Radar Drone Detection: Multi-Target Tracking

BSc Thesis

M. J. Pehlivan & I.T.A. Kool

Delft University of Technology

TU Delft

Radar Drone Detection: Multi-Target Tracking

BSc Thesis

by

M. J. Pehlivan & I.T.A. Kool

Supervisors: prof.dr. O. Yarovyi
dr. N. Kruse
Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Nick Núñez, Pexels

TU Delft

Abstract

In this thesis, the design and implementation of a target tracking subsystem is outlined, as part of a radar-based drone detection system. The end-goal of the tracking subsystem is to provide actionable data to the end-user, to allow appropriate action to be taken within the target application. Radar detections are simulated with their associated noise, uncertainty, and clutter. Field measurements show that the tracker is able to initiate tracks and follow the drone trajectory despite the added challenge of sensor measurement noise, missed detections and false detections.

Preface

This project is realized in close collaboration with the Dopplium company as part of the BSc electrical engineering curriculum. We would like to sincerely thank our supervisors prof. dr. Olexander Yarovyi and dr. Nicolas Kruse for their direct support, dr. Francesco Fioranelli and dr. Geethu Joseph for their valuable feedback during the project. Lastly, dr. Ignacio Roldan from Dopplium Radar and the MS3 department at TU Delft for their expertise and generous granting of access to the necessary equipment.

M. J. Pehlivan & I.T.A. Kool
Delft, June 2026

Contents

1 Introduction

Unmanned aerial vehicles (UAVs), more often referred to as drones, have become increasingly common in the past decade, due to a combination of low cost, versatility, and ease of use. The risk associated with drone use remains high. For example; malicious cyberattacks can be performed to potentially repurpose drones to carry out a terrorist attack [1]. Furthermore, monitoring restricted airspaces such as airports is vital to ensure the safety of the public. It is important to take measures to mitigate these threats. A radar-based drone detection system can aid in early spotting of potential threats and give actionable data to the responders. Tracking is a vital part of this detection system, predominantly to provide this actionable data from noisy measurement data with lower interpretability, where the measurement data is target range and angle. The noisy measurement data is combined with predictions based on an internal dynamical model to increase the tracking accuracy. Furthermore, the tracking system is designed to handle a multi-target environment, even with overlapping flight paths.

1.1. The Goal of the Project

The end-goal of the overarching project is to develop an end-to-end radar detection system. The first step is to characterize the electromagnetic properties at relevant frequencies. Secondly, to develop a processing pipeline that turns raw Frequency Modulated Continuous-Wave (FMCW) radar ADC (Analogue-to-Digital Converter) data into detections. Lastly, to process these cluttered detections into target trajectories (tracks). This thesis focuses on the last sub-system. The aim is to provide resources of actionable information to parties that benefit from integrity in their airspace such as organizers of public events, airports etc. Furthermore, a secondary goal is to aid the Dopplium radar company in their efforts of improving their radar systems.

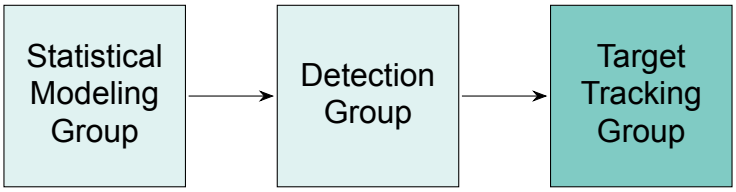


Figure 1.1: The overview of the project.

1.2. Problem Definition

To achieve the previously established goal, the problem has to be defined. As previously mentioned, the tracker combines measurements with prior knowledge and assumptions of the system to process the incoming detections. It has to take advantage of the fact that a drone, being a physical object, has to conform to physically plausible behaviour patterns.

The system input is received in 2D measurements; range and azimuth. Detections from multiple targets

can be received, and have to be assigned to their tracks.

Furthermore, the data-stream of the detection pipeline is non-ideal by nature. The tracker has to be robust to a sequence of missed detections through keeping track of the estimated trajectories of previously tracked drones, or false detections caused by environmental clutter or interfering sources.

The performance will be evaluated using the OSPA distance, common in multi-target tracking applications. After assessing the performance of each algorithm, an appropriate algorithm has to be selected.

A stringent limitation during the design process is the lack of radar recordings during a substantial portion of the project progress. Without real data, the system has to be validated using simulated detections. A detection simulator has to be developed to provide a foundation for further decision-making.

1.3. State-of-the-Art Analysis

Radar target tracking is an active research topic and the methods used for lower frequency radar can be directly applied to a higher frequency system. Most of the recent efforts relate to the classification of targets, for which multiple methods exist. Liu et al. [2] use passive bistatic radar to detect drones, while Jian et al. [3] use a phase-interferometric doppler radar. Ganeshan et al. [4] and Teh et al. [5] use kinematic properties of drone flight behaviour to classify targets and separate drones from other targets like birds. Some deep learning methods have also been developed. Like Haifawi et al. [6] which use the computer-vision YOLO framework to classify drones based on range-doppler plots. And Garcia et al. [7], that use a convolutional neural network (CNN) to detect and classify drones.

However, it was decided that classification is outside the scope of this project. The main objective of this subgroup was drone tracking. Dogru et al. [8] tracked drones using other drones equipped with mm-wave radar, with a tracker based on the Extended Kalman Filter (EKF). Hwang et al. [9] reconstructed drone flight paths based on visual data from a camera, which is impressive but less useful for this work. In general, most trackers are based on variants of the Kalman Filter, usually the EKF. Son et al. [10] employ a simple Kalman Filter using visual data, while others use more sophisticated variants. Like Barbary et al. [11], who use a Cubature Kalman-TBD-Multi-Bernoulli filter. Srihari et al. [12] use the Extended Kalman Filter, in combination with DBSCAN (Density-Based Spatial Clustering of Applications with Noise) to cluster nearby detections, and Global Nearest Neighbor (GNN) as a data associator. Solaiman et al. [13] use a similar tracking pipeline, while using a CNN for classification. Mohsen et al. [14] compared several different Kalman Filter variants, including the Unscented Kalman Filter (UKF) and Particle Filter (PF), and found that the selection of an optimal filter was highly dependent on the use case. At the high end of Kalman-based trackers, there are multimodel approaches, like those described by Lizzio et al. [15], that combine several different motion models to generate more accurate flight path predictions.

The most advanced methods combine detection and tracking to improve the performance of both. Liu et al. [16] use knowledge from the tracking pipeline to adjust detection thresholds. Amrouche et al. [17] use a Track-Before-Detect framework, increasing the performance of the detection subsystem by utilizing a tracking method on a set of frames, before declaring a detection.

1.4. Thesis Structure

This thesis comprises several parts. Firstly, the requirements for the system are stated in Chapter 2. Then, the methods for the field-measurements are discussed in detail in Chapter 3. The different algorithms will be explicated in Chapter 4, and finally, the results will be shared and discussed in Chapters 5 and 6. Furthermore, the ethical side of this project is explored in Appendix A. The compliance with the specified requirements is assessed in Appendix C, along with the source code (D) and extra plots (B).

2

Program of Requirements

2.1. Full System Requirements

Here, a brief overview of the functional requirements of the overall system is given. This includes the contributions of the full project group.

[F1.1] The system shall provide the estimated RCS of a drone as a function of the aspect angle.

[F1.2] The system shall provide a processing pipeline from raw ADC radar samples to detections.

[F1.3] The system must be able to track targets using a combination of measurements and an internal dynamical model.

2.2. Tracking Requirements

The focus of this thesis is target tracking, this section will cover different categories of requirements pertaining to this specific sub-system. Categories include: functional, performance, usability and security requirements

2.2.1. Functional requirements

The functional requirements specify the necessary features of the final tracking system.

[F2.1] The system must be able to track at least 2 targets simultaneously.

[F2.2] The system must be able to simulate drone flight paths with turns and changes in velocity.

[F2.3] The system must be able to simulate at least 2 targets.

[F2.4] The system must be able to reject false detection by implementing a data-association method.

[F2.5] The system must be able to maintain a track when a detection is missed.

[F2.6] The system must be implemented in Python.

2.2.2. Performance Requirements

The following requirements are specified regarding the performance of the final tracking system.

[P.1] The tracking performance shall be evaluated using the Optimal Sub-Pattern Assignment (OSPA) metric.

[P.2] The track average OSPA distance in a real-world test must be less than 5 m

[P.3] The track average OSPA distance in a linear simulation without measurement noise must be less than 0.05 m

[P.4] The track average OSPA distance in a multi-target simulation must be less than 3 m

[P.5] The tracking system shall use less than 10% of total recording time for its computational load.

2.2.3. Usability Requirements

An important part of this project is the value to the end user. It is in our interest to maximise usability through the following requirements:

[U.1] The system must visually present the tracking data.

[U.2] The system shall have a simple method that allows the user to tune relevant hyperparameters.

[U.3] The software must be developed modularly to allow for maintenance.

2.2.4. Security Requirements

This technology can be deployed for safety purposes, it is important to ensure the continued integrity of the system. Therefore, the current tracking information should be shielded from malicious parties that can adjust strategies following the compromised data.

[S.1] The system must run locally.

3

Modelling and Performance Evaluation

To facilitate the development and selection of a suitable algorithm, a modelling and performance evaluation framework must be established. This has been implemented in two parts; simulating drone flightpaths and their related detections, and a comprehensive and rigorous metric to quantitatively assess performance.

3.1. Field Measurement Setup and Characterization

In this section, the methods regarding the field-measurements are discussed.

3.1.1. Radar and GNSS Hardware

The field measurements are conducted in an open field with a certified drone operator. The radar used is the Dopplium TinyRad operating at 24 GHz. The radar uses 1 transmitter and 4 receivers, yielding a 4×1 array, capable of finding the Direction-of-Arrival (DOA) of the targets. The radar is positioned and angled towards open space, and the exact coordinates are noted down. The drones used are the DJI Neo and the DJI Air 3S, these drones have internal support of GPS tracking (actually, the drones use a multi-constellation GNSS system, including the European Galileo, the American GPS, and the Chinese BeiDou constellations. However, GNSS and GPS will be used interchangeably in this document), and produce a measurement every 100 ms. This GNSS track is used to generate a ground truth for the tracking.

3.1.2. GNSS Coordinate Projections

The GNSS measurements are converted into timestamped Cartesian data points. This conversion is done using the Pyproj [18] cartographic projections and coordinate transformations Python library. The *WSG84* datum is used in combination with a custom coordinate reference system (CRS) centred at the sensor position. The track is then rotated to align with the radar, based on a deduced reference coordinate directly in front of the radar, and validated with the field measurements. The radar is positioned at the origin (0,0) of every plot. The code can be found in Appendix D.5

3.1.3. Data Synchronization and Timing

The radar detections and the GNSS ground truth are not inherently synchronized in time. Furthermore, since the timestamps in the radar measurements show inconsistent delays, it was decided to manually synchronize the data based on an instance of recognizable behaviour in both sets. Furthermore, the period between the GNSS measurements and radar detections is not equal. It is necessary for the evaluation that the data points have the same timestamp. Therefore, the GNSS data is interpolated to provide a synchronous measurement, and to allow the performance to be assessed. The code for the interpolator can be found in Appendix D.6

3.2. System Model

The system consists of the drone itself, and the measurement platform; the radar. The drone is modelled as a single point in 2D Cartesian space. The algorithms itself also use an internal kinematic model, which is elaborated on further in Section 4.2. The simulation uses a random acceleration model to generate the ground-truth. A trajectory, or track, is then a temporal sequence of coordinates. The measurement noise is modelled as Gaussian White Noise (GWN) for mathematical and algorithmic convenience. The radar inherently provides range and azimuth-angle information, which result in distinct variances in both the azimuthal and radial direction for the measurement model.

3.2.1. Measurement Noise Characterization

To accurately simulate the radar measurements, the measurement noise has to be characterized. The measurement noise is modelled as GWN, with zero mean and a certain variance. This variance is important for the tuning of the algorithms. The distribution and variance of the real measurement error can be inferred from real detection data, using the GPS location of the drone as a reference. It should be noted that similar to the radar, the GPS data is imperfect. However, it does provide grounds of comparison to a theoretical value. To justify this decision, the variance of the GPS range is calculated in the same manner as outlined in the following analysis. The GPS variance in the range domain is equal to 0.000840, which is sufficiently small.

The theoretical values are calculated using the variance formula of a uniform distribution, similar to a quantization noise characterization. This is done because the radar provides detection information with predetermined range-, and azimuth-bin widths. σ_R^2 and σ_ϕ^2 denote the variances in the radial and azimuthal directions respectively, which are calculated using Equation 3.1.

$$\sigma_R^2 = \frac{\Delta R^2}{12}, \quad \sigma_\phi^2 = \frac{\Delta \phi^2}{12} \tag{3.1}$$

The range-bin width is determined by the range resolution (ΔR) of the radar system, which is determined by the bandwidth of the FMCW radar. The formula for the range resolution is given in Equation 3.2, along with the case-specific calculated value.

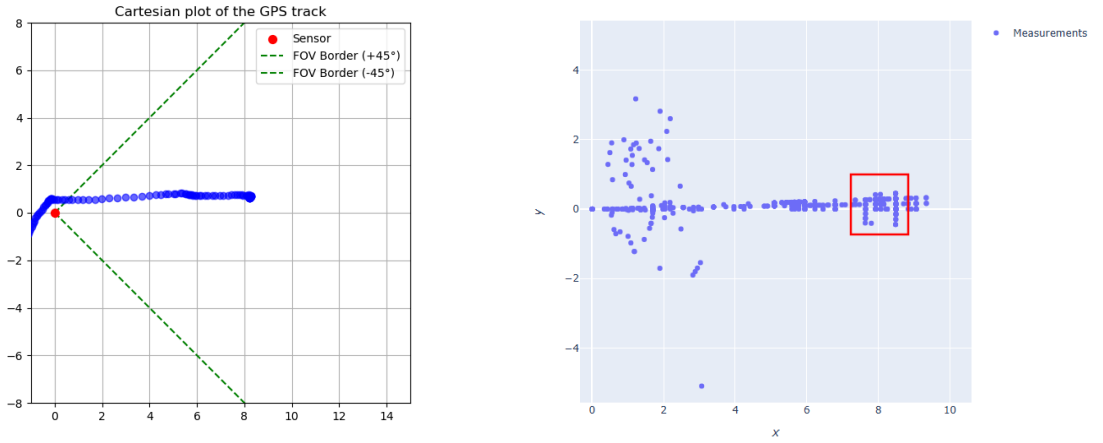
$$\Delta R = \frac{c_0}{2B} = 2.14 \text{ m} \tag{3.2}$$

Where the bandwidth B is equal to 70 MHz. The azimuth-bin width is received from the detection subgroup, and is equal to $\Delta\phi = 0.0838$ rad or $\Delta\phi = 4.81°$. The theoretical values for the variances are then calculated using Equation 3.1, and displayed in Table 3.1 below.

Table 3.1: Theoretical variance values for range and azimuth

Parameter	**Theoretical variance σ^2**
Range (R)	0.383 m^2
Azimuth (ϕ)	0.000585 rad^2

These theoretical values are compared to real data points collected from field-measurements. The recording used is one where the drone hovers 8.25m in front of the radar for a time duration of 6.8s. The GPS trajectory is displayed in figure 3.1a, along with the associated detections in figure 3.1b. The detections corresponding to the moment the drone is hovering at 8.25m are denoted by the red box.



(a) GPS track ending at the hovering position

(b) Detections at the hovering positions

Figure 3.1: The relevant scene in both GPS and detection domain

The data points corresponding to the relevant time-window are collected and the sample variance and mean are calculated. The code of the analysis can be found in Appendix D.8, including the list of collected data points. The calculated sample variances are presented below:

Table 3.2: Experimental variance values for range and azimuth

Parameter	**Experimental variance** σ^2
Range (R)	0.444 m^2
Azimuth (ϕ)	0.000720 rad^2

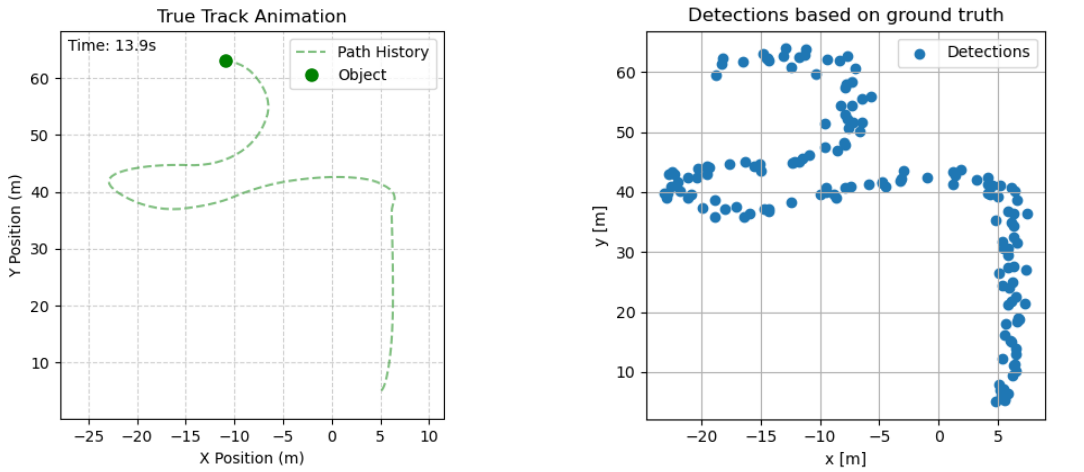
The theoretical and experimental values coincide, notable is that the azimuthal measurement error is very low in both cases. This did not align with initial expectations, as the angular resolution of a 4-element virtual array is unfavourable. However, the calculated variance characterizes the *precision*, which is different from the ability to separate different targets, which the resolution characterizes.

3.3. 2D Trajectory and Detection Simulator

Ground-truth paths are generated by a Finite State Machine (FSM) with various states corresponding to a certain manoeuvre, and different probabilities of starting these manoeuvres. The probabilities are empirically chosen. These states include: coasting with constant velocity, accelerating in a random direction (turning), braking and hovering. The position of the drone is propagated in time using standard kinematic equations, outlined in Equation 3.3 below.

$$x_{t+1} = x_t + v_x \Delta t + \frac{1}{2} a_x \Delta t^2 \tag{3.3}$$

The aim is to simulate drone-like behaviour. Additionally, a linear path can be simulated, which was used for validation of early iterations of the recursive filters. One example of a manoeuvring ground truth is displayed in figure B.3, with its associated detections in figure 3.2b. The code for the simulator can be found in Appendix D.4. The detections are generated according to the previously calculated radial and azimuthal variances.



(a) Ground-truth track simulation

(b) Detections associated with the ground-truth trajectory

Figure 3.2: Ground-truth path and its associated detections

Additionally, false alarms can be added to the detections, following a uniform distribution over the target area. This increases the complexity facing the tracker. An example of a simulation including false alarms can be seen in Figure 5.3. Furthermore, to make these detections compatible with the Stone Soup framework, they must be wrapped into `Detection` objects. The code for this is shared in Appendix D.4.1.

3.4. Evaluation Framework (OSPA)

The tracking performance will be assessed by comparing the ground-truth tracks to the estimated tracks. The metric used for this is the Optimal Sub-Pattern Assignment (OSPA) distance [m] [19], [20]. This metric is used widely in multi-target tracking related literature, because of its various advantages in this context. The OSPA distance measures the distance between two sets; $\mathfrak{X}_k = \{x_1, ..., x_m\}$ and $\mathfrak{Y}_k = \{y_1, ..., y_n\}$. The first containing the m ground truth track vectors, and the second containing the n tracker produced tracks at a time t_k. The implementation is based on the Schuhmacher *et al.* [19] article, and is realised using the Stone Soup Python library [21]. First, the distance between $x, y \in W$ where $W \subset \mathbb{R}^n$, with a cut off at $c > 0$, is defined.

$$d^{(c)}(x, y) := min(c, d(x, y)) \tag{3.4}$$

Let Π_n be the set of permutations of ground-truth to track combinations, and $1 < p < \infty$ the magnitude order to control the sensitivity to large localization errors. The OSPA distance $\bar{d}_p^{(c)}(\mathfrak{X}, \mathfrak{Y})$ is defined as:

$$\bar{d}_p^{(c)}(\mathfrak{X}, \mathfrak{Y}) := \left(\frac{1}{n} \left(\min_{\pi \in \Pi_n} \sum_{i=1}^{m} d^{(c)}\left(x_i, y_{\pi(i)}\right)^p + c^p(n - m) \right) \right)^{1/p} \tag{3.5}$$

A strength of this metric is its inherent support of multi-target tracking. Namely, the cardinality (the number of estimated tracks and ground truth tracks) of these sets does not have to be equal, the OSPA metric takes this cardinality error into account by assigning a penalty value (c) to the mismatch.

Moreover, the OSPA metric is mathematically rigorous, satisfying the identity, symmetry, and triangle inequality axioms [20]. Another advantage of the OSPA metric is that it returns a single scalar per time index, instead of an arbitrary set of traditional MoEs (measures of effectiveness). It is a clear cost function for any hyperparameter tuning. The OSPA distance itself can be tuned to more strictly penalize localization errors or cardinality errors through the parameters p and c respectively. In this thesis, the parameters $c = 10$, and $p = 1$ are used, which are default values. The source-code for this metric calculation can be found in Appendix D.3. In the following sections, the time-average of the OSPA

distance is used to validate the algorithms. For the simulations, a Monte Carlo method is applied to find the final mean OSPA from a sample size of 100 trials.

3.5. Track Visualization

The final output of the system is a sequence of estimated positions, this is plotted in a dynamic and interactive window. The tracks are displayed alongside the received detections, the ground truth path, and a visualization of the uncertainty in each direction (error covariance matrix P).

4

Tracking Algorithms

In this chapter the relevant tracking algorithms will be presented. Several different algorithms were developed, all based on variants of the Kalman Filter. The utility of these algorithms will be compared and quantified in chapter 5.

4.1. Notation and Definitions

As mentioned in chapter 3, drones are modelled as two-dimensional point targets, since the detections from the radar are represented in the 2D range-azimuth plane. The point target representation consists of position (x, y), velocity $(\dot{x}, \dot{y})$, and acceleration $(\ddot{x}, \ddot{y})$. Measurement of the drone is modelled in either Cartesian coordinates with measurement variances of σ_x^2 on the x axis and σ_y^2 on the y axis, or in polar coordinates with measurement variances of σ_r^2 on the range axis and σ_ϕ^2 on the azimuth axis. The system is described by the following state space equation:

$$\begin{cases} x_{t+1} = Fx_t + w_t \\ z_t = Hx_t + v_t \end{cases} \tag{4.1}$$

With state-vector x, transition matrix F, process noise vector w, observation z, observation matrix H and measurement noise vector v

4.2. Kinematic Models of a Drone

To be able to track a drone using a Kalman Filter, target motion must predicted. For that, an accurate kinematic model is required. For this purpose, two different models were used: a constant velocity (CV) and a constant acceleration (CA) model. These models are associated with their transition matrix F, which propagates the state-vector towards the next time step. The utility of these models will later be compared in chapter 5.

4.2.1. Constant Velocity model

In a 2D constant velocity (CV) model, the target is represented by a four-dimensional state-vector with position and velocity in the x and y axis, as seen in Equation 4.2.

$$x = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix} \tag{4.2}$$

Motion proceeds according to Equation 4.3, defining its transition matrix F.

$$x_t = Fx_{t-1} \Rightarrow \begin{bmatrix} x_t \\ \dot{x}_t \\ y_t \\ \dot{y}_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ \dot{x}_{t-1} \\ y_{t-1} \\ \dot{y}_{t-1} \end{bmatrix} \tag{4.3}$$

Where Δt represents the time difference between times t and $t+1$.

4.2.2. Constant Acceleration Model

The 2D constant acceleration (CA) model adds an acceleration component to the state vector, to more accurately model manoeuvres where a target is speeding up or slowing down. The state vector is now six dimensional, as can be seen in Equation 4.4.

$$x = \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \\ y \\ \dot{y} \\ \ddot{y} \end{bmatrix} \tag{4.4}$$

Motion proceeds according to the state transition matrix F, outlined in Equation 4.5 below.

$$x_t = Fx_{t-1} \Rightarrow \begin{bmatrix} x_t \\ \dot{x}_t \\ \ddot{x}_t \\ y_t \\ \dot{y}_t \\ \ddot{y}_t \end{bmatrix} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ \dot{x}_{t-1} \\ \ddot{x}_{t-1} \\ y_{t-1} \\ \dot{y}_{t-1} \\ \ddot{y}_{t-1} \end{bmatrix} \tag{4.5}$$

Where again, Δt represents the time difference between times t and $t+1$.

4.3. Observation Models

Observation of the drones is represented as a coordinate transformation from the state vector x to an observation vector z. This can be done in a Cartesian coordinate system, as well as in polar coordinates. In the case of Cartesian coordinates, the observation is a linear transformation that strictly observes the position values from the state vector. Since the transformation is linear, it can be represented by a matrix H, as seen in equations 4.6 and 4.7 for CV and CA models, respectively.

$$z_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ \dot{x}_t \\ y_t \\ \dot{y}_t \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} = Hx_t \tag{4.6}$$

$$z_t = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ \dot{x}_t \\ \ddot{x}_t \\ y_t \\ \dot{y}_t \\ \ddot{y}_t \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} = Hx_t \tag{4.7}$$

However, the actual observations coming from the radar are range-azimuth measurements, which lend themselves much better to the polar coordinate system. A transformation from the Cartesian to the polar coordinate system is non-linear, and so is represented by a function h, as seen in Equation 4.8, which works for both CV and CA models.

$$z_t = \begin{bmatrix} \sqrt{x_t^2 + y_t^2} \\ atan2(y_t, x_t) \end{bmatrix} = \begin{bmatrix} r \\ \phi \end{bmatrix} = h(x_t) \tag{4.8}$$

4.4. Definition of Matrices P, Q and R

The Kalman Filters make use of certain information in matrix form, the relevant matrices will be defined below.

4.4.1. Error Covariance Matrix P

The error covariance matrix P is a dynamic $N \times N$ matrix, where N is the length of the state-vector (i.e. $N = 6$ for the CA state-vector, and $N = 4$ for the CV state-vector). It is updated in every iteration of the recursive filter, as outlined in Sections 4.5.1 and 4.5.2. It is defined by:

$$P = \mathbb{E}[(x - \hat{x})(x - \hat{x})^T] \tag{4.9}$$

Where $\hat{x}$ is the estimated state, and x denotes the current truth-state. This matrix is used to calculate the Kalman gain matrix, i.e. it helps specify to what extent the tracker should rely on measurement and prediction. The matrix P can also be visualized as an error ellipse by calculating its eigenvalues and eigenvectors, which can provide readily available insight during the tracking process. Because we do not have a good estimate of the initial estimation error, or a strong need for very accurate values in the first few iterations, it is sufficient to initialize P as a large diagonal matrix to quickly adjust to the first few measurements.

4.4.2. Process Noise Covariance Matrix Q

The process noise matrix Q models the perturbations in the system model, also an $N \times N$ matrix. It is defined as the covariance matrix of the process noise vector w_t:

$$Q = \mathbb{E}[w_t w_t^T] \tag{4.10}$$

The 1-dimensional Q matrix is derived in [22, p. 270] for the CV model.

$$Q_{1D} = \begin{bmatrix} \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} \tilde{q} \tag{4.11}$$

Then, to model both the x-, and y-dimensions, this matrix is extended to a 2-dimensional version, as displayed in Equation 4.12

$$Q_{CV} = \begin{bmatrix} Q_{1D} & 0 \\ 0 & Q_{1D} \end{bmatrix} = \tilde{q} \begin{bmatrix} \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 & 0 & 0 \\ \frac{1}{2}\Delta t^2 & \Delta t & 0 & 0 \\ 0 & 0 & \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} \tag{4.12}$$

Where $\tilde{q}$ is proportional to the maximal expected velocity-jump as outlined in [22, p. 270]. The expected order of velocity change per timestep (or the standard deviation) follows Equation 4.13.

$$\sqrt{Q_{22}} = \sqrt{\tilde{q}\Delta t} \tag{4.13}$$

Using $\tilde{q} = 4$, similar to the method in [23], the velocity change is on the order of 0.03 m/s in a time-window of 0.2 ms, which roughly equates to 10G's of acceleration for a short time, which is a reasonable estimate for the targets under consideration. First Person View (FPV) racing drones can reach 18G's of acceleration, consumer drones like the DJI Air 3s can typically reach 5G's continuously.

For the CA case, the process is the same. The matrix is derived in [22, p. 271], and is presented in equations 4.14 and 4.15.

$$Q_{1D} = \begin{bmatrix} \frac{1}{20}\Delta t^5 & \frac{1}{8}\Delta t^4 & \frac{1}{6}\Delta t^3 \\ \frac{1}{8}\Delta t^4 & \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ \frac{1}{6}\Delta t^3 & \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} \tilde{q} \tag{4.14}$$

$$\Rightarrow Q_{CA} = \begin{bmatrix} \frac{1}{20}\Delta t^5 & \frac{1}{8}\Delta t^4 & \frac{1}{6}\Delta t^3 & 0 & 0 & 0 \\ \frac{1}{8}\Delta t^4 & \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ \frac{1}{6}\Delta t^3 & \frac{1}{2}\Delta t^2 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{20}\Delta t^5 & \frac{1}{8}\Delta t^4 & \frac{1}{6}\Delta t^3 \\ 0 & 0 & 0 & \frac{1}{8}\Delta t^4 & \frac{1}{3}\Delta t^3 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & \frac{1}{6}\Delta t^3 & \frac{1}{2}\Delta t^2 & \Delta t \end{bmatrix} \tilde{q} \tag{4.15}$$

4.4.3. Measurement Noise Covariance Matrix R

Similarly, the measurement noise covariance matrix R is defined as the covariance of the measurement noise vector v. The R matrix will specify to what extent the measurements can be trusted by the tracker.

$$\mathbb{E}[v_t v_t^T] \tag{4.16}$$

The measurement vector consists of a range and azimuth measurement, which means that R is a 2×2 matrix with the respective measurement variances on the trace, as shown in Equation 4.17.

$$R = \begin{bmatrix} \sigma_\phi^2 & 0 \\ 0 & \sigma_R^2 \end{bmatrix} \tag{4.17}$$

These variances have been characterized previously, in Section 3.2.1.

4.5. The Kalman filter

To track manoeuvring targets, predictions from the kinematic models have to be combined with noisy measurements. To do this, several variants of the Kalman Filter are used. The most basic of these is the standard Kalman Filter [24]. The Kalman Filter requires a transition matrix F and an observation matrix H, which are both linear for this basic variant. Uncertainty about the model and observations are represented by the process noise covariance Q and observation noise covariance R, respectively. Target tracking happens in two steps: prediction and updating. At each step, the filter's estimate for the system state x is updated, as well as the covariance P of this estimate. The code for each implementation can be found in Appendix D.2.

4.5.1. Prediction Step

During the prediction step, a new target state estimate $\hat{x}_t$ and covariance P_t is computed *a priori* from the estimate of the previous time-step based on the system model, without taking into account measurements.

$$\hat{x}_t^- = F\hat{x}_{t-1} \tag{4.18}$$

$$P_t^- = FP_{t-1}F^T + Q \tag{4.19}$$

Where $\hat{x}_t^-$ is the *a priori* estimate of the current state, whereas $\hat{x}_{t-1}$ is the *a posteriori* estimate of the previous state, with measurements taken into account.

4.5.2. Update Step

During the update step, the estimate from the prediction step is updated using observations, while also taking into account the inherent uncertainty of these observations represented by the measurement covariance R. The Kalman Filter balances between model predictions and observations using the Kalman gain K, which is optimal for linear systems.

$$K_t = P_t^- H^T (HP_t^- H^T + R)^{-1} \tag{4.20}$$

$$\hat{x}_t = \hat{x}_t^- + K_t(z_t - H\hat{x}_t^-) \tag{4.21}$$

$$P_t = (I - K_t H)P_t^- \tag{4.22}$$

The Kalman Filter yields the optimal filtering of measurements and model predictions for linear systems. However, its constraint of linearity makes it unsuitable for use with the non-linear polar coordinate measurement model. For this purpose, the Unbiased Converted Measurement Kalman Filter (UCMKF) was developed.

4.6. Unbiased Converted Measurement Kalman Filter

The Converted Measurement Kalman Filter (CMKF) operates on a linear system and observation model internally, but converts the polar measurements to Cartesian coordinates, supporting the polar measurements from the radar. However, the measurement noise will actually introduce a multiplicative bias into the converted measurements, which needs to be adjusted for. The measurement noise covariance R also needs to be recomputed for every new measurement, to account for the fact that polar noise in angle introduces a different error in the Cartesian plane depending on the polar range. To accomplish this, the *Unbiased* Converted Measurement Kalman Filter (UCMKF) is used [25].

4.6.1. Conversion Bias

If measurement noise is present in the system, the measured polar range and angle can be represented as shown in equations 4.23 and 4.24.

$$r_m = r + \upsilon_r \tag{4.23}$$

$$\phi_m = \phi + \upsilon_\phi \tag{4.24}$$

Where r_m and ϕ_m are the measured range and azimuth values, and υ_r and υ_ϕ the measurement noise values in range and azimuth, respectively. If υ_r and υ_ϕ are assumed to be distributed with symmetric PDFs, the expected values of the converted measurements in Cartesian are shown in equations

$$E[x_m] = \lambda_\phi r \cos(\phi) \tag{4.25}$$

$$E[y_m] = \lambda_\phi r \sin(\phi) \tag{4.26}$$

Where $\lambda_\phi = E[cos(\upsilon_\phi)] = e^{\frac{-\sigma_\phi^2}{2}}$. It can be seen that the measurement bias introduced by the noise is multiplicative in nature. De-biasing thus amounts to multiplying the converted measurements by the inverse of this bias.

4.6.2. Debiasing

To de-bias the converted measurements and yield a more accurate conversion under noise, the measurments are multiplied with the inverse of the conversian bias, according to equations 4.27, 4.28, and 4.29.

$$\lambda_\phi = e^{\frac{-\sigma_\phi^2}{2}} \tag{4.27}$$

$$x_m^u = \lambda_\phi^{-1} r_m \cos(\phi_m) \tag{4.28}$$

$$y_m^u = \lambda_\phi^{-1} r_m \sin(\phi_m) \tag{4.29}$$

Due to the polar noise, the Cartesian measurement noise covariance is different for different ranges and angles, so R is computed for each measurement according to equations 4.30, 4.31, 4.32, and 4.33.

$$R_{11} = (\lambda_\phi^{-2} - 2) r_m^2 \cos^2(\phi_m) + \frac{1}{2}(r_m^2 + \sigma_r^2)(1 + \lambda_\phi^4 \cos(2\phi_m)) \tag{4.30}$$

$$R_{22} = (\lambda_\phi^{-2} - 2) r_m^2 \sin^2(\phi_m) + \frac{1}{2}(r_m^2 + \sigma_r^2)(1 - \lambda_\phi^4 \cos(2\phi_m)) \tag{4.31}$$

$$R_{12} = R_{21} = (\lambda_\phi^{-2} - 2) r_m^2 \cos(\phi_m)\sin(\phi_m) + \frac{1}{2}(r_m^2 + \sigma_r^2)\lambda_\phi^4 \sin(2\phi_m) \tag{4.32}$$

$$R = \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} \tag{4.33}$$

After the de-biasing, the prediction and update steps proceed identically to the regular Kalman Filter. Due to the unbiased conversion, the UCMKF can deal with the non-linear polar measurements, where the KF cannot, while still using linear operations under the hood.

4.7. Extended Kalman Filter

Unlike the UCMKF, the Extended Kalman Filter (EKF) can work with the non-linear measurement model directly without conversion. It works by linearizing the model at the current estimate using Jacobians. Instead of matrices F and H, it uses functions $f(x)$ and $h(x)$.

4.7.1. Prediction Step

The prediction step works similarly to the regular Kalman Filter, except that the state transition matrix F is now computed as the Jacobian of the state transition *function* $f(x)$ at the current estimate.

$$F = \frac{df}{dx}|x = \hat{x}_{t-1} \tag{4.34}$$

$$\hat{x}_t^- = f(\hat{x}_{t-1}) \tag{4.35}$$

$$P_t = FP_{t-1}F^T + Q \tag{4.36}$$

Where $\frac{df}{dx}$ is the Jacobian of the system model $f(x)$. Both the CV and CA models are linear, so in this case the Jacobians are equal to the state transition matrix F as seen for the regular Kalman Filter.

4.7.2. Update Step

The update step follows the same principle, where the observation matrix H is now the Jacobian of the observation *function* $h(x)$ at the current estimate.

$$H = \frac{dh}{dx}|x = \hat{x}_t^- \tag{4.37}$$

$$K_t = P_t^- H^T (HP_t^- H^T + R)^{-1} \tag{4.38}$$

$$\hat{x}_t = \hat{x}_t^- + K_t(z_t - h(\hat{x}_t^-)) \tag{4.39}$$

$$P_t = (I - K_t H)P_t^- \tag{4.40}$$

Where $\frac{dh}{dx}$ is the Jacobian of the observation model $h(x)$. The CV and CA models will have different Jacobians for $h(x)$ due to their different state vectors. Equation 4.41 shows the Jacobian for the CV model while Equation 4.42 shows the Jacobian for the CA model.

$$\frac{dh}{dx} = \begin{bmatrix} \frac{-y}{x^2+y^2} & 0 & \frac{x}{x^2+y^2} & 0 \\ \frac{x}{\sqrt{x^2+y^2}} & 0 & \frac{y}{\sqrt{x^2+y^2}} & 0 \end{bmatrix} \tag{4.41}$$

$$\frac{dh}{dx} = \begin{bmatrix} \frac{-y}{x^2+y^2} & 0 & 0 & \frac{x}{x^2+y^2} & 0 & 0 \\ \frac{x}{\sqrt{x^2+y^2}} & 0 & 0 & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 \end{bmatrix} \tag{4.42}$$

4.8. Unscented Kalman Filter

While the EKF deals with non-linearities by linearizing at a single estimated point, the Unscented Kalman Filter (UKF) computes a set of carefully selected *sigma points* around the estimate, which are then passed trough the non-linear model function. State and covariance estimates are produced by computing a weighted sum of sigma points. Due to the fact that the UKF samples multiple points instead of just one, it can be more robust to high levels of non-linearity.

4.8.1. Sigma Point Calculation

In general, the UKF computes $2n + 1$ sigma points, where n is the number of dimensions of the state vector. In addition to this, it also required three parameters: α, β, and κ. α can be tuned to control the spread of sigma points around the mean, and is usually set to a relatively small value such as 0.1. β incorporates information about the distribution of the state. For a normal distribution, $\beta = 2$ is optimal. κ is a secondary tuning parameter and is usually set to zero [26]. The sigma points are calculated according to Equation 4.43. Each sigma point has corresponding weights for either mean or covariance calculations, according to Equation 4.44.

$$\begin{aligned} \chi_0 &= \hat{x}_{t-1} \\ \chi_i &= \hat{x}_{t-1} + (\sqrt{(n+\lambda)P_{t-1}})_i \qquad i = 1, \ldots, n \\ \chi_i &= \hat{x}_{t-1} - (\sqrt{(n+\lambda)P_{t-1}})_{i-n} \quad i = n+1, \ldots, 2n \end{aligned} \tag{4.43}$$

$$
\begin{aligned}
W_0^m &= \frac{\lambda}{n+\lambda} \\
W_0^c &= \frac{\lambda}{n+\lambda} + (1-\alpha^2+\beta) \\
W_i^m &= W_i^c = \frac{1}{2(n+\lambda)} \qquad i = n+1, \ldots, 2n
\end{aligned} \tag{4.44}
$$

4.8.2. Prediction Step

To predict the state of the system, the calculated sigma points are passed trough the model function $f(x)$, according to Equation 4.45, to generate a new set of transformed sigma points. These transformed points are then used to estimate the new mean (state) $\hat{x}_t^-$ and covariance P_t^-, according to Equations 4.46 and 4.47, respectively.

$$Y_i = f(\chi_i) \quad i = 0, \ldots, 2n \tag{4.45}$$

$$\hat{x}_t^- = \sum_{i=0}^{2n} W_i^m Y_i \tag{4.46}$$

$$P_t^- = \sum_{i=0}^{2n} W_i^c (Y_i - \hat{x}_t^-)(Y_i - \hat{x}_t^-)^T + Q \tag{4.47}$$

4.8.3. Update Step

The update step for the UKF works by a similar principle. The sigma points are recalculated based on the state estimate from the prediction step. These sigma points are then passed trough the observation function $h(x)$, and their weights used to calculate state and covariance updates.

$$Z_i = h(Y_i) \quad i = 0, \ldots, 2n \tag{4.48}$$

$$y_t = \sum_{i=0}^{2n} W_i^m Z_i \tag{4.49}$$

$$P_{yy} = \sum_{i=0}^{2n} W_i^c (Z_i - y_t)(Z_i - y_t)^T \tag{4.50}$$

$$P_{xy} = \sum_{i=0}^{2n} W_i^c (Y_i - y_t)(Y_i - y_t)^T \tag{4.51}$$

$$K_t = P_{xy} P_{yy}^{-1} \tag{4.52}$$

$$\hat{x}_t = \hat{x}_t^- + K_t(z_t - y_t) \tag{4.53}$$

$$P_t = P_t^- - K_t P_{yy} K_t^T \tag{4.54}$$

4.9. Data Association and Track Management

To finalize the tracking algorithms and yield a full tracking pipeline, the tracker needs to be able to assign the incoming measurements to their associated targets, and manage these targets in time. This enables support for clutter rejection (improbable target locations are not associated to a track, and therefore ignored), and multi-target tracking.

4.9.1. Data Association

The data association method used is the Global Nearest Neighbor algorithm (GNN) [27], with the Mahalanobis distance [28] measure used for both gating, and scoring of the distances.

The Mahalanobis distance is preferred over the simple Euclidean distance, because the Euclidean distance weighs the distance in the x-, and y-directions similarly, which is not preferred for a range-azimuth framework with different variances in each direction. The Mahalanobis distance is more lenient (reports lower distances than the Euclidean distance) towards dimensions with higher uncertainty (variance), by dividing by the variance in a particular direction (see Equation 4.56).

According to [27], GNN is the simplest and probably the most widely used data association method. The GNN algorithm is implemented using the Stone Soup Python library for simple integration. The code can be found in Appendix D.1. Firstly, the measurements are ellipsoidally gated around the new *a priori* estimated state $\hat{x}_t^-$ by placing a threshold on the Mahalanobis distance, calculated using the measurement residual covariance matrix S. S is defined according to equation 4.55.

$$S_t = H P_t^- H^T + R \tag{4.55}$$

Then, a gate $\sqrt{G}$ is defined such that association is allowed by the Mahalanobis distance d_M of the residual vector $\tilde{y}$.

$$d_M = \sqrt{\tilde{y}^T S^{-1} \tilde{y}} \leq \sqrt{G} \tag{4.56}$$

Where $\tilde{y}$ is the difference between the incoming measurement, and the predicted measurement. Assuming the non-linear case.

$$\tilde{y} = z_t - \hat{z}_t = z_t - h(\hat{x}_t^-) \tag{4.57}$$

This gate is specified as the `Association Distance` hyperparameter. In this thesis, a value of 4 is commonly used. The gate probability P_G can be calculated for a 2-dimensional system ([27, p. 337]).

$$P_G = 1 - e^{-G/2} \tag{4.58}$$

With an association distance of 4, meaning $G = 16$, the gate probability is calculated to be 99.97%. This is on the high end, however, justified by the high manoeuvrability of UAVs. To illustrate, agile drones can sustain a 10G turn, where a large gating threshold is needed. After gating, a cost matrix is generated based on a generalized statistical distance for the assignment of measurement j to track i.

$$C_{ij} = d_{ij} + \ln \|S_{ij}\| \tag{4.59}$$

Where d_{ij} is the Mahalanobis distance, and $\|S_{ij}\|$ is the determinant of the residual covariance matrix. Finally, the matrix $W = \{w_{ij}\}$ is found that minimizes the total cost J.

$$J = \sum_{i=0}^{n} \sum_{j=0}^{m} C_{ij} w_{ij} \tag{4.60}$$

With the constraints in equation 4.61 stating that a detection can be associated with a single track, and vice versa.

$$\sum_{i=0}^{m} w_{ij} = 1, \forall j, \quad \sum_{j=0}^{n} w_{ij} = 1, \forall i \tag{4.61}$$

4.9.2. Track Initiation

Non-associated points are stored in a separate Stone Soup `Track` object. After a certain number of points are sequentially associated to this particular track, a new target is assumed to have entered the scene, and a track is initiated. The number of points is defined by the `Initiation points` hyperparameter.

4.9.3. Track Deletion

Tracks are deleted once the trace of the error covariance P_t exceeds a threshold. Allowing a target to coast along through a sequence of missed detections. Gradually, the uncertainty will rise, and the target will be assumed to have left the scene. This threshold is defined by the `Deletion covariance` hyperparameter

5 Results

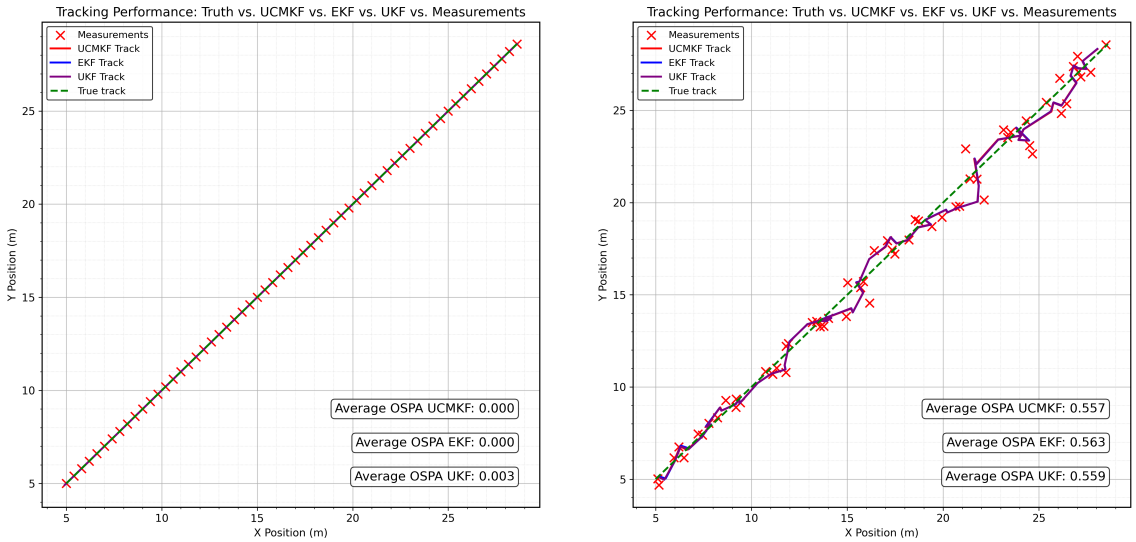
In this chapter, the results of the simulations and field measurements will be shared and evaluated. The raw data is processed into detection data-points by the detection sub-system.

5.1. Simulation Results

To verify the design of the tracking algorithms, simulations were conducted using the trajectory and detection simulator described in Chapter 3.

5.1.1. Linear Track Simulation

To verify the basic functionality of the Kalman Filters, simple linear tracks with constant velocity were simulated, along with detections according to the measurement variance in table 3.2. Example plots and OSPA scores can be found in Appendix B. Additionally, the design requirements specified an upper limit of 0.05 for the OSPA of a linear simulation without added noise, which was met. This simulation is displayed below in figures 5.1b and 5.1a, with zero variance, and the measurement variance from table 3.2 respectively.



(a) Tracks generated for a linear path with zero measurement error

(b) Tracks generated for a linear path

Figure 5.1: Tracks generated by different Kalman Filters for a target on a linear path

5.1.2. Single Target Simulation

To evaluate the performance of the algorithms on more complex tracks, the simulator from Chapter 3 was used to generate flights paths and detections for a single drone with random acceleration. A typical simulation along with the tracks generated by the different filters can be seen in figure 5.2. To quantify the relative performance, the OSPA scores of the different algorithms were averaged over a dataset of $N = 100$ random simulations. The mean OSPA scores can be seen in table 5.1.

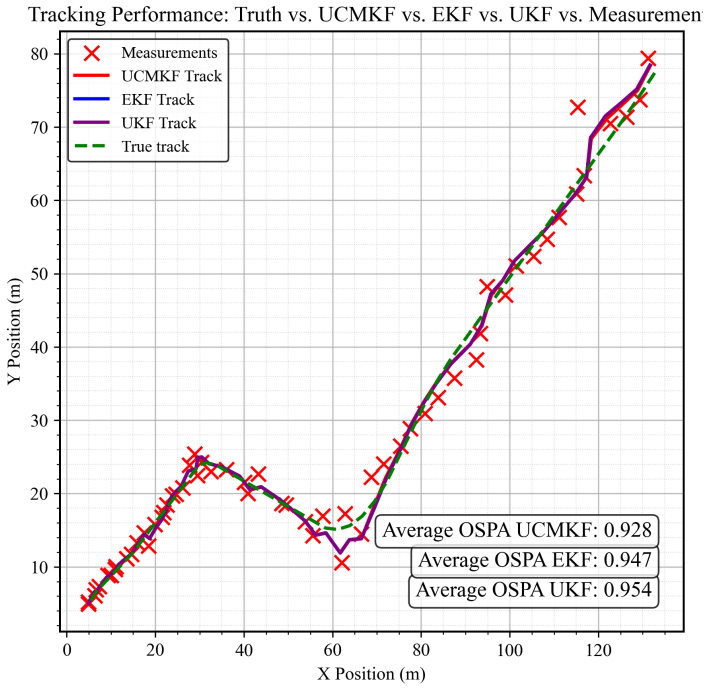


Figure 5.2: Tracks generated by different Kalman Filters for a single simulated target

Table 5.1: OSPA values for each tracking algorithm, for a single-target simulation

Tracking Algorithm	Kinematic Model	Mean OSPA [m]	Variance OSPA [m^2]
UCMKF	CV	1.42	0.065
	CA	1.59	0.082
EKF	CV	1.36	0.052
	CA	1.63	0.088
UKF	CV	1.33	0.044
	CA	1.63	0.118

As can be seen, the CV model performs slightly better compared to the CA model, while The different Kalman Filters perform roughly equal. Overall, these differences are non-significant, as the different OSPA means all lie within one standard deviation from each other.

5.1.3. Multi-Target Simulation

To verify multi-target tracking, flights paths from multiple targets were also simulated using the simulator from chapter 3, with starting parameters shown in table 5.2. A typical plot of this simulation, along with generated tracks, can be seen in figure 5.3. A dataset of $N = 100$ flights was simulated for multiple targets. The OSPA scores over all flights were averaged for all three kinds of Kalman Filters, as well as for the different kinematic models. Average OSPA scores can be seen in table 5.3. Typical OSPA scores over time for a single simulation of the UCMKF can be seen in figure 5.4. Additional plots for

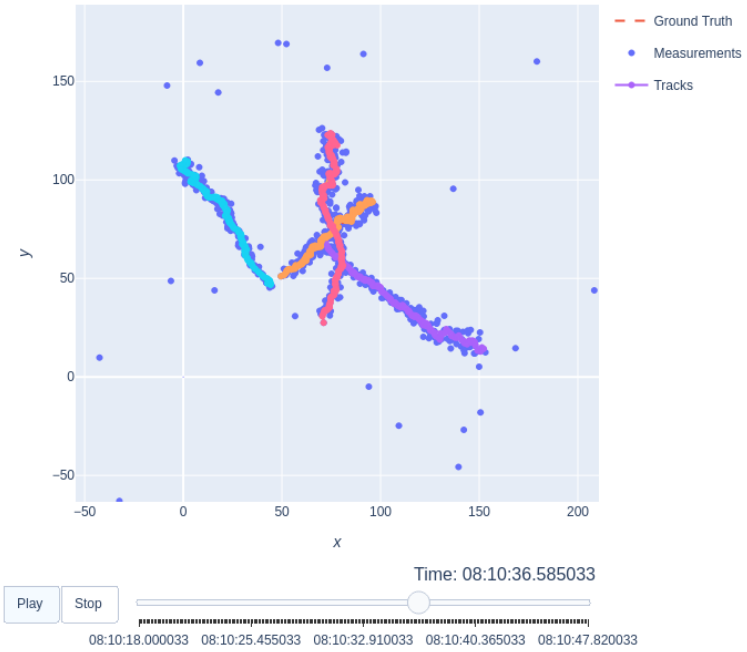


Figure 5.3: Tracking simulation with multiple targets

the EKF and UKF can be found in Appendix B.

An additional plot showcasing a multi-target simulation, with two targets crossing paths can be found in Figure B.1

# Drone	x (m)	y (m)	x velocity (m/s)	y velocity (m/s)	time delay (s)
1	50	50	5	5	0
2	45	45	-5	5	0
3	70	70	3	-4	10
4	70	30	3	5	30

Table 5.2: Initial simulation parameters for four drones

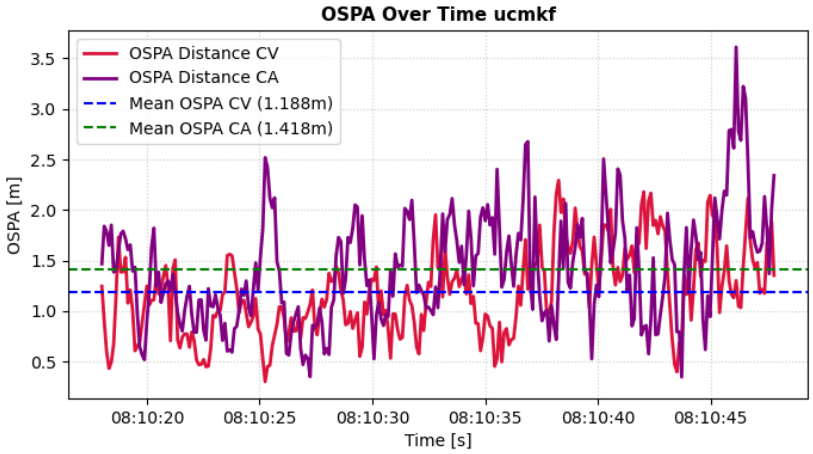


Figure 5.4: OSPA score over time for UCMKF on simulated data

Table 5.3: OSPA values for each tracking algorithm, in a multi-target simulation

Tracking Algorithm	Kinematic Model	Mean OSPA [m]	Variance OSPA [m^2]
UCMKF	CV	1.27	0.017
	CA	1.49	0.029
EKF	CV	1.25	0.014
	CA	1.43	0.019
UKF	CV	1.24	0.012
	CA	1.43	0.022

Similar to the single target case, the CV model performs slightly better compared to the CA model, while the different filters perform equally well.

5.2. Field-Measurement Results

The field measurements are conducted as described in Section 3.1. The first recording considered is that of a drone entering the field-of-view (FOV) of the radar and continuing a linear tangential path until the drone leaves the radar FOV, repeated back and forth. The hyper-parameters of the algorithm are given in the table below, based on manual tuning:

Table 5.4: Hyper-parameters for the field recordings

Hyper-parameter	Value
Association Distance	4
Initiation Points	15
Deletion Covariance	15

5.2.1. Tangential Flight

The recording consists of three tangential passes in total, which are displayed in figures 5.6, 5.7, and 5.8. When the drone leaves the radar view, tracking is no longer possible. However, this is not the result of a shortcoming of the tracker, therefore it was decided to omit them in calculation of the *Corrected OSPA* value, to provide a better indication of the OSPA distance during tracking.

The first result is obtained with the UKF, employing a CV model:

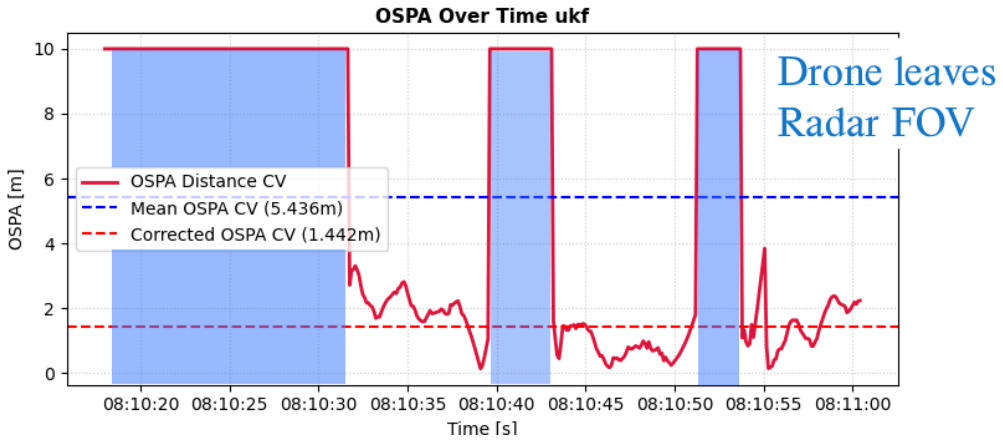
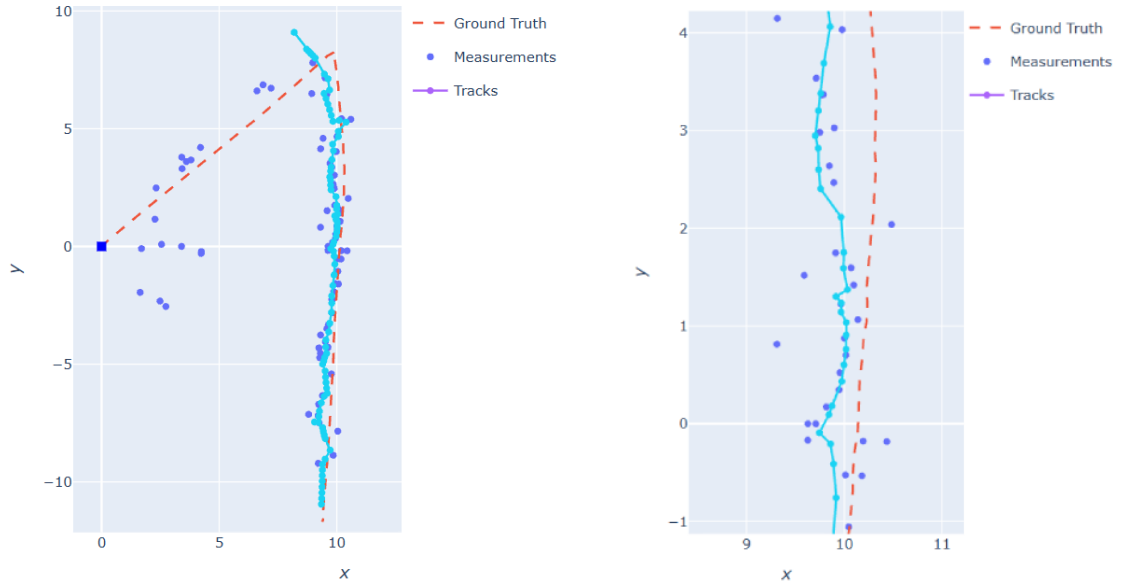


Figure 5.5: OSPA-time plot for the tangential flight recording using the UKF.

The first pass of the estimated track is displayed below, along with a zoomed in view of the path, please note that the radar is denoted by the blue square at the origin, and is facing to the right:

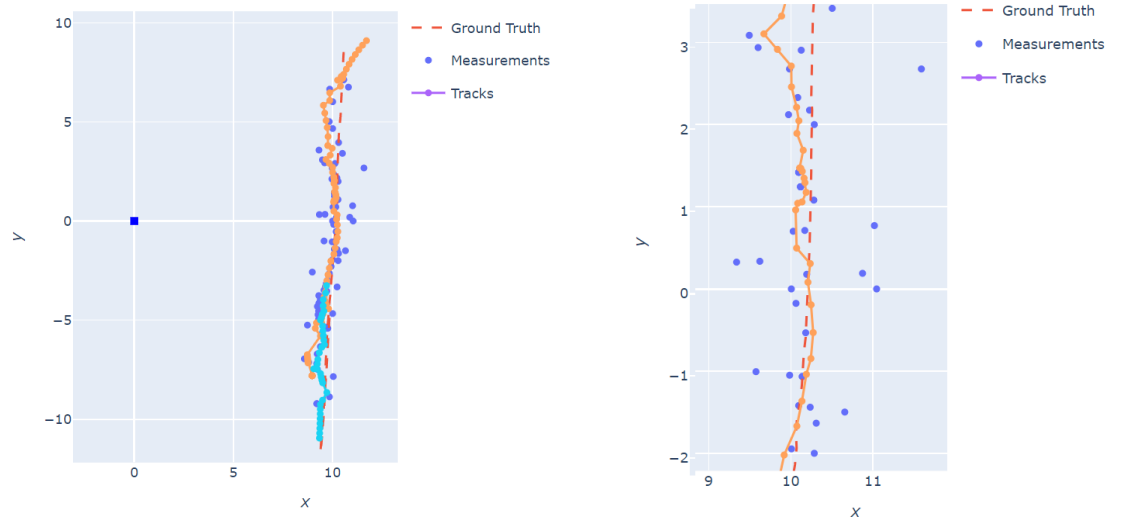


(a) UKF estimated track visualization

(b) Zoomed-in view of the UKF estimated track

Figure 5.6: Estimated UKF track of a tangential flight, the first pass.

The second pass (in orange going upward) is displayed below. Note that the older datapoints, tracks and ground truth are being omitted for clarity



(a) Second pass UKF estimated track visualization

(b) Zoomed-in view of the second pass of the UKF estimated track

Figure 5.7: Estimated UKF track of a tangential flight, the second pass.

The light-blue trail of the first pass is still visible in the lower region. Finally, the third pass (in purple going down):

(a) Third pass UKF estimated track visualization

(b) Zoomed-in view of the third pass of the UKF estimated track

Figure 5.8: Estimated UKF track of a tangential flight, the third pass.

A comparison between the CV and CA models is shown in figure 5.9.

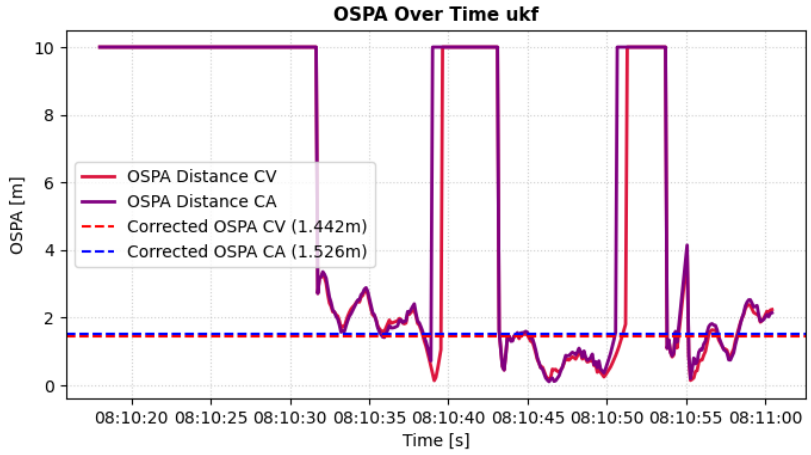


Figure 5.9: OSPA-time plot for the tangential flight recording using the UKF, a comparison between CV and CA models.

Again, the CV model performs better, showing a lower corrected OSPA distance, this is also emphasized by Table 5.5 below. The corrected OSPA distance is calculated for each algorithm, and displayed in the table below, the plots can be found in Appendix B.4.

Table 5.5: Corrected OSPA values for each tracking algorithm

Tracking Algorithm	Kinematic Model	Mean Corrected OSPA [m]
UCMKF	CV	1.59
	CA	1.52
EKF	CV	1.44
	CA	1.53
UKF	CV	1.44
	CA	1.53

The values are very close to each other, indicating very similar performance from all algorithms. However, for the UKF and EKF algorithms, the CV model scores 5.9% better than the CA models. The UCMKF employing the CA model performed slightly better than the CV variant. Overall, the differences are non-significant

5.2.2. Hovering Flight

The next flight features a linear path with pauses at certain ranges (8.35m, 16m and 30m), the same recording as used in Section 3.2.1. However, the radar loses the drone after around 15m. This is a product of the detection sub-system, and again, not a shortcoming of the tracking algorithm. It finds another target, which was identified as clutter, the tracker initiates a track at this alternative target. After the drone was drowned out, it enters the field of view again and a track is initiated. This is displayed in the OSPA graph shown in figure 5.10. The first blue region denotes the drone being behind the radar (not in view of the radar), and the second blue region denotes the drone being lost to the environment clutter by the detection system.

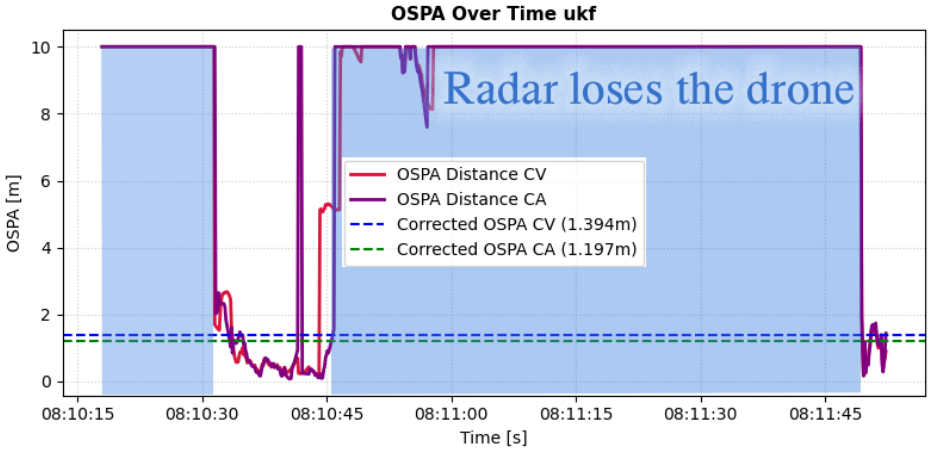
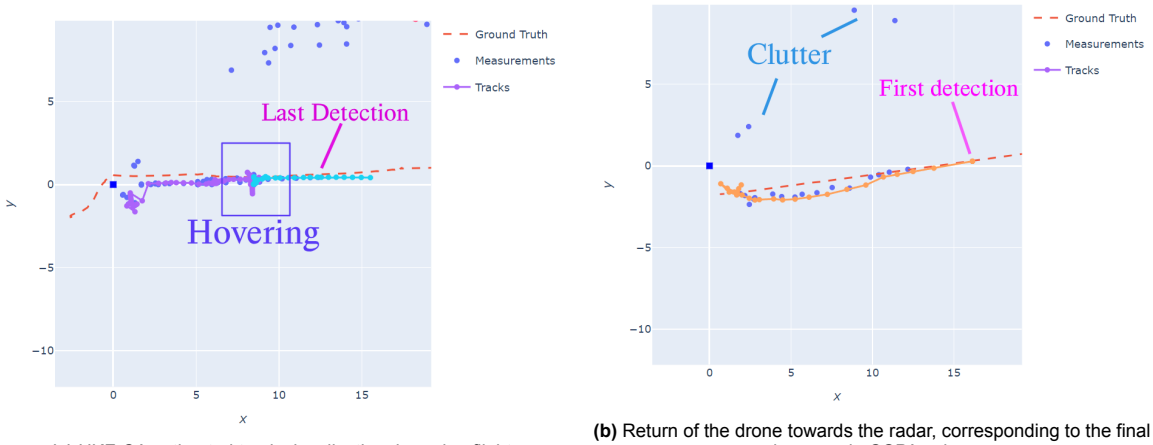


Figure 5.10: OSPA-time plot for the hovering flight recording using the UKF, a comparison between CV and CA models.

The drone is being detected in the non-coloured region. Again, the OSPA is adjusted to only take this window into account, to isolate the tracking performance. The exact timestamps of the drone leaving and returning are empirically, but precisely found using the interactive plot and an accurate time-axis. The OSPA distance is shown to decrease rapidly, after which it is cut off at 10, while the drone is not being detected (but the GPS ground truth is still active, causing the OSPA calculator to punish the cardinality error).

A large spike in OSPA can be found in the CA model trace, this is a result of the CA track management deleting the previous track and initiating a new track in its place, causing a brief moment of cardinality mismatch (2 tracks present simultaneously). This is unwanted behaviour. The CV model did manage to retain a single track. Another observation is that the clutter seemed coherent enough to warrant the initiation of a track. The visualizations of the track are displayed below:



(a) UKF-CA estimated track visualization, hovering flight

(b) Return of the drone towards the radar, corresponding to the final decrease in OSPA values

Figure 5.11: Estimated UKF-CA track of a hovering flight

In Figure 5.11a it is visible that the CA UKF has two separate tracks, with some overlap in time. In this particular recording, the CV model consistently avoids this issue, as displayed in Figure 5.12 below:

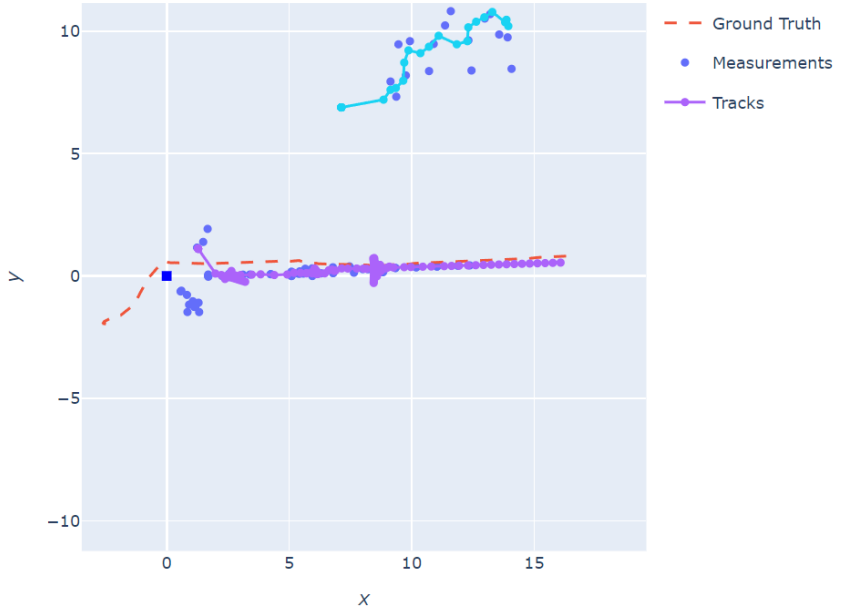


Figure 5.12: Estimated track of the hovering flight, using the CV model and the UKF

This should result in a lower mean corrected OSPA for the CV model. However, due to the temporal overlap of the clutter-track, the CV model shows a slightly higher corrected OSPA. The mean corrected OSPA value of each algorithm is calculated, and displayed in Table 5.6

Table 5.6: Corrected OSPA values for each tracking algorithm, hovering flight

Tracking Algorithm	Kinematic Model	Mean Corrected OSPA [m]
UCMKF	CV	1.39
	CA	1.69
EKF	CV	1.40
	CA	1.50
UKF	CV	1.39
	CA	1.20

5.3. Execution Time

The different algorithms are benchmarked using their execution time, which is potentially important for a real-time application. A 52 second recording with 10 samples/second is used for this benchmarking. The runtime of the recursive loop is measured and averaged over 500 runs for each algorithm separately. The results are displayed in Table 5.7:

Table 5.7: Computational Execution Runtime Performance Over 500 Iterations.

Tracking Algorithm	Kinematic Model	Mean Runtime (s)
UCMKF	CV	0.1418
	CA	0.1319
EKF	CV	0.1414
	CA	0.1359
UKF	CV	0.2539
	CA	0.2773

The execution time is at worst (with the UKF and the CA model) 0.53% of the recording time. The CV-EKF uses0.27% of the total recording time.

6

Discussion

6.1. Simulation discussion

Simulations on linear tracks verified the basic function of the algorithms, showing accurate tracking with OSPA performance satisfying design requirements (Chapter 2). Extending to random acceleration tracks for both single target- and multi target scenarios resulted in good OSPA performance as well, continuing to satisfy the requirements from Chapter 2. In simulation, the CV model outperformed the CA model to a small degree. Later trials on real flight data showed a smaller gap in performance between the CV and CA models, which indicates that the simulator disproportionally exaggerated the prominence of acceleration in drone flight paths. The extension to a multi target scenario, as well as the inclusion of clutter, verified the data association algorithms, showing overall similar OSPA performance compared to the single target case.

6.2. Field-measurement discussion

6.2.1. Tangential flight

The tangential flight (Section 5.2.1) showed that the tracking algorithms performed better than required in Chapter 2. Namely, the requirement stated that the (corrected) OSPA distance should be less than 5m, and all algorithms satisfy this condition. Furthermore, the CV model is found to outperform the CA model, with a slightly lower OSPA distance. The CA model was expected to perform better, due to its increased ability to model dynamic trajectory changes, and its capacity of modelling the underlying dynamics more accurately. However, the simplicity of the CV model allowed it to reach better performance. An unexpected fact is that the three different algorithms performed very similarly, which did not align with expectation. The UKF was expected to perform better because it provides a more accurate approximation of the underlying probability distribution, but we believe that the large amount of incoming measurements closed the performance gap, allowing the simpler algorithms to perform at their best. This recording showed that the data association rejected some clutter points further away, and correctly initialized the track on all three passes of the drone.

6.2.2. Hovering flight

This flight showed that the radar has difficulty picking up the signal of the drone past about 15m, again, this was not a shortcoming of the tracker. The signal was drowned out by a source of clutter, this makes it harder to use for validation of the tracking, but the corrected OSPA provides a representative figure. the corrected OSPA performance satisfies the design requirements specified earlier. Furthermore, the track initiation was demonstrated when the drone returned to the radar. Furthermore, this flight showed that the CV model is generally better at estimating the ground truth. However, due to some non-idealities of the field-recordings, the CA model can sometimes perform better.

6.3. Execution Times

The execution times are minimal compared to the recording time, 0.53% for the worst-case scenario, and 0.27% for the CV-EKF. The EKF and UCMKF have similar execution times, and the UKF has a 79% longer execution time. The difference between the CA and CV models did not entirely align with expectation. The CA model is expected to perform worse than the CV model because all computations must be executed with larger matrix dimensions, but this is not what the measurements showed. In both the EKF and UCMKF cases, the CV model performed worse than the CA model. From this, we may conclude that matrix multiplication is not the main contributor to the execution time. On the other hand, the UKF did show a slowing when employing the CA model. This may be attributed to the fact that the UKF simply has to do more computation, allowing matrix size to enter the forefront. Effectively, due to the relative insignificance of the computational times of the algorithms, there it is not the most critical factor to consider when selecting an algorithm.

7

Conclusions and Future Work

7.1. Conclusion

The goal of the overarching project was to develop an end-to-end drone detection pipeline to provide readily available actionable information about the surveyed airspace to the end-user, and this thesis has outlined the development of the tracking sub-system. The multi-target tracking system has been implemented and has been verified to satisfy performance requirements, as specified in the program of requirements (Section 2). Algorithms have been evaluated using simulated data and field recordings. Multiple tracking algorithms have been surveyed, in combination with the relevant kinematic models. While the UKF seemed promising, the three algorithms have been found to perform very similarly. Furthermore, The CV model slightly outperforms the CA model (Section 5.2.1), without a significant compromise in execution time (Section 5.3). We conclude that this application does not warrant the complexity of the UKF, as the EKF performs almost identically with a significantly shorter execution time (not critical but in light of similar performance). Thus, the selected algorithm is the EKF employing a CV kinematic model.

7.2. Future work

In future projects, this sub-system can be extended with a number of features and some aspects can be worked out further. Relevant tasks are enumerated below:

- A track-before-detect architecture, as mentioned in the SOTA analysis of Section 1.3, has the potential to greatly increase the detection accuracy by incorporating a tracking system in the detection sub-system. This is a promising next step for future research and development. However, it was found to be unfeasible in the allocated time-frame.
- A large unexplored facet of target tracking, is target classification, as mentioned in the SOTA (1.3). This may be done using machine learning (ML) methods such as convolutional neural networks (CNN) or decision trees. Classifying drones and evaluating threat-levels fall in line with the goal of providing actionable information to the responders. This classification can be used to distinguish birds from drones, or payload-carrying (high-risk) drones from consumer recreation (lower-risk) drones. This will inform the end user, and allow appropriate decision-making regarding the counter-measures taken.
- The system can be extended to work in real-time. This has not yet been achieved due to the complexity in the data acquisition from the radar. This will be necessary for a useful final product, as our goal is to provide actionable insight to the end-user in real-time scenarios, but real-time operation does not change the fundamentals of the techniques used.
- The Doppler velocity can be incorporated in the observation model because it provides valuable information about the velocity of the target. In a future project, it could prove to be useful to explore and evaluate the utility of the radial velocity in the Kalman filtering process.
- Several other promising tracking algorithms can be implemented and explored in the future. Such

algorithms include the Interacting Multiple Model (IMM) and Particle Filter. The IMM uses multiple different models according to the type of manoeuvre that the target is currently predicted to be performing, allowing the tracking system to subvert trade-off decisions, causing overall performance to improve. The PF is also worth exploring; it works similarly to the UKF, but it samples more points using Monte Carlo methods, getting a better picture of the underlying probability function. However, this thesis showed that the UKF performed similarly to the simpler models, so we do not expect a large performance gain from the PF.

- The simulator can be improved by adding missed detections, this has been sufficiently tested using the real radar recordings. However, the simulator lacks support of this feature. Additionally, the simulator can be adjusted to employ a Mixed-Motion model to more accurately model the dynamic behaviour of the UAV targets,

References

[1] Agnieszka A. Tubis et al. “Risks of Drone Use in Light of Literature Studies”. In: *Sensors* 24.4 (2024). ISSN: 1424-8220. DOI: 10.3390/s24041205. URL: https://www.mdpi.com/1424-8220/24/4/1205.

[2] Yuqi Liu et al. “Digital television based passive bistatic radar system for drone detection”. In: *2017 IEEE Radar Conference (RadarConf)*. 2017, pp. 1493–1497. DOI: 10.1109/RADAR.2017.7944443.

[3] Michael Jian, Zhenzhong Lu, and Victor C. Chen. “Drone detection and tracking based on phase-interferometric Doppler radar”. In: *2018 IEEE Radar Conference (RadarConf18)*. 2018, pp. 1146–1149. DOI: 10.1109/RADAR.2018.8378723.

[4] Rohit Ganeshan et al. “Classification of Bird and Drone Targets Based on Radar Data Augmented with Feature Engineering using Trajectory Kinematics Analysis and using Random Forest Algorithm”. In: *2025 IEEE Space, Aerospace and Defence Conference (SPACE)*. 2025, pp. 1–5. DOI: 10.1109/SPACE65882.2025.11170505.

[5] Bing Hong Teh, Samuel Dubos, and Jean-Marc Divanon. “Differentiation between drones and birds using kinematic analysis”. In: *2025 22nd European Radar Conference (EuRAD)*. 2025, pp. 153–156. DOI: 10.23919/EuRAD65285.2025.11234228.

[6] Hani Haifawi et al. “Drone Detection & Classification with Surveillance ‘Radar On-The-Move’ and YOLO”. In: *2023 IEEE Radar Conference (RadarConf23)*. 2023, pp. 1–6. DOI: 10.1109/RadarConf2351548.2023.10149588.

[7] Ann Janeth Garcia et al. “CNN-32DC: An improved radar-based drone recognition system based on Convolutional Neural Network”. In: *ICT Express* 8.4 (2022), pp. 606–610. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2022.04.012. URL: https://www.sciencedirect.com/science/article/pii/S2405959522000686.

[8] Sedat Dogru, Rui Baptista, and Lino Marques. “Tracking Drones with Drones Using Millimeter Wave Radar”. In: Jan. 2020, pp. 392–402. ISBN: 978-3-030-36149-5. DOI: 10.1007/978-3-030-36150-1_32.

[9] Seobin Hwang et al. *3D Trajectory Reconstruction of Drones using a Single Camera*. Sept. 2023. DOI: 10.48550/arXiv.2309.02801.

[10] Sohee Son et al. “Tiny Drone Tracking Framework Using Multiple Trackers and Kalman-based Predictor”. In: *Journal of Web Engineering* 20.8 (2021), pp. 2391–2412. DOI: 10.13052/jwe1540-9589.2088.

[11] Mohamed Barbary and Mohamed H. Abd ElAzeem. “Drones tracking sed on robust Cubature Kalman-TBD-Multi-Bernoulli filter”. In: *ISA Transactions* 114 (2021), pp. 277–290. ISSN: 0019-0578. DOI: https://doi.org/10.1016/j.isatra.2020.12.042. URL: https://www.sciencedirect.com/science/article/pii/S0019057820305619.

[12] Pathipati Srihari et al. “FMCW Radar-Based Detection and Tracking of Drones Using DBSCAN Clustering and Extended Kalman Filter for Anti-Drone Defense Systems”. In: *2024 IEEE 21st India Council International Conference (INDICON)*. 2024, pp. 1–6. DOI: 10.1109/INDICON63790.2024.10958522.

[13] Suhare Solaiman, Emad Alsuwat, and Rajwa Alharthi. “Simultaneous Tracking and Recognizing Drone Targets with Millimeter-Wave Radar and Convolutional Neural Network”. In: *Applied System Innovation* 6.4 (2023). ISSN: 2571-5577. URL: https://www.mdpi.com/2571-5577/6/4/68.

[14] Amani Mohsen et al. “Analysis of the Performance of Four Filter Types for Drone Tracking”. In: *2023 16th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. 2023, pp. 1–6. DOI: 10.1109/CISP-BMEI60920.2023.10373283.

[15] Fausto Francesco Lizzio et al. “Comparative Evaluation of Multiple-Model Kalman Filters for Highly Maneuvering UAV Tracking”. In: *Applied Sciences* 16.5 (2026). ISSN: 2076-3417. URL: https://www.mdpi.com/2076-3417/16/5/2377.

[16] Hongliang Liu et al. “Radar detection during tracking with constant track false alarm rate”. In: *2014 International Radar Conference*. 2014, pp. 1–5. DOI: 10.1109/RADAR.2014.7060437.

[17] Naima Amrouche, Ali Khenchaf, and Daoud Berkani. “Multiple target tracking using track before detect algorithm”. In: *2017 International Conference on Electromagnetics in Advanced Applications (ICEAA)*. 2017, pp. 692–695. DOI: 10.1109/ICEAA.2017.8065341.

[18] Jeff Whitaker et al. *jswhit/pyproj: version 2.0.2 release*. Version v2.0.2. Mar. 2019. DOI: 10.5281/zenodo.2592233. URL: https://doi.org/10.5281/zenodo.2592233.

[19] Dominic Schuhmacher, Ba-Tuong Vo, and Ba-Ngu Vo. “A Consistent Metric for Performance Evaluation of Multi-Object Filters”. In: *IEEE Transactions on Signal Processing* 56.8 (2008), pp. 3447–3457. DOI: 10.1109/TSP.2008.920469.

[20] Branko Ristic et al. “A Metric for Performance Evaluation of Multi-Target Tracking Algorithms”. In: *IEEE Transactions on Signal Processing* 59.7 (2011), pp. 3452–3457. DOI: 10.1109/TSP.2011.2140111.

[21] Steven Hiscocks, J. Barr, Ben D. Oakes, et al. “Stone Soup: No Longer Just an Appetiser”. In: *26th International Conference on Information Fusion (FUSION)*. IEEE. 2023.

[22] Yaakov bar-shalom, X.□Rong Li, and Thia Kirubarajan. *Estimation with Applications to Tracking and Navigation: Theory, Algorithms and Software*. Jan. 2004, pp. 270–272. ISBN: 047141655X. DOI: 10.1002/0471221279.ch11.

[23] Hongliang Liu et al. “Radar detection during tracking with constant track false alarm rate”. In: *2014 International Radar Conference*. 2014, pp. 1–5. DOI: 10.1109/RADAR.2014.7060437.

[24] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[25] Mo Longbin et al. “Unbiased converted measurements for tracking”. In: *IEEE Transactions on Aerospace and Electronic Systems* 34.3 (1998), pp. 1023–1027. DOI: 10.1109/7.705921.

[26] E.A. Wan and R. Van Der Merwe. “The unscented Kalman filter for nonlinear estimation”. In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*. 2000, pp. 153–158. DOI: 10.1109/ASSPCC.2000.882463.

[27] Samuel Blackman and Robert Popoli. *Design and Analysis of Modern Tracking Systems*. Norwood, MA: Artech House, 1999.

[28] P.C. Mahalanobis. “On the Generalised Distance in Statistics.” In: *Sankhya A* (1936).

[29] Michael Beard, Ba Tuong Vo, and Ba-Ngu Vo. “OSPA(2): Using the OSPA metric to evaluate multi-target tracking performance”. In: *2017 International Conference on Control, Automation and Information Sciences (ICCAIS)*. 2017, pp. 86–91. DOI: 10.1109/ICCAIS.2017.8217598.

[30] Stone Soup Contributors. *Stone Soup Framework » Metric Generators » Optimal Sub-Pattern Assignment (OSPA) Metric*. Version 0.1b7. Defence Science and Technology Laboratory (Dstl). 2021. URL: https://stonesoup.readthedocs.io/en/v0.1b7/stonesoup.metricgenerator.ospametric.html (visited on 05/12/2026).

[31] Pavlina Konstantinova, Alexander Udvarev, and Tzvetan Semerdjiev. “A study of a target tracking algorithm using global nearest neighbor approach”. In: *Proceedings of the 4th international conference conference on Computer systems and technologies e-Learning-CompSysTech’03*. 2003, pp. 290–295.

[32] Y. Kosuge. “Maneuvering target tracking using multiple maneuver model joint probabilistic data association”. In: *Proceedings IECON ’91: 1991 International Conference on Industrial Electronics, Control and Instrumentation*. 1991, 2059–2064 vol.3. DOI: 10.1109/IECON.1991.239024.

[33] T. Kirubarajan and Y. Bar-Shalom. “Kalman filter versus IMM estimator: when do we need the latter?” In: *IEEE Transactions on Aerospace and Electronic Systems* 39.4 (2003), pp. 1452–1457. DOI: 10.1109/TAES.2003.1261143.

[34] Qiang Li et al. “Kalman Filter and Its Application”. In: *International Conference on Intelligent Networks and Intelligent Systems* (2015).

[35] Bing Hong Teh, Samuel Dubos, and Jean-Marc Divanon. “Differentiation between drones and birds using kinematic analysis”. In: *2025 22nd European Radar Conference (EuRAD)*. 2025, pp. 153–156. DOI: 10.23919/EuRAD65285.2025.11234228.

[36] Eleanor Sheridan et al. “The effects of radar on avian behavior: Implications for wildlife management at airports”. In: *Applied Animal Behaviour Science* 171 (2015), pp. 241–252. ISSN: 0168-1591. DOI: https://doi.org/10.1016/j.applanim.2015.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0168159115001914.

[37] Manuel Treder et al. “Defined exposure of honey bee colonies to simulated radiofrequency electromagnetic fields (RF-EMF): Negative effects on the homing ability, but not on brood development or longevity”. In: *Science of The Total Environment* 896 (2023), p. 165211. ISSN: 0048-9697. DOI: https://doi.org/10.1016/j.scitotenv.2023.165211. URL: https://www.sciencedirect.com/science/article/pii/S0048969723038342.

A

Appendix - An Ethical Perspective

In this chapter, a focused ethical analysis of a drone-detection system and design process will be conducted. Our aim is to reflect on the ethical responsibilities of the engineer.

A.1. Societal impacts of the product

Radar technology with a focus on UAVs has inherent defence capability, especially in the recent global warfare and reconnaissance landscape. The primary affected party of any defence system is the defended group. This often translates to the general public, but the focus can also lie on politicians or important figures. For example, drone detection radar was widely deployed by the C-UAS battery during the recent NATO summit in The Hague. In this case, the deployment ensures safety of all parties involved, and prevented any potential diplomatic disasters. Furthermore, this technology is not limited to defence applications; airports need to confirm the integrity of their airspace, and sustainable wind farms can use this radar technology to monitor bird migration. However, technology that is applicable in warfare scenarios, even if only defensively, can spark conflicts and uphold arms-races. Improving radar systems can invoke another party to invest in more capable drone technologies. This inevitably pushes the boundary of drone capability, and may lead to more damage long-term (e.g. drone swarms, stealthy drones).

A.2. Ethical aspects of development

The engineering process of developing a radar system is ethically quite innocuous. It consisted of two field-measurement sessions, where a radar is positioned and the drone is flown in view of the radar for a short period (around 30 minutes of active radar measurement). Electromagnetic radiation emitted by the radar is non-ionizing, and at low intensity. However, it has been shown that birds alter their behaviour in response to continuously operating airport radars [36]. Additionally, EM radiation can affect the navigation of insects such as honeybees [37]. We argue that due to the non-harmful nature and short duration of the measurements, that taking field measurements is of no ethical concern to the environment.

A.3. Evaluation

Firstly, the application to warfare is an ethically heavy aspect of this technology, because it has direct effect on foreign affairs, conflicts, and relations. This does raise a large ethical concern. However, this detection radar technology can not be used to directly cause harm, as drones are inherently unmanned. We argue that the positive side (public safety, bird safety, and airport safety) outweighs the possibility of upholding a warfare mentality, implying the ethicality of the development of a drone detection system. A further positive outcome includes the safeguarding of privacy of the general public. Drones may be used by private individuals to invade the privacy of other citizens, a detection method allow the authorities to intervene.

A.4. Reflection on design choices

Similar systems have already been implemented, so our specific project does not cover new ethical ground, but there remains a responsibility. We could have better evaluated the ethical aspect of the implications before starting any project efforts. We have ensured the compliance with legislation of drone flight, through the mandatory certifications and confirmation of used airspace.

A.5. Forward-looking responsibility

If a similar detection system is developed in the future, some measures should be taken to ensure the ethicality of the process and final product. Firstly, the field-measurements have been argued to be non-invasive. However, it should be noted that the operators should comply with all legislation around drone operation, taking note of the type of airspace and mandatory certifications. Secondly, the final product should not be targeted toward offensive purposes. Offensive techniques such as frequency-agility should be avoided.

B

Appendix - Additional Plots

Figure B.1 displays a linear crossing path multi-target simulation, to illustrate the robustness of the tracker.

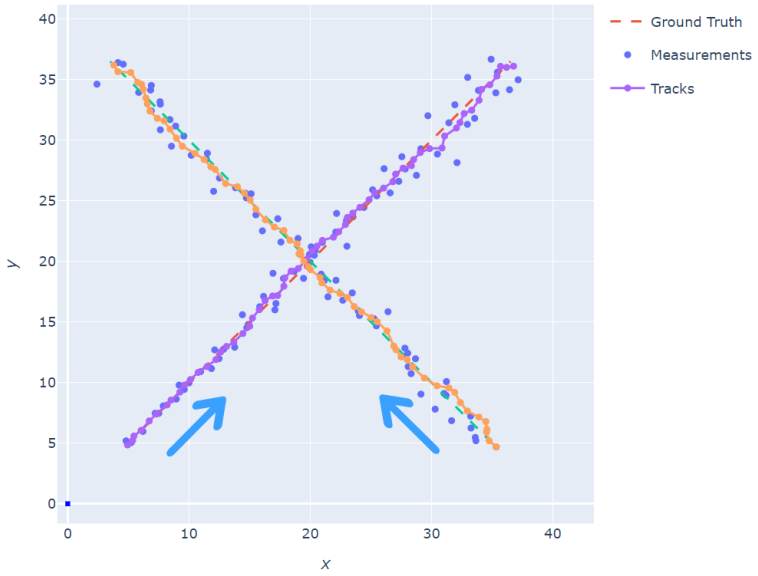


Figure B.1: Crossing paths, multi-target simulation

B.1. Linear Single-Target OSPA

An additional OSPA-time plot for a linear track with zero measurement variance, tracked by the UKF, can be seen in figure B.2.

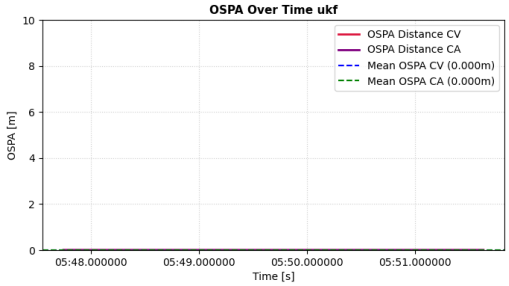
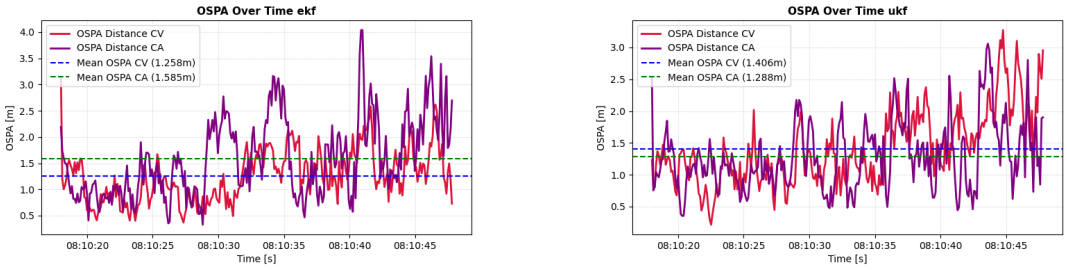


Figure B.2: OSPA-time plot for the UKF on a linear track with zero measurment variance

B.2. Multi-Target OSPA

Figure B.3 displays additional OSPA-time plots for the multi-target simulations.



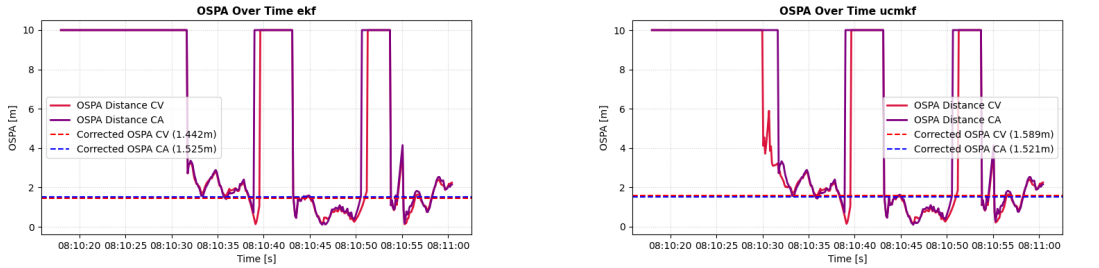
(a) EKF OSPA-time plot for the multi-target simulation

(b) UKF OSPA-time plot for the multi-target simulation

Figure B.3: Additional OSPA results for the remaining algorithms, for the simulation

B.2.1. Tangential Flight - UCMKF & EKF

Figure B.4 displays the additional OSPA-time plots for the tangential linear motion recordings, where the drone leaves and enters the radar FOV.



(a) EKF OSPA-time plot for the horizontal motion recording

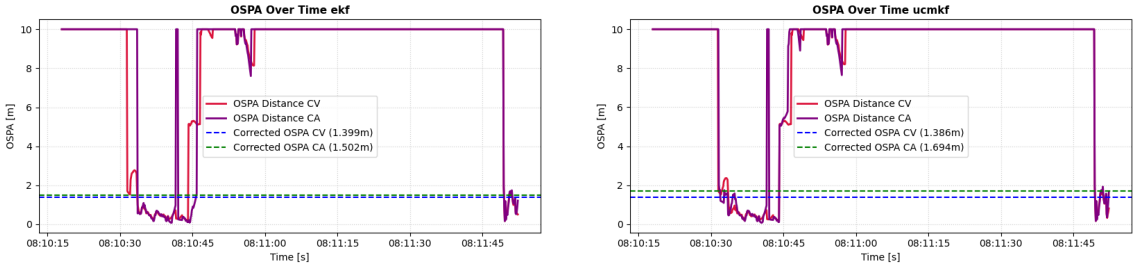
(b) UCMKF OSPA-time plot for the tangential motion recording

Figure B.4: Additional OSPA results for the remaining algorithms, for the tangential motion recording

As is visible in the figures above, the OSPA plots are very similar.

B.2.2. Hovering Flight - UCMKF & EKF

Figure B.5 shows the additional plots for the EKF and the UCMKF.



(a) EKF OSPA-time plot for the hovering recording

(b) UCMKF OSPA-time plot for the hovering recording

Figure B.5: Additional OSPA results for the remaining algorithms, for the hovering recording

C

Appendix - Requirement fulfillment

In this Appendix, the extent of the compliance with the tracking-specific requirements will be assessed, as specified in Chapter 2.

C.1. Functional Requirements

[F2.1]: The system can track an unlimited number of targets. Four targets are shown to be supported in Section 5.1.3. This satisfied the minimum of 2.

[F2.2]: The system is able to simulate turns and changes in velocity, one of such generated flight paths is displayed in Figure 3.2

[F2.3]: The simulation is able to simulate an unlimited number of targets. Again, Section 5.1.3 shows that 4 targets are being simulated. Which exceeds the minimum of 2.

[F2.4]: A data association is method is implemented. The following figure (Figure C.1) shows the data association algorithm rejecting false detections. The recording used, is the hovering flight on its return path to the radar:

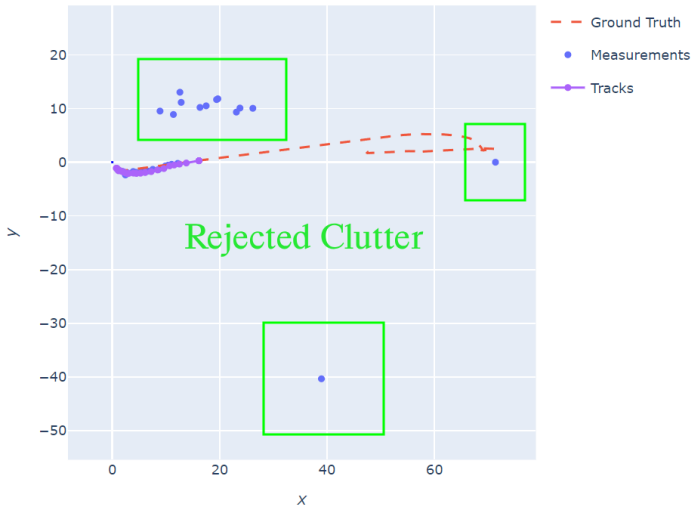


Figure C.1: Return path of the hovering flight, along with annotated rejected clutter

This shows the compliance to the above requirement.

[F2.5]: The hovering recording also showed the maintaining of a track through missed detections. Once the drone is no longer detected, the tracker still propagates the drone position in the right direction:

Figure C.2: Drone leaving detection range, while tracker still maintains the track. Annotated with the last detection and track maintenance

This shows the compliance to the track maintenance requirement.

[F2.6]: The system is fully implemented in Python, refer to Appendix D for the code.

C.2. Performance Requirements

[P.1]: The tracking performance is evaluated using the OSPA metric, satisfying the requirement.

[P.2]: The track average OSPA distance in the considered flights are presented in Sections 5.2. The chosen EKF-CV (and all other algorithms) have been shown to have a OSPA distance lower than 2 for both flights, which is lower than the maximum of 5. This satisfies the above requirement.

[P.3]: The track average OSPA in a linear simulation has been shown to be 0.0 for the CV-EKF (refer to Figure 5.1a), which is less than the minimum of 0.05. Thereby, satisfying the performance requirement.

[P.4]: The track average OSPA distances in a multi-target simulation can be found in Section 5.1.3 (Table 5.3). The CV-EKF shows a OSPA distance of 1.25 m, which satisfies the minimum of 3.

[P.5]: The CV-EKF algorithm uses 0.27% of the total recording time for its computational load. This is considerably less than the maximum of 10% specified by the requirement. This performance requirement is satisfied.

C.3. Usability Requirements

[U.1]: The system can visually present the tracking data, using an interactive plot with an accurate time axis. The plot can animate the tracking behaviour with the correct frame rate.

[U.2]: The pipeline allows the user to easily change the hyperparameter in the main function, satisfying the requirement

[U.3]: The relevant modular functions are shared in Appendix D.

C.4. Security Requirements

[S.1]: The system runs on a local machine, minimizing any security concerns.

D

Appendix - Source Code

In this Appendix, the source code for the full pipeline, and different algorithms is shared.

D.1. The Pipeline & Track Management

Here, the main pipeline, and main file is shared. The pipeline is relatively extensive because we wanted to be able to select the different Kalman filters on the fly. Additionally, the initiator and deleter are initialized in the file below.

```
# External Imports
import numpy as np
from datetime import timedelta
from timeit import default_timer as timer

# Internal Packages

from Evaluation_Metrics import ospa_stonesoup
from Kalman_Filters import UCMKalmanFilter, ExtendedKalmanFilter,
    UnscentedKalmanFilter
from Track_Simulation.TrackSimulator import simulateRandomAccelTrackPolar
from Utils.DecodeGPS import gps_to_ground_truth, interpolate_ground_truth
from Utils.Wrapper_Functions import (
    UCMKFPredictor,
    UCMKFUpdater,
    EKFPredictor,
    EKFUpdater,
    UKFPredictor,
    UKFUpdater,
)
from Utils import ReadDetections, ReadAndClusterDetections

# Stonesoup imports
from stonesoup.models.measurement.nonlinear import CartesianToBearingRange
from stonesoup.types.state import GaussianState
from stonesoup.dataassociator.neighbour import GlobalNearestNeighbour
from stonesoup.deleter.error import CovarianceBasedDeleter
from stonesoup.initiator.simple import MultiMeasurementInitiator
from stonesoup.measures import Mahalanobis
from stonesoup.hypothesiser.distance import DistanceHypothesiser

from Utils.Wrapper_Functions.StoneSoup_Wrappers import (
```

```
    SimulatorPolarMultitarget_stonesoup,
)


def get_detections(
    model,
    measurement_model,
    var_r,
    var_phi,
    start_time,
    dt,
    recording_path,
    gps_path,
    gps_offset_delay,
    drones_params,
    num_datapoints,
    simulation=False,
):
    if simulation:
        shared_config = {
            "sim_function": simulateRandomAccelTrackPolar,
            "start_time": start_time,
            "measurement_model": measurement_model,
            "model": model,
            "num_datapoints": num_datapoints,
            "dt": dt,
            "sigma_r": np.sqrt(var_r),
            "sigma_phi": np.sqrt(var_phi),
            "add_clutter": True,
        }

        detections, ground_truth = SimulatorPolarMultitarget_stonesoup(
            **shared_config, drone_configs=drones_params
        )

        return detections, ground_truth
    else:
        # Get the detections from Abdullahs group
        detections = ReadDetections(
            filepath=recording_path,
            measurement_model=measurement_model,
            dt=dt,
            start_time=start_time,
        )

        ground_truth = gps_to_ground_truth(
            filepath=gps_path,
            model=model,
            start_time=start_time,
            gps_offset_delay=gps_offset_delay,
        )

        return detections, ground_truth


def generate_config(
```

```
    var_r,
    var_phi,
    start_time,
    dt,
    model,
    filter,
    drones_params,
    num_datapoints,
    recording_path,
    gps_path,
    gps_offset_delay=0,
    association_distance=4,
    deletion_covariance=15,
    initiation_points=15,
    simulation=False,
):
    # variances in measurement dimensions.
    range_sigma = np.sqrt(var_r)
    azimuth_sigma = np.sqrt(var_phi)

    # Kalman Filter tuning
    UCMKF_Q = 1
    UCMKF_P0 = 1
    EKF_Q = 1
    EKF_R = 1
    EKF_P0 = 1
    UKF_Q = 1
    UKF_R = 1
    UKF_P0 = 1
    alpha = 0.1

    # Initialize the functions and matrices for the Kalman filter.
    F_generator = lambda dt: (
        np.array(
            [
                [1, dt, 0, 0], # xip
                [0, 1, 0, 0], # vx
                [0, 0, 1, dt], # y
                [0, 0, 0, 1], # vy
            ]
        )
        if model == "cv"
        else np.array(
            [
                [1, dt, 0.5 * dt**2, 0, 0, 0], # x
                [0, 1, dt, 0, 0, 0], # vx
                [0, 0, 1, 0, 0, 0], # ax
                [0, 0, 0, 1, dt, 0.5 * dt**2], # y
                [0, 0, 0, 0, 1, dt], # vy
                [0, 0, 0, 0, 0, 1], # ay
            ]
        )
    )
    F = F_generator(dt) if filter == "ucmkf" else lambda x: F_generator(dt)

    f = lambda x: np.dot(F_generator(dt), x)
```

```
    h = (
        np.array([[1, 0, 0, 0], [0, 0, 1, 0]])
        if filter == "ucmkf" and model == "cv"
        else (
            np.array([[1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0]])
            if filter == "ucmkf"
            else lambda x: (
                np.array([np.arctan2(x[2], x[0]), np.sqrt(x[0] ** 2 + x[2] ** 2)])
                if model == "cv"
                else np.array([np.arctan2(x[3], x[0]), np.sqrt(x[0] ** 2 + x[3] ** 2)])
            )
        )
    )

    H = lambda x: (
        np.array(
            [
                [
                    (-x[2]) / (1e-9 + x[0] ** 2 + x[2] ** 2),
                    0,
                    x[0] / (1e-9 + x[0] ** 2 + x[2] ** 2),
                    0,
                ],
                [
                    x[0] / np.sqrt(1e-9 + x[0] ** 2 + x[2] ** 2),
                    0,
                    x[2] / np.sqrt(1e-9 + x[0] ** 2 + x[2] ** 2),
                    0,
                ],
            ]
        )
        if model == "cv"
        else np.array(
            [
                [
                    (-x[3]) / (x[0] ** 2 + x[3] ** 2),
                    0,
                    0,
                    x[0] / (x[0] ** 2 + x[3] ** 2),
                    0,
                    0,
                ],
                [
                    x[0] / np.sqrt(x[0] ** 2 + x[3] ** 2),
                    0,
                    0,
                    x[3] / np.sqrt(x[0] ** 2 + x[3] ** 2),
                    0,
                    0,
                ],
            ]
        )
    )

    # Process noise matrix.
```

```
    # IMPORTANT: the process noise is dependent on dt
    var_a = 2
    Q1D_generator = lambda dt: (
        var_a
        * np.array(
            [
                [(dt**3 / 3), (dt**2 / 2)],
                [(dt**2 / 2), dt],
            ]
        )
        if model == "cv"
        else var_a
        * np.array(
            [
                [(dt**5 / 20), (dt**4 / 8), (dt**3 / 6)],
                [(dt**4 / 8), (dt**3 / 3), (dt**2 / 2)],
                [(dt**3 / 6), (dt**2 / 2), dt],
            ]
        )
    )
    Q_generator = lambda dt: (
        np.block(
            [
                [Q1D_generator(dt), np.zeros((2, 2))],
                [np.zeros((2, 2)), Q1D_generator(dt)],
            ]
        )
        if model == "cv"
        else np.block(
            [
                [Q1D_generator(dt), np.zeros((3, 3))],
                [np.zeros((3, 3)), Q1D_generator(dt)],
            ]
        )
    )

    Q = Q_generator(dt)

    x0 = (
        np.array([[1], [5], [1], [5]])
        if model == "cv"
        else np.array([[1], [5], [0], [1], [5], [0]])
    )

    # Starting error covariance (should be on the higher side to quickly settle in
        towards the correct values. i.e high uncertainty to start with :))
    P0 = np.eye(4) if model == "cv" else np.eye(6)

    # Initialize measurement error matrix.
    R = np.array([[var_phi, 0], [0, var_r]])

    # Define measurement_model
    measurement_model = (
        CartesianToBearingRange(ndim_state=4, mapping=(0, 2), noise_covar=R)
        if model == "cv"
        else CartesianToBearingRange(ndim_state=6, mapping=(0, 3), noise_covar=R)
```

```
    )

    detections, ground_truth = get_detections(
        model,
        measurement_model,
        var_r,
        var_phi,
        start_time,
        dt,
        recording_path,
        gps_path,
        gps_offset_delay,
        drones_params,
        num_datapoints,
        simulation=simulation,
    )

    # ___Initialize the filter____
    kf = (
        UCMKalmanFilter(
            F=F,
            H=h,
            Q=UCMKF_Q * Q,
            P0=UCMKF_P0 * P0,
            sigma_r=range_sigma,
            sigma_phi=azimuth_sigma,
            x0=x0,
        )
        if filter == "ucmkf"
        else (
            ExtendedKalmanFilter(
                f=f, h=h, F=F, H=H, Q=EKF_Q * Q, R=EKF_R * R, P0=EKF_P0 * P0, x0=x0
            )
            if filter == "ekf"
            else UnscentedKalmanFilter(
                f=f,
                h=h,
                Q=UKF_Q * Q,
                R=UKF_R * R,
                P0=UKF_P0 * P0,
                alpha=alpha,
                beta=2,
                kappa=0,
                x0=x0,
            )
        )
    )

    predictor = (
        UCMKFPredictor(ucmkf=kf)
        if filter == "ucmkf"
        else EKFPredictor(ekf=kf) if filter == "ekf" else UKFPredictor(ukf=kf)
    )
    updater = (
        UCMKFUpdater(ucmkf=kf)
        if filter == "ucmkf"
```

```
        else EKFUpdater(ekf=kf) if filter == "ekf" else UKFUpdater(ukf=kf)
    )

    prior = GaussianState(
        state_vector=x0,
        covar=kf.P,
        timestamp=start_time,
    )

    hypothesiser = DistanceHypothesiser(
        predictor, updater, measure=Mahalanobis(), missed_distance=
            association_distance
    )

    data_associator = GlobalNearestNeighbour(hypothesiser)

    deleter = CovarianceBasedDeleter(covar_trace_thresh=deletion_covariance)

    initiator = MultiMeasurementInitiator(
        prior_state=prior,
        deleter=deleter,
        data_associator=data_associator,
        updater=updater,
        min_points=initiation_points,
    )

    return (
        detections,
        dt,
        data_associator,
        updater,
        deleter,
        initiator,
        ground_truth,
    )


def run_algorithm(
    model,
    detections,
    start_time,
    dt,
    data_associator,
    updater,
    deleter,
    initiator,
    ground_truth,
    simulation=False,
    plot=True,
):
    tracks, all_tracks = set(), set()
    timesteps = []
    start = timer()
    for n, measurements in enumerate(detections):
        timestamp = start_time + timedelta(seconds=dt * n)
        timesteps.append(timestamp)
```

```
366
367         hypotheses = data_associator.associate(tracks, measurements, timestamp)
368         associated_measurements = set()
369
370         for track in tracks:
371             hypothesis = hypotheses[track]
372
373             if hypothesis.measurement:
374                 post = updater.update(hypothesis)
375                 track.append(post)
376
377                 associated_measurements.add(hypothesis.measurement)
378
379             else:
380                 track.append(hypothesis.prediction)
381
382         tracks -= deleter.delete_tracks(tracks)
383         tracks |= initiator.initiate(measurements - associated_measurements, timestamp
                )
384         all_tracks |= tracks
385
386     end = timer()
387
388     duration = end - start
389     # print(f"runtime: {duration} s")
390     interpolated_ground_truth = interpolate_ground_truth(
391         ground_truth[0], timesteps, model
392     )
393
394     # Evaluate using the ALIGNED ground truth
395     OSPA_times, OSPA_values, OSPA_corrected_values = ospa_stonesoup(
396         all_tracks, ground_truth if simulation else interpolated_ground_truth, model
397     )
398
399     # plotter = Plotter()
400     # plotter.plot_ground_truths(ground_truth, [0, 2] if model == "cv" else [0, 3])
401     # plotter.plot_tracks(all_tracks, [0, 2] if model == "cv" else [0, 3])
402     # plotter.plot_measurements(
403     #     [det for det_set in detections for det in det_set],
404     #     [0, 2] if model == "cv" else [0, 3],
405     #     measurement_model=measurement_model,
406     # )
407     #
408     # ax = plt.gca()
409     # x_min, x_max = ax.get_xlim()
410     # y_min, y_max = ax.get_ylim()
411     # data_min = min(x_min, y_min)
412     # data_max = max(x_max, y_max)
413     # # ax.set_xlim(data_min, data_max)
414     # # ax.set_ylim(data_min, data_max)
415     # ax.set_xlim(-10, 50)
416     # ax.set_ylim(-30, 30)
417     # ax.set_aspect("equal", adjustable="box")
418     #
419     # plt.grid()
420     # plotter.fig.show()
```

```
    # plt.show()

    if plot:
        from stonesoup.plotter import AnimatedPlotterly

        plotter = AnimatedPlotterly(timesteps, tail_length=0.7)
        plotter.fig.update_layout(yaxis=dict(scaleanchor="x", scaleratio=1))
        from stonesoup.sensor.radar.radar import RadarBearingRange
        from stonesoup.types.array import StateVector

        plotter.fig.update_layout(
            shapes=[
                dict(
                    type="rect",
                    xref="x",
                    yref="y",
                    x0=-0.2,
                    y0=-0.2,
                    x1=0.2,
                    y1=0.2, # Draws a small circle around (0,0)
                    fillcolor="blue",
                    line=dict(color="black", width=0),
                )
            ]
        )

        plotter.fig.update_layout(width=700, height=700)
        plotter.plot_measurements(detections, [0, 2] if model == "cv" else [0, 3])
        plotter.plot_ground_truths(
            ground_truth, mapping=[0, 2] if model == "cv" else [0, 3]
        )
        plotter.plot_tracks(
            all_tracks, [0, 2] if model == "cv" else [0, 3], uncertainty=False
        )

        plotter.fig.show()

    return duration, OSPA_times, OSPA_values, OSPA_corrected_values
```

The following code block is the main function.

```
# External Imports
from matplotlib import pyplot as plt
import numpy as np
from datetime import datetime

from pipeline import generate_config, run_algorithm

if __name__ == "__main__":

    # CONFIG
    # filter = "ukf" # "ucmkf", "ekf", or "ukf"
    # model = "ca" # "cv" or "ca"

    ndoppler = 500
    dt = 210e-6 * ndoppler
    start_time = datetime(2026, 5, 28, 8, 10, 18, 33)
```

```
17
18      recording_path = "Data/sidetoside/detections_mvdr_argmax_10db_500.csv"
19      gps_path = "Data/sidetoside/flight2-sidetoside-GPS.csv"
20
21      # select simulation or real data
22      simulation = True
23      plot = False
24      plot_ospa = False
25
26      # simulation count
27      N = 100
28
29      num_datapoints = int(30 / dt)
30      x_initial = 50
31      y_initial = 50
32
33      var_r = 0.444
34      var_phi = 0.000720
35
36      association_distance_sim = 15
37      deletion_covariance_sim = 40
38      initiation_points_sim = 10
39      association_distance = 5
40      deletion_covariance = 15
41      initiation_points = 15
42      gps_offset_delay = 40 # offset for the recordings
43
44      drones_params = [
45          {"x0": x_initial, "y0": y_initial, "v_x": 5, "v_y": 5},
46          {"x0": x_initial - 5, "y0": y_initial - 5, "v_x": -5, "v_y": 5},
47          {
48              "x0": x_initial + 20,
49              "y0": y_initial + 20,
50              "v_x": 3,
51              "v_y": -4,
52              "delay_steps": 10,
53          },
54          {
55              "x0": x_initial + 20,
56              "y0": y_initial - 20,
57              "v_x": 3,
58              "v_y": 5,
59              "delay_steps": 30,
60          },
61      ]
62
63      # sidetoside
64      print("running on simulated data") if simulation else print("running on real data
            ")
65      for filter in ["ucmkf", "ekf", "ukf"]:
66          ospa_times_cv = []
67          ospa_times_ca = []
68          ospa_values_cv = []
69          ospa_values_ca = []
70          ospa_corrected_values_cv = []
71          ospa_corrected_values_ca = []
```

```
    for model in ["cv", "ca"]:
        print(f"running {filter} with {model} model for {N} iterations")
        runtime_total = 0
        ospa_scores = []
        corrected_ospa_scores = []
        for _ in range(N if simulation else 1):
            (
                detections,
                dt,
                data_associator,
                updater,
                deleter,
                initiator,
                ground_truth,
            ) = generate_config(
                var_r,
                var_phi,
                start_time,
                dt,
                model,
                filter,
                drones_params,
                num_datapoints,
                recording_path,
                gps_path,
                gps_offset_delay,
                association_distance_sim if simulation else association_distance,
                deletion_covariance_sim if simulation else deletion_covariance,
                initiation_points_sim if simulation else initiation_points,
                simulation=simulation,
            )
            runtime_i, ospa_times, ospa_values, ospa_corrected_values = (
                run_algorithm(
                    model,
                    detections,
                    start_time,
                    dt,
                    data_associator,
                    updater,
                    deleter,
                    initiator,
                    ground_truth,
                    simulation=simulation,
                    plot=plot,
                )
            )

            if model == "cv":
                ospa_times_cv = ospa_times
                ospa_values_cv = ospa_values
                ospa_corrected_values_cv = ospa_corrected_values
            else:
                ospa_times_ca = ospa_times
                ospa_values_ca = ospa_values
                ospa_corrected_values_ca = ospa_corrected_values

```

```
            runtime_total += runtime_i
            ospa_scores.append(np.mean(ospa_values))
            corrected_ospa_scores.append(np.mean(ospa_corrected_values))

        if simulation:
            print(
                f"runtime: {runtime_total / N}s, OSPA mean: {np.mean(ospa_scores)},
                    OSPA variance: {np.var(ospa_scores)}"
            )
        else:
            print(
                f"runtime: {runtime_total / N}s, OSPA mean: {np.mean(ospa_scores)},
                    corrected OSPA mean: {np.mean(corrected_ospa_scores)}"
            )

    if plot_ospa:
        plt.figure(figsize=(8, 4))
        plt.plot(
            ospa_times_cv,
            ospa_values_cv,
            color="crimson",
            linewidth=2,
            label=f"OSPA Distance CV",
        )
        plt.plot(
            ospa_times_ca,
            ospa_values_ca,
            color="purple",
            linewidth=2,
            label=f"OSPA Distance CA",
        )
        plt.axhline(
            np.mean(ospa_values_cv),
            color="blue",
            linestyle="--",
            label=f"Mean OSPA CV ({np.mean(ospa_values_cv):.3f}m)",
        )
        plt.axhline(
            np.mean(ospa_values_ca),
            color="green",
            linestyle="--",
            label=f"Mean OSPA CA ({np.mean(ospa_values_ca):.3f}m)",
        )
        if not simulation:
            plt.axhline(
                np.mean(ospa_corrected_values_cv),
                color="yellow",
                linestyle="--",
                label=f"Corrected OSPA CV ({np.mean(ospa_corrected_values_cv):.3f}m)
                    ",
            )
            plt.axhline(
                np.mean(ospa_corrected_values_ca),
                color="black",
                linestyle="--",
                label=f"Corrected OSPA CA ({np.mean(ospa_corrected_values_ca):.3f}m)
```

```
                    ",
            )

        plt.title(f"OSPA Over Time {filter}", fontsize=11, fontweight="bold")
        plt.xlabel("Time [s]")
        plt.ylabel("OSPA [m]")
        plt.grid(True, linestyle=":", alpha=0.6)
        plt.legend()
        plt.show()
```

D.2. Kalman Filter Classes

Here, the original code for the different Kalman Filters is shared. These classes will later be wrapped into Stone Soup `Predictor` and `Updater` objects, to integrate with the pipeline.

```python
# External Import
import numpy as np

"""
This file contains the class for the simple kalman filter implementation.
References used are contained in the README.md file.
"""


class KalmanFilter:
    """
    Simple Kalman filter class.

    inputs:
        F: State transition matrix (system model)
        H: Observation matrix
        R: Measurement noise covariance TODO: figure out appropriate values 1
        x0: Initial state estimate
        Q: Process noise covariance (uncertainty in the process) TODO: figure out
            appropriate values 2
        P0: Initial Error covariance (needs to be a large value)

    Methods:
        predict: predicts a new state based on current state.

        update: updates the estimate using the measurement and the kalman gain.
            z: Measurement (cartesian coordinates)
    """

    def __init__(self, F, H, Q, R, x0, P0):
        self.F = F
        self.H = H
        self.Q = Q
        self.R = R
        self.x = x0
        self.P = P0

    def predict(self):
        self.x = np.dot(self.F, self.x)
        self.P = np.dot(self.F, np.dot(self.P, self.F.T)) + self.Q
        return self.x

```

```
    def update(self, z):
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))
        y = z - np.dot(self.H, self.x)
        self.x = self.x + np.dot(K, y)
        I = np.eye(self.P.shape[0])
        self.P = np.dot(I - np.dot(K, self.H), self.P)
        return self.x


class UCMKalmanFilter:
    """
    Unbiased Converted Measurement Kalman filter class.

    inputs:
        F: State transition matrix (system model)
        H: Observation matrix
        R: Measurement noise covariance TODO: figure out appropriate values 1
        x0: Initial state estimate
        Q: Process noise covariance (uncertainty in the process) TODO: figure out
            appropriate values 2
        P0: Initial Error covariance (needs to be a large value)

    Methods:
        predict: predicts a new state based on current state.

        update: updates the estimate using the measurement and the kalman gain.
            z: Measurement (cartesian coordinates)
    """

    def __init__(self, F, H, Q, P0, sigma_r, sigma_phi, x0):
        self.F = F
        self.H = H
        self.Q = Q
        self.P = P0
        self.sigma_r = sigma_r
        self.sigma_phi = sigma_phi
        self.labda_phi = np.exp(-1 * (sigma_phi**2) / 2)
        self.x = x0

    def predict(self):
        self.x = np.dot(self.F, self.x)
        self.P = np.dot(self.F, np.dot(self.P, self.F.T)) + self.Q
        return self.x

    def update(self, z):
        z = z.flatten()

        # compute R
        rxx = (self.labda_phi ** (-2) - 2) * z[1] ** 2 * np.cos(z[0]) ** 2 + 0.5 * (
            z[1] ** 2 + self.sigma_r**2
        ) * (1 + self.labda_phi**4 * np.cos(2 * z[0]))
        ryy = (self.labda_phi ** (-2) - 2) * z[1] ** 2 * np.sin(z[0]) ** 2 + 0.5 * (
            z[1] ** 2 + self.sigma_r**2
        ) * (1 - self.labda_phi**4 * np.cos(2 * z[0]))
        rxy = (self.labda_phi ** (-2) - 2) * z[1] ** 2 * np.cos(z[0]) * np.sin(
```

```
            z[0]
        ) + 0.5 * (z[1] ** 2 + self.sigma_r**2) * self.labda_phi**4 * np.sin(
            2 * z[0]
        )
        self.R = np.array([[rxx, rxy], [rxy, ryy]])

        # convert z
        z_converted = np.zeros((2, 1))
        z_converted[0, 0] = self.labda_phi**-1 * z[1] * np.cos(z[0])
        z_converted[1, 0] = self.labda_phi**-1 * z[1] * np.sin(z[0])

        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))
        y = z_converted - np.dot(self.H, self.x)
        self.x = self.x + np.dot(K, y)
        I = np.eye(self.P.shape[0])
        self.P = np.dot(I - np.dot(K, self.H), self.P)
        return self.x


class ExtendedKalmanFilter:
    """
    Extended Kalman filter class.

    inputs:
        f: State transition function
        h: Observation function
        F: Jacobian of f
        H: Jacobian of h
        R: Measurement noise covariance TODO: figure out appropriate values 1
        x0: Initial state estimate
            state vector:
                |   x        |
                |   y        |
                |  x_dot (vx) |
                |_ y_dot (vy) _|
        Q: Process noise covariance (uncertainty in the process) TODO: figure out
            appropriate values 2
        P0: Initial Error covariance (needs to be a large value)

    Methods:
        predict: predicts a new state based on current state.

        update: updates the estimate using the measurement and the kalman gain.
            z: Measurement (cartesian coordinates)
    """

    def __init__(self, f, h, F, H, Q, R, P0, x0):
        self.f = f
        self.h = h
        self.F = F
        self.H = H
        self.Q = Q
        self.R = R
        self.x = x0
        self.P = P0
```

```
152
153     def predict(self):
154         A = self.F(self.x)
155         self.x = self.f(self.x).flatten()
156         self.P = np.dot(A, np.dot(self.P, A.T)) + self.Q
157         return self.x
158
159     def update(self, z):
160         A = self.H(self.x)
161         z = z.flatten()
162         S = np.dot(A, np.dot(self.P, A.T)) + self.R
163         K = np.dot(np.dot(self.P, A.T), np.linalg.inv(S))
164         y = z - self.h(self.x).flatten()
165         self.x = self.x + np.dot(K, y)
166         I = np.eye(self.P.shape[0])
167         self.P = np.dot(I - np.dot(K, A), self.P)
168         return self.x
169
170
171 class UnscentedKalmanFilter:
172     """
173     Unscented Kalman filter class.
174
175     inputs:
176         f: State transition function
177         h: Observation function
178         R: Measurement noise covariance TODO: figure out appropriate values 1
179         x0: Initial state estimate
180         Q: Process noise covariance (uncertainty in the process) TODO: figure out
              appropriate values 2
181         P0: Initial Error covariance (needs to be a large value)
182         alpha: scaling parameter
183         beta: scaling parameter
184         kappa: scaling parameter
185
186     Methods:
187         predict: predicts a new state based on current state.
188
189         update: updates the estimate using the measurement and the kalman gain.
190             z: Measurement (cartesian coordinates)
191     """
192
193     def __init__(self, f, h, Q, R, P0, alpha, beta, kappa, x0):
194         self.f = f
195         self.h = h
196         self.Q = Q
197         self.R = R
198         self.P = P0
199         self.x = x0.flatten()
200         self.L = len(x0)
201
202         self.alpha = alpha
203         self.beta = beta
204         self.kappa = kappa
205         self.labda = alpha**2 * (len(x0) + kappa) - len(x0)
206
```

```
        self.wm, self.wc = self.compute_weights()

    # calculate mean while normalizing angle
    def calculate_polar_mean(self, y):
        y_mean = np.zeros(2)
        y_mean[1] = np.dot(self.wm, y[:, 1])

        sum_sin = np.sum(self.wm * np.sin(y[:, 0]))
        sum_cos = np.sum(self.wm * np.cos(y[:, 0]))
        y_mean[0] = np.arctan2(sum_sin, sum_cos)

        return y_mean

    # keep angle in [-pi, pi]
    def normalize_angle(self, angle):
        return (angle + np.pi) % (2 * np.pi) - np.pi

    # compute sigma point weights
    def compute_weights(self):
        wm = np.full(2 * self.L + 1, 1 / (2 * (self.L + self.labda)))
        wc = np.full(2 * self.L + 1, 1 / (2 * (self.L + self.labda)))
        wm[0] = self.labda / (self.L + self.labda)
        wc[0] = wm[0] + (1 - self.alpha**2 + self.beta)

        return wm, wc

    def generate_sigma_points(self, x, P):
        sigma_points = np.zeros((2 * self.L + 1, self.L))

        try:
            chol = np.linalg.cholesky((self.L + self.labda) * P)
        except (
            np.linalg.LinAlgError
        ): # if matrix is not positive semidefinite, try to make it so
            eps = 1e-9 * np.eye(self.L)
            chol = np.linalg.cholesky((self.L + self.labda) * (P + eps))

        sigma_points[0] = x
        for i in range(self.L):
            sigma_points[i + 1] = x + chol[:, i]
            sigma_points[self.L + i + 1] = x - chol[:, i]

        return sigma_points

    def predict(self):
        # generate sigma points from current estimate
        sigma_points = self.generate_sigma_points(self.x, self.P)
        # pass sigma points trough system model
        transformed_sigma_points = np.array([self.f(s).flatten() for s in sigma_points
            ])

        # estimate mean by summing sigma points with weights
        self.x = np.dot(self.wm, transformed_sigma_points)

        # estimate covariance
        dx = transformed_sigma_points - self.x
```

```
        self.P = np.dot((self.wc * dx.T), dx) + self.Q

        return self.x

    def update(self, z):
        z = z.flatten()

        # recompute sigma points from most recent estimate
        transformed_sigma_points = self.generate_sigma_points(self.x, self.P)
        # pass sigma points trough observation model
        y = np.array([self.h(s).flatten() for s in transformed_sigma_points])

        # estimate mean while keeping angles intact
        y_mean = self.calculate_polar_mean(y)

        # compute residuals
        dy = y - y_mean
        dy[:, 0] = self.normalize_angle(dy[:, 0])
        dx = transformed_sigma_points - self.x

        # estimate covariances
        Pyy = np.dot((self.wc * dy.T), dy) + self.R
        Pxy = np.dot((self.wc * dx.T), dy)

        # kalman gain
        K = np.dot(Pxy, np.linalg.inv(Pyy))

        S = z - y_mean
        S[0] = self.normalize_angle(S[0])

        self.x = self.x + np.dot(K, S)
        self.P = self.P - np.dot(np.dot(K, Pyy), K.T)

        return self.x
```

D.2.1. wrapping the Kalman filter classes

The following code will wrap these classes into the necessary Stone Soup objects.

```
    # External imports
import numpy as np
from datetime import datetime, timedelta

# Predictor and updater for Kalman filter
from stonesoup.predictor.base import Predictor
from stonesoup.updater.base import Updater

# Stonesoup types
from stonesoup.types.detection import Detection
from stonesoup.types.groundtruth import GroundTruthPath
from stonesoup.types.state import State
from stonesoup.types.array import StateVector, CovarianceMatrix
from stonesoup.types.prediction import GaussianStatePrediction
from stonesoup.types.update import GaussianStateUpdate

from stonesoup.base import Property
from stonesoup.models.transition import TransitionModel
```

```
from stonesoup.types.prediction import GaussianMeasurementPrediction


class UCMKFPredictor(Predictor):
    # inherits the properties of the Predictor class (which I want to emulate)
    ucmkf: object = Property()
    # Automatically creates an __init__ with magic syntax and stonesoup magic :0

    transition_model: TransitionModel = Property(default=None)
    # deze ook verplicht for some reason, gebruiken hem verder niet

    def predict(self, prior, timestamp=None, **kwargs):

        self.ucmkf.x = np.array(prior.mean).reshape((-1, 1))
        self.ucmkf.P = np.array(prior.covar)

        x_pred = self.ucmkf.predict()

        return GaussianStatePrediction(
            state_vector=StateVector(x_pred.reshape(-1, 1)),
            covar=CovarianceMatrix(self.ucmkf.P),
            timestamp=timestamp,
        )


class UCMKFUpdater(Updater):

    ucmkf: object = Property()

    measurement_model: object = Property(default=None)

    def update(self, hypothesis, **kwargs):

        prediction = hypothesis.prediction
        measurement = hypothesis.measurement

        self.ucmkf.x = np.array(prediction.mean).reshape((-1, 1))
        self.ucmkf.P = np.array(prediction.covar)

        z = np.array(measurement.state_vector).flatten()

        x_updated = self.ucmkf.update(z)

        return GaussianStateUpdate(
            state_vector=StateVector(x_updated.reshape(-1, 1)),
            covar=CovarianceMatrix(self.ucmkf.P),
            hypothesis=hypothesis,
            timestamp=measurement.timestamp,
        )

    def predict_measurement(self, predicted_state, measurement_model=None, **kwargs):
        x_local = np.array(predicted_state.mean).flatten()
        P_local = np.array(predicted_state.covar)

        z_predicted = np.dot(self.ucmkf.H, x_local)
        z_x = z_predicted[0]
```

```
        z_y = z_predicted[1]
        z_predicted_polar = np.array([np.arctan2(z_y, z_x), np.sqrt(z_x**2 + z_y**2)])

        H_polar = (
            np.array(
                [
                    [
                        (-z_y) / (1e-10 + z_x**2 + z_y**2),
                        0,
                        z_x / (1e-10 + z_x**2 + z_y**2),
                        0,
                    ],
                    [
                        z_x / np.sqrt(1e-10 + z_x**2 + z_y**2),
                        0,
                        z_y / np.sqrt(1e-10 + z_x**2 + z_y**2),
                        0,
                    ],
                ]
            )
            if len(x_local) == 4
            else np.array(
                [
                    [
                        (-z_y) / (1e-10 + z_x**2 + z_y**2),
                        0,
                        0,
                        z_x / (1e-10 + z_x**2 + z_y**2),
                        0,
                        0,
                    ],
                    [
                        z_x / np.sqrt(1e-10 + z_x**2 + z_y**2),
                        0,
                        0,
                        z_y / np.sqrt(1e-10 + z_x**2 + z_y**2),
                        0,
                        0,
                    ],
                ]
            )
        )

        # Standard polar measurement noise
        R_polar = np.array([[self.ucmkf.sigma_phi**2, 0], [0, self.ucmkf.sigma_r**2]])

        S = np.dot(H_polar, np.dot(P_local, H_polar.T)) + R_polar

        # Covariance matrix necessary for the Mahalanobis distance measure, wanted to
            try it
        return GaussianMeasurementPrediction(
            state_vector=StateVector(z_predicted_polar.reshape(-1, 1)),
            covar=CovarianceMatrix(S),
            timestamp=predicted_state.timestamp,
        )

```

```
130
131 class EKFPredictor(Predictor):
132     # inherits the properties of the Predictor class (which I want to emulate)
133     ekf: object = Property()
134     # Automatically creates an __init__ with magic syntax and stonesoup magic :0
135
136     transition_model: TransitionModel = Property(default=None)
137     # deze ook verplicht for some reason, gebruiken hem verder niet
138
139     def predict(self, prior, timestamp=None, **kwargs):
140
141         self.ekf.x = np.array(prior.mean).flatten()
142         self.ekf.P = np.array(prior.covar)
143
144         x_pred = self.ekf.predict()
145
146         return GaussianStatePrediction(
147             state_vector=StateVector(x_pred.reshape(-1, 1)),
148             covar=CovarianceMatrix(self.ekf.P),
149             timestamp=timestamp,
150         )
151
152
153 class EKFUpdater(Updater):
154
155     ekf: object = Property()
156
157     measurement_model: object = Property(default=None)
158
159     def update(self, hypothesis, **kwargs):
160
161         prediction = hypothesis.prediction
162         measurement = hypothesis.measurement
163
164         self.ekf.x = np.array(prediction.mean).flatten()
165         self.ekf.P = np.array(prediction.covar)
166
167         z = np.array(measurement.state_vector).flatten()
168
169         x_updated = self.ekf.update(z)
170
171         return GaussianStateUpdate(
172             state_vector=StateVector(x_updated.reshape(-1, 1)),
173             covar=CovarianceMatrix(self.ekf.P),
174             hypothesis=hypothesis,
175             timestamp=measurement.timestamp,
176         )
177
178     def predict_measurement(self, predicted_state, measurement_model=None, **kwargs):
179
180         x_local = np.array(predicted_state.mean).flatten()
181         P_local = np.array(predicted_state.covar)
182
183         z_predicted = self.ekf.h(x_local).flatten()
184
185         H_matrix = self.ekf.H(x_local)
```

```
186
187         S = np.dot(H_matrix, np.dot(P_local, H_matrix.T)) + self.ekf.R
188
189         # Covariance matrix necessary for the Mahalanobis distance measure, wanted to
                try it
190         return GaussianMeasurementPrediction(
191             state_vector=StateVector(z_predicted.reshape(-1, 1)),
192             covar=CovarianceMatrix(S),
193             timestamp=predicted_state.timestamp,
194         )
195
196
197 class UKFPredictor(Predictor):
198     # inherits the properties of the Predictor class (which I want to emulate)
199     ukf: object = Property()
200     # Automatically creates an __init__ with magic syntax and stonesoup magic :0
201
202     transition_model: TransitionModel = Property(default=None)
203     # deze ook verplicht for some reason, gebruiken hem verder niet
204
205     def predict(self, prior, timestamp=None, **kwargs):
206
207         self.ukf.x = np.array(prior.mean).flatten()
208         self.ukf.P = np.array(prior.covar)
209
210         x_pred = self.ukf.predict()
211
212         return GaussianStatePrediction(
213             state_vector=StateVector(x_pred.reshape(-1, 1)),
214             covar=CovarianceMatrix(self.ukf.P),
215             timestamp=timestamp,
216         )
217
218
219 class UKFUpdater(Updater):
220
221     ukf: object = Property()
222
223     measurement_model: object = Property(default=None)
224
225     def update(self, hypothesis, **kwargs):
226
227         prediction = hypothesis.prediction
228         measurement = hypothesis.measurement
229
230         # Sync UKF with the predicted state
231         self.ukf.x = np.array(prediction.mean).flatten()
232         self.ukf.P = np.array(prediction.covar)
233
234         # Extract measurement vector
235         z = np.array(measurement.state_vector).flatten()
236
237         x_updated = self.ukf.update(z)
238
239         return GaussianStateUpdate(
240             state_vector=StateVector(x_updated.reshape(-1, 1)),
```

```
            covar=CovarianceMatrix(self.ukf.P),
            hypothesis=hypothesis,
            timestamp=measurement.timestamp,
        )

    def predict_measurement(self, predicted_state, measurement_model=None, **kwargs):

        x_local = np.array(predicted_state.mean).flatten()
        P_local = np.array(predicted_state.covar)

        sigma_points = self.ukf.generate_sigma_points(x_local, P_local)
        y = np.array([self.ukf.h(s).flatten() for s in sigma_points])
        y_mean = self.ukf.calculate_polar_mean(y)

        y_diff = y - y_mean
        y_diff[:, 0] = self.ukf.normalize_angle(y_diff[:, 0])

        S = np.dot((self.ukf.wc * y_diff.T), y_diff) + self.ukf.R

        # Covariance matrix necessary for the Mahalanobis distance measure, wanted to
            try it
        return GaussianMeasurementPrediction(
            state_vector=StateVector(y_mean.reshape(-1, 1)),
            covar=CovarianceMatrix(S),
            timestamp=predicted_state.timestamp,
        )
```

D.3. Performance evaluation

```
import numpy as np
import matplotlib.pyplot as plt
from stonesoup.metricgenerator.ospametric import OSPAMetric
from stonesoup.types.state import State
from stonesoup.types.array import StateVector
from stonesoup.dataassociator.tracktotrack import TrackToTruth
from stonesoup.metricgenerator.manager import SimpleManager
from stonesoup.measures import Euclidean

# Reference (Multitarget OSPA): B. Ristic, B. -N. Vo, D. Clark and B. -T. Vo, "A
    Metric for Performance Evaluation of Multi-Target Tracking Algorithms,"
#             in IEEE Transactions on Signal Processing, vol. 59, no. 7, pp.
    3452-3457, July 2011, doi: 10.1109/TSP.2011.2140111

"""
This function calculates and plots the OSPA distance using the stonesoup library
"""

def ospa_stonesoup(track, ground_truth, model, c=10, p=1):

    pos_measure = Euclidean(mapping= [0, 3] if model == "ca" else [0,2])

    ospa_generator = OSPAMetric(c=c, p=p, measure=pos_measure, generator_name="OSPA")

    associator = TrackToTruth(association_threshold=15, measure=pos_measure)

    metric_manager = SimpleManager([ospa_generator], associator=associator)
```

```

    metric_manager.add_data(ground_truth, track)

    metrics = metric_manager.generate_metrics()
    ospa_metric = metrics["OSPA distances"]
    # print(f"Available metrics: {metrics.keys()}")

    ospa_times = [metric.timestamp for metric in ospa_metric.value]
    ospa_values = [metric.value for metric in ospa_metric.value]
    corrected_ospa_values = [value for value in ospa_values if value < 10]

    return ospa_times, ospa_values, corrected_ospa_values
```

D.4. Simulation

Here, the code for three different simulation functions are shared, a linear path, a path with accellerations in random directions, and one with hovering added.

```
    # Imports
import numpy as np
from numpy.random import randn

"""
simulateLinearTrack simulates a linear path taken by a drone

input:
    v_x: initial velocity in x-direction. (m/s)
    v_y: initial velocity in y-direction. (m/s)
    x0: initial x-position. (m)
    y0: initial y-position. (m)
    num_datapoints: number of datapoints in the track.
    dt: timestep (s).
    sigma: standard deviation of the measurement noise.

output:
    measurements: [2 x num_datapoints] array
                (x,y coordinates as rows, samples as columns)

                containing simulated measurements for a straight-line drone track.
"""


def simulateLinearTrack(v_x, v_y, x0, y0, num_datapoints, dt, sigma):
    x = x0
    y = y0
    t = 0
    trueTrack = np.zeros((2, num_datapoints))
    measurements = np.zeros((2, num_datapoints))

    for i in range(num_datapoints):
        trueTrack[0, i] = x + v_x * t
        trueTrack[1, i] = y + v_y * t

        measurements[0, i] = trueTrack[0, i] + sigma * randn()
        measurements[1, i] = trueTrack[1, i] + sigma * randn()

        t += dt
```

```
40
41      return measurements, trueTrack
42
43
44  """
45  simulateLinearTrackPolar simulates a linear path taken by a drone
46
47  input:
48      v_x: initial velocity in x-direction. (m/s)
49      v_y: initial velocity in y-direction. (m/s)
50      x0: initial x-position. (m)
51      y0: initial y-position. (m)
52      num_datapoints: number of datapoints in the track.
53      dt: timestep (s).
54      sigma: standard deviations of the measurement noise.
55
56  output:
57      measurements: [2 x num_datapoints] array
58                  (bearing,range coordinates as rows, samples as columns)
59
60                  containing simulated measurements for a straight-line drone track.
61  """
62
63  def simulateLinearTrackPolar(v_x, v_y, x0, y0, num_datapoints, dt, sigma_r, sigma_phi
        ):
64      x = x0
65      y = y0
66      t = 0
67      trueTrack = np.zeros((2, num_datapoints))
68      measurements = np.zeros((2, num_datapoints))
69
70      for i in range(num_datapoints):
71          trueTrack[0, i] = x + v_x * t
72          trueTrack[1, i] = y + v_y * t
73
74          measurements[0, i] = (
75              np.arctan2(trueTrack[1, i], trueTrack[0, i]) + sigma_phi * randn()
76          )  # azimuth
77          measurements[1, i] = (
78              np.sqrt(trueTrack[0, i] ** 2 + trueTrack[1, i] ** 2) + sigma_r * randn()
79          )  # range
80
81          t += dt
82
83      return measurements, trueTrack
84
85  """
86  simulateRandomAccelTrackPolar simulates a linear path taken by a drone
87
88  input:
89      v_x: initial velocity in x-direction. (m/s)
90      v_y: initial velocity in y-direction. (m/s)
91      x0: initial x-position. (m)
92      y0: initial y-position. (m)
93      num_datapoints: number of datapoints in the track.
94      dt: timestep (s).
```

```
    sigma_r/phi: standard deviations of the measurement noise.
    measure_velocity: standard:False, whether to output velocity info.
output:
    measurements: [2 x num_datapoints] array
                (bearing,range coordinates as rows, samples as columns)

                containing simulated measurements for a straight-line drone track.
"""
def simulateRandomAccelTrackPolar(
    v_x,
    v_y,
    x0,
    y0,
    num_datapoints,
    dt,
    sigma_r,
    sigma_phi,
    sigma_vr=0,
    measure_velocity=False,
):
    curr_x, curr_y = x0, y0
    curr_vx, curr_vy = v_x, v_y
    trueTrack = np.zeros((2, num_datapoints))
    measurements = np.zeros((3 if measure_velocity else 2, num_datapoints))
    v_max = 12
    g = 9.81

    maneuvering = False
    ax, ay = 0, 0

    for i in range(num_datapoints):

        if not maneuvering:
            if np.random.rand() < 0.10:
                maneuvering = True

                accel_mag = np.random.uniform(0.1 * g, 0.7 * g)

                theta = np.random.uniform(0, 2 * np.pi)

                ax, ay = accel_mag * np.cos(theta), accel_mag * np.sin(theta)

        else:
            if np.random.rand() < 0.30:
                maneuvering = False
                ax, ay = 0, 0

        curr_vx += ax * dt
        curr_vy += ay * dt

        v_mag = np.sqrt(curr_vx**2 + curr_vy**2)
        if v_mag > v_max:
            normalization = v_max / v_mag
            curr_vx = curr_vx * normalization
            curr_vy = curr_vy * normalization

```

```
        curr_x += curr_vx * dt
        curr_y += curr_vy * dt

        trueTrack[:, i] = [curr_x, curr_y]

        r = np.sqrt(curr_x**2 + curr_y**2)
        phi = np.arctan2(curr_y, curr_x)

        if measure_velocity:
            v_r = (curr_x * curr_vx + curr_y * curr_vy) / r
            measurements[:, i] = [
                phi + sigma_phi * randn(),
                r + sigma_r * randn(),
                v_r + sigma_vr * randn(),
            ]
        else:
            measurements[:, i] = [phi + sigma_phi * randn(), r + sigma_r * randn()]

    return measurements, trueTrack

"""
simulateRandomAccelHoverTrackPolar simulates a linear path taken by a drone

input:
    v_x: initial velocity in x-direction. (m/s)
    v_y: initial velocity in y-direction. (m/s)
    x0: initial x-position. (m)
    y0: initial y-position. (m)
    num_datapoints: number of datapoints in the track.
    dt: timestep (s).
    sigma_r/phi: standard deviations of the measurement noise.
    measure_velocity: standard:False, whether to output velocity info.
output:
    measurements: [2 x num_datapoints] array
                (bearing,range coordinates as rows, samples as columns)

                containing simulated measurements for a straight-line drone track.
"""
def simulateRandomAccelHoverTrackPolar(
    v_x,
    v_y,
    x0,
    y0,
    num_datapoints,
    dt,
    sigma_r,
    sigma_phi,
    sigma_vr=0,
    measure_velocity=False,
):

    curr_x, curr_y = x0, y0
    curr_vx, curr_vy = v_x, v_y

    trueTrack = np.zeros((2, num_datapoints))
    measurements = np.zeros((3 if measure_velocity else 2, num_datapoints))
```

```
207
208     v_max = 12 # m/s
209     state = "MOVING"
210     hover_timer = 0
211
212     g = 9.81
213     brake_accel = 0.8 * g
214
215     ax, ay = 0, 0
216
217     for i in range(num_datapoints):
218
219         if state == "MOVING":
220
221             if np.random.rand() < 0.1 * dt and i > 5:
222                 state = "BRAKING"
223
224             elif np.random.rand() < dt:
225                 theta = np.random.uniform(0, 2 * np.pi)
226                 maneuver_accel = np.random.uniform(0.5 * g, 1.5 * g)
227                 ax, ay = maneuver_accel * np.cos(theta), maneuver_accel * np.sin(theta)
228
229         if state == "BRAKING":
230
231             v_mag = np.sqrt(curr_vx**2 + curr_vy**2)
232             if v_mag > 5 * dt:
233                 ax = -(curr_vx / v_mag) * brake_accel
234                 ay = -(curr_vy / v_mag) * brake_accel
235             else:
236                 ax, ay = 0, 0
237                 curr_vx, curr_vy = 0, 0
238                 state = "HOVERING"
239                 hover_timer = np.random.randint(5, 15)
240
241         elif state == "HOVERING":
242             ax, ay = 0, 0
243             curr_vx, curr_vy = 0, 0
244             hover_timer -= 1
245
246             if hover_timer <= 0:
247                 state = "MOVING"
248                 launch_theta = np.random.uniform(0, 2 * np.pi)
249                 maneuver_accel = np.random.uniform(0.5 * g, 1.5 * g)
250                 ax, ay = maneuver_accel * np.cos(launch_theta), maneuver_accel * np.sin
                        (
251                         launch_theta
252                     )
253
254         curr_vx += ax * dt
255         curr_vy += ay * dt
256
257         v_mag = np.sqrt(curr_vx**2 + curr_vy**2)
258         if v_mag > v_max:
259             normalization = v_max / v_mag
260             curr_vx = curr_vx * normalization
261             curr_vy = curr_vy * normalization
```

```
262
263         curr_x += curr_vx * dt
264         curr_y += curr_vy * dt
265
266         trueTrack[:, i] = [curr_x, curr_y]
267         r = np.sqrt(curr_x**2 + curr_y**2)
268         phi = np.arctan2(curr_y, curr_x)
269
270         if measure_velocity:
271             v_r = (curr_x * curr_vx + curr_y * curr_vy) / r
272             measurements[:, i] = [
273                 phi + sigma_phi * randn(),
274                 r + sigma_r * randn(),
275                 v_r + sigma_vr * randn(),
276             ]
277         else:
278             measurements[:, i] = [phi + sigma_phi * randn(), r + sigma_r * randn()]
279
280     return measurements, trueTrack
```

D.4.1. Wrapping detections into Stone Soup objects

```
1
2  """
3  input:
4      add_clutter: standard:False whether to add clutter detections
5      start_time: time to provide timestamps
6      sim_function = the function used to simulate (SimulateRandomAccelHoverTrackPolar,
           SimulateLinearTrackPolar etc.)
7      measurement_model = stonesoup object containing the variances and the information
           that it is [bearing, range]
8      dt: timestep
9      drone_configs: all the parameters for the sim_functions (for multiple drones)
10     delay_steps: timedelay for the drone (inside drone config)
11 output:
12     Stone Soup GroundTruth objects
13     Stone Soup Detection objects
14
15
16     example call!!!:
17     measurement_model = CartesianToBearingRange(ndim_state=6, mapping=(0, 3),
           noise_covar=R)
18
19     start_time = datetime.now()
20
21     shared_config = {
22         "sim_function": simulateRandomAccelTrackPolar,
23         "start_time": start_time,
24         "measurement_model": measurement_model,
25         "num_datapoints": num_datapoints,
26         "dt": dt,
27         "sigma_r": range_sigma,
28         "sigma_phi": azimuth_sigma,
29         "add_clutter": True,
30     }
31
32     drones_params = [
```

```
        {"x0": x_initial, "y0": y_initial, "v_x": 5.0, "v_y": 5.0},
        {"x0": x_initial + 100, "y0": y_initial, "v_x": -5, "v_y": 5},
        {
            "x0": x_initial + 100,
            "y0": y_initial + 70,
            "v_x": -5,
            "v_y": 1,
            "delay_steps": 10,
        },
        {
            "x0": x_initial + 150,
            "y0": y_initial + 70,
            "v_x": 1,
            "v_y": 4,
            "delay_steps": 50,
        },
        # {
        #     "x0": x_initial + 30,
        #     "y0": y_initial + 90,
        #     "v_x": 0,
        #     "v_y": -4,
        #     "delay_steps": 30,
        # },
    ]

    detections, ground_truths = SimulatorPolarMultitarget_stonesoup(
        **shared_config, drone_configs=drones_params
    )

"""
def SimulatorPolarMultitarget_stonesoup(**kwargs):
    add_clutter = kwargs.pop("add_clutter", False)
    start_time = kwargs.pop("start_time")
    sim_function = kwargs.pop("sim_function")
    measurement_model = kwargs.pop("measurement_model")
    dt = kwargs.get("dt")

    drone_configs = kwargs.pop(
        "drone_configs", [{}, {}]
    )
    measurements_list = []
    true_tracks_list = []
    delays_list = []

    for config in drone_configs:

        delay_steps = config.pop("delay_steps", 0)
        delays_list.append(delay_steps)

        combined_config = kwargs | config

        meas, track = sim_function(**combined_config)

        measurements_list.append(meas)
        true_tracks_list.append(track)

```

```
    num_drones = len(drone_configs)
    num_time_steps = measurements_list[0].shape[1]

    all_detections = []
    ground_truths = [GroundTruthPath() for _ in range(num_drones)]

    for i in range(num_time_steps):
        timestamp = start_time + timedelta(seconds=i * dt)
        time_step_detections = set()

        for drone in range(num_drones):
            meas = measurements_list[drone]
            track = true_tracks_list[drone]
            delay = delays_list[drone]

            local_i = i - delay

            if 0 <= local_i < meas.shape[1]:

                det = Detection(
                    state_vector=StateVector([meas[0, local_i], meas[1, local_i]]),
                    timestamp=timestamp,
                    measurement_model=measurement_model,
                )
                time_step_detections.add(det)

                if add_clutter:
                    clutter_chance = 0.2
                    if np.random.uniform(0, 1) < clutter_chance:

                        rand_x = np.random.uniform(-30, 150)
                        rand_y = np.random.uniform(-30, 150)
                        rand_phi = np.arctan2(rand_y, rand_x)
                        rand_R = np.sqrt(rand_x**2 + rand_y**2)
                        det = Detection(
                            state_vector=StateVector([rand_phi, rand_R]),
                            timestamp=timestamp,
                            measurement_model=measurement_model,
                        )
                        time_step_detections.add(det)

                x_true = track[0, local_i]
                y_true = track[1, local_i]

                truth_state = State(
                    state_vector=StateVector([x_true, 0, 0, y_true, 0, 0]),
                    timestamp=timestamp,
                )
                ground_truths[drone].append(truth_state)

        all_detections.append(time_step_detections)

    return all_detections, ground_truths
```

D.5. GNSS Coordinate Projecting and Ground-Truth generation

```
   from datetime import datetime, timedelta
from matplotlib.animation import PillowWriter
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.widgets import Slider
from matplotlib.animation import FuncAnimation
# for the gps transformation, coordinate reference system en transformer -> reference is bij de radar (0,0)
from pyproj import CRS, Transformer

from stonesoup.types.state import State
from stonesoup.types.array import StateVector
from stonesoup.types.groundtruth import GroundTruthPath, GroundTruthState


def gps_to_ground_truth(**kwargs):
    delay = kwargs["gps_offset_delay"]
    SensorLat = 51.98880548
    SensorLon = 4.390007015
    start_time = kwargs["start_time"] - timedelta(seconds=delay)
    i=0
    wgs_84 = CRS("EPSG:4326")
    sensor_position = CRS(
        f"+proj=ortho +lat_0={SensorLat} +lon_0={SensorLon} +ellps=WGS84 +datum=WGS84 +units=m +type=crs"
    )
    gps_to_sensor = Transformer.from_crs(wgs_84, sensor_position, always_xy=True)

    path = kwargs["filepath"]
    df = pd.read_csv(path)

    delay = kwargs["gps_offset_delay"]

    timelength = timedelta(milliseconds=int(df.iloc[-1, 0] - df.iloc[0,0]))

    ground_truth = GroundTruthPath()
    ts = []
    for _, row in df.iterrows():

        lat = row["latitude"]
        lon = row["longitude"]

        # time = row["datetime(utc)"]
        # timestamp = datetime.strptime(time, "%Y-%m-%d %H:%M:%S") + timedelta(seconds=38)
        timestamp = start_time + i*timedelta(milliseconds=100) # 100 ms per GPS thing
        i+=1
        x, y = gps_to_sensor.transform(lon, lat)

        # offset angle
        angle_offset = np.deg2rad(72.6)
        x_offset = 0
        y_offset = 0
        x_final = x * np.cos(angle_offset) - y * np.sin(angle_offset) + x_offset
        y_final = x * np.sin(angle_offset) + y * np.cos(angle_offset) + y_offset
```

```
        truth_state = (
            State(
                state_vector=StateVector([x_final, 0, y_final, 0]),
                timestamp=timestamp,
            )
            if kwargs["model"] == "cv"
            else State(
                state_vector=StateVector([x_final, 0, 0, y_final, 0, 0]),
                timestamp=timestamp,
            )
        )
        ts.append(timestamp)
        ground_truth.append(truth_state)

    return [ground_truth]
```

D.6. GNSS Track Interpolation

The next function will interpolate the GNSS track to synchronize timestamps with the estimated track.

```
    def interpolate_ground_truth(ground_truth, timestamps, model):
    original_values = np.array([state.state_vector.flatten() for state in
        ground_truth])
    original_times = np.array([state.timestamp.timestamp() for state in ground_truth])


    target_times = np.array([ts.timestamp() for ts in timestamps])

    num_dims = 4 if model == "cv" else 6
    interpolated_values = np.zeros((len(target_times), num_dims))

    pos_dims = [0, 2] if model == "cv" else [0, 3]

    for dim in pos_dims:
        interpolated_values[:, dim] = np.interp(
            target_times, original_times, original_values[:, dim]
        )

    interpolated_path = GroundTruthPath()
    for ts, state_vec in zip(timestamps, interpolated_values):
        new_state = GroundTruthState(
            state_vector=state_vec.reshape(-1, 1), timestamp=ts
        )
        interpolated_path.append(new_state)

    return [interpolated_path]
```

D.7. Decoding and Wrapping Radar Detections

```
    import numpy as np
from datetime import datetime, timedelta
import pandas as pd

#External
from stonesoup.types.detection import Detection
from stonesoup.types.array import StateVector
```

```
from stonesoup.models.measurement.nonlinear import CartesianToBearingRange
from sklearn.cluster import DBSCAN


def ReadDetections(**kwargs):
    # get path from the keyword arguments
    path = kwargs['filepath']
    measurement_model = kwargs['measurement_model']
    dt = kwargs['dt']
    start_time = kwargs['start_time']

    df = pd.read_csv(path)

    max_blocknum = max(df.iloc[:,0])
    min_blocknum = min(df.iloc[:,0])

    grouped = df.groupby('block')

    all_detections = []
    if path == "Data/sidetoside/detections_mvdr_argmax_10db_500.csv":
        for i in range(min_blocknum, max_blocknum + 1):
            timestamp = start_time + timedelta(seconds=(i-min_blocknum) * dt)

            block_detections = set()

            if i in grouped.groups:
                for col, row in grouped.get_group(i).iterrows():
                    det = Detection(
                        state_vector=StateVector([np.deg2rad(row['angle']), row['range'
                            ]]),
                        timestamp=timestamp,
                        measurement_model=measurement_model
                    )
                    block_detections.add(det)
            all_detections.append(block_detections)
    else:
        for i in range(min_blocknum, max_blocknum+1):
            timestamp = start_time + timedelta(seconds=(i-min_blocknum) * dt)

            block_detections = set()

            if i in grouped.groups:
                for col, row in grouped.get_group(i).iterrows():
                    det = Detection(
                        state_vector=StateVector([np.deg2rad(row['angle_deg']), row['
                            range_m']]),
                        timestamp=timestamp,
                        measurement_model=measurement_model
                    )
                    block_detections.add(det)
            all_detections.append(block_detections)

    return all_detections
```

D.8. Measurement Variance Analysis

Here, the code for the variance analysis is shared:

```
    DURATION = 6.8 #s
#these detections are directly written down from the plot. #48 datapoints in the designated time window
detection_ranges = np.array([7.079, 7.28, 7.362, 7.3587, 7.4754, 7.6418, 7.6418, 7.6418, 7.6418, 7.6418, 7.7631, 7.9, 7.92, 8.005, 8.0664, 8.1513, 8.1513, 8.2128, 8.2533, 8.2483, 8.2786, 8.3, 8.4961, 8.4961, 8.4961, 8.4961, 8.4961, 8.4961, 8.4961, 8.7071, 8.77, 8.8359, 9.9208, 9.9208, 9.0625, 9.0625, 9.0625, 9.344])
detection_angles = np.array([0.0523, 0.0523, 0.0523, 0.0523, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0349, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0.0175, 0,0,0,0,0,0,0,0,0, -0.0175,-0.0175, -0.0342, -0.0342, -0.0523, -0.0523, -0.0523])
GPS_ranges = np.array([8.2207, 8.2398, 8.2723, 8.2821, 8.2967, 8.3019, 8.2995, 8.3035, 8.3018, 8.3070, 8.2730, 8.2752, 8.2547, 8.2580, 8.2425, 8.2474, 8.2310, 8.2340, 8.2319, 8.2343, 8.2247, 8.2275, 8.2214, 8.2224, 8.2189, 8.2172, 8.2105, 8.2095, 8.2303, 8.2325, 8.2263, 8.2276, 8.2332, 8.2354, 8.2365, 8.2383, 8.2546, 8.2556, 8.2493, 8.2489, 8.2511, 8.2535, 8.2674, 8.2707, 8.2837, 8.2862, 8.2859, 8.2897, 8.2967, 8.3051])
GPS_angles = np.array([0.0909, 0.0901, 0.0876, 0.0873, 0.0857, 0.0856, 0.0848, 0.0848, 0.0825, 0.0824, 0.0816, 0.0812, 0.0812, 0.0807, 0.0801, 0.0794, 0.0780, 0.0776, 0.0779, 0.0776, 0.0781, 0.0779, 0.0780, 0.0786, 0.0801, 0.0807, 0.0812, 0.0814, 0.0800, 0.0797, 0.0813, 0.0815, 0.0823, 0.0822, 0.0828, 0.0818, 0.0805, 0.0797, 0.0808, 0.0803, 0.0787, 0.0777, 0.0774, 0.0769, 0.0767, 0.0769, 0.0764, 0.0763, 0.0760, 0.0764])
print("-----GPS------")
print(f"Mean of the GPS ranges: {np.mean(GPS_ranges)}")
print(f"Variance of the GPS range: {np.var(GPS_ranges)}\n")
print(f"Mean of the GPS angles: {np.mean(GPS_angles)}")
print(f"Variance of the GPS angles: {np.var(GPS_angles)}\n")
print("-----Detection data------")
print(f"Mean of the radar range detections: {np.mean(detection_ranges)}")
print(f"Variance of the radar range: {np.var(detection_ranges)}\n")
print(f"Mean of the radar angle detections: {np.mean(detection_angles)}")
print(f"Variance of the radar angles: {np.var(detection_angles)}")
print((np.deg2rad(4.8)**2)/12)
```