis.

Deliverable	Description
Thesis report	Technical document and project documentation.
Plain-text and brochure-abstract	Abstracts used for internal thesis publication.
A0-poster	Visual summary of the thesis for the thesis exhibition.
Source code	Embedded EKF in Rust (ARIS), adapted EKF for Crazyflie and produced code for simulation and evaluation.
Log files and recordings	Flight recordings and log files for evaluation and comparison of the EKF implementations.

Table 59: Overview of project deliverables.

10.3 Processes, Meetings and Roles

This section describes the processes, meeting structures and roles we used throughout the semester.

10.3.1 Processes

We loosely followed an agile development process inspired by Scrum [44] and the Agile Manifesto [45], using *Jira* issues to track tasks and progress across iterations (see Section 10.4). Instead of adopting the full Scrum framework, we adjusted the process to match the scale of our project and to minimize administrative overhead.

We organized the work in *two-week sprints*, planning upcoming tasks together at the start of each sprint cycle. Every Friday, we met on campus to collaborate on the thesis, refine the planning for upcoming sprints and coordinate within the team.

10.3.2 Meetings

We defined regular meetings throughout the course of this project, as listed in Table 60.

Meeting	Frequency	Participants	Purpose
Stand-up	Weekly (Fridays)	Project team	Update on current tasks within the project team.
Sprint retrospective & planning	Bi-weekly (Fridays)	Project team	Retrospective of completed iteration and plan upcoming sprint
Advisor meeting	Bi-weekly	Project team, advisor	Discuss project state and open questions
ARIS meeting	Bi-weekly (alternating)	Project team, ARIS team	Coordinate ARIS-related aspects

Table 60: Overview of recurring project meetings.

10.3.3 Roles

Each team member took on the role of technical lead for one of the three core areas: The ARIS rocket, the Crazyflie drone and the simulation. The divided responsibilities are shown in [Table 61](#). The technical lead was responsible for driving the design, implementation and documentation of their area and acted as the go-to person for decisions within that domain.

Project Member	Technical Lead		
	ARIS Rocket	Crazyflie Drone	Simulation
Sahithyen Kanaganayagam	X		
Luca Moor		X	
Florian Eugster			X

Table 61: Assignment of technical leads to the three core areas.

Every member should maintain a solid understanding of the other two areas through joint discussions, code reviews and documentation. This way, no area depends on a single person and everyone can contribute to all parts of the project if needed.

10.4 Planning Tools

The following section outlines the planning-tools utilized during the thesis.

10.4.1 Issue Tracking

To track and plan sprints, Jira is used. All tasks are recorded and organized in a kanban board. Each ticket includes a description and is assigned to the person responsible for completing it.

We reviewed, refined and assigned tickets during sprint planning every two weeks. Once a sprint was running, the scope was fixed. Exceptions are made only for critical tasks.

The used ticket-types are described in [Table 62](#).

Ticket-type	Description
Epic	A large body of work that can be broken down into smaller pieces. Epics represent a high-level feature or goal spanning multiple sprints.
Story	A requirement describing a desired outcome or feature.
Task	A concrete unit of work that needs to be completed, not directly tied to a user story.
Bug	An unintended defect or error in the software that causes incorrect or unexpected behavior.
Sub-Tasks	A smaller piece of work that breaks down a Story or Task, assigned and tracked individually.

Table 62: Used Jira ticket-types.

10.4.2 Time-Tracking

We handled time-tracking via the Jira plugin *Timetracker*. We estimated each ticket upfront and logged time as we worked on it.

To allow for a structured evaluation, time entries are tagged by activity type. The following tags are used:

- Administration
- Research
- Documentation
- Development
- Testing
- Meeting
- Review

The project overview of the time tracking can be found in [Section 13](#). Some tickets are unassigned and serve as shared collection tickets used by all project members throughout the entire project duration. We did not track the time spent on the final presentation, as it takes place after thesis submission.

11 Quality Assurance

This section describes the measures we took to ensure the quality of both the software artifacts and the written documentation produced throughout this thesis.

11.1 Organisational Measures

To ensure high quality throughout the project, we took several organisational measures.

Meetings are held on regular occasions and take place within the project team, with the thesis advisor and with ARIS. Important meetings are protocolled to document decisions and open action items. A description of all meetings can be found in [Section 10.3.2](#). These meetings ensure that all stakeholders share the same understanding of the current project state and that potential issues are identified early.

The development process follows an agile, Scrum-inspired methodology (see [Section 10.3](#)). We organised work into short iterations centered around stories. We implemented, tested and reviewed each story within its iteration rather than deferring all testing to the end of the project. This approach allows for incremental validation of design decisions and for iterative adjustment of the project direction based on concrete results.

11.2 Requirements and Story Definition

To ensure that all requirements are well defined and tracked, we created a comprehensive list of must-have and should-have requirements based on the task description (see [Section 1.3](#)).

We further decomposed requirements into epics and stories, which we recorded and tracked in Jira (see [Section 10.4.1](#)). Each story includes a short task description as well as a definition of done (DoD) (see [Section 11.2.1](#)). This ensures that the expected outcome of every story is defined in advance and unfinished stories are not closed.

11.2.1 Definition of Done (DoD)

The DoD describes the acceptance criteria that must be met before a story is considered complete. Each story defines its own specific acceptance criteria based on the specific task. In addition to these story-specific criteria, the following general criteria must be met for each story:

- All acceptance criteria defined in the story are fulfilled.
- Relevant documentation is updated if necessary.
- Code reviews are conducted if they are required (see [Section 11.3.1](#)).
- The corresponding branch is merged into the main branch and the CI/CD pipeline passes without errors.

11.3 Guidelines

We defined a set of guidelines at the beginning of the project to maintain consistency across both code and documentation. These guidelines cover review practices, commit conventions and code quality.

11.3.1 Code Review

For the **implementation project repository**, the author of a change decides whether a review is necessary. If the author considers a review beneficial, another team member reviews the change before they are merged.

All code committed to the thesis **documentation repository** must be reviewed via merge request or in person by at least one other team member before merging. This ensures that written content in the report is accurate and well written. This also helps distribute knowledge across the team.

11.3.2 Branches and Commit Conventions

We tracked all open tasks and known issues in Jira (see [Section 10.4.1](#)). To maintain a structured working environment, we adhered to the following conventions inside the thesis repositories:

Guidelines for Commits:

- `TODO` or `#reminder()` comments in the source code or documentation must reference the corresponding Jira ticket number (e.g., `// TODO [BA-12]: handle failure`).
- Commit messages must contain a short description or list of what was changed as well as the corresponding Jira ticket number (e.g., `Add error handling, Issue: BA-12`).
- After a commit is made it is the author's responsibility to check the CI/CD pipeline.

Guidelines for Branches:

- Each Jira ticket gets its own branch that is merged after the ticket is completed.
- Each branch is prefixed with its Jira ticket number (e.g., `BA-12-error-handling`).

11.3.3 Code Quality

The code produced during the thesis generally follows principles stated in the book “Clean Code” [\[46\]](#).

Where applicable, established language-specific conventions are followed. Rust code adheres to the official [Rust API Guidelines](#) and Python code follows [PEP 8](#).

Static code analysis is performed through language-specific linting tools. Rust code is analyzed with `clippy`, Python code is checked with `flake8` and C code is compiled with strict warning flags enabled. This ensures that style violations and common errors are caught before committing code.

11.3.4 Continuous Integration

The project is split across two separate repositories, one for the implementation and one for the documentation. Each repository has its own GitLab CI/CD pipeline.

The implementation repository runs a pipeline that enforces the code quality automatically on every push. The pipeline runs in three stages. The first stage tests linting on both code bases, checking the Python simulation with `flake8` and the Rust EKF with `cargo clippy` and `cargo fmt`. The remaining stages execute all test, running the Python tests with `pytest` and the Rust tests with `cargo test`. A passing pipeline is part of the definition of done (see [Section 11.2.1](#)), so a branch can only be merged into main once all stages succeed.

The documentation repository runs a pipeline that compiles the Typst documentation automatically, producing a full report and a condensed version for the eprints publication. Building both variants verifies that the documentation has compiled successfully and identifies errors such as broken references or missing figures. A documentation branch can only be merged once both versions have compiled successfully.

11.4 Testing Strategy

The testing strategy operates on two levels: unit tests that isolate individual components and manually performed integration tests that validate the system as a whole using the simulated data.

Unit tests are written in the Rust EKF library and in the Python simulation modules. The Rust tests verify the physical correctness of the IMU predictor using scenarios and the mathematical correctness of each measurement model in isolation. Where applicable, the Python simulation modules are unit-tested at the level of their individual strategies and components.

The integration tests are carried out through simulation. Several predefined scenarios provide controlled, reproducible input data against which the EKF implementation is tested. Different scenarios cover different dynamics and conditions such as varying trajectories, sensor dropout or amplified noise. Executing the EKF on this simulated data verifies that the components behave correctly and establishes a baseline for analysing improvements or regressions in the filter results.

To ensure the simulated data itself is sound before it reaches the EKF, the Python simulation applies validation checks to each run (see [Section 5.2.6](#)). These checks confirm structural properties such as correct array shapes and dimensionality and guard against invalid values that would otherwise propagate into the filter results.

12 Risk Management

This section describes the potential risks we identified at the beginning of the project and the measures we took to mitigate them [Section 12.1](#). We reviewed the risks at the midpoint of the project (see [Section 12.2](#)) and at the end (see [Section 12.3](#)) to re-evaluate their impact and whether they were successfully mitigated.

12.1 Risk Register

We identified and assessed the risks listed in [Table 63](#) at the beginning of the project.

ID	Risk	Impact	Likelihood	Mitigation Measures
RSK-01	Dividing work across three members can introduce task dependencies and handoff delays, which may block progress in critical sprints.	Medium	Medium	Define interfaces and dependencies during sprint planning, limit work in progress and escalate blockers in weekly stand-ups.
RSK-02	Limited hardware availability can delay development and testing if components fail, are damaged, or are temporarily unavailable.	High	Medium	Reserve hardware test windows early, keep spare parts and fallback test plans and prioritize simulation-based validation when hardware is unavailable.
RSK-03	Knowledge gaps in hardware, Rust, simulation and state estimation can create a steep learning curve and slow implementation.	High	Medium	Plan focused onboarding tasks, pair less experienced members with domain leads and schedule early technical spikes for high-risk topics.
RSK-04	Insufficient contribution tracking can make transparent attribution of work across team members difficult.	Medium	Medium	Maintain Jira ownership and time tracking consistently, link branches to tickets and review contribution logs at sprint boundaries.
RSK-05	Scope creep from late requirement additions can increase workload and jeopardize milestones.	High	Medium	Freeze sprint scope after planning, evaluate new requests with impact assessment and defer non-critical

ID	Risk	Impact	Likelihood	Mitigation Measures
				additions to later iterations.
RSK-06	Dependence on ARIS personnel and hardware without formal availability guarantees can delay integration and validation.	High	Medium	Coordinate with ARIS early, align test windows in advance and continue offline/ simulation validation whenever external support is unavailable.
RSK-07	Integrating Rust code with the Crazyflie firmware may require more effort than expected, delaying implementation and testing.	High	Medium	Create an early integration prototype, validate build and flashing workflow and keep a fallback implementation path for blocked interfaces.
RSK-08	The EKF implementation is on the project critical path; unforeseen technical challenges can significantly impact the overall timeline.	High	High	Prioritize EKF core milestones, test incrementally with synthetic data and reserve schedule buffers for model and tuning iterations.
RSK-09	If no suitable simulation frameworks are established, EKF validation quality and development speed may be reduced.	High	Medium	Timebox framework evaluation, define minimum simulation requirements and proceed with a documented fallback setup if needed.
RSK-10	A high number of stakeholders can increase communication overhead and cause misalignment of expectations and priorities.	Medium	Medium	Use regular coordination meetings, document decisions and action items and confirm priorities explicitly after each planning cycle.
RSK-11	ARIS validation opportunities occur late in the project window (single drop test and no flight test), increasing	High	High	Shift validation left with simulation and offline flight-data replay, define clear acceptance criteria

ID	Risk	Impact	Likelihood	Mitigation Measures
	the risk of late defect discovery.			early and prepare contingency actions for unresolved late findings.

Table 63: Risk register along with the mitigation measures.

12.2 Risk Review at Midpoint

The following section contains the revisited risks listed in [Table 63](#) and reassessed each entry against the current state of the thesis. [Table 64](#) contains the updated impact and likelihood ratings together with the current status of each risk. Updated mitigations are noted where the midpoint review changed the mitigation approach.

The risks are annotated with three statuses. *Active* means the risk is still ongoing and could impact the result. *Mitigated* means measures kept the risk from affecting the result. *Materialized* means the risk occurred.

ID	Risk (short)	Impact	Likelihood	Status
RSK-001	Task dependencies and handoff delays	Low	Low	Mitigated
	Comment: Sprint planning, regular stand-ups and the use of Teams proved effective. No extended blockers occurred. Updated mitigation: No change required.			
RSK-002	Limited hardware availability	Medium	Medium	Active
	Comment: Mitigations were largely effective. The long delivery time of the GNSS modules provided by ARIS delayed their integration, but this was classified as non-blocking so far. Updated mitigation: No change required.			
RSK-003	Knowledge gaps	Medium	Medium	Active
	Comment: Onboarding and pair work closed most gaps. The risk remains active for the adaptive EKF, where research work is still ongoing. Updated mitigation: No change required.			
RSK-004	Contribution tracking	Low	Low	Mitigated
	Comment: Jira ownership and branch-to-ticket linking are consistently maintained by all team members and time tracking is reliable. Updated mitigation: No change required.			
RSK-005	Scope creep	High	Low	Active
	Comment: The scope is largely fixed since the research aspect of the adaptive part was defined and all must-have requirements are currently			

ID	Risk (short)	Impact	Likelihood	Status
	on track. Some evaluation topics remain open and may add more effort than initially planned. Updated mitigation: No change required.			
RSK-006	Dependence on ARIS	Medium	Medium	Active
	Comment: Early coordination, primarily via the shared Slack channel, allowed us to fix scope and requirements within the first weeks and reduced the dependency. Only the integration phase remains. Updated mitigation: No change required.			
RSK-007	Integration on Crazyflie	High	Medium	Materialized
	Comment: Integration took longer than expected due to some Crazyflie limitations. The EKF Rust library was running on the Crazyflie on 09.05.2026, mitigating the immediate risk but the start of the integration on the ARIS hardware had to be shifted back by a few days. Updated mitigation: Reprioritize the second core application logic sprint to recover the lost days and start the ARIS hardware integration in parallel as soon as possible.			
RSK-008	EKF implementation challenges	High	Medium	Active
	Comment: The EKF is running and was validated against the simulation, which removed most of the risk. The remaining work on the EKF implementation concerns the adaptive part.			
RSK-009	Simulation framework	Low	Low	Active
	Comment: A suitable simulation framework was identified for both the rocket and the Crazyflie and initial simulations are running.			
RSK-010	Stakeholder communication	Medium	Low	Active
	Comment: Coordination meetings and meeting minutes proved effective, with open issues typically resolved within a few days.			
RSK-011	Late ARIS validation	High	High	Materialized
	Comment: No drop test has been conducted yet and the EKF is not yet fully running on the ARIS hardware because the Crazyflie integration consumed more time than planned. The delay is not critical but requires intensified effort in the second core application logic sprint to get back on track. Updated mitigation: Prioritize validation via simulation and maintain frequent exchange with ARIS about upcoming drop tests to identify possible validation opportunities early.			

Table 64: Updated risk ratings and status at the project midpoint, relative to the initial assessment in [Table 63](#).

12.3 Retrospective Risk Review at Project End

This section closes the risk management by reviewing each risk from [Table 63](#) against the final outcome of the project. For every entry, [Table 65](#) reports the closing status and a comment. The lessons drawn from the risks that materialized are summarized afterwards.

At the project end we use two closing statuses. *Mitigated* means measures kept the risk from affecting the result. *Materialized* means the risk occurred and is described together with how far we could absorb it.

ID	Risk (short)	Status
RSK-001	Task dependencies and handoff delays	Mitigated
	Comment: Sprint planning and weekly stand-ups kept the three subtasks aligned for the whole project. No extended blockers occurred.	
RSK-002	Limited hardware availability	Materialized
	Comment: The GNSS modules from ARIS arrived late, had to be returned once and the provided antennas where of limited quality. The impact stayed manageable because simulation-based validation did not depend on this hardware, but the modules required continuous attention and led to additional integration efforts.	
RSK-003	Knowledge gaps	Mitigated
	Comment: Onboarding tasks and pair work closed the initial gaps. The adaptive EKF remained the most demanding area but was handled within the team.	
RSK-004	Contribution tracking	Mitigated
	Comment: Jira ownership, branch-to-ticket linking and time tracking were maintained consistently by all members for the entire project.	
RSK-005	Scope creep	Mitigated
	Comment: The scope stayed fixed after the adaptive research part was defined. The remaining pressure came from the evaluation rather than new requirements, which we contained by deferring non-essential topics to future work.	
RSK-006	Dependence on ARIS	Mitigated
	Comment: Early coordination reduced the dependency during development. We started with bi-weekly meetings in the first weeks and then switched to more spontaneous communication via Slack. This worked better, as it was faster and did not require waiting for the next scheduled meeting.	
RSK-007	Integration on Crazyflie	Materialized

ID	Risk (short)	Status
	Comment: Integration took longer than planned due to Crazyflie limitations. The reprioritization worked, as the EKF ran on the Crazyflie a few days later and all drone evaluations were completed within the project window. The trade-off was that the sensor carrier integration had to be shifted back.	
RSK-008	EKF implementation challenges	Mitigated
	Comment: The EKF was implemented and validated against the simulation, including the adaptive part. The remaining limitation was time: The tuning and evaluation of the adaptive methods could only be explored to a limited depth.	
RSK-009	Simulation framework	Materialized
	Comment: RocketPy and RotorPy proved suitable for the rocket and the Crazyflie respectively and carried most of the evaluation. However, some limitations were detected later in the project. RocketPy switches from a 6-DOF to a 3-DOF model at drogue inflation and no longer tracks attitude, so the attitude could not be evaluated during descent. We absorbed this by freezing the attitude state for that phase, but it limits the descent-phase results.	
RSK-010	Stakeholder communication	Mitigated
	Comment: Bi-weekly coordination meetings kept expectations aligned. Other issues were typically resolved within a few days via the Teams channel.	
RSK-011	Late ARIS validation	Materialized
	Comment: No real launch took place during the project and a validation during a drop test was not possible because the flown ASTERIA hardware still ran an older firmware version while our EKF was implemented against the newer v3 version. The mitigation partially worked: Simulation data provided results and a first insight into the filter behavior, but real-flight verification on the rocket remains open.	
RSK-012	Simulation-to-real gap (unforeseen)	Materialized
	Comment: This risk was not part of the initial register. The simulated UWB model was defined too simple. It delivered all anchor measurements simultaneously and used a simplified noise model, which did not match the asynchronous arrival and varying noise of the real hardware. As a result, the parameters and adaptive methods tuned in simulation did not transfer directly to the real Crazyflie. The gap was identified during real-	

ID	Risk (short)	Status
	flight testing with optimized parameters based on the simulation.	

Table 65: Final risk ratings and status at project end, relative to the initial assessment in [Table 63](#) and the midpoint review in [Table 64](#).

12.3.1 Lessons Learned

The risks that materialized offer the most useful lessons for future work.

We consider shifting validation toward simulation early a good decision in hindsight. Because no real launch took place and the ARIS hardware integration took additional time, we relied on the simulation for the majority of the project duration. At the same time, we learned where simulation reaches its limits. Our oversimplified UWB model only surfaced once we tested the simulation-tuned parameters on the real Crazyflie. This taught us that a simulation gives valuable insight and enables fast testing, but it should be validated against real measurements early.

We underestimated the integration with real hardware. Bringing the Rust EKF onto the Crazyflie and ASTERIA firmware took us longer than planned and the GNSS modules required our attention throughout the project. For future work we would reserve more buffer for hardware integration and treat external hardware deliveries as a high risk from the start.

Finally, the adaptive part carried the most residual pressure to the end. A future project should budget more time for its tuning and evaluation, as this is where we see the most interesting topics that remain open.

13 Time Tracking Overview

This section provides an overview of the time tracking.

We spent a total of **1312 hours** on this thesis. [Figure 66](#) shows the burndown chart over the course of the project and [Figure 67](#) breaks down the total effort by tag.

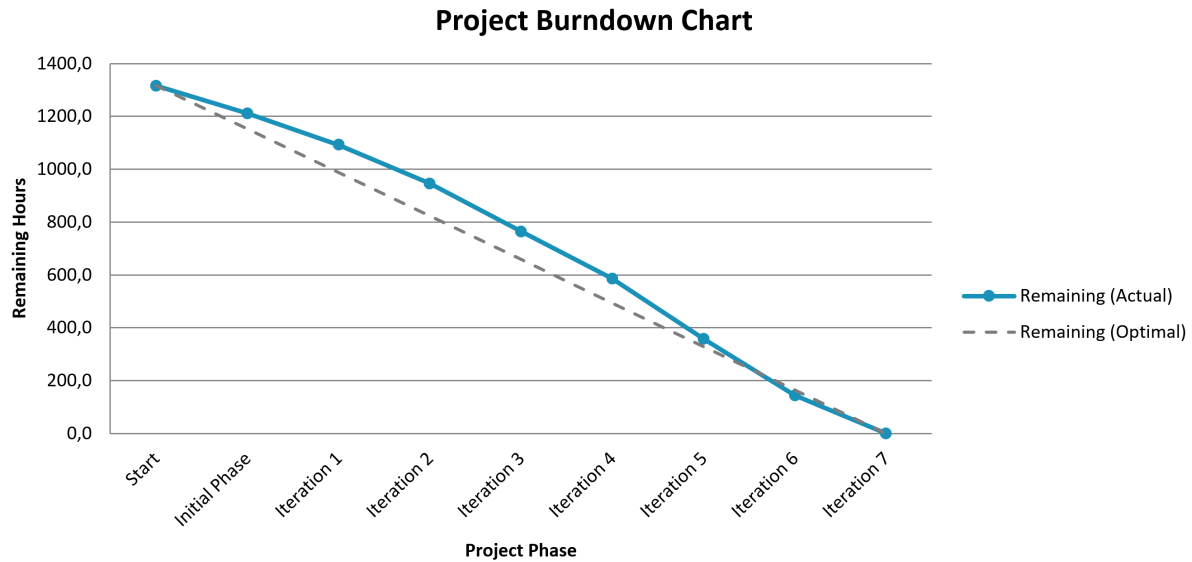


Figure 66: Burndown chart showing the remaining estimated effort over the duration of the thesis.

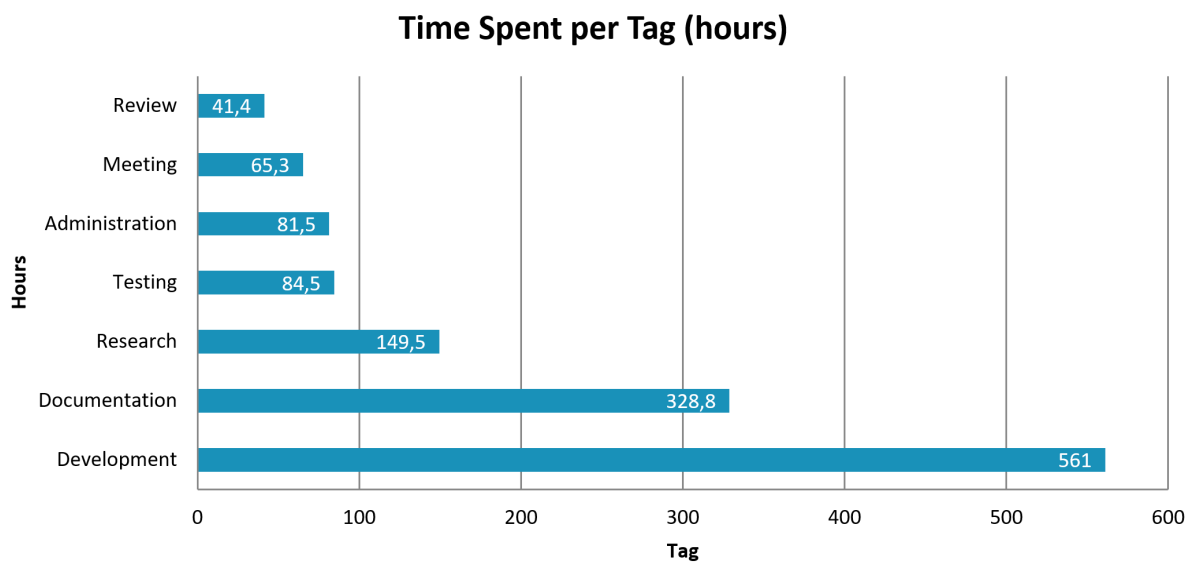


Figure 67: Distribution of the total tracked hours across the different activity tags.

[Figure 68](#) compares the hours contributed by each team member and [Figure 69](#) shows which Jira stories were recorded for the OST part of the thesis, for the ARIS part and which contributed to both.

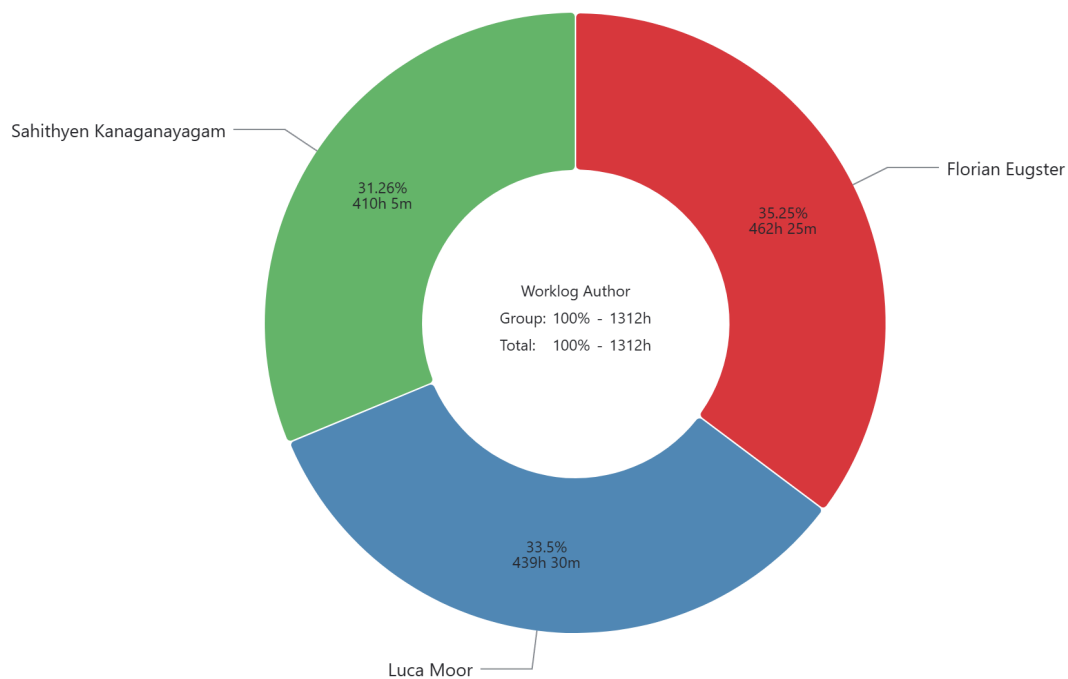


Figure 68: Overview of the tracked hours broken down by team member.

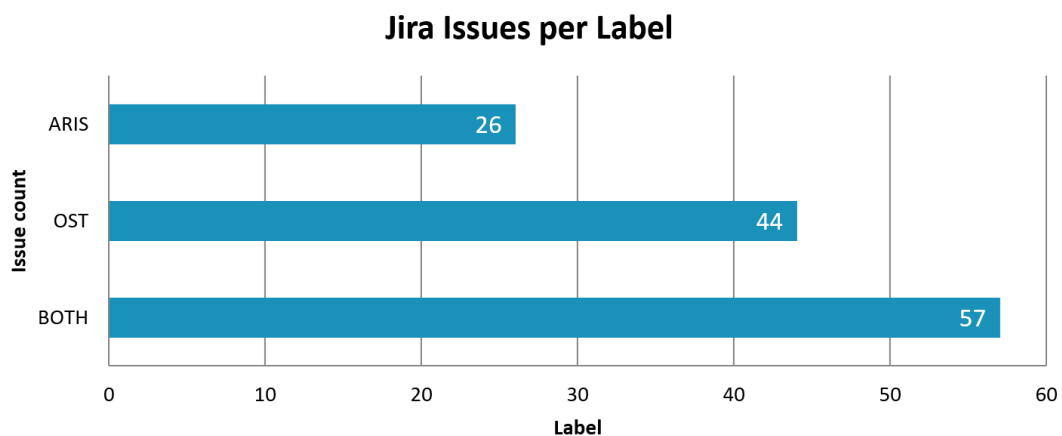


Figure 69: Number of Jira issues per label, distinguishing work attributed to the OST part, the ARIS part and issues contributed to both.

A reflection on the hours spent during this thesis, including why we exceeded our original time budget, can be found in [Section 14](#).

14 Retrospective

This chapter reflects on the project as a whole.

[Section 14.1.1](#) reviews how the project was organized and run, [Section 14.2](#) then discusses how we assess the outcome of our work.

14.1 Reflection on Organisational Management

This section reflects on the project planning, the teamwork and the external dependencies.

14.1.1 Project Planning

Planning was challenging at the start, as the direction of the thesis was not fully defined during the first four weeks. This became easier after some initial research and the finalization of the task description.

We divided the project into 7 two-week iterations, defined milestones and tracked our tasks in Jira. This kept us organized and gave us a clear overview throughout the project. We were able to stick to our planned iterations and defined milestones, which helped us stay on track. In hindsight, we would have reduced the time allocated for development and reserved more time for the integration on real hardware and the evaluation as these phases took more time than we had originally anticipated. In total, we invested 1312 hours, more than originally planned.

We met all of our must-have requirements. However, as the scope of the assignment turned out to be too large for the available time budget, we had to cut back on the should-have requirements and leave some additional topics unexplored.

During the project, four of the eleven identified risks materialized, along with one we had not anticipated (see [Table 65](#)). The realized risks could be handled by quickly finding solutions within the team or together with the advisor. They caused additional unexpected effort but stayed manageable.

14.1.2 Teamwork and Meetings

We worked in two-week sprints with weekly on-site Fridays, bi-weekly sprint planning and retrospectives, bi-weekly advisor meetings and initially bi-weekly ARIS meetings. After the first few weeks, once requirements and scope were fixed, we stopped holding regular ARIS meetings and coordinated via Slack when required, which turned out to be more practical.

The on-site Friday sessions were particularly effective. Working together in person made it easier to discuss open questions, resolve problems collaboratively and stay aligned across all areas of the project.

The division into technical leads for the three core areas (simulation, ARIS and Crazyflie) worked well. The clear ownership of different areas helped us stay organized and allowed us to work independently while still contributing to a shared outcome.

Internal communication ran via Microsoft Teams, coordination with ARIS was handled through a shared Slack channel, both of which proved efficient throughout the project.

14.1.3 External Dependencies

The main external dependency was the ARIS team, both as a source of requirements and as the provider of hardware.

The dependency on ARIS was identified as a significant risk from the beginning, since we depended on the availability of ARIS personnel and access to their hardware. In practice, early and regular coordination via the shared Slack channel allowed us to fix requirements early. The ARIS team was responsive throughout the semester, which kept this dependency from becoming a blocker for most of the project.

The one area in which the external dependency had a direct impact was hardware handling. The GNSS modules arrived later than expected and it took longer than anticipated to resolve hardware issues due to travel time and scheduling constraints.

14.2 Reflection on the Results

We are satisfied with what we built, but we see it as a solid baseline rather than a finished product. The central goal was met and the software we developed proved suitable for the intended use case. Still, the filter needs further tuning and several extensions before it can be considered complete.

What we really like about our implementation, is its ease of extendability. We are proud that it is possible to add adaptive noise measurement to sensors with just one line of code. We are also proud that the EKF is fully written in Rust and integrated into three different platforms, each of which uses bindings to support its own respective language.

The most important lesson concerns the difference between simulation and reality. Several concepts that worked in simulation, such as the parameter optimization, did not transfer cleanly to real hardware. Additionally, the evaluation turned out to be considerably harder and more time-consuming than anticipated. On real hardware, many effects occurred simultaneously, making it difficult to attribute behavior to a single cause.

Looking back, some of these difficulties trace back to our own decisions. We should have addressed the IMU bias before turning to the adaptive methods, as the unmodeled bias complicated the evaluation considerably. In the same way, additional integration topics only became apparent during evaluation and had to be addressed directly on the platforms.

Despite these limitations, we are confident in the foundation we developed, as the remaining gaps are extensions rather than redesigns. The bias states are already implemented, further sensors integrate through the path we demonstrated on the Crazyflie and the simulation is extensible and already supports more sophisticated tuning objectives.

15. Bibliography

- [1] Akademische Raumfahrt Initiative Schweiz (ARIS), “ASTERIA ARIS.” Accessed: Apr. 09, 2026. [Online]. Available: <https://aris-space.ch/our-projects/asteria/>
- [2] S. Folk, J. Paulos, and V. Kumar, “RotorPy: A Python-based Multirotor Simulator with Aerodynamics for Education and Research,” June 2023. doi: [10.48550/arXiv.2306.04485](https://doi.org/10.48550/arXiv.2306.04485).
- [3] STMicroelectronics, “LSM6DSO32 - iNEMO inertial module: always-on 3D accelerometer and 3D gyroscope,” DocID032891 Rev 1, Mar. 2020.
- [4] TE Connectivity, “MS5607-02BA03 Barometric Pressure Sensor, with stainless steel cap,” Mar. 2024.
- [5] STMicroelectronics, “LSM303AGR - Ultracompact high-performance eCompass module: ultralow-power 3-axis accelerometer and 3-axis magnetometer,” DocID027765 Rev 11, Aug. 2020.
- [6] Sensirion, “Datasheet – SHT4x 4th Gen. Relative Humidity and Temperature Sensor,” Mar. 2025.
- [7] u-blox, “ZED-F9P-05B High precision GNSS module Professional grade,” UBX-DOC-963802114-12824-R02, Oct. 2024.
- [8] Bosch Sensortec, “BMI088 6-axis Motion Tracking for High-Performance Applications,” BST-BMI088-DS000-19, Jan. 2024.
- [9] Bosch Sensortec, “BMP388 Digital Pressure Sensor,” BST-BMP388-DS001-07, Nov. 2020.
- [10] PixArt Imaging Inc., “PMW3901MB-TXQT: Optical Motion Tracking Chip,” Mar. 2017.
- [11] STMicroelectronics, “VL53L1X A new generation, long distance ranging Time-of-Flight sensor based on ST’s FlightSense technology,” DS12385, Aug. 2024.
- [12] qorvo, “DWM1000 IEEE 802.15.4-2011 UWB Transceiver Module,” 2016.
- [13] P. Groves, *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems, Second Edition*. Artech House, 2013.
- [14] A. Breitenmoser, “Adaptive Extended Kalman Filtering for Navigation State Estimation of Drones and Sounding Rockets,” 2026.
- [15] Akademische Raumfahrt Initiative Schweiz (ARIS), “HERMES ARIS.” Accessed: Apr. 09, 2026. [Online]. Available: <https://aris-space.ch/our-projects/hermes-2025/>
- [16] u-blox, “ZED-F9P High precision GNSS module Integration manual,” UBX-18010802-R16, Oct. 2024.
- [17] Bitcraze AB, “Crazyflie 2.1.” Accessed: Mar. 31, 2026. [Online]. Available: <https://www.bitcraze.io/products/old-products/crazyflie-2-1/>
- [18] Bitcraze AB, “Crazyflie 2.1 Research.” Accessed: Mar. 31, 2026. [Online]. Available: <https://www.bitcraze.io/portals/research/>
- [19] Bitcraze AB, “Crazyflie 2.1 datasheet,” Nov. 2021.
- [20] Bitcraze AB, “Crazyflie 2.1 Decks.” Accessed: Mar. 31, 2026. [Online]. Available: <https://store.bitcraze.io/collections/decks>

- [21] N. Trawny and S. I. Roumeliotis, “Indirect Kalman Filter for 3D Attitude Estimation: A Tutorial for Quaternion Algebra,” 2005–2, Mar. 2005.
- [22] Bitcraze, “Crazyflie Firmware, UWB Distance Measurement Model.” Accessed: June 04, 2026. [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/master/src/modules/src/kalman_core/mm_distance.c
- [23] R. Marchthaler and S. Dinger, *Kalman-Filter: Einführung in die Zustandsschätzung und ihre Anwendung für eingebettete Systeme*, 2nd ed. Springer Vieweg, 2024. doi: [10.1007/978-3-658-43216-4](https://doi.org/10.1007/978-3-658-43216-4).
- [24] F. L. Markley, “Attitude Error Representations for Kalman Filtering,” Mar. 2003. doi: [10.2514/2.5048](https://doi.org/10.2514/2.5048).
- [25] J. R. Carpenter and C. N. D'Souza, “Navigation Filter Best Practices,” NASA/TP-2018–219822, Apr. 2018.
- [26] Y. Bar-Shalom, X. R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation: Theory, Algorithms and Software*. Wiley-Interscience, 2002. doi: [10.1002/0471221279](https://doi.org/10.1002/0471221279).
- [27] Bitcraze AB, “Crazyflie Firmware, ToF Sensor Model.” Accessed: May 23, 2026. [Online]. Available: https://github.com/bitcraze/crazyflie-firmware/blob/master/src/modules/src/kalman_core/mm_tof.c
- [28] S. Sumithra and R. Vadivel, “An Optimal Innovation Based Adaptive Estimation Kalman Filter for Accurate Positioning in a Vehicular Ad-hoc Network,” *International Journal of Applied Mathematics and Computer Science*, vol. 31, no. 1, Mar. 2021, doi: [10.34768/amcs-2021-0004](https://doi.org/10.34768/amcs-2021-0004).
- [29] B. Suwandi, T. Kitasuka, and M. Aritsugi, “Vehicle Vibration Error Compensation on IMU-accelerometer Sensor Using Adaptive Filter and Low-pass Filter Approaches,” *Journal of Information Processing*, vol. 27, pp. 33–40, Jan. 2019, doi: [10.2197/ipsjip.27.33](https://doi.org/10.2197/ipsjip.27.33).
- [30] OpenRocket, “OpenRocket, Build better rockets.” Accessed: Mar. 26, 2026. [Online]. Available: <https://openrocket.info/>
- [31] RocketPy Team, “RocketPy Documentation.” Accessed: Mar. 26, 2026. [Online]. Available: <https://docs.rocketpy.org/en/latest/index.html>
- [32] RocketPy Team, “RocketPy Documentation, GenericMotor Class Usage.” Accessed: May 16, 2026. [Online]. Available: <https://docs.rocketpy.org/en/latest/user/motors/genericmotor.html>
- [33] H. Hersbach *et al.*, “The ERA5 global reanalysis,” June 2020. doi: [10.1002/qj.3803](https://doi.org/10.1002/qj.3803).
- [34] V. Suvorkin, M. Garcia-Fernandez, G. González-Casado, M. Li, and A. Rovira-Garcia, “Assessment of Noise of MEMS IMU Sensors of Different Grades for GNSS/IMU Navigation,” 2024. doi: [10.3390/s24061953](https://doi.org/10.3390/s24061953).
- [35] G. Ceotto, B. S. Carmo, R. N. Schmitt, L. Pezente, and G. F. Alves, “RocketPy: Six Degree-of-Freedom Rocket Trajectory Simulator,” Aug. 2022. doi: [10.1061/\(ASCE\)AS.1943-5525.0001331](https://doi.org/10.1061/(ASCE)AS.1943-5525.0001331).
- [36] RocketPy Team, “RocketPy Source Code, GnssReceiver Class.” Accessed: May 17, 2026. [Online]. Available: https://docs.rocketpy.org/en/latest/_modules/rocketpy/sensors/gnss_receiver.html#GnssReceiver

- [37] RocketPy Team, “RocketPy Documentation, Parachute Class.” Accessed: May 17, 2026. [Online]. Available: <https://docs.rocketpy.org/en/latest/reference/classes/Parachute.html>
- [38] Bitcraze AB, “Getting Started with Simulation.” Accessed: Apr. 06, 2026. [Online]. Available: <https://www.bitcraze.io/documentation/tutorials/getting-started-with-simulation/>
- [39] C. Llanes, Z. Kakish, K. Williams, and S. Coogan, “CrazySim: A Software-in-the-Loop Simulator for the Crazyflie Nano Quadrotor.” Accessed: Apr. 10, 2026. [Online]. Available: <https://github.com/gtfactslab/CrazySim>
- [40] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A Next-generation Hyperparameter Optimization Framework,” July 2019. doi: [10.1145/3292500.3330701](https://doi.org/10.1145/3292500.3330701).
- [41] Mozilla Project Contributors, “cbindgen: A tool for generating C bindings from Rust code.” Accessed: May 21, 2026. [Online]. Available: <https://github.com/mozilla/cbindgen>
- [42] PyO3 Project Contributors, “PyO3: Rust bindings for the Python interpreter.” Accessed: Apr. 06, 2026. [Online]. Available: <https://github.com/PyO3/pyo3>
- [43] maturin Project Contributors, “Maturin: Build and publish crates with pyo3, cffi and uniffi bindings as well as rust binaries as python packages.” Accessed: Apr. 06, 2026. [Online]. Available: <https://github.com/PyO3/maturin>
- [44] K. Schwaber and J. Sutherland, *The Scrum Guide*. 2020. Accessed: Mar. 05, 2026. [Online]. Available: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf>
- [45] K. Beck *et al.*, “Principles behind the Agile Manifesto.” Accessed: Mar. 05, 2026. [Online]. Available: <https://agilemanifesto.org/principles.html>
- [46] R. C. Martin, *Clean Code, A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

Appendix

I Assignment

Bachelor Thesis

Spring 2026

Adaptive Extended Kalman Filtering for Navigation State Estimation of Drones and Sounding Rockets

Florian Eugster & Luca Moor & Sahithyen Kanaganayagam

1. Introduction

ARIS (Akademische Raumfahrt Initiative Schweiz)¹ is a student-led aerospace association developing advanced sounding rockets for international competitions and research activities. One of its primary missions is participation in the European Rocketry Challenge (EUROC), where ARIS competes in the Liquid 30'000 ft category.

The rocket is a bi-liquid sounding rocket², powered by liquid oxygen (LOX) and ethanol. It is designed for high-performance flight profiles and reaches approximately:

- Apogee: ~30'000 ft (~9 km)
- Maximum velocity: ~353 m/s (~Mach 1.1)
- Maximum acceleration: ~40 m/s²

The rocket consists of multiple modular subsystems, including propulsion, avionics (AV), recovery, structures, and ground support. The avionics section comprises a modular sensor carrier board and a flight computer executing all mission-critical software functions. The sensor system includes redundant inertial sensors, magnetometers, GNSS receivers, and barometric sensors operating at significantly different sampling rates.

While attitude estimation is already handled by a dedicated AHRS pipeline³, the current navigation state estimation relies heavily on raw GNSS data and simplified signal processing. During high-dynamic flight phases and GNSS outages, this limits the quality and continuity of position and velocity estimates. For guided recovery, trajectory reconstruction, and post-flight analysis, a robust navigation filter in form of an Extended Kalman Filter (EKF) is required that fuses inertial, barometric, and GNSS measurements while respecting the physical motion of the rocket.

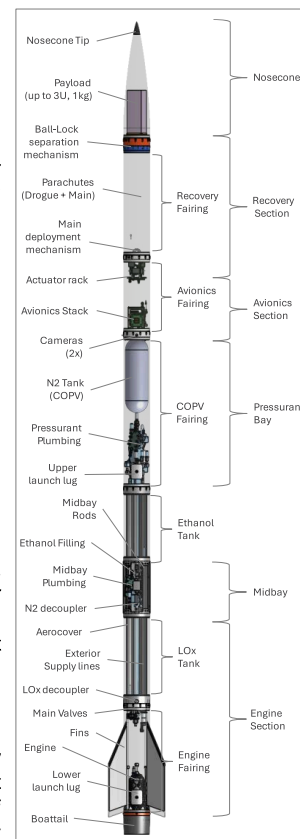


Figure 1: Rocket sections and components of ARIS bi-liquid sounding rocket HERMES I.

¹ [ARIS - Space & Rocket Projects](https://aris-space.ch)

² <https://aris-space.ch/our-projects/hermes-2025/>

³ [AHRS: Attitude and Heading Reference Systems](#)

Another flying system is represented by the Crazyflie 2.1⁴ miniature quadcopter (see Figure 2). Quadcopters also rely heavily on AHRS and EKF for real-time robust state estimation of the vehicle's motion [1] [2] but exhibit quite different underlying dynamics. This leads to the research question of whether a single EKF-based estimator architecture can be derived that adapts – potentially automatically – to aerial platforms across different dynamic regimes.



Figure 2: A Crazyflie 2.1 miniature quadcopter in simulation^{5,6} (left) and real flight (center) performs state estimation with an EKF model⁷ (right) for a fully instrumented Crazyflie platform.

To support system modeling and filter design, a simulation environment will be beneficial. Various simulators exist within the robotics community (e.g., Webots⁸, Gazebo⁹, CrazySim¹⁰) – while each solution offers value, it also comes with its own limitations: Webots may struggle with extremely high dynamics of rocket flight, Gazebo may additionally introduce undesirable system dependencies related to the ROS2¹¹ framework, CrazySim may be limited to Crazyflie-type quadcopter models. For these reasons, using a simulation based on first principles may be recommendable. Python-based simulations like RocketPy¹² or RotorPy¹³ are useful in this regard (also note that Python bindings for Crazyflie quadcopter¹⁴ are available).

Besides allowing safe testing of state estimation algorithms before real deployment, simulation provides the additional advantage of experimenting with a wider range of vehicle platforms, verifying the generalizability of the EKF design, and eventually testing the cross-platform adaptivity claim. Concretely, the EKF could be adapted to a third platform solely existent in simulation. For example, a simulated parachute or balloon could be an interesting addition, since it exhibits completely different dynamics.

2. Task Definition

The objective of this project is to design, implement, and validate an embedded EKF for navigation state estimation of an ARIS sounding rocket and a Crazyflie miniature quadcopter.

⁴ <https://www.bitcraze.io/products/old-products/crazyflie-2-1>

⁵ <https://www.bitcraze.io/tag/simulation>

⁶ <https://www.cyberbotics.com/doc/guide/crazyflie?version=R2023a>

⁷ <https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control>

⁸ <https://www.cyberbotics.com/#webots>

⁹ [Gazebo](#)

¹⁰ <https://github.com/gtfactslab/CrazySim?tab=readme-ov-file>

¹¹ [ROS](#)

¹² <https://docs.rocketpy.org/en/latest/#>

¹³ <https://github.com/spencerfolk/rotorpy>

¹⁴ <https://www.bitcraze.io/2024/04/python-bindings-for-the-crazyflie-firmware/>

The EKF instance for the rocket shall estimate position and velocity by fusing data from GNSS, inertial acceleration, and barometric altitude, while using externally provided attitude information for frame transformations. The filter shall be designed to operate in real time on the sensor carrier board and remain stable under high accelerations, sensor noise, and temporary measurement loss.

The solution shall be written in Rust¹⁵ and integrate into the existing sensor carrier firmware and software architecture without requiring modifications to low-level drivers or attitude estimation components.

2.1 Scope of Work

The thesis will be split into four parts:

Work Package 1: Modeling the EKF for the sounding rocket

Objective:

Develop a physically consistent EKF formulation suitable for rocket navigation under EUROCC flight conditions.

Tasks:

- Familiarize yourself with the underlying theory and concepts of EKF [1] [2]¹⁶ ¹⁷
- Define EKF state vector (position, velocity, biases as applicable)
- Define coordinate frames and transformations
- Develop a rocket-appropriate process model
- Define measurement models for:
 - GNSS position
 - Barometric altitude
- Noise modeling and tuning strategy
- Offline validation using simulated and recorded flight data
- Analysis of filter behavior during:
 - Boost phase
 - Coast
 - Descent
 - GNSS outages

Key Learnings:

- Practical EKF design for high-dynamic systems
- Trade-offs between model fidelity and numerical stability
- Handling asynchronous and intermittent measurements

Work Package 2: System integration and test with the rocket's avionic section

Objective:

Integrate the EKF into the existing avionics software stack and validate its real-time behavior, using Rust as programming language.

¹⁵ [Rust Programming Language](#)

¹⁶ [Python Toolboxes - Peter Corke](#) and [petercorke/robotics-toolbox-python: Robotics Toolbox for Python](#)

¹⁷ [PROBABILISTIC-ROBOTICS.ORG](#) and [09d-ekf-localization.pdf](#)

Tasks:

- Familiarize yourself with the existing ARIS avionics section hardware/software solution and code projects
- Embedded implementation of the EKF
- Interface definition to sensor fusion and logging layers, initialization and reset logic
- Handling of sensor availability and dropouts
- Real-time performance evaluation
- Subsystem-level testing using log replay and bench tests
- Definition of outputs for telemetry and post-flight analysis

Key Learnings:

- Embedded software design for estimation algorithms
- Real-time constraints and numerical robustness
- Verification of safety-critical software behavior

Work Package 3: Adaption of EKF formulation for Crazyflie and other systems

Objective:

Adapt the EKF filter across platforms and make it usable on Crazyflie miniature quadcopters, using Python and/or Rust or C/C++ as programming language.

Tasks:

- Familiarize yourself with state-of-the-art adaptation strategies for the EKF and its application across different dynamic regimes, as well as their easy-to-use configuration, through focused literature research
- Look into noise tuning, adaptive covariance formulation, and sensor abstraction
- *Optional:* Configuration framework, which automatically adapts measurement/process noise, using innovation statistics

Key Learnings:

- Design of platform-agnostic reusable state estimation framework (or reuse existing frameworks)^{18 19 20}
- Adaptive noise modelling and estimator tuning
- Cross-platform sensor abstraction and integration

Work Package 4: System simulation and verification on potential applications

Objective:

Set up a suitable simulation environment to generate realistic flight scenarios and sensor data (where needed) for different platforms (e.g., sounding rocket, Crazyflie quadcopter, and optionally a parachute or balloon) to test, validate, and compare the performance of the EKF under varying dynamics and sensor conditions. Recommendation: use a Python-based simulation.

¹⁸ [ethz-asl/ethz-asl-msf](https://ethz-asl.ethz.ch/msf/): MSF - Modular framework for multi sensor fusion based on an EKF

¹⁹ https://docs.px4.io/main/en/advanced_config/tuning_the_ecl_ekf and <https://github.com/PX4>

²⁰ https://github.com/cra-ros-pkg/robot_localization

Tasks:

- Develop a simulation for representative vehicle dynamics (rocket ascent/descent, quadcopter flight, optional balloon drift in wind).
- Add realistic sensor models (e.g., IMU, GNSS, barometer) including noise, bias, different sampling rates, and potential measurement dropouts.
- Generate synthetic datasets for different flight scenarios and operating conditions
- Integrate the EKF implementation with the simulation framework for offline testing and evaluation
- *Optional:* Establish a hardware-in-the-loop (HIL) setup in which the EKF runs on embedded hardware, while the remainder of the system is simulated.
- Evaluate filter performance using suitable metrics (e.g., estimation error, innovation statistics, robustness during sensor outages).
- Compare EKF behavior across platforms and analyze how system dynamics influence estimator performance.

Key Learnings:

- Design and implementation of simulation for state estimation
- Creation of synthetic sensor data to test navigation algorithms
- Validation and performance evaluation of estimation algorithms
- Understand the influence of system dynamics and sensor characteristics
- Gain insight into how different vehicle dynamics and sensor configurations affect navigation state estimation

2.2 Deliverables

The main deliverables of this work are summarized below:

- EKF concept and mathematical formulation for sounding rocket and Crazyflie miniature quadcopter
- Offline validation results and performance analysis
- Simulation of rocket, drone and optionally further platforms, such as a parachute or balloon – all showing different dynamic ranges and varying inputs from potentially different sensor suites.
- *Optional:* Exploration of EKF adaptivity, including optimality and ease-of-use in configuration for the different selected platforms (rocket, drone, balloon, ...)
- Embedded EKF implementation integrated into the avionics software and the miniature drone
- Subsystem-level validation results for rocket and drone systems
- Final technical documentation, including design rationale, test and measurement results, as well as identified limitations and improvement recommendations.

3. Additional Directions

3.1 Confidentiality

The intellectual property rights to this work belong to the client, respectively the OST.

Communication between the students and any third party must be agreed with the steering committee and/or with the industrial partner.

At the end of your work, back up your data on the drive and delete your files on the computers.

3.2 Resources

Own programs and software packages may only be used where:

- Legal licenses are available
- The data can also be read and processed at school and by ARIS

When designing with the school CAD, the relevant design guidelines must be observed.

The resources, infrastructure and data repositories of ARIS may be accessed under the present conditions.

3.3 Contacts & organization

ARIS Contact		
Adrian Senn (PM)	OST Rapperswil 8.222	adrian.senn@ost.ch
Cyrill Bucher (SE Electronics)	ARIS Dübendorf	cyrill.bucher@aris-space.ch
Carl Ceccucci (TL Avionics)	ARIS Dübendorf	carl.ceccucci@aris-space.ch

4. Project Schedule

Plan at least a total of 360 working hours (12 ECTS x 30h / ECTS) per student.

At the beginning of the thesis, a project plan and specifications must be prepared and discussed with the advisors within the first two weeks. In the course of the thesis, regular status review meetings are to be scheduled to check important work steps. The status review meetings shall be documented by written meeting minutes.

If the work is carried out in a team of two or three, activities and subtasks shall be divided in a reasonable and fair manner within the team. This also applies to the final report, which shall clearly indicate who is responsible for which part of the report and the work.

We recommend that you document the thesis from the beginning and keep a lab journal. The lab journal can help you record your thoughts and allows you to use them for later reference. The lab journal can be viewed by the advisors during meetings. However, the lab journal is not part of the final report.

5. Documentation

The thesis must be documented in a final report, which must contain all considerations, clarifications, calculations and investigations in detail in text and figures. The report should be written in a legible and clearly structured way. An external person with appropriate expertise shall be able to follow the report.

The report shall include the following sections:

- Table of content
- Abstract of 0.5 - 1 page length

II Directory of Tools Used

The [Table 66](#) provides an overview of which tools were used during the execution of this bachelor thesis.

Category	Tool
Literature Research	Swisscovery, Google Scholar, Arxiv
Coding	Visual Studio Code, Jupyter Notebooks, PyCharm, Copilot, Claude, Claude Code
Version Control	Git, GitLab, GitHub
DevOps and CI/CD	GitLab CI/CD, Docker
Text creation, optimization and checking	Typst, Visual Studio Code, Copilot, Claude, Claude Code, Deepl Write
Translation	Google Translate, Deepl
Diagramming	draw.io, PlantUML, Claude, Claude Code
Project Management	Jira, Timetracker Plugin, Excel
Communication	Microsoft Teams, Microsoft Outlook, Slack

Table 66: Directory of Tools Used in this project

III Hardware and Versions

This appendix lists all hardware components and software versions used throughout the project.

III.A Crazyflie

The following tables list the hardware components ([Table 67](#)) and software versions ([Table 68](#)) used during the Crazyflie experiments.

III.A.a Hardware

Component	Details
Drone	bitcraze Crazyflie 2.1
Positioning Deck	bitcraze Loco Positioning Deck
Optical Flow Deck	bitcraze Flow Deck v2
UWB Anchors	bitcraze Loco Positioning Node (4× in sessions 1–2, 8× from flight tests onward)
Debugger	STLink V3 (used in Lab Journal stationary session 3)

Table 67: Crazyflie hardware components

III.A.b Software

Component	Version / Details
cfclient	2025.12.1
cflib	0.1.31
CF Firmware	Git-commit: 87be2d0d
Rust EKF Library	Git-commit: 8e0e67d8 (repo: ba-implementation)
Loco Positioning Node Firmware	2022.09

Table 68: Crazyflie software versions

III.B ARIS ASTERIA

The following tables list the hardware components ([Table 69](#)) and software versions ([Table 70](#)) used during the ARIS ASTERIA experiments.

III.B.a Hardware

Component	Details
Sensor carrier board	HERMES Sensor Carrier Board Version 0.9a (Serial number 1)
GNSS module	ZED-F9P-05B

Table 69: ARIS ASTERIA hardware components

III.B.b Software

Component	Version / Details
ASTERIA Firmware	Git-commit: 288e790 (repo: asteria-firmware)
Rust EKF Library	Git-commit: d741e70 (repo: ba-implementation)

Table 70: ARIS ASTERIA software versions

IV Lab Journals

IV.A Drone Test Session #1 - Loco Positioning Test

Date

24.03.2026

Hardware

Crazyflie 2.1, Loco Positioning Deck, 4× UWB Anchors

Software / Firmware

CF Firmware 2025.12.1, cfclient 2025.12.1, cflib 0.1.31

Objectives

- Test the standard Crazyflie firmware and collect flight data into CSV files

Setup

Four UWB anchors (A0–A3) were placed at fixed positions in the test area, each at least 15 cm away from walls and the floor. The positions were then measured and the anchors were configured with these values:

Anchor	x (m)	y (m)	z (m)
A0	2.32	−1.43	2.08
A1	2.32	1.39	1.32
A2	−2.32	1.42	0.44
A3	−2.32	−1.41	0.21



Figure 70: UWB anchor A0



Figure 71: UWB anchor A1

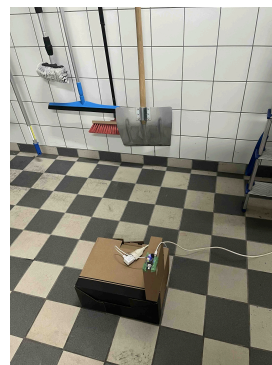


Figure 72: UWB anchor A2



Figure 73: UWB anchor A3

The Crazyflie 2.1 was equipped with the Loco Positioning Deck.

Procedure

1. Carried the Crazyflie by hand through the room and observed the position estimate in cfclient

2. Wrote a Python script using cflib commanding: Take off, move 50 cm forward, 50 cm left, 50 cm back, 50 cm right, land
3. Executed the script three times and collected CSV logs
4. Visualized the data in a Jupyter Notebook (placeholder repository)

Results

The position estimate in cflib did not match the physical position during the manual walkthrough. The drone reported itself as airborne while resting on the floor.

The automated flight script failed for the same reason: The drone believed it was already in the air and did not take off, only sliding across the floor due to motor spin-up. Three CSV log files were nonetheless successfully captured and visualized in Jupyter Notebooks.

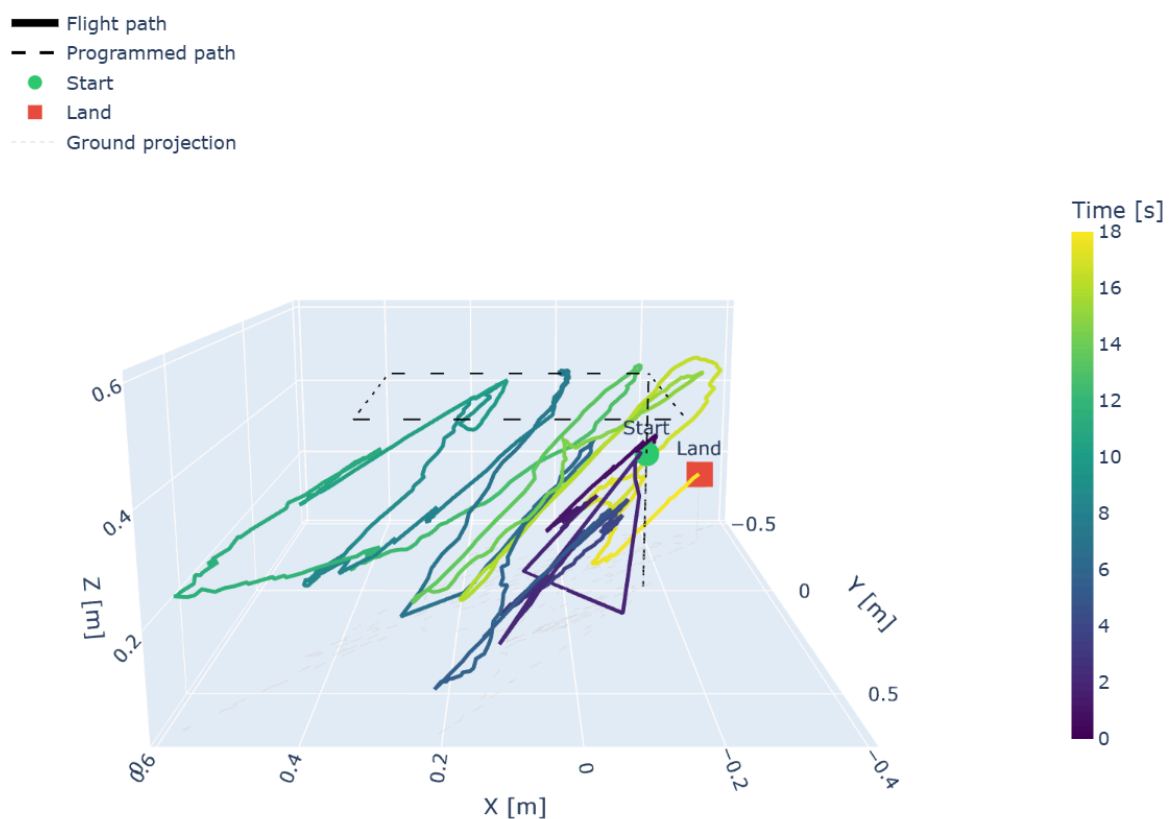


Figure 74: Visualized flight path from one of the CSV logs

Issues & Observations

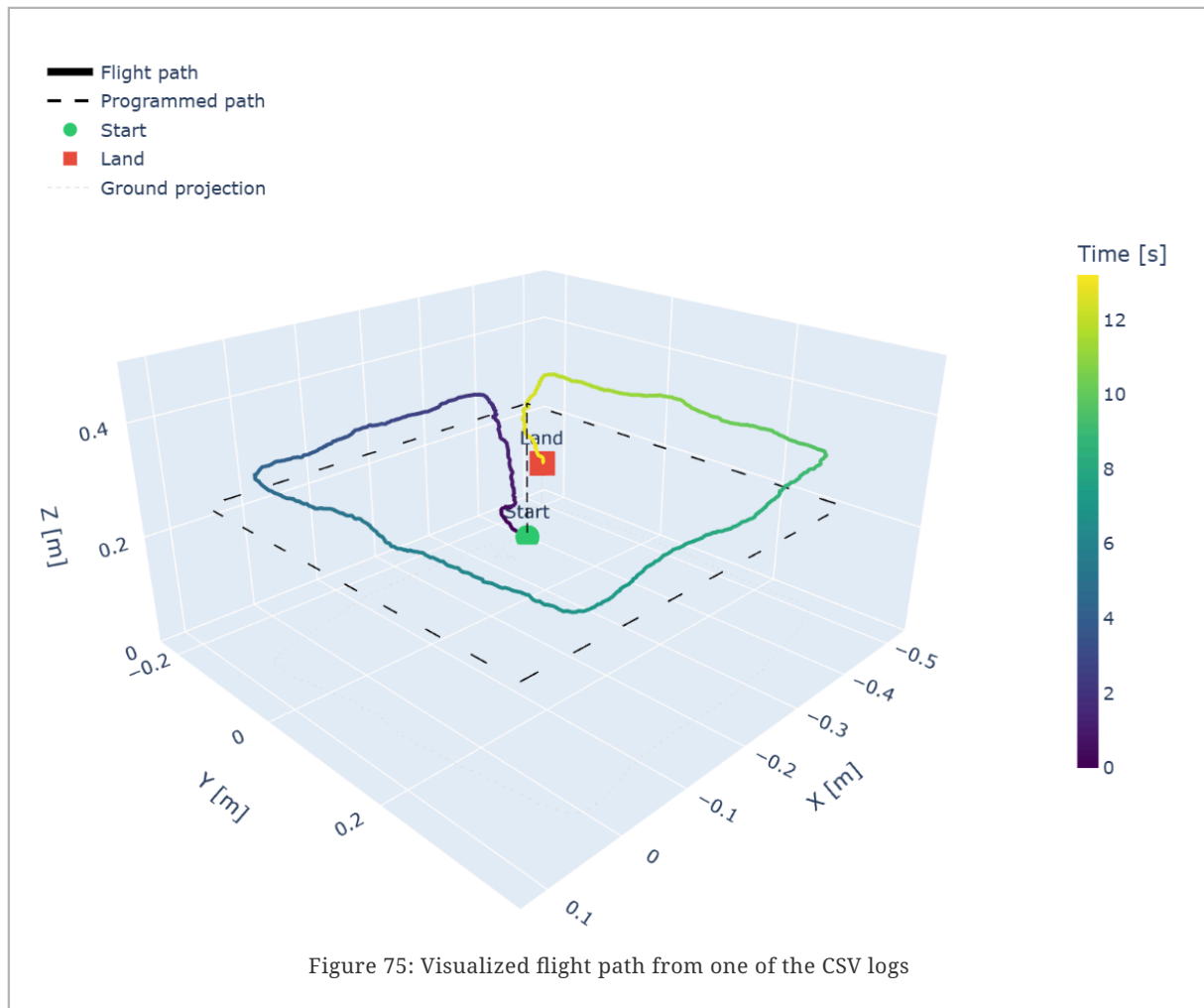
- Position estimate incorrect: Drone reported itself as airborne while on the floor
- Automated flight script failed: Drone did not lift off in any of the three runs
- Propellers detached repeatedly during motor spin-up

Next Steps

- Ask advisor for tips on the positioning issue
- Retry with Flow Deck v2 combined with Loco Positioning Deck

IV.B Drone Test Session #2 - Loco Positioning with Flow Deck v2

Date 02.04.2026
Hardware Crazyflie 2.1, Loco Positioning Deck, Flow Deck v2, 4× UWB Anchors
Software / Firmware CF Firmware 2025.12.1, cfclient 2025.12.1, cflib 0.1.31
<u>Objectives</u> • Retry the flight test with the Flow Deck v2 added to improve positioning stability • Collect CSV flight data across multiple different routes
<u>Setup</u> Same anchor placement as Session 1 IV.A (A0–A3, identical positions). The Crazyflie 2.1 was equipped with both the Loco Positioning Deck and the Flow Deck v2.
<u>Procedure</u> 1. Set up anchors in the same configuration as Session 1 IV.A 2. Executed multiple flight runs with different routes using Python scripts via cflib and collected CSV logs for each
<u>Results</u> With the Flow Deck v2 added, the positioning worked significantly better and the drone flew as expected. Multiple routes were successfully flown and CSV data was collected for offline analysis. However, the positioning was still not perfect and some inaccuracies were observed.



Issues & Observations

Propellers continued to detach during motor spin-up, as observed in [Session 1 IV.A](#). The issue was resolved by applying a small amount of hot glue to the propellers. Hot glue was chosen specifically because it can be removed easily without damaging the propellers or motors.

IV.C Drone Test with Rust Library stationary #1

Date

01.05.2026

Hardware

Crazyflie 2.1

Software / Firmware

CF Firmware Git-commit: a5c38377 (modified for our EKF), Rust library Git-commit: a420c1e2 (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Run our EKF on the Crazyflie 2.1 for the first time in a stationary configuration without flying.

- Validate that our EKF can operate as a shadow EKF running in parallel to the existing Crazyflie EKF without interfering with normal operation.

Setup

- The Crazyflie was placed on a flat surface and kept stationary throughout all tests (no flight attempted).
- Our EKF was used as a shadow estimator running in parallel to the Crazyflie EKF, receiving the same sensor data but not influencing control.

Procedure

1. Flashed the modified firmware including our EKF integration onto the Crazyflie.
2. Powered on the drone and attempted to establish a radio connection via cfclient to observe our EKF state estimates.
3. Observed repeated firmware crashes shortly after boot or immediately upon radio connection. The firmware entered error mode and required a restart.
4. Suspected a stack overflow or RAM shortage as the root cause, since the Rust library requires more stack memory than available in the CCM (Core Coupled Memory) region.
5. Attempted mitigations:
 - Increased the task stack size for our EKF thread.
 - Moved allocations from CCM to the main RAM region.
6. Isolated our EKF by disabling the original Crazyflie EKF and running only our EKF to determine whether the crash was caused by concurrent operation or by our EKF itself.
7. Logged our EKF state estimates during the isolated run.

Results

- The firmware crashed consistently and entered error mode when the shadow EKF configuration was active. The crash occurred reliably as soon as a radio connection was established.
- Increasing the stack size and relocating memory from CCM to main SRAM reduced the crash frequency but did not fully resolve the instability.
- Running our EKF alone (without the Crazyflie EKF) was more stable but still exhibited occasional crashes.
- Our EKF state estimates produced clearly unexpected values even in the stationary case, most notably visible in the gyroscope data. The root cause of these incorrect estimates was not identified within this session.

Next Steps

- Refactor the Rust library to reduce its stack footprint and systematically investigate the source of the erroneous output values of our EKF.

IV.D Drone Test with Rust Library stationary #2

Date 05.05.2026
Hardware Crazyflie 2.1
Software / Firmware CF Firmware Git-commit: a5c38377 (modified for our EKF), Rust library Git-commit: a420c1e2 (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31
<u>Objectives</u> Investigate the root cause of the unexpected state estimates of our EKF observed in session 1 Section IV.C
<u>Setup</u> - Crazyflie placed on a flat surface, stationary, no flight attempted. - Our EKF ran as a shadow estimator alongside the active Crazyflie EKF, receiving the same sensor data without influencing control.
<u>Procedure</u> - Reviewed our EKF predict step and inspected how the time delta is passed to the filter. - Identified that the time delta was passed in milliseconds instead of seconds. - Fixed the unit conversion and reran our EKF with the corrected input. - Observed our EKF state estimates after the fix.
<u>Results</u> - The root cause of the unexpected values was found: The predict step received the time delta in milliseconds instead of seconds, causing the filter to scale dynamics incorrectly. - After correcting the unit, our EKF state estimates were no longer obviously wrong and appeared plausible for a stationary drone.
<u>Issues & Observations</u> The firmware stability issues from session 1 Section IV.C were not yet resolved within this session.

IV.E Drone Test with Rust Library stationary #3

Date 08.05.2026
Hardware Crazyflie 2.1, 1× UWB Anchor, STLink V3 Debugger
Software / Firmware CF Firmware Git-commit: 31ec635b, Rust library Git-commit: ed6efa48 (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Test the updated firmware with UWB anchor support added to provide position measurements to our EKF.
- Investigate the watchdog reset occurring when radio and UWB are used simultaneously.

Setup

- Crazyflie placed on a flat surface, stationary, no flight attempted.
- Our EKF ran as a shadow estimator alongside the active Crazyflie EKF, receiving the same sensor data without influencing control.
- UWB anchor added to provide ranging data to our EKF alongside the existing sensor inputs.
- For some tests: STLink V3 debugger connected to the Crazyflie for deeper inspection.

Procedure

1. Enabled both the radio connection and UWB ranging simultaneously and observed the firmware behavior.
2. Observed that the firmware consistently hit the watchdog and reset when both radio and UWB were active at the same time.
3. Added timing measurements around individual method calls to identify which part of the code was taking too long.
4. Connected the STLink V3 debugger to narrow down the location in the code where the timeout occurred.

Results

- The firmware reliably triggered the watchdog when radio and UWB were used at the same time.
- Timing measurements revealed that certain method calls took 4 – 5 ms, far exceeding the expected budget.
- The location in the code where the slowdown occurred was narrowed down, but the root cause of the excessive execution time could not be determined within this session.

IV.F Drone Test with Rust Library stationary #4

Date

09.05.2026

Hardware

Crazyflie 2.1, 1× UWB Anchor

Software / Firmware

CF Firmware Git-commit: 31ec635b, Rust library Git-commit: 0df01779 (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Test the updated Rust library, which was previously refactored to be generic over the floating-point type and configured to use single-precision float for the Crazyflie to match the STM32F405 hardware FPU and confirm stable operation with radio and UWB active simultaneously.

Setup

- Crazyflie placed on a flat surface, stationary, no flight attempted.
- Our EKF ran as a shadow estimator alongside the active Crazyflie EKF, receiving the same sensor data without influencing control.
- 1× UWB anchor active for ranging data.
- The Rust library had been updated beforehand: The library originally used double throughout, but the STM32F405 only has a hardware FPU for single-precision float. The library was therefore refactored to be generic over the floating-point type and configured to use float for the Crazyflie target.

Procedure

1. Flashed the updated firmware with the single-precision float library onto the Crazyflie.
2. Enabled radio and UWB simultaneously and observed firmware behavior and timing.

Results

- The previously slow method call dropped from 4 – 5 ms to a fraction of that after switching to single-precision float.
- The firmware ran stably with radio and UWB active at the same time, no watchdog resets occurred.

Next Steps

- Test our EKF during an actual flight.

IV.G Flight Test with our EKF #1

Date

11.05.2026

Hardware

Crazyflie 2.1, Flow Deck v2, Loco Positioning Deck, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 28d20ef0, Rust library Git-commit: e7d642ca (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Test our EKF during an actual flight for the first time.
- Compare our EKF shadow path (UWB only) against the Crazyflie EKF (UWB + Flow Deck v2).

Setup

Eight UWB anchors (A0–A7) were placed at fixed positions in the test area. The Crazyflie EKF used both UWB and the Flow Deck v2 for position estimation. Our EKF ran as a shadow estimator using UWB only.

Anchor	x (m)	y (m)	z (m)
A0	2.32	1.39	2.08
A1	2.32	−1.43	2.08
A2	2.34	1.39	0.23
A3	2.32	−1.43	0.20
A4	−2.32	1.39	2.25
A5	−2.32	−1.43	1.87
A6	−2.32	1.39	0.23
A7	−2.32	−1.43	0.24



Figure 76: UWB anchor A0–A3

Procedure

1. Wrote a Python script using MotionCommander commanding the drone to: Take off to 30 cm, fly 1 m forward, 1 m left, 1 m back, 1 m right (square), climb 1 m, repeat the square, then land.
2. Executed the script and logged both the Crazyflie EKF path and our EKF shadow path.

Results

- Our EKF shadow path deviated significantly from the actual flight path, as visible in the plot.
- The Crazyflie EKF also deviated and did not fly the intended route correctly.
- The root cause was identified as a user error: The MotionCommander was used, which accepts velocity and duration as inputs rather than absolute positions. This caused the drone to not follow the intended trajectory.

Crazyflie · 3D Flight Path

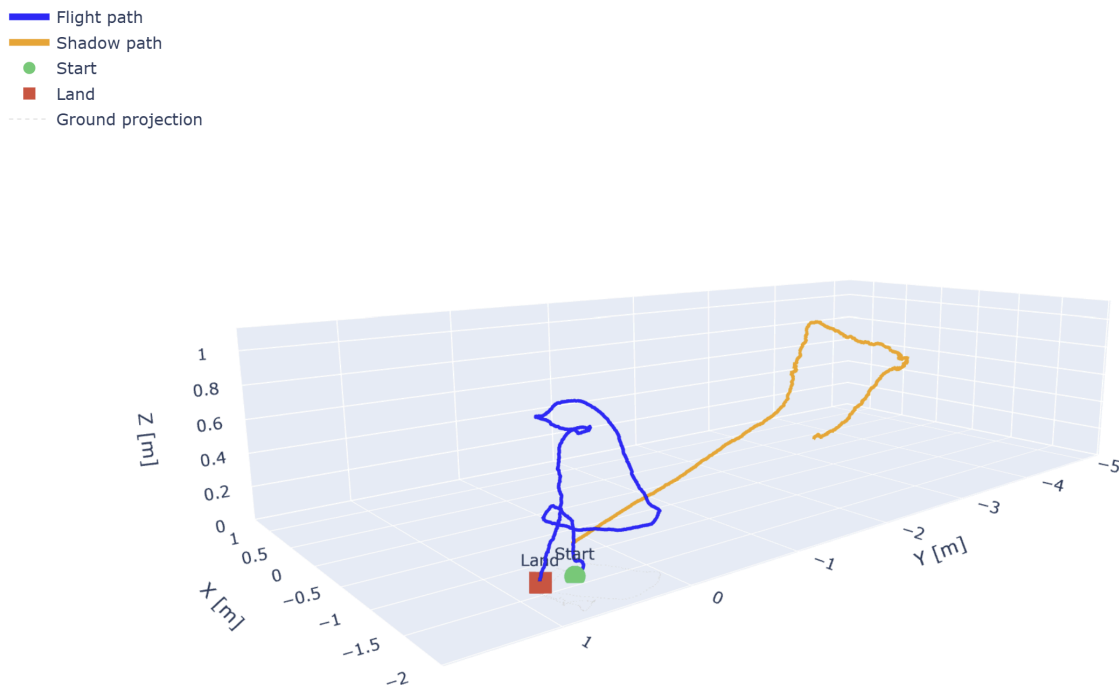


Figure 77: Visualized flight path showing both the Crazyflie EKF and our EKF estimates

Issues & Observations

- Wrong commander used: MotionCommander interprets commands as velocity $\times$ time, not as target positions. The intended square route was therefore not flown correctly.

Next Steps

- Retry the flight test using PositionHlCommander, which accepts absolute target positions.

IV.H Flight Test with our EKF #2

Date

12.05.2026

Hardware

Crazyflie 2.1, Flow Deck v2, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 28d20ef0 (modified: coordinate frame fix), Rust library Git-commit: e7d642ca (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Retry the flight test using PositionHlCommander so the drone follows the intended trajectory.
- Observe our EKF shadow path during a correct flight and assess its accuracy.

Setup

Same as the previous session: 8 UWB anchors at identical positions, Crazyflie EKF using UWB + Flow Deck v2, our EKF running as shadow with UWB only.

Procedure

1. Replaced MotionCommander with PositionHlCommander in the flight script and executed the square trajectory.
2. Observed that the drone now followed the intended path correctly.
3. Compared our EKF shadow path against the Crazyflie EKF path and observed a clear divergence.
4. Identified during the test that the Crazyflie firmware uses an East-North-Up (ENU) frame, while our EKF uses a NED (North-East-Down) body frame and the conversion between the two was incorrect.
5. Made several adjustments to the coordinate frame conversion during the test and reran the flight to observe the effect.

Results

- With PositionHlCommander the drone successfully followed the intended square trajectory.
- Our EKF shadow path diverged clearly from the actual flight path.
- During the test the coordinate frame mismatch between ENU (Crazyflie) and NED (our EKF) was identified as the root cause.
- After adjustments during the session the shadow path improved significantly, but something still did not match completely.

Crazyflie · 3D Flight Path

- Flight path
- Shadow path
- Start
- Land
- Ground projection

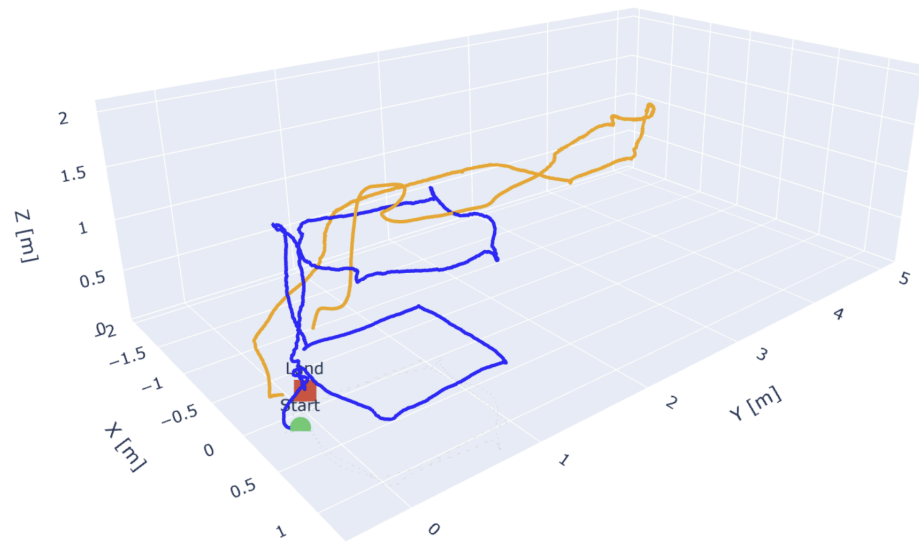


Figure 78: Visualized flight path of the first flight

Crazyflie · 3D Flight Path

- Flight path
- Shadow path
- Start
- Land
- Shadow Start
- Shadow Land
- Ground projection

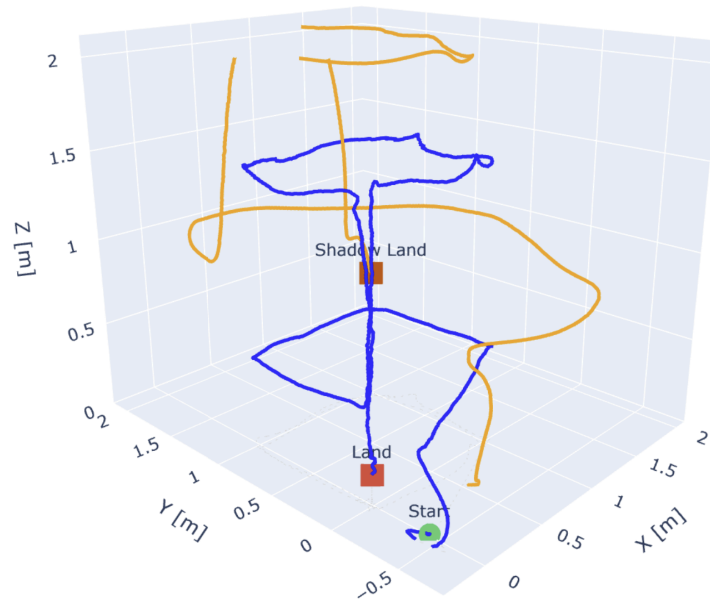


Figure 79: Visualized flight path of the last flight

Issues & Observations

- The coordinate frame conversion between ENU (Crazyflie) and NED (our EKF) was incorrect, identified during this session.
- Even after in-session adjustments our EKF output was not fully correct, the exact remaining discrepancy was not resolved.

Next Steps

- Analyse the coordinate frame conversion in detail to identify and fix the remaining discrepancy.

IV.I Flight Test with our EKF #3

Date

13.-14.05.2026

Hardware

Crazyflie 2.1, Flow Deck v2, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 28d20ef0 (modified: coordinate frame fix), Rust library Git-commit: e7d642ca (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Verify the improved results of our EKF after the firmware was updated done before this session.
- Compare our EKF shadow path (UWB only) against the Crazyflie EKF on equal footing by disabling the Flow Deck v2 on the Crazyflie EKF.

Setup

Same as the previous sessions: 8 UWB anchors at identical positions, our EKF running as shadow with UWB only. The firmware was updated before this session to improve the coordinate frame handling, which had already led to better results.

Procedure

1. Noted that the Crazyflie EKF uses both UWB and the Flow Deck v2, while our EKF uses only UWB, making a direct comparison unfair.
2. Disabled the Flow Deck v2 input on the Crazyflie EKF so both estimators relied on UWB only.
3. Ran the flight again and compared state and shadow paths for x, y and z.

Results

The x and y estimates of our EKF matched the Crazyflie EKF closely throughout the flight. The z axis showed slightly more deviation, particularly at higher altitudes, but the overall tracking was good.

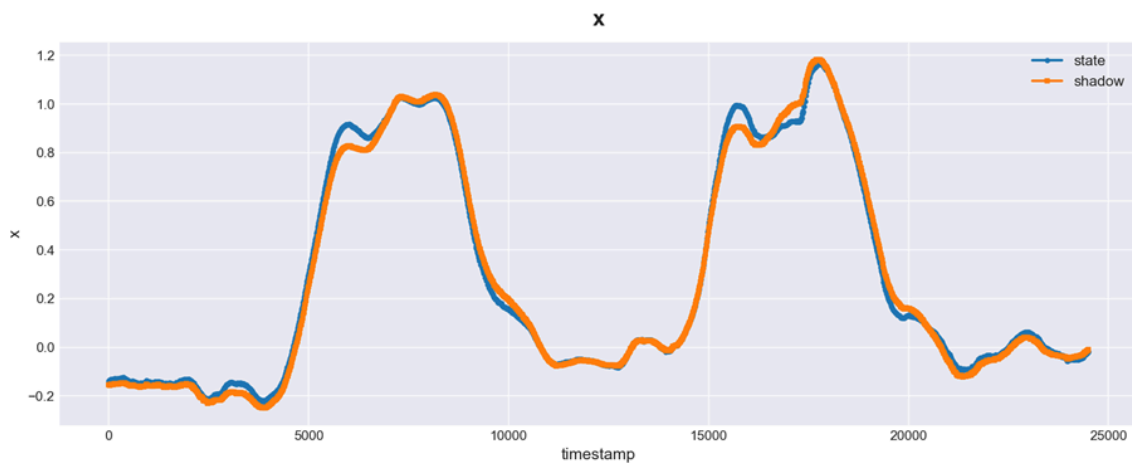


Figure 80: x position: state vs. shadow

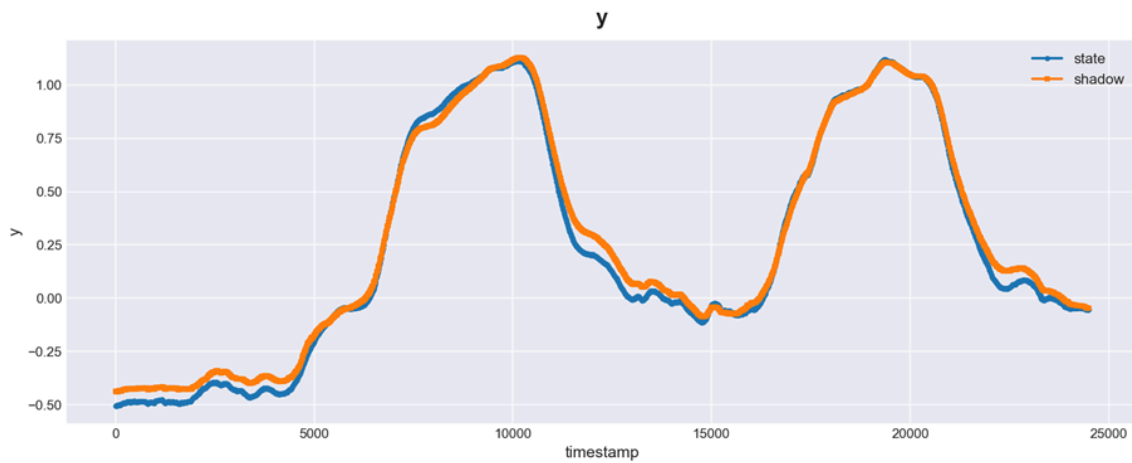


Figure 81: y position: state vs. shadow

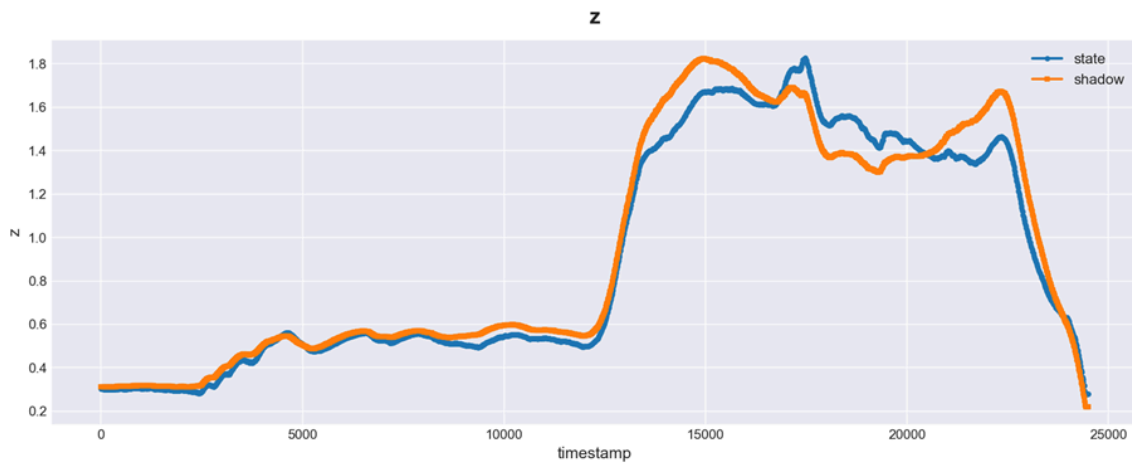


Figure 82: z position: state vs. shadow

Issues & Observations

- The z axis still shows some deviation between state and shadow, especially at higher altitudes.

IV.J Flight Test with our EKF #4 - ToF Sensor

Date

18.05.2026

Hardware

Crazyflie 2.1, Flow Deck v2, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 28d20ef0 (modified: coordinate frame fix), Rust library Git-commit: 4f254ae1 (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Test the newly integrated ToF sensor as an additional measurement source to improve z estimation in our EKF.

Setup

Same as the previous sessions: 8 UWB anchors at identical positions, our EKF running as shadow. The ToF sensor measures the distance to the ground and was added to improve the z axis accuracy.

Procedure

1. Ran the flight test and observed our EKF shadow path with ToF measurements active.
2. Compared the resulting flight path and attitude angles against the previous session.

Results

The ToF sensor did not improve the z estimation — the results were worse than without it.

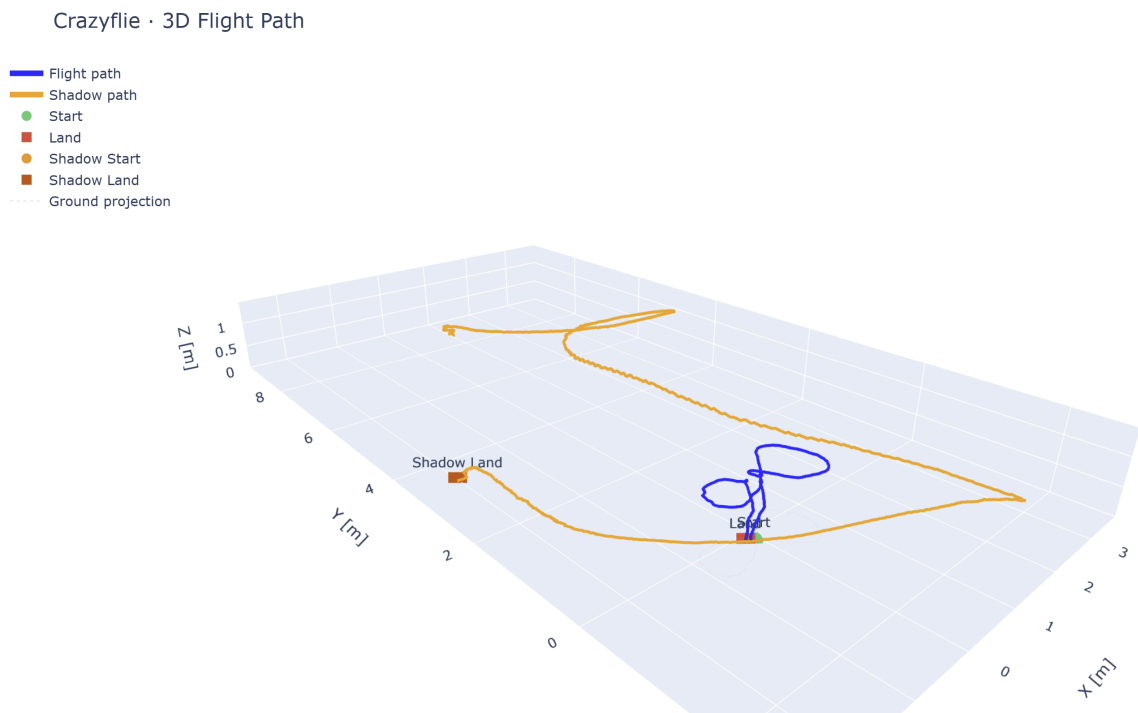


Figure 83: Flight path with ToF sensor active

The pitch and roll angles also showed discrepancies, which contributed to the problem.

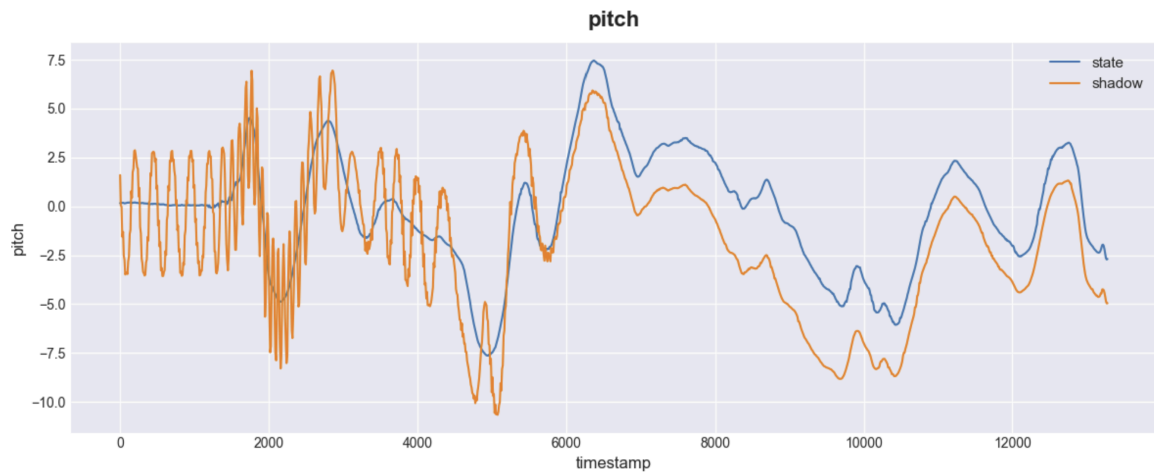


Figure 84: Pitch: state vs. shadow

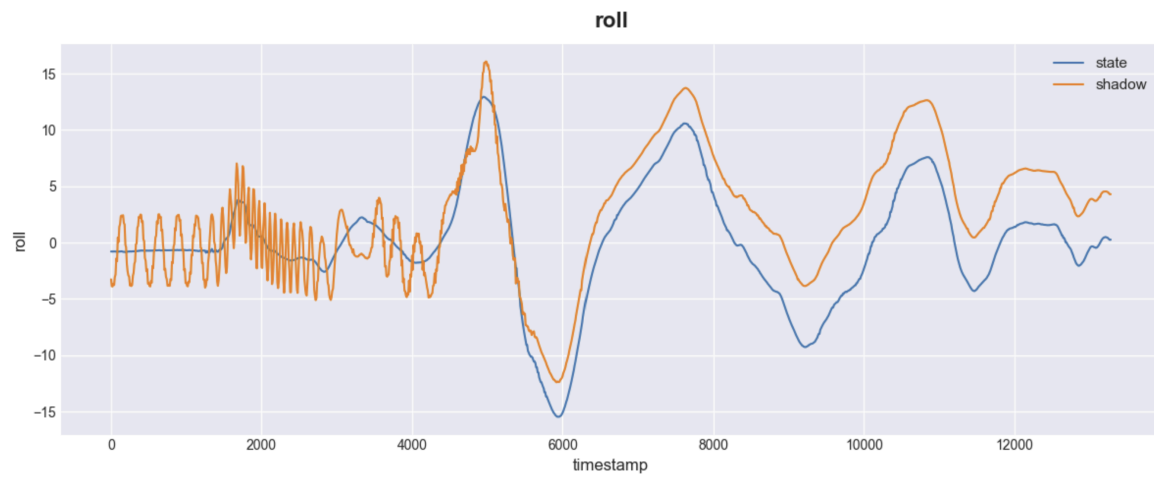


Figure 85: Roll: state vs. shadow

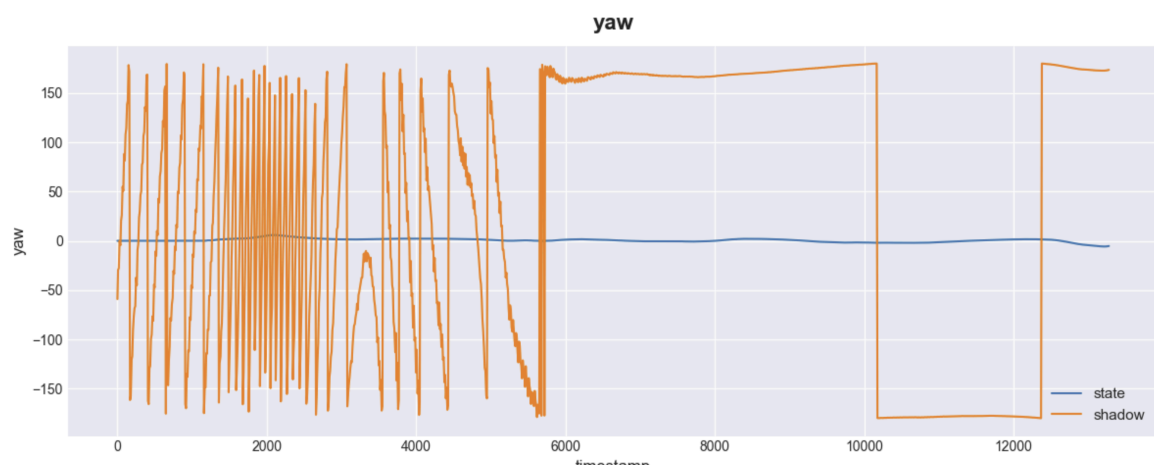


Figure 86: Yaw: state vs. shadow

Issues & Observations

- The ToF sensor has a 15° cone angle. When the drone is tilted, the measurement point is no longer directly below the drone, so the raw distance reading must be corrected using the current pitch and roll angles. The angle estimates from our EKF are not yet accurate enough for this correction, which caused the ToF measurements to degrade the z estimate rather than improve it.

IV.K Flight Test with our EKF #5 - Attitude Fix and ToF Re-Test

Date

25.05.2026

Hardware

Crazyflie 2.1, Flow Deck v2, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 108434f, Rust library Git-commit: 76ec538f (repo: ba-implementation), cfclient 2025.12.1, cflib 0.1.31

Objectives

- Fix the attitude (pitch/roll/yaw) estimation problem identified in Flight Test 4 by tuning the gyroscope and accelerometer noise standard deviations.
- Correct the frame transition handling in our EKF.
- Re-test the ToF sensor integration once attitude estimates are reliable.

Setup

Same as the previous sessions: 8 UWB anchors at identical positions, our EKF running as shadow with ToF measurements active. Prior to this session, the gyroscope and accelerometer noise standard deviations were adjusted and the frame transition in our EKF was corrected. Both changes were implemented and committed before this flight test.

Procedure

1. Ran the flight test with the updated noise parameters and corrected frame transition.
2. Compared pitch, roll and yaw estimates of our EKF shadow against the Crazyflie EKF.
3. Observed the z estimation with ToF measurements active.

Results

The attitude angles now match the Crazyflie EKF closely. Pitch, roll and yaw all track the Crazyflie EKF accurately throughout the flight. As a consequence, the ToF cone-angle correction (which depends on accurate pitch/roll estimates) also works correctly and the z estimation improved compared to Flight Test 4

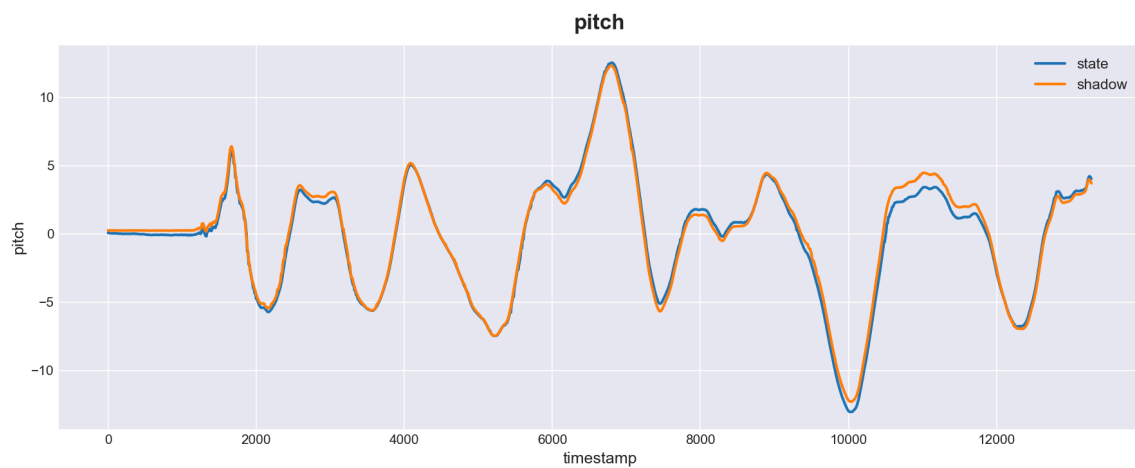


Figure 87: Pitch: state vs. shadow

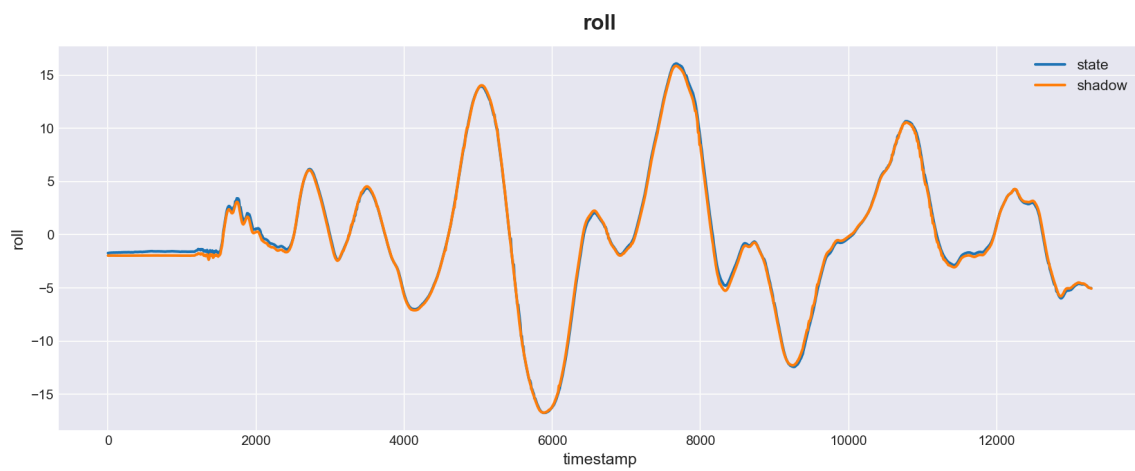


Figure 88: Roll: state vs. shadow

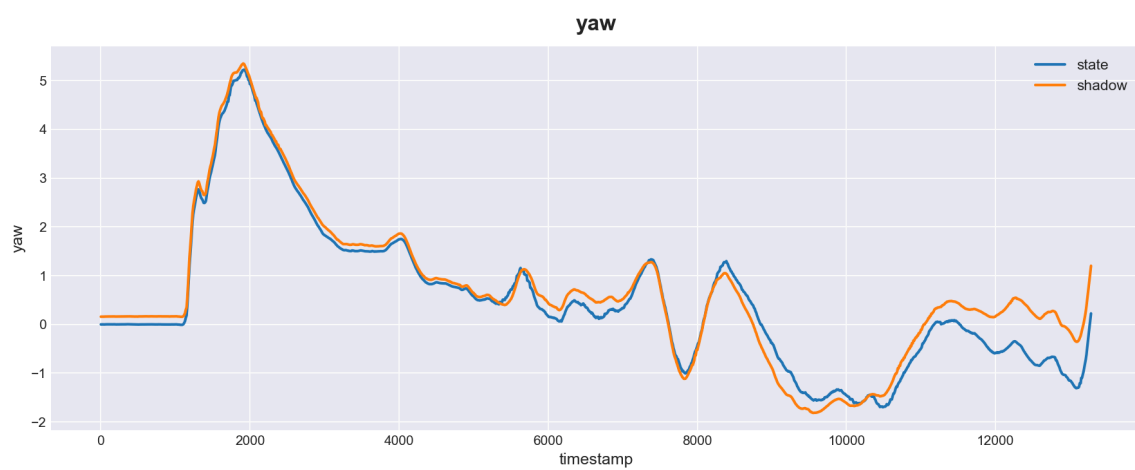
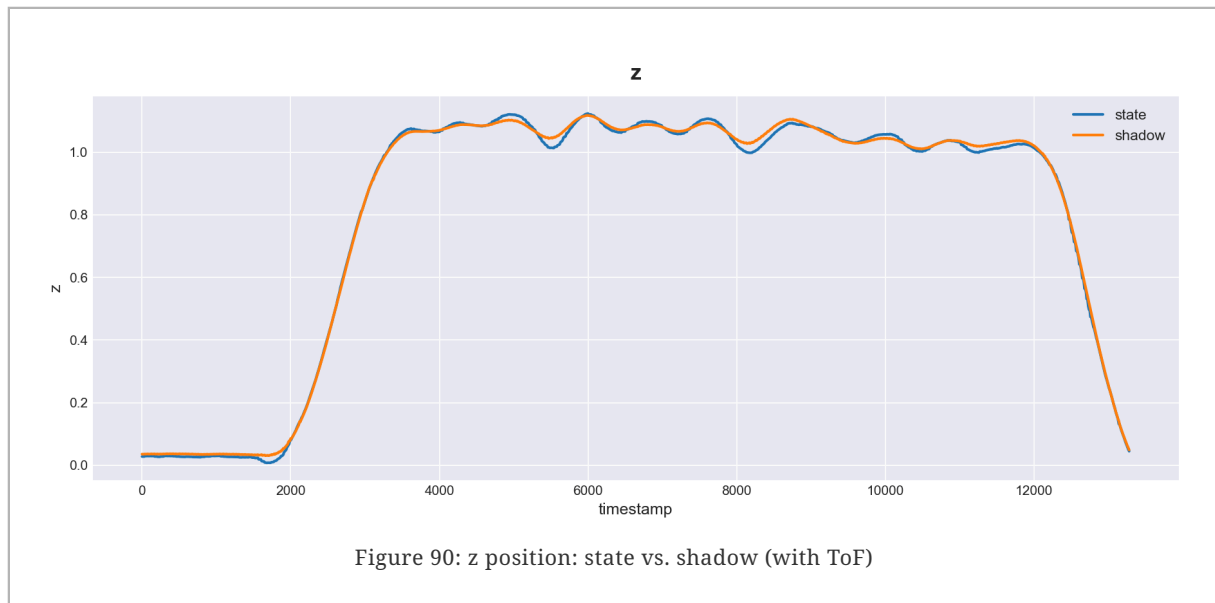


Figure 89: Yaw: state vs. shadow



IV.L Evaluation Flight Session

Date

01.-10.06.2026

Hardware

Crazyflie 2.1, Flow Deck v2, 8× UWB Anchors

Software / Firmware

CF Firmware Git-commit: 621c1d0a (modified: NIS logging, committed later as 87be2d0d),
Rust library Git-commit: 8e0e67d8 (repo: ba-implementation), cfclient 2025.12.1, cflib
0.1.31

Objectives

- Evaluate our EKF on real indoor flights in both shadow and active configurations across four different trajectories and ten filter configurations each.
- Tune filter parameters on real hardware using NIS consistency analysis.

Setup

Same as the previous flight sessions: 8 UWB anchors at identical positions.

Results

Full results and analysis are documented in [Section 7.5.1](#).

V Architectural Decision Records

This chapter contains the architectural decision records (ADRs) we made during this thesis.

V.A EKF Rust Library Architecture

PyO3 for Rust EKF integration in Python	06.04.2026
<u>Context</u>	
The EKF library is implemented in Rust and the simulation/evaluation is written in Python. The two languages must be bridged so that the EKF library can be called directly from the Python evaluation container.	
<u>Decision</u>	
We expose the Rust EKF library to Python via PyO3 bindings. The bindings live alongside the Rust crate and are built into a Python module that is imported directly by the runners. Runners are the only place in the Python codebase where the PyO3-bound API appears.	
<u>Alternatives</u>	
• Direct invocation of a Rust binary: The Python pipeline launches the EKF as a separate process. Avoids the build-time coupling between Python and Rust but complicates debugging and logging. • Reimplementation of the EKF in Python for evaluation: The evaluation pipeline uses a Python-native EKF implementation that mirrors the Rust one. Removes the language boundary entirely but creates two implementations of the same filter that must be kept in sync. Additionally, this would defeat the purpose of testing the actual Rust implementation.	
<u>Consequences</u>	
Positive	
• The same Rust EKF code is executed in evaluation as on the embedded target. • The PyO3 bindings are relatively easy to integrate and well documented.	
Negative	
• Building the evaluation environment requires a Rust toolchain in addition to Python which complicates the setup. • The PyO3-bound API is coupled to a specific Rust crate version. Updating the EKF library may require regenerating and reinstalling the bindings.	

ADR 1: Decision to expose the Rust EKF library to the Python evaluation pipeline via PyO3 bindings.

Context

A multi-dimensional measurement (e.g. GNSS providing three position scalars) can be applied as a single batched update with a multi-row Jacobian $\mathbf{H} \in \mathbb{R}^{m \times 9}$ and an $m \times m$ innovation covariance that must be inverted, or decomposed into a sequence of scalar updates each with a 1×9 Jacobian row $\mathbf{h}$ and a scalar innovation variance s requiring only scalar division.

Decision

All measurement types, regardless of dimension, are decomposed into scalar updates. The `Measurement` trait requires a compile-time constant `DIM` and a `scalar(i, ekf)` method returning one `ScalarUpdate` at a time. The `Ekf::correct` loop calls `scalar` for $i = 0, \dots, \text{DIM} - 1$ and applies each update independently.

Alternatives

- **Batched multi-row update:** The full $\mathbf{H}$ and $\mathbf{z}$ are assembled at once and the standard $\mathbf{S} = \mathbf{H}\mathbf{P}\mathbf{H}^\top + \mathbf{R}$ inversion is applied. Theoretically more accurate when measurement components are correlated, but requires an $m \times m$ matrix inversion and complicates the Joseph form.

Consequences**Positive**

- Innovation variance reduces to a scalar, requiring only division instead of a matrix inversion.
- The Joseph form covariance update applies per scalar without modification.
- Individual scalars can self-gate by returning `None` without affecting the remaining dimensions.
- Adaptive noise estimation operates independently per scalar dimension.

Negative

- Successive scalar updates within the same measurement are not statistically independent: Each update modifies $\mathbf{P}$ before the next one reads it, introducing a sequential correlation not present in a single batched update.

ADR 2: Decision to decompose all measurements into sequential scalar updates instead of a batched multi-row update.

Context

The filter's predictor and measurement extension points can be implemented with trait objects (`dyn Trait` backed by `Box<dyn ...>` on the heap) or with generic type parameters resolved monomorphically at compile time. The `no_std` requirement prohibits a heap allocator, which rules out `Box<dyn Trait>` in the embedded build.

Decision

All extension points use generic type parameters. `Ekf::predict<P: Predictor>` and `Ekf::correct<M: Measurement>` accept any type satisfying the respective trait bound. No `dyn Trait` or heap allocation appears anywhere in the crate.

Alternatives

- **Dynamic dispatch with trait objects:** Predictors and measurements are stored as `Box<dyn Predictor>` or `Box<dyn Measurement>`, enabling runtime configuration. Requires a heap allocator and is therefore incompatible with the `no_std` embedded target.
- **Enum-based dispatch:** A central enum variant per predictor or measurement type is matched at runtime. Compatible with `no_std` and supports runtime selection, but requires modifying a central enum for every new type and couples all variants into a single compilation unit.

Consequences

Positive

- No vtable overhead or heap allocation; fully compatible with `no_std`.
- The compiler can inline and optimize each concrete predictor and measurement type fully.
- Adding a new predictor or measurement type requires no changes to existing code.

Negative

- Each distinct combination of predictor and measurement types produces a separate monomorphized binary artifact, which can increase binary size.
- The active filter configuration is fixed at compile time and cannot be changed at runtime.

ADR 3: Decision to use trait-based static dispatch (monomorphization) instead of dynamic dispatch or enum dispatch.

Context

Adding online noise estimation to a concrete measurement type `M` requires a newtype struct wrapping `AdaptiveMeasurement<M, DIM, S, V>`, constructor methods for both the `Window` and EMA strategies, named accessor methods to the inner measurement and a delegating `Measurement` impl. For three measurement types (Gnss, UwbAnchor, Tof) this is structurally identical boilerplate repeated three times.

Decision

We provide the `impl_adaptive_measurement!` declarative macro. A single invocation per measurement type generates the full wrapper: The newtype struct, `Window` and EMA constructors (including concrete-float variants), named accessors and the `Measurement` delegation. The three adaptive types (`AdaptiveGnss`, `AdaptiveUwbAnchor`, `AdaptiveTof`) are each produced by one invocation.

Alternatives

- **Manual wrapper types per measurement:** Each adaptive wrapper is written out in full. Fully visible to IDEs and Rust-Analyzer and produces readable error messages, but every new measurement type duplicates the same structural code.
- **Blanket impl via extension trait:** A blanket `impl<M: Measurement> Adaptive for M` adds adaptive behavior automatically. Simpler, but prevents per-type constructor customization and can cause trait impl conflicts.

Consequences**Positive**

- Adding adaptive noise to a new measurement type requires a single macro call.
- The generated API surface is uniform across all adaptive wrapper types.

Negative

- Macro internals are partially opaque to Rust-Analyzer and IDE navigation.
- Errors inside a macro expansion can produce difficult-to-read diagnostics.
- The macro requires careful maintenance when the `AdaptiveMeasurement` interface changes.

ADR 4: Decision to generate adaptive measurement wrapper types with a declarative macro instead of writing manual per-type wrappers.

Context

The Crazyflie firmware consuming the EKF requires a plain C function interface. Rust's ownership model requires the `EkfState` to be owned by a well-defined location. Two common approaches are an opaque pointer handle (the C caller holds the allocation) or a library-owned global state.

Decision

The C API owns a single `static UnsafeCell<MaybeUninit<EkfState>>` initialized by `rkf_init`. All `rkf_*` functions operate on this global without requiring the caller to manage a handle. The design assumes exactly one filter instance per firmware image, which holds for the Crazyflie firmware.

Alternatives

- **Opaque pointer handle:** The caller receives an opaque `*mut RkfHandle` from a creation function and passes it to every subsequent call. More flexible for multiple instances, but requires the firmware to manage a heap-allocated object, which contradicts the zero-allocation requirement.

Consequences

Positive

- Zero allocation on the caller side; the firmware does not need a heap or an externally-managed buffer for the filter state.
- The C API surface is minimal: One init function and one function per operation.
- Matches the single-instance convention common in embedded firmware.

Negative

- Not reentrant; only one filter instance is possible per build.
- Calling any `rkf_*` function before `rkf_init` is undefined behavior, which cannot be statically detected at the C level.

ADR 5: Decision to use a library-owned global state with an `rkf_init` initializer instead of an opaque pointer handle.

Context

The library must produce three distinct build artifacts: A Rust library (`rlib`) for Rust consumers, a static archive (`staticlib`) with a C API for the Crazyflie firmware and a PyO3 native extension for Python. Each artifact requires different dependencies (standard library vs. `libm`, `pyo3`, `cbindgen`) and a different crate type. Maintaining separate crates per target would create three codebases that must be kept in sync.

Decision

We use Cargo feature flags to select the active build surface within a single crate. The `std` feature (the default) enables the standard library; `no_std` switches to `libm`-backed math. The `c_api` and `c_api_sliding_window` features activate the C bindings module. The `python` feature activates the PyO3 module. Features compose: `--no-default-features --features no_std,c_api --crate-type=staticlib` produces the firmware-ready static archive; `--features python` produces the extension module.

Alternatives

- **Separate crate per target:** An `ekf-core` crate holds the filter; `ekf-c` and `ekf-python` wrappers produce the respective artifacts. Separates concerns cleanly, but creates three versioning and release surfaces that must remain aligned.

Consequences

Positive

- One codebase, one version; changes to the filter propagate to all three surfaces simultaneously.
- The feature matrix is explicit and documented in `Cargo.toml`.

Negative

- CI must validate multiple feature flag combinations to cover all build surfaces.
- The feature set and test matrix grow if new targets are added.

ADR 6: Decision to use Cargo feature flags for multi-target delivery instead of separate crates per output artifact.

V.B Simulation Architecture

Unified pipeline with per-platform drivers

05.04.2026

Context

The thesis requires simulation support for two or more platforms with very different dynamics and sensor suites. Each platform uses a different external simulation engine (RocketPy and RotorPy) and produces a different set of sensor measurements. The downstream evaluation pipeline must consume data from both platforms.

Decision

We implement a **single simulation pipeline** with a per-platform driver. The drivers wrap the external simulation engines and adapt their outputs to a shared `SimulationResult` contract. Downstream transformations and evaluations act on this contract and are simulator-agnostic.

Alternatives

- **Two independent pipelines, one per platform:** Simpler per-platform code with no abstraction overhead but every shared module (noise models, dropout, metrics, plots) would need to be implemented and maintained twice.

Consequences

Positive

- Noise models, dropout strategies, synthetic sensor generation and evaluation are written once and applied independent of simulation engine.
- Adding a new platform requires only a new driver that adopts the `SimulationResult` contract. No changes to the pipeline or evaluation code is needed.

Negative

- The driver layer must hide all simulator-specific concepts behind the contract, which adds an upfront design cost and constrains what contract can express.
- Features available in only one simulation engine cannot be exposed without either extending the contract or breaking the abstraction.

ADR 7: Decision to use a unified pipeline with per-platform drivers instead of two separate pipelines.

Context

Setting up a simulation run requires the construction of all required objects. For the rocket, this includes the motor, the rocket body, the environment and the flight. For the drone, this includes the vehicle, the controller, the reference trajectory and the IMU sensor. Each of these components has many parameters.

Decision

We use the **Builder pattern** to construct the scenario objects required by each simulation driver. For each platform, a builder dataclass exposes a `build()` method that produces the final scenario object with the set parameters. Defaults are encoded in the builder, so a minimal scenario requires only a few explicit parameters to be set by the developer.

Alternatives

- **Direct construction of the scenario objects in each notebook:** Simplest, but every notebook duplicates the full set of parameters. Parameter changes require updating every notebook that constructs a similar scenario.
- **Configuration files (YAML/JSON) parsed into the simulator's constructors:** Cleaner separation of configuration from code, but loses programmatic flexibility.
- **Factory functions per scenario:** Lighter than a builder class for pre-defined scenarios, but gets complicated when many parameters are optional or when scenarios need to be derived from other scenarios.

Consequences

Positive

- Notebooks set only the parameters that differ from the defaults, which keeps scenario configuration minimal.
- New parameters can be added to a builder without breaking existing scenarios, as long as a sensible default is provided.
- Specialized builders enable scenario variants that include random configurations.

Negative

- Each platform requires its own builder class, which is upfront design work before the first scenario can run.
- Purely declarative scenario definitions are not supported. There only exists one default per builder.

ADR 8: Decision to use the Builder pattern for scenario object construction in the simulation drivers.

Context

The simulation container produces data. The evaluation container consumes that data, drives the EKF and computes metrics. The two subsystems must agree on a core data structure to remain independently developable.

Decision

We developed a **Core Domain Model**. It is a set of Python dataclasses (`GroundTruth`, `SensorStream`, `SimulationResult`) implemented in the `core/` module. This is the data contract between simulation and evaluation. The dataclasses must be simulator-agnostic so that additional simulator engines can be added later.

Alternatives

- **Pass simulator-native objects directly between subsystems:** Zero data translations needed, but couples the evaluation to the internal datastructure of the simulator engines. Adding or replacing a simulation engine would require changes in the evaluation code.
- **Use pandas DataFrames as the contract:** Lightweight and widely supported, but forces sensors with different sampling rates onto a shared time index and offers no type safety for sensor-specific fields.

Consequences

Positive

- The simulation and evaluation containers can evolve independently as long as the contract is well defined and used.
- Schema validation and unit checks live in one place (the `validate` method on each dataclass).
- Type hints on the dataclasses give IDE support and static analysis benefits across the entire pipeline.
- Adding a new simulation engine or a new evaluation step does not require changes on the other side.

Negative

- The contract must be designed conservatively. Extending it later requires adjustments in both containers and rerunning all simulation.

ADR 9: Decision to define an explicit domain model as the simulation-evaluation handoff.

Context

The simulation requires a lot of parameters and configurations that a developer has to make. Additionally, the evaluation has a lot of configurable parts (like plots or metrics). Thus, an interface for interacting with these software modules has to be defined.

Decision

We use **Jupyter notebooks** as the primary entry point for running simulations and evaluations. Each simulation notebook constructs a simulation run, invokes a driver, optionally applies pipeline stages, plots the result and exports `SimulationResult` to disk. Each evaluation notebook imports a `SimulationResult`, makes a prediction using an EKF instance, evaluates the result and outputs plots and metrics.

Alternatives

- **Command-line tool with YAML scenario files:** Cleaner separation of code and configuration, easier to script but requires an interaction layer to be present.
- **Individual python scripts:** Lightweight and version-controllable, but can only be executed fully. Thus, rerunning only a single step is not possible.
- **Web UI or dashboard:** Best for non-developer users, but too time intensive to build for this thesis.

Consequences

Positive

- Configuration, execution, plotting and inspection can happen stepwise and individual code changes can be made without rerunning the whole program.
- Plots and metrics can be rendered inline next to the code that produced the run or evaluation.
- Notebooks can contain text blocks that contain additional information.

Negative

- Notebooks are harder to diff and review than plain Python scripts.
- Depending on the IDE, notebooks may need additional plugins to inspect and run.
- Importing logic from a notebook is complicated, so scenario configuration cannot easily be reused across notebooks. Shared logic has to be extracted into a regular Python module.

ADR 10: Decision to use Jupyter notebooks as the simulation and evaluation entry point.

Context

Simulation runs should be persisted and reused across many evaluations to avoid recomputation. The serialization format must be able to save and load instances of `SimulationResult`. The initial implementation used CSV files for sensor streams plus a YAML file for additional information.

Decision

We use the module `pickle` as the primary persistence format for `SimulationResult` instances. The domain model exposes `save()` and `load()` methods that wrap `pickle.dump` and `pickle.load`. The pre-existing CSV+YAML export remains available via `export_csv_data()` and a corresponding `CSVDataLoader` but it is no longer used in the main pipeline.

Alternatives

- **CSV+YAML:** Human-readable and diff-friendly, but every change to the domain model requires updating the (de)serialization logic.

Consequences**Positive**

- Saving and loading a `SimulationResult` requires no per-field code and any change to the domain model is automatically supported.
- The save and loading process happens fast and is easy to implement

Negative

- Pickle files are not human-readable and cannot be inspected with diff tools
- Pickle is tied to the Python version and to the exact configuration used when saving. Renaming a dataclass or adding members invalidates older pickle files.
- Pickle is unsafe to load from untrusted sources. This is acceptable here because all pickle files are produced by our own pipeline.

ADR 11: Decision to use pickle as the primary persistence format and retain CSV+YAML for legacy and inspection use.

Context

The three pipeline stages (`SyntheticSensorGenerator`, `NoiseInjector` and `DropoutInjector`) each apply one of several interchangeable behaviors to a `SensorStream`.

- Synthetic sensors differ in how they derive their measurement from `GroundTruth`
- Noise injection differ in how they add noise to a clean signal
- Dropout differs in how samples are selected and how they are replaced/deleted

Decision

We apply the **Strategy pattern** to all three pipeline stages. Each behavior is implemented as a separate class deriving from a stage-specific abstract base class. Adding a new sensor, noise profiles, or dropout mode requires only writing a new strategy class deriving from the base. The pipeline stage itself does not change.

Alternatives

- **Conditional dispatch inside each stage:** Simpler for a small number of behaviors, but each new behavior modifies the stage's main method. The stage class grows over time.

Consequences**Positive**

- Adding a new strategy can be done without any changes to the pipeline stage.
- Each strategy class can be tested in isolation without involving the full pipeline.
- Strategy classes carry their own configuration parameters as fields, which keeps configuration close to the logic that uses it.

Negative

- More boilerplate per behavior. Each strategy needs a class declaration, even when the logic is a few lines.

ADR 12: Decision to use the Strategy pattern for the synthetic sensor, noise and dropout pipeline stages.

V.C Evaluation Architecture

KalmanEstimate as the runner output contract

22.04.2026

Context

The evaluation stages must be able to consume the output of any EKF runner. Future runners (alternative filter implementations or additional platforms) must be able to integrate into the pipeline without needing modification of other stages.

Decision

We define a runner-agnostic dataclass `KalmanEstimate` as the output contract of every runner. The runner is responsible for unpacking the EKF state, applying any frame conversions and assembling the dataclass. Downstream stages only operate on this `KalmanEstimate`, not directly on outputs of the EKF implementation.

Alternatives

- **Each metric pulls what it needs from the EKF state directly:** The metric stage receives the runner's raw EKF state and decodes it as needed. Removes one layer of abstraction but couples every metric to the EKF's state-vector layout. Any changes to the filter's internals propagates into the metric code.

Consequences

Positive

- New runners can be added by writing a single subclass of `KalmanRunner`, no other component changes.
- Metric and visualizer code references named fields with explicit units. This makes the implementation easier to read.
- The `KalmanEstimate` can be made very similar to the `GroundTruth` class which makes evaluation code easier.

Negative

- Some runner-specific information cannot be expressed in the generalized fields and must be passed through the `metadata` dictionary.

ADR 13: Decision to define a runner-agnostic dataclass as the output contract of every runner.

Context

The evaluation has three individual stages where multiple interchangeable implementations can be used:

- time aligner: Different alignment strategies for aligning `GroundTruth` and `KalmanEstimate` to simplify metric calculation logic
- Runners: One per platform, plus a passthrough variant for testing
- Metrics: One or multiple metrics evaluated on every run

Each of these stages must be replaceable without touching the orchestration code in the `Evaluator` or in the parameter search module.

Decision

The **strategy pattern** is used. Each of the three stages is implemented as an abstract base class with a single primary method. Concrete strategies derive from the base class and are passed into the `Evaluator` as constructor arguments. The orchestration code references only the abstract bases and never the concrete subclasses.

Alternatives

- **Conditional dispatch inside the Evaluator:** The `Evaluator` carries platform-specific or metric-specific code paths and selects them based on configuration flags. Avoids the abstract-base layer but couples the orchestrator to every supported strategy. Thus, orchestration code would need to be changed if a new strategy is added.

Consequences**Positive**

- New runners, metrics, or aligners are added by just writing a new subclass.
- Strategies can be tested in isolation.
- Strategies are composable in arbitrary combinations from a notebook, which enables fast configuration changes.

Negative

- The abstract bases are small and add some boilerplate per concrete strategy.

ADR 14: Decision to use the strategy pattern for runners, metrics and time aligners

Context

The EKF output rate and the ground-truth rate generally differ. Thus, an estimate and a ground truth cannot be compared element-wise directly and need to be aligned. This can be done using different strategies (nearest-neighbour, linear interpolation, spline).

Decision

We extract resampling into a dedicated **time alignment stage** in front of the metric stage. The `TimeAligner` returns a new estimate and ground-truth pair on a shared time grid. The metrics operate on this aligned pair without performing any resampling themselves.

Alternatives

- **Each metric resamples on its own if required:** Every metric pairs estimate and ground-truth samples on its own if required. Avoids the extra stage but duplicates the resampling logic across multiple metrics. Would also add to the complexity of each metric calculation.
- **Force estimate and ground truth to share a time grid:** The runner produces an estimate at exactly the ground-truth timestamps. Avoids the alignment problem entirely but couples the runner to the simulation rate and prevents the runner from being driven at the natural IMU rate. Defeats the purpose of running the EKF at its target rate and introduces coupling between simulation and evaluation.

Consequences

Positive

- Metrics are short and side-effect-free, which makes them easier to implement and test.
- Alternative alignment strategies (for example a future interpolating aligner) can be introduced without touching any metrics.
- The time alignment can be tested in isolation.

Negative

- The `Evaluator` must run two stages in the correct order rather than one, which adds orchestration logic.

ADR 15: Decision to separate time alignment from metric computation and require metrics to operate on aligned data only.

Context

The EKF noise parameters may be tuned experimentally against simulation runs. Performing a manual search for the optimal parameter combination is expensive and time intensive. Thus, an automatic and systematic approach is needed to optimize the noise parameters.

Decision

We use **Bayesian optimization via the Optuna framework** [40] for the parameter optimization. Optuna by default uses a Tree-structured Parzen Estimator (TPE) sampler that is used to find the optimal parameters for a given objective. Multi-objective runs are also possible.

Alternatives

- **Grid search:** The search space is set up as a regular grid and every combination is evaluated. Easy to implement and would also run great in parallel, but scales poorly with the number of parameters.
- **Manual tuning:** Parameters are adjusted by hand based on plotted filter behavior. Would require no additional coding but is impractical and time consuming.

Consequences**Positive**

- Optuna takes care of the parameter selection and reaches good objective values within a much smaller trial budget than grid search.
- Adding a new parameter to the search requires only an additional parameter and no additional changes to the search module.
- Optuna provides multi-objective support which can be utilized if needed.
- Optuna provides a dashboard with plots and figures that help evaluating the results.

Negative

- The Optuna dependency adds an external library to the evaluation environment.
- The parameter selection done by Optuna is a “black-box” and can’t be controlled well.
- The optimization has to be done sequentially. Trials cannot be parallelized because they depend on the result of the previous trial.

ADR 16: Decision to use Bayesian optimization via Optuna for the parameter search.

VI Sensor Specifications

VI.A ARIS Rocket “Asteria”

VI.A.a LSM6DSO32 - IMU (Accelerometer + Gyroscope)

Parameter	Value	Unit
Accelerometer full-scale range	$\pm 4 / \pm 8 / \pm 16 / \pm 32$	g
Gyroscope full-scale range	$\pm 125 / \pm 250 / \pm 500 / \pm 1000 / \pm 2000$	dps
Accelerometer output data rate	1.6 – 6664	Hz
Gyroscope output data rate	12.5 – 6664	Hz
Accelerometer noise density (HP mode)	120 (± 4 g) – 220 (± 32 g)	$\mu\text{g}/\sqrt{\text{Hz}}$
Gyroscope noise density (HP mode)	3.8	$\text{mdps}/\sqrt{\text{Hz}}$
Accelerometer level offset accuracy	± 20	mg
Angular rate level offset accuracy	± 0.5	dps
Linear acceleration sensitivity tolerance	± 0.5	%
Angular rate sensitivity tolerance	± 0.5	%
Operating temperature range	-40 to +85	°C
Available interfaces	SPI / I2C / MIPI I3C	
Accelerometer output unit	mg	
Accelerometer output model	cartesian	
Gyroscope output unit	dps	
Gyroscope output model	euler	

Table 71: Key specifications of the LSM6DSO32 (measured at VDD = 1.8 V, T = 25 °C)

[\[3\]](#)

VI.A.b MS5607 - Altimeter Module

Parameter	Value	Unit
Operating pressure range	300 – 1100	mbar
Extended pressure range	10 – 2000	mbar
Sample frequency via I2C	up to 400	kHz
Total error band, no autozero	at 25°C, 700..1100 mbar ± 1.5 at 0..50°C, 300..1100 mbar ± 2 at -20..85°C, 300..1100 mbar ± 3.5 at 0..70°C, 1100..1600 mbar ± 8 at 0..70°C, 1100..2000 mbar ± 12	mbar
Operating temperature range	-40 to +85	°C
Available interfaces	SPI / I2C	
Output unit	mbar	

Table 72: Key specifications of the MS5607 (measured at VDD = 3 V, T = 25°C)

[\[4\]](#)

VI.A.c LSM303AGR - eCompass (Magnetometer and Accelerometer)

Parameter	Value	Unit
Accelerometer full-scale range	$\pm 2 / \pm 4 / \pm 8 / \pm 16$	g
Magnetic dynamic range	± 49.152	gauss
Accelerometer ODR (I2C)	1 – 5376	Hz
Magnetometer ODR (I2C)	10 / 20 / 50 / 100	Hz
Accelerometer RMS noise (ODR = 100 Hz, HR mode, ± 2 g)	3	mg
Magnetic RMS noise (HP mode)	3	mgauss
Accelerometer RMS noise (HR mode, ± 2 g)	3	mg
Magnetic RMS noise (HP mode)	3	mgauss
Magnetic sensor offset (with offset cancellation)	± 60	mgauss
Operating temperature range	-40 to +85	°C
Available interfaces	SPI / I2C	
Accelerometer output unit	mg	
Accelerometer output model	cartesian	
Magnetometer output unit	mgauss	
Magnetometer output model	three-axis flux density probe	

Table 73: Key specifications of the LSM303AGR (measured at VDD = 2.5 V, T = 25 °C)

[\[5\]](#)

VI.A.d SHT4 - Humidity and Temperature

Parameter	Value	Unit
Max ODR	770	Hz
RH accuracy	±1.8	%RH
Temperature accuracy	±0.2	°C
RH repeatability (high / med / low)	0.08 / 0.15 / 0.25	%RH
Temperature repeatability (high / med / low)	0.04 / 0.07 / 0.1	°C
RH specified range	0 – 100	%RH
Temperature specified range	–40 to +125	°C
RH response time	4	s
Temperature response time	2	s
Operating temperature range	–40 to +125	°C
Available interfaces	I2C	
RH output unit	%RH	
Temperature output unit	°C	

Table 74: Key specifications of the SHT4 [\[6\]](#)

VI.A.e ZED-F9P-05B - GNSS

Parameter	Value	Unit
Supported constellations	GPS + GLONASS + Galileo + BeiDou	
Horizontal position accuracy (CEP)	1.5	m
Vertical position accuracy (median)	2.25	m
Velocity accuracy (50%, at 30 m/s)	0.05	m/s
Dynamic heading accuracy (50%, at 30 m/s)	0.3	deg
Max navigation update rate (PVT, 4 constellations)	25	Hz
Cold start acquisition time	25	s
Hot start acquisition time	2	s
Tracking sensitivity	-167	dBm
Reacquisition sensitivity	-160	dBm
Operational dynamics limit	≤ 4	g
Operational altitude limit	80 000	m
Operational velocity limit	500	m/s
Operating temperature range	-40 to +85	°C
Available interfaces	UART / SPI / I2C / USB	
Output unit	degrees and minutes	
Output model	WGS84	

Table 75: Key specifications of the ZED-F9P-05B

[\[7\]](#)

VI.B Crazyflie 2.1

VI.B.a BMI088 - IMU (Accelerometer + Gyroscope)

Parameter	Value	Unit
Accelerometer full-scale range	$\pm 3 / \pm 6 / \pm 12 / \pm 24$	g
Gyroscope full-scale range	$\pm 125 / \pm 250 / \pm 500 / \pm 1000 / \pm 2000$	dps
Accelerometer output data rate	12.5 – 1600	Hz
Gyroscope output data rate	100 / 200 / 400 / 1000 / 2000	Hz
Accelerometer noise density	160 (X/Y) / 190 (Z)	$\mu\text{g}/\sqrt{\text{Hz}}$
Gyroscope noise density	0.014	$^{\circ}/\text{s}/\sqrt{\text{Hz}}$
Accelerometer Zero-g Offset	20	mg
Gyroscope Zero-g Offset	1	dps
Operating temperature range	-40 – +85	$^{\circ}\text{C}$
Available interfaces	SPI / I ² C	

Table 76: Key specifications of the BMI088 [\[8\]](#)

VI.B.b BMP388 - Barometric Pressure Sensor

Parameter	Value	Unit
Operating pressure range	300 – 1250	hPa
Absolute pressure accuracy (300–1100 hPa, –20–65 °C)	±0.50	hPa
Relative pressure accuracy (900–1100 hPa, 25–40 °C)	±0.08	hPa
Temperature coefficient offset (700–1100 hPa, 0–55 °C)	±0.75	Pa/K
Pressure noise (RMS, full bandwidth)	1.2	Pa
Pressure noise (RMS, lowest bandwidth)	0.03	Pa
Pressure resolution (highest resolution mode)	0.016	Pa
Absolute temperature accuracy (0–65 °C)	±0.50	°C
Output data rate	up to 200	Hz
Operating temperature range	–40 – +85	°C
Available interfaces	SPI / I ² C	

Table 77: Key specifications of the BMP388 [\[9\]](#)

VI.B.c PMW3901 - Optical Flow Sensor

Parameter	Value	Unit
Minimum operating distance	80	mm
Effective viewing angle	42	°
Motion data resolution	16	bit
SPI clock frequency	2	MHz
Operating temperature range	0 – +40	°C
Available interfaces	SPI	

Table 78: Key specifications of the PMW3901 [\[10\]](#)

VI.B.d VL53L1x - Time-of-Flight Distance Sensor

Parameter	Value	Unit
Ranging distance (long-range mode)	up to 3600	mm
Ranging distance (short-range mode)	up to 1360	mm
Minimum ranging distance	40	mm
Ranging accuracy (long-range, typ.)	±20	mm
Field of view	15–27	deg
Measurement rate	up to 50	Hz
Operating temperature range	–20 – +85	°C
Available interfaces	I2C	

Table 79: Key specifications of the VL53L1x [\[11\]](#)

VI.B.e DWM1000 - UWB Transceiver (Loco Positioning Deck)

Parameter	Value	Unit
UWB channels	1, 2, 3, 5	
Operating frequency	3.244 – 6.999	GHz
Channel bandwidth	500	MHz
Data rate	110 / 850 / 6800	kbps
Ranging accuracy (TWR)	±10	cm
Positioning accuracy	10	cm
Supply voltage	2.8 – 3.6	V
Operating temperature range	–40 – +85	°C
Available interfaces	SPI	

Table 80: Key specifications of the DWM1000 [\[12\]](#)

VII Detailed Analysis of the Existing Crazyflie EKF Implementation

The existing Crazyflie EKF is split across two layers. The file `estimator_kalman.c` handles the FreeRTOS task, the measurement queue and the timing logic. The actual filter math lives in `kalman_core.c` and the measurement model files in `kalman_core/`.

VII.A EKF System Model for the Crazyflie

This chapter describes the EKF system model as implemented in `kalman_core.c` and the measurement model files.

VII.A.a State Vector

The state vector contains nine elements organized into three groups:

```
C kalman_core.h
1 // Position in the global (world) frame [m]
2 S[KC_STATE_X], S[KC_STATE_Y], S[KC_STATE_Z]
3
4 // Velocity in the body frame [m/s]
5 S[KC_STATE_PX], S[KC_STATE_PY], S[KC_STATE_PZ]
6
7 // Attitude error [rad]
8 S[KC_STATE_D0], S[KC_STATE_D1], S[KC_STATE_D2]
```

Code Snippet 16: Crazyflie EKF state vector.

Position is expressed in the global frame because sensors like UWB provide global measurements. Velocity is expressed in the body frame because the IMU measures accelerations and angular rates in the body frame, making the prediction equations simpler. The attitude error states D0–D2 represent a small-angle correction rather than the full orientation. This follows the Multiplicative Extended Kalman Filter (MEKF) approach [24], where tracking the quaternion directly would break its unit-norm constraint since the Kalman filter updates states by linear addition. Instead, only the three-component error is tracked in the filter, while the true orientation is maintained separately as a quaternion `q` and rotation matrix `R`. After each update step, D0–D2 are folded back into `q` and reset to zero by `kalmanCoreFinalize()`.

VII.A.b Process Model $f(\mathbf{x}, \mathbf{u})$

The process model describes how the state evolves over time given the IMU inputs (accelerometer `acc` and gyroscope `gyro`). It is implemented in `predictDt()`. The model distinguishes two modes via `quadIsFlying`.

When airborne, PX and PY are updated using only the gyroscope and gravity, since the drone can only thrust along its body Z axis. Only PZ receives the accelerometer input `zacc`. A drag model damps all three velocity components:

C

kalman_core.c

```

1  S[KC_STATE_PX] += dt*( gyro->z*PY - gyro->y*PZ - g*R[2][0] -
dragB_x*PX);
2  S[KC_STATE_PY] += dt*(- gyro->z*PX + gyro->x*PZ - g*R[2][1] -
dragB_y*PY);
3  S[KC_STATE_PZ] += dt*( zacc + gyro->y*PX - gyro->x*PY - g*R[2][2] -
dragB_z*PZ);

```

Code Snippet 17: Crazyflie EKF velocity update equations for the airborne case.

When not flying, all three accelerometer axes drive the velocity update and the drag model is not applied. In both modes, the position update rotates the body-frame displacement into the global frame using R .

VII.A.c Jacobian A

The Jacobian A is computed analytically inside `predictDt()` and used to propagate the covariance as $P^- = A P A^T$.

The diagonal is always initialized to 1 (identity). The off-diagonal entries encode how each state influences others over one time step. The most important entries are:

- Position from velocity: $A[X][PX] = R[0][0]*dt$ etc. the body-frame velocity rotated into the global frame.
- Velocity from velocity: Includes gyroscope and gravity coupling between axes ($A[PX][PY] = gyro->z*dt$) and drag damping when flying ($A[PX][PX] = 1 - dt*dragB_x$).
- Position from attitude error: Encodes how a small rotation changes the predicted global-frame position, e.g. $A[X][D0] = (PY*R[0][2] - PZ*R[0][1])*dt$.
- Velocity from attitude error: Encodes how a small rotation changes the gravity projection into the body frame, e.g. $A[PY][D0] = -g*R[2][2]*dt$.

VII.A.d Measurement Models $h(x)$

Each sensor has its own measurement model that defines how the measurement relates to the current state. All models are linearized around the current state estimate to produce an H-vector, which is the row of the Jacobian H. The H-vector is then passed together with the innovation to `kalmanCoreScalarUpdate()`.

The sensors used on the Crazyflie and their measurement models are:

- **BMI088 (IMU)**: Not used in the update step. Accelerometer and gyroscope data drives the process model exclusively.
- **BMP388 (Barometer)**: Measures atmospheric pressure, converted to altitude above sea level. $h[KC_STATE_Z] = 1$. Disabled by default.
- **VL53L1x (ToF)**: Measures distance to the ground. Corrected for sensor tilt: $h[KC_STATE_Z] = 1/\cos(\alpha)$ where α is the tilt angle derived from $R[2][2]$.
- **PMW3901 (Optical flow)**: Measures pixel displacement between frames. The model back-projects the displacement to body-frame velocity using the current height $S[KC_STATE_Z]$ and corrects for the angular velocity and deck lever arm. Two separate scalar updates (X and Y).
- **DWM1000 (UWB TWR)**: Measures distance to a fixed anchor. The predicted measurement is the Euclidean distance from $S[X, Y, Z]$ to the anchor. H is the normalized direction vector from estimated position to anchor.

VII.B EKF flow diagram

Figure 91 shows the control flow of the Crazyflie EKF, from sensor data ingestion through the predict and update steps to the final state output.

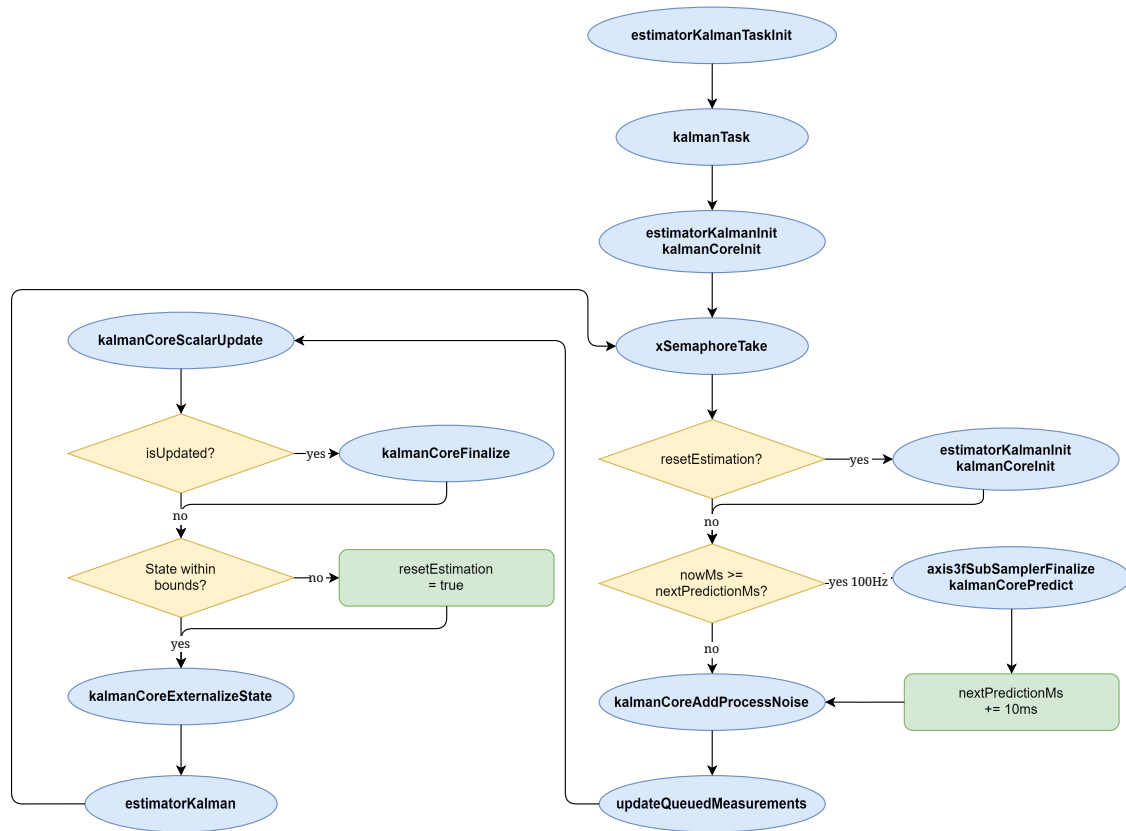


Figure 91: Control flow of the Crazyflie EKF.

VII.C EKF functions

The following sections describe each function and how they call each other.

VII.C.a `estimatorKalmanTaskInit()`

Called once at system startup. Creates the FreeRTOS task `kalmanTask`, the run semaphore `runTaskSemaphore` and the data mutex `dataMutex`.

VII.C.b `estimatorKalmanInit()`

Called when the estimator is selected or when a reset is triggered from `kalmanTask`. It then initializes the two IMU sub-samplers. The scale factors ensure that the accumulated accelerometer samples are converted from raw g to m/s^2 and gyroscope samples from degrees to radians before being passed to `predictDt()`. After the sub-samplers, it calls `kalmanCoreInit()` to initialize the filter state and covariance matrix.

VII.C.c `estimatorKalman()`

Called by the stabilizer loop at 1000 Hz. It copies the latest estimated state from `taskEstimatorState` into the stabilizer's `state_t` struct under the data mutex, then releases `runTaskSemaphore` to trigger one iteration of `kalmanTask`.

VII.C.d kalmanTask()

The main EKF loop. Wakes up each time `estimatorKalman()` releases the semaphore. Each iteration runs the following steps in order:

1. Calls `kalmanCorePredict()` if the 100 Hz prediction interval has elapsed.
2. Calls `kalmanCoreAddProcessNoise()` every tick.
3. Calls `updateQueuedMeasurements()` to process all pending sensor data.
4. Calls `kalmanCoreFinalize()` to incorporate the result.
5. Copies the updated state to `taskEstimatorState` via `kalmanCoreExternalizeState()` under the data mutex.

VII.C.e kalmanCoreInit()

Initializes the filter state. Sets the position to the configured initial values (`params->initialX/Y/Z`), sets all velocities and attitude errors to zero and initializes the quaternion from `params->initialYaw`. The covariance matrix `P` is initialized with the configured standard deviations on the diagonal and zero everywhere else.

VII.C.f axis3fSubSamplerFinalize()

Called inside `kalmanTask` directly before `kalmanCorePredict()`, once every 10 ms (100 Hz). The sub-sampler accumulates all IMU measurements that arrived in the queue since the last prediction via `axis3fSubSamplerAccumulate()`. When `axis3fSubSamplerFinalize()` is called, it computes the average of all accumulated samples, scales them by the factor provided at initialization (`GRAVITY_MAGNITUDE` for the accelerometer, `DEG_TO_RAD` for the gyroscope) and writes the result into `subSample`. The counter is then reset to zero so the next accumulation window starts fresh.

The resulting `accSubSampler.subSample` and `gyroSubSampler.subSample` are passed directly to `kalmanCorePredict()`:

This decouples the IMU sampling rate (1000 Hz) from the prediction rate (100 Hz) without losing any measurements.

VII.C.g kalmanCorePredict()

Computes the elapsed time since the last prediction and calls `predictDt()` with that delta.

VII.C.h predictDt()

The core prediction step. Takes the sub-sampled accelerometer reading `acc` and gyroscope reading `gyro` as inputs and propagates the state forward by `dt` seconds.

The function uses two modes via `quadIsFlying`. When airborne, only the body-frame Z acceleration (thrust) drives the velocity update. `KC_STATE_PX` and `KC_STATE_PY` are updated using only the gyroscope and gravity, since the drone can only produce thrust along its body Z axis.

Only `KC_STATE_PZ` receives `zacc` (the accelerometer Z reading). When not flying, all three accelerometer axes are used for the velocity update.

In both modes the position update rotates the body-frame displacement into the global frame using the rotation matrix `R`:

```

C                                                                    kalmancore.c
1  this->S[KC_STATE_X] += this->R[0][0]*dx + this->R[0][1]*dy + this->R[0]
2  [2]*dz;
3  this->S[KC_STATE_Y] += this->R[1][0]*dx + this->R[1][1]*dy + this->R[1]
4  [2]*dz;
5  this->S[KC_STATE_Z] += this->R[2][0]*dx + this->R[2][1]*dy + this->R[2]
6  [2]*dz
7                          - GRAVITY_MAGNITUDE * dt2 / 2.0f;

```

Code Snippet 18: Crazyflie EKF position update rotating body-frame displacement into the global frame.

The attitude quaternion is updated by integrating the gyroscope reading as a delta quaternion and multiplying it onto the current quaternion q . Finally the covariance is propagated as $P^- = APA^T$ using the analytically computed Jacobian A .

VII.C.i kalmanCoreAddProcessNoise()

Computes the elapsed time since the last process noise update and calls `addProcessNoiseDt()`.

VII.C.j addProcessNoiseDt()

Adds process noise to the diagonal of P to reflect the growing uncertainty between measurement updates. Each state gets a separate noise term scaled by dt :

```

C                                                                    kalman_core.c
1  this->P[KC_STATE_X][KC_STATE_X] += powf(params->procNoiseAcc_xy*dt*dt
2                                     + params->procNoiseVel*dt
3                                     + params->procNoisePos, 2);
4  this->P[KC_STATE_PX][KC_STATE_PX] += powf(params->procNoiseAcc_xy*dt
5                                     + params->procNoiseVel, 2);
6  this->P[KC_STATE_D0][KC_STATE_D0] += powf(params->measNoiseGyro_rollpitch*dt
7                                     + params->procNoiseAtt, 2);

```

Code Snippet 19: Crazyflie EKF process noise terms added to the covariance diagonal.

This runs at 1000 Hz.

VII.C.k updateQueuedMeasurements()

Drains the full measurement queue each tick by calling `estimatorDequeue()` in a loop. For each entry it dispatches by `MeasurementType` to the corresponding update function.

IMU measurements (gyroscope and accelerometer) are not passed to an update function, they are accumulated in the sub-samplers for the next `kalmanCorePredict()` call. All other types trigger a direct EKF update via `kalmanCoreScalarUpdate()`.

VII.C.l kalmanCoreScalarUpdate()

The central update function called by every measurement model. It takes the H-vector (the linearized measurement model row), the innovation (measured minus predicted) and the measurement noise standard deviation. It then performs the full EKF update in three steps:

Kalman gain:


```

C kalman_core.c
1  for (int i=0; i<KC_STATE_DIM; i++) {
2      K[i] = PHTd[i] / HPHR; // K = P H' (H P H' + R)^-1
3  }

```

Code Snippet 20: Crazyflie EKF Kalman gain computation.

State update:

```

C kalman_core.c
1  for (int i=0; i<KC_STATE_DIM; i++) {
2      this->S[i] = this->S[i] + K[i] * error;
3  }

```

Code Snippet 21: Crazyflie EKF state update applying the Kalman gain.

Covariance update:

```

C kalman_core.c
1  // (KH - I) P (KH - I)' + K R K'
2  mat_mult(&Km, Hm, &tmpNN1m); // KH
3  for (int i=0; i<KC_STATE_DIM; i++) {
4      tmpNN1d[KC_STATE_DIM*i+i] -= 1; // KH - I
5  }
6  mat_trans(&tmpNN1m, &tmpNN2m); // (KH - I)'
7  mat_mult(&tmpNN1m, &this->Pm, &tmpNN3m); // (KH - I) P
8  mat_mult(&tmpNN3m, &tmpNN2m, &this->Pm); // (KH - I) P (KH - I)'

```

Code Snippet 22: Crazyflie EKF covariance update using the Joseph form.

After the update, `P` is symmetrized and clamped between `MIN_COVARIANCE` and `MAX_COVARIANCE`. The flag `isUpdated` is set to `true` to signal `kalmanCoreFinalize()` that there is new data to process.

Each measurement model file computes the innovation and H-vector for its sensor and then calls `kalmanCoreScalarUpdate()`. [Table 81](#) lists all measurement models and their key implementation details.

File	Sensor	Implementation note
<code>mm_tof.c</code>	VL53L1x	Guards against tilt ($R[2][2] > 0.1$). H-vector: $h[KC_STATE_Z] = 1/\cos(\alpha)$.
<code>mm_flow.c</code>	PMW3901	Two separate calls to <code>kalmanCoreScalarUpdate()</code> , one for X and one for Y. Accounts for lever-arm of Flow Deck v2 via $\omega \times r$. Uses $S[KC_STATE_Z]$ to scale pixel displacement to velocity.
<code>mm_distance.c</code>	DWM1000 TWR	H-vector is the normalized direction vector from estimated position to anchor. Robust variant <code>mm_distance_robust.c</code> uses Geman-McClure cost function with 2 iterations and calls <code>kalmanCoreUpdateWithPKE()</code> instead.
<code>mm_yaw_error.c</code>	Generic	H-vector: $h[KC_STATE_D2] = 1$.

Table 81: Measurement model files and their calls to `kalmanCoreScalarUpdate()`.

Note that `kalmanCoreUpdateWithBaro()` is implemented directly in `kalman_core.c` rather than a separate file. It is disabled by default because `KALMAN_USE_BARO_UPDATE` is not defined in `estimator_kalman.c`.

VII.C.m `kalmanCoreFinalize()`

Called after all measurement updates. It only runs if `isUpdated` is `true`. It folds the attitude error states $D0 - D2$ back into the quaternion q by computing a correction delta quaternion and multiplying it onto q . The rotation matrix R is then recomputed from the updated quaternion. Finally $D0$, $D1$ and $D2$ are reset to zero and P is symmetrized and clamped.

VII.C.n `kalmanCoreExternalizeState()`

Converts the internal filter state into the `state_t` struct used by the rest of the firmware. Position is copied directly from S since it is already in the global frame. Velocity is rotated from body frame to global frame using R . Attitude is converted from quaternion to Euler angles (roll, pitch, yaw).

VII.D Sensor Dropout Handling

If a sensor stops producing measurements, nothing is enqueued for that type and the corresponding update function is never called. The filter continues on the remaining sensors without any explicit detection or mode switching. The covariance P grows continuously via `addProcessNoiseDt()`, reflecting the increasing uncertainty.

If the state diverges beyond the configured bounds, `kalmanSupervisorIsStateWithinBounds()` returns `false` and `kalmanTask` sets `resetEstimation = true`, triggering a full re-initialization via `kalmanCoreInit()` on the next tick.

The only sensor with an explicit guard is the ToF model in `mm_tof.c`, which skips the update when $\text{fabs}(R[2][2]) < 0.1$ to avoid a singularity in the tilt correction. No other sensor has dedicated dropout logic.