University of Southern Denmark

Bachelor Project

BSc in Engineering (Robot Systems)
6. Semester spring 2021

Swarming Control of Simulated UAVs Using Nature-inspired Flocking Behaviours

Nicoline Louise Thomsen

nthom18@student.sdu.dk

Supervisor: Ulrik Pagh Schultz
Project period: 01.02.2021 - 01.06.2021

Abstract

The use of drones in various applications are increasing in demand, due to their versatile nature, ease of programming and decreasing cost. Many applications would benefit from the use of swarm robotics in drones, as they are able to gather information from multiple points of view or distribute sub-tasks for faster completion. However the task of controlling multiple actors simultaneously is a challenge.

This report focuses on the use of swarm robotics in action, testing nature inspired algorithms and developing behaviours to complete a simple objective. This places most of the control on-board each individual drone, reducing the complexity of controlling swarming drones. An aspect of this project is to explore a realistic simulation environment, and test the developed swarming behaviours on that platform. Testing on simulated drones rather than physical hardware increases testing capacity and greatly reduces cost. Here the Robot Operating System (ROS) is used to communicate with drones running in Docker, and simulated with the Gazebo dynamic simulation tool.

Reading Guide

Any implemented behaviours, classes, files or commands in the sections will be written `in this font.`

The term 'swarm' and 'flock' are used interchangeably throughout the report.

List of Abbreviations

LiDAR	**Li**ght **D**etection **A**nd **R**anging
SITL	**S**oftware **I**n **T**he **L**oop
UA	**U**nmanned **A**ircraft
UAS	**U**nmanned **A**erial **S**ystem
UAV	**U**nmanned **A**erial **V**ehicle
UML	**U**nified **M**odeling **L**anguage

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Description

Executive summary

Apply simple nature-inspired rules to simulate complex behaviour for obstacle avoidance and route planning, in a swarm of multi rotor UAVs. The drones are to function in a simulated environment set up in ROS on a Linux machine using Gazebo and Docker.

Description

UAS based on swarm robotics has the potential to use smaller and more agile UAVs compared to a UAS with a single, larger drone. The swarm as a whole can collectively offer advanced on-board processing capabilities by use of distributed computing. Using distributed sensing and collective decision making, swarms of small UAVs would be able to better navigate unstructured environments or split a large task into smaller ones, to be handled by individual drones simultaneously. An example is inspection of powerline grids, which is currently handled by a helicopter capturing video of the powerlines, an expensive and demanding solution. Using swarming UAVs, such an inspection task could be completed automatically and efficiently.

A key property of swarming UAVs is that in the event that a UAV is obstructed or destroyed, other drones in the swarm can, because of the redundancy, carry on with the task. Multiple UAVs are however a challenge to control, due to the need to coordinate the movement of a larger swarm and the behaviour of the individual drones. Furthermore, testing drones in a real environment is challenging and time consuming for a test platform, and can lead to extra expenses should the drones crash and need repair or replacement.

The solution and objective of this project is to apply simple nature-inspired rules to simulate complex behaviour for obstacle avoidance and route planning, in a swarm of multi rotor UAVs. With inspiration from the Boids Algorithm which is based on three rules of flocking behaviour, potential objectives is for the swarm to navigate from a starting position and reach a goal without collisions. In addition, the possibility of positioning the individual drones to form specific formations.

The drones are to function in a simulated environment to allow for extensive testing without the risk of damaging hardware. To achieve a realistic test platform for the system the simulations are set up on a Linux machine with ROS, Gazebo, and Docker.

To complete this project first a simulation is to be set up in Gazebo using ROS and Docker, where a small swarm will be simulated to ensure the environment is functional. Then the three rules from the Boids Algorithm will be implemented and tested. Additional rules will be developed and applied, and the system will be tested with different applications like inspection of infrastructure or monitoring natural areas.

Project Objectives

1. Decide on what type of UAV to simulate

2. Learn simulation in ROS

3. Determine the means of information distribution between the drones: what does each drone need to know about the others, and how can they communicate?

4. Implement a use case based on simulated swarming UAVs

5. Implement global navigation of the swarm

6. Implement global control of the swarm for flying in formations

1.2 Three Layer Behaviour Model

The objective of navigating an agent through an environment, can be described by three layers; action selection, steering and locomotion, shown in figure 1.1.

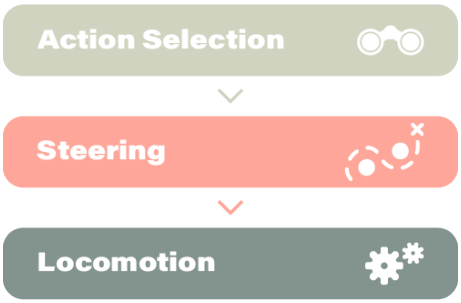


Figure 1.1: The three layers of behaviour.

Craig Reynolds explains this model with a simple example in [1]. Horse-riding cowboys herding a herd of cows, managed by the trail boss. One animal moves away from the rest of the herd, and the trail boss commands the cowboy to go fetch it. This is the action selection, the act of monitoring the state and choosing appropriate action. The cowboy will ride on his horse, steering towards the animal, avoiding obstacles along the way. This is the second layer. Locomotion, the last layer, is the horse itself, and how it is moving its legs to create forward motion.

For a drone this corresponds to a control station selecting an action for the drone to take; follow target, move to point, find object, etc. The steering layer is the behaviours set in place in order to perform an action. If the objective is to move to a specific point, it needs to fly towards that point and fly around obstacles that are in the way. The locomotion is the propellers of the drone spinning at certain speeds to create a movement in a certain direction.

This project will mainly focus on the steering aspect of the behaviour model in scenarios where the action to take is known. The locomotion is handled by the flight controller, in this case the Pixhawk 4, which takes inputs and sends the necessary signals to the speed controllers to perform specific movements. The Pixhawk is used as software in the loop (SITL), meaning the flight controller code is running on the computer, rather than external hardware [2]. The flight controller is not in the scope of this project.

Chapter 2

Background

This section explain the theoretical background for the work done in later sections. Section 2.1 briefly describe flocking behaviours found in nature. Section 2.2 describes Unmanned Aerial Systems and their function, and the different categories that unmanned aircrafs are divided into. Section 2.2.1 goes more in depth about the specific type of aircraft chosen for this project, and the swarm capabilities of drones. Section 2.3 is about the fundamental theory behind the Boids Algorithm, some of the main behaviours applied to the drones in this project. Section 2.4 lists and describes the software tools used to work in the physics based simulation environment.

2.1 Flocking Behaviour in Nature

The phenomenon of flocking is found overall in nature be it flocks of birds or schools of fish. Flocking is to organize into ordered motion, matching the velocity of nearby flock mates and navigating as one with little information about the environment. Animals that inhabit this behaviour greatly varies in size, and are found on land, in water or in the air.

The flock is very fluid in nature, that is it has no rigid structure, and no individual has a specific place in the flock. Depending on the turning of the flock an individual might chance from being in the front to being in the back, edge or middle [3].

Animals rely solely on their senses, mostly sight and feeling, for their ability to flock. Taking inspiration from the nature of flocking behaviours in animals, a control scheme can be implemented on drone platforms to navigate unstructured environments as a flock, that also places most of the decision making on-board each drone. Drones have the advantage over animals that they can communicate detailed information, and base some behaviours of more than just sense alone.

2.2 Unmanned Aerial Systems

An unmanned aerial vehicle (UAV), or unmanned aircraft (UA), is an aircraft either controlled remotely or autonomously, commonly referred to as simply drones. An unmanned aerial system (UAS) is the components that comprises the elements to control and monitor one or multiple drones, without carrying a human operator. In this report it is only relevant to describe the concept of autonomous drones, as drones controlled remotely by human operators are not considered.

A UAS includes the UA itself, the ground control stations and command and control link [4]. As mentioned in section 1.2, this project is concerned with the steering aspect of the three layer behaviour model, and as such focuses on the on-board behaviours of the drones.

UAS was originally developed for military use, but has since expanded to the civilian market, being adopted for a variety of applications. These applications include humanitarian operations like search and rescue or delivery of aid packages. They are also used for traffic control or monitoring fields for agricultural purposes. Drones are considered to be a very versatile robotics platform, because of its adaptability to many different applications.

Different Types of UA platforms exists that typically divides into four categories, each with sub-configurations: Fixed-wing, Rotary-wing, Blimps and Flapping-wing. **Fixed-wing** resemble airplanes the most with unmovable wings, they conventionally requires a runway for take-off and landing, or alternatively a catapult. They are known for their higher speeds and endurance. **Rotary-wing** includes helicopter and multi-rotor configurations, and feature vertical take-off and landing (VTOL). They have the ability to remain stationary mid-air and have high maneuverability. **Blimps** are vehicles lighter than air like balloons. They are large and slow, but have long endurance. **Flapping-wing** have movable wings like those of birds or insects. They also include airplane-like drones which take off vertically and tilt their rotors to fly like a plane. The different types of UAVs are illustrated in figure 2.1.

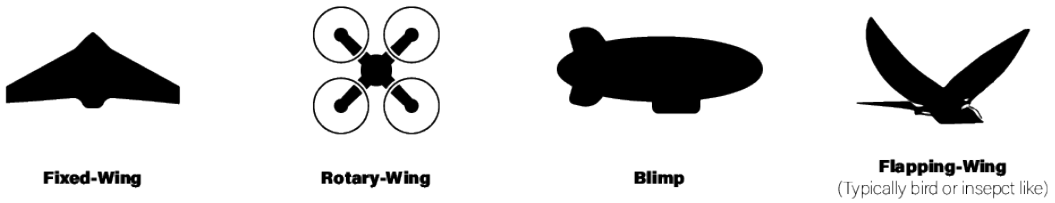


Figure 2.1: Examples of the four different types of UAV.

2.2.1 Swarm Control for Unmanned Aerial Systems

Swarm control means having multiple drones operating simultaneously in the same environment, working on a common task, whether that task be spilt up into sub-tasks, taken on by individual drones or a sub-part of the swarm, or every drone working together on a single objective. As such for the application of swarm control of UAs, the rotary-wing, specifically the multi-rotor UA platform, is ideal because it offers high maneuverability and capability to hover. Other than the fact that these attributes are mandatory for many swarm-related applications, using multi-rotors also reduces the complexity of maneuvering the swarm itself, because it allows drones to stop or move backwards if needed in tight spaces. If a fixed-wing platform was used instead, the route each drone would take has to be quickly and precisely calculated to navigate through narrow passages, which is a huge challenge in unstructured scenarios where the environment is unknown. However, in applications that cover very large areas with no obstacles creating narrow passways, an example is gathering data of the location of icebergs to help ships navigate the northern oceans [5], the fixed-wing configuration could be more beneficial. The higher speeds and endurance providing the advantage.

The advantages and disadvantages of swarm drones over having a system with only one, is application specific. For inspection of power line grids, a swarm proves effective for its ability to cover multiple sub-areas at once, where as for inspection of wind turbines the advantage is insignificant, and instead leads to unnecessary complication.

Navigation and path planning is a fundamental subject for robotics, whether they be stationary robotic arms or flying drones. Many schemes exists to navigate a robot from one configuration state to another. These algorithms include Potential Fields, the Brushfire Algorithm, Voronoi Maps, A*, Cell Decomposition, Rapidly Random Trees and many more.

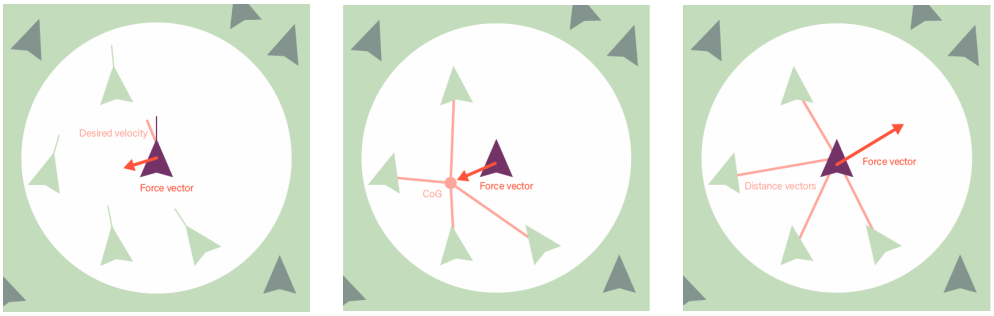
They all however, is best utilized if the environment is known or easily mapped. What happens then, when drones are to complete objectives in unstructured environments, like searching a disaster area or navigating through a forest? This is especially challenging with multiple drones at once. Here, inspiration is drawn from nature, for many animals like insects or birds complete this task daily, seemingly without much thought. In section 2.3, Craig Reynolds observations and resulting algorithm is explained, and in section 3.2 this algorithm is explored as an implementation for controlling drones.

These principles will allow each drone to make individual decisions about where to fly

without knowing the precise end-location of the other drones in the flock. It makes decisions based on the current state, and acts appropriately to any changes, be them in the swarm or the environment. The system to be implemented in the project is dependent on some information being shared between all drones in the flock. This includes the position and velocity of each drone.

2.3 Swarm Behaviour

Craig Reynolds created in 1986 a computer model of coordinated animal motion, to simulate the behaviour of flocks of birds or schools of fish, to be used in computer animation [6]. This 'Boids' algorithm, as he named it, consists of three simple steering behaviours, to model otherwise complex behaviour. These are alignment, cohesion and separation. The principle is illustrated in figure 2.2. The figure shows how an individual boid perceives the world, only accounting for local boids in the flock that are within its perception range, shown by the white circle.



(a) Alignment (b) Cohesion (c) Separation

Figure 2.2: Three rules of the boids algorithm, each applying an acceleration vector (The red arrow).

These steering behaviours, or rules, each describe how individual boids should maneuver in relation to neighboring boids in the flock. Each rule apply an acceleration with direction and magnitude, based on position and velocity vectors, encouraging a boid to steer in a desired direction. The concepts of the boids algoritm in source [1] is described mathematically in the following sections.

Alignment is enabling a boid to align itself with local flock mates in terms of direction and speed. Alignment is shown in figure 2.2a.

The desired velocity is calculated, by averaging the velocities of boids within perception range. The velocity of the current boid is subtracted from this average to compute the steering/force vector. In equation 2.1 the formula for the force the alignment rule applies

is written.

$$\vec{F_a} = \frac{1}{n-1}\sum_{i=1}^{n-1}(\vec{\mathrm{V}}_i) - \vec{V}_{\text{current}} \tag{2.1}$$

$\vec{F_a}$ is the alignment force, n is the number of perceivable boids including itself, $\vec{V_i}$ is the velocity vector of the individual boids in the flock and $\vec{V}_{\text{current}}$ is the velocity vector of the current boid.

Cohesion ensures that local flock mates will form groups, by steering the current boid towards the local center of gravity for the flock. Cohesion is shown in figure 2.2b.

The vector pointing towards the center of gravity is calculated by averaging the positions of local flock mates, and subtracting the position of the current boid, see equation 2.2.

$$\vec{\mathrm{cog}} = \frac{1}{n-1}\sum_{i=1}^{n-1}(\vec{P_i}) - \vec{P}_{\text{current}} \tag{2.2}$$

$\vec{\mathrm{cog}}$ is the vector pointing to the center of gravity, n is the number of boids in the local flock including itself, $\vec{P_i}$ is the position vector of the individual boids in the flock and $\vec{P}_{\text{current}}$ is the position vector of the current boid.

The steering vector is then computed by subtracting the vector to the center of gravity by the current velocity. The formula for the cohesion steering vector is written in equation 2.3.

$$\vec{F_c} = \vec{\mathrm{cog}} - \vec{V}_{\text{current}} \tag{2.3}$$

$\vec{F_c}$ is the cohesion force and $\vec{V}_{\text{current}}$ is the velocity vector of the current boid.

Separation is a counter measure against collision and creates a repulsive force, keeping a certain distance to other boids. Separation is shown in figure 2.2c.

For each neighboring flock mate a repulsive force is calculated by subtracting the position of the current boid with flock mates, and normalizing those vectors. To increase the force for flock mates that are closer, a weighting can be applied, Craig Reynolds has in his paper

suggested a value of $\frac{1}{r}$, where r is the length of the repulsive vector. The repulsive forces are summed and this is the steering force. The formula for the separation force is shown in equation 2.4.

$$\vec{F_s} = \sum_{i=1}^{n-1} \left(\frac{\vec{P}_{\text{current}} - \vec{P_i}}{|\vec{P}_{\text{current}} - \vec{P_i}|^2} \right) \tag{2.4}$$

$\vec{F_s}$ is the separation force, n is the number of perceivable boids including itself, $\vec{P_i}$ is the position vector of the individual boids in the flock and $\vec{P}_{\text{current}}$ is the position vector of the current boid.

The three forces are combined to yield a net force affecting the individual boid. Each rule have an associated weight, adjusting the force exerted from each rule. The formula is seen in equation 2.5.

$$\vec{F}_{\text{net}} = w_a \cdot \vec{F_a} + w_c \cdot \vec{F_c} + w_s \cdot \vec{F_s} \tag{2.5}$$

$\vec{F}_{\text{net}}$ is the net force vector, w is the scalar weighting for each rule.

2.4 Software Development tools

2.4.1 Robot Operating System

The Robot Operating System, or simply ROS, is a framework for software development for robots [7]. The framework is a collection of libraries and tools, providing generic software, that a user has the ability to change with user-written code, enabling the creation of application specific software [8].

The software is open-source, and many developers have contributed packages to expand its capabilities. ROS itself at its core is mostly nodes handling of message passing, and built upon it is a wide selection of packages and tools.

Nodes in ROS is a process that only takes action after registering with the ROS master node, they are executables that are connected to the ROS network [9], using a publisher and subscriber architecture. The nodes created can have different functions, like taking an action based on received information from other nodes, nodes that generate information and sends it to other nodes, or nodes that handles requests for actions from other nodes by sending and receiving [10].

MAVLInk is a messaging protocol created for communication with drones. The protocol is a mix between publish-subscribe and point-to-point protocol [11].

MAVROS is a ROS package that enables communication between computers running ROS, using the MAVLink communication protocol. It essentially is a bridge between the MAVLink protocol and devices running ROS [12].

2.4.2 Simulation in Gazebo

Gazebo is a free 3D simulation environment, that creates realistic conditions for objects put in the world. It is made for simulating populations of robots, in both indoor or outdoor environments. It is a reliable simulation tool, and includes a range of different sensors [13].

It is a very useful tool for testing robots before real deployment. It comes with high performance physics engines and a rendering engine that features high-quality lighting, shadows, and textures. It can generate sensor data and optionally include noise, and the simulation can be expanded with plugins [14]. This simulation is easily integrated with ROS which is optimal for implementation with UAS.

Defining models to be interpreted in the Gazebo engine is simple using the SDFormat and .config files. SDF files is an XML format which describes objects in a simulation, their collision parameters, visuals, physics and control [15].

2.4.3 Docker

"Docker is an open platform for developing, shipping, and running applications." [16]. It is a container based development tool, designed to run and test code that was made on one system on another, regardless of customizations and settings that differ from the original machine. The applications are packaged in a "loosely isolated" environment, that separates the application from the computer's infrastructure. It contains all the packages needed to run an application and its dependencies [17].

Using the `docker build` command, it is easy to create a new Docker image to run applications, and using a `Dockerfile` that contains the information about creating the image. This could be what base image to build upon, example the Python:3 image, the application to be packaged and a `pip` command installing dependencies for the application listed in a text file. Dependencies to include could be numpy, Tkinter for GUI development, or SQL if working with databases. this file also describes which ports needs to be exposed, and what commands should be executed when the container is started. For a Python application this would be a command like `python ./app.py`.

Sharing this image allows other users to run the Python application, even if the needed dependencies are not installed on the local machine [18].

Chapter 3

Analysis

This section analyses applications of interest for the study of drone-swarm technology, and elaborate on the behaviours that are relevant for all applications, and will be implemented in both simulations, the kinematic based and the physics based.

3.1 Application Scenarios

As described in section 2.2.1 the advantages of a drone swarm is best utilized for tasks that needs coverage of larger areas.

Four such scenarios are proposed:

Case a) Surveillance of a large flock of wild animals. This scenario is an open space, but a large area needs to be covered at all times, as the area is not static. Some animals might part from the larger flock for a time, and return again after a while. The swarm needs to adapt to the changing shape of the flock in order to cover everything, while also having the capability to follow those animals who move outside the covered area. In addition, the simulation could implement a battery-life limit of the drones, forcing them to return to a base station for charging. This would mean that the swarm should also be able to handle drones leaving the formation.

Case b) Locating lost people after a disastrous event in urban outdoor environment. This task requires a search of a large space, but once an area has been covered it need not be searched again. Large obstacles like buildings will need to be avoided, but space between obstacles will also be large. The swarm can be divided into sub-groups that each search a delegated area.

Case c) Data collection in the canopies of trees. Obstacles are now not to be only avoided, but are also objects of observation that the drones need to position themselves around. A small group of drones (2-3) will surround a tree and gather data for a while, before moving on to the next undocumented tree.

Case d) Moving a swarm of UAs from point-to-point in an unstructured environment. This scenario is a test of mobility of the swarm as a whole. Moving multiple drones simultaneously will test the Boids Algorithm, while also requiring a different protocol for moving the whole swarm.

3.2 Common behaviours for all cases

The behaviours are implemented as steering vectors, acting as a force to control the drones in certain ways. The forces produced by these behaviours are summed to give the resulting action. When multiple behaviours are active simultaneously, there is a risk of two vectors cancelling each other. This is a crucial point in critical situations. For example, it should be more important to avoid a collision than align with flock mates.

Therefore a priority system has been implemented, as suggested by Craig Reynolds in [1], as an alternative to combining behaviours by adding them. If a high priority behaviour has a non-zero output it will dominate the steering force, otherwise it will let lower priority behaviours take control.

The priority system will consist of two layers, collision prevention will be the first, since for drones as used in this project, avoiding crashes take priority. `Obstacle avoidance` and `separation` is on this layer. The second layer is `alignment`, `cohesion` and any additional case specific behaviours like `seek` or `analyse`. Using this structure, when a drone senses an obstacle it will focus only on avoiding it, flock mates around this drone will move out of the way, and when they are clear of each other and any obstacles, they will follow the lower-priority behaviours.

`Obstacle avoidance` consist of two sub-behaviours; `avoid` and `evade`. `Avoid` uses LiDAR values all around the drone to fly away from objects if they get too close. The method will use the average direction of rays that read a small value under a predefined threshold, indicating a close object, and compute a force in the opposite direction at full force.

Using only `avoid` will lead to very uneven paths, and possibly not preventing a collision if the drone is flying towards the obstacle and is not able to stop in time. `Avoid` only reacts to very close objects, therefore the other sub-behaviour, `evade`, will utilize the longer range of the LiDAR to detect upcoming objects in front of the drone, using a specified field of

view. The drone will aim to fly around the object. It does this by looking for the longest LiDAR ray, checking the outer rays first and then towards the center, and steering towards that reading if an obstacle is detected. It will not react to objects that are within the range of the `avoid` method to prevent them interfering with each other. The `evade` behaviour is illustrated in figure 3.1.

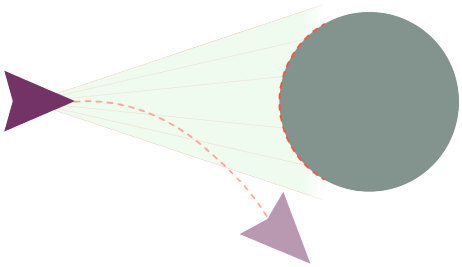


Figure 3.1: Illustration of `evade`: How a drone detects an obstacle and follows the longest LiDAR reading.

The main difference to be stated between the Boids Algorithm and an implementation meant to control physical objects, is that Boids was originally made for entertainment applications like movies or games that feature a swarm of animals, either birds, fish or insects. The Boids behaviours therefore focus on the aesthetics rather than practical operation [6], a consequence is that agents occasionally fly into one another. However, this project control digital drones that simulate realistic action, and as such a crash will occur if two agents collide. To compensate for this issue, a more strict approach will be implemented for the separation aspect of the Boids Algorithm. It will hinder smooth movement, but aim to significantly decrease the collisions between drones.

`Separation` works by evaluating flock mates that are closer than a fourth of the perception range, and computing an opposite force. And average is calculated and this is the desired direction. The steering vector is calculated by subtracting the desired vector by the drone's velocity. Due to the importance of preventing collisions, the steering vector is scaled to the maximum force.

`Alignment` is implemented as described in equation 2.1 by taking the average of the velocity vectors of the perceived flock mates, and subtracting that average with the agent's own velocity. This gives a steering vector, a force, used to steer the agent to align with its local flock mates. If the force exceeds the maximum force the drone's can output, the vector's magnitude will be set to the value of the maximum force.

`Cohesion` is implemented according to equation 2.3 by calculating the center of mass in the cluster of local flock mates, that is the average of their positions. This average is

subtracted with the agent's own position. Like with `alignment` this desired vector is subtracted from the agent's own velocity to calculate the force needed to steer the agent towards the desired vector. This force vector is also cropped to the maximum force.

The method of calculating vectors for each behaviour, for each drone at all time steps, is a computing heavy tasks, that slows down the execution time. To combat this a combining scheme suggested by Craig Reynolds in [1], is implemented for some of the low-priority behaviours. Only one behaviour is applied at a time step, utilizing the drones momentum as a low-pass filter to blend the behaviours together. In the current implementation only `align` and `cohesion` are applied this way, all other behaviours are either priority determined or summed.

Chapter 4

Swarming Drones in Kinematic Simulation

To setup and adjust the behaviours of the drone swarm, a simple kinematic simulation has been coded in Python, serving as a test platform for easy deployment of new behaviours, as well as tuning the parameters. The script that controls the behaviour of the swarm will be the same, with a few modifications, that is also controlling the behaviour of the simulated drones in Gazebo.

The following section describes the simple Python simulation platform, and section 4.2 explains the different behaviours needed for different applications and their implementation. One of the tested application scenarios will be implemented in the physics based simulation.

How to find the source code for the kinematic simulation is described in appendix B.

4.1 Simple simulation platform

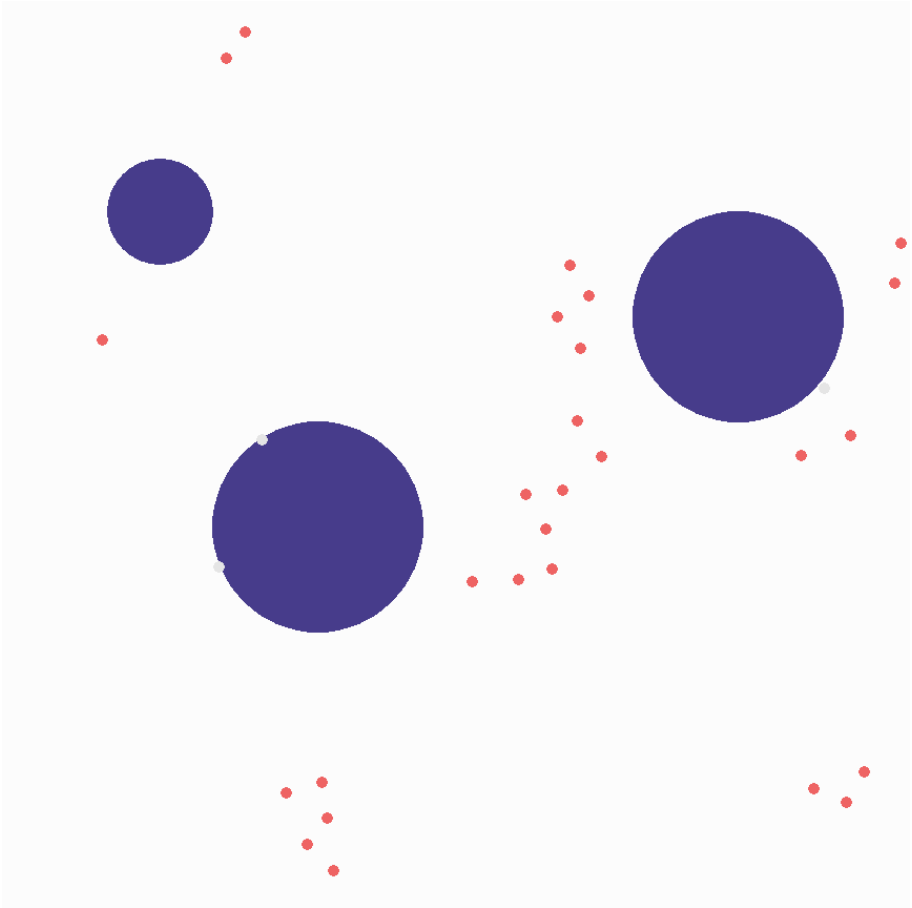


Figure 4.1: An arbitrary scene in the simple simulation. Blue circles are obstacles, red dots are the drones and grey dots are collided drones that no longer register as part of the swarm.

The purpose of the simple kinematic simulation is to quickly test and edit behaviour strategies, to be later implemented in the Gazebo simulation. An image of the simulation is seen in figure 4.1.

The simulation platform itself is designed to create and manage agents, providing details about position and sensor values to the behaviour script, and use the returned force values as reference input, the same way that the physics based simulation will do. In this manner the same behaviour script is supposed to work with both simulations, with few modifications.

This simulation is written in Python and is a simplification of the Gazebo world with no uncertainties or noise implemented. It is also only a two-dimensional representation of the world.

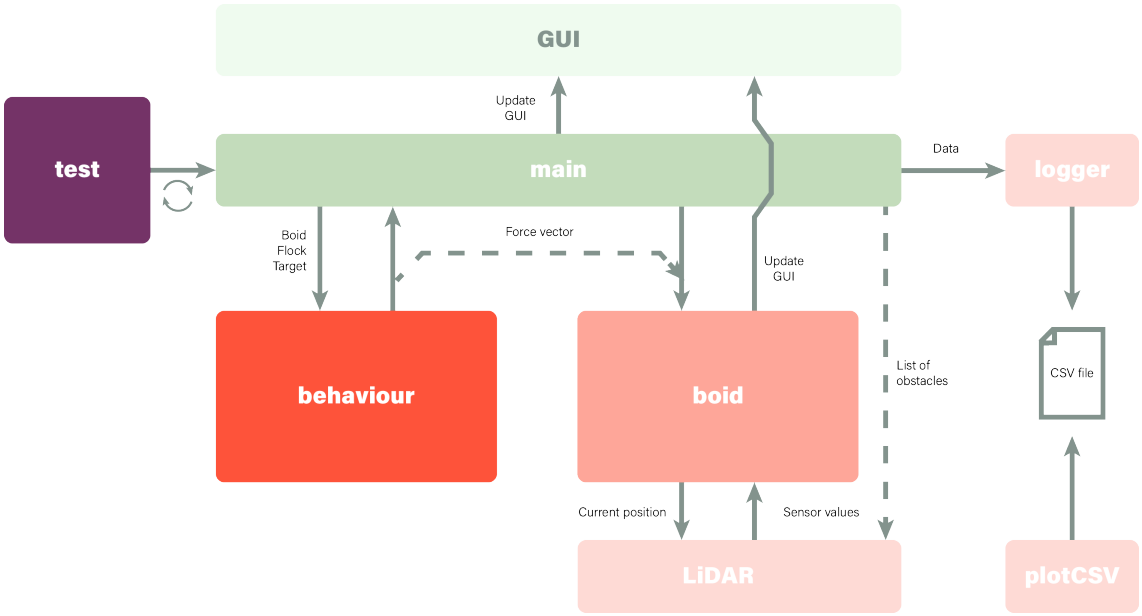


Figure 4.2: Block diagram of the kinematic simulation.

Figure 4.2 is an abstraction of the program structure of the kinematic Python simulation. The filled-in lines are direct information transfer, and the dashed lines are indirect transfer, where the block in between passes the data. The `main` file is the control loop for the program, and creates and updates the swarm using the `behaviour` script and updates the GUI to show the simulation. Each agent in the swarm is of the `Boid` class, represented graphically by a dot. They have attributes like position, velocity and acceleration.

Each boid uses a simulated LiDAR that provides information of distance and direction of nearby obstacles, which to the LiDAR sensor is defined in advance from `main`. Is does this by defining a distance function for circles, and using ray marching, specifically sphere tracing [19], to create the artificial sensor readings.

The `main` file also sends information about the state of the simulation to the `logger` scripts which saves the data in a csv file format. The `plotCSV` script simply plots the data in a graph. For each case specific data set there is a specific `plotCSV_<case-ID>` file.

To perform the tests of the cases, a `test_<case-ID>.py` script has been made for each case. It executes the simulation a specified number of times, and combines any logged data if needed. This is described in detail in appendix C for case c) and appendix D for case d).

4.2 Relevant Behaviours

Different behaviours will be relevant for different applications, but four behaviours are to be present for most applications. These are the three rules from the Boids Algorithm and obstacle avoidance. The Boids behaviours will be mostly relevant for situations where the swarm flies in a cluster. The three rules of alignment, cohesion and separation will aid each drone to remain airborne without crashing, and maintain the cluster formation of the swarm.

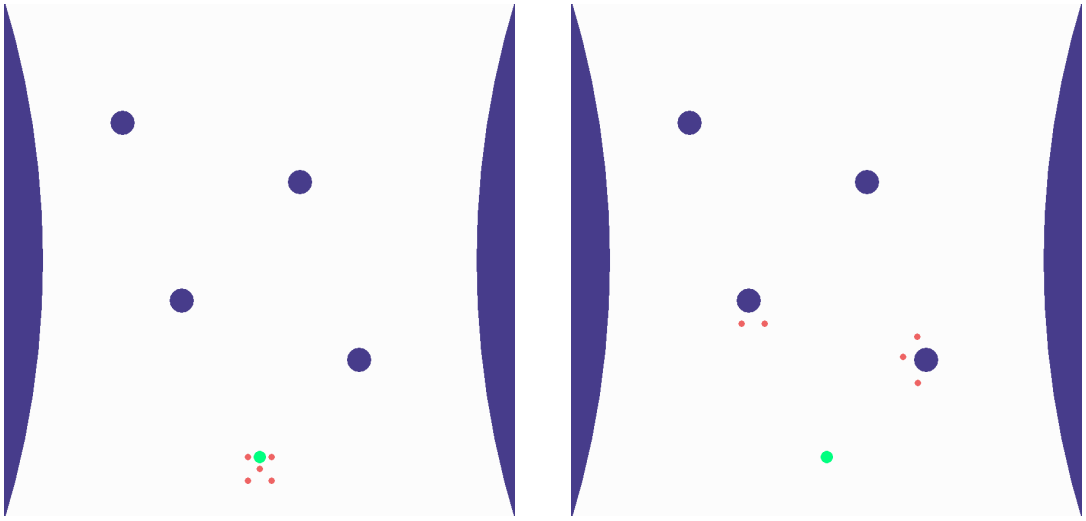
Two out of the four cases has been chosen to be implemented in the kinematic simulation, these are case c) and case d).

4.2.1 Case c) - Data Collection in the Canopies of Trees

For this application the obstacles are not only to be avoided, but objects that are to be analyzed by the drones. The scenario will be simplified to only concern four spread out trees, and to analyse a tree drones just have to be within proximity. For a description of the experiment performed for this case, see appendix C.

This case will rely on some information being shared between the drones stored at a central location, that all the drones can write to and read from. This is information about which trees have been analysed, how many drones are analysing each tree and the progress to completion, or the 'tree timer'. Each tree has its own timer, and at every frame that value is reduced by the number of drones analyzing the tree. A drone will only join another group analyzing a tree, if that group has less that three drones already, unless it is the last tree remaining. On figure 4.3 an example of the start and middle of the simulation is shown. The drones will maintain the same starting position throughout all of the tests, but will be initialized with a random velocity.

For this case it proved to be a disadvantage to have all of the Boids Rules implemented, namely `alignment` and `cohesion`. These rules would try to force the drones to stay together, making it ineffective to divide into the needed sub-groups. This is due to these forces cancelling the `seek` behaviour that directs the drones towards a tree, because these behaviours are on the same priority level. It was chosen remove these behaviours rather than demoting them, because they would not be helpful in completing the scenario. The third Boids rule `separation` however, is still crucial to this case for preventing collisions.



(a) Start of scenario (b) Simulation example after 100 frames

Figure 4.3: Start position and example of the simulation after 100 frames in implementation of case c). The green dot is the starting location.

In figure 4.4 four graphs are plotted together, each representing the frames for completion for different flock sizes, namely of 1, 3, 5 and 7.

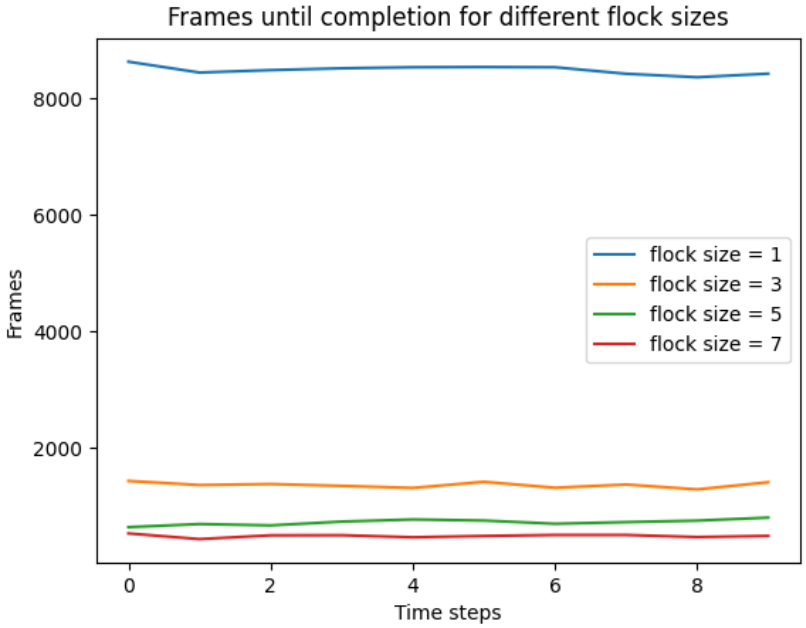


Figure 4.4: Amount of frames before completion in 10 tests for different flock sizes.

It can be clearly seen that increasing the flock size from 1 to 3 greatly improves the time. From 3 to 5 is still considered magnificent, as it is almost halved. Increasing flock size beyond this point improves the time, but at a decreasing factor. As such the thorough

test is performed with a flock size of five. The average frames of the tests and percentage decrease are shown in table 4.1.

Flock size	Average	Percentage decrease from last step
1:	8483.8	-
3:	1369.3	-83.85%
5:	730.2	-46.67%
7:	495.6	-32.13%

Table 4.1: The average frames for completion

On figure 4.5 is the time, measured in number of frames, it took for the swarm to finish analyzing all four trees, repeated 50 times with a flock size of five. The graph shows that the amount of frames it takes are steady around the average, which in this test was 720.9. This is expected due to the constant starting positions.

At test 41 a collision occurs, and this has a great impact on the amount of time it takes to complete the task, approximately twice as much. However, due to the nature of swarm robotics the objective can still be completed.

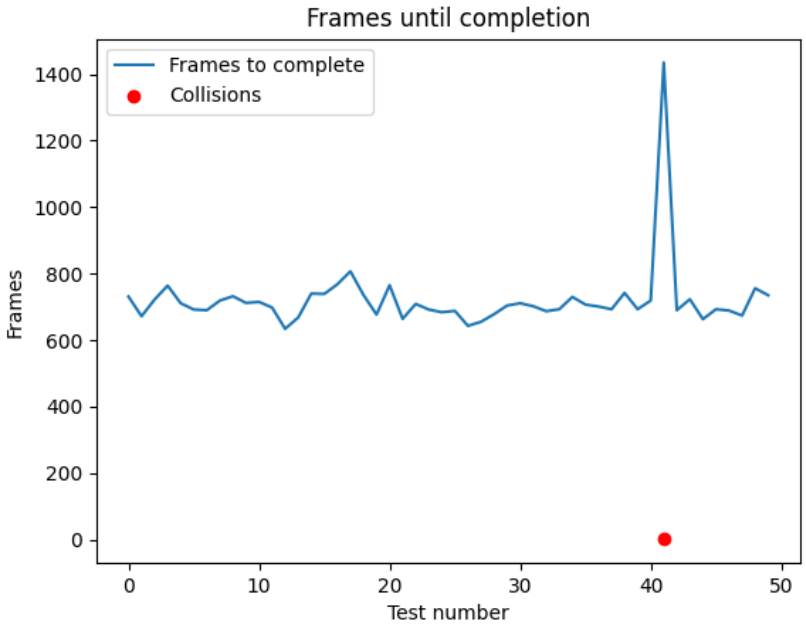


Figure 4.5: Amount of frames before completion in 50 tests, and collisions.

The number of collisions during the 50 tests was 1. For each test there where 5 drones for

a total of 250, this yields a collision percentage of:

$$\left(\frac{1}{250}\right) \cdot 100 = 0.4\% \tag{4.1}$$

This is an acceptable collision percentage.

In this test it is shown that it makes a huge difference whether a drone takes on this task alone, or as a swarm. It was found that for this scenario of analyzing four trees, a flock size of five is appropriate. Should a collision occur the time will increase, but the objective will still be completed by the remaining drones. This illustrates some of the many advantages of using swarm robotics.

4.2.2 Case d) - Moving a Swarm of UAs From Point-to-Point in an Unstructured Environment

This scenario will rely on the Boids Algorithm and the `obstacle avoidance` behaviour to maneuver the terrain. To navigate to the goal one other behaviour is implemented; `seek`. This is a very simple method, which steering force is in the direction of the goal from the current position. However, when more members of the flock reaches the goal, they will all be attracted towards the middle. To prevent this tug-of-war, a larger area around the goal will be the goal-zone, the size of which is dependent on the flock size. Drones within it should slow down to a halt, while only keeping the first priority behaviours active. Another area is the goal-zone expanded by twice its size, in this region drones will only focus on seek as a second-priority behaviour. All areas outside these regions will function as normal. This is illustrated in figure 4.6.

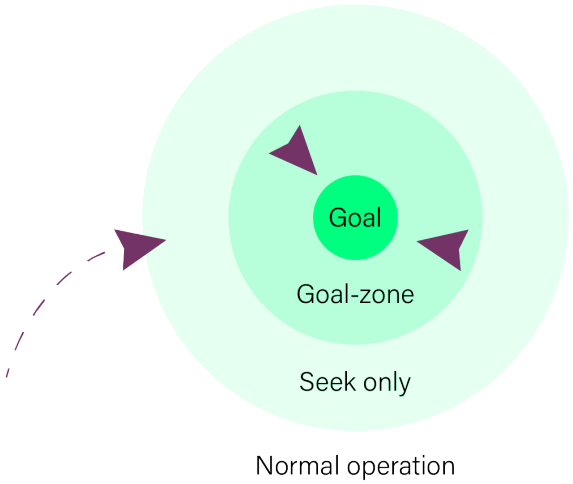
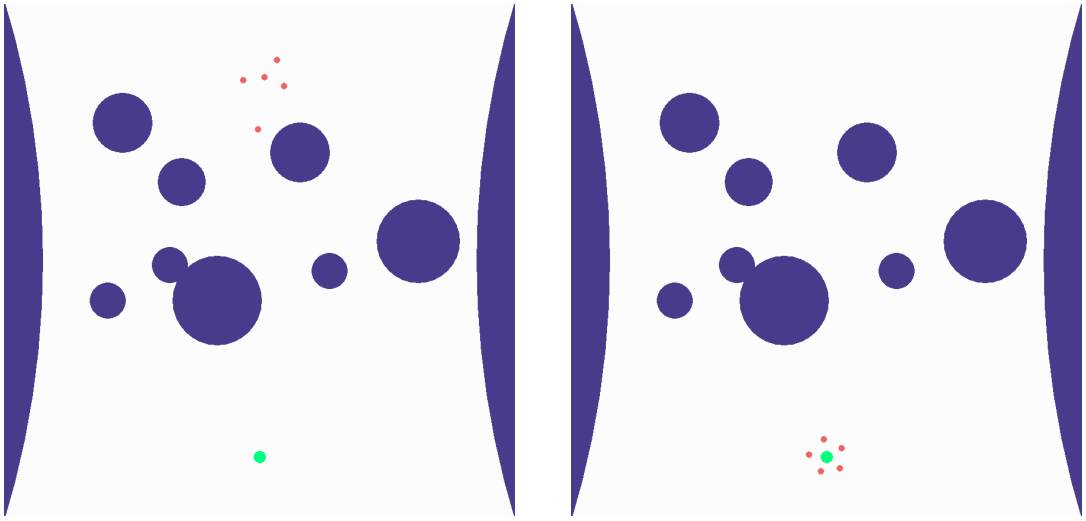


Figure 4.6: Illustration of the different areas around the goal. The first priority behaviours are active in all layers.

Figure 4.7 show an implementation of case d). Figure 4.7b shows how the swarm is not on the goal, but rather around it. This is the desired effect - the protocol populates the edges of the goal-zone, if more drones wishes to reach the goal, they would enter the goal-zone and the other drones on the edge would by `separation` move towards the middle.

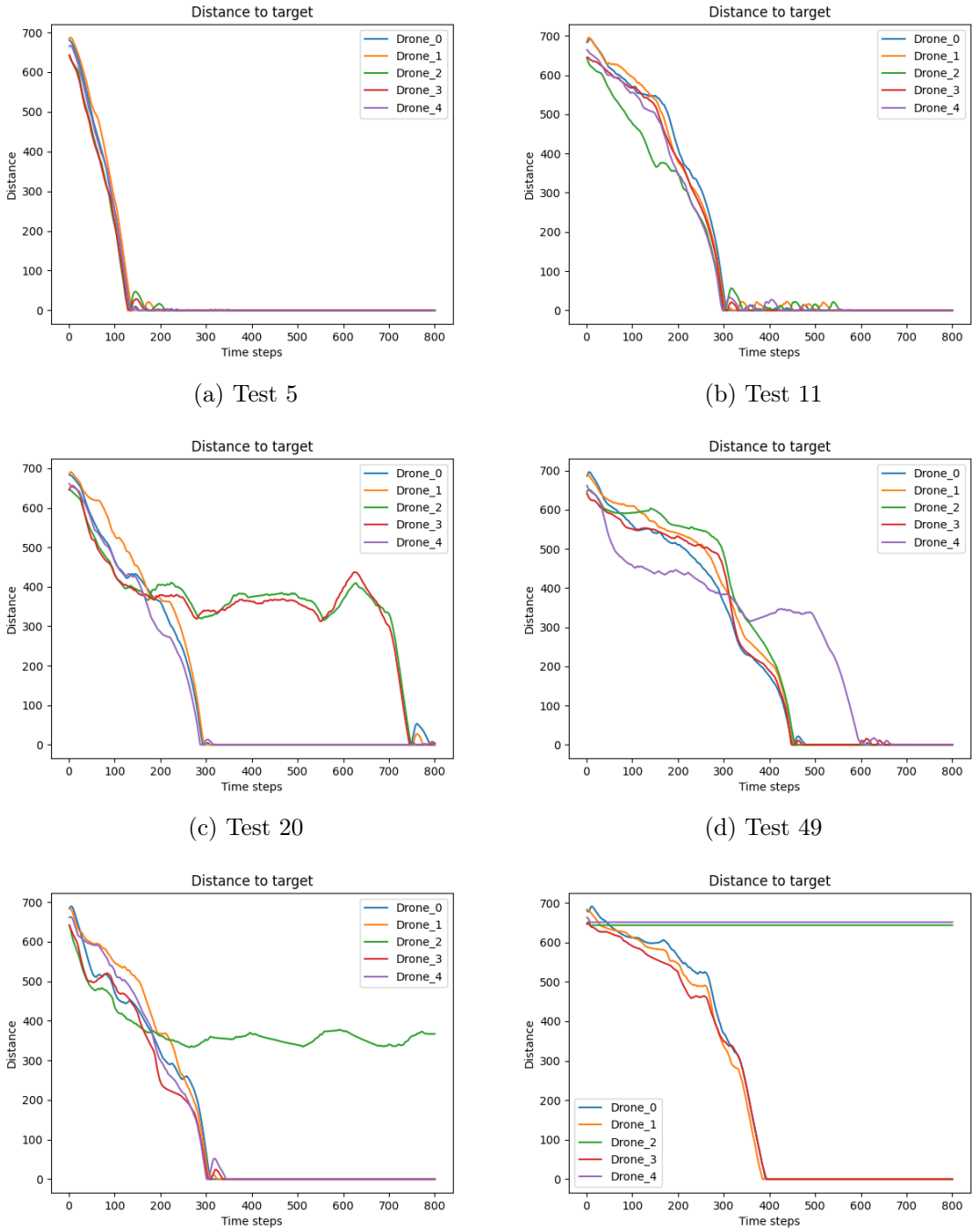


(a) Start of scenario (b) End of scenario

Figure 4.7: Start and end of simulation implementing case d). The green dot is the goal.

An experiment was done to determine how well the implemented behaviours would perform when reaching the goal. 50 tests were done, each where the drones had the same starting position, but a random initial velocity. for a description of how this experiment was conducted, see appendix D.

In figure 4.8 six different plots are shown, these are the distance from each drone to the goal zone for six different individual tests, chosen from the pool of 50 tests. Figure 4.8a and 4.8b shows that the drones are able to quickly find a path through the obstacles, and reach the goal as one whole flock. Figure 4.8c and 4.8d shows examples of the flock splitting up, one where it splits into two smaller flocks and the other where just a single drone gets separated from the rest. It is also clear form these plots that when the later flock reaches the goal-zone, the other drones move out of the way before settling back into a distance of 0. In figure 4.8e one drone also splits from the flock, but this time it gets stuck and never reaches the goal-zone before the time runs out. This illustrates an issue with the current priority based implementation, discussed further in section 6.1. In figure 4.8f is an example of two drones having the same constant distance throughout that whole test, indicating that those two collided.



(a) Test 5

(b) Test 11

(c) Test 20

(d) Test 49

(e) Test 40

(f) Test 30

Figure 4.8: Handpicked plots from the pool of 50 tests, that show different outcomes.

The number of collisions during the 50 tests was 3. For each test there where 5 drones for a total of 250, this yields a collision percentage of:

$$\left(\frac{3}{250}\right) \cdot 100 = 1.2\% \tag{4.2}$$

This is a reasonably low value, but for deployment on real hardware platforms this would be too high, and would need further improvement.

Figure 4.9 is a plot of the average distance from the drones to the goal-zone, for each individual drone over the 50 iterations. The figure shows a clear correlation between the distance from the drones to the goal-zone over time, in that it decreases as expected. At frame 800, the end of the simulation, the average distance is not zero for all drones. This indicates that these drones did not reach the goal-zone before the simulation terminated in all iterations. The notches on the graphs are caused by an oversight in the program, if a drone reaches the bottom of the screen, it will be teleported to the top, resetting the distance in the time span of a single frame.

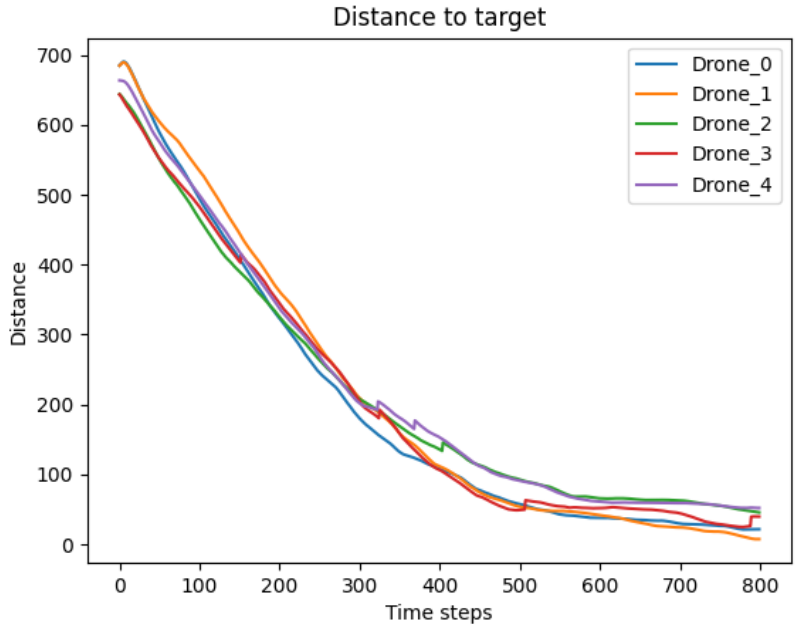


Figure 4.9: Average distance for each drone in a flock of five over time, the unit of time is frames.

This test demonstrate that a flock of five drones can in a simple kinematic simulation navigate an obstacles course as a flock, or sub-flocks, to reach a goal with few collisions. There are improvements to be made, since not all the drones reach the goal-zone within reasonable time. Decreasing the collision percentage of 1.2% would also be mandatory before implementing it on a real system, but for simulation purposes this is an acceptable value.

Chapter 5

Swarming Drones in Physics Based Simulation

The kinematic simulation described in section 4, lacks crucial parameters to simulate a realistic environment. In the simple simulation drones are mass-less dots, but to test whether or not the suggested behaviours would work on an actual drone platform, parameters like the weight of the drone and the maximum force its actuators can apply, is important to consider. The following section describes the set up for a physics based simulation, and section 5.2 is about how the behaviours from the kinematic tests are implemented, and how they perform in a physics based environment.

How to find the source code for the physics bases simulation is described in appendix B.

5.1 System Setup

Figure 5.1 is a UML deployment diagram that shows the system architecture of the physics based simulation. Setting up the different aspects are explained in the following sections.

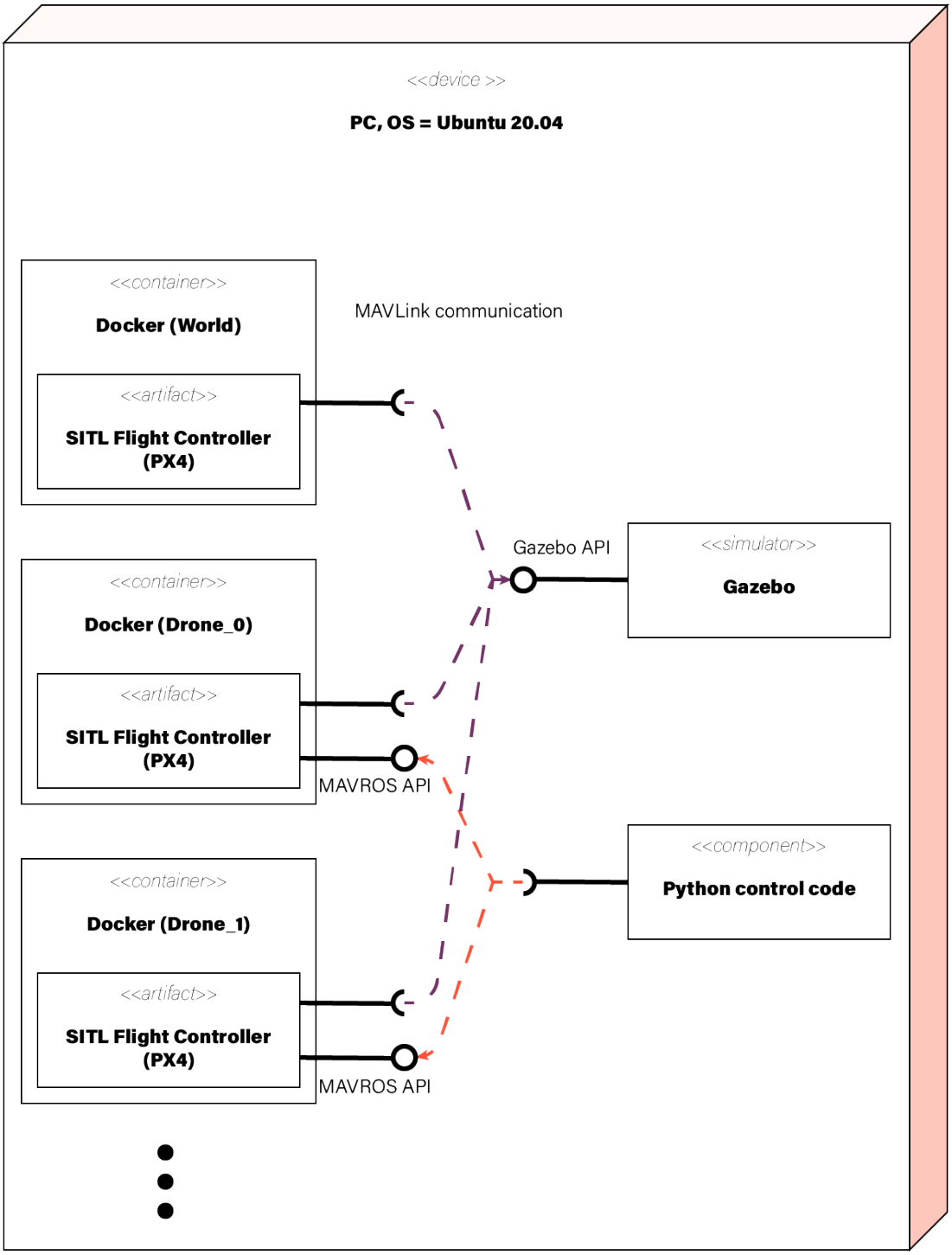


Figure 5.1: UML Deployment Diagram of physics based simulation setup.

5.1.1 Setting up Docker container

In this project, Docker is used to create simulated drones and environments to run in simulation software, in this case Gazebo. The Docker images are provided by SDU UAS Center, and pulled from the repository using the `docker pull` command in a Linux terminal. These container environments sets up the flight controller as software in the loop (SITL), so it runs on the computer rather that external hardware.

Two types of Docker containers are pulled from the repository, a world and a drone container. The world container has a Gazebo and ROS master port, that multiple drone containers can connect to [20]. For a simulation only one world container is run, but for each drone present a new drone container has been created. Because of this there is always $N + 1$ containers running on the machine at one time, where N is the swarm size.

For a list of commands to manage Docker containers see appendix A.

The world container includes a preset of worlds to run, among them a simple empty world used for most of the testing. The drone model used in this project is also a preset from the repository and is a hexa-copter configuration called sdu_drone, see figure 5.2.

Figure 5.2: Rendered image of the sdu_drone model used in the project. In this image the model is without its propellers.

5.1.2 Enabling Offboard Control

Offboard control means to control the flight stack with external software other than the autopilot, and is done through the MAVLink protocol [21]. It is necessary to autonomously control the simulated drones, to be able to apply the custom behaviours.

For each drone container a provided Python script class is initiated, from `offb_postctl.py`. This script initializes a ROS node called `offboard_ctrl` and sets the mode to offboard. It also initializes a publisher that publish the set point location, and subscribers to receive information about the state. Figure 5.3 shows an example of a drone in Gazebo and

a sample of the output from the `/<container-ID>/mavros/setpoint_position/local` node, initialized by the `offboard_ctrl` script.

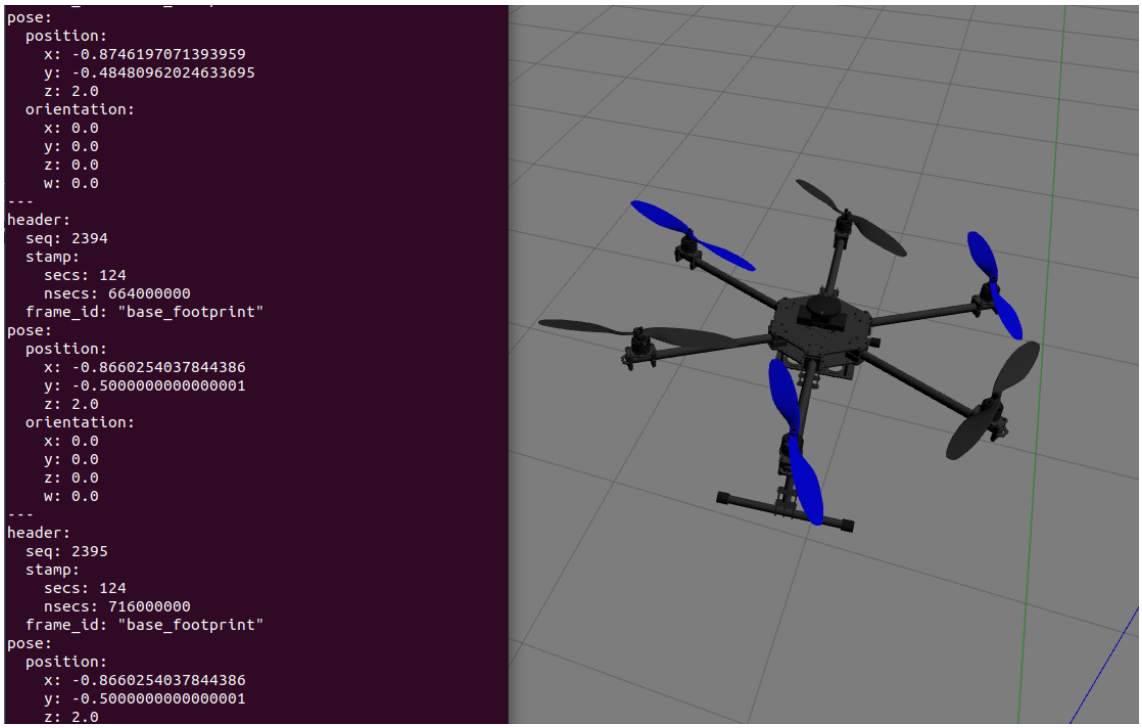


Figure 5.3: Image showing a simulated drone in Gazebo and an example of the terminal output from the MAVROS node `/<container-ID>/mavros/setpoint_position/local`.

5.1.3 Managing and Controlling the Simulation

To automate the process of running the proper Docker containers, setting up offboard control and run the behaviour methods, another Python script has been written, `control.py`.

This script executes bash commands to run the Docker containers, and initializes an `OffboardControl` class for every drone container in the swarm. Alternatively, the commands listed in appendix A would have to be executed every time the simulations was to run.

This script will after the initialization steps enter a while-loop where custom code can be executed to control the drones, see section 5.2 for an explanation of this. This while-loop is only interrupted by hovering over the 'Initialize Shutdown' button, that the program have created on the screen. This will stop offboard commands, send a ROS shutdow request and stop all running containers.

Having a script that handles these processes greatly increases test frequency, due to these steps otherwise having to be done manually.

5.1.4 Simulating in Gazebo

To test the behaviours in a realistic setting, a specific application that was also tested in the kinematic simulation is chosen to be implemented. Case d) is the chosen case for its simplicity and demonstration of the behaviours inspired from the Boids Algorithm, `Obstacle Avoidance`, and the `seek` behaviour.

To recreate the scenario of the kinematic simulation, specifically case d), the world must first be changed to spawn cylinders to match the map. This allows for a comparison between the results. However, using the provided Docker world container, the .world files themselves are inaccessible. Instead, the Docker container was to be rebuilt, with a new custom .world file. For a description of how to build this custom Docker image see appendix A.

This world file, `case_d.world`, is a copy of the `empty.world` file with the addition of included cylinders. These cylinders are defined by creating a `model.sdf` and `model.config` for every size variation of the cylinders. The world file refrences these models and places them at the appropriate location in the world. The coordinate system is scaled by a factor of $\frac{1}{20}$ to convert into the Gazebo units of meters. This factor was chosen to test the setup of the system, but should instead have have been chosen from the size of the drones. The width of the drone model, sdu_drone, in meters is 0.74, whereas it is 10 pixels in the kinematic simulation. The appropriate scaling factor would be:

$$\frac{0.74}{10} \approx \frac{1}{13.51}$$

The world of `case_d.world` running in Gazebo is shown on figure 5.4.

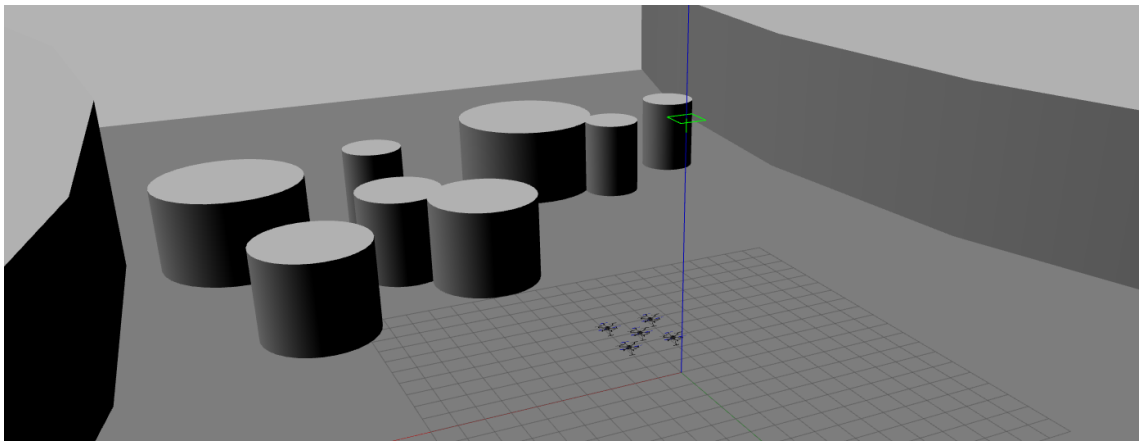


Figure 5.4: The map of case d) implemented in Gazebo.

5.2 Implementation of Kinematic Behaviours

For the `behaviour.py` script from the kinematic setup to be implemented with the Gazebo simulation and ROS, an interpreter is needed between the information received about the drones through MAVROS and the script, and the forces calculated to be applied to the drones. Comparing it to the kinematic simulation, this is a replacement for the `boid.py` script, see figure 4.2 for a reference of the system structure of the kinematic simulation. Here, a `drone.py` script has been written to read and store information that the drones publishes through MAVROS, to be called by the `Behaviour` class. It sets the publisher node from `offb_posctl.py` that controls the setpoint to a value, depending on the force calculated by `Behaviour`.

To simplify the process of configuring obstacle avoidance for the drones in Gazebo, the same `lidar.py` script is reused to simulate a LiDAR sensor for each drone. This is an alternative to adding a LiDAR sensor from the Gazebo library which would be more realistic, but present an issue not addressed in this implementation. This is discussed further in section 6.1.

Unfortunately, not enough time was spent on simulating in the physics based simulation, as the setup was more complicated to work with than expected.

However, a simulation environment was successfully set up, using the Docker containers provided and customising them to have specific obstacle placements. The drones were spawned in the simulation, and was connected to the custom python script by initializing offboard control. Controlling the drones however, using the same behaviour script as in the kinematic simulation, was not a successful implementation. The drones did not behave as intended, see figure 5.5, and more time should have been spent tuning this script.

This means that a comparison between the results from the kinematic simulation and the same behaviour in a physics based simulation environment is not possible, despite it being one of the goals of this report.

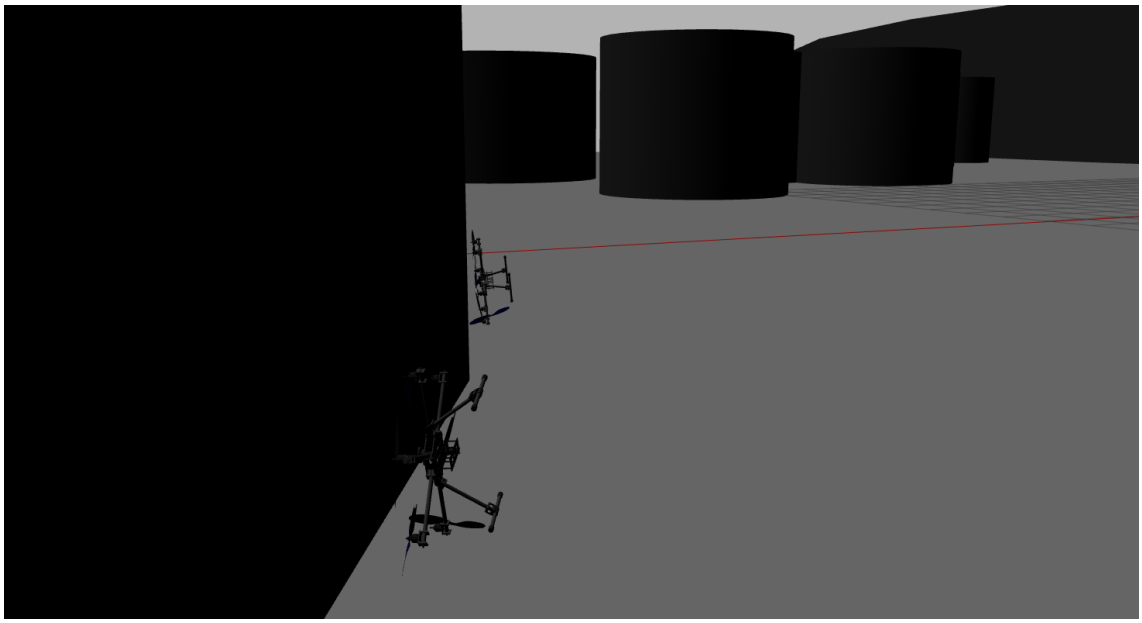


Figure 5.5: Example of how the drones are not supposed to behave.

Chapter 6

Perspectivation

6.1 Discussion

Unsuccessful Implementation of Case d) in Gazebo

Complications in the setup resulted in an unfinished product, that made it impossible to compare the kinematic behaviours in a realistic simulation within the time frame. Though the work done to complete the setup is base for further tinkering after the completion of this report.

Sensing Flock Mates with LiDAR

As described in section 5.2 the LiDAR used in the physics based simulation is the same as that from the kinematic simulation. This is not a realistic representation of the sensor, and for further implementations this should be addressed.

The current custom LiDAR creates sensor values based on the location of the drone, and the known locations of obstacles in the world. A real LiDAR sensor does not know the placement of obstacles, but produce sensor values based on the physical surroundings. This means that other flock mates would be visible to the sensor, and as such a method for filtering out other drones has to be developed.

Such a solution could involve only perceiving flock mates in the front of the drone to reduce complexity, since it is rarely needed to react to drones behind itself in most applications. A filter is needed to know whether or not an obstacle should be perceived as a drone. The approximate location of flock mates are known, and with this information it could be decided if the detected readings match any flock mates.

It should also be tested if it can be effectively utilized, that the drones are not invisible

to the LiDAR sensor. Since each drone reacts to only the current state, and does not consider the future, other drones could be treated as obstacles, and replace the `separation` behaviour entirely.

Locating Drones

The method of locating the drones in the Gazebo simulation is to know the start position and the chosen set points. This is a way of locating the drones with no noise, however it might be inaccurate. Since the position is based on the set point, it is not the actual position of the drone itself, but rather where it will go a few steps ahead.

Alternatively, to achieve a more realistic scenario in the simulation, a GPS should be used. This will introduce noise, and demand for the behaviours to take into account this uncertainty for the location of flock mates.

Tunnel Vision for Obstacle Avoidance

In the current implementation the different behavioural rules are organized in a priority system. This however has the downside that the applied force is exclusive to the chosen priority behaviours. This means that when an obstacle is in sight of a drone, it is the only thing that is focuses on, creating the 'tunnel vision' effect.

This means that in situations with many obstacles the drones will be ineffective for reaching a goal, like illustrated in case d) section 4.2.2. Here it is shown how sometimes a drone will not reach the goal in time. This is because it has come across obstacles that surrounds it and forcing the drone to evade them continuously, suppressing all other behaviours.

A solution could be to make a critical change to the priority system, giving each drone a "force quota" [3], that is divided between the different priorities. If the highest order priority is not using all of this force, lower priorities can apply what remains. As of now, even if `obstacle avoidance` is not using the max force, it still excludes all other behaviours.

This would also require a chance to the `obstacle avoidance` rule, due to it currently always use max force. As an alternative, `obstacle avoidance` could be dependent on the distance to the object rather than always use full power.

Binary Separation Scheme

As it is implemented currently, the drones will steer away from each other if they get closer than a fourth of the perception range. This helps avoiding collisions. It does mean, however, that this behaviour will consistently use full force, which might not always be

necessary. In Craig Reynolds' original presentation of the `separation` behaviour, the force applied was dependent on the distance to other drones, see equation 2.4, such that closer drones would affect the force more.

This was problematic to implement, as for drones further away the desired force vector would be very small, essentially making the drone come to a halt. For closer drones the force might not be enough, and they would collide. This is not a problem in computer animation, but for real drone platforms it is something to be avoided. A solution might be to implement the priority system mentioned in the previous section using the force quota principle.

Over Simplified Environments

Every obstacle in the simulated worlds are represented as circles or cylinders, and treated as such. This is due to the simplicity in defining a distance function to circles, and using them to create artificial LiDAR readings. It was done for ease of programming, but is a very unrealistic representation of real environments. In future work a LiDAR simulated by Gazebo itself should be used, and as such more complex obstacles can be placed in the worlds.

Expanding to 3D

Due to the simplicity of the method of steering the drones, by calculating force vectors that can be added, expanding to 3D should be a simple task. It would involve creating a new `Vector3D` class that operates with three elements, which is done by adding to the already existing `Vector2D` class, which was originally posted here [22]. Besides this it is a matter of changing all declared vectors to 3D, using this new class.

It is a task that is simple but time consuming. However the real world is in three dimensions, accordingly it would be more realistic if the algorithm was operating in three dimensions as well. Yet it is not crucial to demonstrate the algorithm at work, and two dimensions will satisfy the project objectives of the report.

6.2 Conclusion

By setting up a kinematic simulation in Python, the simple nature-inspired rules of the Boids Algorithm and other behaviours like obstacle avoidance, could be tested on a small swarm of dots representing the UAs. Two case scenarios was implemented and tested, one, case c), demonstrating the advantages of using swarms over a single drone in applications that can be divided into sub-tasks. The other, case d), was demonstrating how a flock of drones could navigate an unstructured environment, with few collisions.

A simulation setup in Gazebo was prepared, but the implementation of case d) was unsuccessful, and yielded no useful results.

Project objective 1, "Decide on what type of UAV to simulate", was completed, and a multi-rotor configuration was chosen. They have the capability to hover which makes them ideal for applications like the one of case c), to analyse static objects such as trees.

Project objective 2, "Learn simulation in ROS" was mostly completed. Despite not having a working simulation of case d) in Gazebo, the process of enabling offboard control through ROS, and communicate with the Docker containers using MAVROS is better understood, and the knowledge can be used to work on future projects with ROS and Gazebo.

Project objective 3, "Determine the means of information distribution between the drones", is indirectly stated, as in case d) it was a matter of having access to information about position and velocity of flock mates. In case c) this applied too, as well as relying on some information being shared between the drones stored at a central location, that all the drones can write to and read from. The matter of how they communicate this information has not been explored further.

Project objective 4, "Implement a use case based on simulated swarming UAVs", was partly completed, as two cases was implemented in the kinematic simulation, where as none was in the physics based simulation.

Project objective 5, "Implement global navigation of the swarm", this was addressed in case d), where the global navigation was done by implementing the `seek` behaviour in each drone, effectively moving the whole flock.

Project objective 6, "Implement global control of the swarm for flying in formations", was not examined.

Chapter 7

Bibliography

[1] Craig W. Reynolds. Steering behaviors for autonomous characters. https://www.red3d.com/cwr/papers/1999/gdc99steer.html.

[2] The Dronecode Foundatio. Simulation | px4 user guide. https://docs.px4.io/master/en/simulation/. Online; accessed 16 May 2021.

[3] Daniel Sinkovits. Flocking behavior. https://guava.physics.uiuc.edu/~nigel/courses/569/Essays_Spring2006/files/Sinkovits.pdf.

[4] Suraj Gupta, Mangesh Ghonge, and Pradip Jawandhiya. Review of unmanned aircraft system (uas). *International Journal of Advanced Research in Computer Engineering Technology*, 9, 04 2013.

[5] SDU Dronecenter. Artdrone. https://www.sdu.dk/da/forskning/sduuascenter/researchprojects/artdrone. Online; accessed 22 May 2021.

[6] Craig W. Reynolds. Boids. https://www.red3d.com/cwr/boids/. Online; accessed 3 February 2021.

[7] Open Robotics. About ros. https://www.ros.org/about-ros/. Online; accessed 15 May 2021.

[8] Wikipedia the free encyclopedia. Software framework. https://en.wikipedia.org/wiki/Software_framework. Online; accessed 15 May 2021.

[9] Open Robotics. Writing a simple publisher and subscriber (c++). http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29. Online; accessed 30 May 2021.

[10] Wikipedia the free encyclopedia. Robot operating system. https://en.wikipedia.org/wiki/Robot_Operating_System. Online; accessed 24 May 2021.

[11] The Dronecode Foundation. Mavlink developer guide. https://mavlink.io/en/. Online; accessed 15 May 2021.

[12] Ros with mavros installation guide. https://docs.px4.io/master/en/ros/mavros_installation.html. Online; accessed 15 May 2021.

[13] Open Source Robotics Foundation. Gazebo - beginner: Overview. http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1. Online; accessed 15 May 2021.

[14] Open Source Robotics Foundation. Gazebo. http://gazebosim.org/. Online; accessed 22 May 2021.

[15] Open Source Robotics Foundation. Gazebo: Projects. http://gazebosim.org/projects. Online; accessed 22 May 2021.

[16] Docker. Docker overview. https://docs.docker.com/get-started/overview/. Online; accessed 25 April 2021.

[17] Opensource.com. What is docker? https://opensource.com/resources/what-docker. Online; accessed 25 April 2021.

[18] Prakhar Srivastav. Learn to build and deploy your distributed applications easily to the cloud with docker. https://docker-curriculum.com/. Online; accessed 31 May 2021.

[19] Jamie Wong. Ray marching and signed distance functions. http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/#the-raymarching-algorithm. Online; accessed 27 May 2021.

[20] Jes Grydholdt Jepsen. Px4 docker setup, 2 2021.

[21] The Dronecode Foundation. Offboard control. https://docs.px4.io/master/en/ros/offboard_control.html. Online; accessed 24 May 2021.

[22] Christian Hill. A 2d vector class. https://scipython.com/book2/chapter-4-the-core-python-language-ii/examples/a-2d-vector-class/. Online; accessed 17 February 2021.

[23] Docker. Install docker compose. https://docs.docker.com/compose/install/#install-using-pip. Online; accessed 23 May 2021.

[24] The Dronecode Foundation. Qgroundcontrol. http://qgroundcontrol.com/. Online; accessed 29 May 2021.

Chapter 8

Appendix

A. Terminal Commands

B. Source Code

C. Test of Kinematic Simulation - Case c)

D. Test of Kinematic Simulation - Case d)

Appendix A

Terminal Commands

If not using `control.py` the following commands are used to set up the physics based simulation, without the behaviours.

Docker

Run the world container with the empty world:

```
docker run -name world -network host -id -rm
sduuascenter/px4-simulation:vm-server-sdu-world 17550 11311 empty
```

Run a drone container using the sdu_drone model:

```
docker run -name drone -network host -rm -id
sduuascenter/px4-simulation:vm-server-sdu-drone 16550 17550 11311 sdu_drone
0 -1 -1
```

The last three numbers are ID, x and y. The name also has to be uniqe for each running drone container.

To run the world created for case d), a new custom Docker container needs to be built that contain this world. For this Docker Compose needs to be installed [23]. Go to where the Docker container folder is located:

```
cd  /<path-to-docker-container-folder>/px4-simulation-docker
```

Build the custom world container:

```
docker-compose build compose-px4-sim-sdu-world
```

Run the custom world container with the case d) map:

```
docker run -name world -network host -id -rm vm-server-sdu-world-custom
17550 11311 case_d
```

No changes has been made to the drone container, so the same docker run command as listed previously is used.

To stop any Docker container:

```
docker stop <container-name>
```

Gazebo

To start Gazebo client that uses the port from the Docker containers:

```
GAZEBO_MASTER_URI=http://localhost:17550 gzclient -verbose
```

Control

The drones can be controlled through a program like QGroundControl [24], or using offboard control, which is how the drones are controlled in this project to fly autonomously. To enable offboard control for one drone:

```
<path-to-script>/./offb_posctl.py
```

This is a class that takes as input the ID of the drone container, defaults to sdu_drone_0, and has to be initiated for every drone container.

MAVROS

When `offb_posctl.py` is enabled, is will publish information about the local location to the MAVROS node `/<container-ID>/mavros/setpoint_position/local`. This information can be written to the terminal using the `rostopic echo` command:

```
rostopic echo /<container-ID>/mavros/setpoint_position/local
```

`offb_posctl.py` also provides services to control the drones from the terminal. Example, to enable flying in a circle:

```
rosservice call /setpoint_controller/circle "{}"
```

Appendix B

Source Code

These are the three folders that contain all the source code material for this project:

- `kinematic_simulation`
- `physics_simulation`
- `px4-simulation-docker` - folder for creating custom Docker image

They can all be cloned form:

`https://github.com/Nthom18/nthom18_bachelor_online_appendix.git`

The `kinematic_simulation` and `physics_simulation` folders are included in a .zip folder handed in with this report, named `nthom18_appendicies.zip`. The `px4-simulation-docker` folder is only found in the Github repository.

Appendix C

Test of Kinematic Simulation - Case c)

Introduction

The purpose of this experiment is to determine whether or not the implemented behaviours are successful in completing the criteria for application in case c) described in section 3.1 and restated here:

Case c) Data collection in the canopies of trees. Obstacles are now not to be only avoided, but are also objects of observation that the drones need to position themselves around. A small group of drones (2-3) will surround a tree and gather data for a while, before moving on to the next undocumented tree.

Theory

Four simulation obstacles, that in this case represents trees, are placed in the world, The scene is shown in figure C.1. Each tree are to be analyzed, this takes a fixed amount of time, but can be sped up if more drones are analyzing the same tree. Each tree has its own timer, and every frame that value is reduced by the number of drones analyzing that tree. A drone will only join another group analyzing a tree, if that group has less that three drones, unless it is the last tree remaining. The drones will be spawned on one end of the map, and have a random initial velocity.

There are two tests to be performed, one that studies the difference between the time taken for different flock sizes, and another more thorough test for one chosen flock size.

The parameters of interest are the number of frames it takes for all the trees to be analyzed,

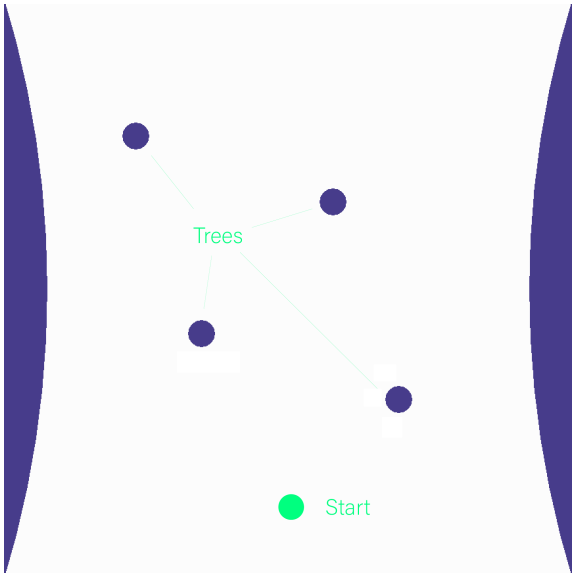


Figure C.1: Obstacle placement for case c).

as well as any collisions during the test.

Experiment

Setup

The folder used in this experiment is `kinematic_simulation`, see appendix B for instruction on where to find it.

Procedure

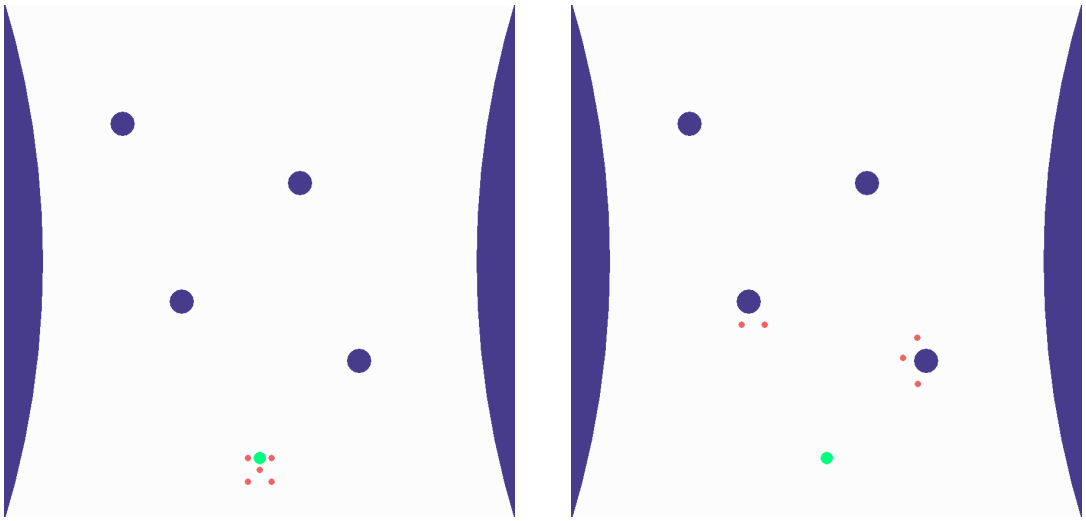
The first experiment is to test different flock sizes. Run the `test_c_sizetest.py` Python script in the folder `kinematic_simulation` with Python interpreter, for example write in Linux terminal: `python3 test_c_sizetest.py`. This will start and stop the kinematic simulation a specified number of times, and log parameters of interest to the appropriate `data_c_.csv` file in the sub folder `logs`. At the end the program will output the total number of collisions during the tests to the terminal.

To perform the second test run the `test_c.py` Python script in the folder `kinematic_simulation` with Python interpreter. This will also produce a new csv file, and output the number of collisions.

To plot the data in the csv files, run the Python scripts `plotCSV_c_sizetest.py` and `plotCSV_c.py` respectively. These scripts will also output the average for each flock size.

Results

On figure C.2 an example of the start and middle of the simulation is shown.



(a) Start of scenario (b) Simulation example after 100 frames

Figure C.2: Start position and example of the simulation after 100 frames in implementation of case c) with a flock size of five.

Different flock sizes are tested for this case. In figure C.3 four graphs are plotted, each representing the frames for completion for different flock sizes, namely of 1, 3, 5 and 7.

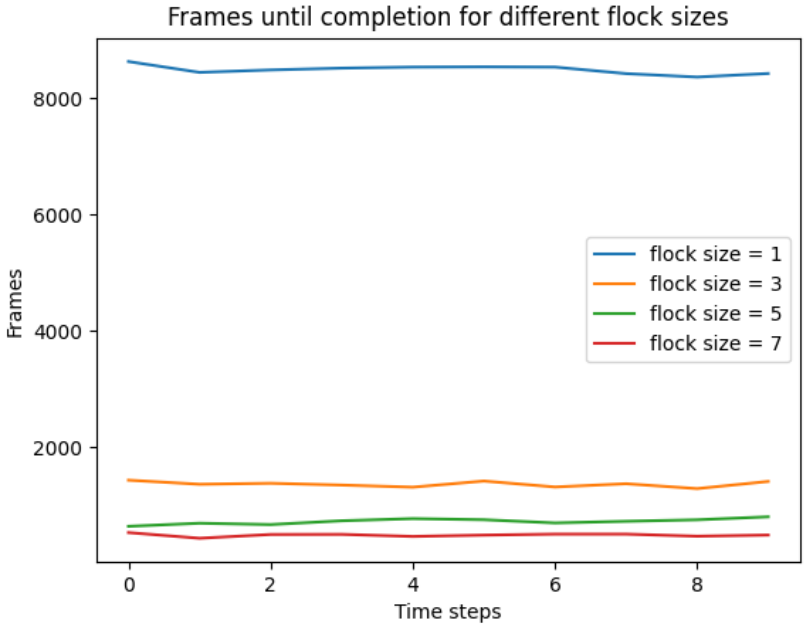


Figure C.3: Amount of frames before completion in 10 tests for different flock sizes.

It can be clearly seen that increasing the flock size from 1 to 3 greatly improves the time. From 3 to 5 is still considered magnificent, as it is almost halved. Increasing flock size beyond this point improves the time, but at a decreasing factor. As such the thorough test is performed with a flock size of five. The average frames of the tests and percentage decrease are shown in table C.1. 53.32

Flock size	Average	Percentage decrease from last step
1:	8483.8	-
3:	1369.3	-83.85%
5:	730.2	-46.67%
7:	495.6	-32.13%

Table C.1: The average frames for completion

On figure C.4 is the time, measured in number of frames, it took for the swarm to finish analyzing all four trees, over 50 tests. The graph shows that the amount of frames it takes are steady around the average of this test of 720.9. This is expected due to the constant starting positions.

At test 41 a collision occurs, and this has a great impact on the amount of time it takes to complete the task, approximately twice as much. However, due to the nature of swarm robotics the objective can still be completed.

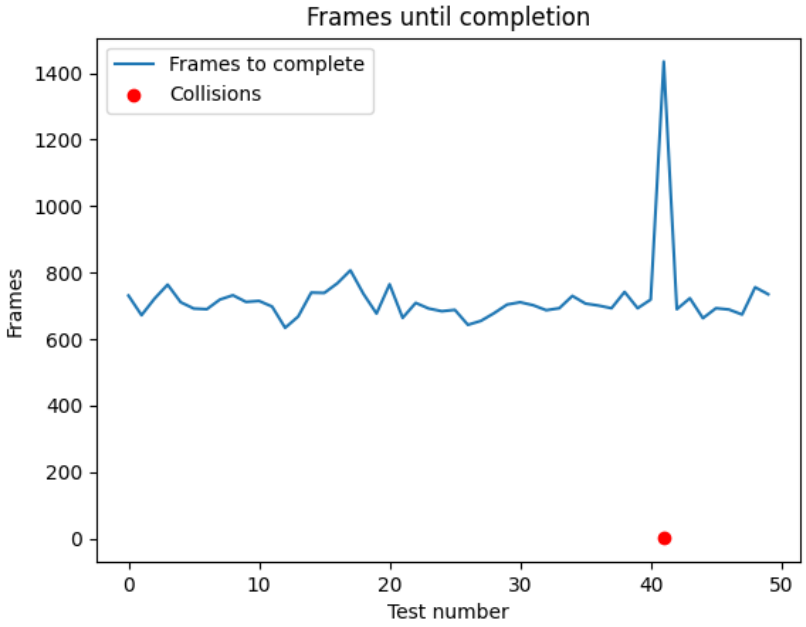


Figure C.4: Amount of frames before completion in 50 tests, and collisions.

The number of collisions during the 50 tests was 1. For each test there where 5 drones for a total of 250, this yields a collision percentage of:

$$\left(\frac{1}{250}\right) \cdot 100 = 0.4\% \tag{C.1}$$

This is an acceptable collision percentage.

Conclusion

In this test it is shown that it makes a huge difference whether a drone takes on this task alone as apposed to a swarm. It was found that for this scenario, analyzing four trees, a flock size of five is appropriate.

Should a collision occur, the time will increase, but the objective will still be completed by the remaining drones.

Appendix D

Test of Kinematic Simulation - Case d)

Introduction

The purpose of this experiment is to determine whether or not the implemented behaviours are successful in completing the criteria for application in case d) described in section 3.1 and restated here:

Case d) Moving a swarm of UAs from point-to-point in an unstructured environment.

> This scenario is a test of mobility of the swarm as a whole. Moving multiple drones simultaneously will test the Boids Algorithm, while also requiring a different protocol for moving the whole swarm.

Theory

The simulation obstacles are to be placed in a manner that allows for multiple passages through, but also presents narrow passageways. This will represent the unstructured environment that a small swarm of drones shall navigate. The scene is shown in figure D.1. The chosen size for the swarm is five. They will be spawned on one end of the map, and a goal will be placed on the other.

The parameters of interest to test are the distance from the drones to the goal over time, as well as any collisions during the test. The drones will have a limited time to reach the goal of 800 frames.

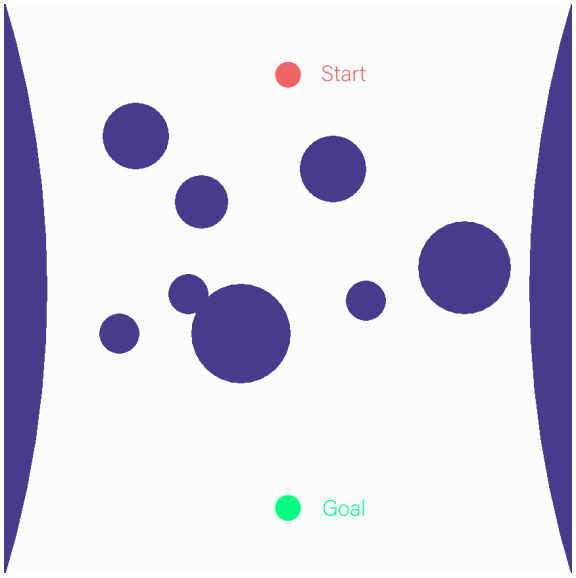


Figure D.1: Obstacle placement for case d).

Experiment

Setup

The folder used in this experiment is `kinematic_simulation`, see appendix B for instruction on where to find it.

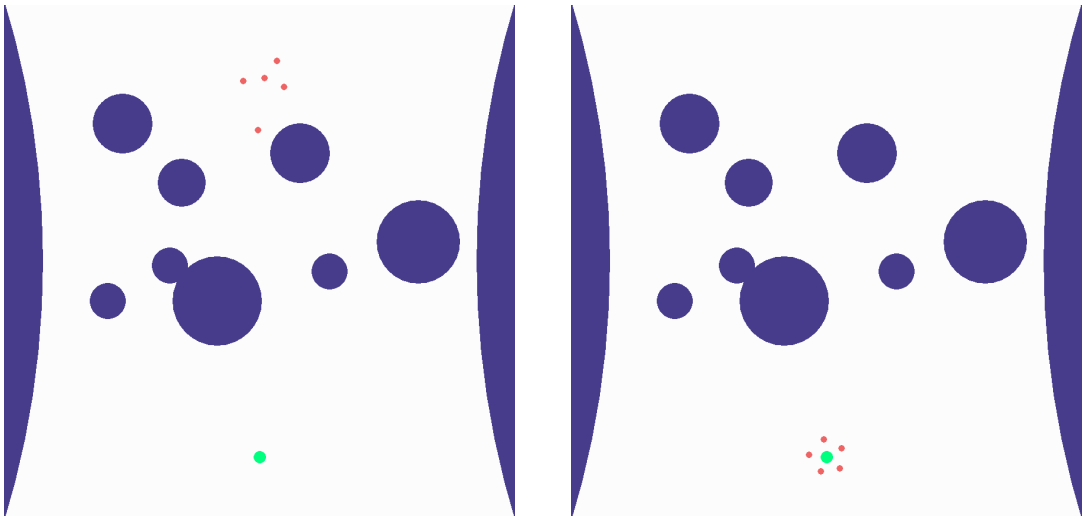
Procedure

To perform the test run the `test_d.py` Python script in the folder `kinematic_simulation` with Python interpreter, for example write in Linux terminal: `python3 test_d.py`. This will start and stop the kinematic simulation a specified number of times, and log the average data to the `data_d_combined.csv` file in the sub folder `logs`. At the end the program will output the total number of collisions during the tests to the terminal.

To plot the data in `data_d_combined.csv`, run the Python script `plotCSV_d.py`.

Results

The simulation was run 50 times, on figure D.2 an example of the start and end of the simulation is shown.

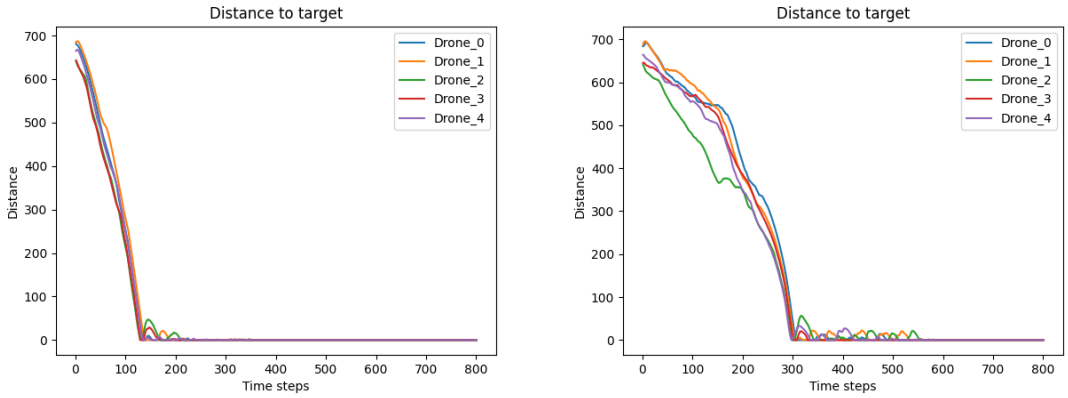


(a) Start of scenario (b) End of scenario

Figure D.2: Start and end of simulation implementing case d).

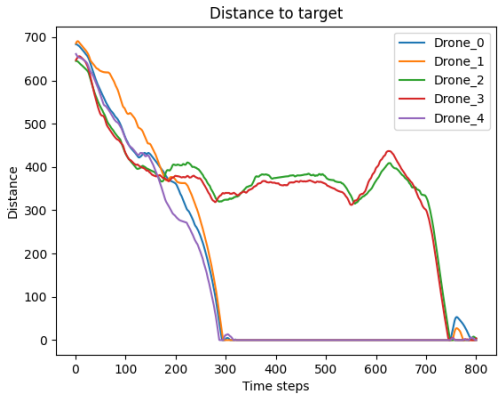
In figure D.3 six different plots are shown, these are the distance from each drone to the goal zone for six different individual tests, chosen from the pool of 50. In every test the drones start at the same location, but have random initial velocities to create variation.

Figure D.3a and D.3b shows that the drones are able to quickly find a path through the obstacles, and reach the goal as one whole flock. Figure D.3c and D.3d shows examples of the flock splitting up, one where it splits into two smaller flocks and the other where just a single drone gets separated from the rest. It is also clear form these plots that when the later flock reaches the goal-zone, the other drones move out of the way before settling back into a distance of 0. In figure D.3e one drones also splits from the flock, but this time it gets stuck and never reaches the goal-zone before the time runs out. In figure D.3f is an example of two drones having the same distance throughout that whole test, indicating that those two collided.

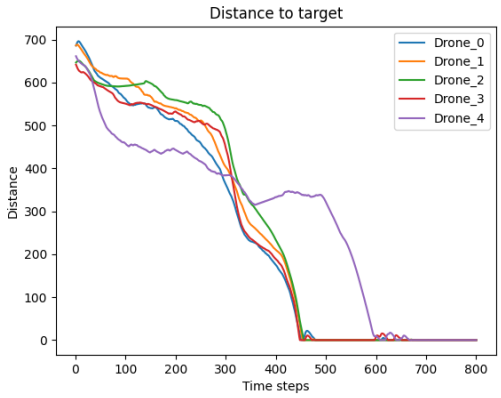


(a) Test 5

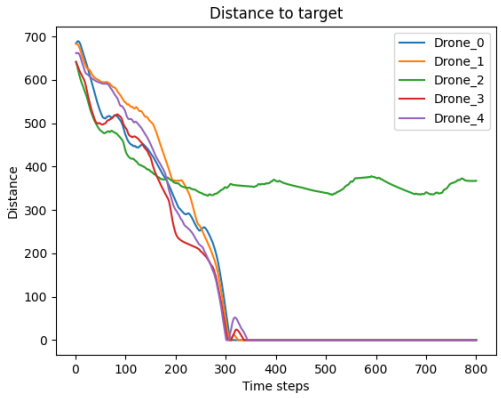
(b) Test 11



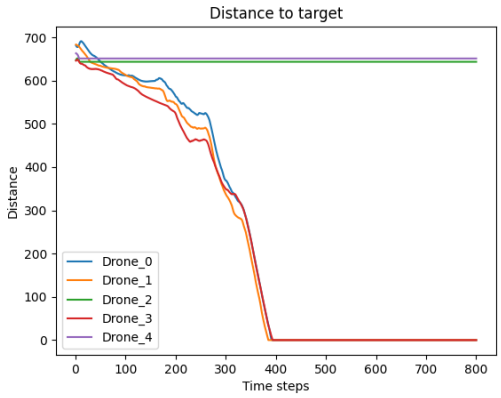
(c) Test 20



(d) Test 49



(e) Test 40



(f) Test 30

Figure D.3: Handpicked plots from the pool of 50 tests, that show different outcomes.

Figure D.4 is the average distance for each drone over the course of the 50 simulations. From this graph the trend seems to be that the drones will find a collision free path to the goal-zone as a flock, or divided into sub-flocks.

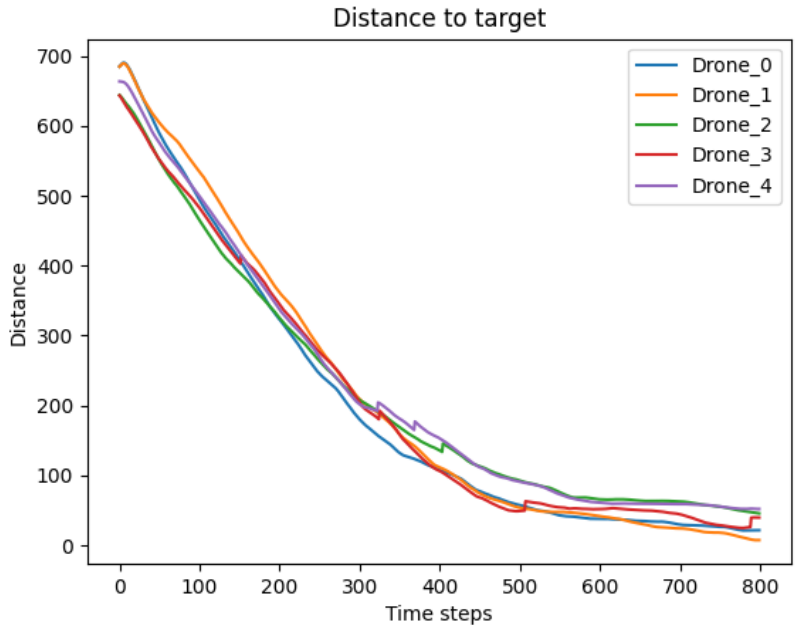


Figure D.4: Average distance for each drone in a flock of five over time, the unit of time is frames.

The number of collisions during the 50 tests was 3. For each test there where 5 drones for a total of 250, this yields a collision percentage of:

$$\left(\frac{3}{250}\right) \cdot 100 = 1.2\% \tag{D.1}$$

This is a reasonably low number, but for deployment on real hardware platforms this would be too high, and would need further improvement.

Conclusion

This test demonstrate that a flock of five drones can in a simple kinematic simulation navigate an obstacles course as a flock to reach a goal, with few collisions.

Decreasing the collision percentage of 1.2% would be mandatory before implementing it on a real system.