Lund University
Automatic Control

FRTN70 Project in Systems, Control and Learning
Spring 2024

Formation Control and Coordination of Crazyflie UAVs in Dynamic Environments

Project Plan Crazyflie Group 2

Maria Tavemark[1] Harald Haglund[2] Oskar Stenberg[3] Alexandra Annedotter[4]

Project Advisor: Lara Laban

[1]ma1766ta-s@student.lu.se
[2]ha4384ha-s@student.lu.se
[3]oskar.stenberg@control.lth.se
[4]al0403ka-s@student.lu.se

1 Project Purpose

The goal of this project is to make up to four Crazyflie-UAVs fly as a swarm and keep formations whilst moving as a synchronized unit. This project is an extension to the MSc Thesis (TFRT-6181) [1] by Stevedan Ogochukwu Omodolor, done at the Control Department, Lund University. The idea is to enhance the operational efficiency and coordination of Unmanned Areal Vehicles (UAV) swarms, introducing more complex formations, adaptive real-time path planning, and improved synchronization algorithms.

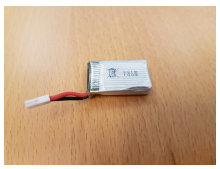
2 Equipment and material

The project will utilize the following hardware components:

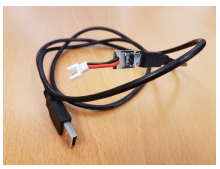
- *Crazyflie 2.1 x 4* (approx. 2200 SEK each): A lightweight open-source flying development platform.
- *Lighthouse positioning deck x 4* (approx. 1200 SEK each): An advanced positioning module that enables high-precision navigation of the UAVs using infrared light houses, crucial for maintaining tight formations and precise maneuvers.
- *Lighthouse V2 base station x 4* (approx. 2900 SEK each): Provides the infrared signals for the positioning decks, ensuring accurate and reliable spatial orientation for the UAV swarm within a designated operational area.
- *Crazyradio PA x 2* (approx. 400 SEK each): A long-range 2.4 GHz USB radio dongle that allows for enhanced communication capabilities with the UAVs, facilitating real-time control and telemetry data transmission over greater distances.
- *Crazyflie battery + charger x 10* (approx. 100 SEK each): Flight time about 20min interchangable batteries.

The appended prices are based on the Bitcraze online store. Some of the equipment can be seen in Figure 1.

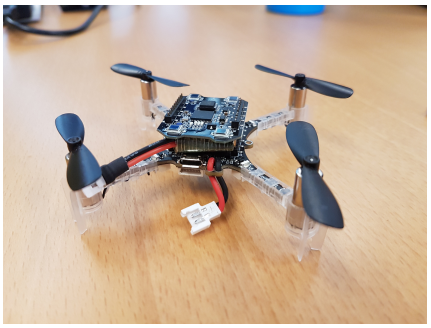
(a) Crazyradio PA.



(b) Crazyflie battery.



(c) Crazyflie charger.



(d) Crazyflie 2.1.

Figure 1: Equipment.

3 Modelling and System Design

We will be writing an implementation of a swarm-controller for the Crazyflie-UAVs, where the controller can run against Robot Operating System (ROS) (or potentially ROS2). ROS2 is a new verison

of ROS that is better for real-time systems and low-latency communication because of changes in the data middleware-layer, and ROS2 also features better multicore-performance which enables it to run more efficiently on modern computers. ROS2 also gives us more flexibility in which language the controller can be implemented in. [2]If we use ROS, we will use the officially supported operating system Ubuntu 20.04 LTS (Long Term Support). With ROS2, we will use the newer version of Ubuntu called 22.04 LTS which is mostly equivalent with Ubuntu 20.04 LTS but features newer kernel versions and upgrades to various modules that should not affect the performance of our controller. [3] The swarm-controller will send commands to individual Crazyflie-controllers that have been implemented in the Bitcraze Crazyflie Python API. The UAVs will be controlled from a central controller on the computer that is running ROS.

The swarm controller is going to be based on flocking-approaches as mentioned in the MSc Thesis (TFRT-6181 [1] and TFRT-6084 [4]) as well as the Bitcraze swarm controller library [5]. The goal is to make the UAVs follow a reference, while simultaneously avoiding other agents and matching the speed of neighbours. In case that the above works as expected, we will also try to implement a max-distance aspect in the controller where the swarm is constrained to have a specific size. This could for example mean that all UAVs in the swarm should be contained within a certain radius around a reference point.

4 Division of labour

The project is split into three main parts: Controller Implementation, Swarm Simulation, and Swarm Integration. They are described in the following subsections in detail.

4.1 Controller Implementation

The Swarm Controller is the main part of the project, and needs to be up and running in some form before the project can continue in any meaningful way. This part includes the actual flocking-based control algorithm.

One of the classic approaches to implement flocking behavior can be modeled after Reynolds' Boids algorithm, which is often used for simulating the flocking behavior of birds. The algorithm combines three simple steering behaviors - separation, alignment, and cohesion, in order to enable complex flocking dynamics.

1. Separation aims to keep simulated birds (boid) apart in order to avoid crowding,

$$F_{\text{sep}}(b) = -\sum_{b' \in N} \frac{P_{b'} - P_b}{\|P_{b'} - P_b\|^2}$$

where F_{sep} is the force of separation applied to a boid b, P_b is the position of boid b, $P_{b'}$ is the position of a neighboring boid b', and N is the set of neighboring boids within a certain distance.

2. Alignment ensures that a boid aligns its direction with the average direction of its neighbors,

$$V_{\text{align}}(b) = \frac{1}{|N|} \sum_{b' \in N} V_{b'} - V_b$$

where V_{align} is the alignment velocity for boid b, V_b is the velocity of boid b, $V_{b'}$ is the velocity of a neighboring boid b', and N is the set of neighbors.

3. Cohesion moves boids towards the average position of their neighbors to keep the flock together,

$$P_{\text{coh}}(b) = \frac{1}{|N|} \sum_{b' \in N} P_{b'} - P_b$$

where P_{coh} represents the cohesion position towards which boid b should move, P_b is the position of boid b, $P_{b'}$ is the position of a neighboring boid b', and N is the set of neighboring boids.

The final steering force applied to each boid can be a weighted sum of these three behaviors,

$$F(b) = w_{\text{sep}} \cdot F_{\text{sep}}(b) + w_{\text{align}} \cdot V_{\text{align}}(b) + w_{\text{coh}} \cdot P_{\text{coh}}(b)$$

where w_{sep}, w_{align}, and w_{coh} are weights that determine the relative importance of separation, alignment, and cohesion, respectively [6].

4.2 Swarm Simulation

During the following part of the project, we will try to integrate and run the Swarm Controller in ROS and visualize using ROS Visualization (Rviz) to see how the controller is behaving. We will spend a lot of time debugging the controller and changing the behaviour of the swarm during this phase. The work here can start at the same time as the controller implementation as we need to get a working ROS and Rviz-system that can simulate the behaviour of the UAVs before the controller is ready for testing.

4.3 Swarm Integration

Finally, after the previously described integration is completed, we can start integrating the swarm controller with the physical UAVs. We still plan to have ROS and rviz running as a base-layer, but to integrate the physical radios and UAVs into the system. This might require some additional coding, but should not be advanced since the Bitcraze library already offers most (if not all) of the functionality we need.

5 Time Plan

5.1 Subtasks

- Code centralized controller in python using Bitcraze library. **Estimated deadline: April 26**
- Simulate one agent in ROS. **Estimated deadline: April 26**
- Fly one agent in lighthouse using ROS. **Estimated deadline: May 3**
- Simulate all agents in ROS. **Estimated deadline: May 17**
- Fly all agents in lighthouse using ROS. **Estimated deadline: May 24**

5.2 Important dates

- Mar 25 - Hand in project plan.
- Mar 27 - Git repo in shape.
- Apr 15 - Self-evaluation 1.
- May 3 - Preliminary report.
- May 7 - Preliminary report - peer review.
- May 27 - Final report.
- May 29 - Self-evaluation 2.
- May 29 - Final report - peer review.
- Jun 4 - Revised final report.

5.3 Gantt Chart

We have formalized the time plan as a Gantt chart. The major tasks can be seen plotted in Figure 2. See updated Gantt chart in Figure 3.

Fixed tasks:	Finished:	Vecka 12	Vecka 13	Vecka 14	Vecka 15	Vecka 16	Vecka 17	Vecka 18	Vecka 19	Vecka 20	Vecka 21	Vecka 22	(Vecka 23)
Preliminary Project Plan	☑	March 22											
Revised project plan (25/3)	☐		March 25										
Git repo in shape (27/3)	☑		March 27										
Self evaluation 1 (1/4)	☐			April 1									
Preliminary report (3/5)	☐							May 3					
Preliminary report per review (7/5)	☐								May 7				
Final report (27/5)	☐											May 27	
Self evaluation 2 (29/5)	☐											May 29	
Final report peer review (29/5)	☐											May 29	
Revised final report (4/6)	☐												June 4
Subtasks:													
Code centralized controller in python using Bitcraze library (26 April)	☐												
Simulate one agent in ROS. (26 April)	☐												
Fly one agent in lighthouse using ROS.	☐												
Simulate all agents in ROS.	☐												
Fly all agents in lighthouse using ROS.	☐												
				Omtenta-P	Omtenta-P								

Figure 2: The Gantt chart describing the work flow of our project.

Fixed tasks:	Finished:	Vecka 12	Vecka 13	Vecka 14	Vecka 15	Vecka 16	Vecka 17	Vecka 18	Vecka 19	Vecka 20	Vecka 21	Vecka 22	(Vecka 23)
Preliminary Project Plan	☑	March 22											
Revised project plan (25/3)	☑		March 25										
Git repo in shape (27/3)	☑		March 27										
Self evaluation 1 (1/4)	☑					April 15							
Preliminary report (3/5)	☐							May 3					
Preliminary report per review (7/5)	☐								May 7				
Final report (27/5)	☐											May 27	
Self evaluation 2 (29/5)	☐											May 29	
Final report peer review (29/5) + artifact upload	☐											May 29	
Revised final report (4/6)	☐												June 4
Subtasks:													
Code centralized controller in python using Bitcraze library (26 April)	☑												
Simulate one agent in ROS. (26 April)	☑												
~~Fly one agent in lighthouse using ROS.~~	☐												
Simulate all agents in ROS.	☑												
Fly all agents in lighthouse using ROS.	☐												
Update project plan for the preliminary report	☑												
Extend code with A* path planning algorithm	☑												
Improve path visualization in ROS2	☐												
Display forces in ROS2-visualizations	☐												
Add obstacles to path planning algorithms	☐												
Add obastcles to ROS2-visualization.	☐												
Add emergency landings when certain conditions are not met.	☐												
Try implemenitng RRT*-algorithm	☐			Omtenta-P	Omtenta-P								

Figure 3: The UPDATED Gantt chart describing the work flow of our project.

5.4 Risk Assessment

- Since we are bound to a small amount of UAVs and spare parts, a risk that follows is that the destruction of UAVs or components can lead to limited (or no) testing. We are approaching this risk by being extra careful when flying the UAVs and simulating the flights in a virtual environment (rviz) before performing any real flights.
- Another risk with this project is that we might have trouble synchronizing ROS with our controller. We believe this could be a risk specifically because not all group members have previous experience in ROS. This risk could lead to delays in the time-plan and insufficient time to finish the other sub-tasks. We are tackling this by trying to stick to our deadlines in the gantt chart as much as possible.

6 Update to the Project Plan

We have successfully and solely been using ROS2 for our project and will thus not have to revert to ROS. We have implemented all code needed for basic swarm control, which includes the Boid algorithm and A*-path planning with integrated obstacle avoidance. Overall, simulations in ROS2 have been quite successful. For instance, we have managed to make individual UAVs follow calculated paths and avoid obstacles. Moreover, we have also been successful in making the UAVs fly, and were therefore able to verify that our code for emergency landing was sufficient and functional. However, despite these achievements, testing has not been as successful.

Many hours have been spent on debugging without much improvement. It was later discovered that the Lighthouse system functions poorly and that the Crazyflie-firmware has unresolved bugs. We are now waiting for new UAV decks to arrive, since the current decks could be causing errors.

In terms of obstacle avoidance, our aim was to have the swarm split up when passing an obstacle instead of keeping all UAVs in the original group formation for this event. However, when trying the code in the simulations, a deadlock situation was encountered. This incident was caused by our path planning method and how the Boid forces had been applied. Due to time constraints, we have decided to keep things simple and instead focus on getting the UAVs to follow an obstacle-free path while keeping the formation of a swarm. In short, the core basics need to be in place before we implement anything else. Once a reliable base has been established, we will start investing more time in implementing obstacle avoidance again and also employ fancier path planning algorithms such as Rapidly-Exploring Random Trees (RRT*).

See our updated Gantt chart in Figure 3 for an overview of how we plan to spend the next few weeks in the course. Apart from what has been mentioned above, we have also decided to add more safety to our UAV control, and improve visualization in our ROS2 simulations.

6.1 Radio Limitations

During our initial physical testing, we came across latency issues where the position estimates would experience jitter and delays up to a few seconds. This would lead to Crazyflies crashing and irregular behaviour when the Crazyflies and our swarm controller got out of sync. The swarm controller would get an old measurement from one of the UAVs and generate a new control signal for it based on stale data. This would in turn cause crashes. The delays in communication would also lead to Crazyflies crashing since their internal supervisors would go into a locked panic state due to the UAV not receiving a signal from the swarm controllers within a specified window of time. Despite being documented in the Bitcraze overview of the Commander Framework, the exact windowsize was only found to be 500ms after consulting the source code of the Crazyflie firmware (the file supervisor.c contains the definition of COMMANDER_WDT_TIMEOUT_STABILIZE as 500ms). [7] [8]

The high latencies and delays only occurred with more than one Crazyflie active, and got worse as the number increased. According to the work done by Green and Månsson, our desired polling rate of 100 Hz is too much for the radio to handle despite the 2Mbit/s transmission speed. When using their proposed ideal sampling rate of 20 Hz, the behaviour of the Crazyflies improved drastically. This initial packet loss is illustrated by the graph in Figure 4 [4].

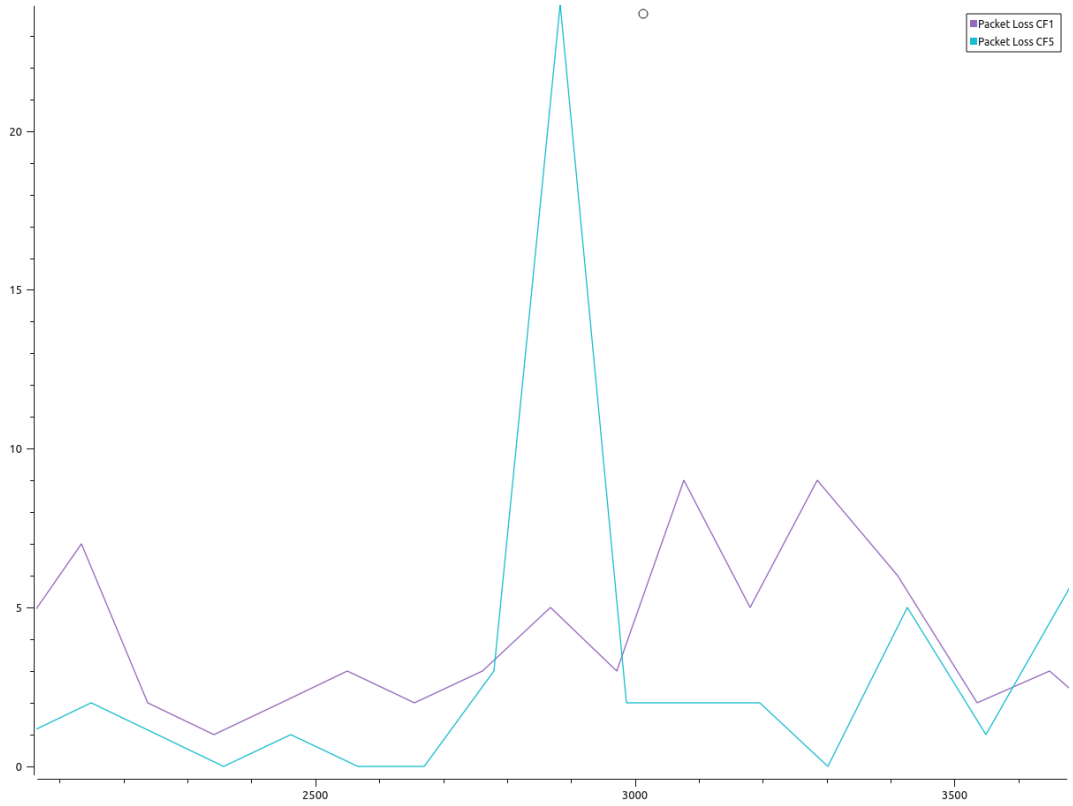


Figure 4: Packetl loss with polling at 100 Hz

References

[1] Stevedan Ogochukwu Omodolor. *Distance and orientation-based formation control of UAVs and coordination with UGVs*. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9096642&fileOId=9096646.

[2] Arun Venkatadri. *Ros 1 vs Ros 2 what are the biggest differences?* Apr. 2023. URL: https://www.model-prime.com/blog/ros-1-vs-ros-2-what-are-the-biggest-differences.

[3] Lukasz Zemczak. Feb. 2024. URL: https://discourse.ubuntu.com/t/jammy-jellyfish-release-notes/24668.

[4] Pontus Månsson Sebastian Green. *Autonomous control of unmanned aerial multi-agent networks in confined spaces*. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8991431&fileOId=8991432.

[5] *Bitcraze swarm documentation*. URL: https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/api/cflib/crazyflie/swarm/.

[6] Craig W. Reynolds. "Flocks, Herds and Schools: A Distributed Behavioral Model". In: *SIGGRAPH Comput. Graph.* 21.4 (1987), pp. 25–34. DOI: 10.1145/37402.37406. URL: https://dl.acm.org/doi/10.1145/37402.37406.

[7] BitCraze AB. *The Commander Framework*. https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/commanders_setpoints/. Accessed: April 27, 2024. 2023.

[8] Bitcraze AB and contributors to the open-source project. *Crazyflie-firmware GitHub Repository*. https://github.com/bitcraze/crazyflie-firmware. Accessed: April 27, 2024. 2021.

Formation Control and Coordination of Crazyflie UAVs using A* Path Planning in Dynamic Environments

Alexandra Annedotter[1] **Harald Haglund**[2] **Oskar Stenberg**[3] **Maria Tavemark**[4]

[1]al0403ka-s@student.lu.se [2]ha4384ha-s@student.lu.se [3]oskar.stenberg@control.lth.se
[4]ma1766ta-s@student.lu.se

Abstract: The rapid evolution of Unmanned Aerial Vehicles (UAVs) demands advanced control and coordination strategies, particularly in dynamic environments. This paper investigates the formation control and coordination of Crazyflie UAVs, aiming to enable synchronized flight in a swarm while maintaining desired swarm sizes. Utilizing the A-Star (A*) algorithm for global path planning, we plan the shortest path through a previously unknown environment. Simultaneously, Reynolds' Boids algorithm is adapted to govern swarm interactions through separation, alignment, and cohesion behaviors, ensuring fluid collective movement. Our approach integrates these algorithms within a simulated environment that mirrors real-world dynamics, using Software in the Loop (SIL) for simulations of UAV behaviour. The experimental setup includes a swarm of up to five Crazyflie UAVs navigating through generated paths and avoiding virtual obstacles, facilitated by real-time data integration and ROS2-based communication. Initial results demonstrate the system's robustness in maintaining formation integrity under varying conditions and its capabilities of static obstacle avoidance. The experimental tuning of Boid-parameters showed optimal values that balance swarm cohesion and individual UAV autonomy.

1. Introduction

The necessity and desire to effectively manage and coordinate Unmanned Aerial Vehicles (UAVs) in dynamic environments is growing every day, due to the evolving landscape of aerial robotics. Therefore, the purpose of this report is to investigate how to best make up to five Crazyflie as a swarm whilst keeping the desired formation when moving as a synchronized unit. A fundamental aspect of this is path planning, which can be either global or local. One of the algorithms that can be used to implement global path planning is the A* algorithm, and in recent years was found to be very successful when it comes to swarm control [1].

For our paper we have chosen this path planning algorithm, as it is a widely used method in regards to finding the shortest path from start to target node in similar systems, first introduced in the following paper [2]. The A* algorithm is an heuristic algorithm, meaning that there is no guarantee that the algorithm finds the shortest path in all instances. Instead, it employs a best-search-strategy and uses an heuristic function to estimate the cost between the current node and the goal. In allowing the heuristic function to guide the search, it is possible to prioritize nodes that seem most likely to be at a shorter distance, in terms of steps, in relation to the goal. As a result, it is likely that the goal can be reached in fewer steps with this algorithm as opposed to other algorithms, such as the Dijkstra algorithm, that does not use heuristics, [3] is a paper that cleary demonstrates this claim.

Since all experiments will be conducted in a dynamic environment, it is appropriate to investigate the spatial and dynamic constraints as demonstrated in paper [4]. Some dynamic constraints that should be taken in to consideration in regards to our crazyflie swarm include maximum speed, acceleration and turning capabilities as these aspects affect the safety and accuracy of the pathplanning. Thus, ensuring each crazyflie in the swarm has a feasible and safe route to the goal node without collisions or crashes caused by physical limitations. The Crazyflie 2.0 nano-quadcopter is described as a highly versatile and accessible platform for UAV research and educational purposes. It features a light and compact design and is ideal for experiments involving UAV dynamics, control systems and and swarm behavior [5].

Considering the dynamic and spatial constraints critical for ensuring safe and effective path planning in a dynamic environment [4], the Crazyflie 2.0 nano-quadcopter can be refered to as an ideal research tool, as it is highly adaptable in UAV dynamics, control systems, and swarm behavior experiments. This adaptability aligns well with the methodologies from "Swarm Intelligence for Autonomous UAV Control" [6], this paper showcases an enhanced UAV fleet autonomy and efficiency through advanced, real-time decision-making and swarm intelligence principles. Furthermore, in the article,

innovative methodologies that enable UAVs to autonomously adapt and optimize flight patterns are discussed as well as decision-making processes in real-time, mimicking the collaborative behaviors seen in natural swarms.

The following paper serves as an extension to the MSc Thesis dabbling in formation control [7], further developing the concepts and methodologies previously explored. The idea is to enhance the operational efficiency and coordination of Unmanned Areal Vehicles (UAV) swarms, introducing more complex formations, adaptive real-time path planning, and improved synchronization algorithms.

This paper is structured in the ensuing manner first a description of the theory is given, which consists of control strategy, path planning and software in the loop. In the next section, swarm architecture is explained, both in terms of software and the physical system. Thereafter, the setup for simulations and planned experiments are described, followed by simulation results and physical experiments and the finishing segment that consists of a conclusion, where considerations for future work is explored.

2. Theory

In this section we present the theoretical formulations used in this paper. It is split up in three parts, consisting of control strategy, path-planning, and software in the loop (SIL).

2.1 Preliminaries

To initiate motion in an Unmanned Aerial Vehicle (UAV), a force is applied, enabling its propulsion and maneuverability. . We can imagine the motion of a single UAV as the sum of all force vectors acting on it, where each UAV has two force vectors [1] applied: One representing the movement towards our target, and another to keep the formation with the other UAVs. The creation of these forces are essentially what we aim to define inside the path-planning and control strategy parts respectively.

2.2 Control strategy

We seek a mathematical definition of how the UAVs should move relative to each other as a part of the swarm. A flocking behavior model can be created based on Reynolds' Boids algorithm, which is commonly used for simulating the flocking behavior of birds [8]. The algorithm combines three simple steering behaviors - separation, alignment, and cohesion, in order to enable complex flocking dynamics.

[1] These forces refer to the internal ones, constructing the core of the movement. In some cases there are also external forces that act on the UAVs, such as in the case of obstacles.

1. Separation aims to keep simulated birds (boid) apart in order to avoid crowding,

$$F_{\text{sep}}(b) = -\sum_{b' \in N} \frac{P_{b'} - P_b}{\|P_{b'} - P_b\|^2} \tag{1}$$

where F_{sep} is the force of separation applied to a boid b, P_b is the position of boid b, $P_{b'}$ is the position of a neighboring boid b', and N is the set of neighboring boids within a certain distance.

2. Alignment ensures that a boid aligns its direction with the average direction of its neighbors,

$$V_{\text{align}}(b) = \frac{1}{|N|} \sum_{b' \in N} V_{b'} - V_b \tag{2}$$

where V_{align} is the alignment velocity for boid b, V_b is the velocity of boid b, $V_{b'}$ is the velocity of a neighboring boid b', and N is the set of neighbors.

3. Cohesion moves boids towards the average position of their neighbors to keep the flock together,

$$P_{\text{coh}}(b) = \frac{1}{|N|} \sum_{b' \in N} P_{b'} - P_b \tag{3}$$

where P_{coh} represents the cohesion position towards which boid b should move, P_b is the position of boid b, $P_{b'}$ is the position of a neighboring boid b', and N is the set of neighboring boids.

The final steering force applied to each boid can be a weighted sum of these three behaviors,

$$F(b) = w_{\text{sep}} \cdot F_{\text{sep}}(b) + w_{\text{align}} \cdot V_{\text{align}}(b) + w_{\text{coh}} \cdot P_{\text{coh}}(b) \tag{4}$$

where w_{sep}, w_{align}, and w_{coh} are weights that determine the relative importance of separation, alignment, and cohesion, respectively.

2.3 Path-planning

The goal is to present an algorithm that generates a path of target points that the swarm should follow. We want our swarm to maintain its formation and because of this, a single path should be created for the average point of our swarm to follow. Once a path of target points has been generated for our swarm, we can proceed to apply the same force from our average swarm point to our target for all of our UAVs. This will result in the entire swarm moving in a formation. The entire process can be visualized in Figure 1. We chose the A* algorithm for this task due to its efficiency in finding the shortest path between two points. By employing A*, we also ensure that the generated path not only leads the swarm towards its goal but also avoids potential obstacles.

The A* search algorithm To use the A* algorithm, we first need to treat the 3d space as a grid where each cell corresponds to a preset dimension (10x10x10 mm in our case). Because A* is a best-first search algorithm, it uses both the cost to reach a node from the start node and the heuristic estimate of the cost

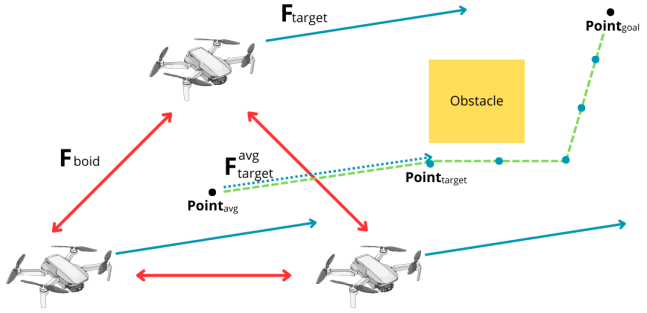


Figure 1. Showcases the applied Boid and target forces (red and blue arrows respectively) and desired path (dotted green line) for our swarm. The target points (blue points) gets generated through our path-planning algorithm, which in turn allows us to generate the resulting target forces. Note that the external forces generated from the obstacle are not visualized in this image.

from that node to the goal. The final cost is then calculated by summing up these two costs.

$$f(n) = g(n) + h(n) \tag{5}$$

Where $f(n)$ is the final cost, $g(n)$ is the cost of the path from the start node to node n, and $h(n)$ is the heuristic estimate of the cost from node n to the goal. Note that the heuristic is essentially just an estimate, used to guide the search for the optimal path, and this estimate can be defined in different ways. In ignoring the heuristic term completely we simply arrive at Dijkstras algorithm [9]. A perfect heuristic would in theory give the exact cost, but this is not always possible since a complex heuristic function comes at the expense of computational efficiency. The heuristic function is in our case calculated using the Manhattan distance. It is thus computed by calculating the total number of squares moved in all axes to reach the target square from the current square. We ignore diagonal movement and any obstacles that might be in the way.

$$h = |x_{\text{start}} - x_{\text{destination}}| + |y_{\text{start}} - y_{\text{destination}}| + |z_{\text{start}} - z_{\text{destination}}| \tag{6}$$

In order to explore the algorithm efficiently, we maintain two sets: The open set, and the closed set. The open set contains nodes to be evaluated. Initially, it contains only the start node. The closed set contains nodes that have already been evaluated. The algorithm then repeatedly selects the node in the open set with the lowest f-score, evaluates it, and adds it to the closed set. It then expands the node by considering its neighbors and calculates their f-scores. When the recursion has finished the optimal path has been found. We then loop through the set one last time to retrieve at all the parents. The pseudo code can be seen in 1. Note that if we ever find an obstacle, we simply ignore that path, and if we ever find a new path to a neighbour that's shorter than the old path, the state of that node (f-cost and parent) gets updated.

2.4 Software in loop

This section will be added in the final report

Algorithm 1: A* Search Algorithm

```
OPEN // the set of nodes to be evaluated
CLOSED // the set of evaluated nodes
Add the start node to OPEN;
while OPEN is not empty do
    current ← node in OPEN with the lowest f_cost;
    Remove current from OPEN;
    Add current to CLOSED;
    if current is the target node then
        return;
    end
    foreach neighbour of current do
        if neighbour is not traversable or neighbour
         is in CLOSED then
            continue;
        end
        if new path to neighbour is shorter OR
         neighbour is not in OPEN then
            Set f_cost of neighbour;
            Set parent of neighbour to current;
            if neighbour is not in OPEN then
                Add neighbour to OPEN;
            end
        end
    end
end
```

3. Swarm Architecture

3.1 Physical System

We utilize Bitcraze's Crazyflie UAVs along with their Lighthouse system in our project. The UAVs themselves lack intercommunication and have no awareness of their own positioning relative to the world. Therefore we need the Lighthouse system. The Lighthouse system employs infrared light to track the positioning data of each UAV within a confined space, which is then transmitted via the Crazy RealTime Protocol (CRTP) to the respective UAVs. This data is further relayed to our computer, where our UAV controller, implemented in ROS2 using Python code and Bitcraze's Crazyflie libraries, is situated. The Crazyflie UAVs are equipped with onboard sensors, including accelerometers, gyroscopes, and barometers, providing data on acceleration, orientation, and altitude, respectively. This sensor data is also transmitted to our UAV controller. The UAV controller issues real-time commands to each UAV for movement control, and these commands are then processed by the UAVs' internal PID controllers (which we do not have access to). Our UAV controller is designed to coordinate the UAVs to perform different tasks, including formation flying, path following and obstacle avoidance, together as a swarm, and employs established algorithms such as the A* algorithm and Reynold Boid Flocking Algorithm to orchestrate these tasks.

3.2 Software

ROS2 Integration At the core of our architecture lies the Robot Operating System 2 (ROS2). It is an open-source framework tailored for robotics applications and works by enabling communication between different components of a robotic system through nodes and topics. Nodes are software modules that perform specific tasks, such as controlling a robot or processing sensor data. Topics are channels for communication between nodes, where messages containing data are sent and received. Publishers send messages on topics, while subscribers receive and process these messages.

The communication network between nodes and topics of our UAV-controller is illustrated in Figure 3. In our UAV-controller, we have one node (called a "frame listener" in Figure 3) for each UAV. This node listens to the topics with information about positioning and UAV status. This node also enables publishing to the topics from the crazyflie server node that enables control of the UAV. The swarm controller node constitutes the centralized controller, and carries out operations that affect the whole swarm as well as handle safety-related aspects such as collision avoidance and battery protection. The swarm controller node publishes the average position of the swarm, control signals, actual path taken, and the desired paths, to relevant topics.

Crazyswarm2 Project The base of our swarm system will be built upon the Crazyswarm2-project that provides us with ROS2-nodes that communicate with the Crazyflies and publishes information such as positioning, battery status and any other information from the UAV that we wish to receive to relevant ROS2-topics. This ROS2-node also provides topics that can be published to in order to control the positioning of the UAV. In addition, Crazyswarm2 also has a simulation-mode where the python-bindings from the Crazyflie firmware can be used to simulate the UAVs and their behaviour [10] [11].
Some of the classes that we have implemented in order to implement the swarm controller include Crazyflie, Controller and GraphicsHandler. The main purpose of the Crazyflie class is to control and manage the individual UAVs within the swarm, which is used when implementing swarm controlling algorithms and facilitates communication between the central controller and the UAVs. This class provides functionality in terms of goal setting, position tracking and status monitoring. The Controller class makes up the control unit for a crazyflie swarm. In short, it functions as a central command hub that is used for managing all aspects associated with swarm operations. For example, the class handles both the swarm's safety management and path planning. The third class, GraphicsHandler, serves as the graphical interface for swarm operations and provides visual feedback in real time of various swarm behaviours. As a result, this class is useful for debugging, monitoring and analytical purposes in regards to swarm control.

4. Simulation Setup and Experiments

Employing Software-in-the-Loop-testing (SIL) greatly facilitates the process of developing and testing the swarm controller. This approach enables remote testing, as it involves running the flight control software in a simulated environment as opposed to directly on the hardware. The simulated environment is then integrated with ROS2, which creates a flexible platform for testing and debugging of the swarm controller. The behavior of the crazyflies can then be observed and visualized in real time when using RViz, i.e ROS Visualization, which is a ROS graphical interface that enables visualization.

In order to conduct the experiments, proper simulation is needed. To properly visualize what is going on in our simulation and experiments, we need to show our different control signals and sensor values. Two tools that can be used to aid in this are PlotJuggler and rqt_graph. PlotJuggler is a tool that allows plotting signals from ROS2-topics into graphs and performing data-analysis on them. Rqt_graph is used to visualize the different ROS2-nodes and topics in our system and how they communicate with one another.
For visualizing the UAVs movements and control signals, ROS2 Visualization (RViz2) will be used in conjunction with PlotJuggler and rqt_graph. RViz2 is the primary visualisation tool used, and displays the positions of all waypoints, UAVs, swarm average point and the rotation of all relevant entities as well as the forces applied to each UAV. This tool setup is used both for simulation as well as actual experiments, where the only systematic difference is the use of the Crazyflie firmware with SIL or the Bitcraze radio in conjunction with the Crazyflies [12], [13], [14].

PlotJuggler is used to generate plots for all data timeseries that the UAVs and our controller publish to ROS2, and can thus be a valuable tool for debugging control signals. It can also be used to record data from all available ROS2 topics and replay the data after a simulation or actual flight is finished, to enable closer investigation of potential issues in swarm behaviour. The republished positioning data can also be visualized in RViz2 to properly replay the experiments.

Some of the planned experiments are presented in the ensuing subsections.

4.1 Tuning of boid forces to achieve certain average distances (swarm sizes) for various amounts of boids

To execute this experiment, PlotJuggler will be setup with a curve which followed the average distance between boids. The boids will follow the same path during each experiment run and have roughly the same starting positions, but otherwise left untouched.

To tune the size of the swarm, the weights of the different boid-forces will be changed to find the optimal values to fulfill our goal of achieving swarm-sizes according to table 1. If the desired swarm size causes instabilities, we wish to minimize them as well so our system is mostly free from oscillations.

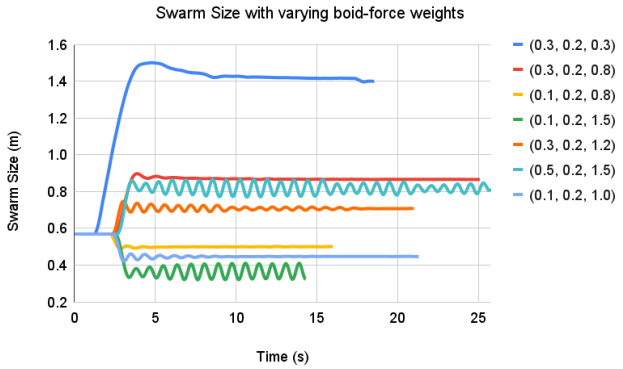


Figure 2. Average distance between three boids when following path. Line titles are on the format ($w_{separation}$, $w_{alignment}$, $w_{cohesion}$).

4.2 Object avoidance of virtual objects whilst maintaining swarm formation

This section is in the works for the final version of the report.

4.3 Performance testing - investigating successful pathfollowing with various amount of boids

This section is in the works for the final version of the report.

5. Simulation Results

For all simulated experiments, the boids got initial positions as seen in table 2 and followed a path from the initial average swarm position to a point with offset (m) [0.5, 0.5, 0.5] and subsequently from that position to a point with offset (m) [-1.5, -1.5, -1.0] after the initial takeoff to a height of 1.0 meters.

5.1 Tuning of boid forces to achieve certain average distances (swarm sizes) for various amounts of boids

The different weights were adjusted for each number of boids until the desired swarm-size was reached, after which the experiment was repeated to verify the results. See tables 3, 4, 5, 6 to see the different values tried. In Figure 2, the curves from which the average swarm size was read out for each set of weights is displayed. When generating the average, the initial liftoff-phase and the concluding landing were trimmed from the dataset - thus avoiding errors that stem from the inactivity of boid-forces during these phases.

A swarm size of two boids got a distance of 0.35 meters when the parameters (0.1, 0.2, 0.8) were used for $w_{separation}$, $w_{alignment}$, $w_{cohesion}$. Although smaller than wanted, these values were stable and found to result in desired swarm behaviour. Table 3 shows some of the tried values of the parameters and their resulting swarm sizes.

Table 1. Swarm-sizes with different amounts of boids

Boids	2	3	4	5
Distances (m)	0.4	0.5	0.7	0.8

With three boids, the optimal weights were found to be (0.1, 0.2, 1.0). These weights resulted in a swarm size of 0.45 meters, which is slightly smaller than wanted - but stable with very few oscillations after initial startup. The initial oscilations lasted for about two seconds after the boid-forces started being applied. There is a DNF (Did Not Finish) value in the table 4, which is due to a collision between two UAVs when using that set of weights.

When four boids were in use, the optimal weights were found to be (0.15, 0.5, 1.0) which resulted in a swarm size of 0.67 meters. This is roughly the same weights as for three boids, but with the $w_{alignment}$ significantly larger. This helped reduce in oscillations as the swarm moved more cohesively, which was significantly more noticable with four boids and a lower $w_{alignment}$ than with three boids. See table 5 for the exact results.

The transition from four boids to five did not produce a noticable effect in the swarm size, but the swarm was found to oscillate less in size when the weights (0.17, 0.5, 1.0) were used. Table 6 contains the exact results for this test.

5.2 Object avoidance of virtual objects whilst maintaining swarm formation

This section is in the works for the final version of the report.

5.3 Performance testing - investigating successful pathfollowing with various amount of boids

This section is in the works for the final version of the report.

6. Physical Experiments

This section is in the works for the final version of the report.

6.1 Radio limitations

During initial tests, latency issues caused jitters and delays in position estimates, leading to crashes and irregular behavior as the swarm controller relied on outdated sensor measurements. Green and Månsson found that reducing the polling rate from 100 Hz to 20 Hz significantly improved the performance, addressing issues that worsened with increasing numbers of active Crazyflies [15].

7. Conclusion and Future Research

One of the biggest disadvantages with our current approach is that the simulation and real experiments cannot run at the same time. To enable the use of strategies for fault and attack detection in this Cyber-Physical System, you could employ the use of a digital twin if the simulation ran at the same

time as the actual experiments. This is investigated in "Digital Twins for Cyber-Physical Systems Security: State of the Art and Outlook", and could be used for Intrusion Detection and detecting anomalies in hardware and configuration [16].

We have also considered adding more complex path planning algorithms such as Rapidly-Exploring Random Trees (RRT*) which could potentially enable the swarm to split up to move around objects and alter paths based on various costs on the chosen path [17]. The tuning of the weights for the boid-forces has been performed iteratively and experimentally in this paper, and in an ideal scenario there would have either been a mathematical model for the swarm system that could be used to solve for the weights or some automated testing that found the optimal values based on repeated tests.

References

[1] Y. Z. Zhenjian Yang Ning Li and J. L. "Mobile robot path planning based on improved particle swarm optimization and improved dynamic window approach". *Journal of Robotics* (2023). URL: https://www.hindawi.com/journals/jr/2023/6619841/.

[2] B. R. Peter E. Hart Nils J. Nilsson. "A formal basis for the heuristic determination of minimum cost paths" (1968). DOI: 10.1109/TSSC.1968.300136. URL: https://ieeexplore.ieee.org/document/4082128.

[3] K. Fransen and J. van Eekelen. "Efficient path planning for automated guided vehicles using a* (astar) algorithm incorporating turning costs in search heuristic". *International Journal of Production Research* **61**:3 (2023), pp. 707–725. DOI: 10.1080/00207543.2021.2015806. eprint: https://doi.org/10.1080/00207543.2021.2015806. URL: https://doi.org/10.1080/00207543.2021.2015806.

[4] M. Debord, W. Hönig, and N. Ayanian. "Trajectory planning for heterogeneous robot teams". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 7924–7931. DOI: 10.1109/IROS.2018.8593876.

[5] G. Silano, E. Aucone, and L. Iannelli. "Crazys: a software-in-the-loop platform for the crazyflie 2.0 nano-quadcopter". In: *2018 26th Mediterranean Conference on Control and Automation (MED)*. 2018, pp. 1–6. DOI: 10.1109/MED.2018.8442759.

[6] N. Frantz and N. (U.S. "Swarm intelligence for autonomous uav control /" (2005).

[7] S. O. Omodolor. *Distance and orientation-based formation control of uavs and coordination with ugvs*. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9096642&fileOId=9096646.

[8] C. W. Reynolds. "Flocks, herds and schools: a distributed behavioral model". *SIGGRAPH Comput. Graph.* **21**:4 (1987), pp. 25–34. DOI: 10.1145/37402.37406. URL: https://dl.acm.org/doi/10.1145/37402.37406.

[9] E. W. Dijkstra. "A note on two problems in connexion with graphs". *Numerische mathematik* **1**:1 (1959), pp. 269–271. URL: https://www.cs.yale.edu/homes/lans/readings/routing/dijkstra-routing-1959.pdf.

[10] B. AB and contributors to the open-source project. *Crazyflie-firmware github repository*. https://github.com/bitcraze/crazyflie-firmware. Accessed: April 27, 2024. 2021.

[11] K. M. Wolfgang Hönig and contributors to the open-source project. *Crazyswarm 2 github repository*. https://github.com/IMRCLab/crazyswarm2. Accessed: April 27, 2024. 2021.

[12] OpenRobotics and contributors to the open-source project. *Ros2 visualization (rviz) github repository*. https://github.com/ros2/rviz. Accessed: April 27, 2024. 2022.

[13] D. Faconti and contributors to the open-source project. *Plotjuggler website and documentation*. https://plotjuggler.io. Accessed: April 27, 2024. 2022.

[14] OpenRobotics and contributors to the open-source project. $Rqt_graph - roswiki$. http://wiki.ros.org/rqt_graph. Accessed: April 27, 2024. 2022.

[15] P. M. Sebastian Green. *Autonomous control of unmanned aerial multi-agent networks in confined spaces*. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=8991431&fileOId=8991432.

[16] M. Eckhart and A. Ekelhart. "Digital twins for cyber-physical systems security: state of the art and outlook". In: 2019, pp. 383–412. ISBN: 978-3-030-25311-0. DOI: 10.1007/978-3-030-25312-7_14.

[17] B. Chandler and M. A. Goodrich. "Online rrt* and online fmt*: rapid replanning with dynamic cost". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 6313–6318. DOI: 10.1109/IROS.2017.8206535.

Appendix A

A Comprehensive Overview of the Images and Tables

Table 2. Initial positions with different amounts of boids N denotes the amount of boids, and P_i is the position (x, y) in meters of boid i. Leading zeroes omitted.

N	P_1	P_2	P_3	P_4	P_5
2	(.0, .5)	(-.5, .0)			
3	(.0, .5)	(-.5, .0)	(-.5, .5)		
4	(.0, .5)	(-.5, .0)	(-.5, .5)	(.5, .5)	
5	(.0, .5)	(-.5, .0)	(-.5, .5)	(.5, .5)	(-.5, -.5)

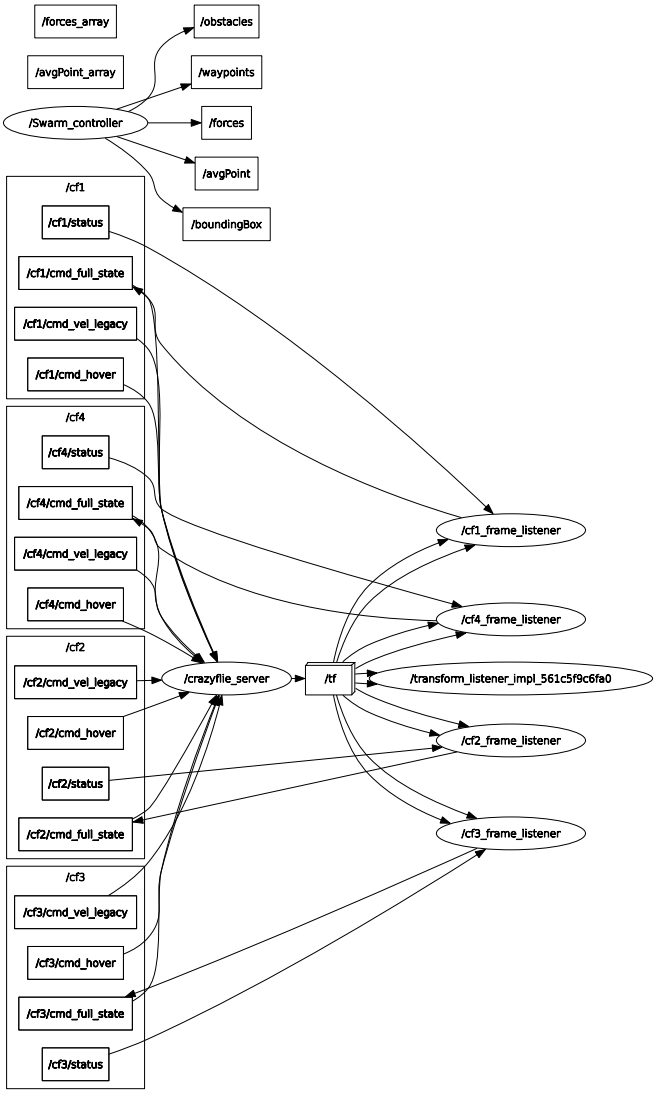


Figure 3. Graph of ROS2-nodes and communication-topics between them

Table 3. Swarm-sizes with two boids and various weights

$w_{separation}$	$w_{alignment}$	$w_{cohesion}$	$\overline{distance}$
0.3	0.2	0.3	1.01
0.1	0.2	0.3	0.56
0.1	0.2	0.8	0.35

Table 4. Swarm-sizes with three boids and various weights

$w_{separation}$	$w_{alignment}$	$w_{cohesion}$	$\overline{distance}$
0.3	0.2	0.3	1.42
0.3	0.2	0.8	0.91
0.1	0.2	0.8	0.56
0.1	0.2	1.5	DNF
0.3	0.2	1.2	0.74
0.5	0.2	1.5	0.81
0.1	0.2	1.0	0.45

Table 5. Swarm-sizes with four boids and various weights

$w_{separation}$	$w_{alignment}$	$w_{cohesion}$	$\overline{distance}$
0.1	0.2	1.0	0.54
0.2	0.05	1.0	0.78
0.2	0.5	1.0	0.76
0.15	0.5	1.0	0.67

Table 6. Swarm-sizes with five boids and various weights

$w_{separation}$	$w_{alignment}$	$w_{cohesion}$	$\overline{distance}$
0.15	0.5	1.0	0.78
0.2	0.5	1.0	0.91
0.17	0.5	1.0	0.75