VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF AEROSPACE ENGINEERING

DEPARTMENT OF AERONAUTICAL ENGINEERING

Azar Musayev

RESEARCH ON THE ROTORCRAFT UAV SWARM POSSIBILITES

Master‘s degree Thesis

AEROSPACE ENGIEERING, state code 6211EX060

Aerospace Engineering_specialisation

Aerospace Engineering_study field

Vilnius, 2023

VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

FACULTY OF AEROSPACE ENGINEERING

DEPARTMENT OF AERONAUTICAL ENGINEERING

Azar Musayev

RESEARCH ON THE ROTORCRAFT UAV SWARM POSSIBILITES

Master‘s degree Thesis

AEROSPACE ENGIEERING, state code 6211EX060

Aerospace Engineering_specialisation

Aerospace Engineering_study field

Supervisor _____Doc. Dr. Darius Rudinskas_____
(Title, Name, Surname)

Consultant _____
(Title, Name, Surname)

Consultant _____
(Title, Name, Surname)

OBJECTIVES FOR MASTER THESIS

In the master`s final thesis, conduct a study of the flight behavior of various swarms based on the formation of UAVs. Determine how the UAV swarm avoids obstacles. Conduct research using a virtual environment.

Vilnius Gediminas Technical University
Antanas Gustaitis' Aviation Institute
Department of Aeronautical Engineering

ISBN | ISSN
Copies No. .....
Date .....-.....-.....

Master Degree Studies **Aerospace Engineering** study programme Master Graduation Thesis

Title	**Research on the Rotorcraft UAV Swarm Flight Possibilities**
Author	**Azar Musayev**
Academic supervisor	**Darius Rudinskas**

Thesis language: English

Annotation

This Master's thesis presents a study of UAV swarm possibilities. Different tools such as Robot Operating System framework, GAZEBO and Rviz simulator has been used in order to test and analyze different UAV swarm topologies in simulated environment. Mapping technique used to map the environment simultaneously during flight. Two different approaches implemented for navigation purpose such as global and local path planner and social proximity layer technique used for collision avoidance. Four different topologies has been implemented in order to compare behavior of UAV swarm possibilities such as individual decision making, leader follower, predecessor and two nearest predecessor topologies. All of these topologies' performance has advantages and disadvantages according to results extracted from log files by plotting graphs. UAV swarm flight performance has been improved by giving different input parameters such as velocity, acceleration, cost map and searching radius. Different input parameters contributed to improve flight performance in terms of execution time, blocking time etc.

Structure: introduction, applications of UAV swarm, classification of UAVs, sensors, UAV swarm communication architecture, path planning, collision avoidance, obstacle avoidance, conclusions and references.

Thesis consist of: 78 p. 53 figures, 8 tables and 32 bibliographical entries.

Keywords: Unmanned Aerial Vehicle, swarm, obstacle avoidance, collision avoidance, global path planning, local path planning, social proximity layer, costmap, infilation radius, searching radious.

Table of Contents

INTRODUCTION 9
1. LITERATURE REVIEW 10
1.1. UAV swarm applications 10
1.2. Classification of UAV 12
1.3. UAV sensors analysis 15
1.4. UAV swarm communication architecture 18
1.5. UAV swarm control architecture. 22
Conclusion 26
2. METODOLOGY 29
2.1 Robot operating system (ROS) 29
2.2. Mapping of the environment. 31
2.3. Path planning 36
2.4. Collision Avoidance 40
Conclusion 41
3. PRACTICAL PART 42
3.1. Overall system description 42
3.2 Mapping 50
3.3 Navigation 55
3.3 Collision Avoidance 59
3.4 Formation-based Leader-Follower Control 61
3.5 Comparison of different formation-based swarm 67
3.6. Compare different parameters using LF topology 71
3.7 Conclusion 74
CONCLUSIONS 75
REFERENCES 76

List of Figures

Fig. 1. Drones with fire extinguishers. 10
Fig. 2. Bayraktar TB-2 11
Fig. 3. Classification of the UAV system. 12
Fig. 4. DJI Phantom 4 Quadcopter. 14
Fig. 5. UAV autonomous system 18
Fig. 6. Infrastructure (GCS) based on swarm architecture. 20
Fig. 7. FANET swarm architecture. 21
Fig. 8. Cellular network UAV swarm architecture. 22
Fig. 9. Deliberative architecture 23
Fig. 10. Reactive architecture 24
Fig. 11. Hybrid architecture 25
Fig. 12. The Behavior Control Architecture 26
Fig. 13. ROS structure. 30
Fig.14. System overview of HECTOR SLAM 33
Fig. 15. Global and local path planning. 36
Fig. 16. System diagram for multi-UAV collision avoidance 42
Fig. 17. Field of view 43
Fig. 18. Hector quadrotors 44
Fig. 19. Important ROS launches and scripts. 45
Fig. 20. Launch command 45
Fig. 21. Initial positions of drones. 46
Fig. 22. Lunch Rviz function 46
Fig. 23. Four quadrotor installed with 2D LIDAR on a Gazebo Simulator 47
Fig. 24. mapping_and_navigation_multi_uav_4.launch 48
Fig. 25. Launch command for flight test. 49
Fig. 26. Logger files for extracting data. 49
Fig. 27. Hector SLAM parameter configuration 50
Fig. 28. Different map generated by Hector SLAM with different free and occupied update factor parameters. (a) 0.45 and 0.8 (b) 0.25 and 0.8 (c) 0.45 and 0.9 51
Fig. 29. Code snippet of quadrotor_move_base.launch. 52
Fig. 30. Costmap parameters configuration 54
Fig. 31. Global costmap using different parameters 54
Fig. 32. Global path. 55
Fig. 33. A* parameters. 56
Fig. 34. DWA parameters 58
Fig. 35. Execution time between different velocity limit by using individual control 58
Fig. 36. Social proximity process. 60
Fig. 37. Several topologies for leader-follower method 61
Fig. 38. Leader control algorithm pseudo code 62
Fig. 39. Code snippet for leader control. 62
Fig. 40. Follower control algorithm pseudo code 63
Fig. 41. How each leader-follower topology works. 63
Fig. 42. Snippet codes for follower control 66
Fig. 43. The average mission time among different topologies 68
Fig. 44. The average blocking time among different topologies 68
Fig. 45. The average distance to neighbors among different topologies 69
Fig. 46. The accumulative trajectory length among different topologies 69

Fig. 47. The average distance to nearest obstacle among different topologies 70
Fig. 48. The trajectory of using different topologies..... 70
Fig. 49. The average mission time among different parameters 71
Fig. 50. Blocking time among different parameter 72
Fig. 51. The average distance among different parameters..... 72
Fig. 52. Average trajectory length among different parameters..... 73
Fig. 53. Average distance to nearest obstacle among different parameters 73

List of Tables

Table 1. Classification of UAV depend on number of propeller..... 13
Table 2. UAV types with main characteristics. 14
Table 3. UAV sensors used in agriculture monitoring. 15
Table 4. Sensors with detection and avoidance capabilities..... 17
Table 5. General characteristics of UAV swarm communication architectures..... 27
Table 6.General characteristics of UAV swarm control architectures. 27
Table 7. Initial and goal positions of drones. 46
Table 8. Performance of different formation-based swarms. 74

INTRODUCTION

Relevance of the topic. Recent attention has been focused on unmanned aerial vehicle (UAV) technologies due to their increasing military and commercial applications. It has been discovered that swarms of UAVs execute certain duties, such as tracking, surveillance, path planning, and coordination, significantly more effectively and with superior operating parameters than applications employing a single UAV. A fleet of collaborating drones generates novel problems that cannot be resolved with a single UAV deployment. Swarms that operate as a single entity necessitate techniques for averting collisions both within the swarm and with external obstacles. Failure of these systems can cause bodily injury and increase manufacturing costs.

Problem – Collision avoidance, obstacle avoidance and communication are critical problems for UAV swarm flight performance. However, these are still not fully solved in this industry.

Research object – Possibilities of rotorcraft UAV swarm possibilities in indoor environment.

Aim – to develop and evaluate behaviour of different UAV formation-based swarm flights.

Tasks to solve the problem.

1. Formation based swarm of unmanned aerial vehicles,
2. Avoid from obstacles,
3. Avoid from collision between other UAVs
4. Shortest path planning.

Research Methods.

Establish virtual environments and simulate the behavior of the unmanned aerial vehicle swarm through the utilization of simulation tools or platforms, such as ROS, Rviz, and Gazebo. Execute the obstacle avoidance algorithm that was formulated and evaluate its efficacy across varying scenarios within the simulation environment.

1. LITERATURE REVIEW

In this chapter, some essential topics such as UAV swarm communication architectures, applications for both military and commercial purposes and control techniques will be discussed in more detail.

1.1. UAV swarm applications

Recent advancements in the sensory technology integrated into unmanned aerial vehicles (UAVs) have facilitated the emergence of novel unmanned operational services and applications, thereby expanding the potential use cases for drones. This section provides a brief summary of the primary use cases for unmanned aerial vehicles (UAVs).

Historically, unmanned aerial vehicles (UAVs) have been employed for the purpose of conducting military surveillance missions. Unmanned aerial vehicles, commonly known as drones, have exhibited remarkable versatility and cost-effectiveness in various sectors, such as geophysics and agriculture, for conducting aerial surveys, monitoring activities, and performing surveillance tasks. A monitored structure or setting may require revisions in response to any detected motion occurring beyond regular business hours. To ensure comprehensive surveillance, a substantial workforce would be required for monitoring a vast building or area manually. On the contrary, a group of unmanned aerial vehicles has the potential to provide superior coverage or surveillance of a given region with minimal human intervention, as it can promptly notify the ground station of any detected motion. (Liu,2019)

Unmanned aerial vehicles (UAVs) possess the capability to rapidly and securely penetrate disaster zones that would otherwise pose a risk and prove to be arduous to approach in the absence of a calamity. Consequently, they are capable of aiding in disaster assessments and the establishment of effective safeguards.

Fig. 1. Drones with fire extinguishers.

(https://dronelife.com/2021/04/28/drone-swarms-for-firefighting-the-future-of-fire-

supression/)

In the event of a wildfire, a considerable area can be rapidly surveyed and controlled without endangering human lives by employing a swarm of drones equipped with fire extinguishers or similar tools.

The capabilities of swarm systems have direct applicability to intelligence and surveillance operations. Swarms have the capability to be strategically positioned to conduct surveillance on a specific target and its environs, while simultaneously tracking stationary or mobile entities. This pertains to fundamental activities such as surveillance of a suspected adversary establishment and monitoring the vicinity for incoming or outgoing automobiles. The utilization of a swarm of drones enables the acquisition of images from multiple locations in a simultaneous manner, while also facilitating the survey of a vast expanse of land, thereby reducing the time required for an individual drone to execute a task (Hambling,2020). The capability renders it advantageous in expeditiously capturing recurrent images across an extensive area while simultaneously monitoring alterations in activity at a specific location.

Swarms have demonstrated efficacy in performing focused offensive operations. Despite Armenia's numerical and technological advantages, Azerbaijan emerged victorious in the 2020 Karabakh War by defeating Armenian forces. The primary reason for this outcome is attributed to the utilization of diverse armed drones by Azerbaijan troops, including the Turkish Bayraktar TB-2. The aforementioned drone effectively neutralized 40 fighter aircraft and over 250 armored vehicles.

Fig. 2. Bayraktar TB-2
(https://www.yenisafak.com/ekonomi/bayraktar-tb2-16-ulkede-ucuyor-3755997)

The military capabilities of Armenia's land and air defenses were significantly impaired by the swarms, prompting the Azerbaijani troops to initiate motorized infantry

operations aimed at recapturing the contested region. A group that initially appeared to be in a weaker position utilized 16 swarms to enhance its offensive capabilities. The swarms had a debilitating impact on Armenia's ability to engage in ground and air warfare, as they caused extensive damage to numerous assets.

1.2. Classification of UAV

In contemporary times, with the progression of technology, there has been a surge in the utilization of unmanned aerial vehicles, commonly known as drones, for various purposes such as transportation of food, production of captivating cinematography, and military operations. The payload is a crucial factor that necessitates consideration during the development of a drone. The payload refers to the supplementary mass that an individual drone has the capacity to elevate (Jang, 2020). A drone designed for delivery purposes is capable of carrying a maximum payload of 25 kg, thereby enabling transportation of packages weighing up to the aforementioned weight.

UAVs are available in various forms and may be utilized in a variety of situations. There are numerous different categorization methods, as seen in Fig.1.

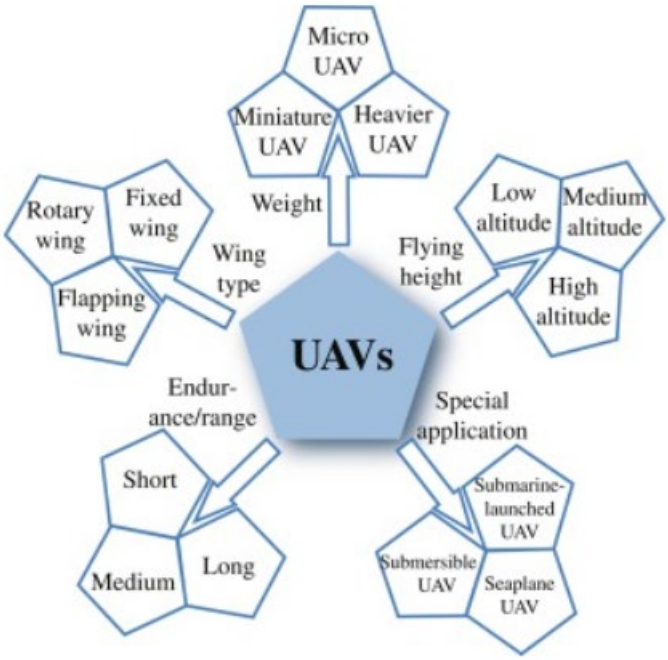


Fig. 3. Classification of the UAV system.

(https://www.sciencedirect.com/topics/engineering/unmanned-aerial-vehicle

Drones can be classified based on various factors such as size, range, and technology. The dimensions of objects can be classified into four categories: nano, small, standard or

huge. Similarly, the extent of a range can be categorized as extremely close, close, short, mid or endurance. Drones have the potential to be equipped with various technological features such as cameras, stabilizers, sensors and Global Positioning Systems (GPS).

According to the information presented in Table 1, drones can be classified into four primary classifications, namely fixed-wing, fixed-wing hybrid, single rotor, and multirotor. The predominant application of fixed wing unmanned aerial vehicles (UAVs) is for the purposes of conducting aerial mapping and inspection. These items are expensive and necessitate specialized training for operation (Chand, Mahalakshmi, Naidu 2019). Despite requiring additional space for takeoff and landing, they possess the capability to cover a broader expanse. This particular drone model is deemed unsuitable for typical aerial photography applications due to its lack of vertical takeoff and landing (VTOL) or hovering capabilities. Nevertheless, in case they are propelled by gasoline-powered engines, they can persist in the atmosphere for up to sixteen hours.

Table 1. Classification of UAV depend on number of propeller

Drones	Number of propeller
Tricopter	3
Quadcopter	4
Hexacopter	5
Octocopter	8

On the contrary, fixed-wing hybrid unmanned aerial vehicles integrate automated and manual gliding techniques. At present, the subject in question remains in the developmental phase and exhibits limited proficiency in both hovering and forward flight. The utilization of delivery drones is incorporated by Amazon in its business operations. In contrast, single rotor unmanned aerial vehicles (UAVs) exhibit intricate mechanical configurations and operational hazards such as vibration and sizable rotating blades. Consequently, the operator necessitates proficiency training. These devices are costly and possess the ability to maneuver larger loads such as LiDAR sensors. They have the potential to be powered by a combustion engine, thereby increasing their overall durability.

Table 2. UAV types with main characteristics.

Drones	Main characteristics
Fixed-Wing	long endurance and fast flight speed
Fixed-Wing Hybrid	VTOL and long endurance flight
Single Rotor	VTOL, hover and long endurance flight
Multirotor	VTOL, hover and short endurance flight

Multirotor unmanned aerial vehicles (UAVs) are considered to be a cost-effective and relatively simple option for construction. Unmanned aerial vehicles (UAVs), commonly referred to as drones, are frequently employed for routine activities such as aerial photography and video monitoring (Plathottam, 2018). Various types of unmanned aerial vehicles, such as tricopters, quadcopters, hexacopters, and octocopters, are viable options for a range of tasks. (Table 1). Nonetheless, the limited velocity, flight range, and energy efficiency of such unmanned aerial vehicles render them unsuitable for conducting extensive aerial cartography and long-range monitoring.

Fig. 4. DJI Phantom 4 Quadcopter.
(https://www.bhphotovideo.com/explora/video/buying-guide/introduction-drones-and-uavs)

Quadcopter is one of these multirotor drones with four rotors. Every rotor has its own

motor and propeller. Swarm of quadcopter will be analyzed in next chapters in this research. DJI Phantom 4 Quadcopter is shown as an example in photo 3.

1.3. UAV sensors analysis

Unmanned Aerial Vehicles (UAVs) operate within the aerial environment. Aircrafts are required to ascertain their location and attitude, ground and air speed, angle of attack, and barometric pressure. Additionally, they may need to acquire their location data or exchange it with other aircrafts. UAVs' flight location and orientation can be determined by employing accelerometers in conjunction with tilt sensors and gyroscopes. Subsequently, the flight control system is furnished with the aircraft's position and orientation data to maintain its horizontal flight. Flight routes and directions are regulated through the utilization of inertial measurement units (IMUs) in conjunction with the Global Positioning System (GPS)/Global Navigation Satellite System (GNSS). The GPS/GNSS signal may exhibit instability in areas with dense forestation, urban canyons, and enclosed spaces. This is due to various factors that can cause interference or jamming, resulting in a weak or lost signal. As a result, many indoor UAV systems utilize optical cameras, often in conjunction with other technologies such as ultrasonic (US) technology.

Table 3. UAV sensors used in agriculture monitoring. (Jang, G. Review: Cost-Effective Unmanned Aerial Vehicle (UAV) Platform, 2020)

Functions	Sensors	Specifications
Detailed digital terrain and surface models; penetrating vegetated landscapes	LiDAR	Range from 100 m to 340 m FOV (deg) (Vertical) 20, 30, 40 (Horizontal) 360 Accuracy from 1 to 3 cm Weight from 0.59 kg to 3.5 kg
Landscape matrix contrast detection	Multispectral camera	Resolution 1280 × 960; 1280 × 1024; 2048 × 1536; 2064 × 1544 Spectral range Blue, Red, Green, Near-infrared, Red-edge, long-wave infrared
Landscape matrix contrast detection	Hyperspectral camera	Resolution 640 × 640; 640 × 512;1024 × 1024; 2048 × 1088 Spectral range from 380 nm to 13,400 nm Weight from 0.45 kg to 2 kg
Detection of measurably distinct variations between the features and their soil matrix	Thermal camera	Resolution 160 × 120; 320 × 240; 320 × 256; 336 × 256; 382 × 288; 640 × 480; 640 × 512 Accuracy (±°C) from 1 to 5 Spectral range from 7 nm to 14 nm Weight from 39 g to 588 g

Table 3 presents a selection of UAV sensors utilized in various fields, including agriculture for crop monitoring and management, archaeology for site visualization, excavation documentation, and aerial reconnaissance, as well as for general purposes to

ensure UAV sensing and avoidance capabilities (Jang, G.2020). RGB digital cameras offer high spatial resolution measurements of radiation values within the red, green, and blue spectral bands. The spatial resolution of the RGB sensor is a determining factor in the quality of the captured images. Through the analysis of aerial photographs captured by a camera equipped with this particular sensor, it is possible to obtain measurements related to plant area, plant height, and color indices. Spectral sensors are utilized to collect data by monitoring the light that is reflected, emitted, and transmitted from the objective. These sensors are categorized as either multispectral or hyperspectral, depending on the number of frequency bands and the width of each band. Thermal sensors generate visual representations by detecting and capturing the electromagnetic radiation emitted by an object within the infrared (IR) wavelength spectrum. Remote sensing technologies are employed in the agricultural sector owing to their capacity to provide information on plant surface temperature and crop water stress index. Multispectral cameras offer notable advantages over RGB cameras in the agricultural domain due to their ability to capture additional information, including the detection of imperceptible physiological changes in plants. Compared to other sensor types, RGB cameras have the potential to offer superior spatial resolution. The utilization of hyperspectral cameras in agriculture applications is not common due to their weight and size, as well as their need for integration with additional equipment, such as a battery, frame grabber, and data storage device, to ensure proper functionality on UAV platforms. The utilization of hyperspectral sensors is expected to increase due to their continued miniaturization, leading to a greater number of tasks that may incorporate them. Variations in atmospheric conditions and the existence of objects that emit or reflect thermal infrared radiation can potentially undermine the precision of thermal camera data. As a result, regular calibration is required to ensure accuracy.

LiDAR sensors utilize illumination to target a specific point and analyze the reflected light in order to determine the distance to said point. This particular sensor has the potential to provide a wide-ranging field-of-view (FOV), denoting the extent of the observable area, while simultaneously exhibiting exceptional precision. Nevertheless, with regards to the requirements of UAV payloads, the dimensions and mass could potentially pose a significant issue. LiDAR sensors are utilized in archaeological contexts to gather information on the impact of ancient artifacts buried beneath the surface on the topography of the surrounding terrain. They have the capability to offer extensive digital representations of terrain and surface features. Spectral sensors provide measurements in the field of archaeology that facilitate the identification of variations in the terrain matrix, thereby aiding in the assessment of the significance of an archaeological site. Thermal sensors have the potential to be utilized

in the field of archaeology for the purpose of gathering pertinent information through the analysis of recorded readings. The presence of static and/or dynamic obstructions poses a significant challenge that poses a direct threat to the dependability of unmanned aerial vehicles (UAVs), particularly those that operate at lower elevations. It is imperative for all Unmanned Aerial Vehicle (UAV) applications to consider these factors. Table 4 presents a selection of sensors and their respective detection ranges that have the potential to facilitate unmanned aerial vehicles (UAVs) in acquiring information pertaining to the presence of obstacles and environmental factors.

Table 4. Sensors with detection and avoidance capabilities

Sensors	Detection range
Radar	35 km
LIDAR	15 km
Electro-optic sensor	20 km

Radars are capable of detecting obstacles by emitting electromagnetic waves that propagate at the speed of light on a continuous basis (Adamopoulos,2020). When waves are emitted and subsequently reflected back towards an obstacle, the presence of said obstacle is detected. The temporal duration between the waves that are emitted and subsequently reflected is utilized to compute the distance of the obstacle.

Radar technology is capable of expeditiously scanning a designated area and possesses a wide-ranging capacity for detection. Despite having a shorter detection range compared to radar, LiDAR offers the advantage of furnishing data on both the range and distance of impediments.

Electro-optic sensors possess the capability to ascertain the obstruction's elevation and azimuth through employment of a camera. However, they lack the ability to provide data regarding the distance or velocity of the obstruction. The efficacy of electro-optic sensors is notably influenced by meteorological conditions, in contrast to radar and LiDAR.

Ultrasonic sensors achieve target detection through the emission of sound waves. Despite its cost-effectiveness and compact size, this method's precision is compromised, and its limited scope may lead to blind spots during target identification operations. Furthermore, ultrasonic sensors are employed for the purpose of detecting obstacles, often in conjunction

with other sensors such as visual cameras, which are utilized to detect obstacles such as scattered rocks that may not be accurately mapped, or thermal sensors, which are used to account for the impact of temperature fluctuations on the accuracy of distance detection.

1.4. UAV swarm communication architecture

A swarm is commonly defined as a group of entities that collaborate to produce a significant or favorable result or conduct. Numerous occurrences of collective behavior, known as swarming, can be observed in the natural world. The collective efforts of bees are crucial for the sustenance of their colony. Migratory geese exhibit efficient aerial coordination to successfully complete their journey. A swarm of UAVs refers to a group of unmanned aerial vehicles that operate in a coordinated manner to accomplish a specific mission or a predetermined set of tasks (Zhang, 2020). The degree of autonomy exhibited by unmanned aerial vehicles (UAVs) is subject to variation. The degree of self-governance exhibited by a vehicle is determined by its capacity to perform tasks, coordinate actions, and make decisions without the need for human intervention. It is conceivable that a collective of unmanned aerial vehicles (UAVs) could potentially attain a sufficient degree of self-governance. A Cyber-Physical System (CPS) can be classified as an Unmanned Aerial Vehicle (UAV) swarm. The pivotal characteristic of an autonomous system is its ability to make decisions independently, without human intervention. The operation and movement of an unmanned aerial vehicle (UAV) is completely managed and directed by a human operator who is responsible for making decisions related to the UAV's mission accomplishment. Algorithms can make decisions within a fully autonomous system. Algorithms are capable of making decisions within a fully autonomous system. The decision-making paradigm employed by an autonomous CPS comprises three key steps, namely data, control, and process. The decision-making process of a swarm of unmanned aerial vehicles (UAVs) would adhere to the paradigm illustrated in Figure 5.

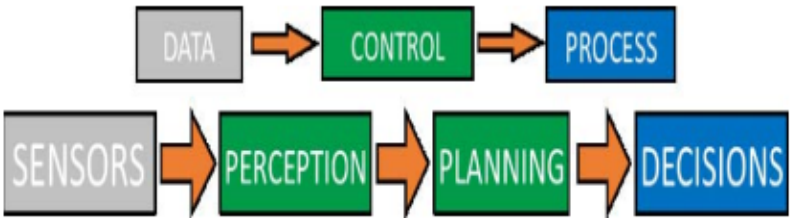


Fig. 5. UAV autonomous system

(S.Plathottam and P. Ranganathan, “Next Generation Distributed and Networked Autonomous Vehicles Review”)

The sensor-based data component of the paradigm. Sensors are utilized to collect pertinent data regarding the surrounding environment in which a specific task is to be executed, and subsequently transmit this information to a computer system to facilitate its execution. UAV swarm sensors may comprise a range of components, such as GPS, airspeed, sound sensors, cameras, and other relevant devices, contingent upon the specific application. The control stage comprises the subphases of perception and planning.

Transforming confusing data into informative information defines perception. In order to make a choice to carry out the activity, planning is the process of utilizing the perceived information. Most of the time, a GCS is simultaneously in charge of each individual UAV. A computer performing as a GCS and run ground control software is being utilized to control UAV swarms. The computers have a transceiver that transmits and receives telemetry information from linked UAVs. GPS data, groundspeed, and other metrics gathered by payload sensors are examples of telemetry data. These transceivers mainly transmit and receive data using unlicensed Radio Frequency (RF) channels like 900MHz. Drones with higher degrees of autonomy would be able to use their internal processing capability to make decisions (Gualda,2019). One of two common swarm communication architectures used in current UAV swarm demonstrations are infrastructure-based swarm architecture and ad hoc network-based architecture.

A ground control station (GCS) that collects sensor values from each drone in the swarm and transmits orders back to each UAV independently makes up the infrastructure-based architecture. Sometimes the GCS sends orders to the flight controllers of each UAV in real-time communication with the individual drones. In other situations, a flight operation is pre-programmed on each UAV, and the separate flight plans of every UAV are simultaneously performed with the GCS in this case duties of GCS is only about to observe the system (Navilli,2020). These UAV swarms are regarded as being semi-autonomous since they still need guidance from a centralized controller in order to execute the given mission.

The most popular swarm design for UAV swarms is infrastructure-based. The foundational infrastructure-based swarm capabilities are already present in GCS software. Infrastructure-based swarming has the benefit that optimization and calculations may be carried out in real time by a GCS through a higher speed computer than could be carried on a UAV. Additionally, there is no requirement for drone networking.

Infrastructure-based swarm designs rely on the GCS to manage drone coordination. Lack of system redundancy is brought on by this reliance. The working principle of the entire swarm is endangered in the case of an attack or a failure to any GCS activity. Additionally, infrastructure-based techniques need for all UAVs to be in the GCS's propagation range.

Unlicensed RF communications have the disadvantage that they may be subject to interference. (Argyriou,2018) The technology required to establish dependable connection with an infrastructure may restrict the value of infrastructure-based swarms due to the small payload capacity of UAVs. Lack of dispersed decision-making is another negative. In an infrastructure-based design, the GCS sets up all UAV decision-making based on calculations and algorithms created in the GCS. Figure 6 displays a swarm design that is built on infrastructure.

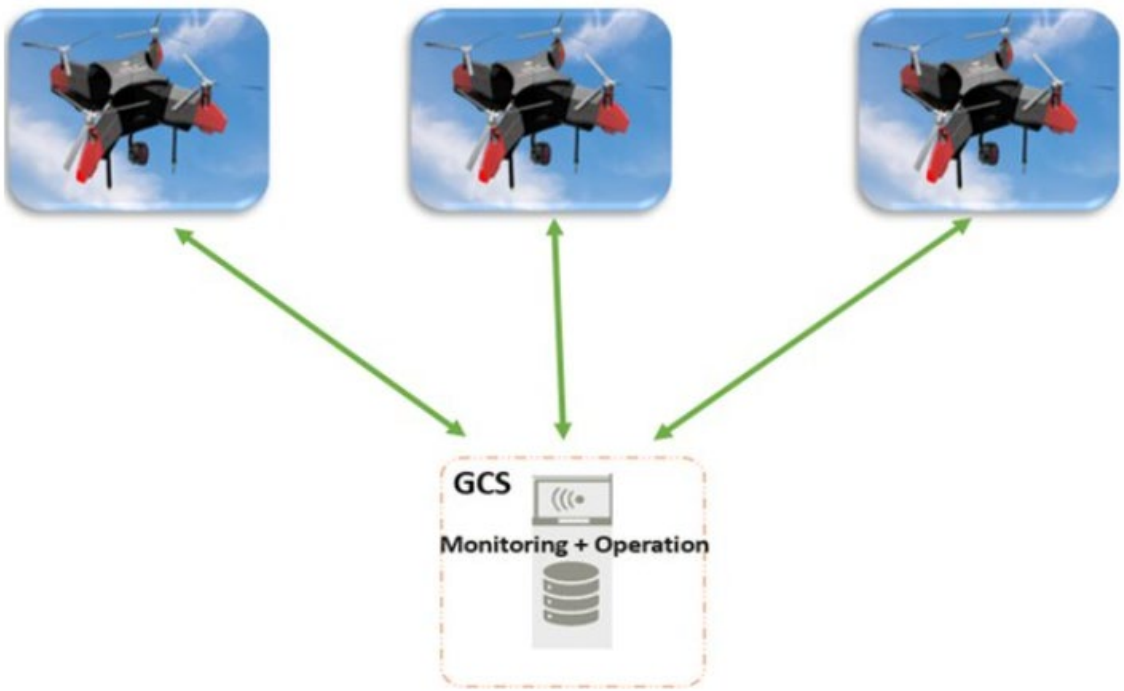


Fig. 6. Infrastructure (GCS) based on swarm architecture.
(https://www.researchgate.net/publication/365780355_A_Proposed_System_for_Multi-UAVs_in_Remote_Sensing_Operations/figures?lo=1)

A single network is proposed for the usage of Flying Ad-Hoc Networks (FANETs) to coordinate drone communication. A wireless ad hoc network (WANET) is a type of wireless network that is not built using preexisting infrastructure. Adhoc networks don't need routers or access points. Instead, depending on dynamic routing algorithms, nodes are assigned and reassigned dynamically (Smith, 2019). In a FANET, a system of communications is created between the UAVs that includes all the UAVs. UAVs may communicate in real time thanks to this network.

As compared to an infrastructure-based decision engine, direct communication among

UAVs necessitates dispersed decision making (Aguilera,2019). As the total swarm does not rely on a structure to carry out the necessary duties, this also offers built-in redundancy. The main benefit of FANETs is this. Each UAV needs networking devices in order to create a FANET. In a FANET, the maximum distance that UAVs may successfully communicate with one another is a restriction on its application. Additionally, dynamic route reconfiguration for UAV swarm applications is a difficult process that might cause packet loss. Establishing a trustworthy FANET is a challenge for situations where precise data transmission between UAVs is essential. Figure 7 displays a block schematic of a FANET.

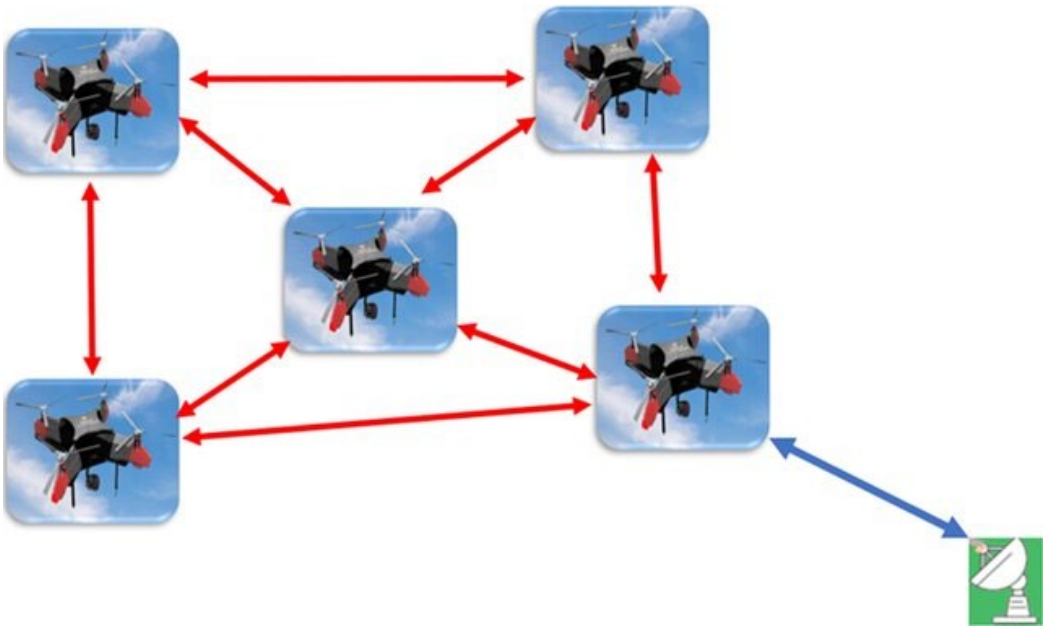


Fig. 7. FANET swarm architecture.
(https://www.researchgate.net/publication/365780355_A_Proposed_System_for_Multi-UAVs_in_Remote_Sensing_Operations/figures?lo=1)

This research suggests a hybrid infrastructure-based network that uses cellular network infrastructure while also building network protocol amongst drones without the assistance of a GCS. The suggested UAV swarm design makes use of both systems' advantages while minimizing some of its disadvantages.

The proposed architecture is a modification of an infrastructure-based ad hoc network. The infrastructure specifically supports comprehensive UAV-to-UAV communication, in which each UAV's telemetry is sent to every other UAV over cellular mobile infrastructure. Although communication is relayed over infrastructure, unlike a pure FANET, it is similar to a FANET in that the infrastructure does not make any choices. Instead, decision-making is divided among the UAVs and the infrastructure serving just as a means of data transmission.

Figure 8 displays a block diagram of the suggested architecture.

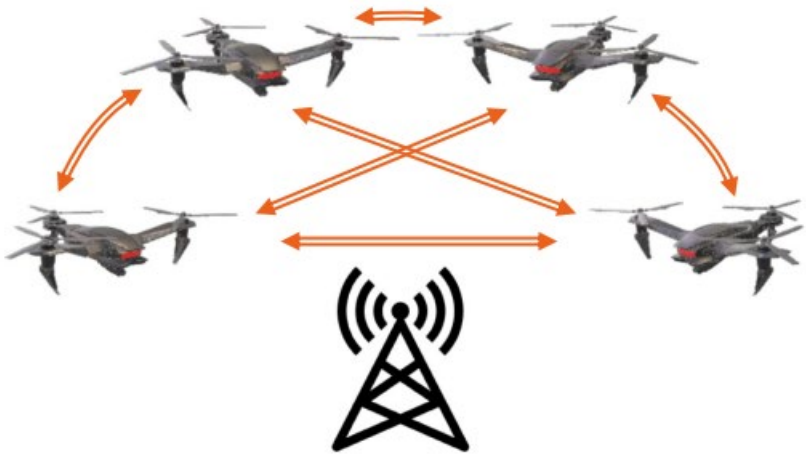


Fig. 8. Cellular network UAV swarm architecture.
(https://cdnsciencepub.com/doi/pdf/10.1139/juvs-2018-0009)

We explore a case where a swarm of cellular-connected unmanned aerial vehicles (UAVs) must work together to monitor a target area and transmit the sensing data to a distant base station (BS) (Argyriou,2018). The devices may self-organize without much manual assistance due to cooperative multi-UAV deployment, which also expands the service's service area. Furthermore, using a swarm of cellular-connected UAVs could be significantly cost-effective than using a single UAV for a task.

1.5. UAV swarm control architecture.

A UAV's sensing and perception skills, task determination and behavior in certain environmental situations are all defined by the UAV control architecture which is a global strategy and set of specialized algorithms. The processing time, the requirement to properly understand the operational environment, the ability to handle a wide range of operations, the capacity to fulfill goals in the face of uncertainty and the amount of autonomy are all impacted by the control architecture.

In terms of control architectures, several advancements have been made. Beyond the well-known control systems, deliberative strategy based on the sense-plan-act paradigm is discovered. A collection of condition-action pairings constitute the reactive architecture. The hybrid method is a synthesis of proactive and reactive capacities. The behavior method has also been described as a collection of behavior sequences each of which accomplishes a particular task (Paredes, 2018). The main goal for all these contributions is to create an

autonomous control system that is capable of making appropriate decisions, carrying out several jobs, planning a feasible trajectory, and escaping both static and moving obstacles. High performance systems have been developed using a variety of control structures. They all provide innovative thinking in an effort to create an autonomous robot. The control architectures now in use are examined in detail in the following sections. Upper edge perspective is used in the architecture of deliberative control.

The deliberative method considers goals and restrictions to ultimately carry out low-level orders to complete a specific task. Sensing, planning and acting modules constitute the majority of its three general sequential functions (Zhu, 2019). The sensing module observes the robot's surroundings to modify a predetermined world model for the objectives of each mission. The planning module creates a legitimate work plan taking into account the drone's limitations in order to accomplish the mission aim. Finally, the acting module converts the job plan into low-level commands for drones and then carries out these actions. After that, the drone performs these successive functions until it completes its objective.

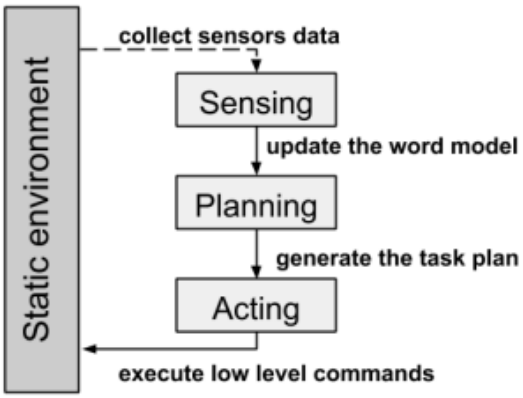


Fig. 9. Deliberative architecture

(S. Emel'yanov, D. Makarov, A. I. Panov, and K. Yakovlev, "Multilayer cognitive architecture for UAV control,", 2018.)

In some situations, this design offers a significant source of vulnerability. Here are a few of them:

• The entire design will collapse if one of the components malfunctions.

• In a dynamic or unpredictable context, it is ineffectual.

• It has a larger likelihood of failure if the representation of the world model is not correct nor full.

• It demands high performance computing capabilities: memory and processing time.

A bottom-up method named reactive control architecture was created to address several flaws in the deliberative control architecture. This architecture provides a control strategy as a group of condition-action relations that relates sensor input to robot action. It comprises of reactive rules set that reacts with environmental changes. It may function in a dynamic context without creating a world model or carrying out planning tasks by merely producing control instructions based on sensory data.

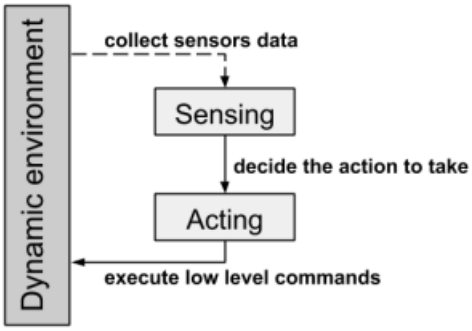


Fig. 10. Reactive architecture

(D. Nakhaeinia, S. H. Tang, S. M. Noor, and O. Motlagh, "A review of control architectures for autonomous navigation of mobile robots," 2019.)

As seen in Figure 10, the drone gathers sensor data to comprehend its environment before making a decision on what to do. This design responds more quickly to dynamic changes without any prior knowledge of the environment and is computationally quicker than the deliberative method(Yakovlev, 2019).

A combination of reactive and deliberative architecture elements is necessary to carry out a drone's task in the actual world. The hybrid technique was created to handle high-level goals and comprehensive limitations in a dynamic environment. It presents a middle ground between reactive and deliberate methods. The hybrid control architecture typically uses three hierarchical levels (see Figure 11).

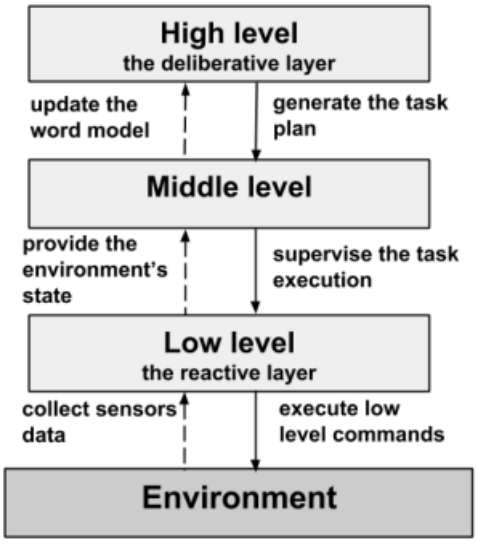


Fig. 11. Hybrid architecture

(C. Sampedro, , "A flexible and dynamic mission planning architecture for UAV swarm coordination,", 2018.)

Decision-making at a high level (the deliberative layer). This level carries out complex calculations to provide a task plan that corresponds to a list of actions. Each action specifies a particular command flow that is transmitted to the reactive layer to produce the intended action. The relationship between the high level and low level is supervised by the middle level. Low level for low control perceives the surroundings (the reactive layer). It looks after the drone's immediate safety including obstacle avoidance. Low level executes the actions specified by the deliberative layer to produce the decision-making at a high level (the deliberative layer).

The goal of the biologically inspired behavior-based control architecture is to execute a reactive mapping among perception and action modules. Basically, as illustrated in Figure 12, the behavior approach separates the control technique into a group of behaviors. Each of these are in charge of a certain duty.

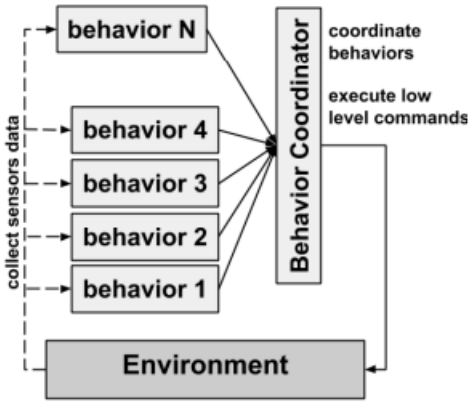


Fig. 12. The Behavior Control Architecture

(H. T. Dinh, "Sound and complete reactive uav behavior using constraint programming," 2018.)

This design has a few benefits that give this strategy an edge over the reactive one. Both reactive and deliberate skills can be provided by each conduct. Without needing to be familiar with the surroundings, this design might handle an unanticipated circumstance the drone could encounter. Additionally, it provides a set of concurrent and parallel actions that work independently to accomplish the robot's goals. Additionally, it offers a suitable answer to the issue of drones doing duties in various uncharted areas.

But this architecture has a few inconveniences:

This method requires combining and coordinating many actions in order to operate a robot. However, in some circumstances, it might be challenging to decide which previous behavior to carry out first. Since actions relate to low-level control, high-level goals may not be addressed by them. Lack of a planning module might make it difficult to complete challenging jobs.

Conclusion

This chapter has summarized recent UAV swarm application areas, classification of UAVs, UAV swarm communication and control architecture. General characteristics will be summarized in the table 5.

Table 5. General characteristics of UAV swarm communication architectures.

	GCS swarm	FANET swarm	Cellular network swarm
Advantages	No need for UAV to UAV network.	Individual decision making. Long distance propagation.	Individual decision making. Long distance propagation. Reliable communication
Disadvantages	Short propagation Not reliable.	Difficult to build trustworthy communication.	
Common	Real time operation	Real time operation	Real time operation

Cellular network swarm architecture has more advantages comparison with other communication architectures while minimizing others' disadvantages as shown in the table.

Table 6.General characteristics of UAV swarm control architectures.

	Deliberative control	Reactive control	Hybrid control	
Disadvantages	Vulnerable to dynamic environment.	Execute low level commands.	Do not operate concurrent and parallel actions.	Lack of planning module.
Advantages	Carries out complex calculations.	Function in dynamic environment.	Carries out complex calculations. Function in dynamic environment.	Set of concurrent and parallel actions.

In a static environment, the deliberate technique can achieve difficult tasks. The reactive approach stays clear of moving obstacles. The reactive and deliberative capacities are both combined in the hybrid architecture. The behavior approach specifies a number of modules that may all be used separately. The following characteristics are necessary for constructing an autonomous UAV for civil purposes as a conclusion to this work.

• Using continuous improvement to achieve complicated objectives

• Quickness of reaction to avoid moving impediments

• The capacity to adapt to a variety of missions.

• Flexibility to incorporate new features

• The capacity to expand in order to increase the control architecture's existing level of autonomy

My ongoing research will be focused on developing a control architecture for unmanned aerial vehicles (UAVs) that operate in civil areas. The suggested architecture must do challenging tasks, calculate a workable trajectory, avoid hazards and provide a suitable flight plan.

2. METODOLOGY

This chapter will examine some of the design-related tools, various path planning methods and swarm functionality. Robot operating system (ROS) for UAV swarm control, different algorithms for mapping, localization and navigation will be discussed in more detail.

2.1 Robot operating system (ROS)

The present study aims to execute the creation of a group of unmanned aerial vehicles utilizing the Robot Operating System (ROS) platform within an enclosed setting. The ROS framework is a widespread platform utilized for building of robotics software. The platform offers a diverse array of tools and libraries that streamlining the process of creating robotics applications.

ROS is an open-source computing platform that enables smooth communication and cooperation among different components of a robotic system. Under the publish-subscribe messaging model of the system, diverse components have the ability to both send and receive to messages on multiple topics. This enables inter-component communication and information exchange, even across disparate machines.

One of the primary advantages of ROS is its modularity. The system is designed to be adaptable to growth by organizing various components into packages. As a consequence, a robotic system can readily incorporate novel components and features or repurpose pre-existing components for use in alternative projects.

A variety of tools are available through ROS for viewing and troubleshooting robotic systems. In addition to tools for logging and analyzing system data, these tools also provide a graphical user interface for displaying the system's components and connections.

Numerous robotics-related fields, such as mobile robotics, industrial automation and research use ROS extensively. A significant developer and user base that supports the system works together to create new tools, libraries, and components.

The ROS Master serves as the foundation of ROS. The Master enables communication between and discovery of all further ROS software modules (Nodes). So, we never have to explicitly say, "Send this sensor data to that computer. We only need to instruct Node 1 to communicate with Node 2.

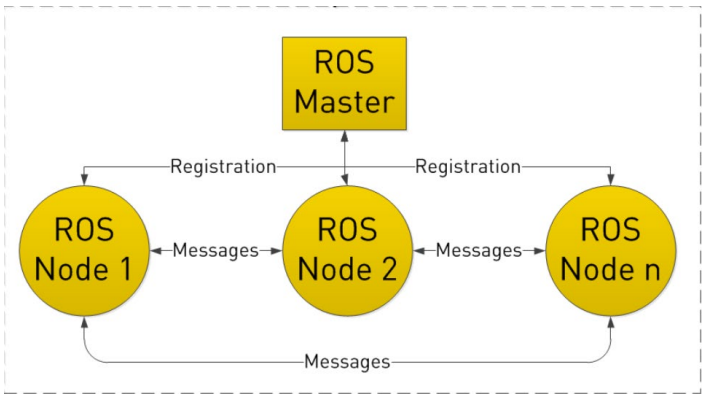


Fig. 13. ROS structure.
(https://www.clearpathrobotics.com/assets/guides/melodic/ros/Intro%20to%20the%20Robot%20Operating%20System.html)

One of the potential applications of ROS is the deployment of swarms of Unmanned Aerial Vehicles (UAVs). A group of unmanned aerial vehicles that are capable of collaborating to achieve a shared goal is commonly referred to as a UAV swarm. A group of unmanned aerial vehicles engage in inter-drone communication to share data and coordinate tasks. The software utilized by these drones to operate in a collective manner was designed on a robust platform known as ROS.

ROS has a modular architecture that enables programmers to combine smaller, reusable components to create complex systems. ROS makes it possible to create software parts for UAV swarms that can communicate with one another over a network. These elements could include communication modules, controls, and sensors. Additionally, the ROS framework offers a sizable collection of libraries that make it simple to integrate features like navigation, mapping, and localization.

The availability of a standardized communication protocol offered by ROS makes it one of the main benefits of employing it in the development of UAV swarms. This implies that the drones in the swarm may communicate the same language to one another. This facilitates the integration of additional drones into the swarm and guarantees that all the drones are cooperating efficiently. ROS offers a simulation environment which is another advantage of employing it in the building of UAV swarms. As a result, programmers can test their software components in a simulated setting before putting them into use on actual drones. The simulation environment is also helpful for testing the swarm's behavior under various scenarios, such as shifting environmental factors or various mission objectives.

Finally, ROS offers an effective platform to develop software for UAV swarms. It is the best option for designing complicated systems that need for cooperation amongst

numerous drones due to its modular architecture, common communication protocol and simulation environment. With ROS, programmers may develop software that enables drones to cooperate to complete a task, like as mapping a region, carrying out search and rescue missions or keeping track of wildlife populations.

2.2. Mapping of the environment.

The process of mapping plays a crucial role in the context of indoor drone swarming, as it enables the drones to gain a comprehensive comprehension of the surrounding environment in which they are deployed. Unmanned aerial vehicles (UAVs) have the capability to employ a variety of sensors, including but not limited to cameras, LIDAR, and GPS, for the purpose of mapping their immediate environment. Typically, the process of environment mapping entails the following series of steps:

Flight planning is necessary to make sure that the full region of interest is covered by the UAV's fly route. Using software that automatically creates flight plans based on user-defined criteria including altitude, overlap and camera settings or manually planning the flight path is an option.

Data gathering: Using its onboard sensors, the UAV gathers information about the environment while in flight. Images, LIDAR scans or GPS coordinates could all be included in the data.

Processing of data: In order to produce an environment map, the UAV's data must be processed. This could involve processing GPS coordinates to build a topographical map, using LIDAR scans to create a point cloud, or mosaics together photos to make an image.
Map creation. The data can be used to build a map of the environment once it has been processed (Liu, 2019). The map may include details about the landscape, the amount of vegetation and the locations of various structures.

Analysis of the map: The map can be examined to extract important environmental information such as the location of obstacles, changes in terrain elevation, areas of vegetation growth.

Map updating: As new information is gathered, the map can be revised over time, allowing for the tracking and analysis of environmental changes.

Overall, UAVs can map the environment using a combination of sensors and data processing techniques, providing valuable information about the environment that can be used for a wide range of applications including agriculture, environmental monitoring, and urban planning. Mapping will be used for navigation in this research.

Authors have employed a variety of mapping techniques and a brief review of several of these strategies has been provided.

Structure from Motion (SfM): A 3D model of the environment is produced using the photogrammetric approach of Structure from Motion (SfM), which employs photos collected from various perspectives. With this method, characteristics are extracted from the photos and then used to calculate the relative positions of the images. Applications for UAV mapping frequently use SfM.

Simultaneous Localization and Mapping (SLAM): Simultaneous Localization and Mapping (SLAM) is a method that enable a drone (or UAV) map a new environment while also figuring out where it is in relation to the environment. This method involves gathering data about the surroundings and the location of the drone using sensors like cameras, LIDAR and inertial measurement units (IMUs).

Occupancy Grid Mapping: A approach called occupancy grid mapping uses a grid of cells to represent the environment with each cell reflecting the probability that it contains an obstruction. Using this method, a probability map of the environment is created by merging data from several sensors.

Fast SLAM: A particle filter is used in this SLAM variation to estimate the drone's position and image the surrounding area. This method is especially helpful in settings with plenty of details such as indoor settings.

Geometric Hashing: Geometric Hashing is a technique that includes comparing characteristics of a scene to a database of previously computed characteristics (Wang, 2019). In mapping applications where surroundings are recognized in advance such as in a factory or warehouse, this technique is frequently utilized.

After studying these mapping techniques, SLAM technique has been chosen to be utilized in this research due to some reasons.

SLAM techniques are deemed highly appropriate for mapping indoor environments due to the complex layout of interior spaces, which can pose challenges for conventional mapping methods. In indoor environments, a drone is capable of utilizing a diverse range of features to determine its position and orientation with regularity. Landmarks such as walls, furniture, and other objects can be employed to aid the robot in generating a map of its environment. Precise localization is a prerequisite for the robot to navigate effectively in this demanding setting. By utilizing data from various sensors, the unmanned aerial vehicle has the capability to employ simultaneous localization and mapping (SLAM) methodologies to create a spatial representation of the surroundings, while concurrently establishing its own position within the mapped area. The sensors employed for the purpose of identifying

environmental landmarks may include cameras, LIDAR, or other range sensors. Simultaneous Localization and Mapping (SLAM) is particularly suitable for indoor environments due to its ability to effectively handle complex and densely populated surroundings that are commonly encountered in such settings. Unmanned aerial vehicles (UAVs) have the capability to perform precise indoor mapping through the utilization of simultaneous localization and mapping (SLAM) techniques, despite the presence of obstacles and other complexities that may impede traditional mapping methodologies.

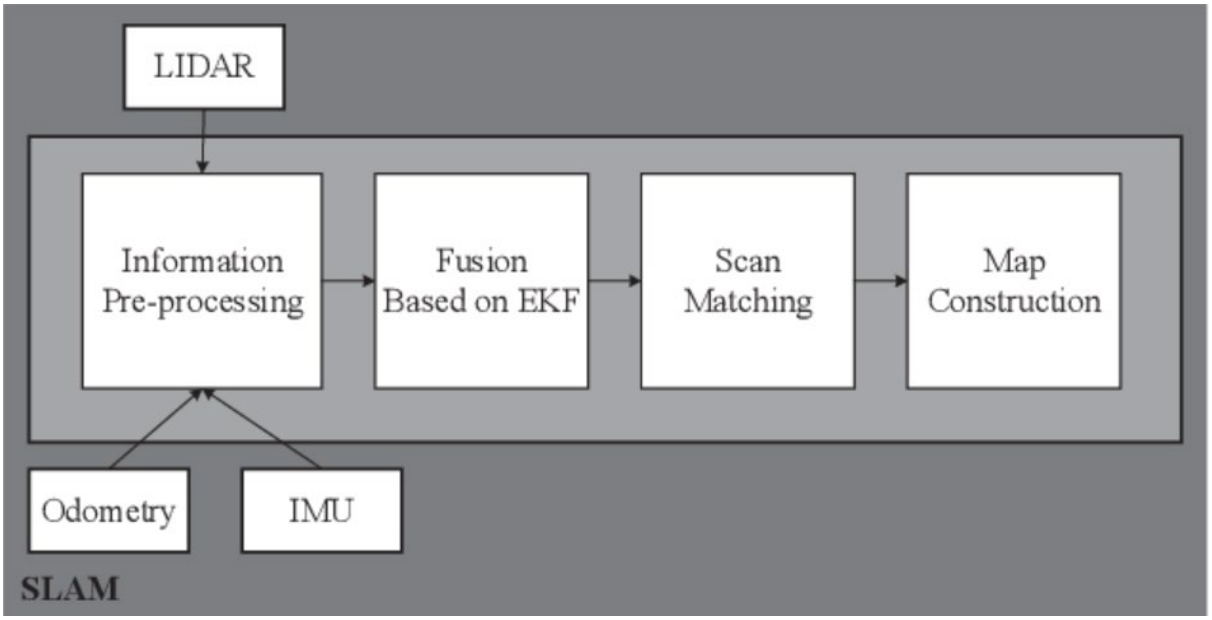


Fig.14. System overview of HECTOR SLAM. (www.semanticscholar.org/paper/An-Improved-Hector-SLAM-Algorithm-based-on-Fusion-Yu-Zhang)

Data collection: The UAV has sensors like LIDAR, IMU and odometry to gather information about its surroundings. Measurements of range and orientation are provided by the LIDAR, estimations of motion are provided by the odometry, and measurements of acceleration and orientation are provided by the IMU.

Information Pre-processing: To extract important details such as the UAV's location, orientation and motion estimates, the acquired sensor data from the UAV's sensors is initially pre-processed. The SLAM algorithm uses this data as an input.

Fusion based EKF: An Extended Kalman Filter (EKF) or another sensor fusion algorithm is then utilized to fuse the collected sensor data. The covariance matrix that illustrates the uncertainty of the predicted pose is updated by the EKF which also estimates the UAV's pose (position and orientation).

Scan Matching: After the sensor data has been fused, the current LIDAR scan is matched with a previous scan or a map. This process aids in determining the UAV's motion

and increases the pose estimate's precision.

Construction of the Map: A map of the environment is created using the aligned LIDAR scans. Depending on the particular HECTOR SLAM implementation, the map may be shown as a grid map, point cloud or occupancy grid.

Localization: After a map has been created, HECTOR SLAM offers real-time localization estimates of the UAV's pose within the map, enabling the UAV to navigate and operate in its surroundings.

The performance of the HectorSLAM method can be modified by modifying the following important parameters:

- **Map resolution:** Resolution of the map: This parameter controls how detailed the algorithm's resulting grid map will be. Higher resolution maps offer more detail, but they also need more storage and processing power.
- **Map size:** The size of the map that the algorithm creates is determined by this option. It's critical to select a map size that is appropriate for the size of the environment being mapped.
- **Laser range:** Using this option, the algorithm can specify the laser sensor's operating range. Although it uses more computing power and may be less accurate, a longer-range laser offers better coverage.
- **Max update rate:** The maximum rate at which the algorithm can update the map is indicated by the max update rate parameter. It's critical to select a value that maintains a balance between computational effectiveness and map quality.
- **Max iterations:** This option determines the most iterations that the algorithm may carry out when generating the map. Higher numbers can offer greater accuracy but also call for more processing power.
- **Min distance:** The minimum distance the robot must travel before updating the map is specified by this option. A lower number can produce more precise maps but can also cause the process to run more slowly.
- **Max distance:** This setting determines how far the robot can move before the map needs to be updated. A greater number may result in more effective mapping but less accurate maps.
- **Use odometry:** This selection controls whether the algorithm updates the map using odometry data from the robot. Odometry can increase the map's accuracy but it can also cause inaccuracies if the data is noisy.

Overall, the environment and task requirements of the mapping task influence the

parameters chosen for HectorSLAM. These settings can be carefully adjusted to improve the algorithm's performance and result in high-quality maps.

Mathematical model of HectorSLAM is consist of prediction and correction step. Based on the robot's present location and velocity, the prediction phase of the Hector SLAM algorithm estimates the robot's position and orientation. The prediction stage is provided by:

$$x_k = f(x_k - 1, u_k) + w_k \quad (1)$$

This equation represents a recursion relation where we are calculating the value of x at the kth iteration or time step(Kumar, 2018). The value of x at the kth iteration depends on the value of x at the previous iteration (k-1) and the input u at the current iteration (k), as well as some noise w_k.

xk is the value of x at the kth iteration

$xk-1$ is the value of x at the previous iteration (k-1)

uk is the input at the current iteration (k)

$f(xk-1,uk)$ is a function that takes the value of x at the previous iteration and the input at the current iteration, and returns a new value of x at the current iteration (k)

wk is some random noise or disturbance at the current iteration (k)

The Hector SLAM algorithm's corrective stage modifies the robot's orientation and location depending on data from the laser range finder. The step for rectification is provided by:

$$z_k = h(x_k) + y_k \quad (2)$$

where z_k is the measurement at time k, h is the nonlinear measurement function and y_k is the measurement noise.

The measurement equation is crucial for drone navigation since it enables us to update our estimation of the drone's state variables using sensor measurements. We may determine an error term that represents the difference between the predicted and actual measurements by comparing the expected measurements from $h(x_k)$ to the actual measurements z_k. This error term may then be used to modify our estimation of the drone's state variables and enhance the precision of our projections of the drone's future orientation and location.

2.3. Path planning

Finding the best route from the drone's current location to a destination while avoiding obstacles is known as path planning. Path planning is essential in indoor drone swarming to allow the drones to move across the space securely and effectively. There are two primary approaches.

- Global path planning
- Local path planning

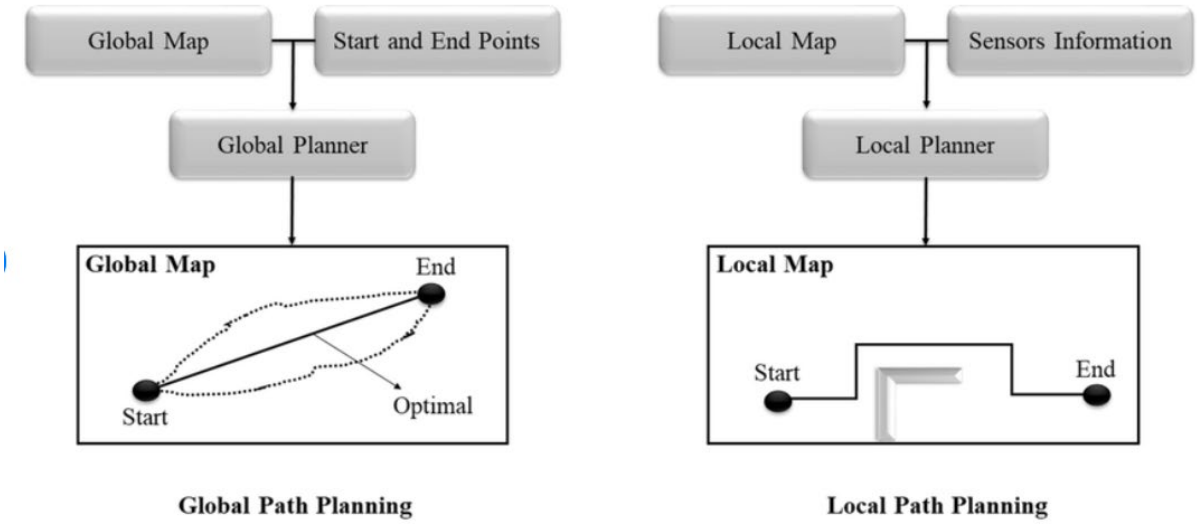


Fig. 15. Global and local path planning. (https://www.researchgate.net/figure/Global-and-local-path-planning-approaches_fig3_360268568)

The distinctions between global and local path planning are summarized as follows: Global path planning is the process of determining the best or nearly best route from the UAV swarm's initial point to the destination or goal position while taking into account the mission requirements and the surrounding environment. (Figure 15.) A high-level model of the environment such as map or grid is often created and path planning algorithms scan through this representation to find the optimum path. The UAV swarm's mission is often planned either offline or before launch.

The task of Local Path Planning involves the creation of a secure and obstacle-free trajectory for each unmanned aerial vehicle within a group, considering its present location, speed, and nearby surroundings. The diagram depicted in Figure 15. During the process of local path planning, various factors are considered such as the UAV's kinematic capacities, sensor limitations, communication restrictions, as well as immediate challenges and dynamic

obstacles. The process of local path planning is frequently executed in an online or real-time manner in the context of a mission involving a swarm of unmanned aerial vehicles (UAVs). Short comparison of some common global path planning algorithms has been reviewed with disadvantages.

Dijkstra's Algorithm: Dijkstra's algorithm is a well-known global path planning algorithm that identifies the shortest route between a starting point and a final destination by first examining the breadth of the graph or search space. It ensures optimality but because it extends nodes uniformly without taking heuristic information into account, it can be computationally expensive, especially in vast and complicated contexts.

The Breadth-First Search: (BFS) global path planning algorithm expands all of the nearby nodes before moving on to the next level of nodes. BFS is a straightforward and commonly used global path planning algorithm. BFS is comprehensive and determines the best course of action but it may not be memory and computationally efficient especially in big and complicated contexts.

Depth-First Search (DFS): Another straightforward global path planning technique is called depth-first search which expands one path as far as it can go before turning around. DFS can be memory-efficient, but it might not always take the best course of action and might become bogged down in dead ends or loops (Kiesel, 2019).

Rapidly exploring Random Trees (RRT): RRT is a probabilistic global path planning technique that grows a tree of possible starting points and goals by using random sampling. It can effectively navigate through challenging situations and is especially well suited for continuous and high-dimensional landscapes. RRT may result in poor pathways and does not ensure optimality.

Probabilistic Roadmaps (PRM): Another probabilistic global path planning approach is called Probabilistic Roadmaps (PRM) which builds a network of interconnected nodes in the environment's free space and then looks for a route from the starting point to the goal inside this network. PRM is effective in high-dimensional environments although it needs pre-processing to create the graph but, in some cases, may be affected by the computational complexity of a path planning algorithm.

A* Algorithm: A* (pronounce "A-star") is a popular global path planning method that combines the best elements of unrestrained best-first search and uniform cost search from Dijkstra's algorithm. It maintains a priority queue to choose the most promising node for expansion and utilizes a heuristic function to calculate the cost of getting there from where it is now. A* is effective and frequently used because it can locate ideal or almost ideal paths in a variety of situations, particularly in discrete or grid-based contexts.

A* start algorithm has been chosen to use in this research for global path planning due to its advantages mentioned above.

A common graph PPA is the A* algorithm. The A* is an accurate iterative heuristic search algorithm, or a popular kind of iterative best-first search. The algorithm executes using the static path tree with the lowest cost between the beginning and target points. In this regard, it functions similarly to Dijkstra's algorithm, which it modifies. The A* algorithm's goal is to determine the shortest path and it uses a heuristic function to focus its search on states that are along that path. As a result, the A* algorithm is more effective than Dijkstra. In some situations, this method is applied in dynamic contexts. The A* algorithm chooses the least expensive route and assesses its cost:

$$\boldsymbol{f(n) = g(n) + h(n)} \qquad (3)$$

Where $\boldsymbol{n}$ shows location of UAV, $\boldsymbol{f(n)}$ - is the cost of path from start point to final point, $\boldsymbol{g(n)}$ – is the actual cost from node $\boldsymbol{n}$ to the first node and $\boldsymbol{h(n)}$ – is the heuristic function which calculates cost of optimal path from node $\boldsymbol{n}$ to the target node. The heuristic is the value of the A* algorithm's minimum cost evaluation from any node to the target node. Additionally, this feature contributes in minimizing the quantity of passing nodes. Therefore, the choice of the heuristic function directly affects the algorithm's effectiveness. Heuristic functions for the method include Euclidean distance, Manhattan distance, Chebyshev distance and diagonal distances. (4-7)

Manhattan distance heuristic function:

$$\boldsymbol{h(n) = |x_p - x_q| + |y_p - y_q|} \qquad (4)$$

Eucklidean distance heuristic function:

$$\boldsymbol{h(n) = \sqrt{|x_p - x_q|^2 + |y_p - y_q|^2}} \qquad (5)$$

Chebyshev distance heuristic function:

$$\boldsymbol{h(n) = max(|x_p - x_q|, |y_p - y_q|)} \qquad (6)$$

Dioganal distance heuristic function:

$$\boldsymbol{h(n) = |x_p - x_q| + |y_p - y_q| + (\sqrt{2} - 2)min(|x_p - x_q|, |y_p - y_q|)} \qquad (7)$$

It is a significantly more straightforward and computationally less intensive method than many other PPAs, given its efficiency appropriate for working in embedded systems. It generates the shortest roads by determining the best course using heuristic data. The complexity of the map however significantly raises computing time and memory

requirements.

Local path planning:

A widely used approach for local path planning in mobile robots, including drones is the Dynamic Window Approach (DWA). To find the optimal route to the target, the algorithm creates a viable velocity, space or "dynamic window," and assesses potential paths within of it. The optimal route is the one that minimizes a cost function while meeting many requirements, including acceleration, velocity, and obstacle avoidance.

Mathematical model:

Calculate the dynamic window:

The collection of potential robot speeds and directions that the robot can go to in the next step is known as the dynamic window. The robot's current speed, maximum speed, maximum acceleration and maximum angular velocity all contribute to defining the dynamic window. The following equation is used to determine the dynamic window:

$$Vdynamic=\{v|vmin\leq v\leq vmax\} \quad (8)$$

$$\omega dynamic=\{\omega|\omega min\leq\omega\leq\omega m\} \quad (9)$$

where *vmin* and *vmax* are the minimum and maximum linear velocities of the robot, ω_{min} and ω_{max} are the minimum and maximum angular velocities of the robot.

Evaluate possible trajectories:

The algorithm sets up a trajectory by simulating the robot's motion for a brief time period for each velocity and direction in the dynamic window. The trajectory will be generated based on its closeness to obstacles, its distance from the objective and if it violates any limitations, the trajectory will be evaluated. The following equation is used to compute each trajectory's cost:

$$C=\alpha\, C_{obs}+\beta\, C_{goal}+\gamma\, C_{smooth} \quad (10)$$

where C_{obs} is the cost of the trajectory based on its proximity to obstacles, C_{goal} is the cost of the trajectory based on its distance to the goal, C_{smooth} is the cost of the trajectory based on its smoothness and α, β, and γ are the weights assigned to each cost term.

Select the best trajectory:

The selection of the optimal trajectory is based on the algorithm's determination of its path with the minimum cost. The velocity and moving of the robot are configured to match those of the optimal trajectory. Subsequently, the robot proceeds along the

designated path for the subsequent movement.

2.4. Collision Avoidance

The UAV should take into account both the position and the velocity of other UAVs in order to ensure free-collision behavior in a multi-agent system. Social proximity approach has been implemented for this reason in order to take advantage of other UAVs' positions to change the costmap. LIDAR (Light Detection and Ranging) sensors can be used in social proximity techniques to determine the position and speed of other drones around. LIDAR operates by emitting laser pulses and calculate how long it takes for the pulses to return from surrounding objects, such as other drones. The position and speed of the drones can be determined by examining the patterns of laser reflections.

A general description of the operation of LIDAR-based social proximity approaches is given below:

Sensor setup: The drone is equipped with a LIDAR sensor or sensors which are orientated and positioned to offer a 360-degree picture of the surroundings.

Detection and tracking: The LIDAR sensors continuously generate laser pulses which they then pick up as reflections from other drones nearby, allowing them to detect and track nearby objects. Through the use of methods like clustering, filtering and data association, the reflections are processed in order to find and locate the other drones (Felner, 2018). The coordinates and velocities of the discovered drones are estimated, and they are given distinctive IDs.

Social proximity calculation: Calculation of social closeness: The other drones' positions and speeds are taken into account when determining the drone's social vicinity. This can be achieved by modeling the drones as point masses and computing the forces between the drones using a potential field or a social force model. The drone's behavior can be controlled by the resulting social proximity measure which can be used to advise it to keep a safe distance or avoid crashes.

Control and decision-making: the drone's control and decision-making algorithms modify the trajectory and speed of the drone based on the estimated social proximity measure. The drone might slow down or change course to avoid a collision, for instance, if the social proximity measure indicates a high danger of collision with another drone.

In general, LIDAR-based social proximity algorithms can implement efficient collision avoidance tactics by providing real-time information on the positions and velocities of other drones nearby. They might be constrained, though, by the LIDAR sensors' precision

and range as well as the environment's complexity and other drones' activity.

The assessment of other drones' position and velocity can be made more precise and robust by combining the 2D Gaussian kernel with LIDAR in social proximity approaches. The probability density function (PDF) of the position and speed of the other drones, based on the LIDAR observations can be modelled specifically using the 2D Gaussian kernel. In basically, the costmap incorporates a 2D Gaussian kernel, where the mean represents the position of the neighbor and the covariance represents the velocity of the neighbor. The social kernel also known as the 2D Gaussian kernel is defined as:

$$\boldsymbol{S(x, y, v_x, v_y) = A \exp\left(-\frac{1}{2}\left(\frac{(i-x)^2}{(\sigma f v_x)^2} + \frac{(j-y)^2}{(\sigma f v_y)^2}\right)\right)} \qquad (11)$$

Some variables $(x, y), (v_x, v_y)$ and (i, j) are UAV's position, velocity and costmap grid coordinate respectively. A is the amplitude of the kernel. This means how strong other UAVs position affect the costmap. σ and f are covariance and weighting factor for the velocity. Moreover, there is another parameter called cut-off which is used as the smallest value to publish on costmap adjustments.

Conclusion

In this chapter, different mapping and path planning algorithms has been analyzed. HECTOR SLAM algorithm chosen for mapping technique for indoor environment. A* algorithm chosen for global path planning and Dynamic Window Approach algorithms chosen for global and local path planning. Social proximity technique has been explained to calculate neighbours' position and velocity

3. PRACTICAL PART

Mapping, navigation and collision avoidance techniques will be implemented in this chapter by using ROS framework and GAZEBO simulation tool in unknown environment. Different formation of UAV swarms will be implemented in order to compare experimental results and the most effective topology will be chosen according to results.

3.1. Overall system description

The present study involved the development and implementation of a collision avoidance mechanism intended for the autonomous navigation of multiple unmanned aerial vehicles (UAVs) within an unfamiliar setting. The utilization of the ROS framework and Gazebo as the simulation environment was employed for the implementation. The hector_quadrotor ROS package serves as a quadrotor model. The Hector SLAM and Move Base ROS packages are utilized for the purposes of localization and mapping, as well as navigation system, correspondingly.

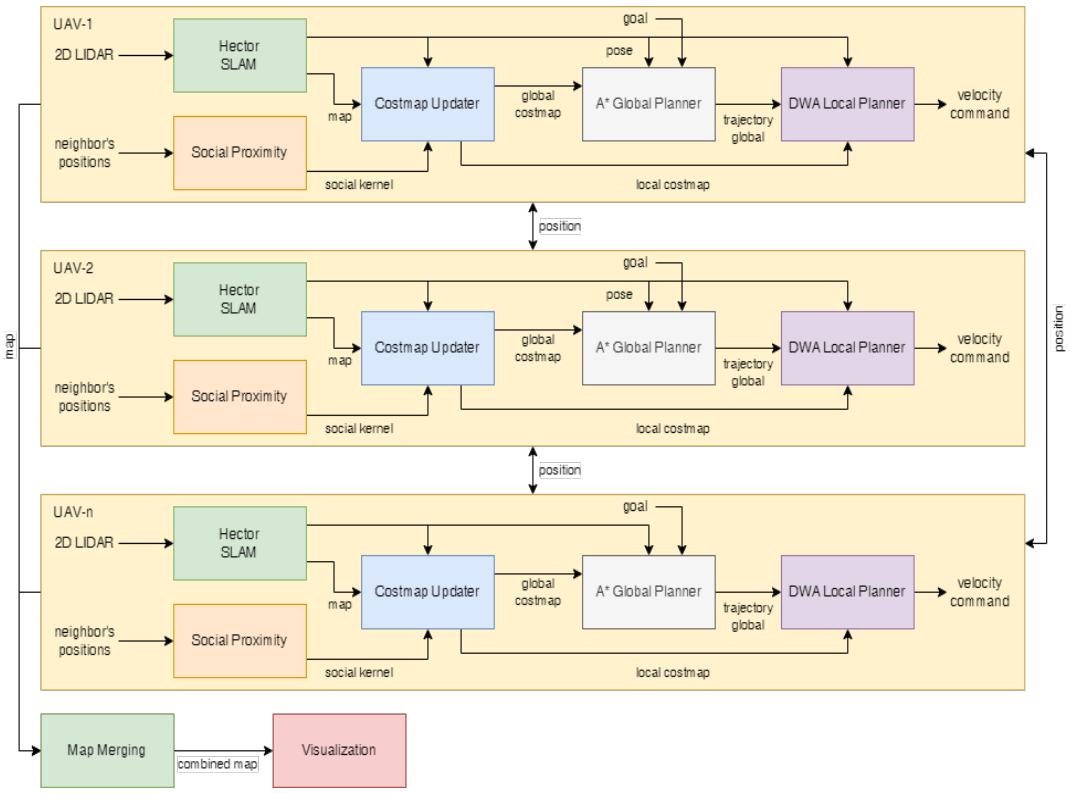


Fig. 16. System diagram for multi-UAV collision avoidance

The Hector Simultaneous Localization and Mapping (SLAM) algorithm produces a map and pose estimation, comprising the position and orientation, that is subsequently utilized by the navigation system. The navigation system comprises three

components, namely the costmap, global planner, and local planner. The costmap is a cartographic representation of the likelihood of traversability of a given area, derived from the underlying raw map data. The construction of a trajectory from the starting point to the destination is facilitated by the implementation of the A* global planner. Subsequently, the trajectory is employed by the DWA local planner to produce a velocity that can pursue the intended trajectory.

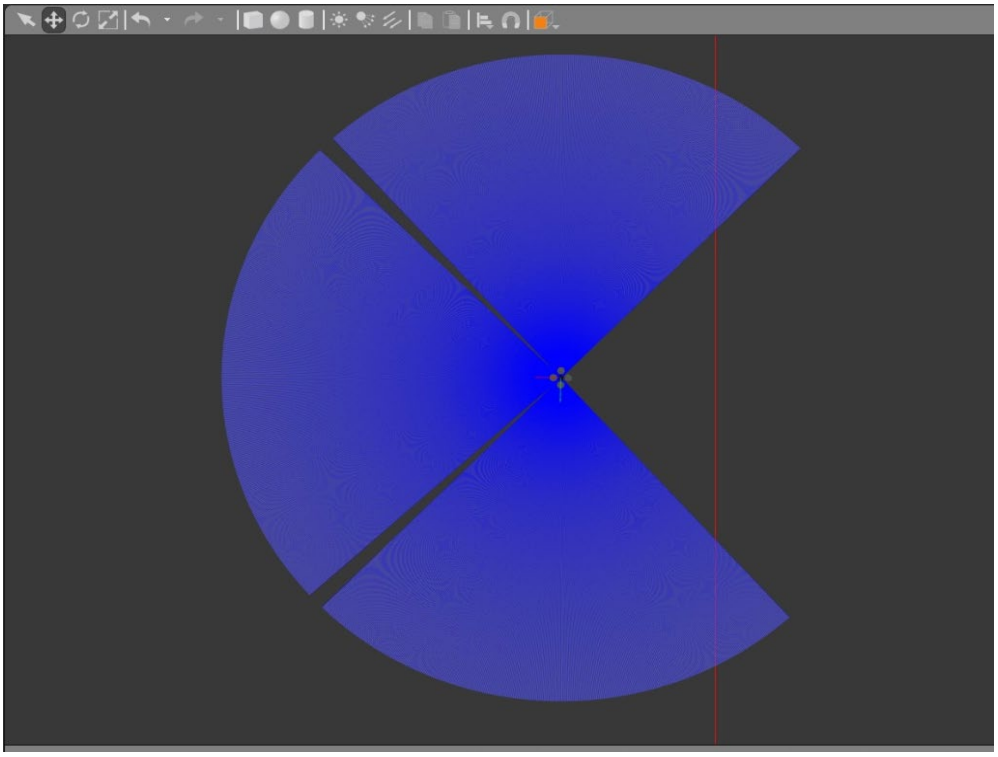


Fig. 17. Field of view

Each UAV is installed with 2D LIDAR with 270 degree Field of View (FoV) as shown in the figure 17.

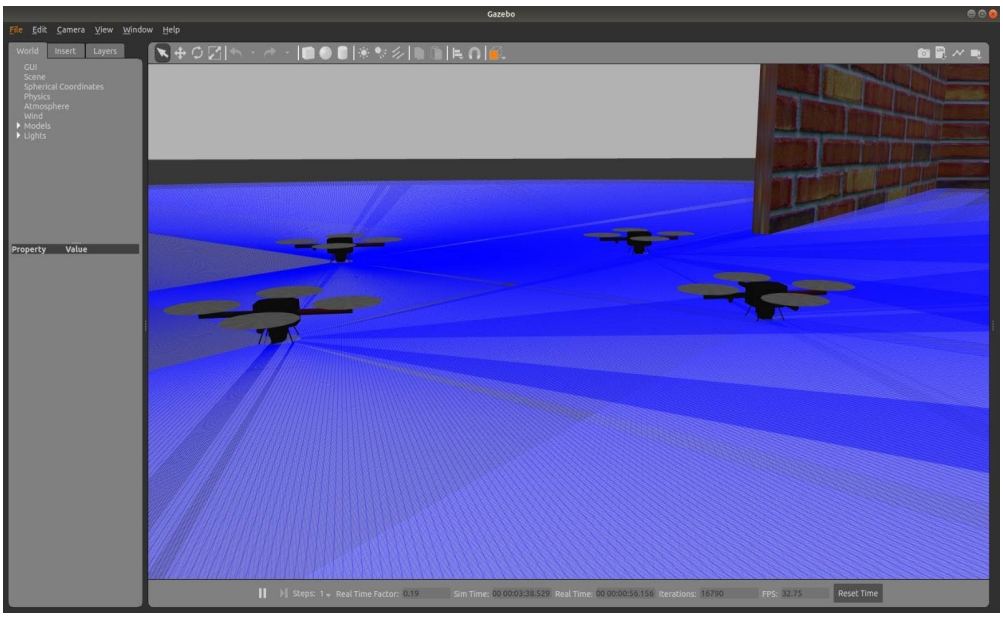


Fig. 18. Hector quadrotors

The coordination of this set of unmanned aerial vehicles (UAVs) is achieved through a decentralized topology, wherein each individual UAV executes algorithms by means of inter-communication with its counterparts. To account for uncertain environmental conditions, it is recommended that each unmanned aerial vehicle (UAV) acquire information regarding its surrounding area through the use of LIDAR technology. In addition, the unmanned aerial vehicles (UAVs) are required to perform location estimation. Simultaneous Localization and Mapping (SLAM) is a technique that can be employed to address this matter. Authors have utilized various SLAM algorithms. Hector SLAM is considered a promising Simultaneous Localization and Mapping (SLAM) algorithm, primarily due to its ability to accommodate 6 degrees of freedom (6DOF) robots, which possess both 3D position and 3D orientation. The Hector Simultaneous Localization and Mapping (SLAM) algorithm produces a map and pose estimation, encompassing both position and orientation, that is subsequently utilized for the navigation system.

Fig. 19. Important ROS launches and scripts.

- Green files: launch files related to Gazebo environment.
- Red files: Main launch files for opening Gazebo and all control system.
- Blue files: Python scripts related to collision avoidance control with and without formation and the social proximity technique for estimation of velocity and position.
- Purple files: Python scripts related to logger. Orange: Launch files related to mapping and navigation.
- Yellow files: Files related to navigation and costmap parameters.

In the implementation, a main ROS launch named **indoor_slam_gazebo_4.launch** is created for:

- Opening the Gazebo with indoor environment (indoor_environment.launch)
- Spawning quadrotors (hector_quadrotor_model package)
- Creating a transformation or coordinate as the map reference for each UAV and
- Opening Rviz for visualization.

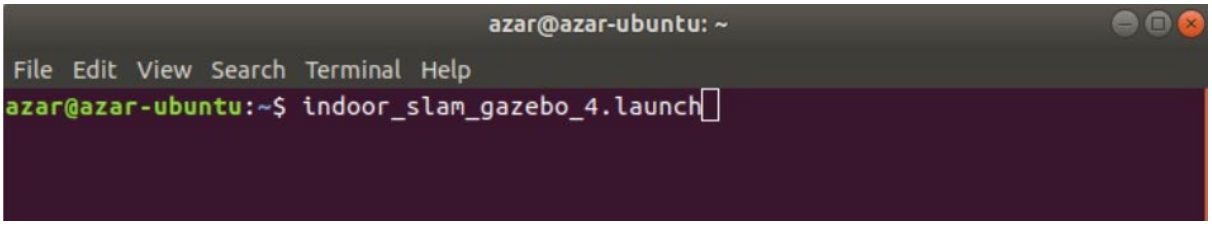


Fig. 20. Launch command

The **indoor_slam_gazebo_4.launch** code snippet is shown in Figure 3 below. Figure 3a shows how the initial position of UAVs are initialized and the indoor_environment.launch is included to open the Gazebo. Figure 3b shows how the quadrotors are spawned using groups with different namespaces. Figure 3c shows how some new frames were created using tf_static_transform_publisher for each UAVs and also opening Rviz function at the end. After launching, the walls and quadrotors installed with 2D LIDAR are spawned on a Gazebo Simulator as it is shown in table 7.

Table 7. Initial and goal positions of drones.

Position	**UAV 1**	**UAV 2**	**UAV 3**	**UAV 4**
X	2	0	2	0
Y	-15	-15	-17	-17
Z	0.3	0.3	0.3	0.3
Position	**UAV 1**	**UAV 2**	**UAV 3**	**UAV 4**
X	42	42	40	40
Y	-2	2	-1	1
Z	0.3	0.3	0.3	0.3

```
indoor_slam_gazebo_4.launch
<?xml version="1.0"?>

<launch>

  <arg name="uav1_x_pos" default="2" />
  <arg name="uav1_y_pos" default="-15" />
  <arg name="uav1_z_pos" default="0.3" />
  <arg name="uav1_yaw" default="1.5707" />

  <arg name="uav2_x_pos" default="0" />
  <arg name="uav2_y_pos" default="-15" />
  <arg name="uav2_z_pos" default="0.3" />
  <arg name="uav2_yaw" default="1.5707" />

  <arg name="uav3_x_pos" default="2" />
  <arg name="uav3_y_pos" default="-17" />
  <arg name="uav3_z_pos" default="0.3" />
  <arg name="uav3_yaw" default="1.5707" />

  <arg name="uav4_x_pos" default="0" />
  <arg name="uav4_y_pos" default="-17" />
  <arg name="uav4_z_pos" default="0.3" />
  <arg name="uav4_yaw" default="1.5707" />

  <arg name="use_uav_name1" default="uav1"/>
  <arg name="use_uav_name2" default="uav2"/>
  <arg name="use_uav_name3" default="uav3"/>
  <arg name="use_uav_name4" default="uav4"/>

  <!-- Start Gazebo with wg world running in (max) realtime -->
  <include file="$(find hector_gazebo_worlds)/launch/indoor_environment.launch"/>
```

Fig. 21. Initial positions of drones.

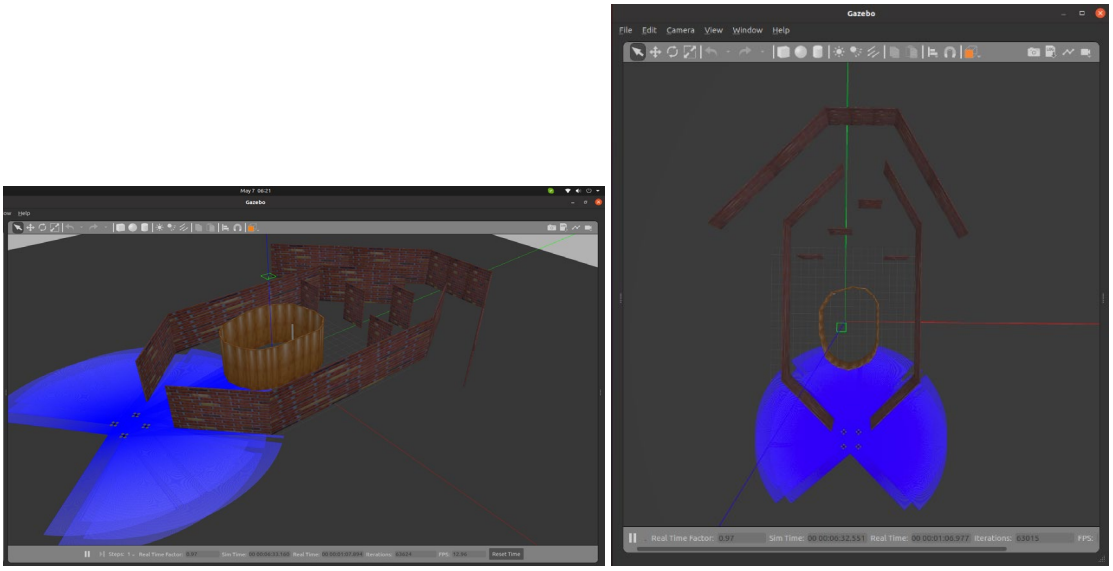
```
<!-- Create new tf as the reference for each UAV -->
<node pkg="tf" type="static_transform_publisher" name="world_to_uav1_broadcaster" args="-1 -1 0 -1.57 0 0 map uav1/map 100"/>
<node pkg="tf" type="static_transform_publisher" name="world_to_uav2_broadcaster" args="1 -1 0 -1.57 0 0 map uav2/map 100"/>
<node pkg="tf" type="static_transform_publisher" name="world_to_uav3_broadcaster" args="-1 1 0 -1.57 0 0 map uav3/map 100"/>
<node pkg="tf" type="static_transform_publisher" name="world_to_uav4_broadcaster" args="1 1 0 -1.57 0 0 map uav4/map 100"/>

<!-- Start rviz visualization with preset config -->
<node pkg="rviz" type="rviz" name="rviz" args="-d $(find hector_quadrotor_demo)/rviz_cfg/indoor_slam_multi_four.rviz"/>
```

Fig. 22. Lunch Rviz function

This function is used to visualize how drones map the environment. In the beginning

of the flight the environment is unknown but the drones map the environment by LIDAR sensors.



(a) (b)

Fig. 23. Four quadrotor installed with 2D LIDAR on a Gazebo Simulator

The move_base ROS package is utilized to implement the navigation system. The navigation system produces a costmap that contains a grid of unknown dimensions, with each grid cell assigned a probability value indicating whether it is occupied or free. The costmap is produced by taking into account two unchanging variables, namely the robot's footprint and inflation radius. The inflation radius serves the function of maintaining a safety buffer between unmanned aerial vehicles and potential obstructions. Nonetheless, the aforementioned approach solely accounts for stationary impediments, such as walls or other immobile entities, as the 2D LIDAR may not always detect other UAVs. The term "footprint" refers to the physical dimensions of a drone.

In order to prevent collisions with other unmanned aerial vehicles (UAVs), it is necessary for each UAV to possess information regarding the location of its neighboring UAVs. The term used to refer to this concept is social proximity. The incorporation of this social factor leads to in the generation of an updated costmap. The configuration of the importance for the dynamic obstacle can be achieved by adjusting the parameters associated with the social proximity algorithm, including amplitude and covariance. Furthermore, utilizing the preceding position to compute the approximated velocity would also enhance safety.

Once the costmap has been prepared to include both static and dynamic obstacles, the global and local costmaps are distributed to the navigation system. The global costmap and

local costmap are two distinct costmaps utilized in unmanned aerial vehicle (UAV) navigation. The former pertains to the costmap that encompasses the entirety of the map, while the latter refers to the costmap that is confined within a specific radius surrounding the UAV. The global planner utilizes the global costmap. The A* algorithm was employed in the present study to produce the global trajectory. Subsequently, the local planner utilizes the global trajectory. The Dynamic Window Approach (DWA) is employed as the local planner. The unmanned aerial vehicle (UAV) adheres to the trajectory that is locally generated by the dynamic window approach (DWA) through the issuance of commands for linear and angular velocity. The map merging algorithm is utilized for visualization purposes, with the assumption of the initial relative position of each UAV.

```
10    <group ns="$(arg use_uav_name1)">
11      <include file="$(find quadrotor_navigation)/launch/mapping.launch">
12        <arg name="base_frame" value="base_footprint"/>
13        <arg name="odom_frame" value="world"/>
14        <arg name="tf_prefix" value="$(arg use_uav_name1)"/>
15      </include>
16
17      <node pkg="hector_quadrotor_demo" type="control_$(arg strategy).py" name="high_level_control" output="screen">
18        <param name="target_altitude" value="2.0"  />
19        <param name="goal_x" value="40.0"  />
20        <param name="goal_y" value="-2.0"  />
21        <param name="goal_z" value="0.0"  />
22        <param name="goal_yaw" value="0.0"  />
23      </node>
24
25      <node pkg="hector_quadrotor_demo" type="social_proximity.py" name="social_proximity" output="screen">
26        <param name="n_agent" value="4"  />
27      </node>
28    </group>
```

(a)

```
103     <node pkg="multirobot_map_merge" type="map_merge" respawn="false" name="map_merge">
104       <param name="robot_map_topic" value="map"/>
105       <param name="robot_namespace" value="uav"/>
106       <param name="merged_map_topic" value="map"/>
107       <param name="world_frame" value="map"/>
108       <param name="known_init_poses" value="true"/>
109       <param name="merging_rate" value="4.0"/>
110       <param name="discovery_rate" value="0.5"/>
111       <param name="estimation_rate" value="0.5"/>
112       <param name="estimation_confidence" value="2.0"/>
113     </node>
```

(b)

```
115     <node pkg="logger" type="logger.py" name="logger" output="screen"/>
```

(c)

Fig. 24. mapping_and_navigation_multi_uav_4.launch

The code snippet is shown in Figure 24. Figure 24 (a) shows a group of the 1st UAV which has mapping, high level control for formation or individual control and the social proximity. Figure 24b shows how some move_base packages are included for each UAV. There are four important parameters related to map merging (figure 24b). Parameter merging_rate is the rate at which the maps are merged. This parameter specifies how often the node will merge the maps. In this case, we set the frequency to 4 Hz which means the map merging algorithm will be executed with 0.25 seconds period. The discovery_rate is

the rate at which the node looks for new robots. This parameter specifies how often the node will check for new robots that have joined to the network. The estimation_rate is the rate at which the node estimates the pose of each robot. This parameter specifies how often the node will estimate the pose of each robot. We set the discovery_rate and estimation_rate to 0.5 Hz. The last parameter is estimation_confidence which is the confidence threshold for the pose estimation. This parameter specifies the minimum confidence required for the node to accept a pose estimate. A higher value will result in more accurate pose estimates but a higher estimation confidence will require more computation time. Figure 24(c) indicates logger for simulation results.

Besides implementing the ROS architecture for quadrotors collision avoidance, another contribution is to investigate different techniques for a swarm of quadrotors using some leader-follower formation-based control strategies. As it is depicted in Figure 16, the global planner needs a goal for each UAV. For individual behavior (no formation), each UAV only needs one static final goal. However, for leader-follower strategy, only the leader which has a static goal. The followers have dynamic goals according to the desired formation.

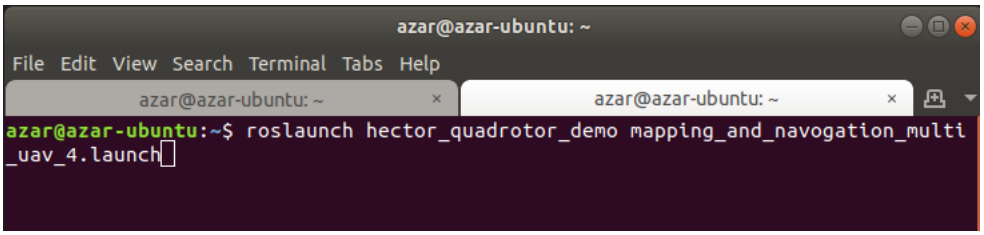


Fig. 25. Launch command for flight test.

All of the functions related to SLAM, navigation and other controls are implemented in a launch file as shown on figure 25.

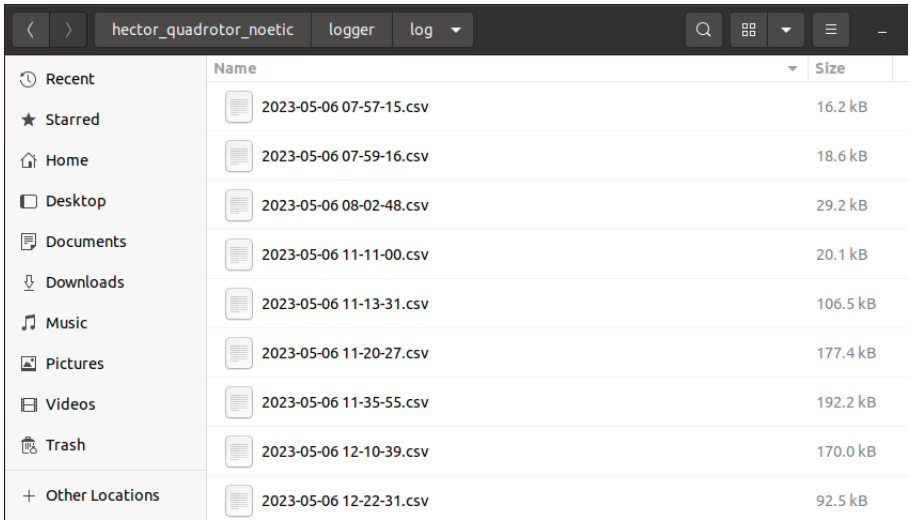


Fig. 26. Logger files for extracting data.

The logger script is used for storing the data such as position, orientation, linear and angular velocity, velocity command and nearest obstacle for each UAV. The data for a mission is stored in a csv file under the log folder in logger package (see Figure 6).

3.2 Mapping

Hector SLAM - Hector SLAM is a popular SLAM algorithm that is often used for UAV equipped with a 2D LIDAR sensor. It is designed to create a map of an unknown environment and localize the UAV within that map in real-time. One of the main advantages of using Hector SLAM with a 2D LIDAR is its ability to generate a highly accurate map of the environment even in the presence of noise and disturbances in the sensor data. It achieves this by using a scan matching technique that aligns successive LIDAR scans to create a continuous map.

Another advantage of Hector SLAM is its ability to operate in real-time which is essential for UAV applications where fast and accurate mapping and localization are critical. To set up Hector SLAM for mapping and localization, several parameters need to be configured in **mapping.launch**. These parameters include: map resolution, distance and angle threshold for map update and free and occupied update factors. The code snippet of mapping.launch related to the parameters is shown in Figure 27.

```
<node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen">

  <param name="map_resolution" value="0.050"/>
  <param name="update_factor_free" value="0.45"/>
  <param name="update_factor_occupied" value="0.8" />
  <param name="map_update_distance_thresh" value="0.2"/>
  <param name="map_update_angle_thresh" value="0.06" />
```

Fig. 27. Hector SLAM parameter configuration

Map resolution determines the granularity of the map and should be set based on the size of the environment and the level of detail required in the map. In Hector SLAM, the map is updated based on the distance and angle between successive LIDAR scans. Especially the map is updated only if the distance between the current pose estimate and the previous pose estimate exceeds a certain distance threshold or if the angle between the current pose estimate and the previous pose estimate exceeds a certain angle threshold. In addition to the distance and angle thresholds, Hector SLAM also uses free and occupied update factors to update the map. The free update factor determines how much weight is given to free space in the map, while the occupied update factor determines how much weight is given to occupied space in the map.

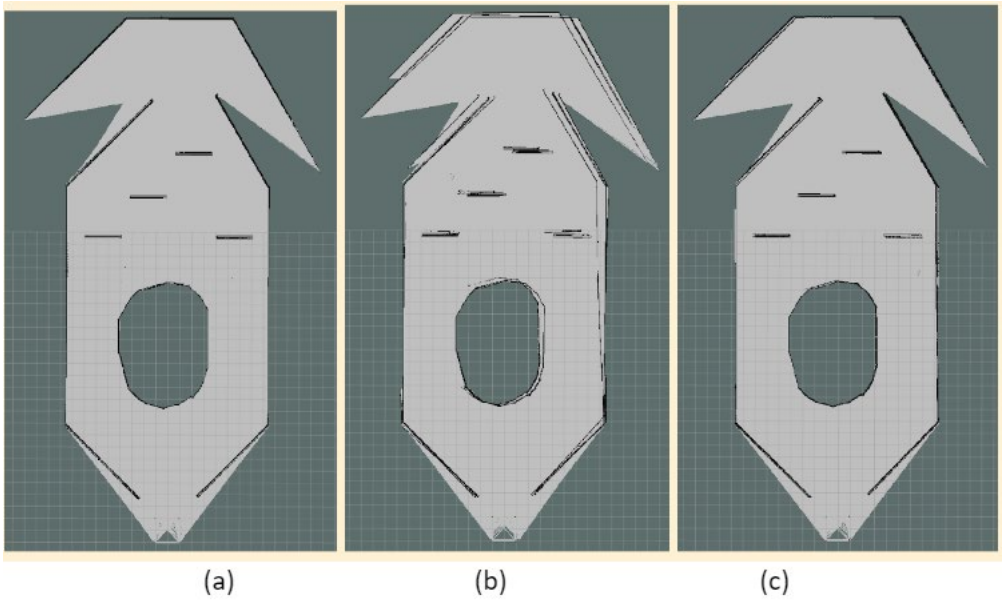


Fig. 28. Different map generated by Hector SLAM with different free and occupied update factor parameters. (a) 0.45 and 0.8 (b) 0.25 and 0.8 (c) 0.45 and 0.9

The optimal values for the update factors of free and occupied states are 0.45 and 0.8, respectively, as depicted in Figure 28. The stability of the map depicted in Figure 28a surpasses that of Figures 28b and 28c. Reducing the free update factor, as illustrated in Figure 28b, results in a decrease in the speed at which the Hector SLAM algorithm updates the cells of the occupancy grid that are categorized as free space. Consequently, the algorithmically generated map may exhibit a higher degree of caution and incorporate imprecise obstructions, even in regions devoid of tangible barriers. The update factor for occupancy is a crucial parameter that governs the pace at which the algorithm labels cells as being occupied. In cases where the occupied update factor is excessively high, as depicted in Figure 28c, Hector SLAM may incorrectly designate cells as occupied, resulting in false positives. This phenomenon may lead to the production of a map that portrays a greater number of obstacles than what exists in reality.

Costmap - A costmap is a cartographic representation that endows each cell in the map with a numerical value or cost, which is determined by the cell's occupancy state and various other factors, such as its proximity to obstacles, terrain characteristics, or other environmental limitations. The primary objective of a costmap is to furnish a depiction of the surroundings, which can be employed by path planning algorithms to produce paths that are free from collisions for an unmanned aerial vehicle or a robot. Nonetheless, the present study solely takes into account the measurement of the distance to stationary obstructions and the

social proximity as the measurement of the distance to mobile obstructions.

The costmap utilizes the map produced by the SLAM algorithm in order to construct a depiction of the surroundings that takes into account the physical limitations and sensor constraints of the unmanned aerial vehicle. The costmap is generated through the superimposition of a grid of cells onto the original map, with each cell being assigned a cost that is determined by its occupancy status and other variables such as proximity to obstructions or other environmental limitations.

The costmap is implemented in an available ROS package named: move_base. The move_base is called in a launch file named: quadrotor_move_base.launch under quadrotor_navigation package. The code snippet of quadrotor_move_base.launch is shown in Figure 29.

```
<launch>

  <arg name="odom_frame_id"   default="map"/>
  <arg name="base_frame_id"   default="base_footprint"/>
  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="/ground_truth/state" />
  <arg name="laser_topic" default="/scan" />
  <arg name="custom_param_file" default="$(find quadrotor_navigation)/param/dummy.yaml"/>
  <arg name="tf_prefix" default=""/>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen" ns="$(arg tf_prefix)">
    <rosparam file="$(find quadrotor_navigation)/param/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find quadrotor_navigation)/param/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find quadrotor_navigation)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find quadrotor_navigation)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find quadrotor_navigation)/param/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find quadrotor_navigation)/param/move_base_params.yaml" command="load" />
    <rosparam file="$(find quadrotor_navigation)/param/global_planner_params.yaml" command="load" />
    <rosparam file="$(find quadrotor_navigation)/param/navfn_global_planner_params.yaml" command="load" />
    <!-- external params file that could be loaded into the move_base namespace -->
    <rosparam file="$(arg custom_param_file)" command="load" />

    <!-- reset frame id parameters using user input data -->
    <param name="global_costmap/global_frame" value="$(arg tf_prefix)/$(arg global_frame_id)"/>
    <param name="global_costmap/robot_base_frame" value="$(arg tf_prefix)/$(arg base_frame_id)"/>
    <param name="local_costmap/global_frame" value="$(arg tf_prefix)/$(arg odom_frame_id)"/>
    <param name="local_costmap/robot_base_frame" value="$(arg tf_prefix)/$(arg base_frame_id)"/>
    <param name="DWAPlannerROS/global_frame_id" value="$(arg tf_prefix)/$(arg odom_frame_id)"/>

    <remap from="cmd_vel" to="cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>

  </node>

</launch>
```

Fig. 29. Code snippet of quadrotor_move_base.launch.

The parameters of significant importance can be utilized to configure the cost attributed to individual cells within the costmap. The inflation radius stands out as the foremost significant parameter. The inflation radius parameter is utilized to specify the extent to which the costmap is expanded in the vicinity of obstacles. The inflation radius denotes the magnitude of the spatial buffer encompassing obstructions within the costmap, serving as a protective boundary. The purpose of this mechanism is to guarantee that the unmanned aerial vehicle (UAV) sustains a secure separation from impediments, and it can be established in accordance with the dimensions and velocity of the UAV. The occupancy threshold represents the second parameter. The occupancy threshold is a crucial parameter that is utilized to classify a cell as either occupied or free in the costmap. The establishment of parameters for path planning can be contingent upon the precision of the LIDAR sensor and the degree of complexity deemed necessary.

The scaling factors of the third parameter play a crucial role in determining the relative importance assigned to various factors in the costmap, such as environmental

constraints and distance to obstacles. It is possible to configure them in a manner that equalizes the impact of said factors on the process of path planning. It is possible to establish a parameter that determines the frequency at which the costmap is updated in response to new sensor data. The parameter can be configured in accordance with the velocity of the unmanned aerial vehicle.

The yaml configuration file specifies the parameters for a UAV's obstacle avoidance system, which is a critical component of autonomous robotic navigation. The file defines four layers: obstacle_layer, inflation_layer, static_layer, and social_layer. Moreover, the radius of the UAV's should also be defined in robot_radius parameter. In this case, we define the radius is 0.6 meters.

The obstacle_layer assumes the responsibility of detecting obstacles and attributing costs to them. The observation source utilized by the system is a laser scanner, denoted as "laser_scan" in the configuration file. The obstacle height parameters have been established with a minimum value of -5.0 and a maximum value of 10.0. Any obstacles that fall outside of this designated range will be disregarded. The established lethal cost threshold is 100. Consequently, any obstacle that surpasses this value will be deemed as lethal, prompting the robot to take measures to evade it. The parameters obstacle_range and raytrace_range have been assigned a value of 9.0, indicating that objects and beams will be taken into account within a distance of 9.0 meters. The combination method has been assigned a value of 1, indicating that the expenses incurred due to the presence of overlapping obstacles will be amalgamated by employing the highest value. The boolean variable track_unknown_space has been assigned a value of true, indicating that the robot's mapping algorithm will dynamically adjust the cost of unexplored areas as it navigates through the environment. The variable known as "unknown_cost_value" is assigned a value of negative one, indicating that any space with an unknown cost is also assigned a value of negative one. The boolean value of publish_voxel_map has been assigned as false, indicating that the publication of the voxel map has been disabled. The inflation layer is tasked with the responsibility of expanding the obstacles in order to generate a safety buffer zone around them. The value assigned to the cost_scaling_factor parameter is 5.0, thereby indicating that the cost associated with the expanded region is multiplied by a factor of 5.0. The inflation radius has been designated as 1.0, indicating that the obstacles will undergo a 1.0 meter inflation.

The static_layer is used to assign costs to areas of the map that are known to be free of obstacles. It is enabled in the configuration file, which means that it will likely use a pre-built map or a map generated by SLAM to assign costs to free space.

The social_layer is used to take into account the presence of other robots or agents in

the environment. It uses a Gaussian distribution to model the influence of other agents. The amplitude is set to 150.0, which determines the strength of the social layer's influence. The covariance is set to 0.3, which determines the shape of the distribution. The factor is set to 7.0, which determines the maximum cost of the social layer.

```
robot_radius: 0.6 # distance a circu

obstacle_layer:
  enabled:                true
  min_obstacle_height:  -5.0
  max_obstacle_height:  10.0
  lethal_cost_threshold: 100
  unknown_cost_value: -1
  combination_method:   1
  track_unknown_space:  true    #tru
  obstacle_range: 9.0
  raytrace_range: 9.0
  publish_voxel_map: false
  observation_sources:  laser_scan
  laser_scan: {
    data_type: LaserScan,
    topic: uav1/scan,
    min_obstacle_height:  -5.0,
    max_obstacle_height:  10.0
  }

inflation_layer:
  enabled:                true
  cost_scaling_factor:  5.0  # expon
  inflation_radius:     1.0  # max.

static_layer:
  enabled:                true

social_layer:
  amplitude: 150.0
  covariance: 0.3
  factor: 7.0
```

(a.)

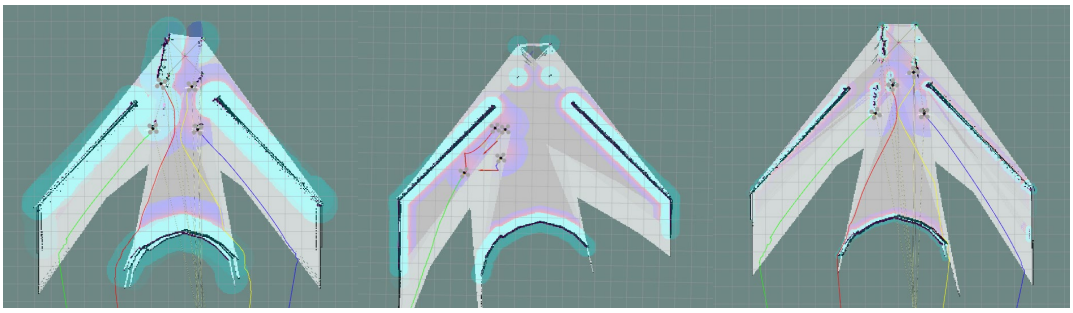
```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 1.0
  publish_frequency: 0.5
  static_map: true
  transform_tolerance: 0.5
  plugins:
    - {name: static_layer,          type: "costmap_2d::StaticLayer"}
    - {name: obstacle_layer,        type: "costmap_2d::ObstacleLayer"}
    - {name: inflation_layer,       type: "costmap_2d::InflationLayer"}
    - {name: social_layer,          type: "social_navigation_layers::ProxemicLayer"}
```

(b.)

```
local_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 14.0
  height: 14.0
  resolution: 0.05
  transform_tolerance: 0.5
  plugins:
   - {name: obstacle_layer,     type: "costmap_2d::ObstacleLayer"}
   - {name: inflation_layer,    type: "costmap_2d::InflationLayer"}
```

(c.)

Fig. 30. Costmap parameters configuration



(a.) (b.) (c.)

Global costmap using different parameters (inflation radius and cost scaling factor).
(a.) 1.0 and 2.0
(b.) 1.0 and 5.0
(c.) 0.2 and 1.0

Fig. 31. Global costmap using different parameters

The inflation radius is a parameter that governs the extent to which the obstacles present in the environment are dilated in the costmap. As depicted in Figures 31a and 31b, an increase in the inflation radius results in a more cautious trajectory for the UAV, as it endeavors to maintain a safe distance from obstacles. Nonetheless, an increased inflation radius may lead to the robot traversing a lengthier route to attain its objective, given that it

might have to circumvent impediments that it could have otherwise traversed.

The cost scaling factor is a crucial parameter that determines the relative importance of the costmap in the computation of the optimal path by the path planning algorithm. Increasing the cost scaling factor (as depicted in Figure 31b) will cause the path planning algorithm to assign greater importance to the costmap. This can result in the generation of safer trajectories that circumvent obstacles. Nevertheless, an elevated cost scaling factor may lead to a delay in the robot's achievement of its objective, as it could prioritize caution over effectiveness. The parameter depicted in Figure 31b is given greater priority to safety as opposed to efficiency. Figure 31c places a higher priority on efficiency over safety, whereas Figure 11a prioritizes both safety and efficiency.

3.3 Navigation

Global planner - The global costmap is used as the underlying graph for path planning by A* algorithm. Specifically, each cell in the costmap is represented as a node in the graph and edges are defined between adjacent cells in the costmap. The cost of each edge is based on the cost assigned to the cells in the costmap and any other environmental constraints that are considered during path planning.

To generate a path using A*, the algorithm searches the graph for the shortest path from the UAV's current location to the goal location while considering the costs of each node and edge in the graph. The algorithm uses a heuristic function that estimates the cost of reaching the goal from each node in the graph and uses this estimate to guide the search towards the goal.

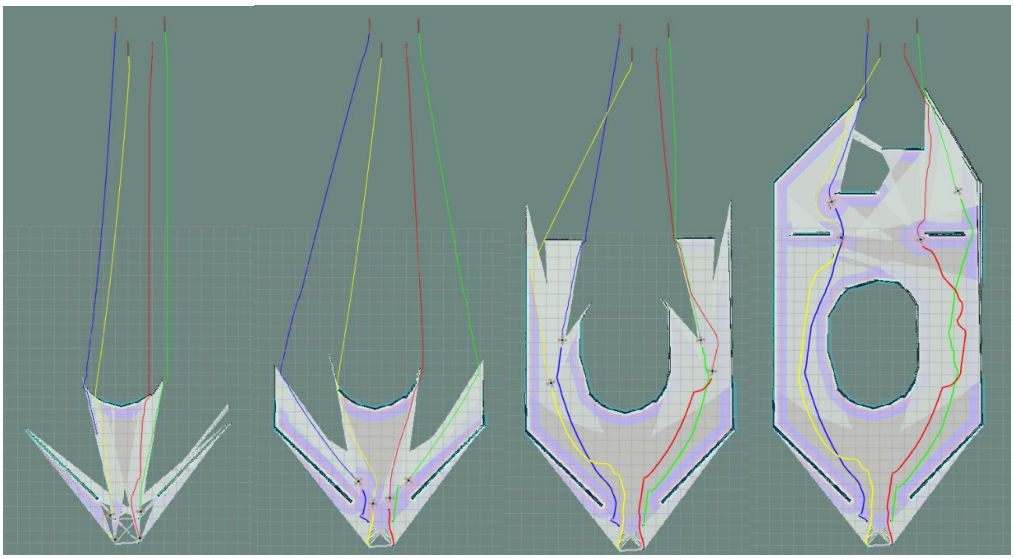


Fig. 32. Global path.

The A* algorithm can be configured using several parameters such as the heuristic

function, cost and goal tolerance. The heuristic function estimates the cost of reaching the goal from each node in the graph and can be used to guide the search towards the goal. The cost of each node and edge in the graph and can be configured based on the environmental constraints and the UAV's physical limitations. Finally, the goal tolerance parameter determines the distance and heading at which the goal is reached and can be set based on the accuracy of the UAV's sensors and the level of precision required for path planning.

```
GlobalPlanner:
  old_navfn_behavior: false
  use_quadratic: true
  use_dijkstra: false
  use_grid_path: false

  allow_unknown: true

  planner_window_x: 0.0
  planner_window_y: 0.0
  default_tolerance: 0.0

  publish_scale: 100
  planner_costmap_publish_frequency: 0.0

  lethal_cost: 253
  neutral_cost: 50
  cost_factor: 3.0
  publish_potential: true
```

Fig. 33. A* parameters.

The same as the costmap, the global planner is also implemented in move_base and called in a launch file named quadrotor_move_base.launch under quadrotor_navigation package. The A* parameters can be set in global_planner_params.yaml (Figure 33).

Here are the details of the parameters shown in Figure 33:

- default_tolerance: This parameter determines the default tolerance used by the Global Planner when checking whether the goal has been reached. The tolerance specifies how close the robot needs to be to the goal location in order for the planner to consider the goal reached.
- publish_scale: This parameter determines the resolution of the published potential map used by the Global Planner. The potential map is a representation of the cost of moving to each point in the map, and the publish_scale parameter determines the resolution of this map
- planner_costmap_publish_frequency: This parameter determines how often the Global Planner will publish the costmap used for planning. The costmap is a representation of the obstacles in the environment, and the planner_costmap_publish_frequency parameter determines how often this map is updated and published.
- lethal cost: This parameter determines the cost assigned to cells in the map that represent lethal obstacles, such as walls or large obstacles that the robot cannot pass

through. The maximum is 255. In this case, we consider lethal obstacles if the cost is more than 253.

- neutral cost: This parameter determines the cost assigned to cells in the map that represent neutral or unknown areas of the environment. In this case, area with the cost less than 50 is considered as safe area.
- cost_factor: This parameter determines the weighting factor applied to the cost of moving from one cell to another in the map. A higher cost factor will encourage the planner to avoid paths with high costs.
- publish_potential: This parameter determines whether the Global Planner will publish the potential map used for planning. Setting this parameter to true will publish the potential map, while setting it to false will not publish the map.

The local planner is responsible for generating a low-level control signal that specifies the velocity and orientation of the UAV to follow the global trajectory generated by the global planner. The purpose of the local planner is to adjust the UAV's velocity and orientation in real-time based on the obstacles and terrain features in the local environment to ensure that the UAV stays on the planned path and avoids collisions.

DWA is a popular algorithm used for local path planning in robotics. It uses a set of dynamically generated candidate trajectories to select a velocity and orientation that avoids obstacles and follows the global trajectory. The advantage of using DWA is that it can handle dynamic environments and adjust the UAV's velocity and orientation in real-time to ensure safe and efficient navigation.

To generate a local path using DWA, the algorithm uses the local costmap which provides a high-resolution map of the local environment around the UAV. The costmap is used to detect obstacles and determine the safest path to follow while staying close to the global trajectory generated by the global planner.

The DWA algorithm has several key parameters that can be adjusted for optimal performance. The first important parameter is the linear and angular velocity limits. These limits define the highest and lowest velocities that the UAV can attain and are based on the physical capabilities of the UAV.

Other important parameters are related to the weighting factors which balance the tradeoff between safety and efficiency when selecting a trajectory. These factors help to adjust the algorithm's preference for trajectories that prioritize either safety or efficiency depending on the needs of the application.

```
DWAPlannerROS:

# Robot Configuration Parameters
  max_vel_x: 1.5  # 0.55
  min_vel_x: -1.5

  max_vel_y: 1.0  # diff drive r
  min_vel_y: -1.0  # diff drive

  max_trans_vel: 1 # choose slig
  min_trans_vel: 0.5  # this is
  trans_stopped_vel: 0.1

  max_rot_vel: 5.0  # choose sli
  min_rot_vel: 0.4  # this is th
  rot_stopped_vel: 0.4

  acc_lim_x: 1.0 # maximum is th
  acc_lim_theta: 2.0
  acc_lim_y: 1.0      # diff dri

# Goal Tolerance Parameters
  yaw_goal_tolerance: 1.57  # 0.
  xy_goal_tolerance: 0.5  # 0.10

# Forward Simulation Parameters
  sim_time: 1.0       # 1.7
  vx_samples: 6       # 3
  vy_samples: 6       #1       #
  vtheta_samples: 20  # 20

# Trajectory Scoring Parameters
  path_distance_bias: 64.0
  goal_distance_bias: 24.0
  occdist_scale: 0.5
  forward_point_distance: 0.325
  stop_time_buffer: 0.2
  scaling_speed: 0.25
  max_scaling_factor: 0.2

# Oscillation Prevention Paramet
  oscillation_reset_dist: 0.05

# Debugging
  publish_traj_pc : true
  publish_cost_grid_pc: true
  global_frame_id: map
```

Fig. 34. DWA parameters.

The DWA local planner is also implemented in move_base and called in a launch file named quadrotor_move_base.launch under quadrotor_navigation package. The DWA parameters can be set in dwa_local_planner_params.yaml (see Figure 19 and Figure 34 for the code snippet). Figure 35 shows the difference of the time elapsed from the start to goal position between difference velocity limitations. This infers that the parameters in dwa_local_planner_params.yaml works as expected.

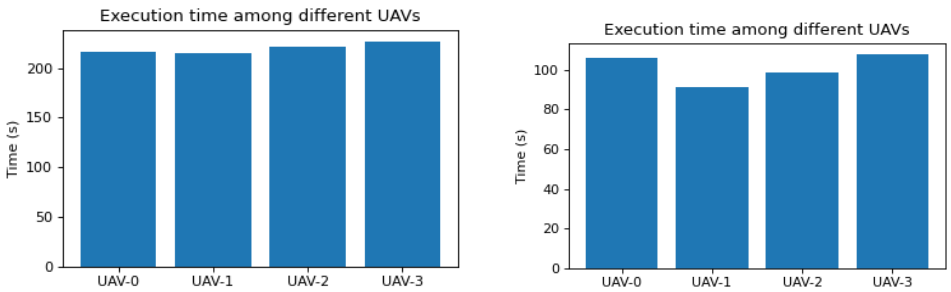


Fig. 35. Execution time between different velocity limit by using individual control.

The parameters changes are: (a.) max_vel_x=1.5, max_vel_y=-1.5, acc_lim_x=1.0 and

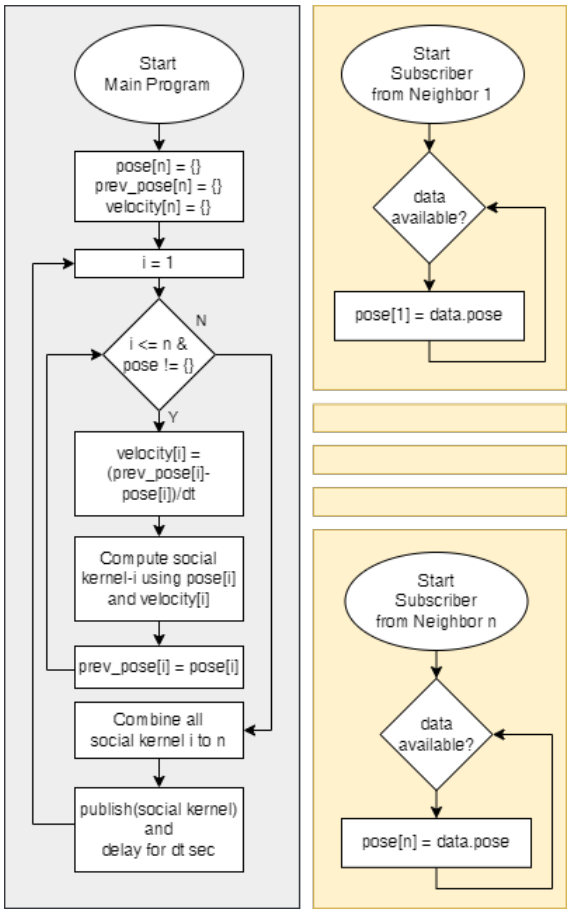
acc_lim_y=1.0 (b.) max_vel_x=4.0, max_vel_y=-4.0, acc_lim_x=2.0 and acc_lim_y=2.0.

As shown from this figure, higher velocity and acceleration improve flight performance. The drones complete the mission faster in figure b than figure a.

3.3 Collision Avoidance

To guarantee the free-collision behavior in a multi-agent system, the UAV should also consider both position and velocity of other UAVs. One of the solutions is exploiting the position of other UAVs to modify the costmap using social proximity technique.

The flowchart of technique for this social proximity behavior is shown in Figure 36 below:



(a.)

```
51  while not rospy.is_shutdown():
52      try:
53          people_msg = People()
54          people_msg.header.stamp = rospy.Time.now()
55          people_msg.header.frame_id = frame_id
56          for i,neighbor in enumerate(neighbors):
57              (trans, rot) = listener.lookupTransform(frame_id, neighbor[1:]+"base_footprint", rospy.Time(0))
58              person_msg = Person()
59              person_msg.name = neighbor
60              person_msg.position.x = trans[0]
61              person_msg.position.y = trans[1]
62              person_msg.position.z = trans[2]
63
64              if prev_pose_x[i] == 0:
65                  person_msg.velocity.x = 0.0
66                  person_msg.velocity.y = 0.0
67                  person_msg.velocity.z = 0.0
68              else:
69                  person_msg.velocity.x = trans[0]-prev_pose_x[i]
70                  person_msg.velocity.y = trans[1]-prev_pose_y[i]
71                  person_msg.velocity.z = 0.0
72
73              prev_pose_x[i] = trans[0]
74              prev_pose_y[i] = trans[1]
75              people_msg.people.append(person_msg)
76          people_pub.publish(people_msg)
77          rospy.sleep(0.5)
78      except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
79          continue
```

(b.) Code snippet related in social_proximity.py.

Fig. 36. Social proximity process.

From the flowchart above, a UAV has some process according to social proximity process. It has 1+n processes which are the main program which processes the social kernel and n-subscriber which obtains the neighbor's pose. The velocity is estimated by subtracting the current position with the previous position divided by the delta time.

The script related to social proximity calculation and how the UAV gets the position from the neighbors are implemented in social_navigation_layers and social_proximity.py respectively. In social_proximity.py, we consider the neighbors as People and one UAV as a person. The relative position is abstained using tf transformation ROS library. The parameters such as amplitude, cut-off, variance and factor can be set in costmap_common_params.yaml.

Line 53-55 shows how the people msg is initialized. From line 56, iterate for each neighbor. Get the relative position of each neighbor in line 57 and get the position relative (trans) to assign the person msg (line 58-62). In line 64, we check whether the previous position is existed or not. If not, we cannot estimate the velocity. Therefore, the velocity is assigned as zero (line 64-67). Otherwise, we can estimate the velocity (line 68-71). In line 73 and 74, the current pose is stored to the previous pose for the next iteration. After getting all the position and velocity of the person (neighbor of UAV), publish the message (line 76). Then, sleep for a particular second. In this code, we sleep for 0.5 second (line 77). Line 78-79 shows how If the relative position to neighbor is failed to get, It should continually try until the relative position is obtained.

3.4 Formation-based Leader-Follower Control

There are applications whereas the formation-based control is useful for. Target tracking in the military domain using a flock of UAVs, a single heavy payload transportation using multi UAVs and entertaining drone performance are some examples of application in formation-based control for multi UAVs. Moreover, the formation-based control, which maintains the distance among the UAVs, is beneficial for the communication between the UAVs. Without maintaining the distance, there is a possibility that a UAV is not in a line of sight with the others. This may cause a higher communication delay or even the connection can be disconnected.

The leader-follower method is a popular approach in formation-based control because it allows for a distributed control strategy where each agent only needs to communicate with its immediate neighbors. This can simplify the communication requirements for large groups of agents and make the overall system more scalable.

In the leader-follower method, one agent (the leader) is designated to determine the desired formation shape and trajectory, while the other agents (the followers) adjust their positions to maintain the desired formation relative to the leader. This method also allows for greater flexibility in the formation shape and trajectory since the leader can be changed dynamically.

There are several topologies in the leader-follower domain. However, in this work, we will compare three topologies: Leader Following (LF), Predecessor Following (PF) and Two-Nearest Predecessor Following (TNPF). Figure 37 shows a diagram for LF, PF and TNPF topologies.

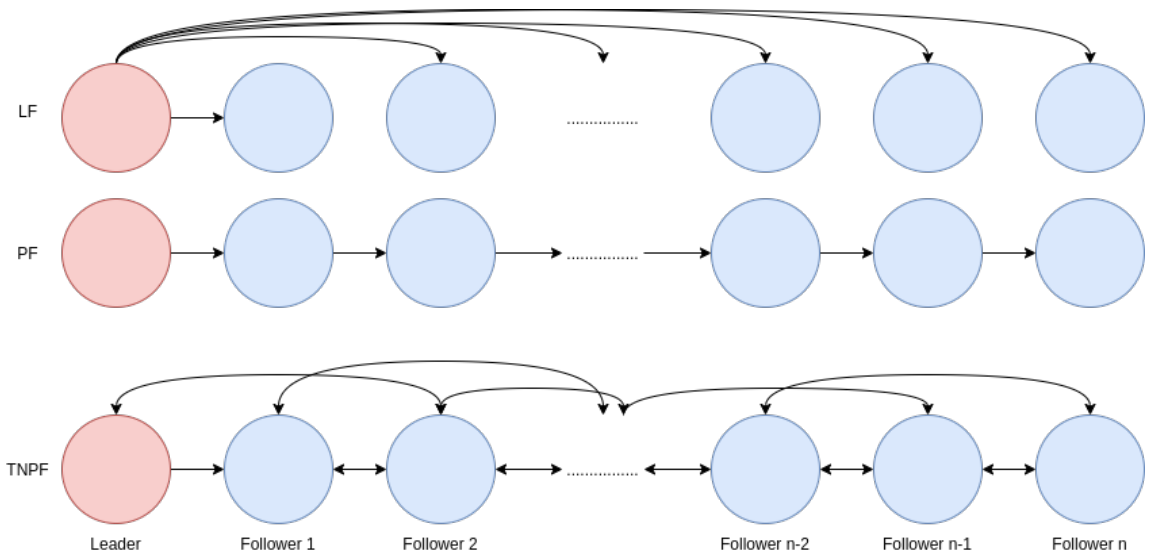


Fig. 37. Several topologies for leader-follower method

The general algorithm for leader-follower is shown in Figure 38 and 40 for leader and follower. What makes it different is the topology used. Each topology has a different rule for generating the goal for each follower.

```
Given: goal

Parameters: MAX_DISTANCE, MAX_TIME

state := 0
WHILE NOT reaching goal DO
    IF there is a follower with distance > MAX_DISTANCE THEN
        state := 1
    ELSE
        state := 0
    END IF

    IF state = 1 for more than MAX_TIME THEN
        state := 0
    END IF

    IF state = 0 THEN
        go to the goal using global and local planner
    ELSE
        waiting (do nothing)
    END IF
END WHILE
```

Fig. 38. Leader control algorithm pseudo code

The leader is not only going to the goal but is also waiting for others if there is a UAV which is far. In the algorithm above, there are two states. State zero is going to the goal and state one is waiting. The leader will change the state to waiting if there is a UAV with the relative distance more than MAX_DISTANCE. The leader may wait for infinite time because there may be a possibility that another UAV is stuck or crashes. To prevent that behavior, timeout is needed. Therefore, the leader will only wait for MAX_TIME seconds. The implementation of the leader control algorithm is in control_leader_follower.py. The code snippet is shown in Figure 39.

```
776     state = 0
777     listener = tf.TransformListener()
778     goal_published = False
779     while not rospy.is_shutdown():
780         poses = []
781         for uav in ['uav2/','uav3/','uav4/']:
782             try:
783                 (pose, rot) = listener.lookupTransform(uav+"base_footprint", "uav1/base_footprint", rospy.Time(0.0))
784                 poses.append(pose)
785             except:
786                 print("Cannot get the transformation between uav1 and",uav)
787
788         if len(poses) == 3: # the poses is available
789             is_it = is_there_far_followers(poses,max_distance)
790             if is_it and state == 0:
791                 state = 1
792                 print("WAITING FOR OTHERS")
793                 start_waiting = rospy.Time.now()
794             elif not is_it and state == 1:
795                 print("GO")
796                 state = 0
797
798             if state == 1:
799                 if rospy.Time.now()-start_waiting > rospy.Duration.from_sec(max_time):
800                     print("TIMEOUT! GO")
801                     state = 0
802
803             if state == 0:
804                 if not goal_published:
805                     client.send_goal(goal)
806                     goal_published = True
807             else: # state is 1
808                 client.cancel_goal()
```

Fig. 39. Code snippet for leader control.

The max_distance in line 709 belongs to MAX_DISTANCE and max_time in line 799 belongs to MAX_TIME parameters. The relative position between the leader and follower is

obtained using the tf transformation in line 783.

```
Given: final_goal, costmap

Parameters: COSTMAP_THRES, SEARCHING_RADIUS, TOPO

WHILE leader and this UAV not reach the final goal DO

    x_leader,y_leader := GET_LEADER_POSITION()
    x_goal, y_goal := GET_FORMATION_GOAL(x_leader, y_leader, TOPO)
    cost = GET_COST_FROM_COSTMAP(costmap, x_goal, y_goal)

    IF cost < COSTMAP_THRES THEN
        go to (x_goal, y_goal)
    ELSE
        x_goal_new, y_goal_new =
            GET_NEAREST_POSSIBLE_GOAL(x_goal,y_goal,x_leader,y_leader, SEARCHING_RADIUS)
        IF new goal obtained THEN
            go to (x_goal_new, y_goal_new)
        ELSE
            no updated goal (keep the last goal)
        ENDIF
    END IF

END WHILE
```

Fig. 40. Follower control algorithm pseudo code

The follower will always ask for the leader position (GET_LEADER_POSITION) and update its goal while the leader and itself have not yet reached the final goal. By using the leader position, each follower calculates the current goal based on the topology (TOPO) which are LF, PF or TNPF. How the GET_FORMATION_GOAL works is shown in Figure 41.

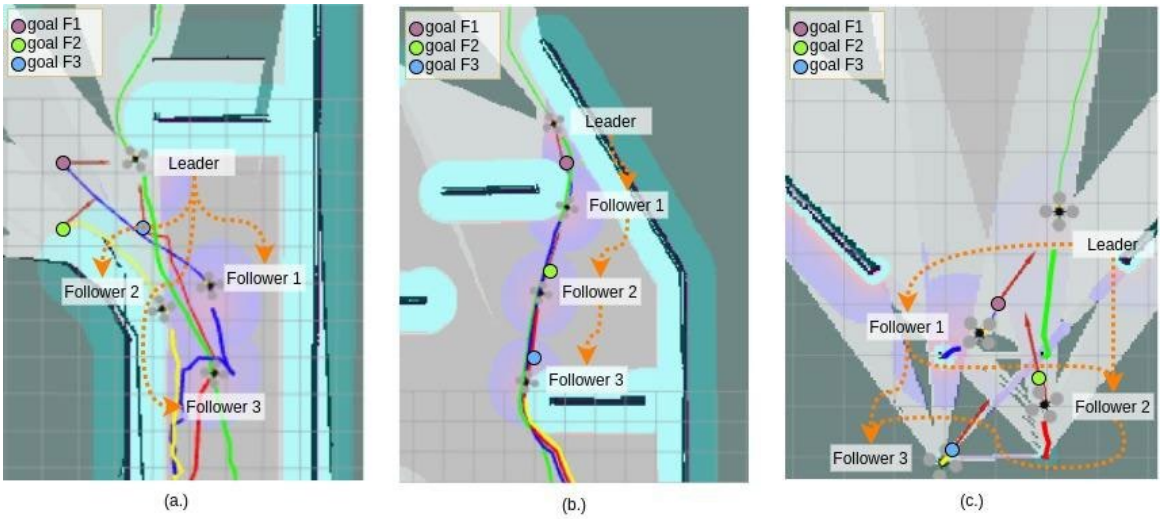


Fig. 41. How each leader-follower topology works.

(a.) LF (b) PF (c) TNPF

LF topology will always give rectangle formation as in Figure 41a. PF topology shape is like a snake or a line graph as it is seen in Figure 41b. The TNPF topology will search for

two nearest predecessor leaders and then get the center between those two nearest predecessor leader positions. The center coordinate is then pulled with a specific distance to the current position of the follower.

After getting the desired goal, the follower needs to get the cost of a costmap in that desired goal coordinate. If it is not considered as an obstacle (the cost is less than COSTMAP_THRES), then the follower can use that coordinate to go. Otherwise, the follower needs to search another nearest possible goal around the desired goal. The searching radius can be set using the SEARCHING_RADIUS parameter. The greater the searching radius value, the higher the possibility to get a new goal but more time consuming. If the new goal is found, the follower can go there. Otherwise, the old goal will not be updated. The follower should wait for the leader to move and make the follower have a valid new goal. The implementation of the follower algorithm is also written in control_leader_follower.py (see Figure 42).

Figure 42a shows the main program for the follower control. The relative position for the leaders and other neighbors are obtained using the tf transformation (line 589-595). If the leader has almost arrived (line 601), the followers should go to the ultimate goal. Otherwise, it should do the formation control (line 619-633) according to the topology (LF, PF or TNPF). Figure 42b shows a function for getting the nearest coordinate from the desired coordinate (x,y) using the SEARCHING_RADIUS parameter. This function iterates the grid from (x,y) to a wider area according to the SEARCHING RADIUS parameter. This function will return a new coordinate if a cell with the cost less than the max_cost or COSTMAP_THRES is found.

```python
            pos_obtained = False
            try:
                # get uav1_pos relative to each UAV's map
                (leader_pos, rot) = listener.lookupTransform("uav1/map", leader_base_footprint_id, rospy.Time(0.0))
                (uav1_pos, uav1_rot) = listener.lookupTransform(frame_id, leader_base_footprint_id, rospy.Time(0.0))
                for key, val in neighbors_pos.items():
                    (neighbors_pos[key], rot) = listener.lookupTransform(frame_id, key+"/base_footprint", rospy.Time(0.0))

                (my_pos, uav_rot) = listener.lookupTransform(frame_id, my_base_footprint_id, rospy.Time(0.0))
                pos_obtained = True
            except:
                print("Transformation cannot be obtained from",frame_id,"to",leader_base_footprint_id)

            if pos_obtained:
                if distance2d(leader_pos,leader_goal) < 2.0:
                    is_leader_arrived = True
                    client.cancel_goal()
                    rospy.sleep(1.0)
                    # go to ultimate goal
                    goal = MoveBaseGoal()
                    goal.target_pose.header.stamp = rospy.Time.now()
                    goal.target_pose.header.frame_id = namespace[1:]+'map'
                    goal.target_pose.pose.position.x = goal_x
                    goal.target_pose.pose.position.y = goal_y

                    quaternion = tf.transformations.quaternion_from_euler(0.0, 0.0, 0.0)

                    goal.target_pose.pose.orientation.x = quaternion[0]
                    goal.target_pose.pose.orientation.y = quaternion[1]
                    goal.target_pose.pose.orientation.z = quaternion[2]
                    goal.target_pose.pose.orientation.w = quaternion[3]
                    client.send_goal(goal)
                if topology == 'LF':
                    leader_pos = copy.deepcopy(uav1_pos)
                    relative_goal = copy.deepcopy(relative_goal_default)
                elif topology == 'PF':
                    leader_pos = copy.deepcopy(neighbors_pos['uav'+str(int(namespace[-2])-1)])
                    relative_goal,leader_pos = get_relative_goal(leader_pos,my_pos,distance,topology)
                elif topology == 'TNPF':
                    leader_pos = get_two_nearest_pred_leader(neighbors_pos, my_pos, 'uav'+namespace[-2])
                    relative_goal,leader_pos = get_relative_goal(leader_pos,my_pos,distance,topology)

                goal = update_goal(leader_pos, my_pos, relative_goal, cost_threshold, searching_radius, topology)
                if goal:
                    goal_pub.publish(goal)
                else:
                    print("NO GOAL FOR", namespace)
```

(a.)

```python
def get_nearest_low_cost(x, y, max_cost, searching_radius, leader_pos):
    global grid

    # Calculate the current cell index
    current_idx = positionToIndex(x,y)

    radius = grid.info.resolution*2
    while radius < searching_radius:
        radius = np.floor(radius * 10) / 10
        for i in np.arange(-radius, radius + grid.info.resolution, grid.info.resolution):
            for j in np.arange(-radius, radius + grid.info.resolution, grid.info.resolution):
                if i < 0:
                    i = np.ceil(i * 10) / 10
                else:
                    i = np.floor(i * 10) / 10
                if j < 0:
                    j = np.ceil(j * 10) / 10
                else:
                    j = np.floor(j * 10) / 10

                if (i > -radius and i < radius) and (j > -radius and j < radius):
                    continue

                if get_cost(x+i, y+j) >= max_cost:
                    continue

                # Calculate the x, y coordinate of the new cell
                new_x = x+i
                new_y = y+j

                return new_x, new_y
        radius += grid.info.resolution*2

    return None, None
```

(b.)

```python
def get_cost(x, y):
    global map_data, map_resolution, map_width, map_height
    global last_global_costmap
    global last_global_costmap_raw
    global namespace
    global grid

    if grid is None:
        return float('inf')

    idx = positionToIndex(x,y)
    if idx >= 0 and idx < len(grid.data):
        return grid.data[idx]
    return float('inf')
```

(c.)

```python
def update_goal(leader_pos, my_pos, goal_offset, cost_threshold, searching_radius, topology):
    global namespace
    x = leader_pos[0] + goal_offset[0]
    y = leader_pos[1] + goal_offset[1]

    cost = get_cost(x, y)
    # Update goal position
    if cost < cost_threshold:
        theta = np.arctan2(leader_pos[1]-y,leader_pos[0]-x)
        q = tf.transformations.quaternion_from_euler(0, 0, theta)

        goal = PoseStamped()
        goal.header.stamp = rospy.Time.now()
        goal.header.frame_id = namespace[1:]+"map"
        goal.pose.position.x = x
        goal.pose.position.y = y
        goal.pose.orientation.x = q[0]
        goal.pose.orientation.y = q[1]
        goal.pose.orientation.z = q[2]
        goal.pose.orientation.w = q[3]

        return goal
    else: # get nearest desired goal
        x = (x+leader_pos[0])/2.0
        y = (y+leader_pos[1])/2.0
        x, y = get_nearest_low_cost(x, y, cost_threshold, searching_radius, leader_pos)
        if x is None:
            return None
        else:
            #print("GOT NEAREST GOAL")
            #theta = np.arctan2(y-my_pos[1],x-my_pos[0])
            theta = np.arctan2(leader_pos[1]-y,leader_pos[0]-x)
            q = tf.transformations.quaternion_from_euler(0, 0, theta)

            goal = PoseStamped()
            goal.header.stamp = rospy.Time.now()
            goal.header.frame_id = namespace[1:]+"map"
            goal.pose.position.x = x
            goal.pose.position.y = y
            goal.pose.orientation.x = q[0]
            goal.pose.orientation.y = q[1]
            goal.pose.orientation.z = q[2]
            goal.pose.orientation.w = q[3]

            return goal
    return None
```

(d.)

```python
def get_relative_goal(leader_pos, my_pos, distance,topology):
    global namespace
    if topology == "TNPF":
        if len(leader_pos) == 1:
            topology = 'PF'
            leader_pos = leader_pos[0]
        else:
            # get center between two leaders

            leader1 = leader_pos[0]
            leader2 = leader_pos[1]
            center_x = (leader1[0]+leader2[0])/2
            center_y = (leader1[1]+leader2[1])/2
            theta = np.arctan2(center_y-my_pos[1],center_x-my_pos[0])
            target_x = -(distance*np.cos(theta))
            target_y = -(distance*np.sin(theta))
            return (target_x, target_y),(center_x, center_y)

    if topology == 'PF':
        theta = np.arctan2(leader_pos[1]-my_pos[1],leader_pos[0]-my_pos[0])
        target_x = -(distance*np.cos(theta))
        target_y = -(distance*np.sin(theta))
        return (target_x, target_y),leader_pos
```

(e.)

```python
def get_two_nearest_pred_leader(neighbors,my_pos,name):
    # filter predecessor only
    neighbors_pred = {}
    for key, val in neighbors.items():
        if key < name:
            neighbors_pred[key] = val

    d = {}
    for key, val in neighbors_pred.items():
        d[key] = distance2d(my_pos, val)
    min_key = min(d, key=d.get)
    first_neighbor = neighbors_pred[min_key]
    del d[min_key]
    if len(d) == 0:
        return [first_neighbor]
    else:
        min_key = min(d, key=d.get)
        second_neighbor = neighbors_pred[min_key]
        return [first_neighbor, second_neighbor]
```

(f.)

Fig. 42. Snippet codes for follower control

(a.) main program which is executed after taking off (b.) code for finding another coordinate with SEARCHING_RADIUS (c.) code for getting the cost in a particular coordinate (d.) code for validating and updating the goal after it has the desired goal (e.) code for getting the desired goal according to the topology used (f.) code for TNPF in getting the nearest two predecessor leader.

Figure 42c is useful for obtaining the cost in particular coordinate of the costmap. If the coordinate is outside the costmap, then, it returns infinite. Figure 42d shows how the desired goal is validated and updated depending on the value of costmap. Line 353 and 354 calculate the desired goal. Line 356 gets the cost. Line 358 checks whether the cost is less than the threshold or not. If that condition is satisfied, then, we can directly update the goal (line 359-372). Otherwise, it should run the function that belongs to Figure 42b. If a new goal is found, then we can update the goal. Otherwise, keep the old goal as it is.

Figure 42e shows the implementation of the calculation of desired goal by using PF or TNPF topology. Line 437-451 shows how the TNPF calculates the desired goal from two leaders. Line 453-457 shows how the PF topology is simpler than TNPF as it only considers one predecessor leader. Figure 42f shows how two nearest leaders are obtained. Line 496-499 shows how to keep only the predecessor neighbors. The set of distance is calculated in line 501-503. The nearest neighbor is found in line 504 and 505. This should also consider the first follower which only has one leader (line 507-508). If this follower is not the first follower (has predecessor more than one), then we should find the second nearest neighbor (Line 510-512).

3.5 Comparison of different formation-based swarm.

In this section, the result and analysis of comparing some leader-follower topology will be explained. The three topologies (LF, PF and TNPF) are also compared with a distributed individual strategy labeled with “None”. Some metrics which correlated with obstacle and collision avoidance in multi-UAVs system are the mission time, blocking time, distance to others, trajectory length, and the distance to the nearest obstacle. There are two parts in this section. Part 1 is focused on comparing different topologies and part 2 is focused on different parameters (speed, SEARCHING_RADIUS and COSTMAP_THRES) using LF topology.

It is expected that without any topology, the group of UAVs will arrive at the goal efficiently (see Figure 43). However, it is not the objective as the distance among the UAVs should be maintained for several purposes. Among LF, PF and TNPF topology, LF topology

has the most efficient time but it is just slightly better than the others. LF topology is considered as efficient topology as it has the least blocking time as it is shown in Figure 44. The blocking time in this case is the time when a UAV is not moving in a particular time. In our case, we consider a UAV to be blocked if it moves less than 0.1 meters for a second. The PF topology has the worst efficiency as it has the highest blocking time. With this topology, it seems that each follower is waiting for each other. This behavior causes a higher possibility of blocking time.

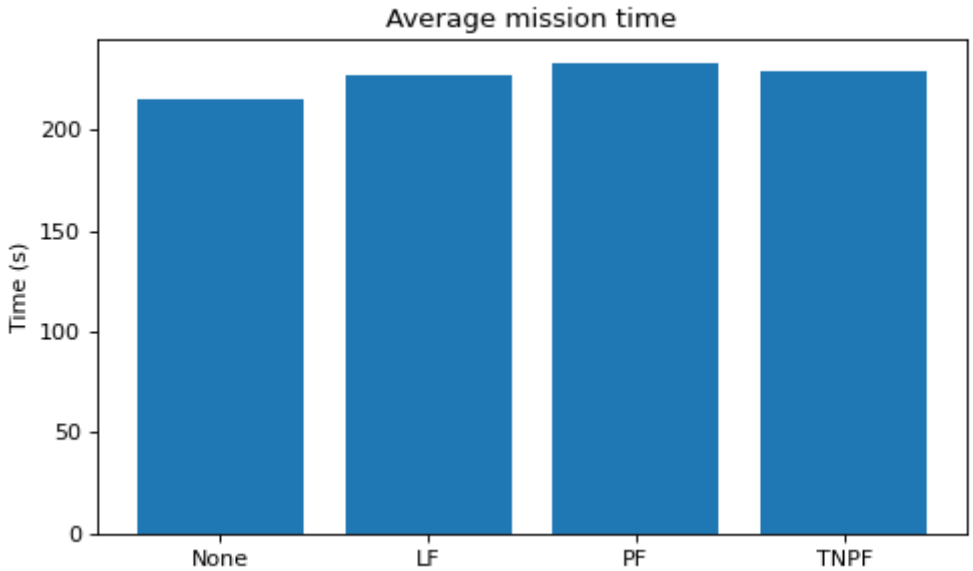


Fig. 43. The average mission time among different topologies

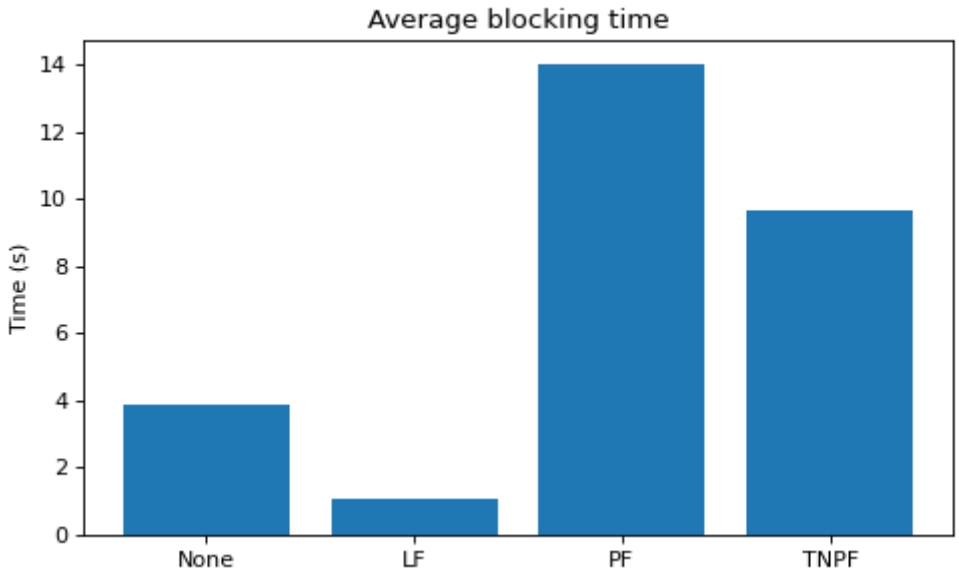


Fig. 44. The average blocking time among different topologies

In this experiment, the static obstacle is simple in early time. At the end, there are more obstacles which makes the obstacle avoidance more challenging. In formation-based

control, if the obstacle is complex and the space is narrow, then the formation will be more chaotic. In Figure 45, it is shown that by using LF topology, the distance between the UAVs is near at first. Nevertheless, after t=150s, the distance significantly increases due to the presence of more challenging obstacles.

Therefore, in a complex environment, PF and TNPF topology behavior are more stable. However, the TNPF topology is better than PF topology as it has two nearest leaders instead of only one nearest leader. Furthermore, it can be investigated that without any formation, the distance among the UAVs is not controllable. A UAV which is separated with a long distance from other UAVs in a real environment contribute to lack of connection with the others.

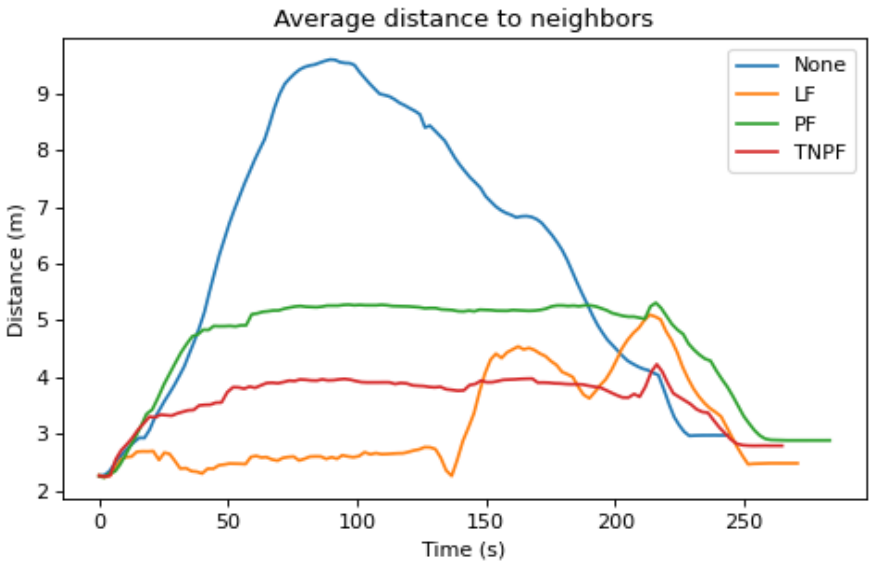


Fig. 45. The average distance to neighbors among different topologies

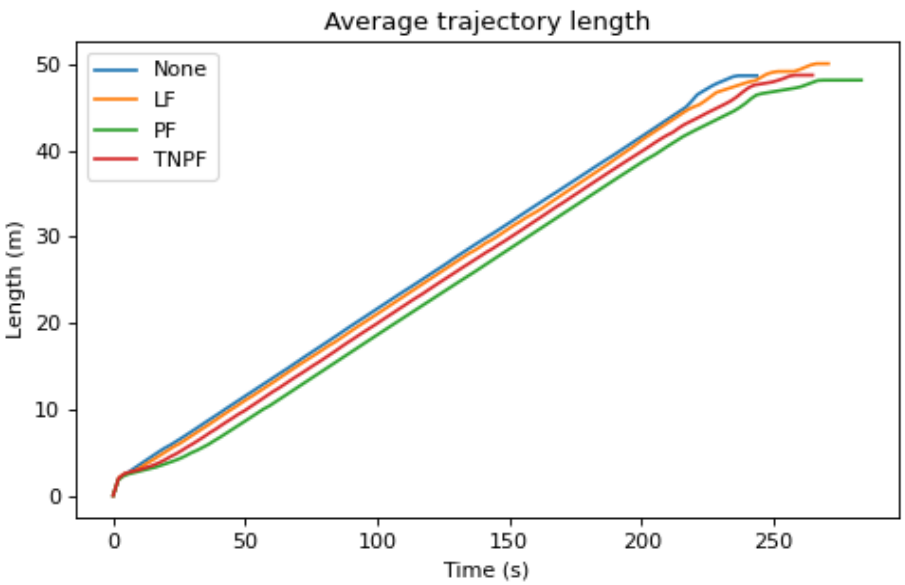


Fig. 46. The accumulative trajectory length among different topologies

Figure 46 shows the average accumulative trajectory length. The result shows that there is no significant difference among the strategies. Interestingly, the individual strategy has the longest trajectory. This metric can also imply the energy consumption as the longer the UAV traveled, the more the battery power used although the time should also be taken into account. The shortest trajectory obtained by using PF topology as the formation is more stable, only considering one nearest leader.

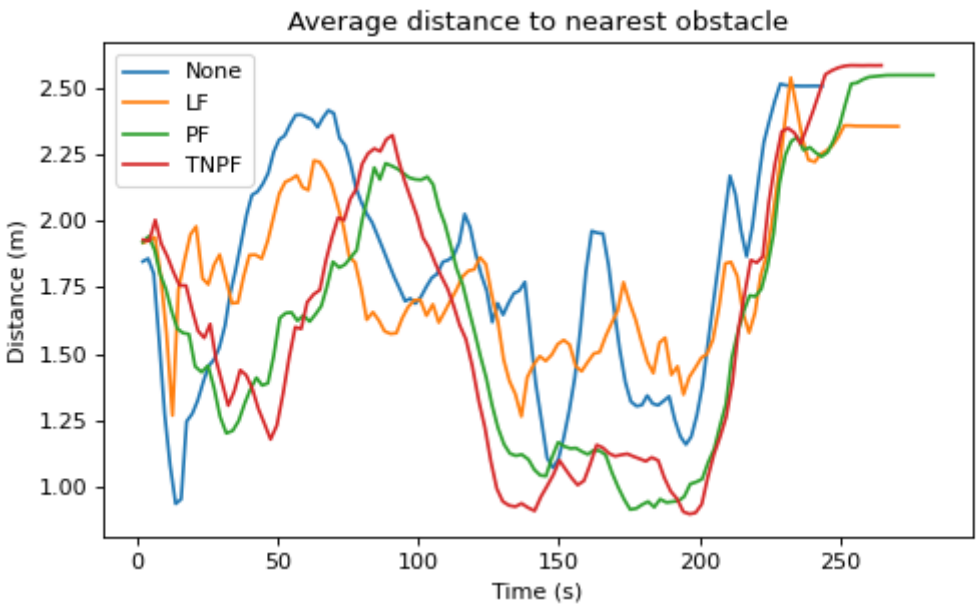


Fig. 47. The average distance to nearest obstacle among different topologies

Figure 47 indicates the safety of each strategy in terms of collision and obstacle avoidance.

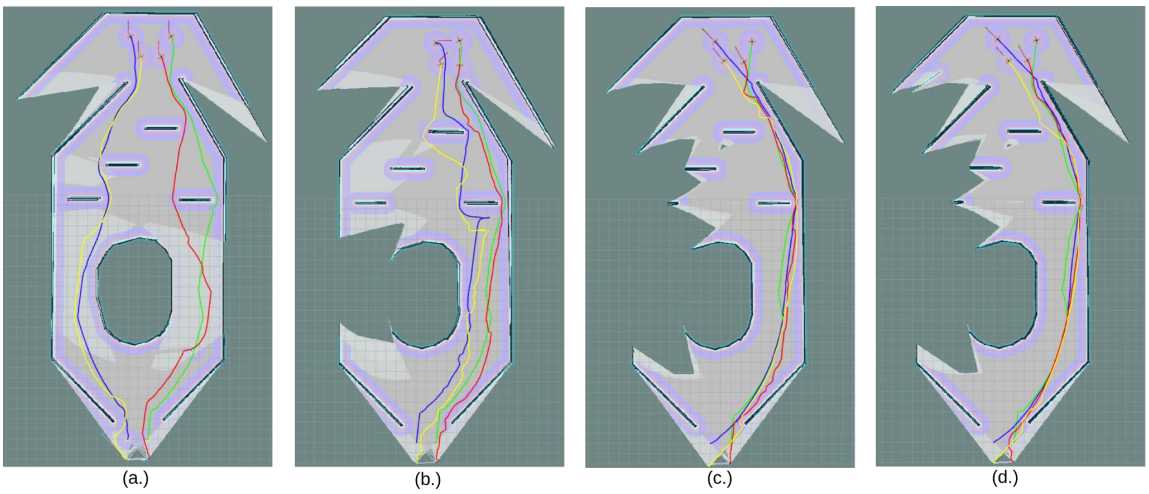


Fig. 48. The trajectory of using different topologies.

(a.) None (b.) LF (c.) PF (d.) TNPF

It seems that LF topology is the safest topology. This is correlated with the trajectory shown in Figure 48. Using PF (Figure 48c) and TNPF (Figure 48d) topology causes four

UAVs to navigate through a narrow gap. By using LF (Figure 48b) topology, several UAVs split up to find another space.

3.6. Compare different parameters using LF topology

In this subsection, the focus is on comparing different parameters (speed, SEARCHING_RADIUS or SR, COSTMAP_THRES or CT). The speed labeled with "Slow" has the linear velocity limit 1.5 m/s and linear acceleration limit 1 m/s^2. The speed labeled with "Fast" has the linear velocity limit 5.0 m/s and linear acceleration limit 2 m/s^2. The topology used in this subsection is LF topology as it is the fastest topology evaluated in subsection 3.5.

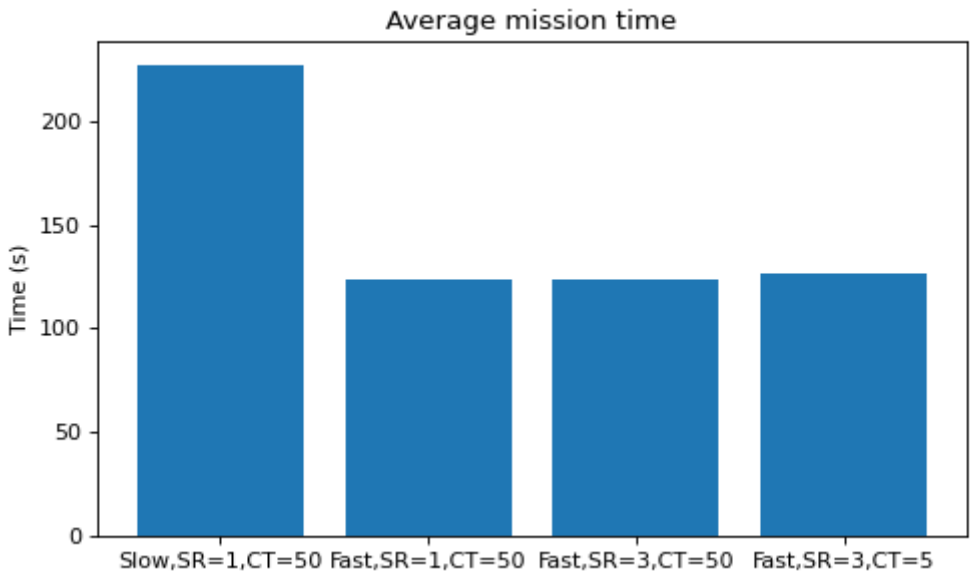


Fig. 49. The average mission time among different parameters

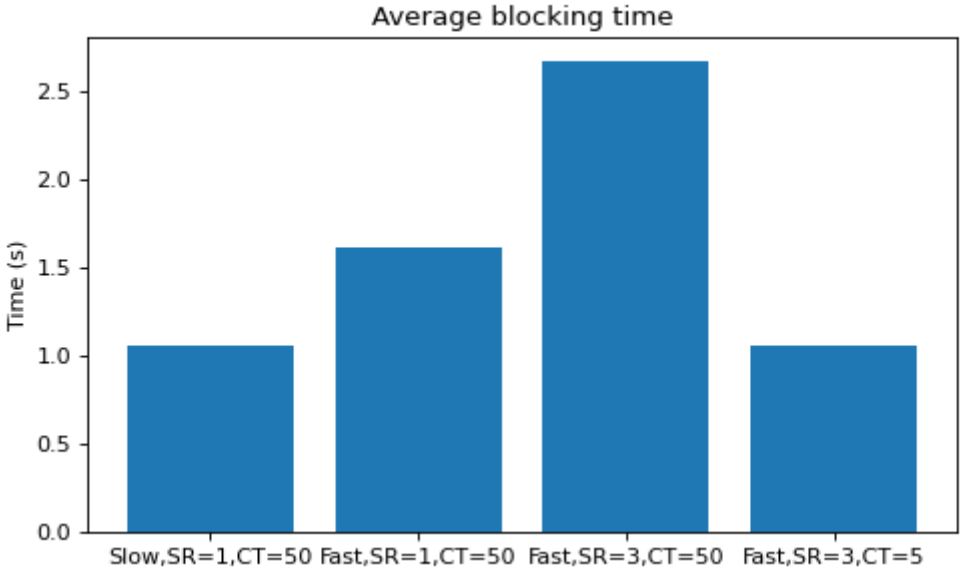


Fig. 50. Blocking time among different parameter

It is expected that increasing the velocity and acceleration limit also increases the efficiency. However, changing the other parameters does not sagnificanly improve the efficiency. The average mission time as it is seen in Figure 49 among different parameters with "Fast" speed seems similar although the blocking time (see Figure 50) is different.

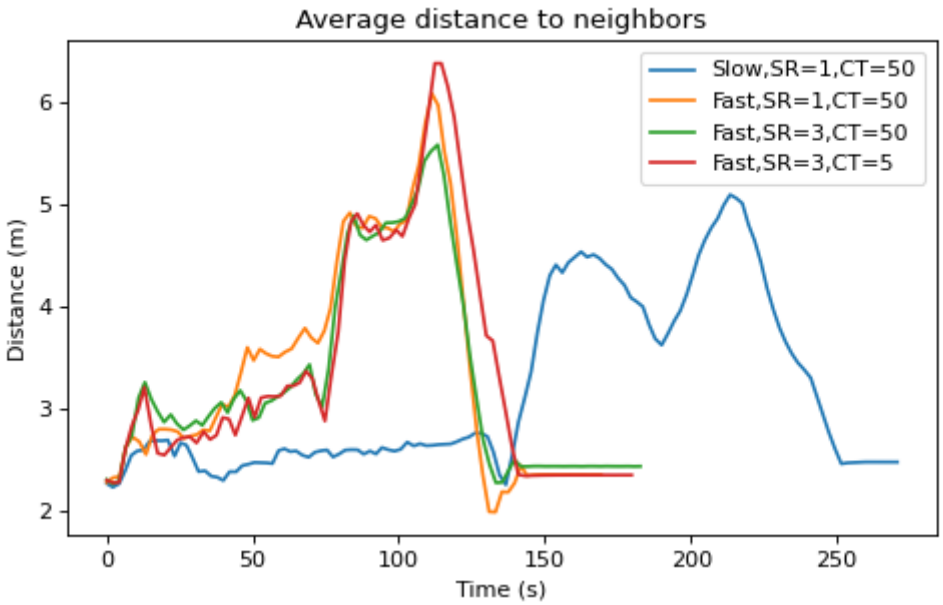


Fig. 51. The average distance among different parameters

In Figure 51, it is investigated that the average distance to the neighbors increases with the increasing of the speed. However, it is difficult to judge the difference of distance to the neighbors by using different searching radius and costmap threshold values.

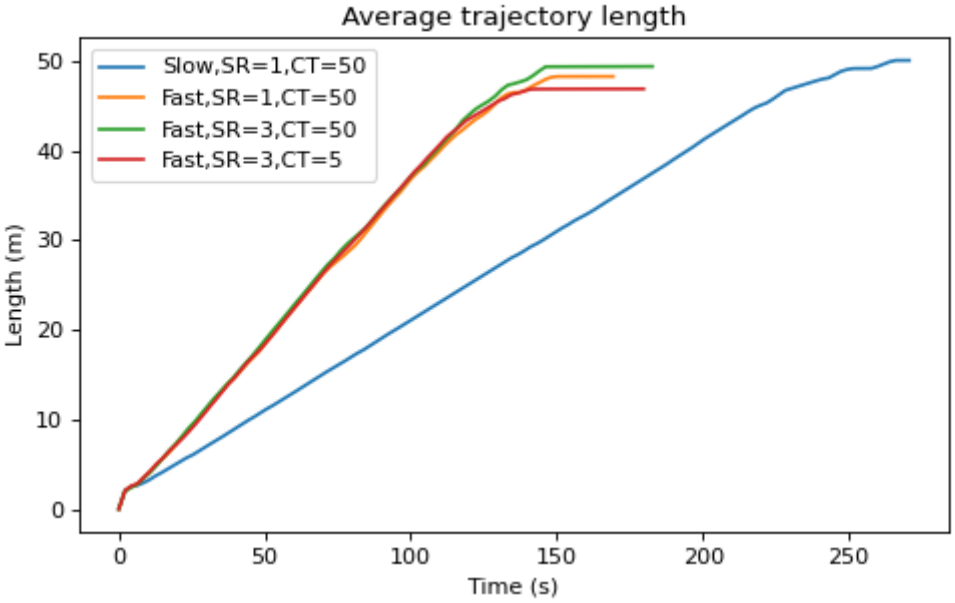


Fig. 52. Average trajectory length among different parameters

This can also be seen from the trajectory length which is almost similar (see Figure 52), whereas the trajectory by using different parameters is identical. However, it seems that a larger searching radius will maintain the distance slightly better. Increasing or decreasing the costmap threshold does not infer anything about the formation maintenance. More comprehensive experiments may be needed to get the correlation between the costmap threshold and the formation maintenance.

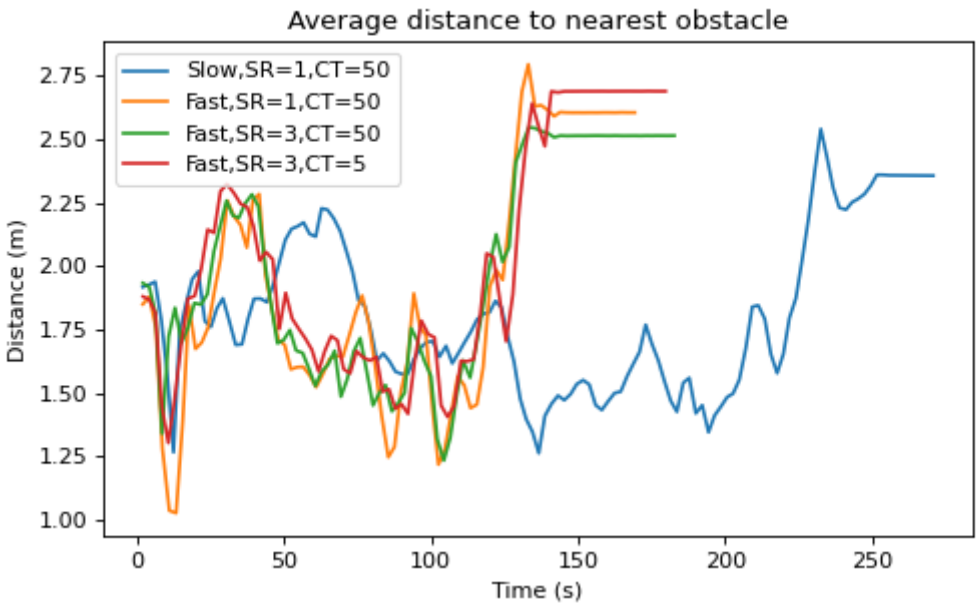


Fig. 53. Average distance to nearest obstacle among different parameters

Figure 53 shows the distance with the nearest obstacle over time. It is expected that increasing the searching radius and decreasing the costmap threshold will slightly improve the

safety factor of this formation-based leader-follower control. By increasing the searching radius, the possibility to find a safer area is higher. Also, by having a lower costmap threshold, the selection of the desired goal further away from the obstacle. This implies that shrinking the costmap makes the UAV movement safer from collisions.

3.7 Conclusion

Each topology has advantages and disadvantages related to trajectory length, safety and formation maintenance.

Table 8. Performance of different formation-based swarms.

Results	**LF**	**PF**	**TNPF**
Average distance to obstacles	1.81 m	1.67 m	1.62 m
Trajectory length	50.04 m	48.13 m	48.71 m
Deviation from neighbors	0.89 m	0.09 m	0.08 m

1. The most efficient topology is LF topology as it has more safe behavior in terms of distance to obstacles. It has the lowest possibility of colliding with the obstacle.

2. However, PF and TNPF topologies are more promising for maintaining the formation. In addition, PF topology has slightly shorter trajectory than others.

3. LF topology does not show satisfying performance in terms of keeping formation in contrast PF and TNPF topologies have significantly better performance as shown in table 8.

4. Changing the parameters related to the leader-follower control does not give significant improvement in efficiency in formation and trajectory. However, it improves safety and mission time. By increasing the searching radius and decreasing the costmap threshold, the UAVs will find more safe area.

CONCLUSIONS

1) UAV swarm is extensively used for military purposes

2) The deliberate approach has been found to be effective in accomplishing challenging objectives within a stable setting.

3) The reactive approach has been developed with the purpose of maneuvering around dynamic obstacles.

4) The hybrid architecture integrates reactive and deliberative capabilities.

5) The most efficient topology is LF topology as it has more safe behavior in terms of distance to obstacles. It has the lowest possibility of colliding with the obstacle**.**

6) PF and TNPF topologies are more promising for maintaining the formation.

REFERENCES

David Hambling. “The ‘Magic Bullet’ Drones Behind Azerbaijan’s Victory Over Armenia.” Forbes, 2020.

Lagkas, T.; Argyriou, V.; Bibi, S.; Sarigiannidis, P. UAV IoT Framework Views and Challenges: Towards Protecting Drones as “Things”. Sensors 2018,

Zhu, K.; Liu, X.; Pong, P.W.T. Performance Study on Commercial Magnetic Sensors for Measuring Current of Unmanned Aerial Vehicles. 2019, 1397–1407. [CrossRef]

Galtarossa, L.; Navilli, L.F.; Chiaberge, M. Visual-Inertial Indoor Navigation Systems and Algorithms for UAV Inspection Vehicles. 2020; pp. 1–16.

Pérez, M.C.; Gualda, D.; Vicente, J.; Villadangos, J.M.; Ureña, J. Review of UAV positioning in indoor environments and new proposal based on US measurements. 2019; pp. 267–274.

Paredes, J.A.; Álvarez, F.J.; Aguilera, T.; Villadangos, J.M. 3D Indoor Positioning of UAVs with Spread Spectrum Ultrasound and Time-of-Flight Cameras. Sensors 2017; pp. 18-89.

Jang, G.; Kim, J.; Yu, J.-K.; Kim, H.-J.; Kim, Y.; Kim, D.-W.; Kim, K.-H.; Lee, C.W.; Chung, Y.S. Review: Cost-Effective Unmanned Aerial Vehicle (UAV) Platform for Field Plant Breeding Application. Remote. Sens. 2020.

Adamopoulos, E.; Rinaudo, F. UAS-Based Archaeological Remote Sensing: Review, Meta-Analysis and State-of-the-Art. Drones 2020.

Chand, B.N.; Mahalakshmi, P.; Naidu, V.P.S. Sense and avoid technology in unmanned aerial vehicles: 2017; pp. 512–517.

S. Plathottam and P. Ranganathan, “Next Generation Distributed and Networked Autonomous Vehicles: Review,” 2018.

NVIDIA, “NVIDIA Announces World’s First AI Computer to Make Robotaxis a Reality,” 2017.

N. Smolyanskiy, A. Kamenev, J. Smith, and S.Birchfield, “Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness,” 2017 pp. 4241–4247.

Y. Zhou, J. Li, L. Lamont, and C. A. Rabbath, “Modeling of packet dropout for UAV wireless communications,” 2019 pp. 677–682,

S.Huang, R.S.H.Teo, Computationally efficient visibility graph-based generation of 3D shortest collision-free path among polyhedral obstacles for unmanned aerial vehicles, 2019 pp1218–1223.

X.Zhang,S.Huang, W.Liang,K.K.Tan, HLT*: Real-time and any-angle path planning in 3D environment, 2019, pp. 14-17

H. Zhu and J. Alonso-Mora, Chance-constrained collision avoidance for maps in dynamic environments,IEEE Robotics and Automation Letters, 2019, pp. 776-783.

X.Zhang,J.Ma, S.Huang,Z.Cheng,T.H.Lee, Integrated planning and control for collision-free trajectory generation in 3D environment with obstacles, Proc.of the 19th International Conference on Control, Automation and Systems, 2019

S.Huang, R.S.H.Teo, W.Liu, Distributed cooperative collision avoidance control for multi-unmanned aerial vehicles, Actuators,8(1)(2019), pp.1– 25.

A. Andreychuk, K. Yakovlev, D. Atzmon, R. Stern, Multi-agent pathfinding with continuous time, 2019

J. Li, D. Harabor, P. Stuckey, A. Felner, H. Ma, S. Koenig, Disjoint splitting for multi-agent path finding with conflict-based search. Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling 2019, pp.279–283.

W. H¨onig, S. Kiesel, A. Tinka, J. W. Durham, N. Ayanian, Conflictbased search with

optimal task assignment, Proceedings of the 17th International Conference on Autonomous Agents and Multi Agent Systems 2018, pp. 757–765

A. Felner, R. Stern, S. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N. Sturtevant, G. Wagner,P. Surynek, Search-based optimal solvers for the multi-agent pathfinding problem. 2018, pp. 20-37

W.H¨onig, S. Kiesel, A. Tinka, J. W.Durham, N. Ayanian, Persistent and robust execution of MAPF schedules in warehouses.IEEE Robotics and Automation Letters, 2019, pp.1125–1131

https://dronelife.com/2021/04/28/drone-swarms-for-firefighting-the-future-of-fire-supression/

https://www.yenisafak.com/ekonomi/bayraktar-tb2-16-ulkede-ucuyor-3755997

https://www.sciencedirect.com/topics/engineering/unmanned-aerial-vehicle
https://www.bhphotovideo.com/explora/video/buying-guide/introduction-drones-and-uavs

J. R. Galán-Martín, M. A. Gómez-Martínez, J. M. Cañas-Guerrero, and M. A. Vega-Rodríguez, "ROS-based indoor swarming of drones for surveillance missions," 2018, pp. 821-829.

M. P. Kumar, S. S. S. Kumar, and S. S. Kumar, "Autonomous indoor navigation of drone swarm using ROS and machine learning," 2018, pp. 2316-2321.

J. Wang, H. Li, Z. Li, and Y. Li, "Swarm navigation of indoor drones based on ROS and deep reinforcement learning," Sensors, 2019.

J. Luo, Y. Zhang, L. Zhang, and J. Li, "Cooperative SLAM-based swarm indoor navigation with heterogeneous MAVs," Robotics and Autonomous Systems, 2019.

Y. Liu, X. Ren, Y. Zhang, and Y. Xu, "Autonomous indoor navigation of multiple drones using a hybrid approach based on ROS," pp. 171-184, 2019.