UAV OBJECT TRACKING WITH MODULAR ARCHITECTURE

by

Nathaniel Bowness

Under the supervision of

Dr. Miodrag Bolic

An Engineering report

presented to the University of Ottawa and Carleton University

in partial fulfillment of the requirements for the degree of

Master of Computer Science

in the Program of

Computer Science

Ottawa, Ontario, Canada, 2025 © (Nathaniel Bowness) 2025

Declaration

I declare that this major research paper is my original work and has been composed by me. This work has not been submitted, in whole or in part, for any other degree or professional qualification. I confirm that all content within this paper, except where explicitly noted, is my own. Sections that include data and experimental setup images were provided by members of the CARG research group are duly acknowledged.

I also declare that this work adheres to the principles of academic integrity.

UAV OBJECT TRACKING WITH MODULAR ARCHITECTURE

Nathaniel Bowness, Master of Science in Computer Science, University of Ottawa, 2025

Abstract

This paper presents a modular architecture for UAV object tracking designed for deployment on embedded systems using Docker containers. The system integrates radar and video processing pipelines, combining detections with a GMPHD-based tracking algorithm to achieve near real-time performance. The radar data processing algorithm effectively tracks hovering objects within a 0.18-second synchronization window, while the initial video pipeline demonstrates accurate distance estimation of static objects. Testing on an Nvidia Jetson Orin shows the system can process and record data within 0.2-second intervals, making it suitable for real-time applications. This work provides a solid starting point for UAV tracking on embedded systems, allowing for easy testing and improvements to individual pieces. The current solution has room for refinement and scalability to handle more dynamic environments and evolving requirements as algorithms improve over time.

Acknowledgements

I would like to thank my supervisor, Dr. Bolic, for his guidance and consistent support throughout this project. His feedback and insights helped keep me on track and steer the project in the right direction. I would also like to thank Dr. Mehta, who collected and provided all the data in this project and contributed to many discussions that shaped its direction. Additionally, I would like to sincerely thank Dr. Azad for his expertise and assistance with various signal-processing tasks. Lastly, I would like to thank Fardad, Brian, and other members of the CARG research group for their helpful discussions and for fostering such a professional and welcoming environment.

Dedication

I dedicate this work to my wife, Émélie, for your unwavering love and encouragement, and to my parents and brother for their endless support throughout my journey. I also want to express my appreciation to my dogs, Freyja and Svana, for always getting me out of the house and keeping me grounded, and to my friends and other family for always being there to cheer me on. This work would not have been possible without all of you.

Table of Contents

Declaration ii
Abstract iii
Acknowledgements iv
Dedication v
Table of Contents vi
List of Tables viii
List of Figures ix
Definitions and Acronyms xi
1 Introduction 1
2 Literature Review and Background 3
2.1 FMCW Radar 3
2.1.1 Radar Signal Processing 4
2.1.2 Object Detection 4
2.1.3 Object Angle Estimation and Velocity 5
2.2 Video Processing 6
2.2.1 Object Detection and Classification 6
2.2.2 Distance Estimation 7
2.3 Object Tracking 7
2.3.1 Object State Prediction 8
2.3.2 Track Estimation 9
2.3.3 Stone Soup – Library for Track Estimation 9
2.3.4 Evaluation of Object Tracking 10
2.4 Sensor Synchronization for Real-Time Processing 11
2.5 Software 12
2.5.1 Containerized Applications 12
2.5.2 Docker Images 12
3 Methods 14
3.1 System Architecture 14
3.1.1 Running On Different Environments 15
3.1.2 Container Integration With Operating System 16

3.1.3 Hardware 17
3.2 Software Application Design 17
3.2.1 Sensor Data Capture and Processing 17
3.2.2 Data Synchronization 19
3.3 Radar Processing 20
3.3.1 Radar Data Collection 20
3.3.2 Radar Data Processing 21
3.3.3 Radar Experimental Setup with Hovering UAV 23
3.4 Video Processing 24
3.4.1 Object Detection from Video 24
3.4.2 Object Distance and Angle Estimation 25
3.4.3 Distance Estimation Experiment 27
3.5 Object Tracking 27
3.5.1 Picking an Object Tracking Method 27
3.5.2 Modifying Stone Soup for Real-Time 28
4 Results 29
4.1 Real-Time Processing 29
4.2 Radar Processing 30
4.2.1 Hovering UAV Detections 30
4.3 Object Tracking 32
4.3.1 Initial Algorithm Testing 32
4.3.2 Results for Hovering UAV Tracking 33
4.4 Video Processing 35
5 Discussion 38
5.1 Real-Time Processing 38
5.2 Radar Processing 38
5.3 Object Tracking 39
5.4 Video Processing 40
6 Future Work 41
7 Conclusion 43
8 References 44

List of Tables

Table 1: Main technical specifications of the Jetson Orin Nx 16GB board used for initial real-time data collection [65]. This was not used for object-tracking evaluations, that will be future work. 17
Table 2: CA-CFAR parameters used for all signal processing in the experiments. 21
Table 3: FMCW Radar configuration for all test results. 23
Table 4: Average processing time to get detections from the video and radar on the Orin Board, including processing. 29
Table 5: Approximate data creation for a 1-minute trial, broken down into the video, radar and object tracking portions. This is an approximation when saving 1280x720-sized images from the camera at each interval. 30
Table 6: Performance testing results of PDA, JPDA and GMPHD over 1000-time intervals when there is one "true" object to track for each interval with a clutter rate of 3.0. 33
Table 7: Performance testing results of PDA, JPDA and GMPHD over 1000-time intervals when there are ten "true" objects to track for each interval with a clutter rate of 3.0. 33
Table 8: CLEAR MOT metrics, for the gathered data for a UAV hovering above the radar. As well as the maximum deviation from the true object distance. The metrics consider a true positive to be within ~0.4m of the known distance to the object. This is approximately two range bins. 35
Table 9: Measured distances versus the reported distances of a static chair and cup after calibrating the distance coefficients for the camera. 36

List of Figures

Figure 1: Representation of an FMCW chirp transmitted from a transmitter, Tx, for a chirp duration of Tc with a signal bandwidth of B. The reflected signal is later received td time later on each receiver channel Rx [16]. 3
Figure 2: Overview of the CA-CFAR processing algorithm with guard cells [23]. 5
Figure 3: View angle estimation, α, based on FMCW radar with two receiver antennas (Rx1, Rx2) [19]. 5
Figure 4: Graphical representation of the predict-upgrade process of the Kalman filter [39]. 8
Figure 5: Overview of how containers use the underlying infrastructure [58]. 12
Figure 6: System architecture of the containerized object tracking software running on a computer with a Radar Kit and Camera connected to it. 14
Figure 7: Dockerfile for creating a docker image that can be run on Linux (left) and Jetson-Jetpack5 (right) operating systems. They have identical steps, aside from swapping the Ultralytics base image. 15
Figure 8: Docker run commands for running containers with GPU access for Linux (top) and JetPack 5 (bottom). Both containers also mount a container volume to store the processed data directly on the OS. 16
Figure 9: Overview of the containerized tracking application. It contains 3 main parts: the tracking program is responsible for starting data collection and data processing in separate threads and monitoring the queue of detections to find and report current tracks. 18
Figure 10: Flow chart diagram of the main processing loop that synchronizes asynchronous sensor data that is pushed to the *detect_queue* based on the time of the detections. 19
Figure 11: Example output of the frequency versus time graph that is created after taking the SFTF. This graph shows a higher power region at ~-0.6 kHz indicating movement over the entire time interval. 22
Figure 12: Sliding window technique used against the Radar data to continuously calculate the STFT across a window of n samples [70]. 22
Figure 13: Experiment with the DJI mini-UAV hovering about the FMCW Radar at distance d for various experiments. This image shows the DJI mini 1.3 meters from the radar. 23

Figure 14: Code snippet from the primary portion of the video processing algorithm. Using the configured video source, it runs the YOLOv8 model to find bounding boxes, calls a function to get the relevant detection details and submits it to the data queue for tracking. 25
Figure 15: A plot of the signal versus power for the Rx1 and Rx2 signals with lines showing their CA-CFAR threshold is required to be considered a peak. The plot shows a true detection at 29m and noise that resulted in detections at ~1m and ~85m. 30
Figure 16: Frequency versus time plot, with the power levels shown. The plot shows higher power at 29m, indicating motion over the entire 1.3-minute trial. 31
Figure 17: Frequency versus time plot for six samples, with the power levels shown. The plot shows higher power at 29m, indicating there was motion at that distance over the six samples. 31
Figure 18: Plot of the radar detections for a single time interval without checking for movement, i.e. just using the CA-CFAR on the FD signal (left), and a plot of detections after only including detections that also had movement identified with STFT. 32
Figure 19: Sample track shown from the Stone Soup UI, showing the individual detections, the track points and track uncertainty for a specific time. 34
Figure 20: Picture of a video frame with the YOLOv8 bounding boxes, annotated with the calculated distance after calibrating the coefficient for the chair and cup. The measured distance for both was approximately 1.85 m. 36
Figure 21: Updated software application diagram with the tracking algorithm sending data to a collision avoidance algorithm to help calculate actions to prevent a collision. 42

Definitions and Acronyms

Acronym	Definition
AoA	Angle of Arrival
BB	Bounding Box(es)
CA-CFAR	Cell-Average Constant False Alarm Rate
CFAR	Constant False Alarm Rate
CLI	Command-line Interface
CNN	Convolutional Neural Networks
DL	Deep learning
EKF	Extended Kalman Filter
FD	Frequency Domain
FFT	Fast Fourier transform
FMCW	Frequency Modulate Continuous Wave
FOV	Field of View
GMPHD	Gaussian Mixture Probabilistic Data Association
JPDA	Joint Probabilistic Data Association
ML	Machine Learning
MOT	Multiple Object Tracking
MOTA	Multiple Object Tracking Accuracy
MOTP	Multiple Object Tracking Provision
PDA	Probabilistic Data Association
SIAP	Single Integrated Air Picture
SSD	Single Shot multibox Detector
STFT	Short-Time Fourier Transform
TD	Time-Domain
UAV	Unmanned Aerial Vehicles
YOLO	You only look once
2D	Two-dimensional
3D	Three-dimensional

1 Introduction

Unmanned aerial vehicles (UAVs) have seen rapid adoption in many applications over the last few years. Their recent popularity can be attributed to their versatility, maneuverability, and cost-effectiveness. Recent advancements in battery life and sensor technology have further advanced UAV effectiveness. Today, UAVs are commonly used in applications such as: agriculture [1], search and rescue [2], environmental monitoring [3], and object tracking of both aerial and non-aerial object detection [4], [5].

The integration of UAVs with deep learning (DL) and other machine learning (ML) algorithms that can run directly on embedded hardware has significantly improved UAVs' independence. [6]. These algorithms enable UAVs to perform complex tasks independently without relaying data to a central computer for processing. For instance, convolutional neural networks (CNNs) models such as YOLO can quickly process high-resolution images on a UAV [7]. This allows it to quickly and accurately classify fast-moving objects, which is essential for applications like monitoring traffic or wildlife. [3]. DL models are also capable of processing radar data to perform object detection, as demonstrated by the RadarFormer model created by Dalbah et al [8]. Using radar for object detection is particularly useful in conditions where visual sensors alone may fail, such as poor weather or visually cluttered environments. Using on-board UAV sensors and tailored DL modes has opened numerous ways to perform tasks.

The combination of DL models and sensor data has also significantly enhanced UAV capabilities to perform object-tracking [9]. Object tracking is a computer vision technique that monitors the movement of objects seen through a UAV's cameras and sensors over time [10]. By leveraging ML models that can analyze a stream of real-time data, a UAV can track various objects that it has identified in its vicinity [11]. Utilizing diverse sensor data in the object detection and tracking algorithm can be beneficial for operating in various environments. Different sensors allow the UAV to perform consistently across different conditions and obstacles, whereas a system with one type of sensor could not. The ability to track objects is crucial for many applications where UAVs monitor dynamic objects that may be continuously moving themselves or just with respect to the UAV.

Object tracking on UAVs is complex and presents many challenges due to their maximum load and battery life restrictions. These restrictions limit the UAV's processing power, which in turn limits the complexity of algorithms that can be executed on board. It also means any algorithm must strike a careful balance between the performance of tracking the object and the computation processing required on the real-time data. As highlighted by Sampedro et al., UAVs must deploy ML-based techniques that are computationally efficient while still delivering reliable results [12]. Addressing these challenges requires using more efficient computable models. This can be done through either hardware optimization or improved algorithm design, as well as ML models that are tailored to operate in a resource-constrained environment. This could potentially be done through model quantization or other techniques.

The primary motivation for this project is the need for a modular framework capable of supporting object tracking on embedded systems. Radar integration on UAVs provides an essential advantage for tracking objects in environments where visual sensors may face limitations, such as poor weather or visual clutter. The addition of video also adds the ability to classify and detect objects in the field of view quickly. Both fields constantly adapt, and the ability to test different algorithms and models or experiment with variations in processing raw sensor data is crucial for improving UAV tracking systems. With operating systems and hardware constantly evolving, designing a flexible framework to adapt is vital while remaining efficient on resource-constrained embedded devices.

2 Literature Review and Background

2.1 FMCW Radar

The frequency-modulated continuous wave (FMCW) technique [13], [14] is widely used in radar UAV detection due to its high-range resolution and capability to perform real-time surveillance [15]. Unlike pulsed radar systems, FMCW radars continuously transmit a frequency-modulated signal, commonly called a "chirp," that linearly increases or decreases in frequency over a set period, known as the chirp duration T_c. This technique allows for precise measurement of both the range and velocity of targets using relatively low-power [14]. An example of an increasing chirp can be seen in Figure 1.

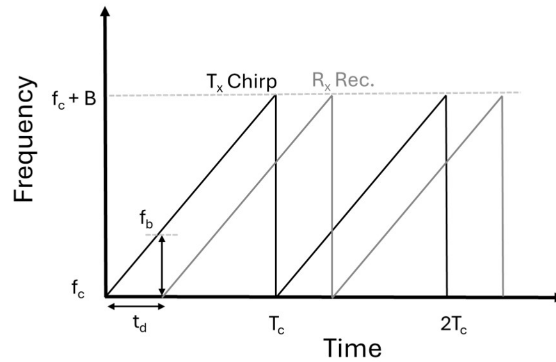

Figure 1: Representation of an FMCW chirp transmitted from a transmitter, Tx, for a chirp duration of Tc with a signal bandwidth of B. The reflected signal is later received td time later on each receiver channel Rx [16].

The following formula can approximate the linear sweeps of FMCW radars:

$$x_t(t) = Ae^{j(wt+\pi \quad ^2)} + A^* e^{-j(wt+\pi \quad ^2)} \qquad (1)$$

where A is the signal amplitude, $S = \frac{B}{T_c}$ is known as the chirp rate, $\omega = 2\pi f_c$ is the lower chirp frequency, and B is the bandwidth of the signal [14], [17].

Newer FMCW radar systems are compact, cost-effective, and energy-efficient, making them suitable for embedded applications [18]. FMCW radars typically come in two-dimensional (2D), like the IMST sr-1200e [19], and three-dimensional (3D) configurations [20]. A 2D FMCW radar typically includes one transmitter and two or more receivers, while a 3D FMCW radar system leverages multiple transmitters and receivers. Due to this configuration, 2D radar can provide

range and velocity information in a single plane. While 3D FMCW radar systems allow for the localization of objects in three-dimensional space, they also allow for more versatility in object tracking.

2.1.1 Radar Signal Processing

FMCW radar sensors usually transmit multiple chirp signals, as described by Eq. 1, sequentially in time. Objects in the radar's environment will reflect the emitted signal and are later captured by one or more receivers. The round-trip delays the received reflections of the transmitted signal t_d, as seen in Figure 1 and mixed with the original signal to produce a beat frequency f_b. Using signal processing techniques like the fast Fourier transform (FFT), high-resolution range data can be obtained from the received signals [17]. Taking the FFT converts the time-domain signal into the frequency domain (FD) so we can find amplitude data across a series of range bins. The frequency bandwidth of the radar will determine the range resolution of these bins, as it impacts the range bin size [14]. The range bin size can be calculated as seen in equation 2, where c is the speed of light, and B is the frequency bandwidth of the Radar.

$$\Delta R = \frac{c}{2B} \qquad (2)$$

2.1.2 Object Detection

It is common to have noise and other various signal inconsistencies for the frequency domain data for the received signals. This can make it challenging to determine which portions of the signal represent a true "detection" representing an object in the radar's view or other noise. To identify objects among the noise, one approach is using a Constant False Alarm Rate CFAR [21], [22], [23]. CFAR allows users to change the detection threshold based on the amount of noise and false detections expected.

There are numerous CFAR algorithms, Figure 2 below is an overview diagram of Cell-Averaging CFAR (CA-CFAR). For CA-CFAR, a range of cells surrounding the Cell-Under-Test (CUT) are divided into training and guard cells to prevent signal increases from the CUT itself. The noised power is calculated at the average of the signal power of the training cells, and the detection threshold is determined by multiplying this noise estimate by a threshold factor. The algorithm then compares the signal in the CUT against this adaptive threshold, marking it as a detection if the signal exceeds the threshold.

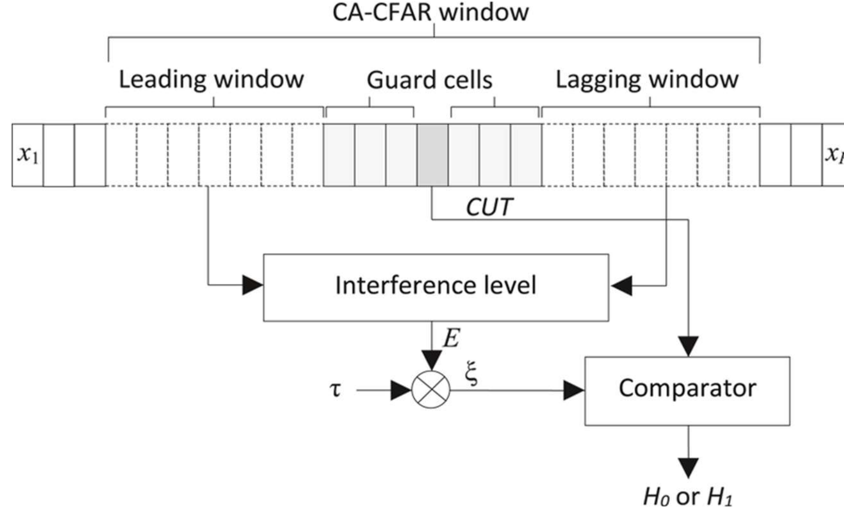

Figure 2: Overview of the CA-CFAR processing algorithm with guard cells [23].

2.1.3 Object Angle Estimation and Velocity

FMCW radar sensors can estimate the angle of arrival (AoA) for detected objects using the spatial separation between their different receivers [17]. Signals measured by two receivers (e.g., Rx1 and Rx2) exhibit a phase shift due to the distance between them. An example diagram of this can be seen in Figure 3.

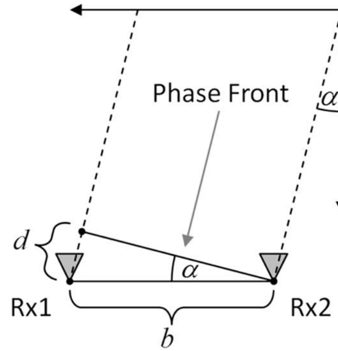

Figure 3: View angle estimation, α, based on FMCW radar with two receiver antennas (Rx1, Rx2) [19].

The detected object's view angle α, is related to the phase shift φ, by equation 3. Where c_0 is the speed of light, f_c is the central frequency of the transmitted signal and b is the distance between receivers.

$$\alpha(\varphi) = \sin^{-1}\left(\frac{\varphi \cdot c_0}{2\pi \cdot f_c \cdot b}\right) \quad (3\ [19])$$

FMCW radars can also estimate a target's velocity using the Doppler shift of the reflected signal. The Doppler shift is analyzed by examining the phase difference of successive chirps, allowing for simultaneous range and velocity measurements [17]. This dual capability is crucial for distinguishing between stationary and moving objects.

The Short-Time Fourier Transform (STFT) is an effective method for analyzing and interpreting movement in UAV radar data [24]. Segmenting the radar signal into short time intervals and applying the Fourier Transform to each segment allows for detecting movement patterns in the data, which can help distinguish moving objects, including the rotation of UAV rotor blades. A spectrogram can visualize the STFT output, a time-frequency plot that provides a clear visual of these behaviours [25].

2.2 Video Processing

Video processing has become a highly effective tool for detecting UAVs [26], [27]. Image processing techniques can identify and classify objects by analyzing individual frames from video feeds, enabling real-time detection and tracking. Analyzing image frames has the benefit of performing more complex detections than you can do with Radar, including UAV payload detection [27] or detection of UAVs versus birds [28]. Recent studies have also shown that UAV to-UAV detection and tracking can be fast and accurate even using hardware embedded in the UAV [29].

2.2.1 Object Detection and Classification

Object detection is a computer vision technique that identifies and classifies objects within an image or video, labelling their locations using bounding boxes (BB) [7]. Numerous object detection ML models have been created that balance speed, accuracy and different techniques for object detection. Some notable models are Singe Shot MultiBox Detector (SSD) [30], EfficientDet [31] and YOLO [7] among many others. This project will leverage and focus on the YOLO series of object detection models as they have become the leading solution for real-time image detection due to their speed and accuracy. One of YOLO's key advantages is its ability to perform quick object detection by segmenting input images into a grid and predicting bounding boxes and class probabilities simultaneously.

This design makes YOLO well-suited for deployment on embedded systems that may have limited computational power such as NVIDIA Jetson boards, which are optimized for edge AI applications. Ultralytics [32], an ML company that has recently committed to maintaining and releasing new versions of YOLO has published docker images that can be run directly on Nvidia Jetson devices, making it a great choice for edge applications, including UAV tracking solutions.

2.2.2 Distance Estimation

Beyond object detection, the bounding boxes produced by YOLO models provide a foundation for estimating object distance and orientation. For a monocular camera, simple distance estimation typically requires calibration of the software and camera, correlating the size and position of bounding boxes with real-world distances. Angle estimation can also be done by analyzing the relative positions and sizes of bounding boxes within the image frame.

Recent research, including [33], and Dist-YOLO [34] have taken these distance estimation algorithms further by training ML algorithms to estimate distances based on image scaling and geometry using a monocular camera. These estimations are generally vital for UAV tracking and interception systems, which rely on accurate spatial data to predict trajectories and behaviours.

2.3 Object Tracking

Object tracking of moving targets is a complex task because of the constantly changing target position, speed and location. Often, sensors do not pick up all targets during each time interval. This can be caused by targets quickly entering and exiting the sensor's field of view, leading to intermittent detections. Sensors are also affected by noise and surrounding environmental issues, which can cause misidentifications of objects that may not be present. Multiple objects in the FOV further complicate tracking, as they can cross paths, hide behind one another, and change speed or direction, which need to be accounted for. These challenges necessitate robust tracking algorithms capable of maintaining accurate object trajectories over time.

Object tracking can be achieved in video processing by analyzing individual frames and associating detected objects across consecutive frames. Models like YOLO (You Only Look Once) facilitate this process by detecting objects and providing bounding box coordinates in each frame [35]. By comparing these coordinates frame by frame, the system can track the movement

and changes of objects over time. This method relies on spatial information within the image to maintain object identities throughout the video sequence [36].

Conversely, radar-based object tracking utilizes distance (range) and angle measurements to estimate object positions. Instead of relying on visual data, radar systems detect objects based on their spatial coordinates relative to the sensor. This approach is particularly effective in conditions where visual data may be unreliable, such as low visibility or adverse weather. However, sophisticated algorithms are required to associate these measurements accurately with specific objects, especially in cluttered environments [37].

2.3.1 Object State Prediction

A critical component of object tracking is the prediction of a target's future state. The Kalman Filter [38] is an algorithm used for state prediction, providing estimates for a target's position and velocity. It operates using a two-step approach for each time interval. First, it predicts the state of the target in the next time step using the system's covariance and motion model. Then, once new data is received, the filter will refine those predictions based on the real data and provide another update in a consecutive loop. The algorithm assumes linearity and Gaussian noise, making it suitable for systems where these conditions hold. Figure 4 shows the two-step process, where $p(x_k|x_{k-1})$ is the state transition probability given the state and time k -1 and z_k is the sensor measurement as time k.

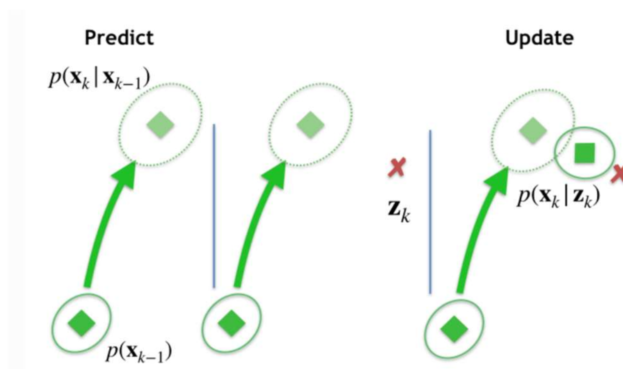

Figure 4: Graphical representation of the predict-upgrade process of the Kalman filter [39].

For systems exhibiting non-linear behaviours, the Extended Kalman Filter (EKF) [40] extends the Kalman Filter to handle non-linearities by linearizing the system around the current estimate.

This adaptation allows the EKF to provide more accurate state estimations in complex scenarios, such as tracking objects with non-linear motion patterns.

2.3.2 Track Estimation

Effective object tracking necessitates advanced estimation techniques to accurately associate measurements with existing tracks and remove older tracks that have not been updated. It's also important to manage uncertainties in multi-object environments, as objects could potentially be associated with different tracks. There are numerous methods that help achieve this with various benefits and drawbacks, three will be discussed below.

One method that was evaluated in this project was the Joint Probabilistic Data Association (JPDA) method [41]. JPDA builds on the original Probabilistic Data Association (PDA) [42], which assigns probabilities to potential associations between measurements and existing tracks by considering the likelihood of each association. JDPA instead considers evaluates all possible associations jointly, offering a more robust solution for tracking multiple targets. This joint consideration ensures that overlapping or closely spaced objects can be accurately tracked without significant ambiguity. The JPDA approach was further developed to address the complexities of multi-target environments.

Another method the Gaussian Mixture Probability Hypothesis Density (GMPHD) filter [37] models the target state as a mixture of Gaussian components, enabling efficient handling of multiple targets. It dynamically initiates new tracks and terminates obsolete ones, making it particularly effective in scenarios where objects frequently appear and disappear. By managing to track births and deaths probabilistically, GMPHD adeptly distinguishes true targets from false alarms, even in high-clutter environments.

2.3.3 Stone Soup – Library for Track Estimation

The Stone Soup software project [43] provides a comprehensive library for implementing object tracking systems, integrating advanced data association techniques into a flexible and extensible platform. Designed for researchers and developers, Stone Soup supports modular components that allow for easy customization and experimentation. Key features include support for a wide range of tracking filters, including Kalman, Extended Kalman, and Particle Filters; built-in implementations of advanced data association algorithms, such as PDA, JPDA, and GMPHD;

and tools for multi-sensor fusion and track evaluation, enabling the development of sophisticated tracking solutions. The flexibility of Stone Soup makes it an ideal choice for UAV tracking applications, where the ability to integrate radar and video data seamlessly is critical. By leveraging its pre-built modules and extensive documentation, users can focus on adapting the framework to their specific use case, reducing development time.

2.3.4 Evaluation of Object Tracking

Evaluating the accuracy of object tracking is an important part of tracking algorithms to see how accurate they are and how they compare against each other. There are two sets of metrics that are often used for performance comparisons between object tracking algorithms, the CLEAR multiple object tracking (MOT) [44] metrics and the single integrated air picture (SIAP) [45] metrics.

The CLEAR MOT metrics are commonly used to evaluate UAV object-tracking algorithm and will be used in this report [46], [47] [48]. CLEAR MOT consists of two common metrics that are used for evaluation: multiple object tracking accuracy (MOTA) and multiple object tracking provision (MOTP). MOTA measures the overall accuracy of a tracking system by accounting for three primary factors: missed detections, false positives, and mismatches. This can be seen in equation 4 below. Misses occur when a ground truth object is present, but the system fails to associate the detection to a track because it either was not picked up by sensors or was identified as too far away to match the correct track. False positives, on the other hand, occur when the tracking algorithm identifies an object through a series of detections that do not correspond to any ground truth object, often due to noise or incorrect associations [46]. The MOTP evaluate the precision of the tracker by calculating the average distance between predicted positions and real positions for each of the tracked objects. Lower values indicate better precision. Equation 5 below shows a general formula for calculating MOTP.

$$MOTA = 1 - \frac{Misses + False\ Positives + ID\ Switches}{Total\ Ground\ Trut\ \ \ Objects} \quad (4)$$

$$MOTP = \frac{\sum_{i,t} Distance(i,t)}{Number\ of\ Match} \quad (5)$$

2.4 Sensor Synchronization for Real-Time Processing

The synchronization of sensors is a critical component of real-time object detection and tracking algorithms. Any misalignment can lead to errors in associating detections across sensors and reduce system performance [49]. However, it can be difficult to synchronize the outputs perfectly when dealing with multiple sensors that may operate with different sampling rates or generate data asynchronously. There are a few common techniques that can be used for sensor synchronization including hardware-level synchronization, dynamic time-warping [50], resampling and event-based synchronization using queues or event buses. This report will mainly focus on event-based synchronization, but we'll also cover resampling as it could have been used as well.

One common technique for handling differences in sampling rates between sensors is **resampling** [51]. For instance, radar systems might output data at a fixed rate, such as every 0.25 seconds, while cameras may produce frames at irregular intervals or higher frame rates. Resampling involves interpolating or down-sampling the higher-frequency data to match the lower-frequency sensor or vice versa. This creates a uniform temporal framework, ensuring that data points from both sensors are comparable and aligned.

Event-based synchronization can be achieved by having each data entry from the asynchronous sensors timestamped upon collection or arrival in the queue [52] [53], [54] [55], [56]. The main system then compares timestamps associated with events in the queue for a predefined temporal window. If a pair of events occurs within that temporal window, both are associated together [55]. This synchronizes the data between 2 different sensors with a maximum offset between sensor data of the defined temporal window length. The event-based strategies allow for buffering detections to wait for new incoming data from sensors that may be slower or simply forward along one of many sensors' data if the temporal window is completed. The event-based synchronization can also use more advanced techniques that may involve averaging measurement if multiple occur with a single temporal window [56].

2.5 Software

2.5.1 Containerized Applications

Containerized applications benefit from dependency isolation for their required libraries and folders while still sharing the same kernel with other containers in the group [57]. The container engine is used to spin up the containers and provide them with CPU and RAM from the base kernel they are on. The significant benefit is the dependency isolation between applications while only requiring one operating system for many applications. An overview of how containers use the underlying host kernel/OS and infrastructure can be seen in Figure 5 below.

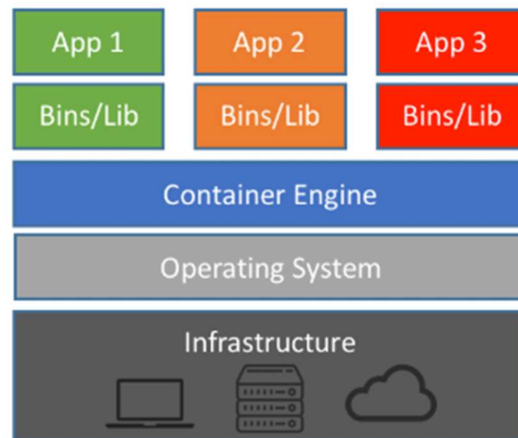

Figure 5: Overview of how containers use the underlying infrastructure [58].

Another key advantage of containerized applications is their portability, which stems from their ability to package an application along with all its dependencies into a single, self-contained unit [57]. This makes running a container across different operating systems easy, given the kernel has a container engine like docker installed. This is particularly important for running applications that need access to specialized hardware like GPUs, or other infrastructure setups.

2.5.2 Docker Images

A Docker image serves as a template for creating one or more containers [59]. At its core, it's a rooted filesystem that includes all the file dependencies required for the applications that will run inside the containers. These are often referred to as base images, which your application software is added on top of. This means that when you launch a container from a Docker image,

everything the application needs to operate—libraries, binaries, and other runtime dependencies—is already baked into the image. Docker makes the process of running containers on different operating systems easier using the base image templates as it supports the main operating systems like Linux or Windows as well as other variants like JetPack 4, 5 and 6, which are used on Nvidia Jeston devices [60].

3 Methods

3.1 System Architecture

The system architecture for this project is designed for modularity, enabling easy replacement of its different components without requiring changes to the others. It is composed of 4 main components: radar, video, data synchronization and object tracking modules. All these modules are currently running in a single container and can interact with the operating system's console and storage using a container runtime. Figure 6 illustrates the architecture and different modules of the system.

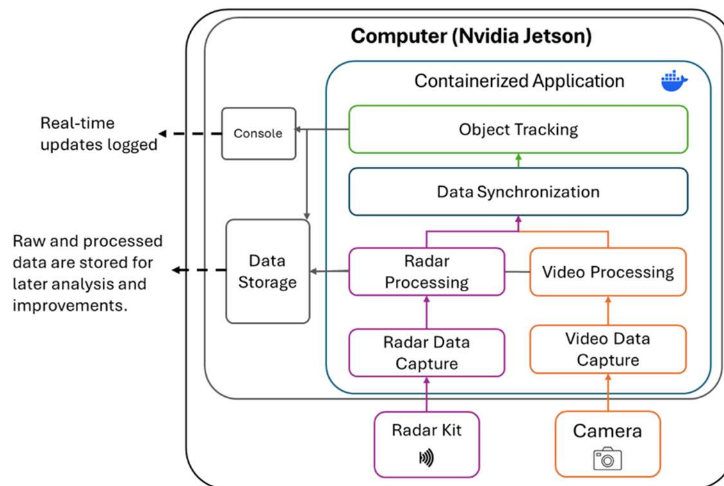

Figure 6: System architecture of the containerized object tracking software running on a computer with a Radar Kit and Camera connected to it.

This system architecture allows for both the radar module and the video module to be run asynchronously, collect data at different rates, and eventually synchronize their outputs to feed into the object-tracking algorithm. The internals of each module will be detailed further in a future section. The synchronization layer also enables additional sensors to be added in the future if required since their outputs could also be synchronized within the same time interval.

An important consideration of the system design is storing the raw data received from the radar and camera sensors for later reproducibility. Radar data is saved upon receipt, while both raw and processed video frames, including bounding box annotations, are stored for later reprocessing as well. The saved image frames can also be compiled into videos for future analysis. This allows

for tests to be done and later re-analyzed, and the raw data can be processed with potentially different algorithms to improve the results later.

Since this is designed as a containerized application, it can work on a local computer or embedded hardware if a container engine can be installed onto the operating system. This includes systems like Windows, Linux or the Nvidia Jetson Orin Boards. The only requirement will be the device has sufficient hardware capabilities to prevent bottlenecks in one of the applications that may require a GPU to keep up with running the ML model processing in real-time.

3.1.1 Running On Different Environments

Ultralytics, a primary maintainer of YOLO, provides pre-built Docker images specifically designed for GPU-enabled containers on different operating systems. These images include all necessary dependencies and are available for various Linux distributions and embedded platforms like Jetson JetPack5 and Jetson JetPack6 [61]. This project adds additional software on top of those base images, such as Python packages for Stone Soup [43] and other required for radar processing and object tracking. Using these base images simplifies the deployment process across different hardware and operating systems. Figure 7 below are the Dockerfiles to create both the Linux and JetPack 5 images. The Dockerfiles are identical to the base image that is used from Ultralytics. Using this approach, it becomes very simple to migrate any operating system that Ultralytics will offer in the future as new software options like JetPack6 or others are required.

```
# Create Linux image
FROM ultralytics/ultralytics:8.2.103

# Copy and install python requirements
COPY ./requirements.txt .
RUN pip install -r requirements.txt

# Copy over folders and files with source code
COPY ./radar ./radar
COPY ./tracking ./tracking
COPY ./video ./video

# Additional copying steps
.....
```

```
# Create Jetson JetPack 5 image
FROM ultralytics/ultralytics:8.2.103-jetson-jetpack5

# Copy and install python requirements
COPY ./requirements.txt .
RUN pip install -r requirements.txt

# Copy over folders and files with source code
COPY ./radar ./radar
COPY ./tracking ./tracking
COPY ./video ./video

# Additional copying steps
.....
```

Figure 7: Dockerfile for creating a docker image that can be run on Linux (left) and Jetson-Jetpack5 (right) operating systems. They have identical steps, aside from swapping the Ultralytics base image.

To streamline the image-building process and software availability, GitHub Actions [62] were implemented to automate the creation of Docker images. These actions build both Linux and JetPack5-compatible images on demand whenever changes are pushed to the GIT repository. [63]. The images are then published to Dockerhub [64] so they are available to any system that has internet access. This allows for building default Linux and ARM-specific images that can be pulled onto the corresponding operating systems and quickly run without ever directly interacting with the code. This automation simplifies getting new code, or algorithms onto all systems.

3.1.2 Container Integration With Operating System

There are small differences between running the containerized application based on the operating system. This is caused by the different OS architectures and how GPUs are integrated. Jetson boards required the "—runtime=nvidia" flag, as seen in Figure 8, because of their integration of GPU and ARM-based architecture. Whereas Linux serves discrete GPUs that can use the "—gpu" flag provided by the Nvidia toolkit to access and run workloads on the GPU. Aside from the slightly different runtime flags, the containers running the software function the same.

```
# Linux - docker run command, that attaches all gpus to the container, and an output volume
docker run -it --gpus all tracking-image -v "$(pwd)"/output:/output

#JetPack5 – docker run command, that will use the ipc host and attach an output volume
docker run --ipc=host --runtime=nvidia -it tracking-image -v "$(pwd)"/output:/output
```

Figure 8: Docker run commands for running containers with GPU access for Linux (top) and JetPack 5 (bottom). Both containers also mount a container volume to store the processed data directly on the OS.

Another aspect of container integration is accessing the sensors. The sensors for all experiments were either directly attached to the operating system, so they could be accessed by the container, or available through the. It's also required to write output files directly to the host operating system for later analysis. This can be done by mounting an output container volume from the container to a folder on the host OS as shown in Figure 8. Since each trial run might involve different parameters or settings, it's essential to preserve the results for reproducibility. This ensures all generated data, whether from radar, video processing, or object tracking, is accessible outside the container and can be used for future offline evaluation and testing if desired.

3.1.3 Hardware

For real-time software testing, the Nvidia Jetson Orin 16GB was used for initial testing to ensure the software works on embedded hardware. A full breakdown of the Nvidia Jetson Orin 16GB can be seen in Table 1. For testing, the FMCW Radar was connected to over the local network, and the USB webcam was directly plugged into the Jetson board and made available to the container to stream video.

Table 1: Main technical specifications of the Jetson Orin Nx 16GB board used for initial real-time data collection [65]. This was not used for object-tracking evaluations, that will be future work.

Computer Component	Description
CPU	8-core Arm® Cortex®-A78AE v8.2 64-bit 2MB L2 + 4MB L3
Memory	16GB 128-bit LPDDR5 102.4GB/s
Storage	128GB NVMe
GPU	1024-core NVIDIA Ampere architecture GPU with 32 Tensor Cores

3.2 Software Application Design

3.2.1 Sensor Data Capture and Processing

The software application for this project is entirely Python-based and designed to be executed from the command-line interface (CLI) of a container. An overview of the application structure is shown in Figure 9 below. The application has various configuration options through the CLI or through mounted configuration files to tune aspects of the application, including CFAR params, synchronization windows, IP addresses for the radar and more. The configuration also includes options for disabling the image or radar processing loop for individual testing of one or the other if desired. As mentioned in the system architecture, the design is highly modular, allowing different components to be updated or replaced without impacting the rest of the system.

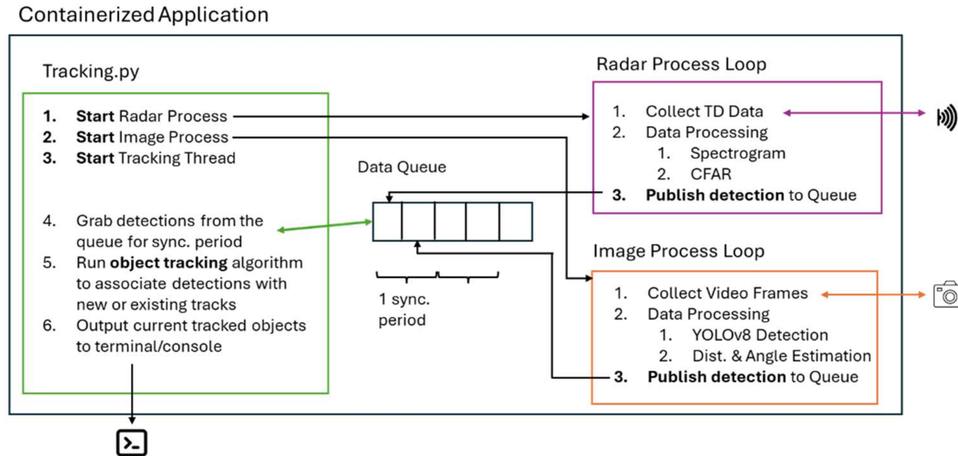

Figure 9: Overview of the containerized tracking application. It contains 3 main parts: the tracking program is responsible for starting data collection and data processing in separate threads and monitoring the queue of detections to find and report current tracks.

The application is divided into 3 main processes: the radar processing loop, the image processing loop, and the main tracking process which contains the synchronization and tracking loop. The main process, "Tracking.py" in Figure 9, serves as the entry point and is responsible for initiating the radar and image processing, which run in separate asynchronous Python processes. These loops handle sensor data capture and independently process the raw data to detect objects and publish them to the event data queue. Both the radar and video processing loops also save both raw sensor data and processing results to the local computer. This enables future offline analysis to reproduce results or try a different algorithm on the data that was captured during the real-time process.

While the radar and image processing loops run, the main process continuously monitors the data queue for incoming detections. The radar and image processing loops add details to the queue, including the detection source, the x and y coordinates of detected objects, and additional object classification details from YOLO for video processing. The main thread then gathers radar and image processing detections during each synchronization period—if data from both sources is available. This aligns the incoming detections from the two sensors. More details about this synchronization will be given in the next section. After synchronization, the tracking system performs data association and object tracking. The software will determine if new detections below with new or existing tracks, as well as remove old tracks if needed. This approach blends the multiple sensor inputs and simply treats them all as detections with a plane.

3.2.2 Data Synchronization

The software uses a single shared queue to manage and synchronize data from both the radar and video processes, implementing an event base architecture. This queue simplifies the architecture while ensuring the asynchronous sensor data can be processed with a defined maximum time difference defined by the synchronization time window. The overall workflow can be seen in Figure 10. The process starts with the video or radar processing, capturing their sensor data, labelling it on a collection with a timestamp generated from the Python program, and processing it. After the processing is complete and detections are potentially found, the processes will add it to the detect_queue.

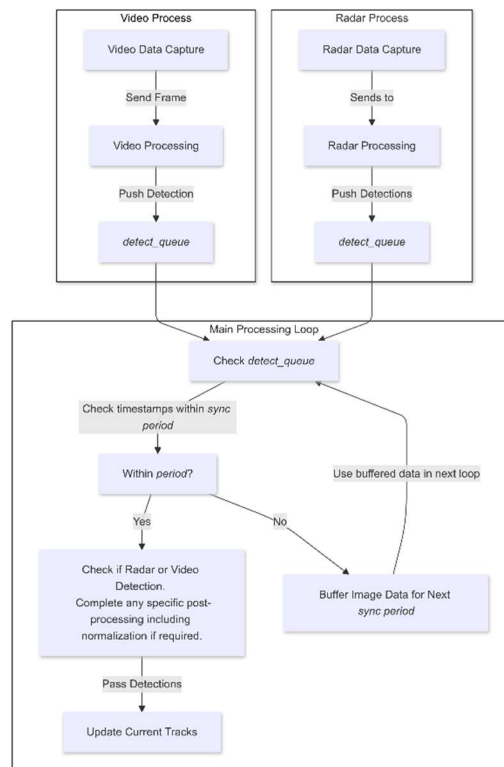

Figure 10: Flow chart diagram of the main processing loop that synchronizes asynchronous sensor data that is pushed to the *detect_queue* based on the time of the detections.

The Main Processing Loop acts as the orchestrator, continuously monitoring the detect_queue, for incoming detections. During each synchronization period, the timestamps of detections in the queue to check if they fall within the defined synchronization window. If the detection is within

this window, it is passed to the object tracking algorithm to associate it with an existing track or create new ones as needed. This synchronization approach means sensor data is aligned but can result in a maximum time difference of the sync window's time between associated detections. If there are multiple batches of detections within a sync window, the current implementation will use the ones with the most recent timestamp to the synchronization window for object tracking. However, in the future multiple sets of detection could be normalized if they occur within the same sync window.

Any detections outside the synchronization period are buffered for the next processing cycle, ensuring no data is lost while maintaining temporal alignment. If any detection in the queue has a timestamp from before the window, they are discarded.

By using the timestamp of when data is collected, as created by the python code, it acts as a unified reference, and the system avoids potential issues such as clock drift or misalignment between sensors. This is a significant advantage of having a single Python application, where time association is consistent and centrally managed. The queue-based architecture also allows for configurable synchronization periods based on the sensor's uses and real-time constraints. Overall, by combining modular sensor processing with synchronization and buffering through the detection queue, the system enhances detection reliability and efficiency in real-time environments.

3.3 Radar Processing

3.3.1 Radar Data Collection

The radar data collection was performed by connecting to the radar using the development kit DK-sR-1200e [19]. The developer kit provides easy integration with the FMCW radar system for configuration and data collection. The developer kit was used for all experiments to retrieve real-time time-domain (TD) data generated from the FMCW radar system based on the transmitted and received signals. This data is used downstream in subsequent processing to identify objects in the radar's FOV.

3.3.2 Radar Data Processing

The radar data processing pipeline begins by applying a hamming window to the TD data collected from the radar. This is done using the NumPy hamming Python library [66]. The hamming window, applied to all 1024 collected data samples, helps reduce the spectral leakage before applying the FFT, improving the accuracy of range and velocity measurements. After the hamming window, the FFT calculation is done on the TD data, converting it into the FD. This conversion enables the identification of range bins corresponding to detected objects. The FFT calculation is done using the NumPy FFT Python library [67].

After converting to the FD, a CA-CFAR algorithm is used to identify peaks that indicate the presence of objects. CA-CFAR detection is applied across all 512 range bins in each of the radar's FD signals from receivers 1 and 2. The algorithm currently assumes any detections made in both receivers correspond to the same object and, therefore, can be treated as a single detection for that range bin [68]. If only one receiver detects an object while the other does not, it is assumed to be a real object rather than noise, and the detection is sent to the tracking algorithm. The best CA-CFAR parameters do vary per experiment and the conditions, but for this report, the CA-CFAR parameters used for all testing are defined in Table 2 below.

Table 2: CA-CFAR parameters used for all signal processing in the experiments.

CA-CFAR Parameter	**Value**
Threshold	4
Guard Cells	2
Reference Cells	5

After using CA-CFAR to identify object detections, there were false detections found within the FD signal. This could have been reflections of nearby objects or just noise in the signal measurement. To improve processing, a step to compute the STFT of recent samples was added to the processing. The STFT helps identify regions in the signal that indicate movement. This helps reduce any false detections by eliminating regions where the STFT identifies no movement. The STFT was calculated with the SciPy python library, allowing the system to analyze the power level of specific frequencies over time [69]. Detected frequencies with high-power regions, such as the ~0.6 kHz frequency in Figure 11, were extracted by applying CA-

CFAR to the average power spectrum of the STFT output, isolating high-power regions corresponding to movement. Based on the radar's parameters, these frequencies were then converted back to the correct distance and range bin using the corresponding frequency.

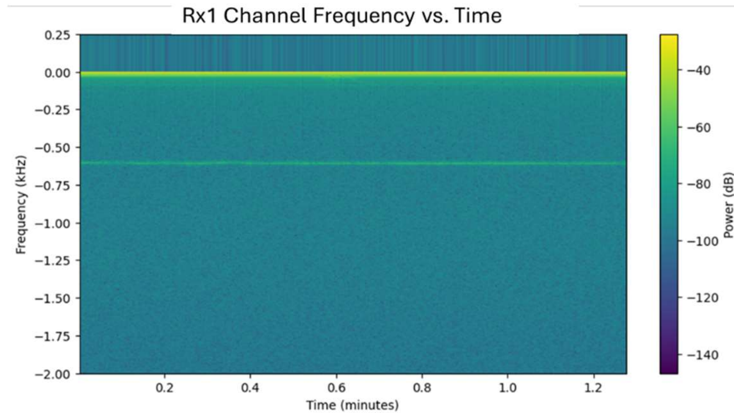

Figure 11: Example output of the frequency versus time graph that is created after taking the SFTF. This graph shows a higher power region at ~-0.6 kHz indicating movement over the entire time interval.

The STFT is often applied across the entire signal to identify movement, however given the real-time nature of this design the STFT will be applied using a windowed approach. A visual of the sliding window approach can be seen in Figure 12. This means STFT will be calculated using the most recent n samples of FD data. For objects where there is minimal movement within the n samples, it is easy to estimate the range bin where the distance occurs. For these experiments, a window size of 6 samples was selected, as it was sufficient to see specific frequency regions with increased power that identified moving objects. The current and initial implementation only looks for movement using the STFT, but eventually, it could be improved by looking for micro-doppler signatures.

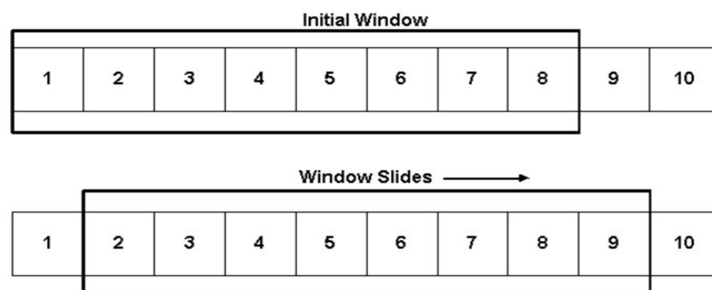

Figure 12: Sliding window technique used against the Radar data to continuously calculate the STFT across a window of n samples [70].

After identifying the object detections for specific range bins using the CA-CFAR algorithm and spectrogram analysis, the AoA is estimated using the original FFT results. We can map detections onto the polar or cartesian planes for object tracking using the range bin for objects and the angle.

3.3.3 Radar Experimental Setup with Hovering UAV

To test the radar processing pipeline, experiments were conducted using the stationary 24 GHz FMCW radar [19], using the radar settings found in Table 3. All object detection sets were done using a DJI Mini 3 Drone [71] hovering at predetermined distances above it. The radar faced the sky, capturing reflections from the hovering UAV. The experimental setup can be seen in Figure 13 below.

Figure 13: Experiment with the DJI mini-UAV hovering about the FMCW Radar at distance d for various experiments. This image shows the DJI mini 1.3 meters from the radar.

Table 3: FMCW Radar configuration for all test results.

Setting	Value (Units)
Start-Frequency	24000 (Mhz)
Stop-Frequency	24750 (Mhz)
Ramp Time	3 (ms)
Number of Samples	1024
Range Bin Size	199.939 (mm)
Bin Size	339 (Hz)
Zero Pad Factor	1
Normalization	1

Tests were conducted with the DJI Mini UAV hovering above the radar at various distances, d of: 1.3m, 5m, 15m, 20m, and 29m. These tests provide a baseline for UAV detection using the radar processing algorithm for a stationary object.

3.4 Video Processing

The video processing component for this project is designed as a preliminary example to show how video-based object detection can be integrated into this system. The current implementation captures raw video frames from a monocular camera and then processes them using a GPU-accelerated ML model, YOLOv8, to perform object detection and estimate the object's location. These detections will then be added to the detection queue and passed for object tracking. The following sections will cover the initial implementation of YOLOv8 and object location estimation [72].

3.4.1 Object Detection from Video

The object detection algorithm can connect to a configurable video source, including online video streams or locally attached cameras. The current algorithm captures image frames from the video using OpenCV [73] at a predefined interval. This enables saving each raw image for later processing and analysis of the video data. Each frame is processed independently using the YOLOv8 object detection model [74]. It performs a frame-by-frame image analysis to identify objects and their corresponding BB. The BB is used to estimate the distance and angular position of each detected object in the frame. Details on the distance and angular position calculations will be discussed further in the next section. Once the estimated position of each object is found, the detected object's position is sent to the data queue for use by the object tracking algorithm. A code snippet for the overall video processing algorithm can be seen in Figure 14 below.

```
# Run detections on image frames from the configured source, using the desired mode and get back a stream of results
results = model.detect(source=source, conf=confidence_threshold, …)
for i, result in enumerate(results):
    # Save the original image to disk, in the <output_folder>/raw/*
    orig_img_rgb = Image.fromarray(result.orig_img[..., ::-1])  # Convert BGR to RGB
    orig_img_rgb.save(os.path.join(output_folder, "raw", f"image_{i}_{orig_img_w}x{orig_img_h}.jpg"))

    detections = []
    detectionTimestamp = result.timeStamp
    # Iterate over the detected objects on this frame, and add all tracking details list before sending it to the data_queue
    for box in result.boxes:
        detected_object_name = result.names[box.cls[0].item()]
        detection = detection_from_bbox(box, detected_object, camera_details=camera, print_details=video_config.printDetectedObjects)
        detections.append(detection)

    # Put the detections into the queue
    data_queue.put(DetectionsAtTime(detectionTimestamp, IMAGE_DETECTION_TYPE, detections))
```

Figure 14: Code snippet from the primary portion of the video processing algorithm. Using the configured video source, it runs the YOLOv8 model to find bounding boxes, calls a function to get the relevant detection details and submits it to the data queue for tracking.

While YOLOv8 offers built-in tracking algorithms that extend the frame-by-frame analysis into temporal object tracking, they were not used in this implementation. Incorporating YOLO's built-in tracking capabilities could be explored in the future.

3.4.2 Object Distance and Angle Estimation

This project's method for estimating object distance and angle relies on the BB geometry derived from the object detection algorithm, combined with the camera's specifications. Using a monocular camera, this technique provides a way to localize objects in 2D space without requiring additional sensors or pose estimation of the object.

The classified object's distance is estimated based on the BB width relative to the image width, calculated using Equation 6. The width of the bounding box, in pixels, is normalized by the image width to provide a fraction that serves as an inverse indicator of distance. We can find a calibration coefficient CBB to relate the bounding box to the actual distance by using the relationship between object size in the image and physical distances. This coefficient can be calibrated before running object tracking using a known object at a known distance. Once you have the coefficient, it can be used to calculate the known object's distance in a new image frame using Equation 7 – where the distance is unknown.

$$BB_{WidthInPix} = \frac{botto \quad right_x - top_{left_x} + 1}{ImageWidt} \qquad (6)$$

$$D = C_{BB} \frac{1}{\frac{BB_{WidthInPix}}{ZoomFactor}} \qquad (7)$$

A default calibrated coefficient can be used for distance estimation of all objects. However, the estimated distance can vary depending on the disparity between the detected objects' shapes. Instead, the coefficient can be calibrated for several different objects. Since YOLOv8 will classify the object, it’s possible to have a coefficient for each object to determine the distance effectively. For this project, the coefficient was only calculated for the UAV and Human, but others would need to be calculated for accurate distance estimation.

Angular estimation involves calculating the bounding box's horizontal and vertical center positions, normalized to the image width and height. Equations 8 and 9 can be used to determine the BB’s centroid in the image, which can then be used to calculate the azimuth, θ, and elevation, ϕ, angles using Equations 11 and 12. The angles calculated are scaled using the camera's horizontal and vertical FOV. A camera’s horizontal FOV is typically part of the specification, and the vertical FOV can be calculated using Equation 10. The centroid’s displacement from the image center provides the angular offsets that describe the object’s position in the camera’s frame of reference.

$$BB_{HorCenterInPix} = \frac{top_{left_x} + bott \quad right_x}{2 \cdot ImageWidt} \qquad (8)$$

$$BB_{VertCenterInPix} = \frac{top_{left_y} + bottom_{right_y}}{2 \cdot ImageHeight} \qquad (9)$$

$$FOV_{vert} = 2 \cdot tan^{-1}\left(tan\left(\frac{FOV_{hor}}{2}\right) \cdot \frac{ImageHeigh}{ImageWidth}\right) \cdot \frac{1}{AspectRatio} \qquad (10)$$

$$\theta = \begin{cases} 90 - \left(\frac{BB_{HorCenterInPix} - \frac{ImageWidth}{2}}{ImageWidth}\right) \cdot \frac{FOV_{horiz}}{ZoomFactor} & , if\ BB_{HorCenterInPix} \geq \frac{ImageWidth}{2} \\ 90 + \left|\left(\frac{BB_{HorCenterInPix} - \frac{ImageWidth}{2}}{ImageWidth}\right)\right| \cdot \frac{FOV_{horiz}}{ZoomFactor} & , otherwise \end{cases} \qquad (11)$$

$$\phi = \begin{cases} \left|\left(\frac{BB_{VertCenterInPix} - \frac{ImageHeight}{2}}{ImageHeight}\right) \cdot \frac{FOV_{vertical}}{ZoomFactor}\right| & , if\ BB_{VertCenterInPix} < \frac{ImageHeight}{2} \\ \left(\frac{BB_{VertCenterInPix} - \frac{ImageHeight}{2}}{ImageHeight}\right) \cdot \frac{FOV_{vertical}}{ZoomFactor} & , otherwise \end{cases} \qquad (12)$$

This approach assumes a simple monocular camera setup, where the bounding boxes directly represent the object. This algorithm does not account for the object’s pose or potential rotation. The algorithm prioritizes simplicity for an initial real-time object-tracking algorithm for video

processing that can be integrated with other sensor data. However, pre-calibration is required to find the CBB for an object you want to track. Future iterations of the system could incorporate more advanced techniques.

3.4.3 Distance Estimation Experiment

Some simple initial distance estimation experiments were done to verify the above algorithm was feasible. A connected camera, specifically an Anker PowerConf C200 Webcam that collects data at 1920(H)x1080(V)@60fps, was used to test the distance and angle estimation of various objects that were calibrated beforehand. The main experiment was moving a chair, with a bottle/cup on top of it, at different distances throughout the room to verify it was able to get the distance correct for both images based on the object's BB size.

3.5 Object Tracking

The object tracking and data association implementation for this project utilizes the Python library Stone Soup, which contains several state estimation and tracking algorithms [43]. Stone Soup is typically designed for offline analysis using pre-recorded data. However, its algorithms were modified for real-time analysis as part of this project.

3.5.1 Picking an Object Tracking Method

Stone Soup provides several object-tracking algorithms, including JPDA, PDA, and GMPHD, all with their strengths. To select the most suitable, experiments were conducted to evaluate the real-time processing and tracking performance. The metrics used for comparison included processing time and accuracy, as the memory requirements with the track trimming discussed in the next section make their impact minimal.

The tracking performance was evaluated using two main experiments comparing Stone Soup's implementation of PDA, JPDA, and GMPHD—the first test simulated 1,000 frames with ten actual tracks per frame. The tracks were spread within 100-meter spacing, using a cluster rate of 3.0 and a 70% probability of true detections from a sensor. Metrics such as processing time and accuracy were analyzed using the CLEAR MOT metrics. The second test focused on more straightforward conditions with only one detection per frame, examining how efficiently the detections handle noise and a low number of objects. Once again, the single track was spread

within 100-meter spacing, using a cluster rate of 3.0 and a 70% probability of true detections from a sensor. Each test was done 5 times to reduce any inconsistencies. These tests highlighted trade-offs between the algorithms, which will be discussed in the results section.

3.5.2 Modifying Stone Soup for Real-Time

For most examples, Stone Soup is typically used for offline evaluation, where all data is preloaded and processed in a single loop. Several main modifications were required to make it work for real-time tracking: handling detections on the fly in near real-time, reducing memory usage, and improving responsiveness.

In this project, each synchronization period from the detection queue is treated as a "time frame," and the detections from that period are sent to Stone Soup incrementally for association with new or existing tracks. This approach removes the need to preload the dataset in advance and instantly lets us dynamically associate incoming detection.

To manage memory efficiently, tracks that haven't been updated within a predefined number of synchronization periods are pruned from the system. These tracks are considered inactive and no longer contribute to the tracking process, so removing them reduces memory usage and speeds up processing. Additionally, every few synchronization periods, the system checks to see which tracks have been updated recently. Tracks that are still active are reported to the console, giving the user real-time feedback on the system's performance and the number of objects being tracked. Any tracks with more than 50 data points also have their state trimmed to keep memory requirements low during this period. With both modifications, Stone Soup can handle incoming real-time data, perform tracking and keep memory requirements small.

4 Results

4.1 Real-Time Processing

The system can achieve near real-time processing by pulling data from a queue for each synchronization window. The performance of this software is highly dependent on the underlying hardware, particularly the CPU and GPU used to run the ML models. Initial testing was conducted on an Nvidia Jetson Orin 16GB over a 1-minute interval, with the results summarized in Figure 13. The radar data showed an average collection interval of approximately 0.1 seconds, with further processing bringing the total time to around 0.15 seconds before it was added to the queue. Video processing, was slightly faster, averaging 0.113 seconds per frame. A synchronization window of 0.17 seconds will be used for all future processing based on these results, ensuring at least one detection from both radar and video processing during each interval. This will ensure that the tracks have approximately five detection points per second from the sensors. This setup highlights the system's ability to operate within real-time constraints while maintaining synchronized sensor inputs.

Table 4: Average processing time to get detections from the video and radar on the Orin Board, including processing.

Processing Type	Average Time (s)
Video Processing (with YOLOv8)	0.113
Radar Processing (with IMST FMCW Radar)	0.154

Since the object tracking algorithm records real-time raw sensor data for offline playback, including radar data and image frames from the video, it's important to consider how much storage is required on the machine running it. If this process is intended to run for extended periods on embedded hardware. The total amount of data collected and saved during a 1-minute trial is summarized in Table 5. The overall data size per minute is most sensitive to the size of the images collected in each sample. The smaller the images, the less storage is used; compression could also be used to reduce storage size.

Table 5: Approximate data creation for a 1-minute trial, broken down into the video, radar and object tracking portions. This is an approximation when saving 1280x720-sized images from the camera at each interval.

Processing Type	Data Size
Video Processing (Raw + Processed Images)	~155 MB
Radar Processing	~12 MB
Object Tracking	~0.1 MB
Total	~167.1 MB

4.2 Radar Processing

4.2.1 Hovering UAV Detections

For the initial analysis of this software, pre-recorded radar data was used as input to the radar processing pipeline, which has been discussed thus far. The pre-recorded data was used as an input source instead of capturing data directly from the real radar. The radar data for hovering UAVs was captured at various distances as previously described; all figures in this section will correspond to the trial when the UAV was hovered 29 meters above the radar.

The first step of the radar processing pipeline is to calculate CA-CFAR on the incoming signal and look for peaks in the signal corresponding to potential object detections. A sample of the CA-CFAR on the Rx1 and Rx1 signals for one single time frame can be seen in Figure 15. This graph shows the true detection at 29m and some noise detections at ~1m and ~85m.

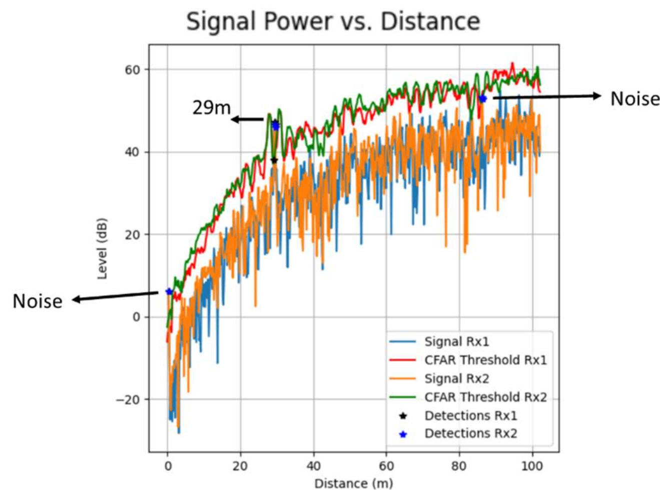

Figure 15: A plot of the signal versus power for the Rx1 and Rx2 signals with lines showing their CA-CFAR threshold is required to be considered a peak. The plot shows a true detection at 29m and noise that resulted in detections at ~1m and ~85m.

Looking at a single frame from the signal analysis, there were more detections from noise than from the specific objects. To improve the results, the algorithm looks for higher power regions in the signal frequency over time. Plots of the signal frequency and the power of it over the entire trial can be seen in Figure 16. A similar plot, but only using a window of 6 samples, so ~1 second can be seen in Figure 17.

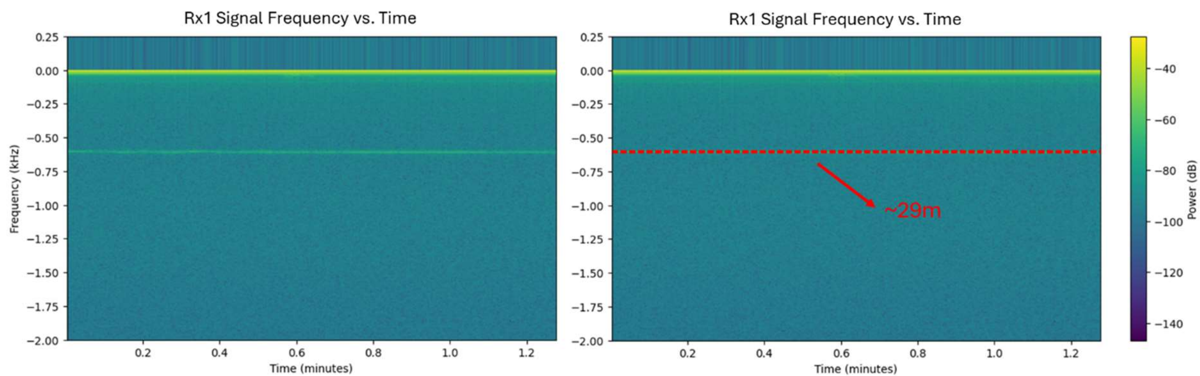

Figure 16: Frequency versus time plot, with the power levels shown. The plot shows higher power at 29m, indicating motion over the entire 1.3-minute trial.

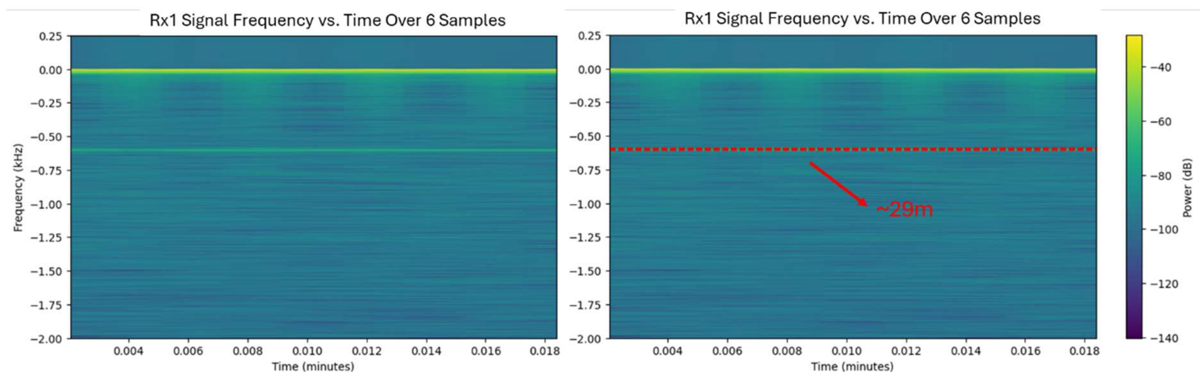

Figure 17: Frequency versus time plot for six samples, with the power levels shown. The plot shows higher power at 29m, indicating there was motion at that distance over the six samples.

By combining the original detections from the CA-CFAR processing of the FD signal with the identified high-power regions from the STFT output, we can effectively reduce false detections before sending them to track. Figure 18 shows the improvement in detection performance, the reduction in false detections, and overall detection performance improvement on a cartesian plot.

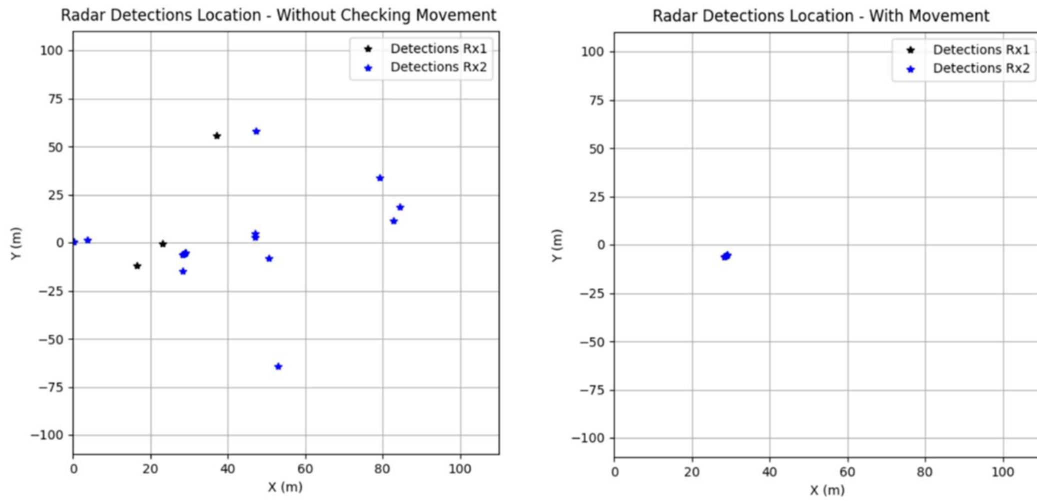

Figure 18: Plot of the radar detections for a single time interval without checking for movement, i.e. just using the CA-CFAR on the FD signal (left), and a plot of detections after only including detections that also had movement identified with STFT.

4.3 Object Tracking

4.3.1 Initial Algorithm Testing

As described in the methodology, it was important to test different tracking options in Stone Soup to evaluate their performance and their balance between speed and accuracy. The performance of the algorithms was analyzed in terms of their average time to associate detections, Multiple Object Tracking Precision (MOTP), and Multiple Object Tracking Accuracy (MOTA).

The first set of results, shown in Table 6, compares the performance of PDA, JPDA, and GMPHD algorithms over 1,000-time intervals with one true object to tracking and a clutter rate of 3.0. In this scenario, JPDA was the most accurate, achieving the highest MOTA and the best MOTP while requiring less processing time than GMPHD. GMPHD also performed well, but on the single object detection had the slowest processing speed.

Table 6: Performance testing results of PDA, JPDA and GMPHD over 1000-time intervals when there is one “true” object to track for each interval with a clutter rate of 3.0.

Algorithm	Average Time to Associate Detections(s)	MOTP (m)	MOTA (%)
PDA	8.5	3.5	94.1
JPDA	10	2.9	96.8
GMPHD	10.2	3.2	95.3

In a more complex trial with ten true tracks and a clutter rate of 3.0, as summarized in Table 7, the results highlight the trade-offs between the algorithms. JPDA remained the most accurate in this scenario with a MOTA of 92.3% and a MOTP of 3.1 meters. However, its processing time increased significantly due to the higher number of tracks and the need to evaluate probabilities across all detections. PDA struggled in this high-clutter, multi-object environment, though it maintained the fastest processing time at 21.10 seconds. GMPHD found a good middle ground, with a good accuracy of 91.8% and a processing time of 25.93 seconds between the 2.

Table 7: Performance testing results of PDA, JPDA and GMPHD over 1000-time intervals when there are ten “true” objects to track for each interval with a clutter rate of 3.0.

Algorithm	Average Time to Associate Detections(s)	MOTP (m)	MOTA (%)
PDA	21.10	4.2	84.2
JPDA	31.13	3.1	92.3
GMPHD	25.93	3.3	91.8

Overall JPDA provided the highest accuracy in both single- and multi-object scenarios but at the cost of significantly higher processing times, particularly in complex environments. PDA performed the fastest but struggled with clusters. GMPHD performed well overall, balancing accuracy and processing time, making it a good fit for near real-time tracking requirements. Based on the results, GMPHD was selected for the remaining object-tracking experiments done as part of this project.

4.3.2 Results for Hovering UAV Tracking

The experimental results for the UAV hovering above the radar, as defined in Section 3.3.3, were processed through the radar pipeline and tracked using the GMPHD algorithm in Stone Stoup.

Figure 19 shows a single time frame of the detections, showing the UAV detections found through CA-CFAR and the STFT processing. The figure also shows the uncertainty of the detection made with the GMPHD algorithm, which appears to be about 1.5 meters. That uncertainty did improve further into the processing as the track was established for longer. After processing the entire dataset, tracking results were evaluated using CLEAR MOT metrics, with ground truth provided by the UAV's GPS controller, which recorded the UAV hovering at 29 meters throughout the experiment. A detection was considered accurate if it was within two range bins, ~0.4 meters, of the ground truth.

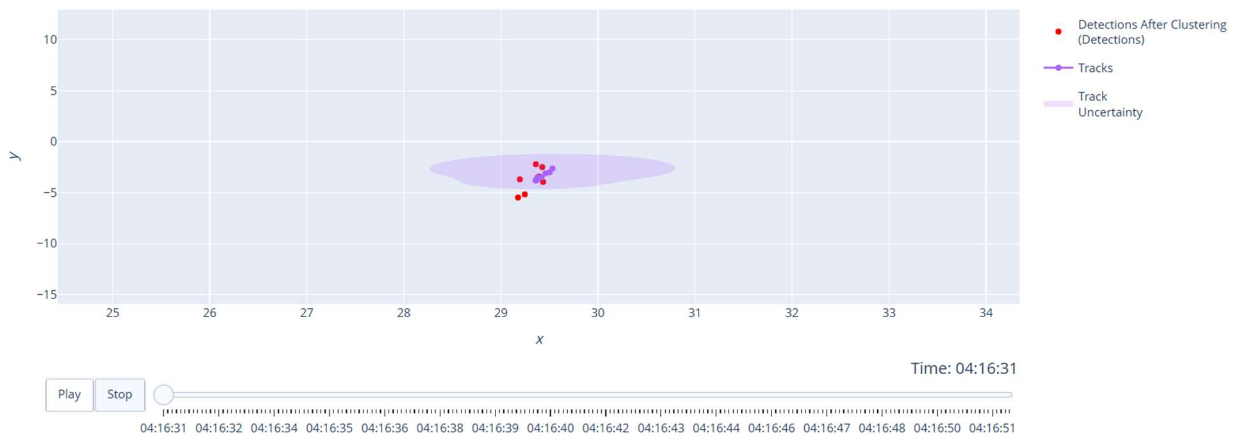

Figure 19: Sample track shown from the Stone Soup UI, showing the individual detections, the track points and track uncertainty for a specific time.

The metrics are summarized in Table 8, showing the tracking accuracy for the different trials where the UAV was hovering at different vertical distances. At shorter distances, the radar processing struggled to consistently identify detections for the UAV, so the tracking algorithm only achieved a MOTA of 60%. At 15 meters, the MOTA was also lower at 66.7% due to consistent false detections and a track being established from those detections for approximately one-third of the trial. Overall, for mid to long ranges, such as 5 meters and 20 meters, the system performed well, with MOTA values of 100% and 99%, respectively. At the maximum range of 29 meters, the MOTA was 65.4%, but this can be partially attributed to the choice of using only two range bins for "true" detections. If the true detection threshold was with three range bins, ~0.6m, the MOTA would have been 94% for 29m.

Table 8: CLEAR MOT metrics, for the gathered data for a UAV hovering above the radar. As well as the maximum deviation from the true object distance. The metrics consider a true positive to be within ~0.4m of the known distance to the object. This is approximately two range bins.

Distance (m)	MOTP (m)	MOTA (%)	Maximum Deviation (m)
1.3	0.07	60	-0.1
5	0.133	100	0.2
15	0.05	66.7	0.1
20	0.148	99	0.3
29	0.522	65.4	0.7

The results also highlight that the MOTP for most of the trials is good, where the distance of the tracks is, on average, always within 5% of the recorded GPS distance value. The maximum deviation throughout the trials also shows any associated tracks are at most 8% off their true value. Overall, the results are somewhat promising but also point to areas for improvement. Additional processing in the radar processing and association phase, including adjustments to thresholds, could significantly improve tracking performance, particularly at close and long distances.

4.4 Video Processing

The results for video processing are preliminary and should be seen as an initial proof of concept. At the time of writing this, the official testing of a camera tracking UAVs or a camera on a UAV tracking other objects has not been completed. However, some sample testing was conducted to demonstrate the video processing algorithm’s capabilities. Using calibrated bounding bound coefficients, C_{BB}, for a chair and cup, as shown in Figure 20, it was possible to estimate the object's positions and angles at various distances.

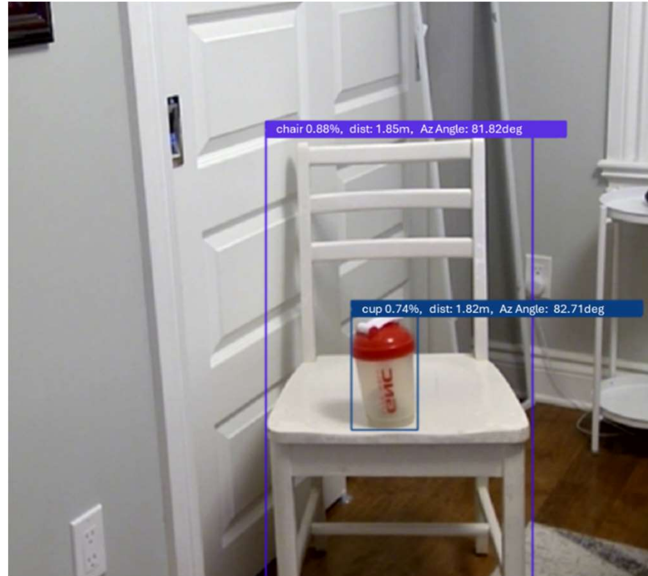

Figure 20: Picture of a video frame with the YOLOv8 bounding boxes, annotated with the calculated distance after calibrating the coefficient for the chair and cup. The measured distance for both was approximately 1.85 m.

The results of the tests are summarized in Table 9, showing the algorithm was able to successfully classify and provide distance estimates that closely matched the measured values. For example, at 1.85 meters, the reported distances were nearly identical, with classification confidence scores of 88% for the chair and 74% for the cup. Similarly, at longer distances like 5 and 10 meters, the algorithm continued to provide accurate distance estimates, with small deviations of a few centimetres from the measured values.

Table 9: Measured distances versus the reported distances of a static chair and cup after calibrating the distance coefficients for the camera.

Measured Distance (m)	**Chair Reported Distance (m)**	**Chair Classification Confidence (%)**	**Cup Reported Distance (m)**	**Cup Classification Confidence (%)**
1.85	1.85	88	1.82	74
3	2.97	85	2.95	73
5	5.01	89	4.96	74
10	9.94	78	9.85	65

The confidence scores for both objects decreased slightly as the distance increased, which is expected due to the limitations of monocular distance estimation at greater ranges. In general, the cup's classification appears to struggle a bit, but this is likely due to the test object being

described as a “shaker bottle” rather than a real cup. Overall, the results are promising. However, it’s important to note the simplicity of this experiment. These results demonstrate the potential of the video processing component as a foundational step, but all detections were performed on static objects. More advanced estimation algorithms will likely be required for this in dynamic environments, with moving objects.

5 Discussion

5.1 Real-Time Processing

From the initial results, the software demonstrates it can meet the near real-time processing constraints, while successfully recording and processing data within a ~0.2 second synchronization window. Both the radar and video processing pipelines are efficient enough to handle the incoming data while maintaining a steady flow of detections for tracking. One challenge noted from the current algorithm is the storage requirement for embedded hardware. This is caused primarily due to the size of the raw and processed images from the video processing pipeline. With two images being saved approximately every 0.113 seconds, this quickly adds up, especially for extended runs on embedded hardware.

To address this, there are opportunities to optimize storage without losing critical data. One approach could be compressing the saved images to reduce their size. Another more efficient method would involve saving only the images while storing the bounding box information as separate text files. This approach would drastically reduce the storage footprint—bounding box data for one minute would be approximately 0.5 MB, compared to the 75 MB currently used for half the video data. These adjustments would make the system more practical for long-term use, especially on resource-constrained devices that may be attached to a UAV.

As more complex algorithms are added to this architecture, it will be important to continuously monitor their processing and adjust that synchronization window as needed if the processing takes longer.

5.2 Radar Processing

The radar processing pipeline demonstrates good performance in detecting hovering UAVs at various distances, but there are many areas for improvement. In a single-frame's signal analysis, more detections often come from noise than the actual target, which remains a challenge in many scenarios. While increasing the CFAR parameters could reduce false detections, it sometimes results in losing the original target, making it a trade-off between sensitivity and noise filtering. The approach of combining CA-CFAR detections with the STFT-based movement analysis has shown improved results for static UAV detections. This approach performs well for detecting

objects at 1.3, 5, 15, 20 and 29 meters as shown in the results section for radar and object tracking. However, the approach does require the objects to remain stationary within the one-second window used for STFT calculations. A key limitation of this method is its reliance on minimal movement within the detection window. If the UAV moves significantly during this period, the algorithm struggles to identify the high-power regions needed for accurate detections. Expanding the algorithm to pull out signals from adjacent regions could improve performance for slightly dynamic objects.

Future improvements could involve training a classifier to validate UAV detections based on combined features such as power level, range, and frequency signature. Adding true micro-Doppler analysis would also be critical, allowing the system to differentiate between UAVs and other moving objects, such as birds, similar to the results in [75]. This classification would make the system more robust in real-world scenarios where noise and non-target objects are common. This classification would make the system more robust in real-world scenarios where noise and non-target objects are common.

More data collection and experimentation with moving objects are necessary to identify and address weaknesses in the current approach. This will help refine the algorithm further, ensuring it performs reliably in both static and dynamic scenarios.

5.3 Object Tracking

The results from the initial UAV tracking results show the system performs okay for association and well for distance accuracy. For example, at 29 meters, the radar processing consistently provides distance estimates within 0.5 meters of the ground truth, translating to less than 2% distance error. This level of accuracy is good, but there's still room for improvement with further refinement of the parameters and processing pipeline to improve. One example is the issue of not consistently detecting objects at 1.3m, this is likely a signal processing issue. The CA-CFAR detection does not always identify the tracks that are causing the errors. This highlights the requirement for all processing to be tuned correctly and accurately. More complex algorithms, in general, would be helpful to accurately identify objects. One potential avenue for improvement is adopting a machine learning-based approach for association [76] [77], which could use the data to optimize track associations based on patterns in the environment.

In general object tracking will greatly benefit from the integration of video data as an additional source for the Stone Soup algorithm. Incorporating detections from the video camera could help improve the robustness of the tracking, especially in cases where radar data alone struggles with noise or sparse detections. This multi-sensor approach could reduce reliance on a single modality and improve the system's ability to handle varying environments.

The current results provide a strong foundation, but additional work on multi-modal integration, parameter tuning, and association algorithms will be key to scaling the system for more dynamic scenarios. There are plans to test and try these association methods on data with moving UAVs, but at the time of writing this, the GPS data was not available, so capturing MOT metrics wasn't possible.

5.4 Video Processing

The video processing results are still in the early stages, with limited experimentation completed so far. The initial tests show that using calibrated coefficients for estimating object distance is feasible, but the method is highly sensitive to the positioning of the targets. Since the approach relies on the relative size of the bounding box, even small changes in object orientation or placement could significantly influence the results. The objects in this experiment were not rotated, which would have increased distance estimation for irregular shapes.

While the current implementation is simple and serves as a proof of concept, it highlights the need for more advanced distance estimation algorithms when using a monocular camera. Methods that go beyond YOLO-based detection and bounding box size, such as the deep learning approach described by Patel et. al [33] could provide more robust and accurate results. Keeping this initial pipeline simple did help validate that the processing pipeline could run detection on the embedded Nvidia Jetson board with this simple algorithm. It lays some of the groundwork for more sophisticated ML-based distance algorithms in future iterations.

6 Future Work

There are several areas of improvement and expansion for this project, particularly in integrating video and radar data. While the current implementation demonstrates promising results, we were unable to test the radar and video feeds simultaneously since synchronized data collection hasn't occurred yet but should soon. Future work will focus on combining these modalities to explore how the radar and video detections can complement each other for better tracking and localization accuracy.

For the radar, a major next step is to move beyond the current STFT-based movement detection and incorporate full micro-Doppler analysis. This would allow the system to identify specific UAV signatures and differentiate them from other objects, like birds. Alternatively, moving to a machine learning-based approach for signal analysis could improve the radar's ability to distinguish between different object types while still handling noise and clutter effectively. Fine-tuning the CFAR parameters with real UAV data is another priority, as this could significantly improve detection accuracy and reduce false positives in various scenarios. When using airborne radar, the presence of motion artifacts and the absence of typical stationary clutter will introduce additional challenges. Addressing these issues will be critical for ensuring the system can reliably process and track moving objects.

For the video processing, advancing beyond the current YOLO-based setup is a natural progression. Incorporating pose estimation or fusing radar and camera data could enhance the system's ability to localize objects with greater accuracy. Additionally, using a more complex CNN or ML-based approach for distance estimation from the monocular camera would be valuable for detecting moving objects and handling more complex scenes. YOLO itself offers built-in tracking algorithms that assign track IDs across frames, which could be explored further to bridge the gap between radar and video tracking and potentially improve object association.

With more collected data, additional adjustments to the Stone Soup parameters will also be necessary. There are many settings related to noise handling, association probabilities, and clutter rates that can be optimized as we gather more varied and dynamic datasets. Once the tracking and estimation systems are better tuned and performing reliably, the next step would be to integrate a collision avoidance system, leveraging the tracking results to create actionable

outputs for real-world UAV operations. A modified software design diagram can be seen in Figure 21 showcasing where a collision avoidance algorithm could fit.

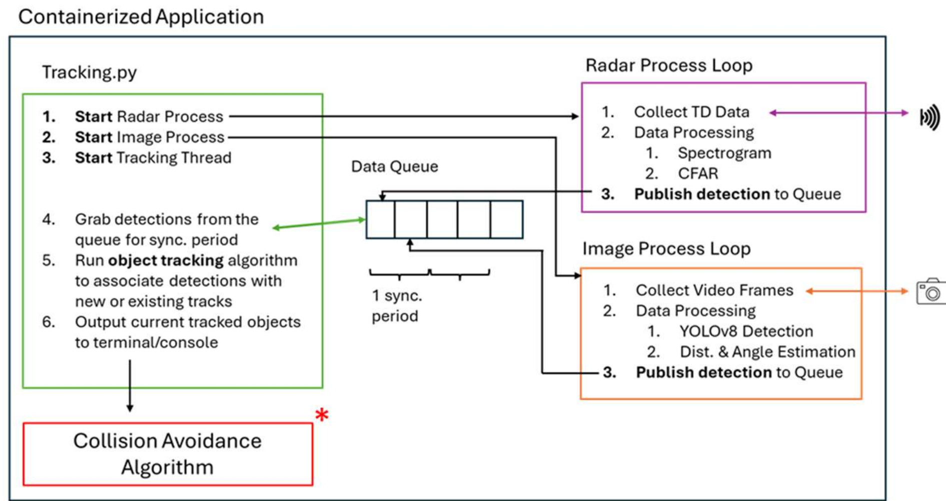

Figure 21: Updated software application diagram with the tracking algorithm sending data to a collision avoidance algorithm to help calculate actions to prevent a collision.

7 Conclusion

This paper has presented a modular architecture for a UAV object tracking algorithm, designed to be easily deployed on various systems, including embedded hardware, using a Docker container. The results demonstrate that the current implementation can run efficiently on an Nvidia Jetson Orin, collecting and processing data in under 0.2-second intervals. The radar processing pipeline and tracking algorithms effectively detect and track hovering objects within the radar's field of view, operating well within the 0.18-second synchronization window. Additionally, the initial video tracking algorithm, when calibrated, shows promise for estimating object distances accurately.

While the individual components have been tested and shown to be effective as a foundational system, there is clear room for improvement. The modular nature of the architecture ensures that future enhancements, such as advanced algorithms or additional processing steps, can be integrated seamlessly. This flexibility makes the system not only a robust starting point for UAV tracking but also a platform that can evolve over time to adopt leading methods and further optimize performance based on collected data.

8 References

[1] C. Zhang and J. Kovacs, “The application of small unmanned aerial systems for precision agriculture: A review,” *ResearchGate*, Oct. 2024, doi: 10.1007/s11119-012-9274-5.

[2] M. Silvagni, A. Tonoli, E. Zenerino, and M. Chiaberge, “Multipurpose UAV for search and rescue operations in mountain avalanche events,” *Geomat. Nat. Hazards Risk*, vol. 8, no. 1, pp. 18–33, Jan. 2017, doi: 10.1080/19475705.2016.1238852.

[3] S. Manfreda, M. McCabe, and P. Miller, “On the Use of Unmanned Aerial Systems for Environmental Monitoring,” ResearchGate. Accessed: Nov. 17, 2024. [Online]. Available: https://www.researchgate.net/publication/323755402_On_the_Use_of_Unmanned_Aerial_Systems_for_Environmental_Monitoring

[4] S. Ojha and S. Sakhare, “Image processing techniques for object tracking in video surveillance- A survey,” in *2015 International Conference on Pervasive Computing (ICPC)*, Jan. 2015, pp. 1–6. doi: 10.1109/PERVASIVE.2015.7087180.

[5] L. Liu, D. Wang, Z. Peng, C. L. P. Chen, and T. Li, “Bounded Neural Network Control for Target Tracking of Underactuated Autonomous Surface Vehicles in the Presence of Uncertain Target Dynamics,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 4, pp. 1241–1249, Apr. 2019, doi: 10.1109/TNNLS.2018.2868978.

[6] A. N. Wilson, A. Kumar, A. Jha, and L. R. Cenkeramaddi, “Embedded Sensors, Communication Technologies, Computing Platforms and Machine Learning for UAVs: A Review,” *IEEE Sens. J.*, vol. 22, no. 3, pp. 1807–1826, Feb. 2022, doi: 10.1109/JSEN.2021.3139124.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91.

[8] Y. Dalbah, J. Lahoud, and H. Cholakkal, “RadarFormer: Lightweight and Accurate Real-Time Radar Object Detection Model,” Apr. 17, 2023, *arXiv*: arXiv:2304.08447. doi: 10.48550/arXiv.2304.08447.

[9] J. Moon, S. Papaioannou, C. Laoudias, P. Kolios, and S. Kim, “Deep Reinforcement Learning Multi-UAV Trajectory Control for Target Tracking,” *IEEE Internet Things J.*, vol. 8, no. 20, pp. 15441–15455, Oct. 2021, doi: 10.1109/JIOT.2021.3073973.

[10] A. Yilmaz, O. Javed, and M. Shah, “Object tracking: A survey,” *ACM Comput Surv*, vol. 38, no. 4, pp. 13-es, Dec. 2006, doi: 10.1145/1177352.1177355.

[11] L.-Y. Lo, C. H. Yiu, Y. Tang, A.-S. Yang, B. Li, and C.-Y. Wen, “Dynamic Object Tracking on Autonomous UAV System for Surveillance Applications,” *Sensors*, vol. 21, no. 23, Art. no. 23, Jan. 2021, doi: 10.3390/s21237888.

[12] C. Sampedro, A. Rodriguez-Ramos, H. Bavle, A. Carrio, P. de la Puente, and P. Campoy, “A Fully-Autonomous Aerial Robot for Search and Rescue Applications in Indoor Environments using Learning-Based Techniques,” *J. Intell. Robot. Syst.*, vol. 95, no. 2, pp. 601–627, Aug. 2019, doi: 10.1007/s10846-018-0898-1.

[13] D. E. Barrick, “FM/CW radar signals and digital processing”, Accessed: Dec. 07, 2024. [Online]. Available: https://repository.library.noaa.gov/view/noaa/18645

[14] A. G. Stove, “Linear FMCW radar techniques,” *IEE Proc. F Radar Signal Process.*, vol. 139, no. 5, pp. 343–350, Oct. 1992, doi: 10.1049/ip-f-2.1992.0048.

[15] F. Foster, H. Rohling and U. Lubbert, “An automotive radar network based on 77 GHz FMCW sensors,” in *ResearchGate*, doi: 10.1109/RADAR.2005.1435950.

[16] Y.-S. Son, H.-K. Sung, and S. W. Heo, “Automotive Frequency Modulated Continuous Wave Radar Interference Reduction Using Per-Vehicle Chirp Sequences,” *Sensors*, vol. 18, no. 9, Art. no. 9, Sep. 2018, doi: 10.3390/s18092831.

[17] J. Michalczyk, C. Schöffmann, A. Fornasier, J. Steinbrener and S. Weiss, “Radar-Inertial State-Estimation for UAV Motion in Highly Agile Manoeuvres.” Accessed: Dec. 07, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9836130

[18] E. Hyun, W. Oh, and J.-H. Lee, “Multi-target detection algorithm for FMCW radar,” in *2012 IEEE Radar Conference*, May 2012, pp. 0338–0341. doi: 10.1109/RADAR.2012.6212161.

[19] Sentire Radar by IMST, “DK-sR-1200e | SENTIRE RADAR.” Accessed: Dec. 08, 2024. [Online]. Available: https://radar-sensor.com/products/developer-kits/dk-sr-1200e.html

[20] J. Yang, J. Thompson, X. Huang, T. Jin, and Z. Zhou, “FMCW radar near field three-dimensional imaging,” in *2012 IEEE International Conference on Communications (ICC)*, Jun. 2012, pp. 6353–6356. doi: 10.1109/ICC.2012.6364681.

[21] C. Ben and T. Islam, “Field study of a 24 GHz FMCW radar system suitable to detect small-sized RPAS under 5 kg MTOW,” 2015. Accessed: Dec. 07, 2024. [Online]. Available: https://www.semanticscholar.org/paper/Field-study-of-a-24-GHz-FMCW-radar-system-suitable-Ben-Islam/22cde2a9890c0968d60f29a328dae244396d9551

[22] M. Hägelen, R. Jetten, R. Kulke, C. Ben, and M. Krüger, “Monopulse Radar for Obstacle Detection and Autonomous Flight for Sea Rescue UAVs,” in *2018 19th International Radar Symposium (IRS)*, Jun. 2018, pp. 1–7. doi: 10.23919/IRS.2018.8448240.

[23] J. Perdoch, S. Gažovová and M. Pacek, “(PDF) An improved radar clutter suppression by simple neural network,” *ResearchGate*, Oct. 2024, doi: 10.1049/rsn2.12510.

[24] D. B. Herr and D. Tahmoush, “Data-Driven STFT for UAV Micro-Doppler Signature Analysis,” in *2020 IEEE International Radar Conference (RADAR)*, Apr. 2020, pp. 1023–1028. doi: 10.1109/RADAR42522.2020.9114726.

[25] R. A. Zitar, M. Al-Betar, M. Ryalat and S. Kassaymeh, “(PDF) A review of UAV Visual Detection and Tracking Methods,” in *ResearchGate*, Nov. 2024. Accessed: Dec. 07, 2024. [Online]. Available: https://www.researchgate.net/publication/371314484_A_review_of_UAV_Visual_Detection_and_Tracking_Methods

[26] M. Liu, X. Wang, A. Zhou, X. Fu, Y. Ma, and C. Piao, “UAV-YOLO: Small Object Detection on Unmanned Aerial Vehicle Perspective,” *Sensors*, vol. 20, no. 8, Art. no. 8, Jan. 2020, doi: 10.3390/s20082238.

[27] V. Mehta, H. Azad, F. Dadboud, M. Bolic, and I. Mantegh, “Real-Time UAV and Payload Detection and Classification System Using Radar and Camera Sensor Fusion,” in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Oct. 2023, pp. 1–6. doi: 10.1109/DASC58513.2023.10311246.

[28] A. Coluccia *et al.*, “Drone-vs-Bird Detection Challenge at IEEE AVSS2021,” in *2021 17th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, Nov. 2021, pp. 1–8. doi: 10.1109/AVSS52988.2021.9663844.

[29] J. Li, D. H. Ye, M. Kolsch, J. P. Wachs, and C. A. Bouman, “Fast and Robust UAV to UAV Detection and Tracking From Video,” *IEEE Trans. Emerg. Top. Comput.*, vol. 10, no. 3, pp. 1519–1531, Jul. 2022, doi: 10.1109/TETC.2021.3104555.

[30] W. Liu *et al.*, “SSD: Single Shot MultiBox Detector,” Dec. 29, 2016, *arXiv*: arXiv:1512.02325. doi: 10.48550/arXiv.1512.02325.

[31] M. Tan, R. Pang, and Q. V. Le, “EfficientDet: Scalable and Efficient Object Detection,” Jul. 27, 2020, *arXiv*: arXiv:1911.09070. doi: 10.48550/arXiv.1911.09070.
[32] Ultralytics, “Home.” Accessed: Nov. 23, 2024. [Online]. Available: https://docs.ultralytics.com/
[33] V. Patel, V. Mehta, M. Bolic, and I. Mantegh, “A Hybrid Framework for Object Distance Estimation using a Monocular Camera,” in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Oct. 2023, pp. 1–7. doi: 10.1109/DASC58513.2023.10311189.
[34] M. Vajgl, P. Hurtik, and T. Nejezchleba, “Dist-YOLO: Fast Object Detection with Distance Estimation,” *Appl. Sci.*, vol. 12, no. 3, Art. no. 3, Jan. 2022, doi: 10.3390/app12031354.
[35] N. M. Krishna, R. Y. Reddy, M. S. C. Reddy, K. P. Madhav, and G. Sudham, “Object Detection and Tracking Using Yolo,” in *2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA)*, Sep. 2021, pp. 1–7. doi: 10.1109/ICIRCA51532.2021.9544598.
[36] L. Tan, X. Dong, Y. Ma, and C. Yu, “A Multiple Object Tracking Algorithm Based on YOLO Detection,” in *2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, Oct. 2018, pp. 1–5. doi: 10.1109/CISP-BMEI.2018.8633009.
[37] B.-N. Vo and W.-K. Ma, “The Gaussian Mixture Probability Hypothesis Density Filter,” *IEEE Trans. Signal Process.*, vol. 54, no. 11, pp. 4091–4104, Nov. 2006, doi: 10.1109/TSP.2006.881190.
[38] R. E. Kalman, “A New Approach to Linear Filtering and Prediction Problems,” *J. Basic Eng.*, vol. 82, no. 1, pp. 35–45, Mar. 1960, doi: 10.1115/1.3662552.
[39] Defence Science and Technology Laboratory, UK, “1 - An introduction to Stone Soup: using the Kalman filter — Stone Soup 1.5.dev121+g0b20e96 documentation.” Accessed: Dec. 07, 2024. [Online]. Available: https://stonesoup.readthedocs.io/en/latest/auto_tutorials/01_KalmanFilterTutorial.html
[40] S. Yang and M. Baum, “Extended Kalman filter for extended object tracking,” in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2017, pp. 4386–4390. doi: 10.1109/ICASSP.2017.7952985.
[41] T. E. Fortmann, Y. Bar-Shalom, and M. Scheffe, “Multi-target tracking using joint probabilistic data association,” in *1980 19th IEEE Conference on Decision and Control including the Symposium on Adaptive Processes*, Dec. 1980, pp. 807–812. doi: 10.1109/CDC.1980.271915.
[42] Y. Bar-Shalom and E. Tse, “Tracking in a cluttered environment with probabilistic data association,” *Automatica*, vol. 11, no. 5, pp. 451–460, Sep. 1975, doi: 10.1016/0005-1098(75)90021-7.
[43] S. Hiscocks *et al.*, *Stone Soup: No Longer Just an Appetiser*. (2023). Python. doi: 10.23919/FUSION52260.2023.10224185.
[44] K. Bernardin and R. Stiefelhagen, “Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics,” *EURASIP J. Image Video Process.*, vol. 2008, no. 1, Art. no. 1, Dec. 2008, doi: 10.1155/2008/246309.
[45] P. Votruba, R. Nisley, R. Rothrock and B. Zombro, “Single Integrated Air Picture (SIAP) Metrics Implementation.” Accessed: Dec. 07, 2024. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA397225

[46] P. C. Lusk and R. W. Beard, "Visual Multiple Target Tracking From a Descending Aerial Platform," in *2018 Annual American Control Conference (ACC)*, Jun. 2018, pp. 5088–5093. doi: 10.23919/ACC.2018.8431915.

[47] J. Li and H. Li, "Transformer-Based Multi-object Tracking in Unmanned Aerial Vehicles," in *Pattern Recognition and Computer Vision*, Q. Liu, H. Wang, Z. Ma, W. Zheng, H. Zha, X. Chen, L. Wang, and R. Ji, Eds., Singapore: Springer Nature, 2024, pp. 347–358. doi: 10.1007/978-981-99-8537-1_28.

[48] M. Yao, J. Wang, J. Peng, M. Chi, and C. Liu, "FOLT: Fast Multiple Object Tracking from UAV-captured Videos Based on Optical Flow," in *Proceedings of the 31st ACM International Conference on Multimedia*, in MM '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 3375–3383. doi: 10.1145/3581783.3611868.

[49] F. Sivrikaya and B. Yener, "Time synchronization in sensor networks: a survey | IEEE Journals & Magazine | IEEE Xplore." Accessed: Dec. 07, 2024. doi: 10.1109/MNET.2004.1316761

[50] M. H. Ko, G. West, S. Venkatesh, and M. Kumar, "Online Context Recognition in Multisensor Systems using Dynamic Time Warping | IEEE Conference Publication | IEEE Xplore." Accessed: Dec. 07, 2024. doi: 10.1109/ISSNIP.2005.1595593

[51] M. B. Lyons, D. A. Keith, S. R. Phinn, T. J. Mason, and J. Elith, "A comparison of resampling methods for remote sensing classification and accuracy assessment," *Remote Sens. Environ.*, vol. 208, pp. 145–153, Apr. 2018, doi: 10.1016/j.rse.2018.02.026.

[52] F. Baccelli and A. M. Makowski, "Queueing models for systems with synchronization constraints," *Proc. IEEE*, vol. 77, no. 1, pp. 138–161, Jan. 1989, doi: 10.1109/5.21076.

[53] D. Bannach, O. Amft, and P. Lukowicz, "Automatic Event-Based Synchronization of Multimodal Data Streams from Wearable and Ambient Sensors," in *Smart Sensing and Context*, P. Barnaghi, K. Moessner, M. Presser, and S. Meissner, Eds., Berlin, Heidelberg: Springer, 2009, pp. 135–148. doi: 10.1007/978-3-642-04471-7_11.

[54] C.-H. Hsiao, S. Narayanasamy, E. M. I. Khan, C. L. Pereira, and G. A. Pokam, "AsyncClock: Scalable Inference of Asynchronous Event Causality," *SIGPLAN Not*, vol. 52, no. 4, pp. 193–205, Apr. 2017, doi: 10.1145/3093336.3037712.

[55] B. Kragl, S. Qadeer, and T. A. Henzinger, "Synchronizing the Asynchronous," in *29th International Conference on Concurrency Theory (CONCUR 2018)*, S. Schewe and L. Zhang, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 118. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, p. 21:1-21:17. doi: 10.4230/LIPIcs.CONCUR.2018.21.

[56] C. Schranz, S. Mayr, S. Bernhart, and C. Halmich, "Nearest advocate: a novel event-based time delay estimation algorithm for multi-sensor time-series data synchronization," *EURASIP J. Adv. Signal Process.*, vol. 2024, no. 1, p. 46, Apr. 2024, doi: 10.1186/s13634-024-01143-1.

[57] R. Scolati, I. Fronza, N. E. Ioini, A. Samir, and C. Pahl, "A Containerized Big Data Streaming Architecture for Edge Cloud Computing on Clustered Single-Board Devices," in *ResearchGate*, doi: 10.5220/0007695000680080.

[58] B. B. Rad, H. J. Bhatti and M. Ahmadi, "(PDF) An Introduction to Docker and Analysis of its Performance," ResearchGate. Accessed: Dec. 07, 2024. [Online]. Available: https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance

[59] Docker, “ Accelerated Container Application Development.” Accessed: Dec. 07, 2024. [Online]. Available: https://www.docker.com/
[60] NVIDIA, “NVIDIA Embedded Systems for Next-Gen Autonomous Machines,” NVIDIA. Accessed: Dec. 07, 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/
[61] Ultralytics, “ultralytics/docker/Dockerfile-jetson-jetpack5 at main · ultralytics/ultralytics.” Accessed: Dec. 02, 2024. [Online]. Available: https://github.com/ultralytics/ultralytics/blob/main/docker/Dockerfile-jetson-jetpack5
[62] Github, “GitHub Actions documentation,” GitHub Docs. Accessed: Dec. 08, 2024. [Online]. Available: https://docs.github.com/en/actions
[63] N. Bowness, “Publish Latest Docker Images.” Accessed: Dec. 08, 2024. [Online]. Available: https://github.com/nathanbowness/UAV-Object-Tracking/blob/main/.github/workflows/docker-publish-latest.yaml
[64] Docker, “Docker Hub Container Image Library | App Containerization.” Accessed: Dec. 08, 2024. [Online]. Available: https://hub.docker.com
[65] NVIDIA, “NVIDIA Jetson AGX Orin,” NVIDIA. Accessed: Dec. 07, 2024. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
[66] NumPy, “numpy.hamming — NumPy v2.1 Manual.” Accessed: Dec. 08, 2024. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.hamming.html
[67] NumPy, “numpy.fft.fft — NumPy v2.1 Manual.” Accessed: Dec. 08, 2024. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html
[68] Z. Cao, J. Li, C. Song, Z. Xu, and X. Wang, “Compressed Sensing-Based Multitarget CFAR Detection Algorithm for FMCW Radar,” *IEEE Trans. Geosci. Remote Sens.*, vol. 59, no. 11, pp. 9160–9172, Nov. 2021, doi: 10.1109/TGRS.2021.3054961.
[69] SciPy, “Signal processing (scipy.signal) — SciPy v1.14.1 Manual.” Accessed: Dec. 08, 2024. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/signal.html
[70] A. Raza, A. Dhakal, S. Honkanen, and R. Baets, “Glucose sensing using photonics waveguide based evanescent Raman spectroscopy,” ResearchGate. Accessed: Dec. 08, 2024. [Online]. Available: https://www.researchgate.net/publication/325158306_Glucose_sensing_using_photonics_waveguide_based_evanescent_Raman_spectroscopy
[71] DJI, “Buy DJI Mini 3 - DJI Store.” Accessed: Dec. 08, 2024. [Online]. Available: https://store.dji.com/ca/product/dji-mini-3
[72] Ultralytics, “Ultralytics YOLOv8 - NVIDIA Jetson AI Lab.” Accessed: Nov. 23, 2024. [Online]. Available: https://www.jetson-ai-lab.com/tutorial_ultralytics.html
[73] OpenCV, “Home”. Accessed: Dec. 02, 2024. [Online]. Available: https://opencv.org/
[74] Ultralytics, “Ultralytics/YOLOv8 · Hugging Face.” Accessed: Dec. 02, 2024. [Online]. Available: https://huggingface.co/Ultralytics/YOLOv8
[75] V. Mehta, F. Dadboud, M. Bolic, and I. Mantegh, “A Deep Learning Approach for Drone Detection and Classification Using Radar and Camera Sensor Fusion,” in *2023 IEEE Sensors Applications Symposium (SAS)*, Jul. 2023, pp. 01–06. doi: 10.1109/SAS58821.2023.10254123.
[76] P. Emami, P. M. Pardalos, L. Elefteriadou, and S. Ranka, “Machine Learning Methods for Data Association in Multi-Object Tracking,” *ACM Comput Surv*, vol. 53, no. 4, p. 69:1-69:34, Aug. 2020, doi: 10.1145/3394659.

[77] K. Yoon, D. Y. Kim, Y.-C. Yoon, and M. Jeon, “Data Association for Multi-Object Tracking via Deep Neural Networks,” *Sensors*, vol. 19, no. 3, Art. no. 3, Jan. 2019, doi: 10.3390/s19030559.