DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Distributed management and coordination of UAV swarms based on infrastructureless wireless networks

By

Jamie Wubben

Advisors:
Prof. Dr. Carlos Tavares Calafate
Prof. Dr. Juan Carlos Cano

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Valencia, España

July 2023

To you, the reader;

That you might enjoy reading this thesis as
much as I did working on it.

We are all just the average of the five people around us; and I, was fortunate enough to surround myself with the five most brilliant, and compassionate minds.

Jamie Wubben

Acknowledgements

For the reader that does not know me, this acknowledgement might seem rather short. They, that do know me, probably realize that I have never been a man that expresses a lot of emotion. Nevertheless, I really do want to thank both of my advisors, Carlos Tavares Calafate and Juan-Carlos Cano, for their support and excellent guidance. Without them, I would never be able to finish this work that I proudly present. In the same way, I would like to express my gratitude for the entire team from the Computer Networks Research Group (GRC). They, too, have contributed with their knowledge, but most of all by creating a wholesome workplace where I am working many hours, but never with reluctance. Over the years, working in this amazing team, I have come to realize that this wholesome workplace is probably the biggest key to my success, and I am honored that I can call myself a member of GRC. Finally, to my parents who, although with some difficulties, allowed me to continue my studies abroad and provide continuous support.

Jamie Wubben
Valencia, July 7, 2023

Abstract

Unmanned Aerial Vehicles (UAVs) have already proven to be useful in many different applications. Nowadays, they are used for photography, cinematography, inspections, and surveillance. However, in most cases they are still controlled by a pilot, who at most is flying one UAV at a time. In this thesis, we try to take this technology one step further by allowing multiple Vertical Take-off and Landing (VTOL) UAVs to work together as one entity. The main advantage of this group, commonly referred to as a swarm, is that it can perform more complex tasks than a single UAV. When organized correctly, a swarm allows for: more area to be covered in the same time, more resilience, higher load capability, etc. A swarm can lead to new applications, or a better efficiency for existing applications. A key part, however, is that they should be organized correctly. During the flight, different disturbances will make it complicated to keep the swarm as one coherent unit. Once this coherency is lost, all the previously mentioned benefits of a swarm are lost as well. Even worse, the chance of a hazard increases. Therefore, this thesis focuses on solving some of these issues by providing a baseline of building blocks that enable other developers to create UAV swarm applications.

In order to develop these building blocks, we improve a multi-UAV simulator called ArduSim. This simulator allows us to simulate both the physics of a UAV, and the communication between UAVs with a high degree of accuracy. This is a crucial part because it allows us to deploy (well tested) protocols and algorithms on real UAVs with ease. During the entirety of this thesis, ArduSim has been used extensively. It made testing safe, and allowed us to save a lot of time, money and research effort.

We started by assigning airborne positions for each UAV on the ground. Assuming that the UAVs, are placed randomly on the ground, and that they need to

reach a desired aerial formation, we searched for a solution that minimizes the total distance travelled by all the UAVs. We started with a brute-force method, but quickly realized that, given its high complexity, this method performs badly when the number of UAVs grows. Hence, we created a heuristic. As for all heuristics, a trade-off was made between complexity and accuracy. By simplifying the problem, we found that our heuristic was able to calculate a solution very quickly without increasing the total distance travelled substantially. Furthermore, we implemented the Kuhn-Munkres algorithm (KMA), an algorithm that has been proven to provide the exact answer (i.e. minimal total distance travelled) in the shortest time possible. After many experiments, we came to the conclusion that our heuristic is faster, but that the solution provided by the KMA is slightly better. In particular, although the difference in total distance travelled is small, the KMA reduces the numbers of flight paths crossing each other, which is an important metric in our next building block.

Once we developed algorithms to assign airborne positions to each UAV on the ground, we started developing algorithms to take off all those UAVs. The objective of these algorithms is to reduce the time it takes for all the UAVs to reach their aerial position, while ensuring that all UAVs maintain a safe distance. The easiest solution is a sequential take-off procedure, but this is also the slowest approach. Hence, we improved it by first proposing a semi-sequential and later a semi-simultaneous take-off procedure. With this semi-simultaneous take-off procedure, we are able to reduce the takeoff time drastically without introducing any risk to the aircraft.

After taking off, it is important that all the UAVs can move from one point to another while maintaining a stable formation. If one drone would only fly a little bit faster than the others, over time the formation could warp. Hence, we created a protocol called Mission-Based UAV Swarm Coordination Protocol (MUSCOP). It is based on a master-slave pattern, where the master is sending messages to the slaves so that they know where to go, and the slaves are sending their status to the master. MUSCOP enforces all the UAVs to wait and resynchronize at each waypoint. To make the protocol more resilient, we included mechanisms so that the master is replaced in case it fails (e.g. depleted battery, collision, etc.). A similar mechanism was created for the slaves, so that, in case a slave fails, the rest of the swarm can still continue their mission. As a result, our protocol allows a swarm to complete its mission while ensuring that aircraft stay synchronized even in harsh environments.

Besides these basic buildings blocks that are needed in every UAV swarm flight, we also worked on some more advanced features. We developed an algorithm so that UAVs can change their formation (e.g. circle, line, matrix, etc.) during the flight. The algorithm decides first which UAV has to go to which place, and then,

based on the direction the UAV will go, the altitude of the UAV is changed. By using this algorithm, the UAVs that are flying in different directions will fly at a different altitude. Hence, collisions can be avoided. Furthermore, we created an algorithm that automatically changes the altitude of a UAV so that the altitude with respect to ground level remains constant. This is an important detail in mountainous regions, where ground levels can change rapidly, which may lead to crashes. Our algorithm uses Digital elevation models (DEMs) and a PID controller to efficiently control the flight altitude.

Finally, at the end of the flight, the UAVs have to accurately land at a specific location. The accuracy obtained by Global navigation satellite system (GNSS) is often not sufficient, since it can have an error up to 5 meters. Hence, we created an algorithm that, with the use of a simple camera, detects an ArUco marker which is placed on the exact landing place that is targeted. While descending, the UAV is steered towards the ArUco marker. Using this algorithm, we could decrease the error margin to an average of 11 cm.

Overall, this PhD thesis offers a complete framework for developing UAV applications, and represents a meaningful contribution to the research community. We hope that this contribution will be used (by other authors) in the years to come.

Resumen

Los Vehículos Aéreos no Tripulados (o drones) ya han demostrado su utilidad en una gran variedad de aplicaciones. Hoy en día, se utilizan para fotografía, cinematografía, inspecciones y vigilancia, entre otros. Sin embargo, en la mayoría de los casos todavía son controlados por un piloto, que como máximo suele estar volando un solo dron cada vez. En esta tesis, tratamos de avanzar en paso más allá en esta tecnología al permitir que múltiples drones con capacidad para despegue y aterrizaje vertical trabajen de forma sincronizada, como una sola entidad. La principal ventaja de realizar vuelos en grupo, comúnmente denominado enjambre, es que se pueden realizar tareas más complejas que utilizando un solo dron. De hecho, un enjambre permite cubrir más área en el mismo tiempo, ser más resistente, tener una capacidad de carga más alta, etc. Esto puede habilitar el uso de nuevas aplicaciones, o una mejor eficiencia para las aplicaciones existentes. Sin embargo, una parte clave es que los miembros del enjambre deben organizarse correctamente, ya que, durante el vuelo, diferentes perturbaciones pueden provocar que sea complicado mantener el enjambre como una unidad coherente. Una vez que se pierde esta coherencia, todos los beneficios previamente mencionados de un enjambre se pierden también. Incluso, aumenta el riesgo de colisiones entre los elementos del enjambre. Por lo tanto, esta tesis se centra en resolver algunos de estos problemas, proporcionando un conjunto de algoritmos que permitan a otros desarrolladores crear aplicaciones de enjambres de drones.

Para desarrollar los algoritmos propuestos hemos incorporado mejoras al llamado ArduSim. Este simulador nos permite simular tanto la física de un dron como la comunicación entre drones con un alto grado de precisión. ArduSim nos permite implementar protocolos y algoritmos (bien probados) en drones reales con facilidad. Durante toda la tesis, ArduSim ha sido utilizado ampliamente. Su utilización ha

permitido que las pruebas fueran seguras, y al mismo tiempo nos permitió ahorrar mucho tiempo, dinero y esfuerzo de investigación.

Comenzamos nuestra investigación sobre enjambres asignando posiciones aéreas para cada dron en el suelo. Suponiendo que los drones están ubicados aleatoriamente en el suelo, y que necesitan alcanzar una formación aérea deseada, buscamos una solución que minimice la distancia total recorrida por todos los drones. Para ello se empezó con un método de fuerza bruta, pero rápidamente nos dimos cuenta de que, dada su alta complejidad, este método funciona mal cuando el número de drones aumenta. Por lo tanto, propusimos una heurística. Como en todas las heurísticas, se realizó un compromiso entre complejidad y precisión. Al simplificar el problema, encontramos que nuestra heurística era capaz de calcular una solución muy rápidamente sin aumentar sustancialmente la distancia total recorrida. Además, implementamos el algoritmo de Kuhn-Munkres (KMA), un algoritmo que ha demostrado proporcionar la respuesta exacta (es decir, reducir la distancia total recorrida) en el menor tiempo posible. Después de muchos experimentos, llegamos a la conclusión de que nuestra heurística es más rápida, pero que la solución proporcionada por el KMA es ligeramente más eficiente. En particular, aunque la diferencia en la distancia total recorrida es pequeña, el uso de KMA reduce el número de trayectorias de vuelo que se cruzan entre sí, lo cual es una métrica importante para las siguientes propuestas.

Una vez que desarrollamos los algoritmos para asignar posiciones aéreas a cada dron en el suelo, comenzamos a desarrollar algoritmos para despegar todos esos drones. El objetivo de estos algoritmos es reducir el tiempo necesario para que todos los drones alcancen su posición aérea, al mismo tiempo que garantizan que se mantenga una distancia segura entre ellos. La solución más sencilla es un procedimiento de despegue secuencial, pero este enfoque también es el más lento. Por lo tanto, lo mejoramos proponiendo primero un procedimiento de despegue semi-secuencial, y a continuación uno semi-simultáneo. Con este procedimiento semi-simultáneo de despegue, el algoritmo es capaz de reducir drásticamente el tiempo de despegue sin introducir ningún riesgo para las aeronaves.

Después de despegar, es importante que todos los drones puedan moverse de un punto a otro mientras mantienen una formación estable. Si un dron se desplaza un poco más rápido que los demás, con el tiempo, la formación podría romperse. Por lo tanto, diseñamos y desarrollamos un protocolo llamado *Mission-Based UAV Swarm Coordination Protocol (MUSCOP)*. Está basado en un esquema maestro-esclavo, donde el maestro envía mensajes a los esclavos para que sepan a dónde ir, y los esclavos envían su estado al maestro (el cual actua como coordinador). MUSCOP obliga a todos los drones a esperar y resincronizarse en cada punto de referencia. Para hacer el protocolo más resistente, incluimos mecanismos para reemplazar al maestro en caso de fallo (por ejemplo, batería agotada, colisión, etc.). Se desarrolló

un mecanismo similar para los esclavos, para que, en caso de que falle un esclavo, el resto del enjambre aún pueda continuar su misión. Como resultado, nuestro protocolo permite que un enjambre complete su misión, asegurándose de que los aviones se mantengan sincronizados, incluso en entornos complejos.

Además de estos algoritmos, que son necesarios en cada vuelo de enjambre de drones, también centramos en características más avanzadas. Concretamente, desarrollamos un algoritmo para que los drones puedan cambiar su formación (por ejemplo, círculo, línea, matriz, etc.) durante el vuelo. El algoritmo decide primero qué dron tiene que ir a qué lugar, y luego, en función de la dirección a la que se dirigirá el dron, se cambia la altitud de la aeronave. Al usar este algoritmo, los drones que vuelan en diferentes direcciones volarán a diferentes altitudes. Por lo tanto, se minimiza el riesgo de colisiones.

Otra de las aportaciones realizadas consiste en un algoritmo que cambia automáticamente la altitud de un dron para que la elevación con respecto al nivel del suelo permanezca constante. Este es un detalle importante en regiones montañosas, donde la orografía del terreno puede ser muy variable, lo que puede provocar accidentes. Nuestro algoritmo utiliza modelos digitales de elevaciones y un controlador PID para controlar eficientemente la altitud de vuelo.

Finalmente, al final del vuelo, los drones tienen que aterrizar con precisión en un lugar específico. La precisión obtenida con el sistema Global de Navegación por Satélite (GNSS) a menudo no es suficiente, ya que puede tener un error de hasta 5 metros. Por lo tanto, creamos un algoritmo que, con el uso de una simple cámara, detecta un marcador ArUco que se coloca en el lugar exacto de aterrizaje que se está apuntando. Mientras desciende, el dron se dirige hacia el marcador ArUco. Usando este algoritmo, pudimos reducir el margen de error a un promedio de solo 11 cm.

En general, esta tesis de doctorado ofrece un marco completo para el desarrollo de aplicaciones de drones, y representa una contribución significativa a la comunidad de investigación que esperamos pueda ser aprovechada en investigaciones futuras.

Resum

Els vehicles aeris no tripulats (o drons) ja han demostrat la seua utilitat en una gran varietat d'aplicacions. Avui dia, s'utilitzen per a fotografia, cinematografia, inspeccions i vigilància, entre altres. No obstant això, en la majoria dels casos encara són controlats per un pilot, que com a màxim sol controlar el vol d'un sol dron cada vegada. En aquesta tesi, tractem d'avançar un pas més enllà en aquesta tecnologia, en permetre que múltiples drons amb capacitat per a l'enlairament i l'aterratge vertical treballen de forma sincronitzada, com una sola entitat. El principal avantatge de realitzar vols en grup, comunament denominats eixam, és que es poden fer tasques més complexes que utilitzant un sol dron. De fet, un eixam permet cobrir més àrea en el mateix temps, ser més resistent, tenir una capacitat de càrrega més alta, etc. Això pot habilitar l'ús de noves aplicacions, o una millor eficiència per a les aplicacions existents. No obstant això, una punt clau és que els membres de l'eixam han d'organitzar-se correctament, ja que, durant el vol, diferents pertorbacions poden provocar que siga complicat mantenir l'eixam com una unitat coherent. Una vegada que es perd aquesta coherència, tots els beneficis prèviament esmentats d'un eixam es perden també. Fins i tot, augmenta el risc de col·lisions entre els elements de l'eixam. Per tant, aquesta tesi se centra a resoldre alguns d'aquests problemes, proporcionant un conjunt d'algorismes que permeten a altres desenvolupadors crear aplicacions d'eixams de drons. Per a desenvolupar els algorismes proposats hem incorporat millores a l'anomenat ArduSim. Aquest simulador ens permet simular tant la física d'un dron com la comunicació entre drons amb un alt grau de precisió. ArduSim ens permet implementar protocols i algorismes (ben provats) en drons reals amb facilitat. Durant tota la tesi, ArduSim s'ha utilitzat àmpliament. El seu ús ha permès que les proves foren segures, i al mateix temps ens va permetre estalviar molt de temps,

diners i esforç d'investigació. Per tant, es va utilitzar ArduSim per a cada bloc de construcció que vam desenvolupar. Comencem la nostra recerca sobre eixams assignant posicions aèries per a cada dron en terra. Suposant que els drons estan situats aleatòriament en terra i que necessiten assolir la formació aèria desitjada, cerquem una solució que minimitze la distància total recorreguda per tots els drons. Per a això, es va començar amb un mètode de força bruta, però ràpidament ens vam adonar que, atesa l'alta complexitat, aquest mètode funciona malament quan el nombre de drons augmenta. Per tant, vam proposar una heurística. Com en totes les heurístiques, es va fer un compromís entre complexitat i precisió. En simplificar el problema, trobem que la nostra heurística era capaç de calcular una solució molt ràpidament sense augmentar substancialment la distància total recorreguda. A més, vam implementar l'algorisme de Kuhn-Munkres (KMA), un algorisme que ha demostrat proporcionar la resposta exacta (és a dir, reduir la distància total recorreguda) en el menor temps possible. Després de molts experiments, arribem a la conclusió que la nostra heurística és més ràpida, però que la solució proporcionada pel KMA és lleugerament més eficient. En particular, encara que la diferència en la distància total recorreguda és xicoteta, l'ús de KMA redueix el nombre de trajectòries de vol que s'encreuen entre si, la qual cosa és una mètrica important per a les propostes següents. Una vegada que desenvolupem els algorismes per a assignar posicions aèries a cada dron en terra, comencem a desenvolupar algorismes per a enlairar tots aquests drons. L'objectiu d'aquests algorismes és reduir el temps necessari perquè tots els drons assolisquen la seua posició aèria, alhora que garanteixen que es mantinga una distància segura entre aquests. La solució més senzilla és un procediment d'enlairament seqüencial, però aquest enfocament també és el més lent. Per tant, el millorem proposant primer un procediment d'enlairament semiseqüencial, i a continuació un semisimultani. Amb aquest procediment semisimultani d'enlairament, l'algorisme és capaç de reduir dràsticament el temps d'enlairament sense introduir cap risc per a les aeronaus.

Després d'enlairar-se, és important que tots els drons puguen moure's d'un punt a un altre mentre mantenen una formació estable. Si un dron es desplaça una mica més ràpid que els altres, amb el temps, la formació podria trencar-se. Per tant, dissenyem i desenvolupem un protocol anomenat MUSCOP. Està basat en un esquema mestre-esclau, on el mestre envia missatges als esclaus perquè sàpien a on anar, i els esclaus envien el seu estat al mestre (el qual actua com a coordinador). MUSCOP obliga a tots els drons a esperar i resincronitzar-se en cada punt de referència. Per a fer el protocol més resistent, incloem mecanismes per a reemplaçar el mestre en cas de fallada (per exemple: bateria esgotada, col·lisió, etc.). Es va desenvolupar un mecanisme similar per als esclaus, perquè, en cas que falle un esclau, la resta de l'eixam encara puga continuar la seua missió. Com a resultat, el nostre protocol permet que un eixam complete la seua missió, i s'assegura que

els avions es mantinguen sincronitzats, fins i tot en entorns complexos. A més d'aquests algorismes bàsics, que són necessaris en cada vol d'eixam de drons, també ens centrem en característiques més avançades. Concretament, desenvolupem un algorisme perquè els drons puguen canviar la seua formació (per exemple, cercle, línia, matriu, etc.) durant el vol. L'algorisme decideix primer quin dron ha d'anar a quin lloc, i després, segons la direcció a què es dirigirà el dron, es canvia l'altitud de l'aeronau. En usar aquest algorisme, els drons que volen en diferents direccions volaran a diferents altituds. Per tant, es minimitza el risc de col·lisions. Una altra de les aportacions realitzades consisteix en un algorisme que canvia automàticament l'altitud d'un dron perquè l'elevació respecte al nivell del sòl romanga constant. Aquest és un detall important en regions muntanyenques, on l'orografia del terreny pot ser molt variable, la qual cosa pot provocar accidents. El nostre algorisme utilitza models digitals d'elevacions i un controlador PID per a controlar eficientment l'altitud de vol. Finalment, al final del vol, els drons han d'aterrar amb precisió en un lloc específic. La precisió obtinguda amb el sistema global de navegació per satèl·lit (GNSS) sovint no és suficient, ja que pot tenir un error de fins a 5 m. Per tant, creem un algorisme que, amb l'ús d'una simple càmera, detecta un marcador ArUco que es col·loca al lloc exacte d'aterratge on s'apunta. Mentre descendeix, el dron es dirigeix cap al marcador ArUco. Usant aquest algorisme, vam poder reduir el marge d'error a una mitjana d'11 cm només. En general, aquesta tesi de doctorat ofereix un marc complet per al desenvolupament d'aplicacions de drons, i representa una contribució significativa a la comunitat d'investigació que esperem que podrà ser aprofitada en recerques futures.

Contents

Acknowledgements **vii**

Abstract **ix**

List of Figures **xxiii**

List of Tables **xxvii**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Objectives . . . 3
1.3 Structure of the thesis . . . 3

2 From single UAVs to an autonomous swarm: an overview **7**
2.1 The fundamentals of Unmanned Aerial Vehicles (UAVs) . . . 7
2.2 Why we need UAV swarms . . . 14
2.3 Current solutions for swarm coordination . . . 15

3 ArduSim: a multi-UAV simulator **19**
3.1 Original version of ArduSim . . . 20
3.2 Design and implementation . . . 22
3.3 UAV swarm formation . . . 25
3.4 UAV-to-UAV communications . . . 27
3.5 Summary . . . 33

4 Assigning airborne positions efficiently **35**
4.1 Overview 36
4.2 Experiments & results 39
4.3 Summary 47

5 Taking off **49**
5.1 Analysis of possible take-off strategies 49
5.2 Experiments & results 63
5.3 Summary 71

6 Maintaining the swarm coherent **75**
6.1 The original version of MUSCOP 76
6.2 Proposed resilience mechanism 77
6.3 Experiments & results 80
6.4 Summary 90

7 Advanced mid-flight maneuvers **91**
7.1 Swarm reconfiguration 91
7.2 Adjusting the altitude for changing terrain levels 99
7.3 Experiments & results 105
7.4 Summary 114

8 Accurate vision-based landing **115**
8.1 Implementation 116
8.2 Experiments & results 120
8.3 Summary 126

9 Conclusions, Future Work and Publications **129**
9.1 Conclusions 130
9.2 Future work 131
9.3 Publications 132

Acronyms **135**

Bibliography **139**

List of Figures

2.1 Essential components of a UAV 11

3.1 The proposed architecture of ArduSim. 24
3.2 Linear formation. 26
3.3 Circle formation. 26
3.4 Matrix formation. 27
3.5 Random formation. 27
3.6 Connection between ArduSim and OMNeT++. 29
3.7 Example of buildings generated in OMNeT++ based on OpenStreetMap data (Foios, a small town in Spain). 30
3.8 Message delay caused by the inter-process overhead in different scenarios. 31
3.9 End-to-end delay when varying the number of UAVs in the experiment, and the message rate per UAV. All UAVs act as both senders and receivers. 32
3.10 End-to-end delay when varying the number of UAVs in the experiment, and the message rate per UAV. Only one UAV acts as transmitter, and the rest merely act as receivers. 33

4.1 All possible solutions that exist with only four UAVs. 36
4.2 Average calculation time when considering all UAV formations and assignment algorithms. 40
4.3 Calculation time using the KMA algorithm for various UAV formations. 41
4.4 Calculation time using the heuristic algorithm for various UAV formations. 41
4.5 The total distance travelled by all UAVs using the heuristic and KMA for the Circle formation. 42

4.6 The total distance travelled by all UAVs using the heuristic and KMA for the linear formation. 42
4.7 The total distance travelled by all UAVs using the heuristic and KMA for the matrix formation. 43
4.8 A comparison of the heuristic vs. the KMA algorithm in terms of potential collisions when varying number of UAVs (Circle and Matrix formations). 44
4.9 A comparison of the heuristic vs. the KMA algorithm in terms of potential collisions when varying number of UAVs (Linear formation). 44
4.10 An example of the first irregular formation, where the minimal distance between UAVs is closer in the air than on the ground, and the centers of the formations are not aligned. 45
4.11 Example of an irregular ground pattern. 46

5.1 Example of a set of intermediate positions in the take-off flight path. . 52
5.2 Comparison between intermediate positions in the flight trajectories of two drones. 53
5.3 Overview of the CSTH+RSR approach. 55
5.4 Line segment corresponding to the minimal distance between two lines in a 3D space. 60
5.5 Flowchart representing the batch generation mechanism. 62
5.6 Example of take-off batch grouping (A, B) using our algorithm. 63
5.7 Illustration of the flight path to be taken for a UAV during take-off. . 64
5.8 Calculation time according to their granularity. 65
5.9 Number of potential collisions that would remain undetected when increasing the granularity value. 66
5.10 Undetected collisions according to interval size used. 66
5.11 Calculation time for the linear formation when varying the number of UAVs in the swarm. 67
5.12 Calculation time for the circular formation when varying the number of UAVs in the swarm. 68
5.13 Calculation time for the matrix formation when varying the number of UAVs in the swarm. 69
5.14 Calculation time for the circular formation. 69
5.15 Calculation time for the matrix formation. 70
5.16 Calculation time for the linear formation. 70
5.17 Take-off times for the matrix formation. 71
5.18 Take-off times for the circle formation. 72
5.19 Take-off times for the circle formation. 72

6.1 Example of a swarm splitup. 78
6.2 Message frequency w.r.t. distance between UAVs. 81
6.3 Chances of failing just before a waypoint w.r.t. the distance between waypoints. 83
6.4 Distribution of the UAV waiting times at waypoint 1. 86
6.5 Flight time and wait time overhead when varying the number of UAVs that fail. 87
6.6 Time differences for multiple groups at each waypoint. 89
6.7 Working example of a swarm split-up scenario in ArduSim. 89

7.1 Flowchart of the flight formation reconfiguration algorithm. 92
7.2 Transition of 9 UAVs from a linear formation to a compact mesh. . . . 95
7.3 Minimum number of sectors required for a collision-free reconfiguration procedure. 96
7.4 Minimum number of sectors required for a collision-free reconfiguration w.r.t. the type of transition. 97
7.5 Time spent moving horizontally (state 2) and vertically (states 1,3). . 97
7.6 Estimated time overhead vs. real time overhead. 98
7.7 The different types of altitude. 100
7.8 The difference between DTM and DSM, source [42] 102
7.9 Block diagram of the proposed method. 104
7.10 Two important metrics for the step response. 106
7.11 Analysis of the influence of parameters K_p and K_d on the step response of the UAV. 108
7.12 Step response with optimal parameters, i.e. $K_p = 1.5; K_d = 1.9$. 109
7.13 Results for scenario A. 111
7.14 Results of scenario B. 113

8.1 Two examples of ArUco markers of different sizes. 117
8.2 Image retrieved by the UAV camera after processing using OpenCV/ArUco libraries. 117
8.3 Visual representation of the virtual border. 119
8.4 Hexacopter used in our experiments. 122
8.5 UAV landing position comparison. 123
8.6 Landing offset GPS vs visual based approached. 123
8.7 Number of consecutive dropped camera frames using algorithm 8.1 . . 124
8.8 Displacement using Algorithm 8.1. 125
8.9 $beta_x$ and $beta_y$ angle variations vs. flight time. 125
8.10 Estimated X and Y variations associated to UAV positions during landing. 126
8.11 Number of consecutive dropped camera frames using algorithm 8.2. . . 126

8.12 Displacement using Algorithm 8.2. 127

List of Tables

2.1 Mandatory characteristics for each class identification label. 13

4.1 Comparison between KMA and Heurisitic for irregular air formations 46
4.2 Comparison between KMA and Heuristic for irregular ground patterns 47

6.1 Time overhead for the different scenarios at 200 m from the next waypoint. 84
6.2 Time overhead for the different scenarios at 15 m from the next waypoint. 84
6.3 Time overhead for the different scenarios just when reaching the next waypoint (0 m). 84

7.1 Collisions and minimum distance analysis. 95
7.2 Time UAVs spend in each state. 96
7.3 Influence of the look-ahead distance for scenario A. 110
7.4 Rural flight: influence of adjusting the altitude on flight time and energy. 110
7.5 Influence of the look-ahead distance for scenario B. 112
7.6 Mountain flight: influence of adjusting the altitude on flight time and energy. 112

8.1 Parameter values adopted regarding Algorithms 8.1 and 8.2. 119
8.2 Speed values adopted regarding Algorithm 8.2. 120
8.3 Comparative table of the different schemes. 127

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAVs), more commonly known as drones, have been commercially available since 2010. Hence, they are perceived as a recent invention. In reality, however, nothing is farther from the truth. Although this thesis is focussed on how modern UAVs can be together in a swarm, it is worth starting with a little bit of history. Going back, a long, long time ago, the Greek myth of Daedalus and Icarus tells us that humans always had an aspiration to fly, and that flying inherently comes with great danger. This myth is most likely not the first time that people thought about flying, and most certainly not the last. Yet, let us not linger too long in the ancient past, and let us start when flying actually began. The reader would probably think of the Wright brothers, flying for the first time in December 1903, but this would be wrong by almost 1700 years. The first unmanned "aircraft" was a simple sky lantern used by the Chinese for military signaling, a prominent application as we will soon discover. It would, however, take until 1783 before the French brothers Joseph-Michel and Jacques-Étienne Montgolfier improved on the idea so that humans could finally ascend into the air. After only a hundred years, the hot-air balloons invented by the Montgolfier brothers were already in use for exactly the same purposes as nowadays' modern versions: warfare and photography; and in many cases, the pictures taken were used for war intelligence. Although from a humanitarian point of view, this is awful, there was, and there still is, a big market for it. We must acknowledge that, (only) due to that funding, many inventors had the possibility to work and

improve the wonderful world of aviation. After many UAVs carrying bombs and surveilling enemy territory, we arrived at UAVs for consumer use. The French drone manufacturer Parrot was the first in 2010, and a few years later DJI entered the same market. To this day, these are still the two most famous drone manufacturers worldwide. They created the drones that you and I came to know. Namely, a relatively small, lightweight quadcopter, often equipped with a camera, that can be controlled from a smartphone or a remote. At that time, they were mostly seen as toys for children, or adults (which we kindly refer to as hobbyist). Within the group of hobbyists, there must have been a few with an entrepreneurial mindset, because soon after some industries realized that they could use these “toys” as a valuable asset for their business. In many cases, drones could perform a certain task faster, safer, and cheaper compared to traditional methods. This led to a rather slow adoption of UAVs in our lives. Nowadays, it is quite common for a real estate agent to take pictures of a house using a drone, a bridge to be inspected by a quadcopter, and actions shots in movies to be taken by a UAV. Less common, but still seen, are drones delivering parcels or aiding farmers. Due to this increase of flying objects, rules had to be formulated to prevent injuries and misuse. Nowadays, there are laws about where UAVs can fly, and who can fly them. Given that the industry is still relatively young, the law will likely be updated many times while the industry matures. It is, indeed, an industry that is still developing; and I would like to believe that this thesis is helping that process.

1.1 Motivation

While currently most drone applications rely on the deployment of a single UAV, interest is growing in solutions where multiple UAVs are simultaneously deployed to perform a joint task [1], thereby conforming a UAV swarm. Swarms of UAVs allow us to efficiently perform complex applications that cannot be handled by a single UAV [2]. It also allows a larger area to be covered in the same amount of time. This is an important aspect since current battery life impedes most UAVs to fly longer than 25 minutes. Besides that, they also bring additional benefits. For instance, applications can be more resilient (important in surveillance), and the load capacity can be increased by distributing such load among multiple UAVs. Although the benefits are promising, many challenges also exist, including: (i) swarm formation definition, (ii) takeoff procedure, (iii) in-flight coordination, (iv) handling the loss of swarm elements, (v) communication between the swarm elements, (vi) swarm layout reconfiguration, and (vii) accurate landing. These challenges remain mostly untackled by the research community, which delays the widespread adoption of UAV swarms by the industries. Hence, in this thesis, we seek to provide solutions

to the aforementioned challenges. We will explain what will hopefully be common practice in a decade or two, why we want to use multiple UAVs simultaneously, and how this can improve our lives. We will show how a seemingly simple idea can actually be quite complicated. Luckily, at least for a few of these complicated problems, we have found a solution!

1.2 Objectives

The main objective of this PhD thesis is to propose a baseline framework upon which UAV swarm applications can be developed. Such framework enables the application developer to seamlessly and safely operate a UAV swarm, encompassing all the required features associated to the different flight phases.

To achieve this overall goal, several sub-objectives must be defined. These are the following:

- Create a simulation environment that allows developing new solutions for UAVs quickly and efficiently, while guaranteeing that the developed code can be directly ported to real aircraft.
- Devise an algorithm to optimize the assignment of UAVs on the ground to their aerial position in a flight formation.
- Propose an efficient take-off strategy that offers a minimal take-off time, while reducing the risk of aircraft collisions during that process.
- Develop a protocol that is able to maintain a stable and coherent UAV flight formation, while adapting itself to issues like the loss of UAVs, including both formation leader and slave elements.
- Propose a novel solution to enable changing the flight formation throughout a mission in a fast and safe manner.
- Develop a control mechanism that allows adjusting the flight altitude of each individual UAV to account for terrains with irregular profiles.
- Devise a solution for the accurate landing of each individual UAV based on ground markers and vision-based control systems.

1.3 Structure of the thesis

The thesis dissertation is organized in nine chapters. Next, we briefly describe the contents of each part:

In **chapter 2. From single UAVs to an autonomous swarm: an overview:** we start by explaining the fundamentals of UAVs. We classify UAVs by size, type, etc., explain the different parts that make up a UAV, and briefly go over some European legislation. Afterward, we explain why we are interested in UAV swarms, the advantages they have over single UAVs, highlighting which challenges need to be solved. Then, we provide some related works for these different challenges.

In **chapter 3. ArduSim: a multi-UAV simulator:** we introduce our multi-UAV simulator called ArduSim. We explain its current structure, how it can be used, test its performance, and explain its limitations.

In **chapter 4. Assigning airborne positions efficiently** we start with our first swarm algorithm. We explain how UAVs that are randomly placed on the ground can be assigned to a specific airborne location in a formation. We introduce four different algorithms for solving this problem namely: brute-force, heuristic, KMA, and KMA for GPU. These algorithms are then compared to each other in various scenarios using ArduSim.

After assigning the airborne positions, we explain various take-off procedures in **chapter 5. Taking off**. We start with a simple sequential take-off procedure, and improve it to a semi-sequential procedure. Then, we improve it further and present a semi-simultaneous take-off procedure. We compare the three different approaches in terms of calculation time, and take-off time.

We continue, with **chapter 6. Maintaining the swarm coherent**. We detail our Mission-Based UAV Swarm Coordination Protocol (MUSCOP), which ensures that a swarm of UAVs follows a mission while maintaining a stable formation. After explaining the basics of this protocol (which is based on the master-slave pattern), we introduce extra mechanisms that improve resilience. These mechanisms make sure that, in case something goes wrong with the master or slave (e.g. depleted battery, crash, etc.), the swarm can continue the mission.

In **chapter 7. Advanced mid-flight maneuvers**, we explain two new algorithms that enhance swarm flights. First, we introduce a way to change the swarm formation mid-air. This algorithm temporally changes the altitude of some of the UAVs, so that they can move to their new location while avoiding collisions. Secondly, we introduce an algorithm that dynamically adapts the altitude of the UAVs in order to maintain a stable altitude above the ground. This is especially important in mountainous regions, where a quick change of terrain level might result in a crash.

At the end of their mission, the UAVs will need to land; in **chapter 8. Accurate vision-based landing**, we explain how this can be done accurately using a simple camera and an ArUco marker.

Finally, in **chapter 9. Conclusions, Future Work and Publications**, we finish this dissertation by detailing the main conclusions of our work, presenting the various publications made, and going over some future work.

Chapter 2

From single UAVs to an autonomous swarm: an overview

In this chapter, we start our journey towards autonomous UAV swarms. In order to finally arrive at the contributions of this thesis, we must first explain the fundamentals of UAVs. Hence, we will explain the different types of UAVs, the basic components of each UAV, and the current European legislation regarding using them. Afterward, we go into depth about why we actually need a UAV swarm, detailing what advantages they can bring, but also the challenges that we must face. Finally, we will present current solutions for those challenges created by other authors.

2.1 The fundamentals of Unmanned Aerial Vehicles (UAVs)

The title of this thesis clearly indicates that the scope of this work will embrace UAVs. Nevertheless, the term UAV is slightly ambiguous; all it really tells us is that we are using unmanned aerial vehicles. It tells us nothing about the size, the speed, the weight, the capabilities, the price, etc. It is, however, of utmost importance to specify which UAVs are used in this thesis, because we only provide solutions for a specific type of UAVs. Hence, in this section, we will provide the fundamentals of UAVs. Let us start with a common misconception; often (including us, in this thesis) the term UAV and drone are used interchangeably, and technically this is

wrong. The word drone refers to all different types of unmanned robots, including those on land, water, and air, whereas UAV (obviously) only refer to flying robots. Nevertheless, in common speech, the two are synonyms; consequently, we will use them as such.

2.1.1 UAV types

In the world of UAVs there are many categorizations to be made. The biggest distinction, however, is the difference between fixed-wing aircraft and copters. Although both can serve for similar purposes, the way they are built, operated, and used is completely different. Copters have the major advantage that they can take off vertically and hover. This makes them ideal for inspection work, parcel delivery, etc. However, fixed-wing aircraft are faster and more efficient. Hence, they are used more frequently for longer flights surveying a large area. Although hybrid versions do exist, they are not so common due to their complexity. Since our work is based on the copter variation, we will only discuss their structure and configuration. Within the copter variation, there are two types: helicopters, and multicopters. Helicopters are copters with only one horizontal rotor. This rotor is connected to the engine by a shaft. As the engine turns the rotor, the rotor blades spin and generate lift, which is what keeps the helicopter in the air. The rotor can also move, which will change the tilt of the propeller, and as a consequence the helicopter can move horizontally.

Multicopters, however, work based on an entirely different principle. They have multiple rotors which can be controlled individually. Depending on the relative speed of these rotors, the multicopter can change its pitch, roll, and yaw. The exact flight dynamics depend on the number of rotors, which brings us to the first way of categorizing multicopters, i.e. by the number of rotors. The naming convention follows Greek numerical prefixes e.g. tricopter, quadcopter, pentacopter, etc. The minimum number of rotors is three, but this variation is quite rare. The most common version is the quadcopter, followed by the hexacopter. In general, more motors will generate more lift, and thus allow the copter to carry more weight. It also improves the safety as the drone is still able to fly when an engine fails, and, in general, it improves the stability. However, it also makes the drone less agile, and most importantly, more expensive.

Another common way to categorize drones is by their size and/or weight. In most cases, size is a good indicator for their usage, cost, payload, and endurance. The smallest UAVs are called nano UAVs, which are mainly used for indoor flights. Since they tend to be cheap (≈ €100 or less), they are also used for training purposes. Normally, they weight less than 100 grams, and have a wingspan of less than 30 centimeters. Since they are so small and lightweight, their endurance is

generally limited to only a few minutes. Next are the Micro Aerial Vehicles (MAVs); they weigh less than two kilograms, cost around 1000€, and can fly for about 30 minutes. They are powerful enough to fly outdoors under good weather conditions (i.e. wind speed < 30 km/h, and no rain). This makes them very popular for hobbyists, photographers, and researchers. In this thesis, we make use of this type of drone. Beyond this point, UAVs get larger, heavier, faster, and more expensive. It is important to note that, depending on the source, the exact characteristics and naming for each group can change slightly. A formal categorization of (all types of) UAVs is made by the European Union Aviation Safety Agency (EASA). However, since their main concern is about safety, they only take the weight and the proximity of people into consideration. This will be explained in more detail in subsection 2.1.3.

Finally, UAVs can also be categorized by altitude and range. In this thesis we work exclusively with drones that are allowed to fly up to 120 meters. However, in some applications (mostly military purposes), UAVs are flying at much higher altitudes. In most cases, these types of UAVs are fixed-wing aircraft that are using a fuel-cell for energy. They are categorized as follows:

- High Altitude, Large Endurance (HALE) UAVs are the most expensive (several million euros) unmanned aircraft. They fly up to 30 hours at an altitude of 20 km, and have a high payload capacity. Due to their characteristics, they are used in surveillance, and reconnaissance missions.

- Medium Altitude, Large Endurance (MALE) UAVs fly at a medium altitude of 9 km, and are also able to fly for a long time. Although the cost of MALE UAV can vary greatly, they are typically priced anywhere from several hundred thousand euros up to a million.

- Low Altitude, Large Endurance (LALE) UAVs fly closer to the ground, but still maintain an altitude of a few kilometers, which is similar to most commercial flights. Due to their large endurance and payload capacity of several kilograms, they are used frequently for forest inventory, and monitoring, among others.

- Low Altitude, Short Endurance (LASE) UAVs are similar to LALE UAVs, but their endurance is limited to less than 2 hours.

- MAV UAVs are the most common UAVs. They weigh less than 2 kg, and usually cost less than 1.000 euros. They typical use a battery pack as power source and, therefore, the flight time is limited to about 30 minutes. They fly at a low altitude (below 120 meters), and are therefore used frequently

for photography, inspections, etc. In this thesis, we specifically address this type of UAV.

2.1.2 UAV structure & configuration

There is a wide range of UAVs available, including commercial off-the-shelf models, as well as custom-built designs. Despite this great variety of UAV designs, all of them require a few basic components. In Figure 2.1, we show an illustration of these components, and how they are connected. Each UAV has a few motors, which provides the necessary power to generate lift and move the UAV. Coupled with the motor is the Electronic Speed Controller (ESC), which controls the speed of the motors. The ESCs receives a signal (indicating the rotation speed) from the flight controller, which serves as the brain of the UAV. It interprets data from various sensors, and uses that data to control the ESCs. Most UAVs are equipped with a Global Positioning System (GPS) and a compass module, which provide critical location data. This data is used by the flight controller to maintain the UAV stable, but is also often used for autonomous flights. All of these electrical devices need power, and this is most commonly provided by a Li-Po battery. The power module helps regulate and distribute the power from the battery to the various components. Finally, a pilot can operate the UAV using a remote control kit, which sends wireless radio signals (in the 2.4 GHz band) to the flight controller. Although this communication is bidirectional, often a telemetry kit is also included. The telemetry channel is used to send useful data such as flight mode, location, speed, etc. to a smartphone or Ground Station Control (GSC). In Europe, this transmitter is sending a signal at 443 MHz, which allows for long-range communications. Finally, there are some LEDs and a buzzer which indicate what the drone is doing.

The flight controller is a microprocessor that controls the UAV. Since the performance of the controller influences the flight experience greatly, most commercial UAVs develop their own proprietary flight controller. Although their performance is usually excellent, interaction with the flight controller remains limited. Hence, for research purposes, proprietary flight controllers are not a good option. Fortunately, there are several open-source alternatives. Most of these projects are using general purpose microcontrollers such as Arduino, STM32, Teensy, etc. These boards tend to be cheap, but since they are general-purpose microcontrollers, they are not optimized for controlling UAVs. They lack common connectors and inbuilt sensors, which makes using them more complicated. Despite these complications, a student (Lorenz Meier) from ETH Zurich started building a flight controller in 2008. Due to the enormous scale of this project, he asked other students to help as well. After working for almost a year, the team (called Pixhawk [3]) won the European Micro Air Vehicle competition in the indoor autonomy category. Soon

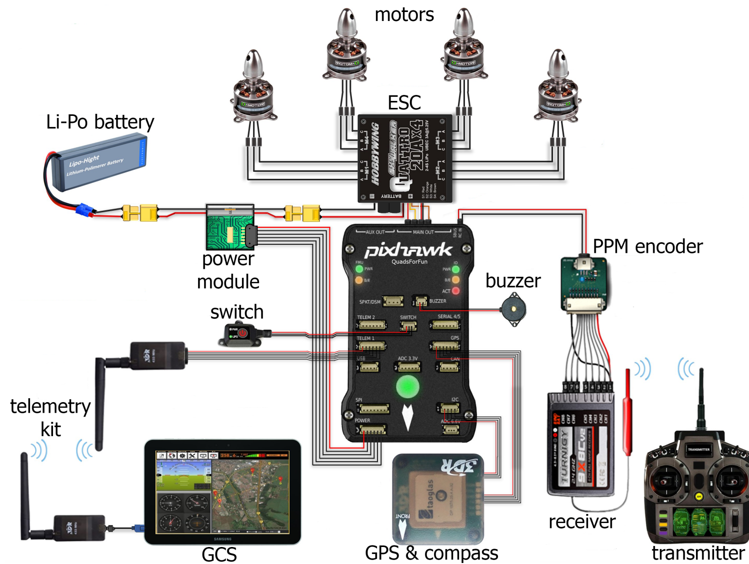


Figure 2.1: Essential components of a UAV

after, the Pixhawk flight controller began to be adopted by others as well. Nowadays, it is by far the most commonly used open-source flight controller. The actual hardware is produced and sold by the UAV hardware manufacturer Holybro[4]. They sell various versions, with different designs and specifications. The cheapest versions cost around €80, and the more expensive versions go up to €350. This is, however, only the hardware part. As with any microcontroller, it needs code to run. The Pixhawk community also provides the firmware that runs on their flight controller. Over the years, there were many updates, but the current version of their (also open-source) software is called PX4[5]. However, since the hardware is completely open-source, other groups also created firmware that is compatible with the Pixhawk. The most popular alternative to PX4 is called ArduPilot[6]. They started around the same time as the Pixhawk group, and initially used an Arduino board. Over the years, many changes were made and, in 2013, they decided to use the Pixhawk as their main hardware. Nowadays, their firmware is able to control all different types of drones including: rovers, boats, submarines, and UAVs. Their specific firmware for controlling MAV UAV is called ArduCopter. Currently the ArduCopter firmware is arguably better than the PX4. It can run on more hardware platforms than the PX4, and is able to maintain a UAV hovering with more stability. In this thesis, we work with the ArduCopter software for the

above-mentioned reasons, but also because ArduCopter is part of the ArduPilot project. This means that integration of our VTOL UAV protocols with other vehicles (planes, rovers, etc.) would be easier in the future. Besides firmware, both PX4 and ArduPilot created software to communicate with the UAV using a GSC. The version from ArduPilot is called Mission Planner, and from PX4 it is called QGroundControl. In order to work interchangeably, and allow other programs to easily communicate with the flight controller as well, the MAVLink[7] protocol was created (also by Lorenz Meier). It is a very lightweight messaging protocol which follows a modern hybrid publish-subscribe and point-to-point design paradigm. It is versatile, efficient, robust, and there are MAVLink libraries for the most common programming languages. Hence, almost all programs and flight controllers support MAVLink.

2.1.3 European legislation

Due to the increasing popularity of UAVs, the possibility of danger, and privacy issues that are related with UAVs, the EU set out a framework for the safe operation of civil drones. This section is written with the intention to give an overview of the current legislation, but does not serve as legal advice.

The specific legislation are explained in EU Regulations 2019/947 [8] and 2019/945 [9], which have been created by the EASA [10]. It is applicable since 31 December 2020 in all EU Member States, including Norway and Liechtenstein. It applies to civil drone operations, and is mainly concerned with managing risk. For this reason, it defines three categories: the open, the specific, and the certified category. The open category addresses low risk drone operations in which the drone operator has to ensure proper safety levels. The category is subdivided in three subcategories based on the proximity of people.

- A1: fly over people, but not over assemblies of people.
- A2: fly close to people.
- A3: fly far from people.

Besides the proximity of people, the characteristics of the UAV itself is also important. Depending on these characteristics, each drone will get a class identification label. It is important to note that currently (2023) we are in a transition period, and that formally the class identification labels will be used starting from January 1st, 2024. The classification is made by verifying if a drone provides the elements presented in Table 2.1. Keep in mind that this table is just a small summary, and that the exact specifications are described in [11].

Table 2.1: Mandatory characteristics for each class identification label.

	C0	C1	C2	C3	C4	C5	C6
Max weight <250 g	x						
Max weight <900 g		x					
Max weight <4 kg			x				
Max weight <25 kg				x	x	x	x
Low speed mode (<3 m/s)			x				
Low speed mode (<5 m/s), unless tethered						x	
Indication of noise emission		x	x	x		x	x
Remote id function		x	x	x		x	x
Geo-awareness function		x	x	x			
Low-battery warning		x	x	x		x	x
Flight termination system, unless tethered						x	x
Geo-caging function							x
Information position, speed, and altitude						x	x

Depending on the combination of subcategories A1, A2, A3, and the class identification labels, the pilot must fulfill certain requirements. These requirements can include a minimum age, proof of completion of a theoretical exam, and proof of completion of a practical exam. Besides the requirements, depending on the combination, there are also some limitations which might include keeping a certain distance from people or buildings. Many combinations are possible, hence we do not go into all of them. However, in most cases, the pilot needs to be older than 16, and pass both a theoretical and a practical exam. Furthermore, there are strict rules about where a UAV can be flown. In the open category, a UAV is never allowed to fly higher than 120 meters above ground level. Flying close to an airport, military, and/or governmental facilities is also prohibited. Many cities also prohibit flights above crowed areas without special authorization. For that reason, it is always important that the pilot takes note of the no-fly zones in the region.

Besides the open category, there is also the specific category which includes flights with a moderate risk such as beyond visual line of sight or UAVs above 25 kg. If the risk is even higher, a flight might fall into the certified category. This category is for the highest level of risk, and includes for instance a future drone flight with passengers on board. Since in this thesis we focus on flights in the open category, we will not go into the details of the specific or the certified categories.

To conclude, the regulation is somewhat complex, but that is mainly because there are many different types of UAVs. Once the details (i.e. weight, capacities,

etc.) of a specific UAV are known, it is relatively easy to find the class identification label. From there on, the specific rules about what a pilot is allowed to do, can be found easily. One major advantage is that the rules are consistent throughout the entire European Union. Besides that, as stated before, most pilots need a license which ensures that UAVs are used in a safe and responsible manner.

2.2 Why we need UAV swarms

In this thesis, we provide various algorithms and protocols which enable us to work with a swarm of UAVs. Before proceeding, we need to explain why we want to use UAV swarms in the first place. After all, there are already many applications with just a single UAV, and it seems that they are doing their job just fine.

Nowadays, UAVs are still most commonly used for photography, and for that application there is no direct need for a swarm. However, new UAV applications arise every day, and some of them are facing the limits of what a single UAV can do. One of the major issues of using a single UAV is its high energy consumption and limited battery capacity. Since a VTOL MAV UAV does not have any wings, it will always have to rely on powerful engines to maintain it airborne. Hence, flights for a single UAV are limited to just 20 or 30 minutes. For many applications, such as precision agriculture, search and rescue missions, surveillance, environmental monitoring, this is not enough. Hence, the most straight-forward solution is to use multiple UAVs at the same time. Note that this does not represent a real swarm, but just multiple aircraft sharing the same space. This can still be very useful, though, as evidenced by the popular light shows that are held for major events. Nevertheless, since most of the UAVs already have the capability to communicate, we can leverage UAV communications to achieve goals in a cooperative fashion. Only then can we speak of a swarm; notice that, besides covering a larger area, they provide many more benefits. First of all, they provide redundancy and robustness. If one UAV fails, the other UAVs in the swarm can take over its place and complete the mission. This can be very useful, especially in applications such as surveillance where drones might experience sabotages. Secondly, a swarm (when managed correctly) is more flexible, it can change its flight formation, and dynamically adapt itself to the needs of each application. For instance, in a search and rescue mission, the emergency services first need to find the person in need of help. To that end, a large area needs to be covered, and thus flying in a linear formation makes the most sense. Yet, upon finding the person of interest, the emergency services might want to obtain more data about the surroundings. Hence, the swarm could adapt and reconfigure itself in a circle. Furthermore, having multiple UAVs increases the payload capacity, allowing to help carry heavy first aid equipment to the person

in need. Overall, all of this can be done in a significantly cheaper manner when compared to traditional solutions.

2.3 Current solutions for swarm coordination

As stated above, a swarm of UAVs can provide many benefits. However, as is often the case, these advantages do not come for free, and significant challenges need to be solved before a swarm of UAVs can be used safely and efficiently. In this thesis, we provide solutions for some challenges, but we start by introducing related works by other authors.

First and foremost, we need a safe environment to test our solutions. Hence, a simulator is needed. When looking for UAV flight simulators, one will likely come across an abundance of flight simulators such as: UAV CRAFT [12], squadrone [13], and Quantum3d [14]. These types of simulators are created to train new pilots, and they usually include some type of 3D rendering, and even specific hardware for realistic emulation. Although very useful in their own right, these proprietary software products are not designed for doing research, nor do they allow the user to easily adapt the code, and thus are of no use for use. Nevertheless, there are a few simulators which are slightly better suited. The popular simulation tool called Simulink has a product called UAV Toolbox [15]. This tool incorporates a lot of features. It includes a scenario simulator that allows the user to (i) visualize (3D) a flight, and (ii) incorporate sensor models and generate synthetic data during a flight. Given that this toolbox is incorporated in Simulink, a license is necessary, but the developers are able to change a lot of the models and incorporate it with other Matlab projects that already exist. Overall, this toolbox is definitely designed for researchers, has an excellent documentation (including examples), and packs a lot of features. However, it can only simulate one UAV at the time. Another relatively new simulator is called AirSim [16]. It was developed by Microsoft research, and it specifically focuses on research towards Artificial Intelligence (AI). Besides UAVs, it is also a car simulator, and it was built upon the Unreal Engine, being available for Linux, Windows, and Mac, and having a good documentation. Since it was built for AI research, the API is equipped with many functions for image processing and AI models. This simulator allows for multiple UAVs to be simulated at the same time. There is also support for software- and hardware-in-the-loop testing. Unfortunately, this repository is currently archived. A new project (Microsoft Project AirSim [17]) is under development and will be available for public preview in the coming year (2024).

The next challenge is the assignment problem, which in the context of UAVs swarms has not been studied by other authors. However, in a more general context,

it has been studied by mathematicians. Currently, there are three options to solve this problem: Global methods, local methods, and (meta)heuristics. A global method promises to find the optimal solution. Currently, the KMA is the fastest option available. In some specific cases, local methods can work better; however this is not guaranteed. Those algorithms, called auction algorithms, are used for example by Causa and Fasano [18] who worked on a strategic path planning for multi-UAVs, and by the authors of [19], who presented a dynamic task and resource assignment algorithm. The last option is to use a metaheuristic. This will provide a suboptimal solution (in terms of reducing cost), but it is typically much faster than the previous solutions.

With regard to the specific issue of achieving a safe take-off of a large number of UAVs participating in a swarm, we find only one work (other than our own) that addresses this topic. In [20] authors consider three simple take-off options for a swarm: manual, sequential, and simultaneous. However, when the swarm is large, and when the formation in the air remains unrelated to positions on the ground, these techniques can take too much time (manual, sequential), or be prone to cause collisions between UAVs (simultaneous), especially when UAVs are close together on the ground.

With respect to swarm resilience, we could find a few proposals. Dano et al. [21] provide resilience from a system engineer point of view. Their work focuses on a higher abstraction layer, and implemented contract-based design invariant contracts in a Matlab-based flight simulator to quantify the defined location resilience metrics of their system. Chen et al. published a paper [22] that is focused on effectively reorganizing the surviving UAVs in a severely damaged UAV swarm. They start by analyzing the damage-resilience problem of unified UAV Swarm Networks (USNETs). The goal of their work was to design a damage-resilient mechanism, which is usually divided into multiple disjoint subnets of isolated nodes. Three challenges are investigated: first, the network will be divided into several disjoint subnets or isolated nodes; secondly, they work on restoring the network connectivity; finally, they explain how to reduce the computational and communication overhead.

The work by V.T. Hoang et al. [23] presents an algorithm to reconfigure a formation of multiple UAVs. This work is especially focused on the application of vision-based inspection of infrastructure. It presents a new algorithm for reconfiguration based on the angle-encoded Particle Swarm Optimization (PSO). They begin with a 3D representation of the surface to be inspected, and a set of intermediate waypoints. Additionally, new constraints are proposed to decrease the chances of collision, and to increase task performance, based on the assumption that an optimal path is produced by using the θ-PSO path planning algorithm. Other works such as [24], [25], and [26] use an approach which is called flocking.

Flocking is a behavior that is common in nature, for instance in a group of fish, birds, or insects. It consists of a few basic rules that are applied to each entity of the group. When those rules are respected, the group will stay united without collisions between the group elements. While flocking mechanisms are great to keep a swarm of UAVs organized, they do not provide the flexibility to completely define and change the formation itself. Furthermore, controlling the altitude of a UAV is usually done with a PID controller. to this end, Muliadi et al. [27] propose an artificial neural network. They show (in simulation) that, with the use of their neural network, a better performance can be achieved.

Finally, different UAV landing approaches have been studied. Chen et al. [28] succeeded in landing a real UAV on an object moving with a speed of 1 m/s. A camera was used to track the position of the landing platform (xy-coordinates), and a LIDAR sensor provided detailed information about the altitude. This research work introduced a robust method to track and land on a moving object. However, the use of a LIDAR sensor discourages the solution, as it tends to be too expensive when scaling to a high number of UAVs. Nowak et al. [29] proposed a system in which a UAV could land both at night, as well as during the day. The idea is simple yet robust and elegant: a beacon is placed on the ground. The light emitted from the beacon is then captured by a camera (without a infrared filter), and the drone moves (in the xy-plane) such that the beacon is in the center of the picture. Once centered, the height is estimated based on the image area occupied by the beacon, and the drone's altitude is decreased in order for it to land safely. Shaker et al. [30] suggested another approach: reinforcement learning. In this approach, the UAV agent learns and adapts its behavior when required. Usually, reinforcement learning takes a lot of time. To accelerate this process, a technique called Least-Squares Policy Iteration (LSPI) is used. With this method, a simulated UAV (AR100) was able to achieve a smooth landing trajectory swiftly.

Chapter 3

ArduSim: a multi-UAV simulator

In the previous chapter, we explained what drones are, and why we want to use swarms of UAVs. Now, we want to start creating algorithms and protocols to use them. However, testing these algorithms directly on real drones is a difficult taks, being time-consuming, expensive, and riskful. Hence, before testing algorithms on real drones, it is a good idea (almost a requirement) to test the code in a simulator. Since, nowadays, UAVs are quite popular, there is no shortage of UAV simulators. However, nearly all of them, are limited to a single UAV, and are not able to simulate multiple at the same time. Furthermore, they are often designed to (i) train pilots how to fly, or (ii) as a video game. Hence, they are not appropriate for our purpose. For that reason, our research group *Grupo de redes de computadores (GRC)* [1] developed its own multi-UAV simulator called ArduSim. At the start of this thesis, ArduSim was already created, and it had the capability of simulating several hundreds of UAVs at the same time, and allowed them to communicate with each other. This version of ArduSim served us well during the first steps of this thesis. However, due to its continuing growing size, it became complicated to maintain well, and to create new features. Especially in research, it is important that our software is flexible, and that we are able to quickly generate new protocols. Hence, we set out to change the architecture of ArduSim completely, so as to facilitate future development. In this chapter, we will explain the design of our improved simulator. However, we start by explaining the characteristics of the old

[1] https://grc.webs.upv.es/

version (which we want to maintain), detailing the shortcomings of that version (which we will improve upon).

3.1 Original version of ArduSim

The original version of ArduSim [31] was created in order to simulate various drones at the same time. The project was developed in Java, and all the code is published under the Apache License 2.0 [32] on GitHub [33] so that everyone can freely use, distribute, and modify the software. ArduSim can be seen as an environment that improves up on the single drone simulator provided by ArduPilot [6]. The simulator of ArduPilot is a Software In The Loop (SITL) simulator that closely represents the real firmware on a flight controller. However, using the SITL directly has several problems: (i) it is complicated, (ii) it poorly supports multiple UAVs, and (iii) it does not include communication models. So, in ArduSim we utilize the high precision obtained from ArduPilot and build around it, in order to facility multi-UAV simulations. The most important features of ArduSim can be summarized as follows:

- **Effortless protocol deployment on real UAVs.** ArduSim was developed to make deployment on real UAVs straightforward. To achieve this, we use the same technology that GSCs are using. In particular, we are using the MAVLink [7] communication protocol to control the UAV. The only requirements to deploy a protocol on a real drone are to attach a Raspberry Pi with a Wi-Fi adapter, and to connect it to the telemetry port of the flight controller.
- **Complete Application Programming Interface.** ArduSim provides its developers an easy-to-use interface, which translate the most common maneuvers during a flight (e.g. taking off, moving, landing, etc.) into MAVLink messages that are then sent to the SITL simulator from ArduPilot.
- **Soft real-time simulation.** In order to represent reality as close as possible, simulations in ArduSim are performed in near real-time.
- **High scalabilty.** ArduSim is also very scalable, and on a standard computer (Intel Core i7-7700, 32 Gb RAM, SSD hard drive), it is able to run up to 100 UAVs in near real-time, and up to 500 UAVs in soft real-time.
- **UAV-to-UAV communication.** An important part of any swarm protocol is the communication between the UAVs. Hence, inside ArduSim, various models were created to virtualize IEEE 802.11a technology. These models

are based on experiments using real UAVs. When ArduSim is used on real drones, it automatically uses the Wi-Fi adapter of the Raspberry Pi.

- **Graphical and command line interface.** ArduSim has both a graphical and a command line interface. The graphical interface (2D) can be used in order to verify if a protocol is working as intended, and to show the results. The command line interface, on the other hand, can be used to automate a batch of simulations, which is very useful when running an extensive amount of experiments.

As stated before, the original version of ArduSim was a great tool, and in fact it has been used to develop and test most of the protocols that we will present in this thesis. Nevertheless, software must develop and improve in order to meet our (future) needs. ArduSim grew as a research project and, since the exact requirements were not known beforehand, it grew organically. As of today (2023), ArduSim has been used for almost seven years, and in those seven years we created a better understanding of what is important. Using this knowledge, we took the time to evaluate what can be improved. The main shortcomings of ArduSim are all originating from the fact that ArduSim is a monolithic program. Hence, we are limited by the following:

- **Programming language dependency.** ArduSim is written in Java. Although this is a robust programming language, it is not the fastest neither the easiest programming language. Nowadays, many applications rely on artificial intelligence, which is overwhelmingly done in Python, and computational expensive algorithms are better implemented in C or Rust. Hence, to better suit our needs, we must remove this restriction and create a system in which any programming language can be used.

- **Single protocols.** ArduSim was created in order to develop, and test, multi-UAV protocols. However, it was created in such a way that only one protocol can be used at the same time. In the future, we will need to develop entire UAV applications. This means that various protocols will be used at the same time. In order to facilitate this process, the new architecture must allow for various protocols to be executed concurrently.

- **ArduPilot version dependency.** ArduSim is using the ArduPilot simulator to simulate the flight behavior of the UAV accurately. However, its implementation makes it highly depended on the version of ArduPilot. This makes it difficult to switch between different versions or between different types of aircraft. In the future, swarms of UAVs might exist of VTOL and

fixed-wing UAVs at the same time. In order to simulate those types of swarms, we need to decouple ArduSim further.

- **Bound to one computer.** Since ArduSim is a monolithic program, we must execute all the code on the same machine. Since it is a lightweight program, this is not a problem yet. However, in the future, more applications will rely on artificial intelligence. Those algorithms need to be trained and executed on servers equipped with GPUs. Hence, we must ensure that ArduSim can be run in a distributed fashion.
- **Steep learning curve.** Although ArduSim facilitates many UAV operations, there is still a steep learning curve. This is mainly due to the fact that it is a large program, and one must learn many aspects of the simulator at the same time. In order to flatten this learning curve, we must try to abstract as many details as possible. In this way, a new user only needs to learn the basics, and can be concerned about the details at a later stage.

In order to solve the above-mentioned issues, we propose a drastic architectural change. Specifically, we propose breaking up the monolith program into smaller, self-contained microservices. Each service will communicate with the other services using standard TCP/IP communication. Since almost all programming languages support TCP/IP communication, we are no longer dependent on one specific programming language. Each algorithm (previously known as protocol) can be executed in its own microservice, and with some modifications we can easily use the input of multiple algorithms in order to build complex applications. By using a standard interface to communicate with the UAV controller, we can create a separate service for each version of the ArduPilot simulator. In this way, switching between versions will be made easier. Furthermore, when different machines are placed in one network, they can exchange TCP/IP messages easily. In that way, we are no longer bounded to one computer. Finally, each service can be studied on its own, which flattens the learning curve.

3.2 Design and implementation

In the new design of Ardusim, the code is split into self-contained microservices. In this subsection, we will detail the various categories of services, and how they will interact with each other. We can categorize the services in the following five basic types: (i) algorithms, (ii) applications, (iii) UAV controllers, (iv) auxiliaries, and (v) networking.

The general idea is that, at any time, various algorithm services will be active. These algorithm services will contain the logic in order to solve a very specific

problem. This can, for instance, be code to: adjust the altitude of a UAV, avoid collisions with other UAVs, move to a waypoint, etc. The algorithms will send suggestions towards a single application service. In this application service, the different suggestions are received, and the logic of the application service will decide which suggestion should be prioritized. To explain this more clearly, let us imagine we want to create an application where a UAV flies from one point to another in a mountainous region. In order to avoid crashes, a constant altitude above the ground must be maintained. In this example, we need two algorithms, one that guides the UAV towards a point, and another that maintains a constant altitude. The first algorithm might suggest that the UAV has to go forward, while the second algorithm might suggest that the UAV must change altitude. In the application service, a choice will be made regarding which one is more critical. Since we want to avoid crashes, it is obvious that changing the altitude first is more important. In some advanced cases, two suggestions can actually be joined together in the application service. It is, for instance, possible to move the UAV forward and change altitude at the same time. Whatever the application service decides, a command will be sent to the UAV controller service. This service contains all the code for a specific version of a UAV. It has, however, always the same interface, and hence we can easily change between different versions. The UAV controller service, will use the SITL simulator provided by ArduPilot. Using the SITL we can simulate the flight behavior very accurately. SITL will update the drone's position, speed, rotation, flight mode, etc. This information is fed back to all the different services. It is also sent to the auxiliaries services, among which we can include various visualization and logging services. The above-mentioned architecture is replicated multiple times in order to create a swarm of UAVs, where each instance will represent one UAV. Note that it is not necessary that we run the same application, algorithms, or even version of UAV controller in each instance. Hence, we can create heterogeneous swarms easily. Of course, in order to create a swarm, the algorithm instances should be able to communicate with the others. For that reason, there are also networking services, which will simulate the communication between UAVs. Finally, there are some auxiliary services connected to the algorithms and the UAV controller. In Figure 3.1, we illustrate our new architecture.

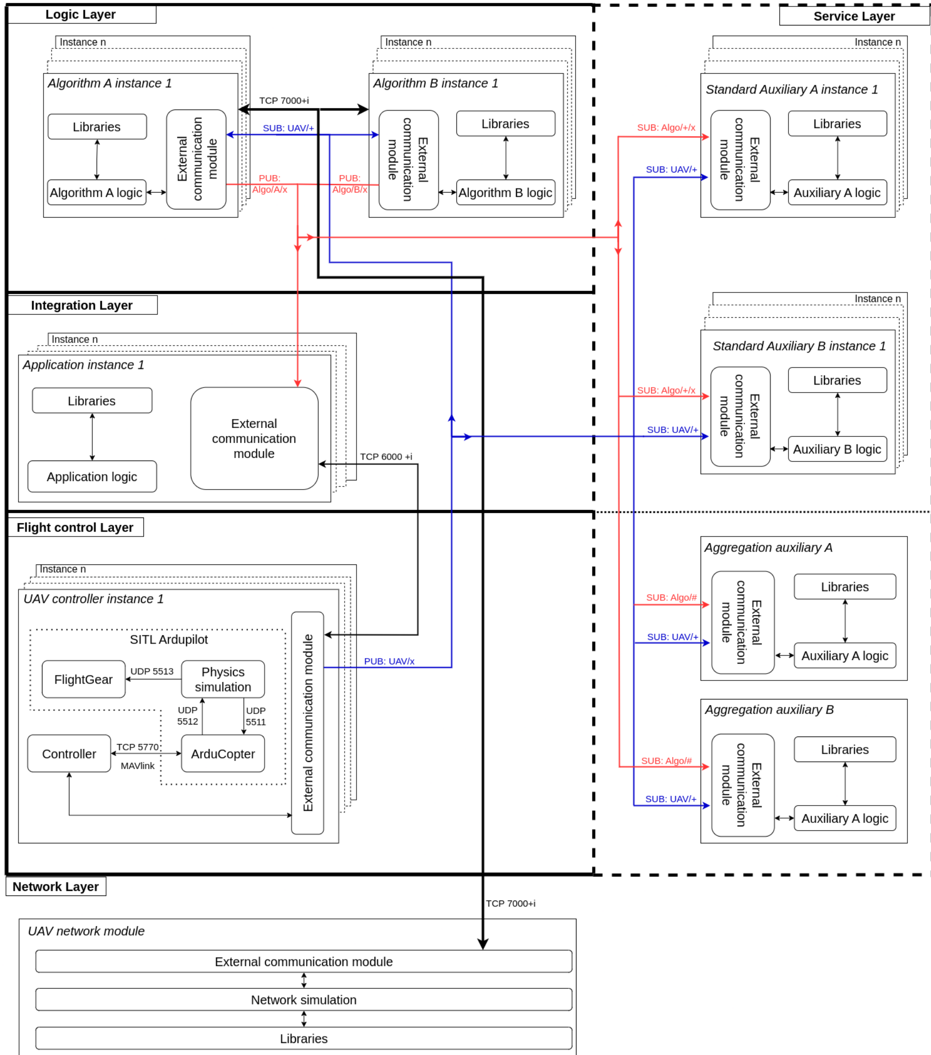


Figure 3.1: The proposed architecture of ArduSim.

Notice that all services are connected using TCP/IP-based communication. In order to facilitate the communication flow, we sometimes use the publish-subscribe pattern. While using this pattern, the sender publishes a message on a specific topic. Receivers have to subscribe to the messages they are interested in. In this

way, it is easy to establish a many-to-many communication flow between various services. As an example, an algorithm service only has to publish its message once, and it can then be received both by the application and the by auxiliary services.

The UAV controller service is probably the most complex service because it includes the SITL simulator from ArduPilot. As stated before, ArduPilot is the firmware that runs on the flight controller. However, they have also created a (single-UAV) simulator, which is partly built upon another simulator called FlightGear. There are different versions of the SITL ArduPilot module. The modules can differ in the type of aircraft (e.g. copters, planes, rovers, etc.), and for each type of aircraft there is a specific release available. Is it worth mentioning that, between the different types of aircraft, the way to control them can differ greatly. Hence, for each version we are interested in, we need to develop a controller module. This controller module will communicate with the SITL ArduPilot module using the MAVLink protocol. The controller module interacts with the external communication module using a general interface. This means that the application service can send general controls to all types of aircraft, which decouples the application from the control logic. Besides simulating the behavior of UAVs, ArduSim should also be able to control real UAVs. Due to our modular design and the use of the MAVlink protocol, it only requires a minor change. Specifically, a small onboard computer (e.g. a Raspberry Pi) should be mounted on the UAV. This onboard computer will run ArduSim and connect to the flight controller using a serial connection. In such case, it is of course no longer necessary to run the SITL simulator, and the commands from the Controller module should be sent to the flight controller using that serial connection.

Finally, within the category of auxiliary services, we distinguish between standard, and aggregation auxiliary services, where the former has one instance per UAV, and the latter just one instance in total. For example, a visualization service that shows the position of the UAVs on a map belongs to the aggregation auxiliary services, because it will show information of all UAVs. Other services, like logging the output of an algorithm, will need various instances, in order to separate the data from the various algorithm instances.

3.3 UAV swarm formation

One algorithm service that we use very often is a service that calculates the positions of each UAV in a specific formation. Currently, we have the possibility to create four different types of formations: (i) linear, (ii) circular, (iii) matrix, and (iv) random. Since, in subsequent chapters, we will use these formations, we now provide a brief overview of each formation type. In general, a formation can be

created for any number of UAVs, and the user can indicate a minimal distance between the UAVs that needs to be maintained.

The linear formation is the easiest formation, but nevertheless a very useful one. Basically, all the drones are on a line with some minimum spacing between them (see Figure 3.2). Although it is a simple formation, it is used very frequently used because it allows to cover a large area.

Figure 3.2: Linear formation.

The circle formation, shown in Figure 3.3, places all but one UAV along the rim of a circle, and one UAV in the center. Although its implementation is equally straightforward, there is a small caveat. We have to take into account that, when the number of UAVs grows, the radius of the circle must grow as well. Otherwise, the minimal distance between the UAVs cannot be guaranteed.

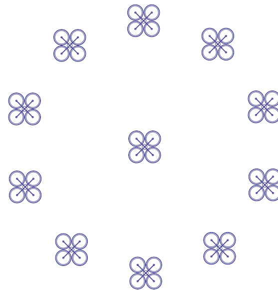


Figure 3.3: Circle formation.

The matrix formation (see Figure 3.4), places the UAVs in a matrix formation. In many cases, a matrix would be created by starting at a corner and fill the rows and columns. However, in the specific case of UAVs formations, this is not a good idea, because it is prone to make the formation asymmetrical, which in turn enlarges the distances between the UAVs, thereby decreasing the network performance. Thus, we create a matrix by starting at the center, and place each new UAV in a vacant spot closest to the center, while maintaining the minimal distance required by the user.

Finally, we have a random formation (see Figure 3.5, that merely places the UAVs in a random spot. Although this formation is not used in practice, it is useful to verify if our other algorithms work for any generic case.

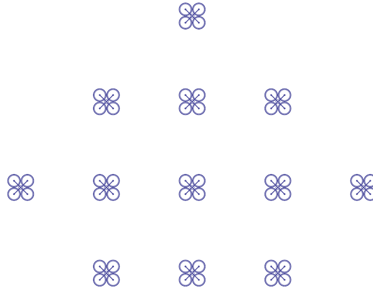


Figure 3.4: Matrix formation.

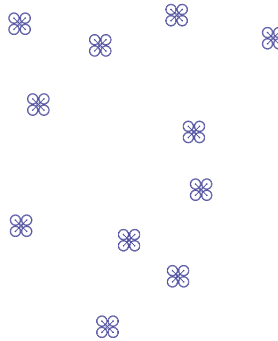


Figure 3.5: Random formation.

3.4 UAV-to-UAV communications

One, very import service in the above-mentioned architecture is the communication between the UAVs. In order to create swarm applications, the UAVs must be able to communicate with each other. The most straightforward way to provide communications between real UAVs is to rely on broadcasting using UDP. However, UDP packages can get lost. We can model this behavior to various degrees of accuracy. In general, the higher the accuracy, the longer it will take to calculate whether a package will get lost. Since the communication between the UAVs is a central part of our simulator, we provide four different models:

- *Unrestricted.* It uses an ideal medium where data packets always arrive to all possible destinations (basic model).
- *Fixed range.* data packets arrive to another UAV only if the distance between them is lower than the defined threshold (simple model).

- *Realistic 802.11a with 5dBi antenna.* the probability that a data packet is received by another UAV depends on the distance between the UAVs. This realistic model is obtained from real experiments where the packet loss rate between two UAVs was measured using a Wi-Fi ad-hoc network link in the 5 GHz band. Out of these experiments, we derived a model for the package loss in function of the distance (x) between the UAVs. Beyond 1350 meters, we consider that packet losses reach 100%, and for smaller distances the package loss is modeled by $y = 5.335 \cdot 10^{-7} \cdot x^2 + 3.395 \cdot 10^{-5} \cdot x$. Furthermore, in this model, we include carrier sensing functionality.

- *OMNet++ co-simulation.* In order to simulate the channel model with a very high degree of accuracy, we include a connection with the network simulator OMNet++[34]. With the use of OMNet++, we are able to test a plethora of scenarios, including different 802.11 standards, the influences of buildings, etc. However, it is computationally very expensive, and it only allows a few UAVs to broadcast messages at the same time.

The first three models are quite straightforward to implement. However, the OMNeT++ extension deserves some extra explanation. In OMNeT++ we represent our UAVs as ad hoc hosts, which are readily available in the INET framework[2]. The INET framework provides protocols, agents, and other models for communication networks. In particular, we are especially interested in their IEEE 802.11 models. By default, all the network traffic is generated and used within OMNeT++. However, in our co-simulation, all the messages are generated and used by ArduSim instead. Hence, we need to be able to send those messages, along with the location of the UAV, from ArduSim to OMNeT++. Using our new microservice architecture, this connection is quite straightforward. We just need to create a new UAV network module, which runs OMNeT++. Inside OMNeT++, we must create a listener that accepts messages from the outside. For performance reasons, there is one listener that receives all the messages coming from the various UAVs. The messages are then handed over to the corresponding UAV in OMNeT++, and transmitted wirelessly by that node. The transmission relies on IEEE 802.11p technology, and models for path loss and obstacle loss are applied. Once the message is received by another UAV in OMNeT++, it is sent back to the UAV network module. In Figure 3.6 we represent this connection graphically.

When using OMNeT++, we are able to create (virtual) obstacles in an attempt to mimic a certain target environment as accurately as possible, which in turn will impact the communication between UAVs. In principle, the features that

[2] https://inet.omnetpp.org/

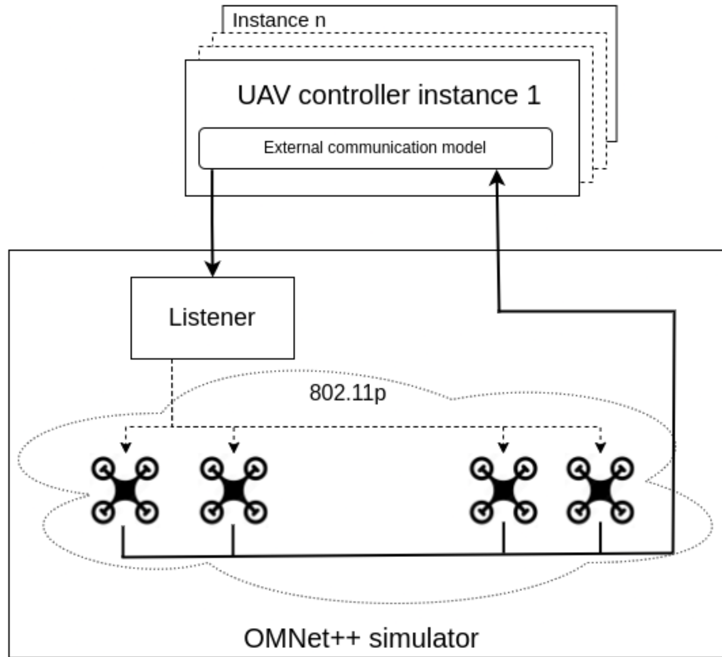


Figure 3.6: Connection between ArduSim and OMNeT++.

OMNeT++ offers to create obstacles are quite rudimentary. With the use of an .xml file, one can place basic shapes (cuboids, spheres, prisms, etc.) on a specific location, and with the properties of a specific material (e.g. concrete). The obstacle file is loaded when the simulation starts. Hence, the obstacles are always static. In order to make our simulation as realistic as possible, we created a Python script that converts georeferenced building information (obtained from OpenStreetMap [35]) and digital elevation models (obtained from the Spanish government [36]) into the format that OMNeT++ requires. In Figure 3.7, we represent a small town in our region in OMNeT++. Our current approach to model buildings in OMNeT++ is limited, and does not allow us to place a roof on top of the buildings (only the walls are modeled). Hence, we represent the buildings by following the shoebox model.

As stated before, using OMNeT++ is computational more expensive than our other channel models. In order to test the limits of this co-simulation, we performed some experiments.

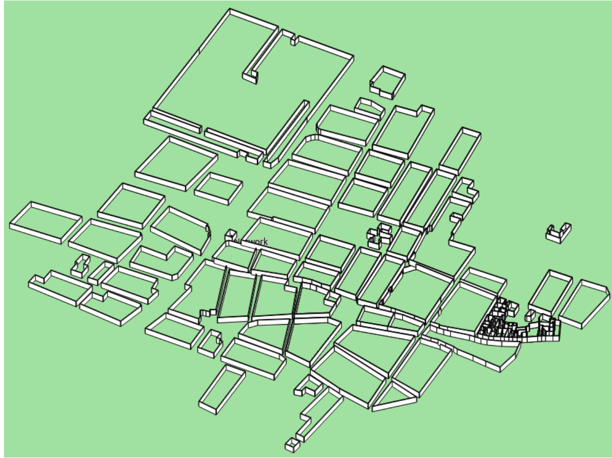


Figure 3.7: Example of buildings generated in OMNeT++ based on OpenStreetMap data (Foios, a small town in Spain).

First, we want to investigate the inter-process overhead. In particular, we want to measure how much longer it takes for one message sent by a UAV to be received by another UAV. To this end, we created a small experiment where two UAVs are placed 100 meters apart from each other, and where one UAV is sending messages of 80 bytes to the other one. In the message, we include a timestamp to register the instant when the message is sent. When the message is received by the other UAV, the time difference is measured; notice that this is possible as the clock used in both cases is the local clock in the scope of a single machine (the one running ArduSim). In order to accurately portray our results, we measure the average and the standard deviation for 100 messages. In our experiments, we compare three scenarios: (i) simulating communication in ArduSim (basic channel model only), (ii) simulating communication via OMNeT++ (advanced channel model), and (iii) simulating communication via OMNeT++, but running the latter on a different PC (in the same LAN) to determine the impact of network overhead as well. We compare these results with a baseline, which is taken when solely using OMNeT++ (i.e. no ArduSim). This baseline represents the approximate time it would physically take for a message to be delivered wirelessly (transmission time plus propagation time), without considering other communication layers; obviously, values cannot go below this baseline threshold. Furthermore, we performed this experiment using different message frequencies.

The results are shown in Figure 3.8. As we can observe, the message's frequency

does not have a significant influence on the end-to-end delay. When using only ArduSim, the time between sending and receiving a message is 50 ms without any fluctuation. The delay decreases (to about 12 ms) when we transfer the messages from ArduSim to OMNeT++ on the same PC. We can also observe a standard deviation (indicated with the error bars) of 2.1 ms. When using OMNeT++ on a different PC (via a LAN), the delay is obviously larger (about 14 ms). However, the difference remains very small. In all cases, the delay is shorter when using OMNeT++, which is most likely because ArduSim is written in Java, and OMNeT++ in C++. This highlights the high effectiveness of the co-simulation strategy devised.

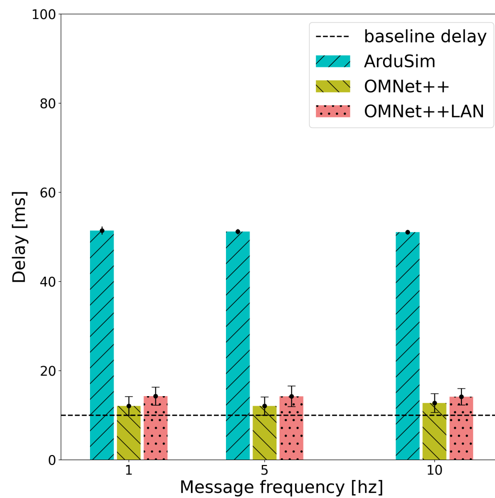


Figure 3.8: Message delay caused by the inter-process overhead in different scenarios.

In our previous experiment, there was only one UAV sending a message. In real applications, multiple UAVs will send (and receive) messages at the same time. Hence, we conduct an experiment to measure the influence of multiple UAVs sending messages (in particular, broadcast messages). In our experiment, we place UAVs in a circle with a radius of 100 meters. One UAV is at the center, and (as before) will be listening to the messages, and calculating the time difference between transmission and reception to determine the end-to-end delay. For this experiment, we run ArduSim and OMNeT++ on the same machine. During the experiment, we will increase the number of UAVs from 2 up to 35. As before,

we measure the delay between transmission and reception. Besides that, we also observe whether OMNeT++ is still running in soft-real time. The results are shown in Figure 3.9.

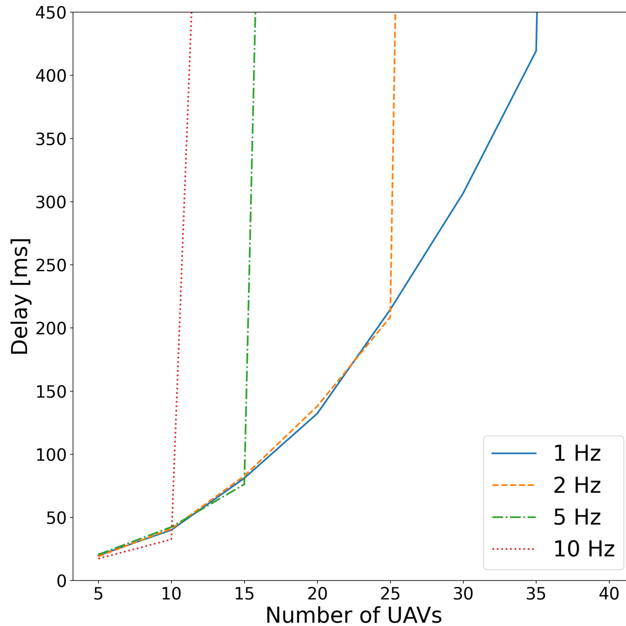


Figure 3.9: End-to-end delay when varying the number of UAVs in the experiment, and the message rate per UAV. All UAVs act as both senders and receivers.

As one can clearly observe, the number of UAVs sending messages has a big impact on the message delay. In general, the higher the load (i.e. more UAVs, or a higher frequency), the longer the message delay, up to a point when OMNeT++ can no longer process the messages in real time, causing the delay to gradually grow up to infinity (theoretically). The main issue is that OMNeT++ is a single-threaded tool and, therefore, it does not allow for parallelism in the scope of a single experiment. Besides that, the number of events (i.e. messages sent and received) grows quadratically with the number of UAVs when broadcasting messages. Therefore, we are limited to a moderate amount of UAVs sending messages. However, when we change our experiment setup to one UAV broadcasting messages, and multiple UAVs receiving those messages, we can observe (see Figure 3.10) that we are able to simulate many more UAVs. Of course, in this experiment, the

load (measured as number of simulation events) grows linearly with the number of UAVs. Hence, we are able to simulate up to 1000 UAVs in the best-case scenario.

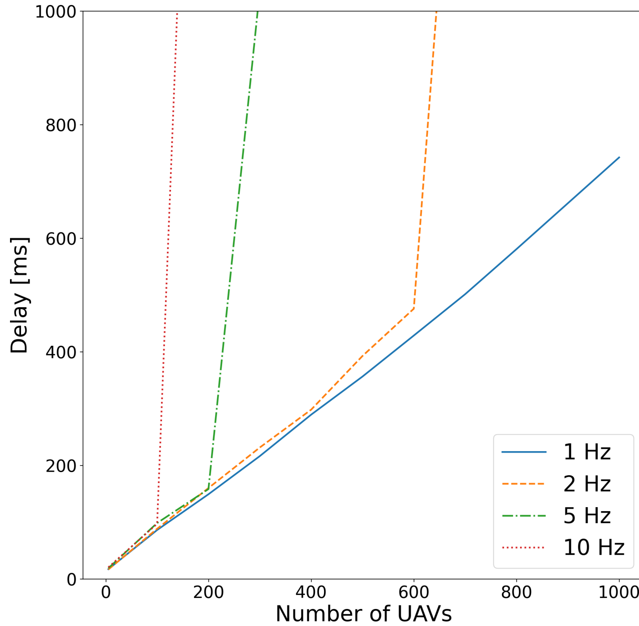


Figure 3.10: End-to-end delay when varying the number of UAVs in the experiment, and the message rate per UAV. Only one UAV acts as transmitter, and the rest merely act as receivers.

3.5 Summary

Simulating the behavior of a UAV swarm is a very important part of the development cycle. It allows us to quickly, and most important safely, test our protocols before deploying them on real UAVs. For that reason, our research group (GRC) developed a multi-UAV simulator called ArduSim. In this chapter, we presented a new architecture for ArduSim, which makes it more flexible, maintainable, and easy to use. We changed the previous monolithic version to a microservices-based architecture. Using this new architecture, we are no longer bounded to a single programming language, and able to execute various algorithms/protocols at the

same time. Furthermore, it allows ArduSim to be executed in a distributed fashion. Finally, in ArduSim, we also simulate the communication between UAVs. We implemented various models that range greatly in accuracy. Our basic models are very lightweight, but simplify a complex reality. In order to simulate the communications very accurately, we created a connection with the popular network simulator OMNeT++, so that ArduSim and OMNeT++ can be used together. While using OMNeT++, we are able to simulate various wireless communications, experiment with different path loss models, and observe the influence of obstacles as well. To validate our co-simulation approach, we performed various experiments to determine its limitations. We found that, when using ArduSim and OMNeT++ together, we are limited to a moderate amount of UAVs sending messages at the same time. Hence, co-simulation is best used when experimenting with a smaller swarm in a complex scenario (e.g. a city), where obstacles can influence the communication greatly.

Chapter 4

Assigning airborne positions efficiently

In previous chapters, we explained what a swarm of UAVs is, and how we can test different protocols in our simulator ArduSim. Using this knowledge, we will start by designing and developing protocols in order to use such a swarm of UAVs effectively. So, we start at the beginning, when the UAVs are still at the ground. First we have to ensure that the UAVs reach their target airborne position. For a single VTOL UAV this issue does not exist; it just has to move towards the GPS location the user has set. For a swarm, however, it is not so trivial. Imagine a swarm of n drones that are all placed, close to each other, on random locations on the ground. All these drones will need to be assigned to a specific (GPS) location in the air. These locations are given by the user, and are often part of a flight formation (e.g. a line, a circle, a matrix, etc.). Now the question that arises is, which UAV on the ground has to go to which place in the air. In principle, this assignment could take any form. For instance, it could be completely random. However, this would not be a very intelligent manner of assigning the airborne positions. Given that the battery power is limited, and that the UAVs are only useful while flying, we do not want to waste a lot of time during this take-off process. Given that the take-off process correlates perfectly with the total distance travelled by all the UAVs, we search for an assignment strategy where this distance is minimal.

4.1 Overview

There are various types of algorithms that can solve an assignment problem. In this chapter, we will discuss three different types. We start with a brute-force method. This is the easiest manner to find the optimal assignment. However, since it checks all the possible options, it takes a long time in order to find the best solution, especially when there are many UAVs in the swarm. In order to reduce this calculation time, there are two solutions. The first solution is to simply make the problem to be solved a little bit easier. Sometimes it is sufficient to find an assignment that is “good enough” instead of finding the optimal assignment. For this reason, we created a heuristic which finds a decent assignment very quickly. The second solution is to try and solve the problem in a more intelligent manner than just checking all possible assignments. This solution inevitably involves more mathematics, and luckily for us, other great minds (Harold Kuhn and James Munkres) had solved this already, and denominated their algorithm the Kuhn-Munkres algorithm (KMA).

4.1.1 Brute-force

The brute force algorithm (shown in Algorithm 4.1) goes through all the possible assignments (i.e. permutations of the air locations), and calculates the total distance travelled for each possible assignment; afterward it will return the assignment where the total distance travelled is minimal. The algorithm is very easy to implement and to understand. However, since we are using permutations, the computational cost grows quickly. As an example, in Figure 4.1 we show all the possible permutations for only four UAVs, i.e. 24 different combinations. As a result, this algorithm has a very high time complexity ($O(n! \times n)$).

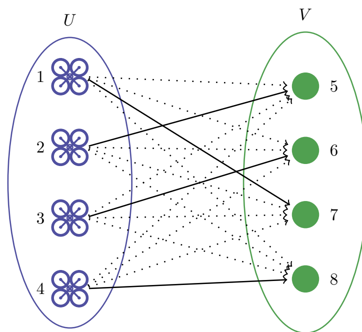


Figure 4.1: All possible solutions that exist with only four UAVs.

Algorithm 4.1 : Brute-force assignment algorithm

$bestDistance = \infty$
$bestPermutation = null$
$permutations = getPermutation(airLocations)$
for all permutation in permutations **do**
  $totalDistance = 0$
  for all id in UAVs **do**
    $groundLocation = groundLocations.get(id)$
    $airLocation = permutation.get(id)$
    $distance = ecludianDistance(groundLocation, airLocation)$
    $totalDistance+ = distance$
  end for
  if totalDistance < bestDistance **then**
    bestDistance = totalDistance
    bestPermutation = permutation
  end if
end for
return $bestPermutation$

4.1.2 Heuristic

Due to the high time complexity of the brute-force algorithm, we cannot utilize it when using many UAVs. So, in order to calculate an assignment for many UAVs, we present a heuristic. This heuristic (as any) offers a trade-off between calculation time and accuracy (i.e. total distance travelled). The heuristic consists of determining a location on the ground, which is central to the UAVs deployed. Then, this central position is used to compute the distance towards all positions in the desired flight formation, which are then sorted in descending order. Using this sorted list, the UAV closer to each of these positions is then assigned to it. All details are explained in algorithm 4.2. This algorithm is able to calculate an assignment very fast ($O(n^2)$) because it reduces all the ground locations to a single point. However, since in general formations are symmetrical, the total distance travelled will only be slightly higher compared to the brute-force solution (which calculates the optimal assignment).

4.1.3 Kuhn-Munkres algorithm

Although our heuristic is very fast, it does not provide the optimal assignment. Therefore, we searched for a solution which is not as slow as the brute-force method,

Algorithm 4.2 : Heuristic(numUAVs, groundLocations, flightFormation)

```
1:  centerLocation = mean(groundLocations)
2:  airList = ∅
3:  for airloc in airLocations do
4:      airList ← (airloc, airloc.distance(centerLocation))
5:  end for
6:  sort airList in descending distance order
7:  fit = ∅
8:  for airLoc in airList do
9:      bestError = MAX_VALUE
10:     for groundLoc in groundLocations do
11:         error = groundLoc.distance(airLoc)
12:         if error < bestError then
13:             bestError = error
14:             bestID = groundLoc.ID
15:         end if
16:     end for
17:     fit ← (id, groundLocations[bestID], aLocation)
18:     groundLocations.remove(bestID)
19: end for
20: return fit
```

but is still able to provide the optimal assignment. Currently, the fastest solution is achieved by the Kuhn-Munkres algorithm (KMA). This algorithm was developed by James Munkres in 1957. He based his work on an algorithm first developed by H. W. Kuhn [37], who in turn was inspired by two Hungarian mathematicians. Therefore, the algorithm is also known as the Hungarian, the Munkres, or the Kuhn-Munkres algorithm. In the original problem, the authors were trying to match a set of n persons to a set of n jobs in the most cost-efficient way. Each person had a specific cost for a job, which was represented in a cost matrix. As one might imagine, the KMA can be used for many applications as long as a cost matrix is provided. In our work, the cost matrix is an $n \times n$ matrix, where n is equal to the number of UAVs, and the elements are calculated by the Euclidean distance between a ground, and an air location. Once the matrix is created, some steps are (repetitively) performed on the matrix. These steps are detailed in Algorithm 4.3. The basic idea behind these steps is to create zeros, which indicate a pair (ground and air position) with the lowest cost possible. Throughout the process, the zeros in the matrix are updated in such a way that the total cost (i.e. the total distance travelled) is minimized. After some repetitions (depending on the matrix), the algorithm terminates. The final set of zeros in the matrix correspond to the

optimal assignment of ground to air positions. The KMA guarantees to find the optimal solution within $O(n^3)$, which is a lot faster than the brute-force approach, but still slightly slower than our heuristic. This difference will become important when n (i.e. the number of UAVs) increases.

Algorithm 4.3 : Kuhn-Munkres CPU(numUAVs, groundLocations)

Require: $groundLocations.size = numUAVs$

Step 1 → create a matrix of size $numUAVs \times numUAVs$, and fill it out with the distance between the ground and air location

Step 2 → **Subtract row minima:** Subtract the smallest entry in each row from each entry in that row in the distance matrix.

Step 3 → **Subtract column minima:** Subtract the smallest entry in each column from each entry in that column in the distance matrix.

Step 4 → **Cover all zeros with the minimum number of lines:** Using the smallest number of lines possible, draw lines over rows and columns in order to cover all zeros in the matrix.

Step 5 → If the minimum number of covering lines is n, **an optimal assignment of** *zeros* **is possible and the process is finished**. If the minimum number of covering lines is less than n, an optimal assignment of *zeros* is not yet possible. In that case, proceed to *Step 6*.

Step 6 → Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Then return to *Step 4*.

4.2 Experiments & results

Now that we have presented various algorithms to solve the assignment problem, we will compare these algorithms and decide which one is the most adequate to use. In order to do so, we perform various experiments using ArduSim. In our experiments, we measure the calculation time, the total distance travelled, and the number of flight paths crossing (i.e. a possibility of collisions). We are mainly focussed on scalability and versatility. Hence, in our experiments, we changed the number of UAVs and the airborne positions. These airborne positions are

calculated based on a specific formation. We mainly use three different formations (i.e. linear, matrix, and circular), but experiment with irregular formations as well.

We start by measuring the calculation time. The results are shown in Figure 4.2, and are exactly as we would expect. The brute-force algorithm cannot be used due to time constrains, and our heuristic is faster than the KMA. We can conclude that our heuristic will always provide an assignment within a reasonable timeframe. Even for swarms up to 2000 drones, it only takes a few tens of a second. The same cannot be said for the KMA. Here we see that, for 750 UAVs, the calculation time is close to 16 minutes, which is definitely too much for any practical application. Furthermore, we can observe (in Figure 4.3) that, for the KMA, the execution time highly depends on the formation. The KMA performs notably worse when the UAVs are spread out over one dimension (linear formation). Yet, whenever the formation is more uniform over the two dimensions, the performance is improved. Our heuristic, however, does not have this defect, and the calculation time is independent of the formation itself (see Figure 4.4).

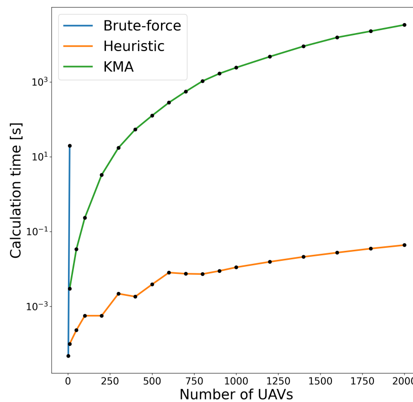


Figure 4.2: Average calculation time when considering all UAV formations and assignment algorithms.

It is worth pointing out that the calculation time is only one part. It is also important that the obtained assignment minimizes the total distance travelled. In this regard, the KMA guarantees a minimal distance cost, whereas our heuristic merely attempts to reduce it. Hence, the total distances travelled will always be higher while using our heuristic (in rare cases it might be equal). Nevertheless, to evaluate the effectiveness of our heuristic, it is interesting to compare both distances. The results are shown in Figures 4.5,4.6,4.7. As we can see in these

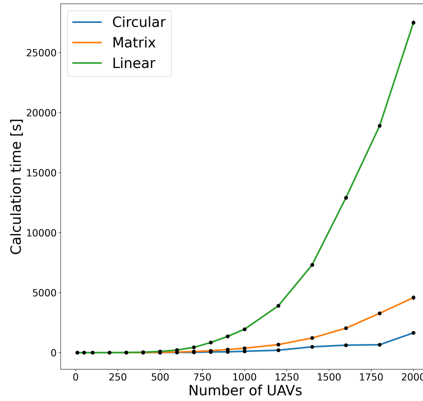


Figure 4.3: Calculation time using the KMA algorithm for various UAV formations.

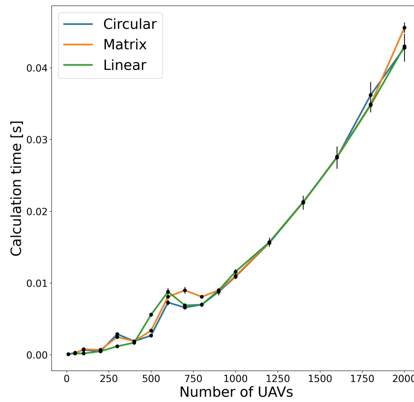


Figure 4.4: Calculation time using the heuristic algorithm for various UAV formations.

figures, and as expected, the total distance travelled is somewhat higher when using our heuristic. As shown on the zoomed-in view, the difference between the two algorithms remains relatively constant. On average, this extra distance is of 2879 m, 2378 m, 2457 m for the circle, linear, and matrix formation, respectively. Although this might seem a lot, one must remember that this is the total distance sum for all UAVs (on average 1000 UAVs). Nevertheless, we can see that there is

a substantial difference between using the KMA and our heuristic, and w.r.t. the total distance travelled, the KMA clearly outperforms the heuristic. Furthermore, we can also observe that the total distance travelled is different for each formation. This is because some formations, such as the matrix formation, are a lot more compact (i.e. more UAVs fit in the same area) than others (linear).

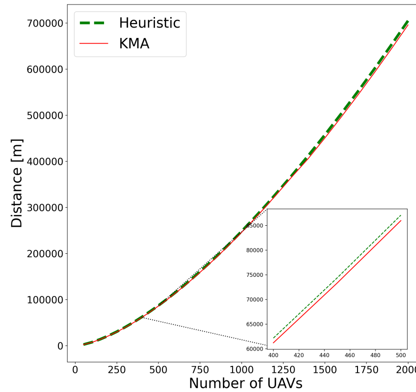


Figure 4.5: The total distance travelled by all UAVs using the heuristic and KMA for the Circle formation.

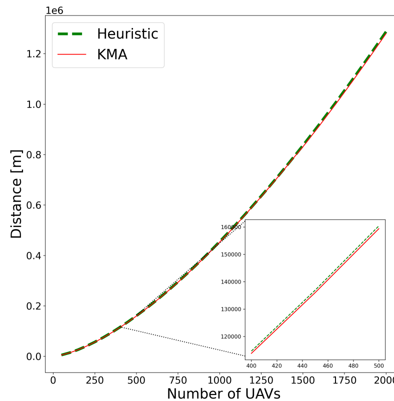


Figure 4.6: The total distance travelled by all UAVs using the heuristic and KMA for the linear formation.

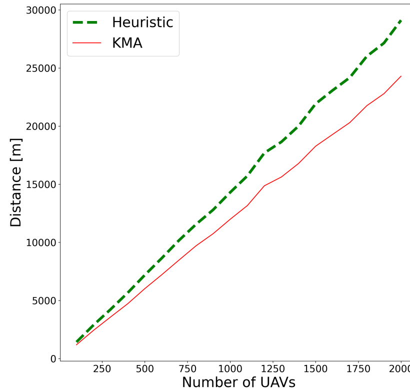


Figure 4.7: The total distance travelled by all UAVs using the heuristic and KMA for the matrix formation.

Another important metric is the number of flight paths that are crossing each other. Notice that flight paths that cross other flight paths result in two UAVs being potentially too close to each other, which might provoke a crash. Hence, an assignment that separates the flight paths is considered better. It will also aid in the take-off process, which we will discuss in the next chapter. We measured the distances between the flight paths and counted the number of times two UAVs are within 5 meters (a typical GPS error) of each other. The results are shown in Figures 4.8, and 4.9. As shown, the KMA performs better for all formations. In some cases, such as the circle, and the matrix formation, it nearly avoids all collisions. In the linear formation, it is still performing better than the heuristic; however, the number of collisions are (in general) still quite high. This is due to its one dimensional property, which forces all the UAVs to move towards similar directions, and hence there are more possible collisions.

Now that we have tested the assignment algorithms for scalability issues, we also want to test them for versatility. In the previous experiments, we used three different formations (linear, matrix, and circular). These formations are all symmetric and quite regular. However, in real world applications, the chosen formation might adopt another shape. Since we have seen that the formation does influence the calculation time of the KMA and the number of flight paths crossing, it is important that we also examine this behavior for more irregular formations. Hence, we changed our regular flight formations and made them irregular in two ways. First, the minimal distance between the UAVs in the air is smaller than on the ground. This forces the UAVs to come closer during takeoff, whereas normally

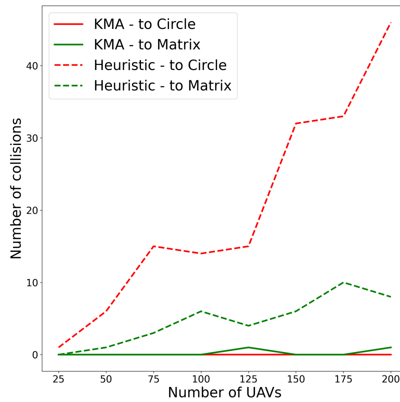


Figure 4.8: A comparison of the heuristic vs. the KMA algorithm in terms of potential collisions when varying number of UAVs (Circle and Matrix formations).

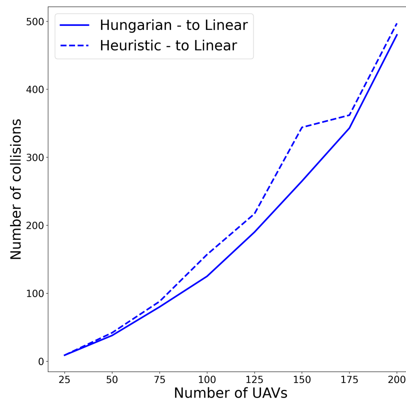


Figure 4.9: A comparison of the heuristic vs. the KMA algorithm in terms of potential collisions when varying number of UAVs (Linear formation).

they spread out during that process. We also shifted the entire flight formation in the x and y directions. In this way, the center of the ground location and the center of the flight formation are not aligned. An example is given in Figure 4.10. For this scenario, we used 25 UAVs, which were placed on the ground in a linear formation with a minimum distance of 50 meters between them. In the air, we

experimented with three formations (matrix, circle, and linear). The minimum distance between the UAVs was reduced to 10 meters. For each formation we performed one experiment without a shift of center, one with an offset of 50 meters in the x direction, and one with an offset of 50 meters in the y direction. The average results combining all these three experiments are shown in Table 4.1. The results are similar to the results obtained with the regular formations. As we can see, the heuristic is faster than the KMA. However, since the size of the swarm is not too large, both algorithms still execute within an acceptable time. As expected, the heuristic will increase the total distance travelled. In order to portray this in a meaningful manner, we calculated the additional distance travelled. In the cases of the irregular matrix and circle formation, this additional distance only represents a small percentage. However, in the case of the irregular linear formation, there is a significant difference of almost 15%. Furthermore, the number of potential collisions is (again) much lower when using the KMA. In these specific cases, there were actually no potential collisions while using the KMA, which makes the KMA a safer option. This is especially true in the case of the irregular linear formation where, when using the heuristic, 95 flight paths were crossing each other.

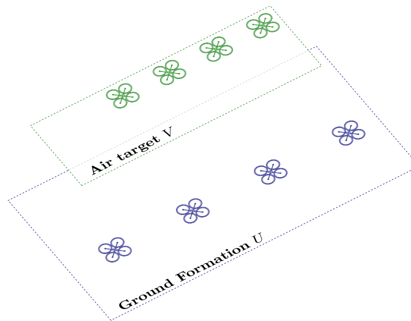


Figure 4.10: An example of the first irregular formation, where the minimal distance between UAVs is closer in the air than on the ground, and the centers of the formations are not aligned.

Besides making the airborne formation irregular, we also experimented with an irregular ground pattern. As shown in Figure 4.11, the UAVs on the ground are now split up into two clusters. One cluster has intentionally more UAVs than the other. In our case, the larger cluster consisted of 20 UAVs, and the small cluster has only 5 UAVs. The distance between clusters was relatively large. We designed the formation in such a way that the air formation lies in between the two clusters on the ground. This ensures that all the UAVs have to move towards the center.

Table 4.1: Comparison between KMA and Heurisitic for irregular air formations

	Linear	Matrix	Circle
Calculation time [ms]			
KMA	4084	4616	3819
Heuristic	2274	2070	2210
Additional distance travelled[%]	14.7	4.3	2.2
Nr. potential collisions			
KMA	0	0	0
Heuristic	95	15	9

In all our experiments, the minimal distance between the UAVs on the ground was 50 meters, and in the air 10 meters. The calculation time and the additional distance travelled values are similar to the previous experiment. The number of collisions, however, has changed, as shown in Table 4.1. The general trend is still the same, and the heuristic leads to a lot of flight paths crossing. However, the main difference between this scenario and the previous one is that, here, even when using KMA, the chances of collision are not null. In particular, for the specific case of the linear formation, we find that there are many collisions (23).

Figure 4.11: Example of an irregular ground pattern.

Table 4.2: Comparison between KMA and Heuristic for irregular ground patterns

	Linear	Matrix	Circle
Nr. potential collisions			
KMA	23	0	2
Heuristic	78	33	13

4.3 Summary

In order to manage a swarm of UAVs safely, and efficiently, various steps have to be carried out. The first step, in any swarm application, is deciding which UAV on the ground goes to which place in the target aerial formation. The final objective is to make sure that the take-off process is quick and safe. Hence, we need to find an algorithm that assigns the UAVs their airborne position in such a way that the total distance travelled by all aircraft is reduced. To this end, we implemented and tested three different algorithms. Our first algorithm uses a brute-force technique. However, since there are many possible assignments, we quickly figured out that this brute-force technique does not scale well. Starting from as little as 10 UAVs, the calculation time is already excessively long. In order to reduce this calculation time, we simplified the assignment problem and created a heuristic. This heuristic is able to assign the airborne positions very quickly, and even for swarms up to 2000 UAVs it only takes a few seconds to calculate an assignment. However, the assignment returned by the heuristic is not the optimal assignment, and therefore the UAVs have to fly greater distances. In most cases, this increase in distance is only a small percentage of the total distance. However, in the case of a linear airborne formation, there is a significant increase of up to 23%. Therefore, we searched for an algorithm that could provide us the optimal assignment, but was still able to do that in a reasonable timeframe. The fastest algorithm that is currently available is the Kuhn-Munkres algorithm (KMA). So, we implemented this algorithm in our simulator ArduSim, and compared its performance with our heuristic. The results clearly show that, when using the KMA, the total distance travelled by all UAVs is reduced. We also observed that, while using the KMA, the number of flight paths crossing is reduced. This is an important safety metric because it indicates the number of possible collisions. These collisions will need to be avoided during the actual takeoff. We could also observe that the KMA is slower than our heuristic, but the effects are only significant when the swarm becomes very large. In our experiments, we tested extensively for scalability, and we show that the KMA can be used for swarms up to 500 UAVs. If the swarm becomes any larger, the calculation time becomes excessively long (tens of minutes). For

all these experiments we used regular formations (i.e. linear, matrix, and circle). However, in real-world applications, this might not be the case, and, therefore, we also experimented with irregular formations. The results obtained followed the same trend. Hence, we can say that the KMA is the most adequate assignment algorithm in the majority of the cases. Only when one wants to use a very large swarm, the calculation time of the KMA becomes excessively long, and in that case our fast heuristic is a good alternative.

Chapter 5

Taking off

In the previous chapter, we outlined how airborne positions are assigned to a swarm of UAVs. We provided different algorithms to calculate this assignment, and came to the conclusion that, in most cases, the KMA is the most adequate algorithm. During our experiments, we also noted that, given a certain assignment, some flight paths may cross. Therefore, we cannot take off all the UAVs of the swarm at the same time, as this would possibly result in a catastrophic crash. Hence, in this chapter, we discuss and propose various procedures to take off all the UAVs in a safe manner.

5.1 Analysis of possible take-off strategies

Our objective is that the UAVs reach their airborne position as fast as possible, while remaining at a safe distance from other UAVs at all times. To this end, we provide three different procedures to take off a UAV swarm. We start with a sequential procedure, which is very simple and secure, but also very slow. Above all, we created this procedure as an easy reference to assess the performance gains of other approaches. In particular, we improved this procedure for a fast-sequential procedure, which allows a few UAVs to take off at the same time. As the name implies, the fast-sequential procedure will reduce the take-off time. However, the best results will be achieved when most of the UAVs take off at the same time. Hence, we created a semi-simultaneous takeoff procedure in which we first calculate

which UAVs might collide, and then create groups of UAVs that can take off together (i.e. those that do not collide).

All of our procedures are based on the master-slave pattern. In this pattern, the master will instruct the slaves what they have to do. We always choose the master to be in the center of the formation, since this will improve the network performance (i.e. the distance between master and slave is smaller).

5.1.1 Sequential

The sequential take-off procedure is a rather simple one. Nevertheless, due to its simplicity, it is often used for small swarms. In this procedure, the master UAV first calculates the assignment (as detailed in chapter 4). After that calculation, the master obtains a list defining where all the slaves have to go (their aerial positions). The master will then sort that list in order of flight distance, i.e. the slave that has to fly the furthest goes first, and the one that has to fly the shortest goes last. The master will then iterate through this list. It will send a message to the corresponding slave, instructing it to take off to its airborne position (obtained by the assignment). The slave will respond that it is taking off, and move towards that GPS location. For safety purposes, the UAV will first take off vertically up to a certain altitude (typically 10 meters). Afterward, it will move diagonally towards its airborne location. Once arrived at that location, the slave will inform the master that it has arrived. When the master receives that message, it will instruct the next slave in the list to take off. This process is repeated until all the slaves have reached their target positions. Finally, the master also goes towards his own airborne position, and the take-off procedure is then considered to be complete.

Since we start with the UAV that has to fly the furthest, there is no possibility for a UAV to collide with another UAV (that is hovering). However, we have to wait until the UAV reaches its airborne location, which might take a while. As a result, this procedure will take a considerable amount of time to complete, especially for large swarms.

5.1.2 Fast-sequential

In order to improve on the take-off time associated to the sequential procedure described above, we have proposed a fast-sequential take-off procedure. This procedure is still based on the sequential procedure, but improves it by reducing the time gap between consecutive UAV take-offs. The first steps of the fast-sequential procedure are the same as for the sequential procedure. Again, the master obtains the position assignments for the target flight formation, and it sorts them (based on travel distance) in a list. Then, it will instruct the slaves,

one-by-one, to take off and move towards their airborne position. Each slave will first take off vertically (for safety reasons), and then move diagonally towards it airborne location. However, in this approach, the slave sends a message to the master once it has completed the initial vertical ascend. At this moment, the master will already instruct the next slave to take off. This procedure is repeated until all the UAVs have reached their airborne position. This procedure allows a UAV to takeoff, while other UAVs are moving diagonally towards their airborne position. Since, while using this procedure, various UAVs are moving at the same time, the takeoff time is reduced w.r.t. the purely sequential procedure.

5.1.3 Semi-simultaneous

Although the fast-sequential take-off procedure already represents some improvement, we can decrease the take-off time even further when we allow multiple UAVs to start their maneuvers at the same time. However, this approach can lead to collisions, which we will need to prevent beforehand. Hence, in our semi-simultaneous take-off procedure, we will first detect if there will be a conflict in the flight paths during take-off. Based on that information, we create batches (i.e. groups) of UAVs that can safely take-off together. Finally, once obtained that information, the master will go over the list of batches, and instruct all UAVs in a batch to take-off and move towards its airborne position.

5.1.3.1 Collision detection algorithms

In order to detect conflicts in the flight paths of the UAVs we have created two algorithms. Our first algorithm, named Collisionless Swarm Take-off Heuristic (CSTH), is a heuristic that calculates the distance between some points on the flight paths of a pair of UAVs. We start with a baseline implementation CSTH, and then explain two different improvements that decrease the calculation time so as to improve performance. Our other algorithm, called Euclidean distance-based Collisionless Swarm Take-off (ED-CST), calculates the minimal distance between the two flight paths (based on the Euclidean distance between the two lines). It is important to note that, in both algorithms, we are assuming that the flight paths are straight lines (in 3D). This is however not a severe restriction, since this is almost always the case for VTOL UAVs.

We start with the most basic version of our CSTH. Although we will quickly discuss ways to improve it, this baseline version will allow us to define some basic concepts and, in turn, it will serve as a reference to see the impact of our improvements. In our CSTH algorithm, we discretize the flight path of each UAV. We consider each flight path as a series of points. These points are determined

by calculating the (normalized) direction vector between the airborne position (p_a) and ground position (p_g), and displacing from the ground position up to the airborne position using this direction vector. In our approach, we scale the direction vector by a scalar number, which we call the granularity (G). Depending on the granularity, we have more or less points between p_a and p_g. In Figure 5.1, we can see an example of intermediate locations in the trajectory that a drone must follow to reach its aerial destination. As can be seen in the image, the distance between two consecutive points will depend on the actual granularity value adopted.

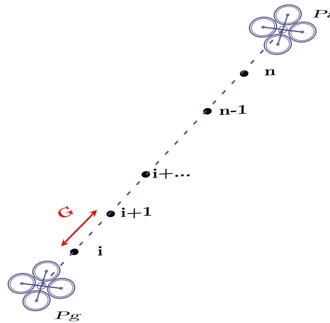


Figure 5.1: Example of a set of intermediate positions in the take-off flight path.

In our CSTH algorithm, we will use each intermediate position of one UAV to check the distance towards each position of another UAV. If this distance is shorter than a certain safety margin, we list this pair of UAVs as a pair that can potentially collide. Once we have detected a collision (in this pair) we no longer need to calculate the distances between the other points. In Figure 5.2, we illustrate this process. Of course, calculating the distances between all the points is a computational, expensive process. To make the process faster, we can check fewer points, which corresponds to a larger granularity. However, this will also increase the possibility of missing a collision. Thus, in the experiments, we will examine which granularity can be used.

In algorithm 5.1, we provide the pseudocode for our CSTH algorithm. It requires a prior assignment, which we already discussed in chapter 4. It will use that assignment to get the ground and air position of all the UAVs. Then, for each UAV, it will check the distance between its current intermediate point, with all the points of the other UAVs flight path. Since, the distances calculated between drone i and j will be exactly the same as the distances between drone j and i, we do not need to check those combinations. To keep track of which UAVs we already checked, we use the list “uncheckedUAVs”.

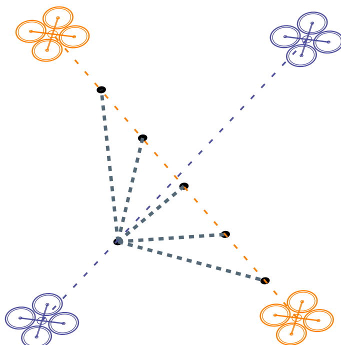


Figure 5.2: Comparison between intermediate positions in the flight trajectories of two drones.

Due to the large amount of points that need to be checked, this algorithm will be quite slow. Hence, we propose two optimizations. Our first optimization will reduce the search range, and in our second optimization, we remove trajectories that are divergent (and hence will never collide). We start by reducing the search range. We denominate it Restricted Search Range (RSR). In the baseline CSTH algorithm, we check each intermediate position of one UAV with all the other points of the other UAV. This is, however, a very costly operation, and most importantly not completely necessary. We can improve the calculation time significantly by simply restricting the number of points. In our solution, we take the altitude position of the current aircraft, and discard all the points with an altitude outside a certain range. Determining this range (similar as with the granularity) will need some testing; but once fine-tuned, it will result in a faster execution of the algorithm without loss of accuracy. In Figure 5.3 we illustrate our RSR optimization visually. As can be seen, an intermediate position of the first drone is compared with only four positions for the second one, thus reducing the number of calculations to be performed.

Going deeper into how this optimization works, the list with the identifiers of all the drones participating in the swarm mission is traversed in the same way as in the first version. For each of the intermediate points along the path of the drone under analysis, we obtain the value of its Z coordinate (equivalent to its altitude). To this value, we will add and subtract the value of the range parameter that the user has previously assigned to obtain the minimum and maximum values of our search interval. Furthermore, we have to take into account that the minimum and maximum values of the interval must be equal to or less than the point p_a, and higher than p_g. Once the altitude range is set, we only compare points within this

Algorithm 5.1 CSTH(assignment)

```
Require: assignment
 1: uavIDs = uncheckedUAVs = assignment.getIDs()
 2: for id in uavIDs do
 3:    uncheckedUAVs.remove(id)
 4:    position = assignment.getGround(id)
 5:    air = assignment.getAir(id)
 6:    while distance3D(position, air) > granularity do
 7:       position = displace(position)
 8:       for nextUAV in uncheckedUAV do
 9:          nextPostion = assignment.get(nextUAV)
10:          nextair = assignment.getAir(nextUAV)
11:          while nextpos.distance3D(nextair) > granularity do
12:             nextPosition = displace(nextPosition)
13:             if distance3D(position, nextPosition) <= safetyDist then
14:                collisionList.add(uavId)
15:                goToLine(1)
16:             end if
17:          end while
18:       end for
19:    end while
20: end for
21: Return: collisionList
```

range of values. Lastly, the system used to check for collisions will be the same as the one discussed in its initial version. In algorithm 5.2, we show the updated version of our algorithm.

Our second optimization, Divergent Trajectory Detection (DTD), works (in parallel) on reducing the number of drones to be checked. When we have to verify fewer pairs of UAVs it will, of course, result in a faster execution time. Since we are assuming that all the UAVs will take off in a straight line, we can determine quickly whether the two flight paths are diverging or not. In the case that two flight paths are diverging, the distance between the points will only increase. Hence, we will never find a distance smaller than the safety distance, so we can discard this pair of flight paths completely. To implement this idea we will need to obtain, from the two aircraft whose paths are under comparison, their first two consecutive intermediate positions, and calculate the distance between those points. We can then find two different situations depending on the values of the resulting distances:

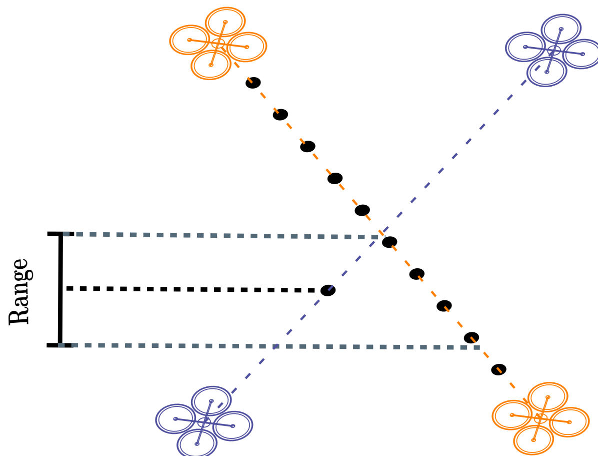


Figure 5.3: Overview of the CSTH+RSR approach.

(i) the distance between the points increases (i.e. the flight paths diverge), or (ii) the distance between the points decreases (i.e. the flight paths converge). In the first case, we can directly determine that the two UAVs will not collide. In the second case, we will need to check for a collision as before. In algorithm 5.3, we show the updated version of our algorithm.

The two optimizations we have presented separately work based on two different principles: our first optimization (i.e. RSR) reduces the number of comparisons between two UAV trajectories, and our second optimization (i.e. DTD), reduces the number of pairs to compare. Hence, we are able to combine them and reduce the calculation time even further. Such improved version has been named as CSTH+RSR+DTD, and is detailed in algorithm 5.5. In this version, we first make a list containing those UAVs whose trajectory is divergent with respect to the drone that is being examined in the first loop. Next, we determine a range of values from the height of the first intermediate position of the aircraft. Finally, we compare this position with the rest of the locations of the other drones that are within the range of values created. If a potential conflict is detected, the affected drones are stored in the collision list.

Our CSTH+RSR+DTD algorithm will be able to calculate quickly which UAVs may collide. However, it is a heuristic and, depending on the granularity adopted, it might miss some collision. Hence, in order to validate our CSTH+RSR+DTD algorithm, we also need to know the exact solution. This will allow us to fine-tune the CSTH+RSR+DTD algorithm (i.e. choose a granularity and search range).

Algorithm 5.2 detectCollisionCSTH+RSR(assignment)

```
Require:  assignment
 1: uavIDs = uncheckedUAVs = assignment.getIDs()
 2: for id in UAVIDs do
 3:    uncheckedUAV.remove(id)
 4:    position = assignment.getGround(id)
 5:    air = assignment.getAir(id)
 6:    while distance3D(postion, air) > granularity do
 7:       position = displace(position)
 8:       for nextUAV in uncheckedUAV do
 9:          nextPostion = assignment.get(nextUAV)
10:          minz = min(minHeight,position.z - range)
11:          maxz = max(maxHeight,position.z + range)
12:          while nextPosition.z <= maxz do
13:             if nextPosition.z <= minz then
14:                nextPosition = displace(nextPosition)
15:                continue
16:             end if
17:             if distance3D(position, nextPosition) <= safetyDist then
18:                collisionList.add(uavId)
19:                goToLine(1)
20:             end if
21:             nextPosition = displace(nextPosition)
22:          end while
23:       end for
24:    end while´
25: end for
26: Return:  collisionList
```

Hence, we propose another algorithm (ED-CST) which will determine the minimal distance between the two flight trajectories by directly calculating the minimum Euclidean distance between the two lines. Since this approach does not discretize the flight path, it will provide us the exact answer and never miss a collision. We will first check the minimal distance between the two (infinite) lines along the flight paths. If that minimal distance is smaller than our safety margin, then we need to verify whether the points (where the distance is minimal) on the infinite lines are actually on the flight paths. If this is the case, then the pair of UAVs is added to the collision list.

Algorithm 5.3 CSTH+DTD(assignment)

```
Require: assignment
 1: uavIDs = uncheckedUAVs = assignment.getIDs()
 2: for id in uavIDs do
 3:   uncheckedUAVs.remove(id)
 4:   position = assignment.getGround(id)
 5:   air = assignment.getAir(id)
 6:   possibleCollisionList = DTD(assignment, id, uncheckedUAVs)
 7:   while distance3D(position, air) > granularity do
 8:     position = displace(position)
 9:     for nextUAV in possibleCollisionList do
10:       nextPostion = assignment.get(nextUAV)
11:       nextair = assignment.getAir(nextUAV)
12:       while nextpos.distance3D(nextair) > granularity do
13:         nextpos = displace(nextpos)
14:         if distance3D(position, nextpos) <= safetyDist then
15:           collisionList.add(uavId)
16:           goToLine(1)
17:         end if
18:       end while
19:     end for
20:   end while
21: end for
22: Return: collisionList
```

We start by calculating the minimal distance between two infinite 3D lines. In general, three different cases are possible: (i) the lines are parallel and the distance between the lines will always be the same, (ii) the lines intersect at one point and the minimal distance is zero, (iii) the lines cross. The easiest case is checking whether the trajectories of two drones are parallel. Here, we only have to verify that the vectors of each of the drones point in the same direction, that is, that their three coordinates are exactly the same. Although it is the least likely case, in such a situation we would confirm that there is no danger of collision between said drones as long as the distance between lines exceeds the safety threshold.

For the remaining cases, which are the most typical, we use the following formula to calculate the minimum distance between two three-dimensional lines:

$$d(r,s) = \frac{|[\vec{v_r}, \vec{v_s}, \vec{PQ}]|}{\vec{v_r} \; \times \; \vec{v_s}} \tag{5.1}$$

Algorithm 5.4 DTD(assignment, id, uncheckedUAVs)

```
1:  uav1_p1 = assignment.getGround(id)
2:  uav1_p2 = displace(uav1_p1)
3:  for uav_i in uncheckedUAV do
4:     uav_i_p1 = assignment.getGround(nextId)
5:     uav_i_p2 = displace(uav_i_p2)
6:     d1 = distance3D(uav1_p1, uav_i_p1)
7:     d2 = distance3D(uav1_p2, uav_i_p2)
8:     if d1 > d2 then
9:        possibleCollisionList.add(uav_i)
10:    end if
11: end for
12: Return possibleCollisionList
```

In this formula, the two lines in the 3D space are represented by letters r and s, while P refers to a point on the line r (p_{g_r}), and Q to another point on the line s (p_{g_s}). In the numerator we must calculate the determinant composed of the two vectors for the two lines, and the vector defined through points P and Q. In the denominator we calculate the cross product of the vectors of both lines. Once both values are obtained, we divide them to obtain the minimum distance between both lines.

Once we know the minimal distance between infinite lines r and s, determining whether there is an actual collision requires checking: (i) if the distance is smaller than the safety distance, and (ii) if this minimal distance occurs within the segment of the flight path. So, we need to calculate the points that correspond to this minimal distance on both flight trajectories, and verify if these points are above ground level and below the airborne position. In order to calculate these points, we create a line that is perpendicular to the lines of each of the UAVs under comparison. This way, the points found on each of the lines are the ones that actually achieve the minimum distance between them. Figure 5.4 shows the new line segment representing the minimum distance between lines r, and s in a 3D space.

To obtain the required points, we need to perform a set of operations. First, we create the equations of the lines formed by the trajectories of the two UAVs we are comparing. After this, the distance between them will be determined as follows:

$$\begin{cases} Line\,1 : r = p_{g_1} + t_1 \cdot d_1 \\ Line\,2 : s = p_{g_2} + t_2 \cdot d_2 \end{cases} \tag{5.2}$$

Algorithm 5.5 CSTH+RSR+DTD (assignment)

```
Require: assignment
 1: uavIDs = uncheckedUAVs = assignment.getIDs()
 2: for id in uavIDs do
 3:   uncheckedUAV.remove(id)
 4:   possibleCollisionList = DTD()
 5:   position = assignment.getGround(id)
 6:   air = assignment.getAir(id)
 7:   while distance3D(position, air) > granularity do
 8:     position = displace(position)
 9:     for nextUAV in possibleCollision do
10:       nextPostion = assignment.get(nextUAV)
11:       minz = min(minHeight,position.z - range)
12:       maxz = max(maxHeight,position.z + range)
13:       while nextPosition.z <= maxz do
14:         if nextPosition.z <= minz then
15:           nextpos = displace(nextPosition)
16:           continue
17:         end if
18:         if position.distance3D(nextPosition) <= safetyDist then
19:           collisionList.add(uavId)
20:           goToLine(1)
21:         end if
22:         nextPosition = displace(nextpos)
23:       end while
24:     end for
25:   end while
26: end for
27: Return: collisionList
```

The letters p refer to the points where the UAVs start the diagonal displacement, while the letters d belong to their normalized vectors. In each of these equations, we find an unknown, symbolized by the letter t. Second, we obtain the vector perpendicular to both lines by taking the cross product of the vectors of the two lines:

$$n = d_1 \times d_2 \tag{5.3}$$

Then, we calculate the cross product between each of the vectors of the lines with respect to the vector obtained in the previous step. In this way, we create a

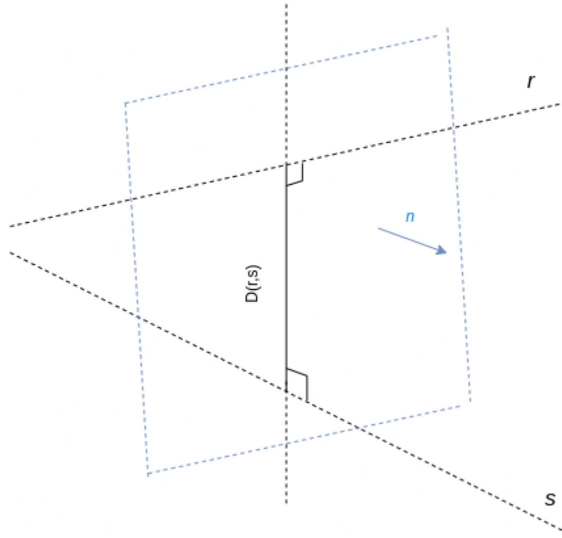


Figure 5.4: Line segment corresponding to the minimal distance between two lines in a 3D space.

plane that is perpendicular to both lines.

$$\begin{cases} n_1 = d_1 \times n \\ n_2 = d_2 \times n \end{cases} \tag{5.4}$$

Therefore, the intersecting point *c1* of Line 1 with the above-mentioned plane, which is also the point on Line 1 that is nearest to Line 2, is given by Equation (5.5). The point (*c2*) on Line 2 nearest to Line 1 is calculated similarly.

$$\begin{cases} c_1 = p_{g_1} + \frac{(p_{g_2} - p_{g_1}) \cdot n_2}{d_1 \cdot n_2} \times d_1 \\ c_2 = p_{g_2} + \frac{(p_{g_1} - p_{g_2}) \cdot n_1}{d_2 \cdot n_1} \times d_2 \end{cases} \tag{5.5}$$

Now that we have the coordinates, we need to check whether those points are on the flight path. We can find three different cases:

1. If the coordinates of both points are above p_g and below p_a, we confirm that there is a potential collision;

2. If both points are outside this range, we say that there is no danger of collision.

3. If one of the two points is within the range, and the other is not (either below the p_g, or above the aircraft's p_a), the point outside the range will be replaced by the nearest point within range. We then check the distance between both points, and if it is smaller than the safety distance, we confirm that there is a collision.

Notice that our ED-CST algorithm is able to calculate the exact minimal distance between two flight paths. However, due to the many operations required, it will typically be slower than the CSHT+RSR+DTD algorithm.

5.1.3.2 Batch generation strategy

Now that we have two different algorithms to determine whether the UAVs may collide, we still need to groups the UAVs that do not collide in different batches. Given the list of collisions obtained by either of the two collision detection algorithms proposed, we now detail an algorithm that is able to define these batches. In this way, aircraft belonging to a same batch are allowed to take off simultaneously, without any conflict between their trajectories.

We visualize our algorithm using a flowchart in Figure 5.5; notice that it uses recursion in order to fill the batches. In particular, our algorithm needs a list of all the UAVs (which can be easily obtained from the assignment), and also the list of collision pairs. For each UAV, we check if it appears in the list of collisions. If this is the case, we directly insert it into a group called (G2). This group is a temporary group that contains all the UAVs that do not maintain a safety distance. If the UAV does not appear in the collision list, it will be placed in group G1. Since no UAV in G1 will collide, they can be put in the first batch. Now we need to split the UAVs in G2 in various batches. In the best-case scenario, G2 is empty, and we can end the algorithm directly. Similarly, if G2 contains only one UAV, it is directly placed in the second batch, and we can end the algorithm. In most of the cases, however, G2 will maintain more than one drone. This means that we need more than two batches to separate all the UAVs in collision-free groups. Using recursion, we can simply run our algorithm again, but this time we only use the UAVs that are in G2. They will again be separated into groups, and eventually placed in the batches. We stop the recursion when G2 is empty or only contains one UAV.

Figure 5.6 shows an example of the batch generation mechanism in a swarm composed by four aircraft. Using our collision detection algorithm, we obtained the information that there is a possible risk of collision between the first and the second UAV, as well as between the third and the fourth UAVs. We start the batch process with UAV 1. Since there is a risk of collision, UAV 1 is placed in

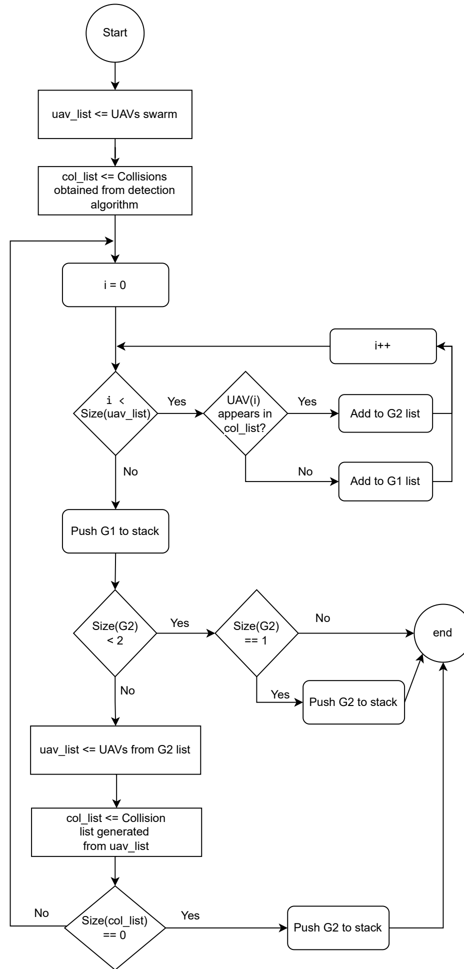


Figure 5.5: Flowchart representing the batch generation mechanism.

batch B. We continue with UAV 2, this UAV has no risk of collision (we already solved the collision with UAV 1). Therefore, it is placed in batch A. Next is UAV 3; this UAV collides with UAV 4 and, therefore, it is placed in batch B. Finally, UAV 4 has no risk of collision, hence it is placed in batch A. This ends the first iteration of the batch process. We start the second iteration of the batch process by going over all the UAVs in batch B, and confirm that they do not collide. Since

UAV 1 and UAV 3 do not collide, we do not have to create additional batches, and so we end the batch process.

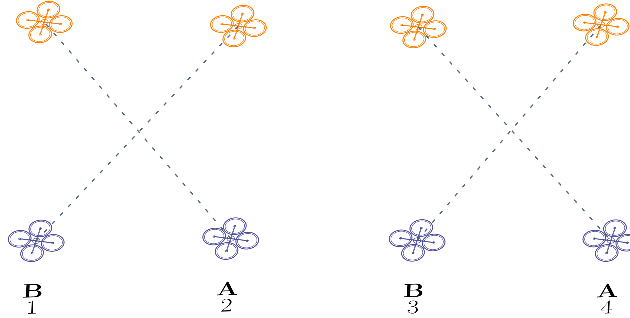


Figure 5.6: Example of take-off batch grouping (A, B) using our algorithm.

5.1.3.3 Flight path of the UAV

Finally, we briefly have to address the flight path followed by a UAV in our semi-simultaneous take-off procedure. For the sake of improved safety, the flight path will be split into three straight-line segments, which are shown in Figure 5.7. First, the UAV will ascend vertically up to a safety altitude SA_1. We do this in order to clear it from surrounding obstacles close to the ground, or even people located nearby. Then it moves diagonally, until it reaches its position SA_2. Finally, the UAV moves upwards to safely fit in its final position. In this way, we can avoid collision between a hovering UAV and one that is moving diagonally in cases of extremely long paths.

5.2 Experiments & results

In the previous section, we presented three different approaches for taking off a swarm of UAVs: sequentially, fast-sequentially, and semi-simultanously. We will now present performance results to assess their usability through different experiments. We start by testing the influence of the granularity in our CSTH algorithm. We then perform a similar test to examine the influence of the search range for our RSR optimization. Afterward, we compare the various versions of the CSTH algorithm. After that, we compare our ED-CST algorithm to the CSTH+RSR+DTD algorithm. Finally, we compare the three different take-off procedures.

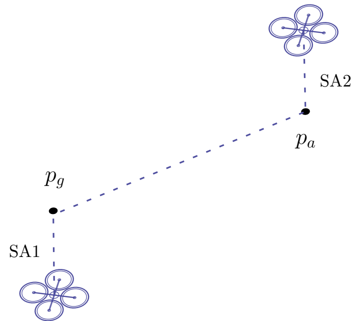


Figure 5.7: Illustration of the flight path to be taken for a UAV during take-off.

5.2.1 Influence of the granularity

As explained earlier, the granularity has an important influence on the performance of the CSTH algorithm. In principle, we would like the granularity to be as coarse as possible since this will reduce the number of points we need to check, thereby reducing the calculation time. Yet, we cannot increase the granularity too much because then the possibility of missing a collision will increase. Therefore, we perform an experiment (in simulation) in order to observe the impact of enforcing different granularities. In our experiment, 200 UAVs are placed randomly on the ground with a minimal distance of 10 meters between them. We use three aerial formations (circular, linear, matrix) with a minimal distance between the UAVs of 20 meters. The safety distance between the UAVs that must be maintained at all times is set to 8 meters. We start with a granularity of one meter, and increase this value by one up to 10 meters. During the experiments, we measured the calculation time, and also the number of collisions that remain undetected. We were able to calculate the undetected collisions by using our ED-CST algorithm that returns the exact answer.

Results are shown in Figure 5.8 and Figure 5.9. As expected, the calculation time can be reduced significantly by increasing the granularity. It is important to note that, in Figure 5.8, the values shown are represented on a logarithmic scale because of the substantial differences in calculation time between the three available formations. As one can see, the calculation time of the linear formation is the highest. This is for the sole reason that, while using a linear formation, the flight paths are longer. Hence, there are more points that need to be compared, and this in turn will result in a longer calculation time. In Figure 5.9 we show that increasing the granularity cannot be done without risks. Starting from a granularity of 3 m, there are already a few undetected collisions. When we increase the granularity,

the number of undetected collisions will increase. In some cases, however, the number of undetected collisions will drop slightly, as shown in Figure 5.9 with granularity 10 meters. This can happen due to the points we select on the flight path. Nevertheless, in general, the number of undetected collisions will keep increasing when the granularity is increased. Thereby, it is preventing the take-off maneuver of the UAV swarm from being reliable. Hence, in order to detect all collisions, the granularity is limited to two meters. Which still allows us to reduce the computational time overhead by half.

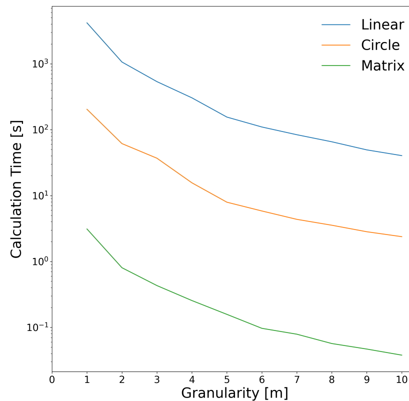


Figure 5.8: Calculation time according to their granularity.

5.2.2 Influence of the search range

In our first optimization (i.e. RSR) we improve the calculation time of our algorithm by restricting the search range. Similar to the granularity, the actual search range we use will influence the calculation time and the number of undetected collisions. The simulation parameters used in these tests are exactly the same as in the first experiment. In addition, to obtain the intermediate positions, the value of the granularity resulting from the previous experiment (2 m) has been used. In Figure 5.10 we show the results from our experiments. As expected, for a very small search range, there will be undetected collisions. When we increase the search range, the number of undetected collisions decreases. We can observe that, in the case of a circular formation, we never miss a collision, and, in the case of the linear formation, only a few are missed. This is most likely due to the fact that there are fewer collisions to detect in the first place. Of course, once the search range

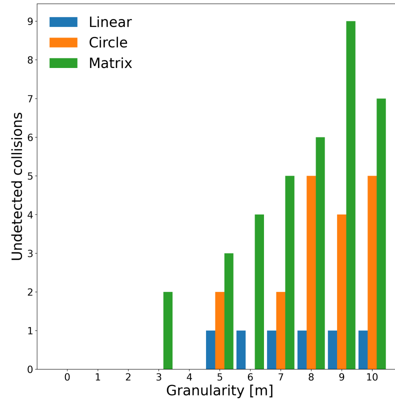


Figure 5.9: Number of potential collisions that would remain undetected when increasing the granularity value.

exceeds the value of the safety margin (in this case 8 meters) we will detect all the collisions. Hence, we can safely say that, independently of the formation, the search range must be set as equal to the safety margin. Depending on the altitude of the airborne position, this behavior can reduce the calculation time significantly.

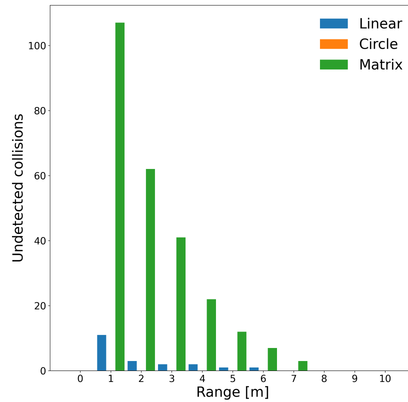


Figure 5.10: Undetected collisions according to interval size used.

5.2.3 Influence of optimization of CSTH

Next, we want to measure the influence of our optimizations. Therefore, we perform a similar experiment as before. However, in this case, we also change the number of UAVs. In this way, we can also make statements about the scalability of our algorithm. We performed this experiment for all different versions: CSTH, CSTH+RSR, CSTH+DTD, CSTH+RSR+DTD. Although, we know CSTH+RSR+DTD will be the fastest, we perform this experiment to assess the relative impact of our optimizations.

We start with the linear formation. As already shown before, this formation in general takes more time to calculate (due to the long distance). Hence, for this formation, our optimization is the most important. As shown in Figure 5.11, we can see that, up to 75 UAVs, the influence is minimal. For larger swarms, however, the influence becomes visible. Our baseline version of CSTH, with calculation times in the tens of minutes, is simply too slow. However, using our optimization, we can reduce this time significantly. In this formation, we can also observe that our RSR optimization outperforms the DTD optimization. This is mainly because, in a linear formation, half of the UAVs will move in the same direction. Hence, there are not many flight paths that are diverging. Of course, the combination of both yields the best results.

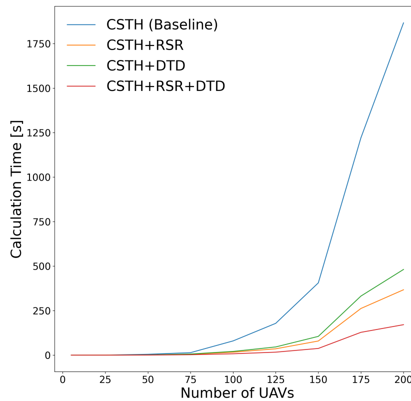


Figure 5.11: Calculation time for the linear formation when varying the number of UAVs in the swarm.

We now, proceed with the circle formation. The results are shown in Figure 5.12, being very similar to the ones we observed for the linear formation. However, there

are two noticeable differences. First of all, the calculation time is, in general, a lot faster. As explained earlier, this is related to the total distance the UAVs have to fly. Furthermore, we can also observe that while using a circle formation, the DTD optimization outperforms the RSR optimization. In this formation, there are more flight paths that are diverging, and therefore we can perceive more benefits from this optimization.

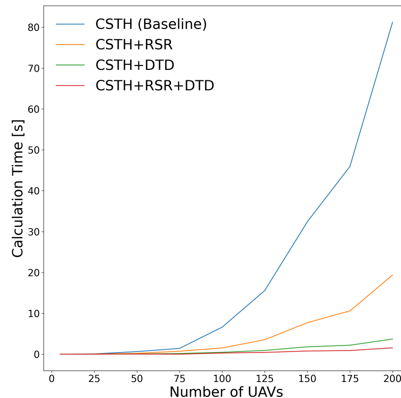


Figure 5.12: Calculation time for the circular formation when varying the number of UAVs in the swarm.

Finally, we perform the experiment with the matrix formation. Results are shown in Figure 5.13. Since, this formation is very compact, the calculation time is even shorter. Besides that, the results stay inline with our previous findings.

5.2.4 Comparison ED-CST and CSTH+RSR+DTD

Now that we have examined the impact of the two optimizations proposed for the CSTH algorithm, we want to compare it with respect to our ED-CST algorithm. Therefore, we performed a similar experiment as before, using both the CSTH+RSR+DTD, and the ED-CST algorithm. From our results, shown in Figure 5.14, and Figure 5.15, we can observe that our heuristic is faster. Notice that this is quite normal because the ED-CST algorithm provides us with the exact minimal distance between the two flight paths, whereas our heuristic obtains just an approximation. However, our objective is to be able to take off multiple UAVs at the same time, and with the use of our heuristic we are able to do that as well. So, in most cases, we recommend using the CSTH+RSR+DTD algorithm. However,

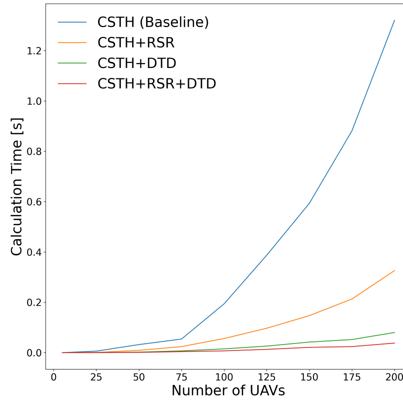


Figure 5.13: Calculation time for the matrix formation when varying the number of UAVs in the swarm.

when comparing the two algorithms for the linear formation (see Figure 5.16, we observed that the ED-CST was actually faster. This is (again) due to the longer flight paths. Given this result, we must conclude that there is a tipping point where it will be faster to use the ED-CST algorithm. This tipping point will mainly depend on the length of flight paths, but also on how much the DTD optimization can be leveraged.

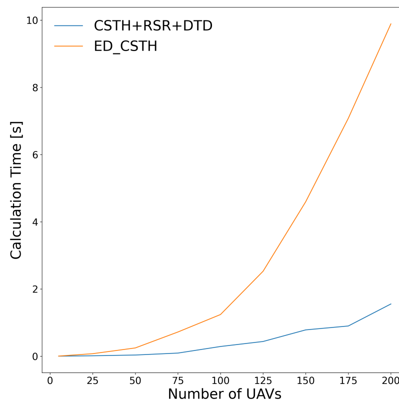


Figure 5.14: Calculation time for the circular formation.

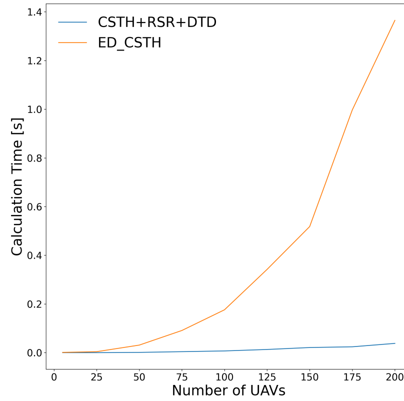


Figure 5.15: Calculation time for the matrix formation.

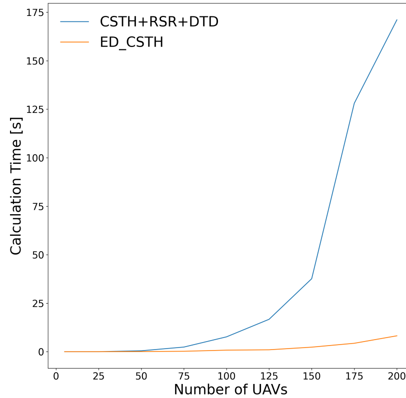


Figure 5.16: Calculation time for the linear formation.

5.2.5 Comparison of take-off procedures

In our final experiment, we compare our proposed take-off procedures. Again we will test for different formations, and for a different number of UAVs. For the sake of completeness, when comparing the different procedures, we also include the calculation time (which is only relevant for the semi-simultaneous procedure). We

start with the matrix formation. As shown in Figure 5.17, our semi-simultaneous approach reduces the take-off time drastically. When using the traditional sequential take-off procedure, it would take almost 25 minutes for 50 UAVs to take off. In this time, the batteries are already depleted. However, when using our semi-simultaneous procedure, it only takes three minutes. We can also observe that, although the semi-sequential procedure does reduce the take-off time, it does not reduce it enough.

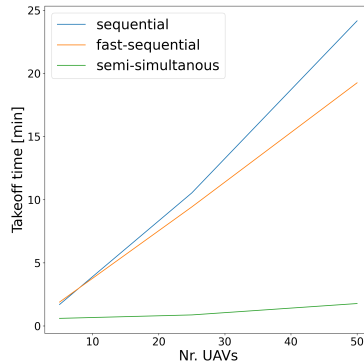


Figure 5.17: Take-off times for the matrix formation.

The matrix formation, however, is a very compact formation and, hence, the UAVs do not heave to fly far. When we compare the results with the circle formation, we can see that it takes longer for all procedures to take off all UAVs. Figure 5.18 shows us that it now takes almost 45 minutes to take off 50 UAVs using the sequential procedure. Nevertheless, using our semi-simultaneous procedure, we can reduce this time to about 5 minutes.

Finally, in the linear formation, the results are very similar. However, due to the long distance the UAVs have to fly to reach their location in the linear formation, it now also takes a long time while using our semi-simultaneous procedure. Even thought the takeoff time is reduced by a factor of 3.5, in this specific case it remains excessive.

5.3 Summary

Using the knowledge from the previous chapter, we gain awareness about which UAV on the ground needs to go to each aerial position. However, this does not

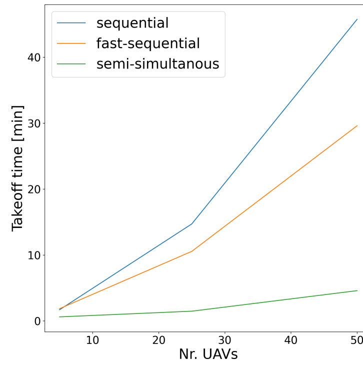


Figure 5.18: Take-off times for the circle formation.

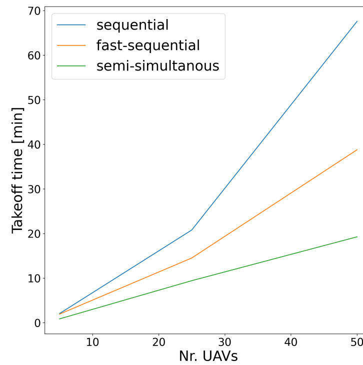


Figure 5.19: Take-off times for the circle formation.

tell us in which order we need to take off the UAVs. Hence, in this chapter, we discussed various take-off procedures. During the take-off, it is important that the UAVs maintain a safe distance from each other in order to prevent collisions. Therefore, we started with a sequential take-off procedure. This procedure is often used because it is simple to implement and very safe. However, when the swarm becomes only a little bit larger, this take-off procedure takes a lot of time. In order to solve this problem, we first introduced a fast-sequential take-off procedure. In this

procedure, UAVs are allowed to take-off with just a small time gap between them. This decreases the take-off time significantly. However, we are able to decrease the take-off time even further if we let the UAVs take-off simultaneously. The main issue that arises is that it introduces the possibility of collisions. Therefore, we create two algorithms to detect those collisions beforehand, and let the UAVs take off in groups. One of our algorithms is a heuristic, which we optimized using two different techniques. We tested our algorithms extensively, and were able to fine-tune some parameters to achieve a faster execution. We compared this heuristic with another algorithm we developed, which is based on calculating the Euclidean distance between the flight paths. Since it involves more calculations, it is generally slower than our heuristic. Yet, in our experiments, we demonstrate that, in certain cases (very long flight paths), it might actually perform better. After testing the two collision detection algorithms extensively, we compared the three take-off procedures. As expected, our semi-simultaneous take-off procedure was the fastest. Using the knowledge of this chapter, we are now able to take off a (large) swarm of UAVs safely and quickly.

Chapter 6

Maintaining the swarm coherent

In the previous chapters, we were concerned with taking off a swarm of UAVs. Now that we know how to do this safely and efficiently, we can start the actual flight. In many applications, the flight will be pre-planned and determined by a number of GPS locations that the UAVs will visit. Each of these GPS locations are commonly referred to as a waypoint, and the conjunction of waypoints as a mission. Going from one waypoint to another with one UAV is relatively straightforward. However, when using a swarm of UAVs, we need to ensure that the swarm stays coherent. For instance, if one UAV goes slightly faster than the others, over time, the swarm formation will be disrupted. Hence, some synchronization between the UAVs is necessary. Therefore, we make use of the Mission-Based UAV Swarm Coordination Protocol (MUSCOP). This protocol was initially published in [38], and makes use of the master-slave pattern to keep the swarm synchronized at each waypoint. Although this protocol is able to maintain a swarm coherent during a mission, it experiences some issues, which we address in this chapter. The original version of MUSCOP assumed that all the UAVs are accessible at all times. However, during a mission, some UAVs might become inaccessible, for instance, due to: a crash, battery depletion, network errors, etc. In the original version of MUSCOP an error in a single UAV would make the entire swarm inoperable. Our enhancement ensures that the remaining swarm elements will continue their mission. Since our work is based on the original version of MUSCOP, we will start by offering an overview of that version. Then we introduce our contribution. Afterward, we

perform experiments with the enhanced version to assess its performance, and present our results, with discussion.

6.1 The original version of MUSCOP

In this section, we present the original version of MUSCOP. We will, however, only present the parts that are necessary to understand our enhancements. A full, in-depth explanation of the original version of MUSCOP can be found in [38].

The objective of the MUSCOP protocol is to maintain a stable flight formation when a swarm of UAVs follows a preplanned mission. The protocol uses a master-slave model to synchronize the swarm at each waypoint. Only when all the UAVs arrive at a waypoint does the master issue the command to go to the next waypoint. In between the waypoints, all the UAVs are following their own mission. This mission is calculated before taking off, and it is a modification of the original mission defined by the user. In particular, the modification takes the relative position of each node in the swarm into account. Throughout the flight, messages are broadcasted periodically by both master, and slaves. Overall, there are six different types of messages:

- **Data:** This message is sent from the master to the slaves. It contains the original mission, as well as information such as position in the formation, for the slaves to calculate their own mission.
- **DataAck:** Acknowledgement messages are used so that the master knows the slave received a certain packet.
- **ReadyToFly:** Once all the slaves have received their data message and acknowledged it, the master will start sending ReadyToFly messages. These messages will prepare the slaves for the take-off.
- **ReadyToFlyAck:** The previous message is acknowledged by this message.
- **ReachWaypointAck:** Once all the UAVs have taken off and have reached the first waypoint, the slaves will start sending this message. The message contains the number of the last waypoint that has been reached.
- **MoveToWaypoint:** The master listens and waits until all the UAVs have reached the next waypoint. When this happens, the master will start sending messages of the *MoveToWaypoint* type. When a slave receives these messages, it will start flying to the next waypoint.

The listed messages are received by all the other UAVs within range. This information, along with the location of the UAV, is used in order to determine if the UAV should move to the next waypoint, or merely wait at a waypoint.

6.2 Proposed resilience mechanism

Although the above-mentioned version of MUSCOP is able to maintain the swarm coherent throughout the mission, it assumes that all the UAVs will always receive the messages *ReachWaypointAck*, and *MoveToWaypoint*. However, due to communication errors, depleted batteries, or potential crashes, a drone might not receive and/or send the messages. In that case, the entire swarm would wait indefinitely (in practice, until the batteries are depleted) at a waypoint. This behavior is of course undesired, and hence we will enhance the MUSCOP protocol. In order to do so, we must distinguish between two different cases: (i) the master can fail, and (ii) one (or multiple) slaves can fail. We start by explaining the latter, since this is the easiest case.

During the assignment (explained in 4), the master UAV creates a list of all the UAVs that are joining the mission. The same list is used by MUSCOP to verify if all of the slaves have reached the last waypoint. In order to make MUSCOP resilient against UAVs failing during the mission, it suffices to update a copy of this list. In our enhancement, every UAV maintains a list with the *ids* of the other UAVs, along with a timestamp (for each UAV). This timestamp is updated each time a message is received from that UAV. At each waypoint, this list is checked and, if the time elapsed since the last timestamp exceed a certain threshold (assigned empirically in later experiments, and referred to as *TimeToLive*), we assume that that UAV has failed. Therefore, such UAV will be excluded from the list. Once all the other "alive" UAVs have arrived at the waypoint, the master will give the order to continue the mission (through message *MoveToWaypoint*). This (simple) update works efficiently, and allows the entire swarm to continue their mission even in the worst case when all the slaves fail. However, it assumes that the master is always available.

In order to solve the issue of a failing master, we should execute various steps. First, we must detect that the master is not able to send or receive messages. We do this in the same manner as a described above, by maintaining a list of timestamps. If the master has failed, a new master should be chosen. Finally, that new master should change its role and start sending the order to move towards the new waypoint. For the best performance of the swarm, the master should be the UAV in the center of the formation. This UAV will always be the UAV that is closest to all others in terms of radio range, as this will improve network

performance. Hence, choosing the new master requires determining which UAV is closest to the center of the formation. We calculate this during the assignment phase, when the UAVs are still on the ground, since this is the safest moment to do so. We order the above-mentioned list with timestamps according to the distance to the center. Since all the UAVs are maintaining their own list, they will notice when they did not receive a message from the master. Once the timeout is exceeded, the master will be removed from the list (just as with the slaves). However, if the master is removed, the UAVs will verify if they are on top of the list (i.e. the closest to the center). If a UAV notices that it is on top of the list, it will change its role, and become the new master. The other slaves will receive the order to move towards the next waypoint as usual, and the entire swarm will complete the mission. The behavior of this protocol is formalized in Algorithm 6.1, which uses pseudocode to adequately detail our extension to MUSCOP.

It is important to note that, since this process is executed in all the UAVs, there is a possibility that the outcome is not the same for all of them. Most commonly, the UAVs in the swarm will fly close enough to each other so that they can receive all the messages that are broadcasted. However, when the distance between the UAVs becomes larger, it is possible that when a new master is chosen, not all the slaves are able to communicate with it. In that case, the only way for the swarm to complete the mission is for it to be split up in two sub-swarms. This is one of the reasons why we have chosen for all the UAVs to maintain their own list. Since, the decision to remove unavailable UAVs from the list is an individual decision, each UAV will automatically remove those UAVs that remain outside its communication range. To provide a better explanation of this scenario, an example of a flight formation is given in Figure 6.1.

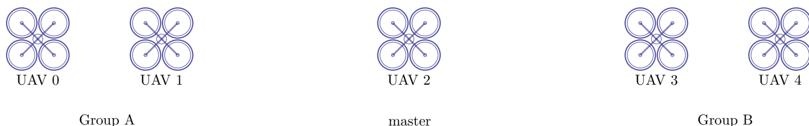


Figure 6.1: Example of a swarm splitup.

In this formation, UAV 2 will be the master, and all of the UAVs can communicate with it, hence allowing the mission to be completed without any problem. However, if for some reason the master UAV 2 fails during the flight, UAV 1 will be (in our example) the new master. Due to the large distance, UAVs 3 and 4 are unable to communicate with UAV 1. If the procedure to switch between masters was executed centrally, this situation would cause a problem, resulting in failure of the mission, or an extra time overhead. Therefore, in our solution, all the UAVs

Algorithm 6.1 UpdateSwarm(numUAVs, listOfMasters)

Require: $listOfMasters.size = numUAVs$

```
1: TimeToLive = 5s
2: Setup phase:
3: Let LastTimeUAV be a hashmap of size(numUAVs)
4: for Id in numUAVs do
5:    if Id != selfId then
6:       LastTimeUAV.put(Id, currentTime)
7:    end if
8: end for

9: Fly phase:
10: while waypoint not reached do
11:    if Message received then
12:       Id = readMessage()
13:       LastTimeUAV.put(Id, currentTime)
14:       Perform actions related to message
15:    end if
16: end while
17: while waypoint reached do
18:    for UAV in LastTimeUAV do
19:       UAVTime = LastTimeUAV.get(UAV)
20:       if currentTime - UAVTime > TimeToLive then
21:          LastTimeUAV.pop(UAV)
22:          ListOfMasters.pop(UAV)
23:       end if
24:    end for
25:    if selfId == ListOfMasters.getFirst() then
26:       IamMaster = True
27:    end if
28:    if IamMaster == True then
29:       Perform actions related to master
30:    else
31:       Perform actions related to slave
32:    end if
33: end while
```

make that decision individually. In this case, such approach will result in two independent swarms to be created, groups A and B, and both of them will continue the mission without interacting with the other.

6.3 Experiments & results

We have performed an extensive amount of experiments to validate our proposal. We started by testing how often a UAV receives messages from another UAV in the swarm using channel parameters obtained from actual real-life tests [39]. Using this experiment, we are able to set the variable *TimeToLive* (see Algorithm 6.1) to a realistic value. Then, we simulated the probability of a UAV failing in one of the three different scenarios: in-between waypoints, at a waypoint, and slightly before a waypoint. We continued our research by performing multiple experiments where we tested the loss of master(s), slave(s), and different combinations of those. To test the robustness of our algorithm, we also performed some experiments in more extreme setups (higher number of UAVs, increased distance, etc.). Finally, we experimented with swarm split-ups. For all the tests, we measured the time overhead introduced by our protocol. Each experiment, together with the results, are discussed in more detail below.

In our first experiment, we wanted to investigate the periodicity of received messages during the flight. We omitted the messages exchanged during the setup phase, since they are not used to check if a UAV is still alive, and therefore are not relevant for our analysis. The UAVs are sending a message every 200 ms; so, in an ideal environment, a UAV would receive messages with that same frequency. However, since we are using UDP broadcasts, it is possible that a message is lost. Inside ArduSim we have a communication model (based on real experiments) for a transmitter based on IEEE 802.11a, and using a 5dBi antenna. This model is used to simulate the broadcasting behavior as accurately as possible. For this experiment, we simulated two UAVs flying at different distances from each other. In particular, one UAV slowly diverts (with a speed of 1 m/s), so that the distance between them increases. While following this mission, the time when each message is received is logged. Then, using the logs, we calculated how many messages are received per second.

The results are shown in Figure 6.2. We can see that the UAVs can communicate in a range between zero, and about 1350 meters. As we expected, the number of messages received drops w.r.t. the distance between UAVs. For a single experiment, the y-values (i.e. the number of messages received) are fluctuating significantly. Therefore, we decided to show the average of 20 experiments, and, even in this case, we can still see important fluctuations. For that reason, it is important that

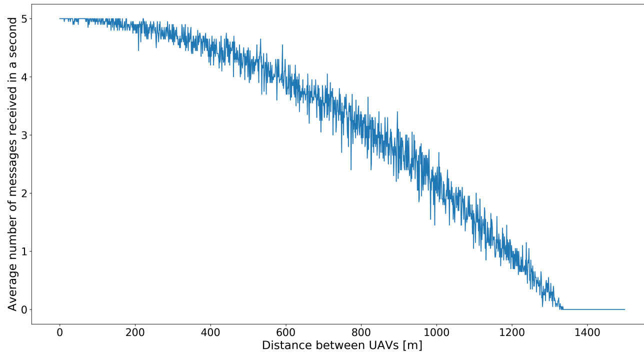


Figure 6.2: Message frequency w.r.t. distance between UAVs.

we keep parameter *TimeToLive* high enough. We want to avoid that a UAV is assumed to have failed, when actually it was a mere false-negative event. The only reason why we would like a low value for the *TimeToLive* parameter is that, in the worst-case scenario, a UAV could fail when just arriving at a waypoint. This would mean that the entire swarm would have to wait for the entire timeout period to elapse prior to continuing with the mission, hence causing an unwanted delay. Since, at this time, there is no possibility for a UAV to reenter the swarm, we tune our scheme to give priority towards reducing any false-negative events. So, we decided to set the *TimeToLive* parameter to 5 seconds, and thereby we virtually remove all false-negative cases. Furthermore, we believe that, in the unlikely event of a UAV failing exactly at a waypoint, introducing a delay of 5 seconds is a small price to pay when considering the mission as a whole.

In our experiments, we classify the location where a UAV can fail into three groups: (i) at a waypoint, (ii) just before a waypoint, and (iii) in-between waypoints. We differentiate among these types of events because the exact location where a UAV fails has a direct impact on the overall time overhead. This is because the UAVs only detect if another UAV has failed after a certain timeout has expired. At each waypoint, the swarm has to wait until all the UAVs arrive, before they are able to continue their flight. For that reason, if a UAV fails just before reaching a waypoint, the swarm will have to wait for the timeout to expire (i.e. *TimeToLive*), causing a longer delay than if that same UAV had failed at another point during the mission. Therefore, we conclude that failing just when arriving at a waypoint is a worst-case scenario. With the same reasoning, failing far away from the waypoint is a best-case situation. In this case, the UAVs will arrive at the waypoint and, since the timer has already expired, the time overhead will be significantly lower

than in the former case. For our second experiment, we want to investigate what the chances are that such an event happens. Such possibility is determined by three parameters: the distance between the waypoints, the speed of the UAV, and the value of parameter *TimeToLive*. For this experiment we will test different flight speeds (5 m/s, 10 m/s, and 15 m/s). The value of the timer is always set to 5 seconds. By varying these parameters, we can investigate how the distance between waypoints affects the probability of failure in one of the three cases, since mathematically the likelihood of a continuous random variable to take place on an exact value is zero. We could say that a UAV will never fail exactly at a waypoint. However, in our code, we can force this behavior, and therefore we include it in the other experiments for the sake of completeness. In real experiments, though, we could reduce the classification to just two cases. Since the value of the timer is set, and since we know the speed of the UAV, we can determine the boundaries associated to these cases. One might think that simply multiplying the speed by the time provides us the boundaries. However, the matter is a bit more complicated because the UAV slowly decelerates if it comes near to a waypoint. Therefore, we have used ArduSim to calculate the exact boundaries. For flight speeds equal to 5, 10 and 15 m/s, the boundaries are 18.45 m, 28.41 m, and 31.06 m, respectively. All the UAVs failing further away than that distance from the waypoint are classified as the best case scenario. To calculate the probability, we designed a script which will randomly choose a distance between zero and the maximum value, and will then decide to which case it belongs to. To be representative, all the experiments are randomly repeated multiple times. The script will start from a distance between waypoints of 1 meter. To be representative, the experiment is repeated 100,000 times before increasing the distance between the waypoints by 1 meter each time. The script will stop when the probability of failing just before a waypoint drops below 5%. The results are shown in Figure 6.3.

Of course, if the distance between the waypoints is shorter than the border distance (i.e. 18.45 m, 28.41 m, and 31.06 m), the UAV will always fail just before the waypoint. Furthermore, we can observe that the chances of failing just before the waypoint drop rapidly at the beginning, but it slows down as the distance increases, ending with an asymptotic behavior. The fact that the chances fall rapidly at the beginning is beneficial for us, because this means that the time overhead introduced by our protocol will be low in most cases.

To test if our protocol works in a wide range of cases, we have considered multiple experiments. In each experiment, we have measured how long a UAV stays at a waypoint. Then, we compared this to a flight scenario where no UAV fails. The time difference between both provides us the extra time in the experiment associated to handling UAV failures. In those experiments, four UAVs are flying by following a linear formation (flight speed of 10 m/s), with a distance of 50

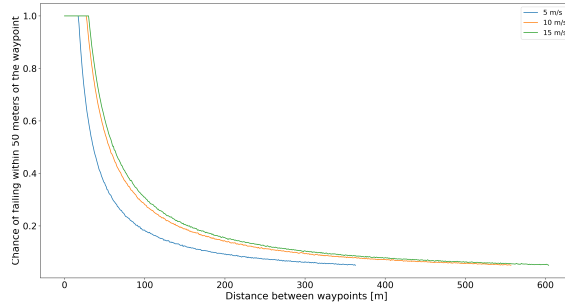


Figure 6.3: Chances of failing just before a waypoint w.r.t. the distance between waypoints.

meters between each UAV. The mission has four waypoints, where waypoint 0 is right above the take-off position, and waypoint 3 is at the landing place. We have tested five different scenarios, all of them in the three categories: at a waypoint, just before a waypoint (15 meters away), and in-between waypoints (200 meters away). This results in 15 different experiments, plus one control experiment where no UAV fails. The experiments we proposed are the following:

A : a single slave failing at waypoint 1.

B : a single master failing at waypoint 1.

C : two slaves failing at waypoint 1.

D : a master and his backup failing at waypoint 1.

E : a backup master failing at waypoint 1, and the master failing at waypoint 2.

Table 6.1 describes the first experiments, where UAVs fail at 200 m from a waypoint. We can observe that there is no extra delay introduced by our protocol, and this is because, during those experiments, the UAVs have had enough time to recognize that a UAV has failed. Once arriving at the waypoint, they can act accordingly, without any extra delay. We can also see that some values are negative; this simply means that, in such case, it was a bit faster than the control case itself, where no UAV failed. Overall the values remain small, and they are caused by ArduSim, and not by our protocol.

Table 6.2 describes the same experiments, but this time the UAVs fail at 15 meters from the waypoint. Earlier, we have established that the boundary distance

Table 6.1: Time overhead for the different scenarios at 200 m from the next waypoint.

Section	A [ms]	B [ms]	C [ms]	D [ms]	E[ms]
0	150	122	- 32	140	112
1	- 50	- 33	- 1	300	-150
2	-148	-184	-104	- 51	99
3	448	-185	197	200	-206

was of 28.41 meters. We have chosen 15 meters since that is half way in-between this threshold value. Here we can see that our protocol works as expected. The delay ranges between 0 and 5 seconds, because the UAVs fail near to the waypoint. We can also observe that the delay is unrelated to which UAV actually fails (the master or the slave), and that it is also unrelated to how many UAVs fail. In fact, the delay is only related to when and where a UAV fails.

Table 6.2: Time overhead for the different scenarios at 15 m from the next waypoint.

Section	A [ms]	B [ms]	C [ms]	D [ms]	E[ms]
0	- 77	594	20	123	75
1	2554	2993	2555	2099	3006
2	57	102	-153	-153	2101
3	- 1	301	147	147	-1

In the last set of experiments (see Table 6.3) the UAVs fail just when arriving at the waypoint, before actually sending the message that notifies neighbors they arrived to that waypoint. Therefore, in this case, the delay is the longest one, and equal to the *TimeToLive* value, which was set to 5 seconds. Also, in this case, we can observe that the delay is unrelated to which UAV fails, or to how many UAVs have failed.

Table 6.3: Time overhead for the different scenarios just when reaching the next waypoint (0 m).

Section	A [ms]	B [ms]	C [ms]	D [ms]	E[ms]
0	-198	383	234	-128	-135
1	4999	5601	5601	5383	5002
2	- 3	-397	-397	-394	4802
3	0	203	4	0	0

From our experiments, we can conclude that the delay depends on when a

UAVs fails. As described in Equation 6.1, the delay will always vary from 0 to *TimeToLive*, which in our case was of 5 seconds.

$$Delay[s] = \begin{cases} TimeToLive - t, & \text{if } t \leq 5 \\ 0, & \text{otherwise} \end{cases} \quad (6.1)$$

Now that we verified that our approach is working, we want to examine if our approach is robust, and if it works in more extreme cases. Therefore, we propose two different experiments: one focusing on reliability, and another one focusing on scalability. The experiment on reliability is inline with our first experiment, where we measured the message frequency. However, in this test, we want to examine if there is an additional delay introduced when UAVs are flying far from each other, rather than just checking if a message is received or not. Therefore, we performed a test with just three UAVs flying next to each other, at a distance of 25 meters, and we let the master UAV fail at a distance of 15 meters from waypoint 1. We measured the time the UAVs remain waiting at waypoint 1 before continuing their flight. We then increased the distance (with steps of 25 meters) until there was no more communication between the UAVs, and a swarm split-up occurred. As shown in Figure 6.4, the distance between the UAVs does not have a significant influence on the delay introduced by the protocol. We had to stop this experiment when the distance between the UAVs was of 450 meters. At this point, a swarm split-up occurred. This happens because, when the master UAV fails, the distance between the two nearest UAVs increases from 450 meters to 900 meters, and thus the distance becomes too large for communication to be feasible.

In the experiment on scalability, we compare a flight with a high number of UAVs (100) against one with a low number of UAVs (4). In those tests, some of the UAVs will fail 15 meters before a waypoint. We measure the flight time and the time waiting at that waypoint. To this purpose, we designed three scenarios:

a) A control flight where no UAV fails.

b) A flight where half the number of UAVs (and the master) will fail at a particular waypoint.

c) A flight where 10% of the UAVs (and the master) will fail at each consecutive waypoint.

We believe that those three scenarios cover the scalability parameter of our approach sufficiently. Scenario *b* is designed to test what happens in the very unlikely case of many UAVs failing at once, whereas scenario *c* is more realistic, although the fail rate is still quite high.

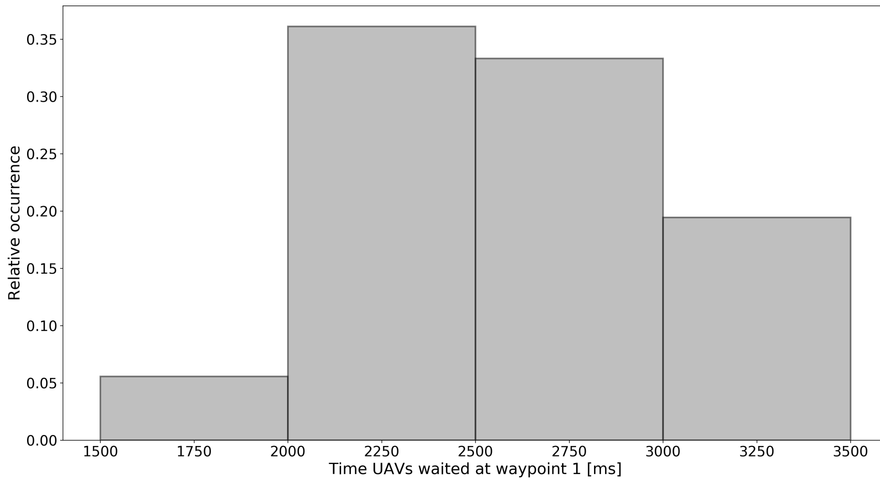


Figure 6.4: Distribution of the UAV waiting times at waypoint 1.

Since one hundred UAVs flying in a linear formation occupy an extremely large area, we decided to set the communication range to unrestricted inside ArduSim. In practical terms, this means that all the UAVs are able to communicate with each other, and thus we avoid any unwanted split-up.

The results are shown in Figure 6.5; to highlight the difference between all the cases in more detail, we show the difference w.r.t. a flight with only 2 UAVs (none failed). As shown in the figure, the results are very similar, and the impact of scaling-up the swarm is minor. However, flying with more UAVs does slow down the system a bit. In our experiments, this extra delay was limited to a maximum of 3 seconds. Overall, it is insignificant with respect to the total flight time. In addition, the delay is independent of the flight time itself; that is to say, the overall flight time is primarily dependent on the flight distance, while the delay is primarily influenced by message buffering. Furthermore, ArduSim can also introduce small and unpredictable delays, as shown in earlier experiments. This figure also confirms some of our earlier statements, since one can clearly see that, in those cases where UAVs are failing, a delay is introduced. As shown in the last case (10% of the UAVs failing at each waypoint), the overall delay grows with the number of times UAVs fail during the mission.

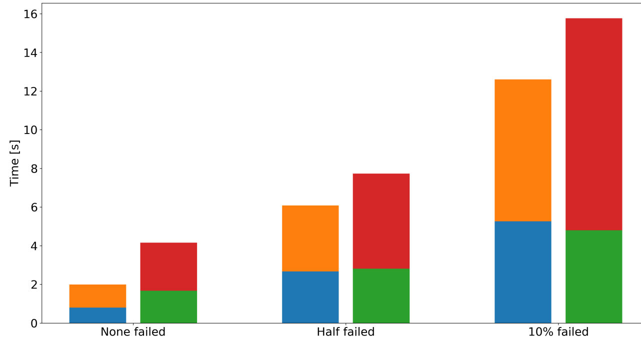


Figure 6.5: Flight time and wait time overhead when varying the number of UAVs that fail.

Finally, as explained before, a swarm split-up can occur whenever UAVs are too far away from each other, thus making direct communication impossible. Our protocol has been developed such that all the UAVs take individual decisions about whether or not another UAV is still alive. Therefore, the protocol is inherently able to handle a swarm split-up correctly. However, this can only be proven by experiments. For that reason, we have devised multiple experiments where UAVs flew according to a specific special flight formation to force partitioning to occur, as exemplified in Figure 6.1, which makes a swarm split-up possible. During the setup phase, 13 UAVs are placed close together so that each UAV knows about the existence of the other UAVs. While taking off, the distance between the UAVs increases such that distinct groups are formed in the formation. They first fly together but, at the second waypoint, the master fails. After this event, due to the large distance between the groups, the UAVs can only communicate with a subset of the original swarm. We will experiment with a different number of groups (2, 3 and 4). For each of them, the overall flight time and waiting time at waypoints is recorded. In addition, the time offset between the different groups is also measured.

Before discussing the measurements, we will provide an overview of the events occurring during the mission:

1. The UAVs are placed close to each other, so that all the UAVs know about the existence of the other UAVs.

2. The UAVs take off and fly to their place in the swarm formation.

3. They reach their place in the swarm formation, and we can visually see that different groups (or subswarms) are formed, being the master UAV in

the middle. Notice that the groups are only formed visually; from a logical perspective, the UAVs still belong to a single swarm.

4. The mission starts, and the UAVs go to the first waypoint.
5. Upon reaching the first waypoint, the UAVs inside one group find they are not able to communicate with the UAVs of other groups, and so they consider that all other groups have failed. However, since they are still in contact with their master, there is no need to switch between masters. Hence, they only remove the presumed failed UAVs from the list of potential masters.
6. In-between waypoint one and two, we let the master UAV fail.
7. Upon reaching the second waypoint, the UAVs notice that the master UAV has failed, and thus a new master will be chosen. This master is different for each group.
8. The mission continues without any problem, and they reach the last waypoint.
9. Upon reaching the last waypoint, the slave UAVs will move towards their master and land. This means that, after the mission, two groups of UAVs remain with a large distance between them, as shown in Figure 6.7. This happens because, in our protocol, we have chosen to land near the current master, rather than picking the original landing position.

Furthermore, we investigated the effect of having a different number of sub-swarms. During the flight, the flight time and the time waiting at a waypoint were measured, and the times were subtracted from a control case (linear flight with 13 UAVs where none of them failed). Those results are shown in Figure 6.6. We performed experiments with 2, 3 and 4 sub-swarms. For each sub-swarm we performed two experiments: one where the master UAV fails, and thus a split-up is created, and another one where the master UAV does not fail, and thus the swarm stays connected.

As the results show, the number of sub-swarms, and whether or not they are formed, has little influence on the flight time, or the time the UAVs are waiting at the waypoints.

To conclude this set of experiments, we compared the time of arrival between the different sub-swarms. In all cases, the difference between the time of arrival was very small (below 500 ms). This is due to the fact that all the UAVs travel with the same speed. Besides, the path of the UAVs was a straight line; whenever the path consists of curves, the outer swarm clusters will experience larger travel distances, and will therefore arrive later.

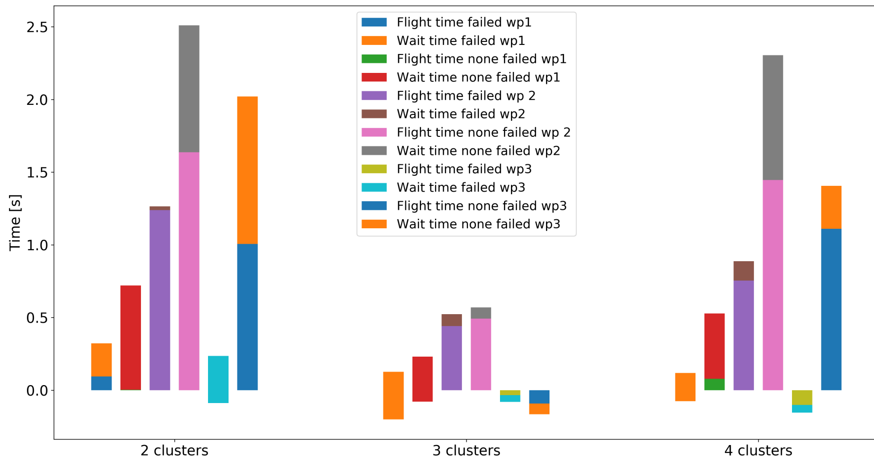


Figure 6.6: Time differences for multiple groups at each waypoint.

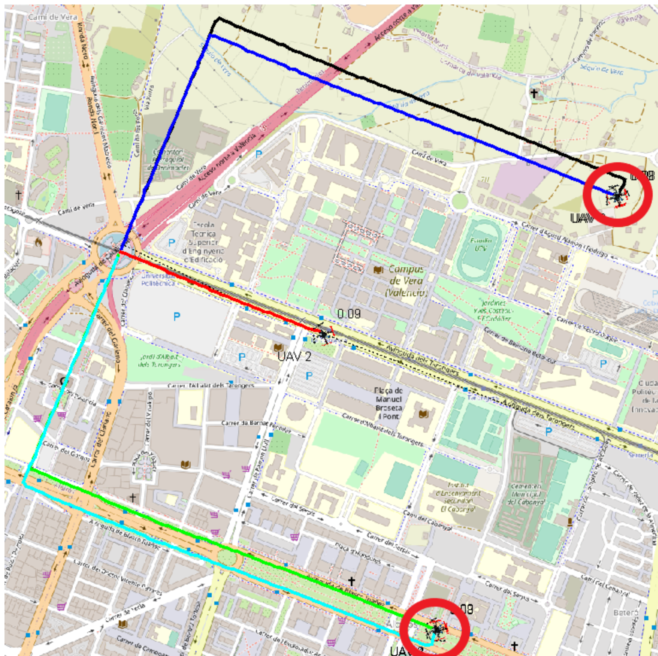


Figure 6.7: Working example of a swarm split-up scenario in ArduSim.

6.4 Summary

Maintaining a swarm of UAVs coherent throughout a mission is not an easy task. Small interferences (e.g. wind) can cause one or more UAVs to fly at different speeds and, over time, this can cause the entire swarm to break. Hence, a protocol called Mission-Based UAV Swarm Coordination Protocol (MUSCOP) was developed in order to synchronize all UAVs throughout their flight. This protocol uses a master-slave pattern, where the master makes the slaves wait at each waypoint until all the slaves have arrived. It assumes, however, that all the UAVs are able to communicate with each other at all times. If only one UAV (master or slave) would fail during the flight, the entire swarm would wait indefinitely (in practice, until the batteries are depleted). Such a failure is of course undesired, and can occur easily due to crashes or communication issues. Hence, in this chapter, we proposed an extension for MUSCOP. This extension uses the messages the UAVs are already transmitting to verify if all the UAVs are still working properly. If this is not the case, appropriate actions are taken so that the rest of the swarm members can continue their flight. Based on the many experiments we have performed, we can conclude that our protocol is able to provide resilience against the loss of aircraft in the swarm. Hence, not only can the protocol be used in many environments, it also does not matter how many UAVs fail, or what role they had in the swarm. We find that the delay is only dependent on the actual place where the UAV failed. For most real world applications, the delay is found to be neglectable, and for the worst-case scenarios it is still bounded to a few seconds. Our approach is also able to handle swarm split-ups in such a way that a smaller (subset) of the original swarm is still able to work together whenever communication with the rest of the swarm members is lost.

Chapter 7

Advanced mid-flight maneuvers

In the previous chapter, we detailed different procedures that allow for a consistent swarm flight. Yet, in certain cases, more advanced features may be required to address the need for a complex or dynamically-defined mission. Hence, in this chapter, we provide contributions in two different directions. On the one hand, we will propose a solution that allows the swarm to be reconfigured mid-air, and on demand, that is both fast, and secure. On the other hand, we will propose a solution that allows each individual UAV in the swarm to adapt to its specific terrain profile, which may cause some UAVs to require more altitude adjustments than others. Both of these solutions are discussed below.

7.1 Swarm reconfiguration

In this section, we provide a novel solution to change the formation of a swarm of UAVs in real time during the flight. This can be useful in many cases, and for different applications. Take for instance a search and rescue mission. While searching, the UAVs will need to cover a large area as fast as possible. Hence, a linear formation might be the most adequate choice. However, once one of the UAVs discovers the person(s) of interest, a circular formation can provide a better overview of the situation. Therefore, we provide a computationally inexpensive protocol that allows a swarm to reconfigure while offering safety guarantees.

7.1.1 Implementation

In our approach, we make use of a master-slave pattern. The master is elected before taking off, and in general the master is the UAV located in the center of the formation in order to minimize losses on the wireless channel. Hence, the master is in charge of the main calculations to be performed, and keeps the swarm synchronized throughout the different stages of the reconfiguration. All the stages are described in Figure 7.1. The reconfiguration will start following a trigger event, which can be a user input or a predefined event. The reconfiguration itself is divided into two phases: (i) an analysis step where the calculations are done, and (ii) a mobility step where the UAVs move to their target locations in an intelligent manner to avoid collisions. After the swarm has reconfigured itself, the mission can continue.

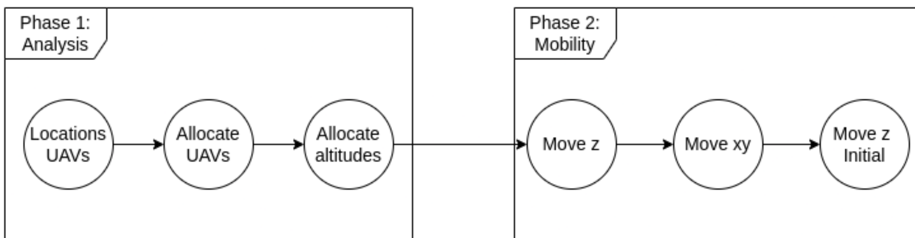


Figure 7.1: Flowchart of the flight formation reconfiguration algorithm.

In the first phase (i.e. the analysis phase), the master assigns a new location for all the slaves (later referred to as intelligent position). This algorithm is very similar to the one already explained in chapter 4. Again, we search for an assignment that minimizes the total distance travelled. In order for the master to execute this algorithm, it needs to know (i) where all the UAVs are currently, and (ii) the new swarm layout. Using MUSCOP (explained in chapter 6), the master is already aware of where all the slaves are at each moment. However, in case MUSCOP is not used, the positions of each UAV should be shared with the master. The new swarm layout is of course provided by the user or the application. It is worth pointing out that, in the mobility stage, the UAVs will fly at different altitudes to reduce the chance of collisions. So, the next thing the master needs to do is to decide which UAV flies at which altitude. That process is fully described in Algorithm 7.1. It details how the master calculates (for each slave) in what direction it has to go. Based on that direction, the UAVs are placed into different angular sectors. Each sector has a different altitude assigned to it (in our experiments, we use a simple five-meter increment from one sector to the next). In this manner, UAVs that are

likely to cross each other's path, will now fly at different altitudes, and thus we significantly decrease the chances of collisions. Note that this algorithm does not completely guarantee a collision free reconfiguration. Once the calculations are done, the master will start sending messages with the target location (x, y, Δz) to all slaves. Upon receiving this message, the slaves will reply with acknowledgements, and, once all the slaves have received their target location, the swarm will transition to the mobility step.

Algorithm 7.1 Section select procedure

Require: $numberOfSections > 0$

```
for UAV in UAVs do
    Δx ← UAV.targetLoc.x − UAV.startLoc.x
    Δy ← UAV.targetLoc.y − UAV.startLoc.y
    α ← atan2(Δy, Δx)
    if α < 0 then
        α = α + 2 × π
    end if
    sectorWidth = (2 × π) / numberOfSections
    sector ← 0
    for i in range(0, numberOfSections) do
        min ← i × sectorWidth
        max ← (i + 1) × sectorWidth
        if min ≤ α < max then
            Sector = i
        end if
    end for
end for
```

In the second phase (i.e. the mobility phase), the UAVs will move from the original place to their new place in the formation. In our protocol, we subdivide this phase into three stages: first, the UAVs will change altitude depending on their sector, as explained above; then, they will go towards their target location (X, Y movement), and finally they will return to their initial altitude (return to the default Z value). In each stage, the master will send messages to the slaves. when a slave receives the message, it will perform the movement and reply with an acknowledgement once the movement is finished. The master receives the acknowledgements and, when all the slaves have sent an acknowledgement message (and the master has reached its position), the master will transition to the next stage. At that moment, the master will start sending messages for the new stage. The slaves will receive those messages, and transition to the next stage. The

messages sent by the master only contain an ID which represents the current state. They do not have to contain the location information because this was already sent in phase 1.

As a final remark, it is worth pointing out that our proposal is computationally efficient. Algorithm 7.1 is the only element with significant computational requirements, and it is limited to $O(N^2)$ in the worst case. Since, in most practical applications, the number of UAVs in a swarm will be low (below 100), this algorithm can be easily executed on the UAV's onboard computer. Also, the network will not be overloaded since the message payloads are quite small.

7.1.2 Experiments & results

We performed a wide set of experiments in ArduSim in order to assess the validity and robustness of our proposed mechanism. As described above, our approach combines an intelligent UAV assignment with a sectorization procedure that divides the UAVs into different altitude levels. To assess the effectiveness of this combined approach, we will compare it to other (simpler variants) where such mechanism is not used. This way we can evaluate which stage introduces a higher overhead, and whether our approach (as a whole) is effective. Therefore, we propose three other (but similar) approaches:

A. Random position assignment, no altitude change.

B. Random position assignment, different altitudes.

C. Intelligent positioning, no altitude change.

D. Intelligent positioning, different altitudes.

In our first set of experiments, 9 UAVs changed from a linear formation towards a compact mesh formation (see Figure 7.2 for a graphical illustration). The minimum distance between the UAVs in that formation was set to 10 meters, the number of sectors was equal to three and the altitude difference between sectors was of 5 meters. These variables can be set by the user. During the experiments we measured (i) the time that the UAVs spent in each state (Move_Z, Move_XY, Move_Z_Initial), (ii) the minimum distance between the UAVs during the MoveXY stage, and (iii) the potential number of collisions. A potential collision is registered when the distance between two UAVs is smaller than 5 meters to account for the GPS offset error.

The results are shown in Tables 7.1 and 7.2. Our experiments have shown (as stated before) that merely changing the formation layout without adopting any

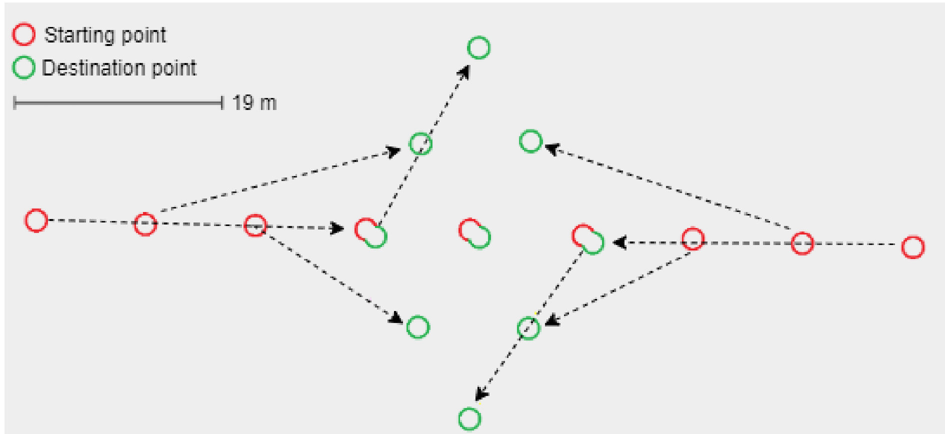


Figure 7.2: Transition of 9 UAVs from a linear formation to a compact mesh.

type of strategy is very dangerous, and prone to cause collisions. We can also observe that just by changing the altitude, or the position assignment of the UAVs, in an intelligent manner, is not enough to avoid collisions in all cases. In fact, only when both were used could collisions be entirely avoided. Furthermore, while changing the altitude does make the process safer, an additional time overhead is introduced. The time overhead depends on the number of sectors and the altitude difference between the sectors; the impact of both parameters is discussed in more detail in the experiments that follow. Implementing an intelligent positioning system reduces the overall flight distance and, therefore, flight times are slightly shorter in experiments C and D.

Table 7.1: Collisions and minimum distance analysis.

Approach	Nr. collisions	Min. Distance between UAVs
A	4	0.44
B	2	0.33
C	2	3.58
D	0	6.15

In our second experiment, we want to evaluate the scalability of our protocol. We searched for the minimal number of sectors needed to complete a collision-free reconfiguration for different number of UAVs, and for different formations. All the formations were prone to collisions due to the small distance between the UAVs that was defined ($\leq$ 10 m). We started with 9 UAVs (as in the previous

Table 7.2: Time UAVs spend in each state.

Approach	Move z [ms]	Move XY [ms]	Move Z ini [ms]
A	404	13607	400
B	6802	13030	7980
C	380	12425	400
D	8600	12415	8600

experiment), and increased this value up to 25 UAVs. The results are shown in Figure 7.3. As expected, the minimum number of sectors required to guarantee a collision-free reconfiguration increases with the number of UAVs. The actual increase highly depends on the type of formation.

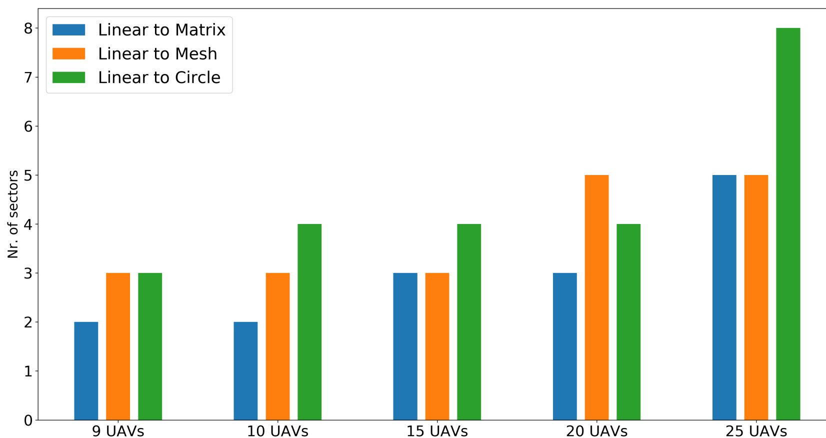


Figure 7.3: Minimum number of sectors required for a collision-free reconfiguration procedure.

Due to our previous findings, we investigated the influence of the type of formation in greater detail. In particular, we tested all the possible transitions between the four flight formations considered (Linear, Matrix, Mesh, and Circle). The experimental settings are similar to the previous ones. We worked with 15 UAVs in formations where the distance between the UAVs is 10 meters. During the experiment, we searched for the minimal number of sectors needed to complete a collision-free reconfiguration. We also measured the time spent at each state.

Results are shown in Figures 7.4 and 7.5. As we can observe from Figure 7.4,

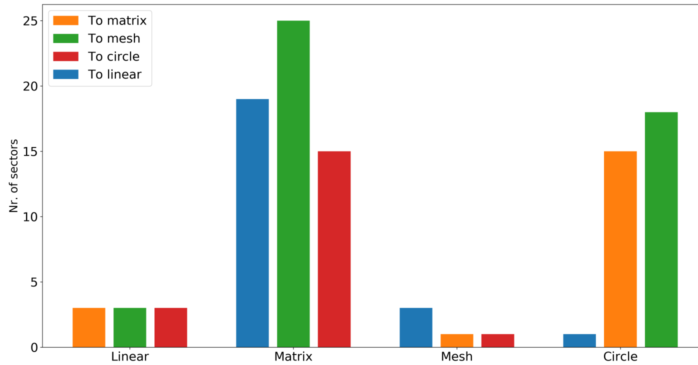


Figure 7.4: Minimum number of sectors required for a collision-free reconfiguration w.r.t. the type of transition.

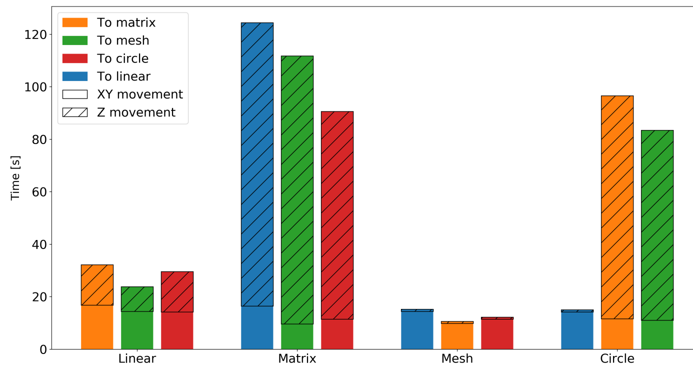


Figure 7.5: Time spent moving horizontally (state 2) and vertically (states 1,3).

results vary significantly depending on the specific transition; in some cases, such as going from a mesh to a matrix formation, just a few sectors are needed. In other cases (e.g. matrix to linear) the angles α calculated in Algorithm 7.1 are very similar, and so many sectors are required in order to separate the UAVs in different altitude groups. In the presence of many groups, the target altitude can grow a lot, resulting in a high time overhead (in the worst-case scenarios), as shown in Figure 7.5. Due to the similar shape of both figures, we can see the correlation between the number of sectors and the overall reconfiguration time. Furthermore, we can

conclude that the time spent moving in the xy-plane fluctuates only a little, being limited to a maximum of 6.8 seconds in our experiments.

Finally, we further investigated the time overhead introduced by changing UAV altitudes during reconfiguration. To achieve this, we start by finding the value of the one-way delay T for which:

$$\int_0^T v(t)dt = D \tag{7.1}$$

where

$$D = num.sectors \times sectors_offset$$

This one-way delay refers to both upward or downward movements. We can approximate this one-way time overhead T as:

$$T \cong \frac{D}{\hat{v}_{0\to T}} + \epsilon \tag{7.2}$$

where $\hat{v}_{0\to T}$ refers to the expected speed during the entire mobility from time 0 to T, and ϵ accounts for the additional time associated to acceleration and deceleration processes. In our experiments, $\hat{v}_{0\to T}$ was set to 2 m/s, and the distance between the sectors ($sector_offset$) to 5 meters. The number of sectors ranged between 2 and 8.

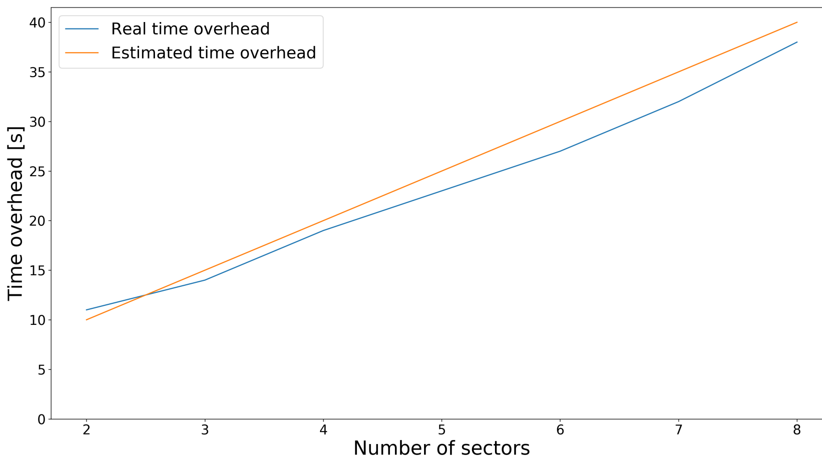


Figure 7.6: Estimated time overhead vs. real time overhead.

Figure 7.6 compares the estimated time overhead for the two-way vertical mobility against the real-time overhead measured in our experiments. We can clearly observe a linear pattern (as suspected by the derivation), and, in our case, the average value of ϵ is of 1.5s. While the speed of the UAVs does influence the time overhead directly, it does not alter the chances of a collision. This is because all the UAVs are flying at the same speed, and thus the distance between them will not change.

7.1.3 Summary

In this section, we focused specifically on the reconfiguration of a swarm. This is a relevant behavior that can be used to make many applications more efficient and/or effective. However, the chances of collision become high during reconfiguration, and so it becomes an issue that must be dealt with. Our proposal is based on an intelligent position assignment system that reduces the chances of flight paths crossing during formation reconfiguration. The chances of collision are further reduced by distributing the UAVs over different altitude levels during the reconfiguration period. This simple, computationally efficient approach can be easily applied to various environments. However, it is not able to fully guarantee a collision-free reconfiguration in all cases, and when scaled-up to many UAVs the time overhead introduced becomes significant.

7.2 Adjusting the altitude for changing terrain levels

In many applications, the UAV flies automatically from one GPS waypoint to another. Throughout its path, which is often defined using mission waypoints, the UAV has to maintain its altitude steady. This is a common built-in feature in many higher-end UAVs, and it is provided by the flight controller, i.e. a small on-board computer with multiple sensors which regulates the speed of the propellers to accomplish its flight goals.

The most commonly used open-source flight controller is called Pixhawk [3], which often uses the Ardupilot firmware [6]. Through the use of Ardupilot (or any other flight controller firmware for that manner) we can maintain a constant altitude. However, this constant altitude is referenced from the take-off point of the UAV (i.e. from the origin). As shown in Figure 7.7, this is problematic if the terrain is not flat. This is because, if the terrain rises, there is a chance that the UAV will crash into the ground. On the other hand, if the terrain is descending, the drone will fly higher; this can decrease the performance of the UAV sensors (e.g. camera feeds), or cause it to surpass the legal constraint regarding maximum flight

altitude (for example, in Europe it is of 120 meters [10]). Therefore, many UAV applications require maintaining a constant Above Ground Level (AGL) altitude. We must point out that third-party applications that consider the topography in mission planning already exist (e.g. FlyLitchi [40]). However, many applications require that drones can take ad-hoc decisions and deviate from the original flight plan (e.g., to avoid collisions, or due to a change in the itinerary). Therefore, we opted for a more holistic approach so that we grant current and future applications this freedom, in addition to simplifying the operator's tasks.

Our solution allows ascending or descending a VTOL UAV while it is following a mission so as to adapt to the terrain profile. Before the start of the flight, the mission file (which consists of GPS waypoints), and the DEM file (i.e. a file with topographical data) are uploaded to the UAV. During the flight, the UAV will use the mission file to determine its direction, and the DEM file to adjust its altitude. It is important to notice that the change of altitude is calculated in real-time, and not before the start of the mission. This approach allows us to (i) spend less time on the calculation at start-up, (ii) make changes to the mission without worrying about the terrain profile, and (iii) make our method compatible with other (future) approaches, which use sensors instead of the DEM file to determine the altitude relative to the ground.

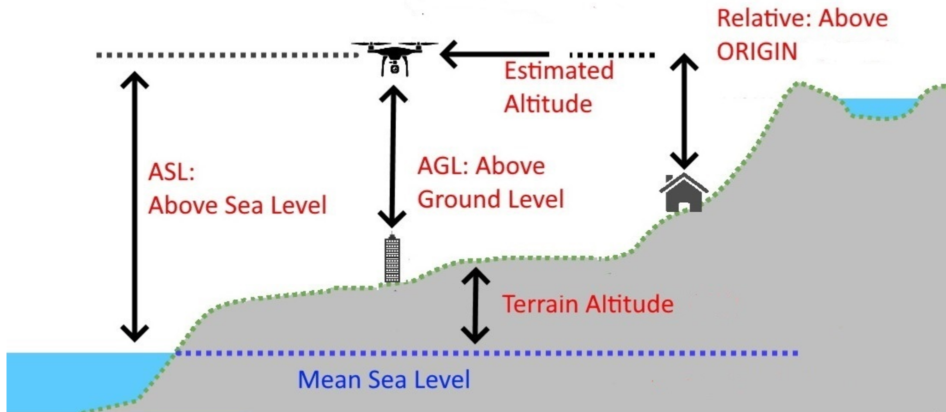


Figure 7.7: The different types of altitude.

7.2.1 Implementation

In order to maintain a stable Above Ground Level (AGL) altitude, we implement a control system. It controls the velocity of the UAV in the z axis (or up/down axis), and aims at minimizing the error between the desired and current flight altitude. For this purpose we use the terrain elevation, coming from the DEM file, as a reference signal.

A DEM consists of a database which represents the height above sea level of a given location. A DEM is characterized by its versatility in terms of accuracy/compression. Despite different formats exist, the most common one divides a land area into regular sized squares. For each square, the highest altitude is measured and saved into a large matrix. In the header of the DEM file, the GPS coordinates of the origin are provided, as well as the size of each square (also called the resolution). Since it is easy to retrieve the row and column number when reading the DEM file, the GPS location of each cell can easily be calculated (i.e. by adding the offset from the origin). In this manner, a relatively large land area can be represented in a small file. The size of the DEM file will depend on the resolution, but, to provide a general idea, using a (high) resolution of 4 m^2 for each square, an area of 500 km^2 can be represented in a file of only 250 MB. Considering that these storage requirements are not a problem nowadays, and that VTOL UAVs can only fly for a few kilometers anyway, size does not represent a problem.

DEM files are readily available on the Internet. For our research, we have used the DEMs provided by the Spanish government [41]. As one might observe, on this site (and in general), two types of DEMs exist (see Figure 7.8): we have the Digital Terrain Model (DTM), which represents the ground level, and the Digital Surface Model (DSM), which also includes buildings, trees, etc. It is obvious that the DSMs represent the real world more accurately (especially in cities), but also become obsolete much faster. Since the file format of DTM and DSM files are very similar, our approach works both on DTM and DSM files. Nevertheless, for our experiments, we have chosen to only work with DTM files.

Furthermore, our control system also needs the relative altitude of the UAV in order to maintain a stable AGL altitude. The relative altitude of the drone is determined by the flight controller, which relies on GNSS and barometric data. The error, which we try to minimize, is determined by the difference between the reference altitude, and the relative altitude of the drone:

$$e(t) = reference_altitude(t) - relative_altitude(t) \tag{7.3}$$

The task of our controller is to calculate an output such that this error is minimized. In our system, the output of the controller is the vertical velocity

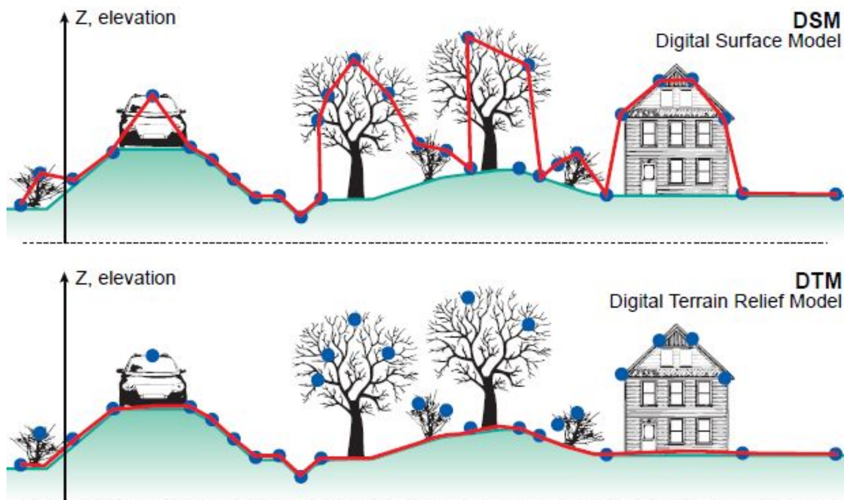


Figure 7.8: The difference between DTM and DSM, source [42]

[m/s]. In theory, those values can range from $-\infty$ to $+\infty$; in practice, however, the electrical motors have a maximum speed which will limit thrust. Of course, different motors have different maximum speeds. Nevertheless, as long as we are not in the drone racing business, most operators prefer a (relatively) slow vertical velocity, mainly for safety and control reasons. Therefore, we have limited the output of our control to the range $\pm 5\, m/s$.

A general proportional–integral–derivativel (PID) controller can be described through the following equation:

$$u(t) = K_p \cdot e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{7.4}$$

The output signal $u(t)$ is thus given by the sum of three terms, which all depend on the error signal: 1) the proportional term (P), which will correct for the current error; 2) the integral term (I), which will grow over time, and thus can ensure that the steady state error becomes zero; and 3) the derivative term (D), which allows predicting the future error, and thus increases the stability of the system. Since in our application the ground level (i.e. the reference signal) will change over time, we do not have to be concerned about a steady state error. Therefore, we removed the integral term (by setting $K_i = 0$), which allowed us to simplify equation 7.4 to a PD controller as follows:

$$u(t) = K_p \cdot e(t) + K_d \frac{de(t)}{dt} \tag{7.5}$$

The equation above is continuous; however, a digital system can never be continuous, and thus we are forced to discretize the equation. For our drone (and all drones alike that use ArduCopter [6]), the vertical velocity vector is updated every 200 ms. Thus, the controller equation can be discretized considering this sampling period (T_s). Furthermore, we change the naming from the generic output signal u to the more appropriate name v_{ver}, to clarify that we use a PD-controller to control the vertical velocity.

$$v_{ver}[t] = K_p \cdot e[t] + K_d \frac{e[t] - e[t-1]}{T_s}, \qquad T_s = 200ms \tag{7.6}$$

At this point, we have designed the vertical controller that allows the UAV to maintain the altitude. However, we also need the UAV to move towards its targets, i.e. to follow the planned mission. The current target (or waypoint) is given in GPS coordinates. In order to move the UAV towards the target, we simply need to define a direction vector $\overrightarrow{u}$ that starts from the drone and ends at the target. After normalization, this vector can be scaled such that the UAV moves at the planned (or max) speed (in our case, max_v_{hor}=15 m/s, as this is the default setting in the flight controller). Thus, we have the following expression to obtain the horizontal velocity vector $\overrightarrow{v_{hor}}$:

$$\begin{aligned} \overrightarrow{u} &= \overrightarrow{target} - \overrightarrow{UAV} \\ \hat{u} &:= \frac{\overrightarrow{u}}{||\overrightarrow{u}||} \\ \overrightarrow{v_{hor}} &= max_v_{hor} \cdot \overrightarrow{u} \end{aligned} \tag{7.7}$$

Given the right parameters, this should be enough to move the UAV from its current position to its target while adjusting its altitude, so that its relative altitude to the ground remains as constant as possible. However, during our simulated tests, we have observed two issues: firstly, when the change in terrain height is significant, the UAV needs more time to ascend/descend, otherwise it might still result in a crash. In practice this means that, if the velocity in the z axis is close to its limit max_v_{ver} (5 m/s in our proposal), we need to stop moving towards the target to avoid a likely collision. Secondly, in equation 7.3, we denoted the error signal as the difference between the reference altitude and the real altitude, at the same time instance. During the actual flight, as we will show later, an improvement can be made by looking ahead to where the UAV will be, and take that altitude

as reference. These small adjustments have been implemented in our proposal, as shown in Figure 7.9.

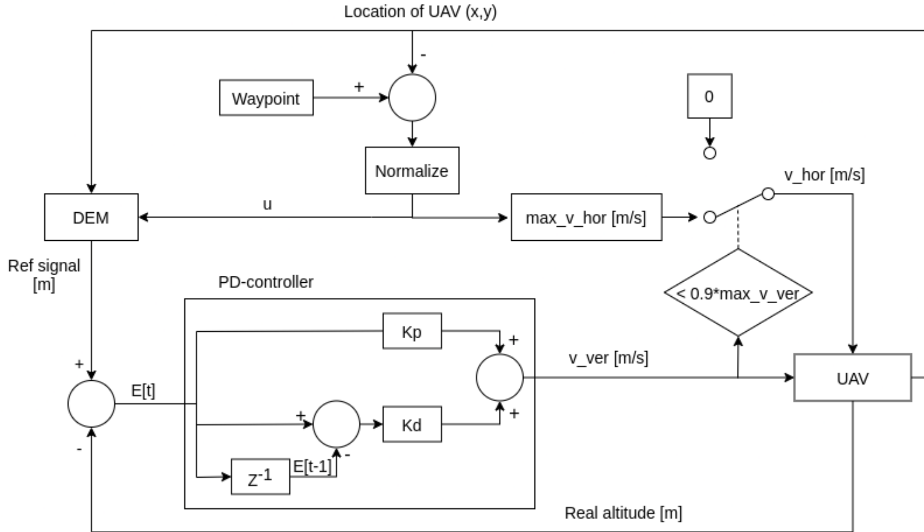


Figure 7.9: Block diagram of the proposed method.

The flight controller of the UAV (block on the right) will provide us with the current (absolute) altitude of the drone. The difference between the reference signal and the current altitude denotes the error signal. This error signal is fed into the PD-controller, which in turn calculates the vertical velocity needed to minimize the error signal. The vertical velocity signal also controls whether the horizontal velocity is set to 0 m/s or not. In case the vertical velocity is high (more than 90% of the maximum allowed), the switch (on the right side of the block diagram) will toggle and set the horizontal velocity to zero. This will give the UAV the required time to adjust to higher altitude differences before moving towards the waypoint again. Notice that we intend to proceed with the planned mission as quickly as possible, only reducing horizontal speed when strictly necessary. The flight controller also provides us with the location of the UAV. This location, together with the location of the waypoint, is used to calculate the direction vector $\overrightarrow{u}$. The horizontal velocity vector is obtained by scaling the direction vector $\overrightarrow{u}$ with the appropriate horizontal speed. Finally, the DEM block calculates the reference signal. This calculation is based on the DEM file, which contains the absolute altitude level for a given location. We use the current location of the UAV, as well

as its direction vector $\overrightarrow{u}$, to calculate its future position. The ground level altitude of this future position is fetched from the DEM file, and the reference signal is then calculated by adding the desired above ground level offset.

7.3 Experiments & results

7.3.1 Parameter tuning

In our model there are five parameters: the proportional gain K_p, the derivative gain K_d, the look ahead distance [m], the maximum horizontal velocity (max_v_{hor}), and the maximum vertical velocity (max_v_{ver}). Hence, we need to find the optimal values for these parameters, as the chosen values will determine how fast the drone responds to changes, as well as the stability of the system. We have decided to set the two maximum speed parameters to 15 m/s for the horizontal speed, and 5 m/s for the vertical speed. As previously stated, these values are a good compromise between safety and speed, and are often used in VTOL drone applications. This section will explain how we adjusted the values for the other three parameters. Our tuning aims at meeting our specific goals, which include achieving an adequate trade-off between vertical error (overshooting included) and mission time. It is important to note that, depending on the characteristics of the UAV, these values might vary. Thus, fine-tuning of these parameters for other types of UAVs will be necessary. Auto-tuning methods do exist, but they require deriving a mathematical model of the UAV, which we currently do not have. Therefore, we will estimate the values for our parameters based on some commonly known rules. In this section, we will explain the process we went through so that results can be replicated.

First, it must be noted that, determining the optimal parameter values based on simulating an entire mission, is impractical. This is due to various reasons:

1. Simulating an entire mission takes a long time;
2. It is difficult to determine the true effect of a change in the parameters;
3. One can be fixed on solving a specific problem (i.e. the mission), while instead the solution should be applicable to different missions (i.e. optimized parameters for just one mission instead of a general solution);
4. When the entire mission is considered, the horizontal movement of the UAV also plays a role;

Therefore, we obtain the parameter values using a classic control theory technique, i.e. observing the step response of the system. The step response is the

response (i.e. the output) of the system when a step (i.e. sudden change in the reference signal) is applied. Such step response is then plotted on a graph (see Figure 7.10), and it is common to normalize the output. Using this graph, we can determine two important values: (i) the rise time, and (ii) the overshoot. The rise time explains us how fast the system will converge to the new reference signal. The overshoot refers to the output exceeding the new desired reference signal. In our application, both metrics are important. We do not want a high overshoot (or undershoot), because this will cause the UAV to fly too high (which might violate legislation) or too low (which might result in a crash). However, the response of the UAV should also be sufficiently quick (i.e. short rise time) for those cases where the terrain profile is constantly changing, and the UAV should be able to keep up with that change.

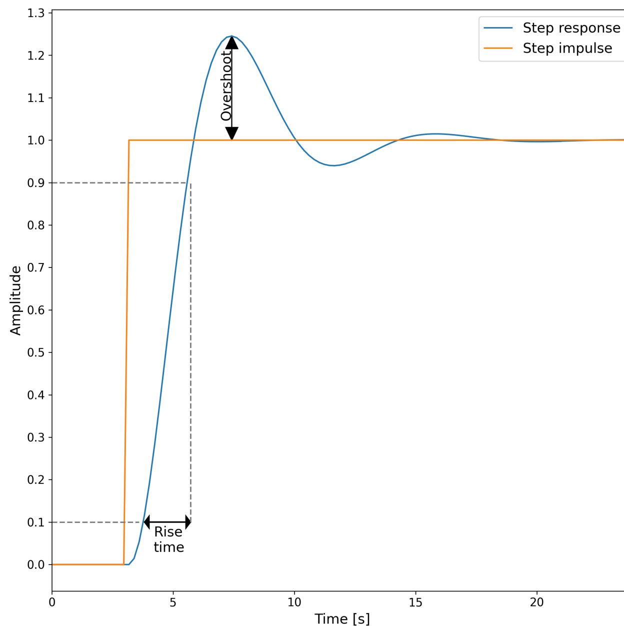


Figure 7.10: Two important metrics for the step response.

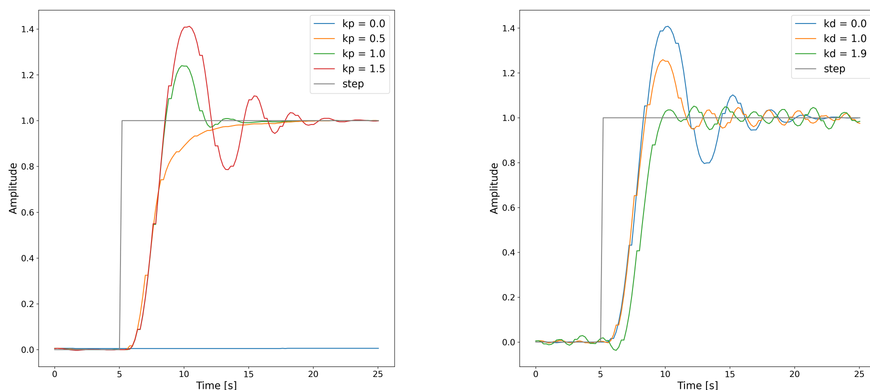
Now that we (roughly) defined our step response (i.e. low overshoot, but also

short rise time), we will proceed by varying the parameters to observe how they influence the step response. We start with parameter K_p, and set K_d (Equation 7.6) to its minimum (the look-ahead distance is not used in the step response since we are directly changing the reference signal). In Figure 7.11a we show the step response for several values of parameter K_p. In this figure, we can observe how parameter K_p influences the step response. We started by setting K_p to zero. Obviously, in this case, the UAV will not change its altitude, and just remains flying according to his original reference signal. We then increased the value of K_p and K_d with steps of 0.1 (we do not show all the possibilities to keep the figure clear). When we raise the value of K_p, we can see the rise time decreasing. At a value of 0.5 the UAV reaches the new altitude without any overshoot. However, the rise time is still too long. If we increase the value of K_p, we can reduce the rise time further. In Figure 7.11a, it seems that, in terms of rise time, there is no difference between $K_p = 1.0$ and $K_p = 1.5$. However, it is always important to take both K_p and K_d into account. As stated above, we changed both parameters with a step size of 0.1 (up to 2.0) and, after 400 experiments, our results show that, with $K_p = 1.5$ and $K_d = 1.9$, the best results are achieved. In Figure 7.11b we show the influence of the parameter K_d. As we could expect from control theory, if we increase the value of the K_d parameter, the overshoot will be reduced. However, an excessively high value will result in an unstable system that tends to oscillate. As we can see, the overshoot decreases with a higher value of K_d. Increasing the value of K_d beyond 1.9 (best choice) only introduces more unwanted oscillation. As shown in Figure 7.12, for the response of the system with the optimal parameters (w.r.t. overshoot and mission time), some oscillation remains for our control approach. However, since in our application the terrain level is constantly changing, this oscillation will not prevail. In addition, the influence of the look-ahead distance can only be clearly observed in the context of an actual mission (longer time periods). Therefore, we will show its influence in the next section.

7.3.2 Evaluation

After modeling our proposed solution and obtaining the optimal parameter values, we tested our approach under different conditions. We experimented with two different scenarios: a rural area with some hills (Cumbre de Calicanto, Valencia, Spain), and a mountainous region (Mulhacén, Granada, Spain). Each scenario was simulated several times, with different values for the look ahead parameter. During our experiments, we measured the terrain altitude and the above ground level altitude of the UAV.

Our first scenario (Cumbres Calicanto) is located in a rural area close to



(a) Influence of parameter K_p ($K_d = 0$). (b) Influence of parameter K_d ($K_p = 1.5$).

Figure 7.11: Analysis of the influence of parameters K_p and K_d on the step response of the UAV.

Valencia, Spain. We have chosen this region for its moderate setoff. The top of the hill is around 210 meters high; on one side the slope of the hill is gentle, in the center there are some abrupt peaks and troughs, and on the other side of the hill the slope is much steeper. As mentioned before, the influence of the look-ahead distance can only be measured during a complete mission. In general, looking further ahead will provide the UAV more time to anticipate altitude changes. However, looking too far ahead will introduce unwanted behavior if the terrain has narrow peaks or troughs. During our experiments, we varied the value of the look-ahead distance in the range from 2 to 20 meters. The DEM file that we use has a resolution of 2 m (square of 2×2 meters). Therefore, we increased the look-ahead distance in two-meter steps each time. During the mission, we measured the error every 200 ms. After the experiment, we calculated the mean error, and the standard deviation (see Table 7.3). Based on those metrics, we can determine that, in this scenario, a look ahead distance of 8 meters is the most effective one at reducing the error.

As mentioned before, the influence of the look-ahead distance can only be measured during a complete mission. In general, looking further ahead will provide the UAV more time to anticipate altitude changes. However, looking too far ahead will introduce unwanted behavior if the terrain has narrow peaks or troughs.

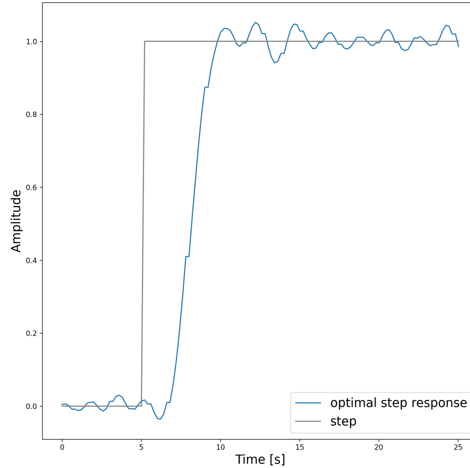


Figure 7.12: Step response with optimal parameters, i.e. $K_p = 1.5; K_d = 1.9$.

During our experiments, we varied the value of the look-ahead distance in the range from 2 to 20 meters. The DEM file that we use has a resolution of 2 m (square of 2×2 meters). Therefore, we increased the look-ahead distance in two-meter steps each time. During the mission, we measured the error every 200 ms. After the experiment, we calculated the mean error, and the standard deviation (see Table 7.3). Based on those metrics, we can determine that, in this scenario, a look ahead distance of 8 meters is the most effective one at reducing the error.

The results of the entire mission are depicted in Figure 7.13, showing that the UAV adapts to the terrain profile correctly. As shown in Figure 7.13b, the error is normally distributed with $\mu = -0.0493$ and $\sigma = 1.6260$. Stated otherwise, 99.7% of the time the error was less than 5 meters.

During our experiment, we also tracked various metrics. One of them was the vertical speed of the UAV, which we show in Figure 7.13c. It is interesting to observe how the vertical speed of the UAV changes w.r.t. the terrain altitude (shown in Figure 7.13a). As we can see, at the beginning (before 150 seconds) the terrain level changes slowly, and the vertical speed of the UAV is bounded between -3 m/s, and 3 m/s. From 150 seconds onwards, the terrain level changes much faster. This is also represented in the vertical speed of the UAV, which often reaches its maximum of $\pm 5m/s$. In addition, we find that changing the altitude

Table 7.3: Influence of the look-ahead distance for scenario A.

Look-ahead distance [m]	Mean error [m]	Standard deviation [m]
2	0.1834	2.5176
4	-0.1464	2.3930
6	-0.0721	1.7832
8	-0.0493	1.6260
10	-0.0724	1.6399
12	0.0203	1.5128
14	0.0049	1.7969
16	0.0449	1.9540
18	0.1118	2.1984
20	0.1678	2.5435

levels does not have a significant impact in terms of extra time or energy. To check this, we measured various metrics for the flight using our altitude adjustment approach, and without using this mechanism. As shown in Table 7.4, the mission is 3238 meters long. Due to the setoff, the UAV had to adjust its altitude by 583 meters overall. This change in altitude caused a slight mission delay of about 3%, and an increase in energy usage of 3.49%. These payoffs are very small, especially considering that the alternative (not changing altitude) will result in a crash.

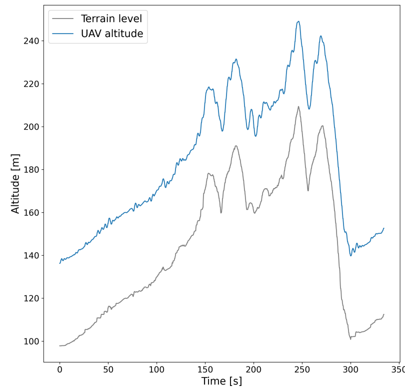
Table 7.4: Rural flight: influence of adjusting the altitude on flight time and energy.

	Without adjustment	With adjustment	Difference
Horizontal distance [m]	3238	3238	0%
Vertical distance [m]	0	583	-
Flight time [s]	324	334	+3.01%
Energy consumed [kWh]	127	132	+3.49%

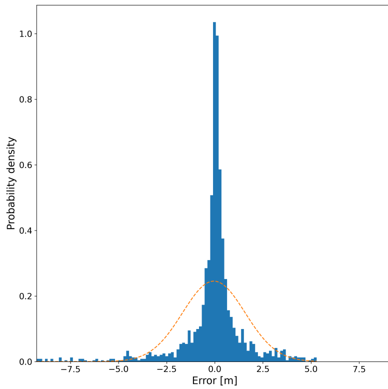
After these promising results in a rural area, we tested our model in a more challenging environment, i.e. a mountainous region. We have chosen to perform this simulation in the area of Mulhacén, Granada, Spain. In these mountains, the ground levels will change faster and more often, which makes it more difficult for the UAV to maintain a constant altitude above the ground.

For this region, we also experimented with the look ahead distance. As shown in Table 7.5, the margin of error is higher due to the increased scenario complexity. In a mountainous region, the look-ahead distance has to be shorter, as expected. In this case, we can obtain the best results with a look-ahead distance of 4 meters.

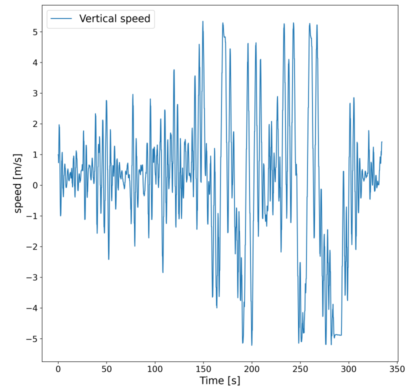
From Figure 7.14a we can observe that the terrain is more challenging, and



(a) Altitude of the UAV w.r.t the terrain altitude.



(b) The distribution of the error.



(c) Vertical speed [m/s] of the UAV.

Figure 7.13: Results for scenario A.

with the use of our approach a ground crash can be avoided. However, as also shown in Figure 7.14b, the distribution of the error is broader than in the previous case. Also for this flight, we tracked various metrics. As shown in Figure 7.14c,

Table 7.5: Influence of the look-ahead distance for scenario B.

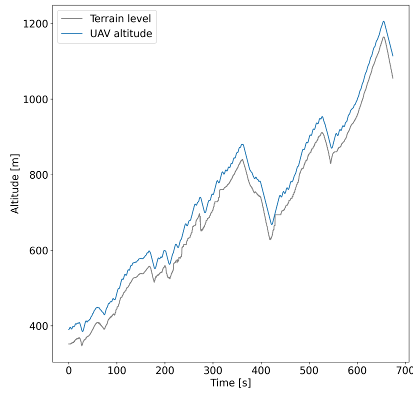
Look-ahead distance [m]	Mean error [m]	Standard deviation [m]
0	-1.5206	8.6427
2	-0.1372	7.4118
4	1.2548	6.1141
6	2.9995	6.5989
8	4.4752	7.7420
10	5.9446	9.3839

the vertical speed of the UAV is often at its maximum in an attempt to maintain a constant altitude in this rapidly changing terrain. Taking this into account, we might achieve better results (i.e. smaller mean error, and standard deviation) if we do not restrict the maximum vertical speed to 5 m/s. However, we do not increase this maximum speed for various reasons. First and foremost, for the obvious safety reasons. We must remember that there is a 200ms delay before we can update the speed, and then there is an extra delay (due to inertia) before the UAV actually reaches this new speed. If we increased the maximum vertical speed, this would most likely lead to an unwanted oscillating behavior. Furthermore, we want our approach to be applicable to many types of drones, and not all drones have enough power to ascend faster than 5 m/s. Finally, increasing the maximum vertical speed would also lead to a higher energy consumption.

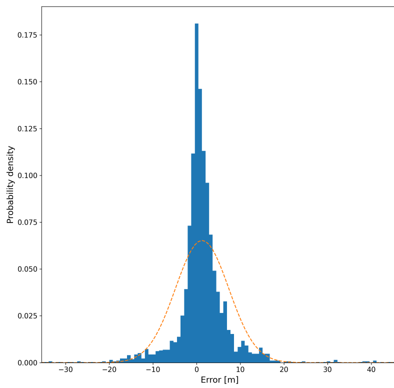
Table 7.6: Mountain flight: influence of adjusting the altitude on flight time and energy.

	Without adjustment	With adjustment	Difference
Horizontal distance [m]	6193	6193	0%
Vertical distance [m]	0	2100	-
Flight time [s]	507	655	+28.99%
Energy consumed [kWh]	200	261	+30.59%

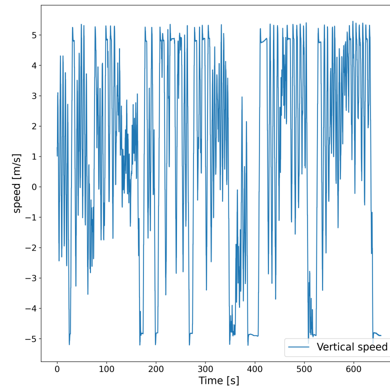
As expected, the influence of adjusting the altitude on flight time and energy consumption is much higher in this demanding scenario compared to the previous (rural) scenario. As shown in Table 7.6, the mission has a horizontal distance of, 6193 meters. The main difference between this scenario and the previous one is that, in the current scenario, the vertical distance traveled is much higher. In order to regulate the altitude above the ground, the UAV had to ascend and descend for a total of, 2100 meters throughout the mission. This caused the flight time to increase by 29% and, consequently, the energy consumption rose by 30% as well.



(a) Altitude of the UAV w.r.t. the terrain altitude.



(b) Error distribution.



(c) Vertical speed of the UAV throughout time.

Figure 7.14: Results of scenario B.

7.4 Summary

For an autonomous application, it is important that a UAV is able to maintain a constant altitude relative to the ground, even when the terrain altitude varies. This avoids that the UAV crashes into the ground, or it exceeds legal altitude constrains. We provided a solution that attempts to maintain a constant altitude relative to the ground. Our approach is based on a PD-controller, which uses the altitude data coming from a DEM file as the reference signal. Since drones differ in size, weight, and many other factors, the parameter values of the PD-controller will likely differ for each UAV. Therefore, we explained in detail our tuning method so that it can be easily replicated.

Once the PD-controller was tuned, we experimented with two different scenarios: a rural area, and a mountainous area. The results of the first (rural) scenario show that the UAV is able to maintain a constant relative altitude above the ground. During that experiment, the error (i.e. the difference between desired altitude and actual altitude) was smaller than 5 meters 99.7% of the time, and the increase in flight time and energy consumed due to such adjustments was of 3.01%, and 3.49%, respectively, a performance that we consider sufficient for most applications. In the more challenging scenario, i.e. the mountainous area, the error increased slightly, and the increase on flight time and energy consumed due to the terrain profile adjustments (2100 meters in total for vertical mobility) were now more significant, reaching 28.99%, and 30.59%, respectively.

Chapter 8

Accurate vision-based landing

The last stage of any flight is the landing procedure. Often, when performed by an experienced pilot, this represents little to no problem. Landing autonomously, however, is much more complicated. Previous proposals heavily rely on the GPS and inertial navigation sensors (INS) as the main positioning approaches [43]. However, altitude data provided by the GPS is typically inaccurate, and needs to be compensated with a close-range sensor, such as a barometric pressure sensor or a radar altimeter. Despite such compensation, these methods still remain inaccurate, especially in the horizontal plane, resulting in a landing position that typically deviates from the intended one by 1 to 3 meters. Furthermore, the GPS cannot be used indoors. For these reasons, GPS and INS systems are mostly used for long range, outdoor flights having low accuracy requirements [43].

Taking the aforementioned issues into consideration, in this chapter we introduce a novel vision-based landing system that is able to make a UAV land in a specific place with high precision. This challenge is addressed by developing a solution that combines the use of a camera, and ArUco markers [44, 45]. This way, the relative offset of the UAV towards the target landing position is calculated using the ArUco library [46] (based on OpenCV). After computing its relative offset, the UAV adjusts its position so as to move towards the center of the marker and start descending, performing additional adjustments dynamically.

8.1 Implementation

Before starting our protocol, the UAV will have to move towards the landing position. This can be done using the regular onboard GPS system. In our approach, the exact landing position has to be indicated with an ArUco marker. This marker is similar to the well-known QR-code. However, it cannot hold as much information as a QR-code (only an ID), and therefore it can be detected more easily. A typical ArUco marker consists of a black border and a 6x6 square of black and white smaller squares. There are different types of configurations (e.g. 3x3, 4x4, 7x7) which are known as dictionaries. A marker from a dictionary with fewer squares is of course easier to detect, but only a few IDs can be provided. In this work, dictionary "DICT_6X6_250" is used. As the name suggests, it provides 250 different IDs, which is more than enough for our purposes.

Using the ArUco marker library [44, 45] we are able to calculate the relative position of the UAV w.r.t. the marker. This positional information can then be used to steer the UAV while descending in such a way that the UAV lands on the marker. Of course, it is necessary that the UAV is able to detect the maker, and to that end two conditions must be met: (i) the marker must be fully inside field of vision of the camera, and (ii) each square must be uniquely identified (black or white). It is possible that the two conditions are not met simultaneously. For instance, when the drone is at a low altitude (i.e. 0.5m) the marker is too big to fit inside the field of view of the camera; in addition, the shadow of the drone may "corrupt" the image. On the other hand, when the drone is flying at a higher altitude (e.g. 12 m) the image may be too small to be detected. In order to deal with this problem, and to allow the UAV to detect the marker from both low, and high altitudes, our proposal combines multiple markers of different sizes (see Figure 8.1).

In this way, the UAV will be able to detect the big marker from a high altitude, and use that marker as a reference until it is able to detect the smaller marker. Once, the smaller marker is detected, the UAV will switch and use it as its new reference. Notice that this approach is easily scalable, and, although in our experiments we only used two markers, more markers can easily be added if required. Figure 8.2 shows a real scenario where the UAV is able to see two markers, but chooses to move towards the smaller marker. The center of this marker is indicated by the red spot.

Once the marker is detected, the drone has to move towards the center of the marker, and descend from there. Due to the effect of wind, and to the inherent instability of the UAV itself, the drone will also move in the horizontal plane while descending. This unwanted movement should be compensated in order to land the drone more precisely. To achieve this behavior, the strategy described in Algorithm

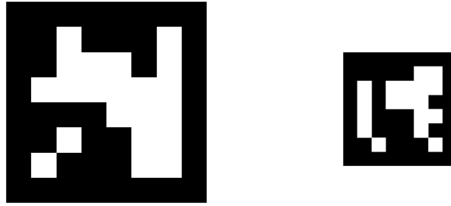


Figure 8.1: Two examples of ArUco markers of different sizes.

Figure 8.2: Image retrieved by the UAV camera after processing using OpenCV/ArUco libraries.

8.1 is proposed, which works as follows: in line 3, the UAV searches for an ArUco marker. If no marker is detected, the flight mode of the UAV is changed to loiter. If this is the case for 30 consecutive seconds, the mission is aborted, and the UAV will land solely using GPS. Otherwise, from the potential list of detected markers, the marker with the highest ID (i.e., the smallest marker) is selected (line 11). With the use of the ArUco library, the location of the marker with respect to the

drone is estimated. If the altitude of the UAV is greater than z_2 (see empirical values in Table 8.1), α is set to 20 degrees; otherwise, it is set to 10 degrees. These values are based on: the detection distance of the markers, the size of the UAV, the size of the markers, and an additional margin which is optimized empirically. In line 20 it is checked if the marker is within the virtual border (explained later). If so, the UAV descends; otherwise, it moves horizontally towards the target position. This algorithm will be executed continuously as long as the altitude of the UAV is greater than z_1. From the moment the UAV's altitude drops below z_1 (very near to ground), the control will be handed over to the flight controller, which will land the UAV in a safe manner, and disarm the engines.

Algorithm 8.1 Static vision-based landing strategy.

```
 1: Start timer 30 s
 2: while altitude > z_1 do
 3:    IDs, detected ← SearchMarker()
 4:    if ¬ detected then
 5:       Loiter()
 6:       if timer exceeded then
 7:          AbortLanding()
 8:       end if
 9:    else
10:       reset timer()
11:       ID ← highest detected ID
12:       Get P(x, y, z)_id
13:       if z > z_2 then
14:          α = 20°
15:       else
16:          α = 10°
17:       end if
18:       β_x = |arctan (x/z)|
19:       β_y = |arctan (y/z)|
20:       if β_x > α or β_y > α then
21:          Move(x,y)
22:       else
23:          Descend()
24:       end if
25:    end if
26: end while
27: Descend and disarm UAV
```

Table 8.1: Parameter values adopted regarding Algorithms 8.1 and 8.2.

Altitude threshold z_1	0.30 m
Altitude threshold z_2	13 m
Virtual border angle α	$\{10°, 20°\}$

In the description above, a virtual border is mentioned. This border defines an area which should enclose the marker (illustrated in Figure 8.3). This virtual border is created in order to distinguish the two cases of (i) descending and (ii) moving horizontally. The size a of this square is defined as:

$$a = 2 \times \tan(\alpha) \times h$$

where h refers to the relative altitude of the UAV.

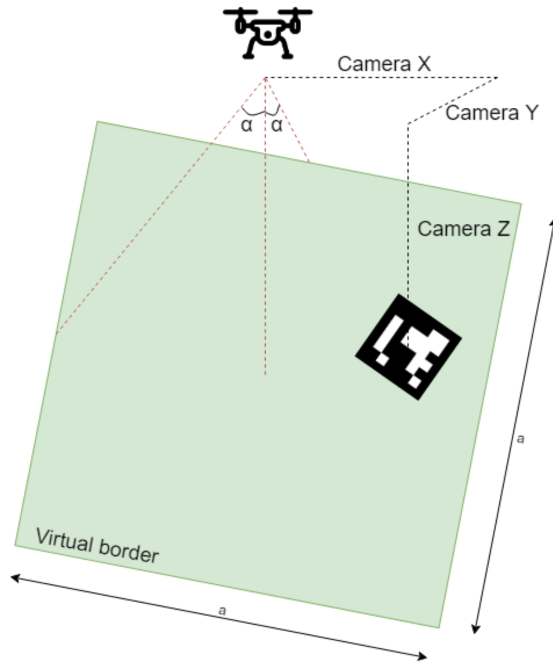


Figure 8.3: Visual representation of the virtual border.

The main advantage of defining the area in this way is that it will decrease as the drone lowers its altitude. Therefore, the drone will be more centered above its

target position when it flies at low altitude. However, when flying at higher altitude, the drone should descend whenever possible to avoid excessive landing times. For this reason, α is increased to 20° if the UAV is flying above 13 meters (z_2). However, in preliminary experiments we observed that this algorithm experiences difficulties. Specifically, it would change quickly between marker IDs whenever one of the markers is not visible for a short amount of time. At first, a small timer was used to eliminate this problem. However, this approach was not satisfactory, and so an extension of this algorithm is proposed as Algorithm 8.2. This second proposal uses the same general ideas, but introduces some improvements. In particular, the altitude *(activationLevel[i])* is now saved whenever a new marker is detected. Contrary to the former version, the algorithm will only switch between markers when its current altitude is lower than half of the *activationLevel[i]* value. With this modification, the typical glitching behavior (of detecting and not detecting a marker) is eliminated. Furthermore, if the UAV is not able to detect a marker, it will switch to a recovery mode. This means that it will increase its altitude by one meter, thereby increasing the chances of finding the marker again. Finally, the horizontal speed of the UAV depends on the horizontal distance to the marker. The pitch and roll values are set to v_1 (see empirical values in Table 8.2) if the distance between marker and UAV is greater than 1 meter (see Table 8.1); otherwise, it is set to v_2.

Table 8.2: Speed values adopted regarding Algorithm 8.2.

speed v_1	15%
speed v_2	5%
speed v_3	10%

Since the area captured by the camera is large when flying at high altitudes, there is less risk of missing the marker. Therefore, the UAV's descending speed can be varied with respect to its altitude according to equation 8.1:

$$descending\ speed = \begin{cases} min\,(60\%, altitude \times 2\%)\, if\ altitude > 6\, meters \\ v_3\ otherwise \end{cases} \tag{8.1}$$

8.2 Experiments & results

Once we finished the implementation of our solution in ArduSim, we performed three sets of experiments with a real multicopter. Due to the vast amount and diversity of UAV models available, it is worth mentioning the actual characteristics of the

Algorithm 8.2 Adaptive vision-based landing strategy.

1: Start timer 30 s
2: **while** altitude $> z_1$ **do**
3:   IDs $\leftarrow$ Search
4:   **if** $\neg$ detected **then**
5:     Recover
6:     **if** timer exceeded **then**
7:       Abort
8:     **end if**
9:   **else**
10:     reset timer
11:     **for all** IDs **do**
12:       **if** First time detected **then**
13:         activationLevel[i] = altitude
14:       **end if**
15:       **if** altitude ¡ activationLevel[i]/2 **then**
16:         id $\leftarrow$ i
17:       **end if**
18:     **end for**
19:     Get $P(x, y, z)_{id}$
20:     **if** z $> z_2$ **then**
21:       $\alpha = 20°$
22:     **else**
23:       $\alpha = 10°$
24:     **end if**
25:     $\beta_x = |\arctan(x/z)|$
26:     $\beta_y = |\arctan(y/z)|$
27:     **if** $\beta_x > \alpha$ **or** $\beta_y > \alpha$ **then**
28:       Move(x,y)
29:     **else**
30:       Descend(speed)
31:     **end if**
32:   **end if**
33: **end while**
34: Descend and disarm UAV

UAV used for our experiments. The UAV adopted belongs to the VTOL category, more commonly known as a multirotor UAV. In the experiments described in this work, a hexacopter model is used (see Figure 8.4). Our hexacopter is equipped with a remote control operating in the 2.4 GHz band, a telemetry channel in the 433 MHz band, a GPS receiver, a Pixhawk flight controller, and a Raspberry Pi with external camera. The Raspberry Pi creates an ad-hoc WiFi connection in the

5 GHz band, which is used to communicate with a ground station, or with other UAVs.

Figure 8.4: Hexacopter used in our experiments.

In the first set of experiments, the UAV was instructed to fly up to an altitude of 20 meters, to move toward a specific GPS location, and to land automatically (by giving the flight controller full control) once that position was reached. During these experiments, the landing time was recorded, as well as the actual landing position. Those experiments, which do not use our protocol, are used as reference.

In the second set of experiments, the landing accuracy of Algorithm 8.1 was evaluated. Again, the UAV took off until an altitude of 20 meters was reached, and then flew towards the target GPS location. The largest available marker (56×56 cm) was placed at that location, and the UAV used this marker as the initial reference point for landing. When the UAV was able to detect the smaller marker (18×18 cm), it used that marker as the reference point instead. For these experiments, the descending speed was defined by lowering the throttle by 10%, and the roll and pitch values were set to a value of 5%. After each experiment, the landing time and the distance between the marker and the actual landing position were recorded. We define the landing time as the time interval from the moment when the UAV detected the largest marker until the time when the landing procedure was finished.

In the last set of experiments, the landing time of Algorithm 8.2 was measured. The markers used were the same. However, in this optimized version, the descending, roll and pitch values were dynamic, as mentioned before.

Without the use of our approach, the UAV was able to land consistently within a time span ranging from 27 to 30 seconds. Nonetheless, this rapid landing comes

at a price. As shown in Figures 8.5 and 8.6, the actual landing position varies substantially, ranging from a maximum error of 1.44 meters to a minimum error of 0.51 meters; the mean value for our experiments was of 0.85 meters. Notice that these errors are smaller than expected (1-3 meters). This is most likely due to the small travelled distance between the takeoff and landing locations. In fact, in these experiments, the UAV flew for only 14 meters, and the total flight time was of about 52 seconds. Longer flights will introduce higher errors, as reported in the literature [47].

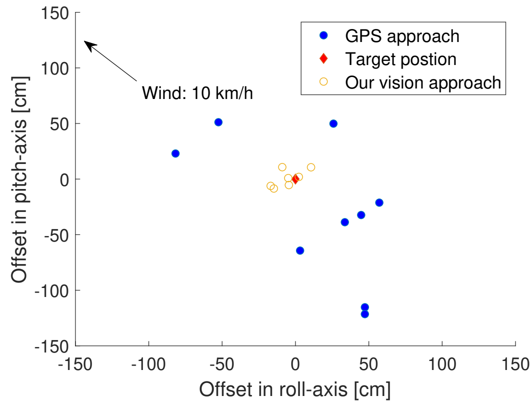


Figure 8.5: UAV landing position comparison.

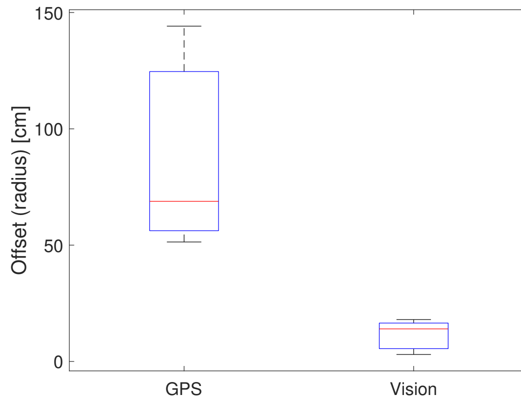


Figure 8.6: Landing offset GPS vs visual based approached.

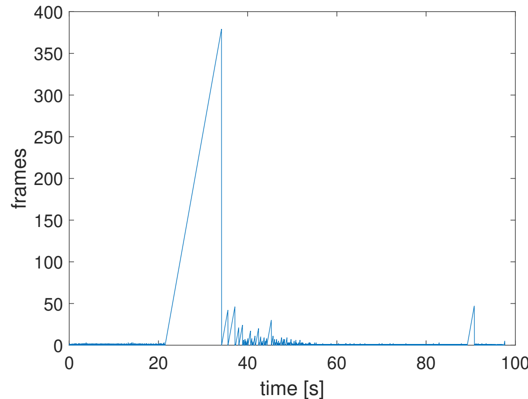


Figure 8.7: Number of consecutive dropped camera frames using algorithm 8.1

As shown in Figure 8.5, the landing position accuracy increased substantially when algorithm 8.1 was adopted. In particular, experiments showed that the error ranged from only 3 to 18 cm, with a mean value of 11 cm (see Figure 8.6). Overall, this means that the proposed landing approach is able to reduce the landing error by about 96%. However, in three out of ten of the experiments performed, the UAV moved away from the marker due to the effect of wind. Since at these moments the altitude was already quite low, the UAV could no longer detect the marker, causing the mission to be stopped after 30 seconds. Furthermore, the average landing time was increased to 162 seconds. This is due to the fact that, during the transition from one marker to another, the algorithm experienced problems at detecting the smaller marker during some time periods, as illustrated in Figure 8.7 (from second 22 to 35).

Besides this malfunction situations, the UAV showed a smooth landing trajectory as shown in Figure 8.8. Notice how the UAV makes more aggressive adjustments in the X axis when the altitude drops below 13 meters; this is due to the fact that the parameter α becomes smaller, restricting the error range. If the malfunction cases are removed, the average descending speed was of 0.3 m/s, which could be considered too conservative. Furthermore, Figure 8.8 shows that most of the adjustments are made when the UAV is close to the ground (constant altitude). This can be better observed in Figure 8.9, where the center of the camera frame and the actual location of the marker with regard to both axes is plotted ($beta_x$,$beta_y$). It can be seen that the drone only moves when the $beta_x$ or $beta_y$ angle exceeds the value of α. The range of estimated values captured is shown in Figure 8.10; we can observe that there is a higher variability in the X axis due

to wind compensation requirements along that direction during the experiments, something that occurs to a much lower extent for the Y axis.

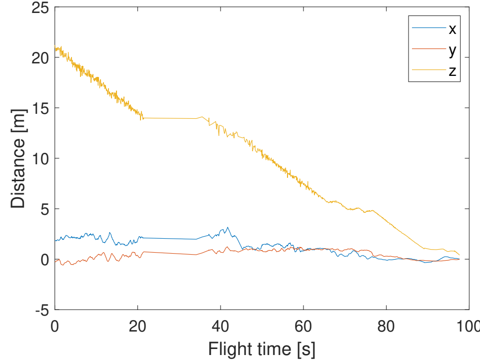


Figure 8.8: Displacement using Algorithm 8.1.

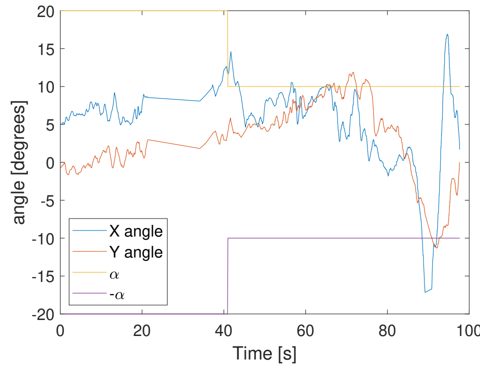


Figure 8.9: $beta_x$ and $beta_y$ angle variations vs. flight time.

The malfunctions of Algorithm 8.1 are solved in the second version. As shown in Figure 8.11, Algorithm 8.2 does not have any issues when switching between markers. Therefore, the landing time is decreased significantly to an average of only 55 seconds. Furthermore, as shown in Figure 8.12, the UAV is descending faster, which also contributes to reducing the landing time. The decrease in landing time did not have any effect on the accuracy of the application. The new recovery mode was also found to be beneficial to ensure that the UAV landed on the marker every time. To compare our proposed solutions, table 8.3 summarizes the results obtained, highlighting the main differences between them. Finally, an illustrative

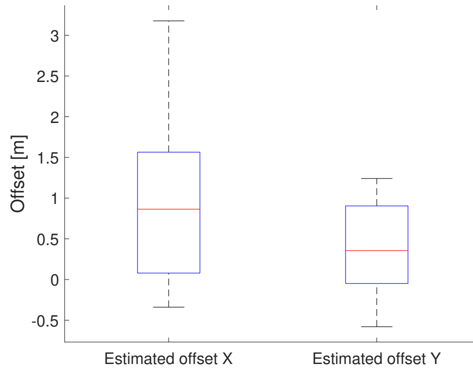


Figure 8.10: Estimated X and Y variations associated to UAV positions during landing.

video[1] has been made available to show how the proposed solution performs in real environments.

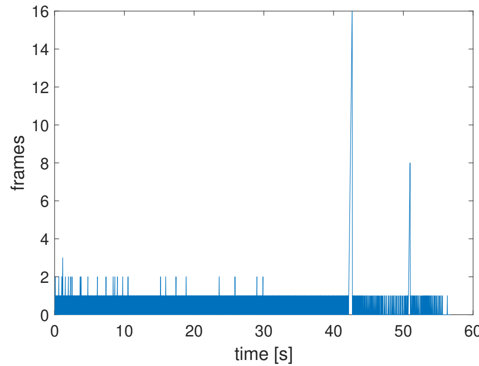


Figure 8.11: Number of consecutive dropped camera frames using algorithm 8.2.

8.3 Summary

Achieving accurate landing of multirotor UAVs remains nowadays a challenging issue, as GPS-based landing procedures are associated with errors of a few meters even under ideal satellite reception conditions. In addition, GPS-assisted landing

[1]https://youtu.be/NPNi5YC9AeI

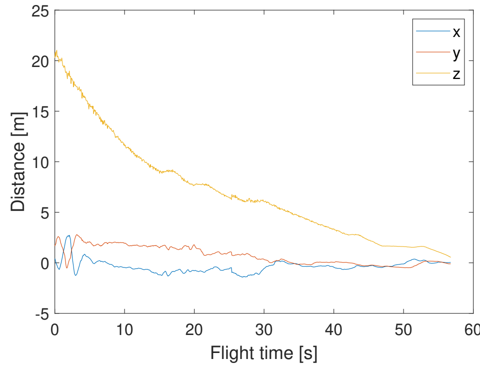


Figure 8.12: Displacement using Algorithm 8.2.

Table 8.3: Comparative table of the different schemes.

Source	accuracy [m]	Landing speed [m/s]	Maximum altitude [m]
Ours/dynamic	0.11	0.3	30
Ours/static	0.11	0.1	20
GPS-based	1-3	0.6	∞

is not an option for indoor operations. To address this issue, in this chapter, a vision-based landing solution that relies on ArUco markers is presented. These markers allow the UAV to detect the exact landing position from a high altitude (30 meters), paving the way for sophisticated applications including automated package retrieval, or the landing of large UAV swarms in a very restricted area, among others.

Experimental results using a real UAV have validated the proposed approach, showing that accurate landing (mean error of 0.11 m) can be achieved while introducing an additional (but small) time overhead in the landing procedure compared to the standard landing command.

Chapter 9

Conclusions, Future Work and Publications

Unmanned Aerial Vehicle (UAV), more commonly known as drones, are starting to become part of our day-to-day lives. They are already used frequently for photography, inspections, environmental monitoring, etc. However, due to their limited battery lifetime, it is complicated for them to cover a large area. Nevertheless, the need to cover a large area does exist. In order to solve it, multiple UAVs can be used in a coordinated fashion. When we let those UAVs communicate with each other, and take autonomous decisions, we can speak of a swarm of UAVs. A swarm has various advantages over a single UAV: it improves redundancy, increases coverage, increases maximum payload weight, is more flexible, etc. However, these advantages can only be fully leveraged if the swarm is managed correctly. In order to do this, some challenges needed to be solved. These challenges include: swarm formation definition, take-off procedure, in-flight coordination, handling the loss of swarm elements, communication between the different swarm elements, swarm layout reconfiguration, and accurate landing. In this thesis, we addressed these different challenges. Hence, we now summarize our overall findings, provide some ideas for future work, and go over the publications that have been produced as a result of this thesis.

9.1 Conclusions

We started out by improving a previously developed simulation platform called ArduSim. We split ArduSim into various self-contained modules that communicate with each other via TCP/IP so as to achieve a micro-services architecture. This major change made Ardusim (i) more flexible, (ii) easier to connect with external programs, (iii) easier to understand, (iv) independent of a specific programming language, and (v) able to be executed in a distributed fashion.

Besides updating our simulator, we provide solutions for some of the challenges that every UAV swarm will face during a flight. We started out by investigating how to assign the UAVs on the ground a position in the air and, most importantly, which UAV has to go to which airborne position. To that end, we experimented with three algorithms: a brute-force algorithm that naively searches for the best solution by testing all possible solutions; a heuristic that simplifies the problem and, therefore, is able to calculate an assignment very quickly; and finally the Kuhn-Munkres algorithm (KMA). We came to the conclusion that the KMA is the best approach as long as the number of UAVs does not exceed 500. For larger swarms, our heuristic is a good alternative.

Afterward, we investigated various proposals to take off all these UAVs. We implemented three procedures: sequential, fast-sequential, and a semi-simultaneous procedure. We compared the different procedures, and found that our semi-simultaneous procedure significantly reduce the take-off time while maintaining a safe distance between the UAVs.

Once the UAVs are in the air, we want to fly them from one point to another. However, due to small disturbances (e.g. one UAV flies slight faster than the others), the swarm might become disrupted over time. Hence, to prevent this incorrect behavior, we need to synchronize the swarm at each waypoint along the path. To do so, we introduced and improved the Mission-Based UAV Swarm Coordination Protocol (MUSCOP). Our improved version made the protocol more robust, and it only introduces a minimal time overhead in the worst-case scenarios.

Furthermore, we created an algorithm that allows us to change the formation of a swarm mid-flight. Our algorithm takes the flight direction of the UAVs into account and, based on their direction, divides the UAVs in groups. Each group will change altitude momentarily in order to safely cross from one side to another.

We also implemented an algorithm that will automatically change the altitude of a UAV in such a way that its above-ground altitude stays consistent. This is a relevant proposal because, by default, the UAV will not take the terrain level into account. When not taken care off, this can easily lead to a crash, especially in mountainous regions.

Finally, at the end of each flight, the UAVs have to land. In order to land more

accurately on a specific location, we implemented a vision-based approach. Using a simple camera, we were able to detect ArUco markers from a reasonably high altitude (30 m), and land a UAV at the target location with an average error of just 11 cm. Compared to the traditional GPS and compass method, which has an average error of about 5 meters, our method clearly improves the accuracy.

With the above-mentioned contributions, the original goal of this thesis has been achieved, and so this dissertation can now be concluded.

9.2 Future work

For future work, there are various paths we can pursue. First of all, we can keep on improving our simulator, ArduSim. In order to bring it closer to reality, we can improve on the network modules, and include hardware-in-the-loop modules. Furthermore, in many applications, sensors will be used to measure and monitor the environment. Hence, an easy-to-use interface to couple virtual sensors to a drone can be very useful. One specific sensors that is often used is a camera, and thus creating a 3D simulation where the UAVs can fly in and observe their environment is of great help.

Besides improving on the simulator, we can also pursue the path of collision avoidance. Outside the scope of this thesis, we already worked briefly on it. There are various approaches to do so. A UAV could communicate with other UAVs, and together decide which UAV has to change path so that a collision can be avoided. Another method is based on artificial potential fields, in which the UAV repels itself from any incoming obstacle. Both methods have been tested in general, but the best of our knowledge, a method that does this on a swarm level has not yet been created. In the future, it will become important that we are able to avoid collisions between different swarms.

Yet another important path that can be taken is the one that deals with communications. In this thesis, we always assumed ad-hoc networks with all the nodes in range of each other. Other researchers are actively investigating the possibilities of using 4G/5G to connect UAVs to different infrastructures. But also within the ad-hoc network communication, we can still introduce improvements. Multi-hop communication can be very useful because, when the size of the swarm grows, there are good changes that having all elements within radio range is not possible. However, due to the quickly changing environment and fast moving drones, it is not easy to build a reliable multi-hop network. Hence, more investigations works are still needed.

Most of all, we need to start using swarms of UAVs in actual applications. These applications can range from precision agriculture, search and rescue missions,

to providing network infrastructure after disasters. The possibilities are endless. In most cases, artificial intelligence will be used to process the data efficiently. When we are able to process the data on board of the UAV, we can reduce network use and latency. For that reason, we need to investigate who we can use GPU-based platforms on board of UAVs.

9.3 Publications

This section lists the publications that have been produced as a result of this thesis, as well as some other collaborations and related publications we published during this time.

International Journals

1. Wubben, J., Hernández, D., Cecilia-Canales, J.M., Imberón, B., Calafate, C.T., Cano, J.C., Manzoni, P. & Toh, C.K. Assignment and Take-Off Approaches for Large-Scale Autonomous UAV Swarms, Transactions on Intelligent Transportation Systems. **I.F. 9.551 JCR:Q1 Category.**

2. Wubben, J., Morales, C., Calafate, C.T., Hernández-Orallo, E., Cano, J.C., & Manzoni, P.(2022). Improving UAV Mission Quality and Safety through Topographic Awareness, Drones 2022. **I.F. 2021: 5.532; JCR: Q2 Category.**

3. Sastre, C., Calafate, C.T., Cano, J.C., & Manzoni, P.(2022). Safe and Efficient Take-Off of VTOL UAV Swarms, Electronics 2022. **I.F. 2021. 2.690; JCR Q3 Category.**

4. Wubben, J., Fabra, F., Calafate, C.T., Cano, J.C., & Manzoni, P.(2021). A novel resilient and reconfigurable swarm management scheme, Computer Networks 2021. **I.F. 2021: 5.493; JCR: Q1 Category.**

5. Wubben, J., Fabra, F., Calafate, C.T., Krzeszowski, T., Cano, J.C., & Manzoni, P.(2019). Accurate Landing of Unmanned Aerial Vehicles Using Ground Pattern Recognition, Electronics 2019. **I.F. 2021: 2.412; JCR: Q2 Category.**

Internation Conferences

1. Wubben, J., Calafate, Granelli, F., C.T., Cano, J.C., & Manzoni, P.(2023). A real-time co-simulation framework for multi-UAV environments offering

detailed wireless channel models. In the IEEE International Conference on Communication (ICC), 2023, **GGS class 2.**

2. Sastre, C., Calafate, C.T., Cano, J.C., & Manzoni, P.(2022). Collision-free swarm take-off based on trajectory analysis and UAV grouping. In 23rd IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2022 (pp. 477-482). IEEE. **GGS class 3.**

3. Wubben, J., Cecilia-Canales, J.M., Calafate, C.T., Cano, J.C., & Manzoni, P.(2021). Evaluating the effectiveness of takeoff assignment strategies under irregular configurations. In 25th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT), 2021 (pp. 1-7). **Core C.**

4. Wubben, J., Aznar, P., Fabra, F., Calafate, C.T., Cano, J.C., & Manzoni, P.(2020). Toward secure, efficient, and seamless reconfiguration of UAV swarm formations. In 24th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT), 2020 (pp. 1-7) IEEE. **Core C.**

5. Wubben, J. Catalán-Gallach, I., Lurbe-Sempere, M., Fabra, F., Martinez, F.J.,Calafate, C.T., Cano, J.C., & Manzoni, P.(2020). Providing resilience to UAV swarms following planned missions. In 29th International Conference on Computer Communications and Networks (ICCCN), 2020 (pp. 1-6) IEEE. **Core B/ GGS class 3.**

6. Fabra, F., Wubben, J., Calafate, C.T., Cano, J.C., & Manzoni, P.(2020). Efficient and coordinated vertical takeoff of UAV swarms. In 91st IEEE Vehicular Technology Conference (VTC2020-Spring), 2020 (pp. 1-5) IEEE. **Core B./ GGS class 2.**

7. Wubben, J., Fabra, F., Calafate, C.T., Krzeszowski, T., Márquez Barja, J., Cano, J.C, & Manzoni, P. (2019). A vision-based system for autonomous vertical landing of unmanned aerial vehicles. In 23rd IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2019), 2019 (pp. 132-138) **IEEE Core B./ GGS class 3**.

Related publications

1. Hernández-Orallo, E., Wubben, J., & Calafate, C.T.(2023). Feasibility and performance benefits of directional force fields for the tactical conflict management of UAVs. In 23rd International Conference on Computational Science (ICCS 2023). **GGS class3.**

2. Clérigues, D., Wubben, J., Calafate, C.T., Cano, J.C, &Manzoni, P.(2023). Supporting geographically widespread UAV swarms through graph-based network relaying. 19th Annual International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT 2023). **IEEE Core B./ GGS class 3.**

3. Paul, J., Wubben. J., Zamora, W., Hernández-Orallo, E., Calafate, C.T. Valenzuela, J.L. Using UAVs for the fast detection and characterization of polluted areas. In the 2023 IEEE 97th Vehicular Technology Conference (VTC2023-spring), **IEEE. Core B/GGS class 2.**

4. Wubben, J., Calafate, C.T., Cano, J.C., & Manzoni, P.(2023). FFP: A Force Field Protocol for the tactical management of UAV conflicts. Ad Hoc Networks, 2023. **I.F. 2021: 4.816; JCR: Q2 Category.**

5. van der Veeken, S., Wubben, J., Calafate, C.T., Cano, J.C., Manzoni, P. & Márquez, J.M. (2021). A Collision Avoidance Strategy For Multirrotor UAVs Based On Artificial Potential Fields. In 18th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks (PE-WASUN), 2021 (pp.95-102).

Acronyms

A

AGL	Above Ground Level

C

CSTH	Collisionless Swarm Take-off Heuristic
GSC	Ground Station Control

D

DEM	Digital elevation model
DTM	Digital Terrain Model
DTD	Divergent Trajectory Detection
DSM	Digital Surface Model

E

EASA	European Union Aviation Safety Agency
ED-CST	Euclidean distance-based Collisionless Swarm Take-off
ESC	Electronic Speed Controller

G

GSC	Ground Station Control
GNSS	Global navigation satellite system
GPS	Global Positioning System

H

HALE	High Altitude, Large Endurance

I

INS	inertial navigation sensors

K

KMA	Kuhn-Munkres algorithm

L

LALE	Low Altitude, Large Endurance
LASE	Low Altitude, Short Endurance

M

MALE	Medium Altitude, Large Endurance
MAV	Micro Aerial Vehicle
MUSCOP	Mission-Based UAV Swarm Coordination Protocol

P

PID	proportional–integral–derivativel

R

RSR	Restricted Search Range

S

SITL	Software In The Loop

U

UAV	Unmanned Aerial Vehicle

V

VTOL	Vertical Take-off and Landing

Bibliography

[1] A. Tahir, J. Böling, M.-H. Haghbayan, H. T. Toivonen, and J. Plosila. "Swarms of Unmanned Aerial Vehicles — A Survey". In: *Journal of Industrial Information Integration* 16 (2019), pp. 1–7. ISSN: 2452-414X (cited on p. 2).

[2] G. Chmaj and H. Selvaraj. "Distributed Processing Applications for UAV/drones: A Survey". In: *Progress in Systems Engineering.* Ed. by H. Selvaraj, D. Zydek, and G. Chmaj. Cham: Springer International Publishing, 2015, pp. 449–454. ISBN: 978-3-319-08422-0 (cited on p. 2).

[3] Pixhawk. *Pixhawk: The hardware standard for open-source autopilots.* https://pixhawk.org/. Accessed: 09/02/2023. 2021 (cited on pp. 10, 99).

[4] Holybro. *Holybro.* https://holybro.com/. Accessed: 2023-03-23 (cited on p. 11).

[5] PX4. *Open Source Autopilot For Drone Developers.* https://px4.io/. Accessed: 2023-03-23 (cited on p. 11).

[6] A. D. Team. *ArduPilot: Copter documentation.* https://ardupilot.org/copter/. Accessed: 09/02/2023. 2021 (cited on pp. 11, 20, 99, 103).

[7] L. Meier. *MAVLink Developer Guide* (cited on pp. 12, 20).

[8] EASA. *on the rules and procedures for the operation of unmanned aircraft* (cited on p. 12).

[9] EASA. *on unmanned aircraft systems and on third-country operators of unmanned aircraft systems* (cited on p. 12).

[10] EASA. *Easy Access Rules for Unmanned Aircraft Systems*. Accessed: 2021-10-26. 2021 (cited on pp. 12, 100).

[11] EASA. *Easy Access Rules for Unmanned Aircraft Systems (Regulation (EU) 2019/947 and Regulation (EU) 2019/945)*. https://www.easa.europa.eu/en/downloads/110913/en. Accessed: 16/03/2023. 2022 (cited on p. 12).

[12] Presagis. *The UAV CRAFT customizable Unmanned Aircraft Vehicle (UAV) simulator*. https://www.presagis.com/en/product/uav-craft/. Accessed: 2022-07-19 (cited on p. 15).

[13] squadrone system. *UAV simulator a drone simulator solution that accelerates the deployment of autonomous drones*. https://squadrone-system.com/en/solutions/uav-simulator/. Accessed: 2022-07-19 (cited on p. 15).

[14] Quantum3D. *Quantum3D Training. simulation. Technology.* https://quantum3d.com/uav-simulator. Accessed: 2022-07-19 (cited on p. 15).

[15] Mathworks. *UAV toolbox - Design, simulate, and deploy UAV applications.* https://www.mathworks.com/products/uav.html. accessed: 2022-07-19 (cited on p. 15).

[16] Microsoft. *Airsim home*. https://microsoft.github.io/AirSim/. Accessed: 2023-03-23 (cited on p. 15).

[17] Microsoft. *Project AirSim for aerial autonomy*. https://www.microsoft.com/en-us/ai/autonomous-systems-project-airsim?activetab=pivot1:primaryr3. Accessed: 2023-03-23 (cited on p. 15).

[18] F. Causa and G. Fasano. "Multiple UAVs trajectory generation and waypoint assignment in urban environment based on DOP maps". In: *Aerospace Science and Technology* 110 (2021), p. 106507. ISSN: 1270-9638. DOI: https://doi.org/10.1016/j.ast.2021.106507 (cited on p. 16).

[19] X. Fu, P. Feng, and X. Gao. "Swarm UAVs Task and Resource Dynamic Assignment Algorithm Based on Task Sequence Mechanism". In: *IEEE Access* 7 (2019), pp. 41090–41100. DOI: 10.1109/ACCESS.2019.2907544 (cited on p. 16).

[20] N. Dousse, G. Heitz, and D. Floreano. "Extension of a ground control interface for swarms of Small Drones". In: *Artificial Life and Robotics* 21.3 (2016), pp. 308–316. ISSN: 1614-7456. DOI: 10.1007/s10015-016-0302-9 (cited on p. 16).

[21] E. B. Dano. "Resilient system engineering in a multi-UAV system of systems (SoS)". In: *ISSE 2019 - 5th IEEE International Symposium on Systems Engineering, Proceedings* (2019). DOI: 10.1109/ISSE46696.2019.8984509 (cited on p. 16).

[22] M. Chen, H. Wang, C.-Y. Chang, and X. Wei. "SIDR: A Swarm Intelligence-based Damage-Resilient Mechanism for UAV Swarm Networks". In: *IEEE Access* PP (Apr. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2989614 (cited on p. 16).

[23] V. T. Hoang, M. D. Phung, T. H. Dinh, Q. Zhu, and Q. P. Ha. "Reconfigurable Multi-UAV Formation Using Angle-Encoded PSO". In: *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*. 2019, pp. 1670–1675 (cited on p. 16).

[24] M. Chen, F. Dai, H. Wang, and L. Lei. "DFM: A Distributed Flocking Model for UAV Swarm Networks". In: *IEEE Access* 6 (2018), pp. 69141–69150 (cited on p. 16).

[25] V. Casas and A. Mitschele-Thiel. "Implementable Self-Organized Flocking Algorithm for UAVs Based on the Emergence of Virtual Roads". In: *Proceedings of the 6th ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*. DroNet '20. Toronto, Ontario, Canada: Association for Computing Machinery, 2020. ISBN: 9781450380102. DOI: 10.1145/3396864.3399702 (cited on p. 16).

[26] Y. Tang, Y. Hu, J. Cui, F. Liao, M. Lao, F. Lin, and R. Teo. "Vision-aided Multi-UAV Autonomous Flocking in GPS-denied Environment". In:

IEEE Transactions on Industrial Electronics PP (Apr. 2018), pp. 1–1. doi: 10.1109/TIE.2018.2824766 (cited on p. 16).

[27] J. Muliadi and B. Kusumoputro. “Neural network control system of UAV altitude dynamics and its comparison with the PID control system”. In: *Journal of Advanced Transportation* 2018 (2018) (cited on p. 17).

[28] X. Chen, S. K. Phang, M. Shan, and B. M. Chen. “System integration of a vision-guided UAV for autonomous landing on moving platform”. In: *IEEE International Conference on Control and Automation, ICCA* 2016-July (2016), pp. 761–766. issn: 19483457. doi: 10.1109/ICCA.2016.7505370 (cited on p. 17).

[29] E. Nowak, K. Gupta, and H. Najjaran. “Development of a Plug-and-Play Infrared Landing System for Multirotor Unmanned Aerial Vehicles”. In: *Proceedings - 2017 14th Conference on Computer and Robot Vision, CRV 2017* 2018-Janua (2018), pp. 256–260. doi: 10.1109/CRV.2017.23 (cited on p. 17).

[30] X. Chen, S. K. Phang, M. Shan, and B. M. Chen. “System integration of a vision-guided UAV for autonomous landing on moving platform”. In: *IEEE International Conference on Control and Automation, ICCA* 2016-July (2016), pp. 761–766. issn: 19483457. doi: 10.1109/ICCA.2016.7505370 (cited on p. 17).

[31] F. Fabra, C. Calafate, J.-C. Cano, and P. Manzoni. “ArduSim: Accurate and real-time multicopter simulation”. In: *Simulation Modelling Practice and Theory* 87 (July 2018). doi: 10.1016/j.simpat.2018.06.009 (cited on p. 20).

[32] A. software foundation. *Apache License Version 2.0,* (cited on p. 20).

[33] GRCDev. *ArduSim: Accurate and real-time multi-UAV simulation.* https://github.com/GRCDEV/ArduSim. Accessed: 2021-05-26. 2020 (cited on p. 20).

[34] O. Ltd. *OMNeT++, Discrete Event Simulator.* https://omnetpp.org/. accessed: July 7, 2023 (cited on p. 28).

[35] OpenStreetMap contributors. *Welcome to OpenStreetMap!* https://www.openstreetmap.org. 2017 (cited on p. 29).

[36] Ministerio de Transportes, Movilidad y Agenda Urbana. *Centro de descargas.* https://centrodedescargas.cnig.es/CentroDescargas/. 2022 (cited on p. 29).

[37] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97 (cited on p. 38).

[38] F. Fabra, W. Zamora, P. Reyes, J. A. Sanguesa, C. T. Calafate, J.-C. Cano, and P. Manzoni. "MUSCOP: Mission-Based UAV Swarm Coordination Protocol". In: *IEEE Access* 8 (2020), pp. 72498–72511. DOI: 10.1109/ACCESS.2020.2987983 (cited on pp. 75, 76).

[39] F. Fabra, C. Calafate, J.-C. Cano, and P. Manzoni. "A methodology for measuring UAV-to-UAV communications performance". In: Jan. 2017, pp. 280–286. DOI: 10.1109/CCNC.2017.7983120 (cited on p. 80).

[40] V. T. Ltd. *Litchi.* https://flylitchi.com/. Accessed: 09/02/2023. 2022 (cited on p. 100).

[41] O. A. C. N. de Información Geográfica. *Centro de descargas.* http://centrodedescargas.cnig.es/CentroDescargas/index.jsp. Accessed: 09/02/2023. 2021 (cited on p. 101).

[42] CDEMA. *Digital Elevation Models.* https://www.cdema.org/virtuallibrary/index.php/charim-hbook/data-management-book/3-base-data-collection/3-2-digital-elevation-models. Accessed: 09/02/2023. 2021 (cited on p. 102).

[43] A. Gautam, P. B. Sujit, and S. Saripalli. "A survey of autonomous landing techniques for UAVs". In: *2014 International Conference on Unmanned Aircraft Systems, ICUAS 2014 - Conference Proceedings.* 2014, pp. 1210–1218. ISBN: 9781479923762. DOI: 10.1109/ICUAS.2014.6842377 (cited on p. 115).

[44] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and R. Medina-Carnicer. "Generation of fiducial marker dictionaries using Mixed Integer

Linear Programming". In: *Pattern Recognition* 51.October (2016), pp. 481–491. issn: 00313203. doi: 10.1016/j.patcog.2015.09.023 (cited on pp. 115, 116).

[45] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer. "Speeded up detection of squared fiducial markers". In: *Image and Vision Computing* 76.June (2018), pp. 38–47. issn: 02628856. doi: 10.1016/j.imavis.2018.05.004 (cited on pp. 115, 116).

[46] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer. *ArUco: Augmented reality library based on OpenCV*. https://sourceforge.net/projects/aruco/. Accessed: 07/02/2023 (cited on p. 115).

[47] M. A. A. Careem, J. Gomez, D. Saha, and A. Dutta. "HiPER-V: A High Precision Radio Frequency Vehicle for Aerial Measurements". In: *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. 2019, pp. 1–6. doi: 10.1109/SAHCN.2019.8824903 (cited on p. 123).