TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-25-08

Implementation of a Path Planning Algorithm for UAV Navigation

Jörn Roth · Reda Harradi · Wolfram Hardt

Dezember 2025

Chemnitzer Informatik-Berichte

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Implementation of a Path Planning Algorithm for UAV Navigation

Master Thesis

Submitted in Fulfillment of the
Requirements for the Academic Degree
M.Sc. Applied Computer Science

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Jörn Roth
Date: 17.07.2025

Supervising tutor: Prof. Dr. Dr. h. c. Wolfram Hardt, Reda Harradi, M.Sc.

Abstract

With the continued move towards autonomy in space of unmanned aerial vehicles (UAVs), the requirements for the integration of state of the art path and mission planning algorithms with collision avoidance systems keep growing. There has been a lot of development in the spaces of mission planning and collision avoidance, while path planning is a broadly understood topic, but integrations of all these concepts into complete systems have been sparse. This thesis attempts to deliver such a system by implementing obstacle avoidance and multiple pathfinding algorithms and integrating them with a mission control expert system. The implementation is then evaluated for multiple performance metrics using an integrated system simulation.

Keywords: Unmanned Aerial Vehicles, Path Planning, Collision Avoidance, Pathfinding, Simulation

Contents

List of Figures

List of Tables

List of Abbreviations

ABI Application Binary Interface

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CAN Controller Area Network

CLI Command Line Interface

CLIPS C Language Integrated Production System

COAS Copter Obstacle Avoidance System

COOL CLIPS Object-Oriented Language

CPU Central Processing Unit

CTH Control Handler

EXS Expert System

FOV Field of view

FPS Frames per Second

FPU Floating-Point Unit

GCC GNU Compiler Collection

GCS Ground Control Station

GPIO General-Purpose Input/Output

GPS Global Positioning System

GUI Graphical User Interface

HITL Human in the Loop

HOTL Human on the Loop

HOOTL Human out of the Loop

HDOP Horizontal Dilution of Precision

I^2C Inter-Integrated Circuit

I^2S Inter-Integrated Circuit Sound

IEEE Institute of Electrical and Electronics Engineers

JSON JavaScript Object Notation

MAL MAVLink Abtraction Layer

MAV Mirco Air Vehicle

NMEA National Marine Electronics Association

RTL Return to Launch

UAV Unmanned Aerial Vehicle

USB Universal Serial Bus

SITL Software in the Loop

SPI Serial Peripheral Interface

SSH Secure Shell

TCP Transmission Control Protocol

TOF Time of Flight

UART Universal Asynchronous Receiver Transmitter

UDP User Datagram Protocol

UPS Updates per Second

1 Introduction

The rise of applications for unnamed aerial vehicles (UAV) seen in recent years continues. With the prevalence of affordable drone systems in multiple size classes and for a wide variety of civilian use cases [38], the technology continues its adaptation. UAVs are used for the inspection of power lines [8] and forests [22] and for assisting in emergencies [9], for surveying wildlife, [12] and in countless other theorized and realized scenarios. The improvements in machine learning combined with the growing maturity of the UAV products have caused the development of an increasing number of autonomous applications. While the flight technologies of UAVs have matured, autonomous mission planning is still in early development. The technology still has to overcome a number of hurdles before a wide adoption can occur. The challenges faced are similar to those faced by autonomous driving [61]. The technical challenges include but are not limited to the situational awareness, precise localization and mapping, route planning, and quick decision making to react to unexpected events. Among the challenges faced by these technologies are also a number of legislative, environmental [14], and social [57] concerns. To address these issues, any autonomous UAV systems must operate safely and efficiently. All autonomous drone systems, even those that do not directly interact with humans, must therefore have a mechanism to safely respond to anomalies in their mission execution and the ability to land in emergencies. The planning of an UAVs path requires the integration of mission planning with obstacle avoidance. To avoid collisions and crashes, a map of the known environment and a sensor suite to detect moving or changed obstacles must be used. Due to the introduction of a third dimension, the navigation problem on aerial systems is harder to solve than the equivalent problem for cars, which has widely accepted solutions.

1.1 Terminology

A weighted graph $G = (V, E, w)$ consists of a set of vertices or nodes V, a set of edges $E \subseteq \{(u, v)|\ u, v \in V \wedge u \neq v\}$ connecting these nodes, and their weight function $w : E \to \mathbb{R}$ that assigns each edge a numerical weight. When all values for w are positive, the graph is called positively weighted. Infinite weight can be used to indicate that an edge does not exist. Nodes that have an edge directly connecting them are called neighbors. An ordered list of nodes where subsequent entries share an edge is called a path $P \in V \times V \times ... \times V$. The weight or length of a path is defined as the sum of weights of its edges.

Pathfinding is the process of finding any path in a graph between two nodes. A

shortest path problem additionally required the assertion to hold that a found path solution has the lowest weight of all possible paths connecting the two nodes. So-called single source shortest path problems solve the shortest path problem for all elements $v \in V$ for a given $u \in V$. In such problems the shortest is often required to be finite, and consequently infinite paths do not solve them.

In the automation of processes, implementations can be placed on a range based on their level of human influence. Systems with full human control are the existing state before automation takes place. The next level consists of human-in-the-loop (HITL) systems. They are characterized by a human taking all decisions to take any action in a process. The automation is merely used to perform physically challenging, dangerous, or repetitive tasks with simple internal logic. Human-on-the-loop (HOTL) systems represent the next level of progress. These systems are capable of largely operating on their own but are fully supervised by a human that can interject to correct mistakes or solve complex problems the system is not capable of handling. The final development stage is human-out-of-the-loop (HOOTL). Systems of this category can operate without human supervision and intervention. These categories do not have strict boundaries, and with an increasing amount of automation, systems may exhibit characteristics of multiple stages. In the aerospace sector, the development stages start with the earliest human controlled planes, adding automation with the introduction of systems such as fly-by-wire systems and autopilots, which can be considered HITL systems. Current development mostly focuses on the HOTL stage. UAVs are capable of performing predefined missions on their own when under human supervision.

1.2 Problem Statement

The goal of this thesis is the implementation of a path planning algorithm for a HOTL-based UAV navigation concept. The system should be capable of planning a path towards a target position from its takeoff point within an unknown environment while avoiding potential obstacles. For this, a set of suitable hardware components has to be selected that can be used to reliably detect the environment. To allow for compatibility with a high number of UAVs an open standard should be used for the communication to with the UAV's flight controller. All software components should be run in real time on the UAV's companion board. For the purpose of mission planning, the possibility of using an existing expert system should be explored. Multiple path planning implementation algorithms should be compared in the developed system to compare possible solutions.

2 State of the Art

For the goal of achieving path planning for UAVs while avoiding obstacles, a number of technologies are required. An overarching approach to the path planning method used must be selected, and the specific set of algorithms to be employed should be narrowed down. To be able to detect obstacles, the selection of one or multiple types of sensors to use is required. Furthermore, solutions for communicating with and controlling the UAV are required. Since the field of UAV research and its predecessor, path planning for terrestrial robots, has a large preexisting knowledge base, the existing approaches to solving these challenges are explored first.

2.1 Existing Approaches

Due to the high interest in autonomous UAVs, a number of approaches exist to solve the problem of route planning and obstacle avoidance for UAV systems. [58] surveys some of the existing approaches for collisions avoidance. It compares approaches for sensor inputs as well as the avoidance implementation. [31] also surveys existing solutions. When working with multiple UAVs the collision avoidance between multiple UAVs, and the sharing of discovered obstacles between them are additional challenges not found with singular UAVs.

The first survey proposed to split all obstacle avoidance into three categories depending on how they approach the problem with the third being a hybrid approach between the first two. The more naive approach involves assuming no obstacles to exist and executing a the mission until one is detected, which is avoided while being seen and immediately being discarded when is does no longer impact the planned mission. The simplicity and following that low computation requirements are useful when working with multiple smalls UAVs [59]. These approaches have the benefit of being good at the ability to handle dynamic obstacles but suffer in more complex obstacle constellations. Since they do not build an environment map, they are susceptible to cycles in which the UAV repeatedly avoids the same obstacles only to find the previous obstacles again. These purely reactive algorithms are rarely used without incorporating at least some aspects of the second approach.

The second category is deliberate collision avoidance that keeps a memory of the environment and plans a route based on that information. This known environment can then be changed based on newly detected obstacles. Moving, especially when the movement does not follow a strictly predictable route, obstacles can not be handled efficiently. Approaches using deliberate avoidance can be further broken down depending on the navigation method used. [58] defines 4 categories: geometric,

force field base, and optimization based approaches and the sense and avoid method. While the survey focuses mainly on multi UAV operation, its concepts can also be applied to single UAVs.

Geometric approaches are suited for static obstacles and objects moving according to a fixed set of rules, e.g., with a constant velocity. The collision cone approach [16] calculates the a linear cone, which describes a subset of the possible flight parameters such as position, velocity and acceleration that lead to collision. Using these constraints, the UAV can preempt collision by ensuring its current flight parameters do not fall within this range. Other mathematical approaches to derive a similar set of constraints can be used, e.g. [40]. Using the derived set of constraints for the flight parameters and suitable a target function, an optimization problem can be solved, resulting in an optimal avoidance of any obstacle. While these approaches can be very precise, solving these problems for a high number of obstacles is often not feasible.

Force field based methods use simulated forces to attract and repel the UAV. Any obstacle detection creates a repulsive force that increases the closer the UAV gets, while the target position attracts the UAV. As [6] shows, this approach can be efficient for finding the target but it comes with a number of restrictions. This repulsive force of detected obstacles causes the UAV to avoid getting near to them. This can be beneficial when more obstacles are expected close to the ones already detected but detrimental when the best path that could be taken is close to them. The above approach is comparatively computationally easy since the calculation time only scales with the first power of the obstacle count. The placement of the obstacles is required to be able to funnel the UAV towards the target. Otherwise it can become stuck in a dead end. While this can be counteracted by adjusting the forces depending on the distance, these solutions are scenario dependent. One major modification of this concept are flow field approaches such as [13]. There the navigation problem is instead solved using methods similar to fluid simulation to create velocity vectors for a map of grid cells, which all point towards the target. While requiring more computations and memory to generate the flow field, this approach can get around the problem of forces balancing out in certain dead spots.

Optimization based approaches use statistical methods to calculate the optimal collision free path. One such possibility is the creation of a number of different random paths, of which the best one is selected. If the selected one is shown to collide with an obstacle, the next best one is selected [43] until the chosen one has no collisions. This kind of algorithm can produce similar results as geometric approaches with a lot less computation, but due to the random nature of the path creation, the worst case outcomes are far from ideal. The use of genetic algorithms [7] also falls in this category. These can be used to generate, optimize and solve a navigation graph.

The method that, on the surface level, incorporates the least amount of computing is sense and avoid. It executes the optimal flight plan until the detection of obstacles causes this plan to change. The lack of ahead of time processing allows for high update rates and quick reaction times. Due to the low amount of

data required, the use of low cost sensors is possible [26]. The use of higher end sensors, on the other hand, can also create good results since earlier obstacle detections allow for earlier avoidance, leading to a more optimal path [60] With the data collection of all detected static obstacle, a detailed obstacle map can be built that proven path generation and pathfinding algorithms can solve to arrive at an optimal solution. The drawback of this method is that in multi UAV operation the cooperation between UAVs can be difficult, but for single UAV concepts, this drawback disappears. The generation of the navigation graph for these approaches can be done in multiple ways. Waypoints can be placed around obstacles [34] or they could be generated with fixed distances in a grid [53]. Since the generation of waypoint graphs using all viable waypoints and edges can be expensive in terms of computation power and memory usage, approaches have been developed to limit the size of navigation graphs, [36] only generates a few of the possible waypoints while still achieving close to optimal resulting paths. The assumption of restriction of straight edges in waypoint graphs ignores the high maneuverability many UAVs offer. When optimizing for travel speed, the use of Bézier curves can be beneficial, as [47] shows. The advantages of planning multiple sections ahead when routing the UAV can lead to substantial speed gains. Such methods require advanced knowledge of the path ahead. Otherwise, the need for constant corrections counteracts the velocity gains, and collisions are more likely since the reaction time is reduced.

While most other concepts rely only on visible objects, [63] explores the concept of occlusion aware path planning. It also uses a designated mission target area,that has to be fully explored, not a single or multiple fixed waypoints.

The recent developments in machine learning had an impact on pathfinding and collision avoidance. The survey [44] reviews a number of approaches to integrate artificial intelligence into UAV navigation. One of the less consequential uses is the prediction of not yet detected obstacles ahead of time to already plan paths. This is a development originating based on the obstacle prediction using more traditional pattern recognition algorithms. Another use case while still relying on traditional solutions for the core path planning is the utilization of machine learning models to categorize obstacles into dynamic moving ones and static permanent ones to be handled separately by the system. Furthering the integration of machine learning, [52] reviews approaches that use machine learning models to act as an alternative to the more established pathfinding algorithms. [54] takes this even further by using the technology to also incorporate dynamic obstacles. The use of machine learning in this field has the same drawback that can be found in most of its applications, being that it does not deliver fully reliable results.

2.2 Pathfinding

Pathfinding is the search for the shortest path between a start point and a destination point. It is applied in a wide range of applications, from navigation software over computer games to network package routing. Most approaches revolve around

the use of weighted graphs $G = (V, E, w)$ with a designated start and end node $S, T \in V$ and edges that have a weight derived from the cost to traverse it. This is often but not always achieved using the Euclidean distance between the connected nodes if they have assigned position coordinates.

The 1959 published Dijkstra's algorithm [21] forms the basis of most further approaches in the field. It operates using a min-priority queue data structure Q that keeps track of nodes to be processed using the length of the shortest path as sort key. The data structure implementation of this queue is essential for time complexity of the algorithm. Fredman and Tarjan [25] proposed using a Fibonacci heap, which performs most required operations in $\mathcal{O}(1)$. Only the removal of the shortest node runs in $\mathcal{O}(log\ n)$. This results in a worst case time complexity of $\mathcal{O}(|E|+|V|log|V|)$. Initially only the start node is inserted into the queue. Then the node with the shortest path from the start is removed, and its neighbors are added to the queue. This is repeated until the target node is processed or until the queue Q is empty. In the later case there is no path between the start and target node.

Since Dijkstra's algorithm does not account for the distance to the target when expanding nodes, it suffers a loss of efficiency when applied to Euclidean maps in two or three dimensions. Nodes are expanded the in shape of a circle or sphere, which leads to the nodes not pointing towards the target often being expanded despite not being used for the shortest path. The A* Algorithm [30] uses the estimated distance to the target node to resolve these inefficiencies. To operate, the algorithm requires an heuristic function $h(n)$ that for each node returns a lower bound for the distance to the target node. Instead of the length of the shortest path $g(n)$ to the start, the next node to be expanded is then selected based on the sum $g(n) + h(n)$. In applications where nodes have coordinates, their Euclidean distance can be used as the heuristic. While the worst case performance of A* matches the Dijkstra's the average performance, depending on the heuristic used, is increased.

Both A* and Dijkstra, which can be seen as a special case of A* [1], require the graph to be fully labeled with weights before finding a solution and can not react to changes. When presented with new nodes or edges, removed nodes or edges or changed weights, the algorithm must be executed again at full computational cost.

In situations where pathfinding and mapping happens simultaneously waypoints and paths might get removed and added during the navigation to the target since previously unknown obstacles get discovered. This incurs a substantial computation time cost. Especially the longer the navigation is already being performed, the higher this downside gets.

To address these downsides, a group of algorithms under the name of D* was developed. The original D* [48] was the first to attempt solving this issue. Other than the previously discussed algorithms, it expands the search tree beginning from the target node. It is disadvantaged due to not incorporating a heuristic, leading to the same drawbacks as those experienced when using Dijkstra, that a lot of paths are expanding that point away from the start position. Instead of a single cost value

[1] with $h(n) = 0$

stored for each node, D* utilizes two costs: the current cost and the minimum cost. Both are initialized with infinity for all nodes in the first execution. The minimum cost represents the length of the path to the target in the last iteration, and the current one is the modified cost calculated in the current iteration. A difference between these costs indicates that a node still has to be expanded, which equalizes the two values. The main loop of D* expands nodes as long as some are in the open list. After each run, the navigation graph is fully expanded. When modifying the navigation graph, raise and lower events propagate through the graph. This is done by adjusting the current cost of a node to the minimum sum of distance to its neighbors and their path to the start node. All neighbor nodes that are routed through the expanded node are then added to the open list.

Focused D*[49] makes use of such a heuristic similar to A* and can therefore have improved performance depending on the heuristic used. D* Lite [33] implements the same properties, but using a smaller footprint. It has been derived from Lifelong Planning A* (LPA*) [32], which itself adjusts A* to work incrementally. One major differentiation to LPA* is that D* operates originating at the target position while LPA* basing its calculations at the start position. D* Lite uses the same approach as D* and starts at the target node. This modification is done since the distances to the target remain unchanged for all nodes that did not have a new obstacle detected on their path to the target. Since new obstacle detections are more likely to be closer to the current position than to the target, this reduces the amount of distance values having to be recomputed. Additionally, the target position will not move, unlike the start position that moves with the vehicle. Both D* Lite and LPA* use a priority queue with a two element numerical key $k = [k_1, k_2]$ which is ordered lexicographically, i.e. the second part is only taken into account if the first is equal.

$$k \odot l := \begin{cases} k_2 \odot l_2 & \text{if } k_1 = l_1 \\ k_1 \odot l_1 & \text{otherwise} \end{cases} \quad for \odot \in \{<, >, \leq, \geq =, \neq\} \tag{2.1}$$

Notably, both key components may be positive infinity, which is used to indicate that there is currently no known reachable path to them. Since infinite values of the same sign are considered equal in the IEEE 754 [1] floating point standard, all unreachable nodes are of equal priority. D*Lite terminates its expansions of the nodes in the navigation graph when it detects that an optimal path from the target position to the start is found. This improvement over D* prevents the repeated recalculation of the paths of nodes that are further from the target than the start position when their expansion is currently not required.

Algorithm	Incremental	uses Heuristic	Complexity
Dijkstra	No	No	Low
A*	No	Yes	Medium
D*	Yes	No	High
Focused D*	Yes	Yes	High
D* Lite	Yes	Yes	Medium

Table 2.1: Comparison of Path finding algorithms

2.3 Distance Sensors

For the obstacle avoidance use case, obstacle detection is a critical component. To detect obstacles and locate them, the distance between them and the UAV must be accurately measured using distance sensors [24]. To measure distances, two kinds of technologies may be used: either sound wave based methods or electromagnetic wave based methods. Since both sound and electromagnetic waves of certain frequencies can be sensed by humans, such sensors usually operate outside the human-observable spectra. The general hearing range of humans is 20-20000Hz [45]. Infrasound with a frequency of under 20Hz can generally not be used in precise measurement technologies since the wavelength is at least 16.7 meters with standard temperature and pressure. Comparatively, ultrasound sound at 42kHz has a wavelength of 8.2mm allowing precise object distance measurements. When using electromagnetic waves, the wavelengths of visible light between 400nm and 700nm are avoided. Additionally, when using short ultraviolet wavelengths or even X-rays, the radiation becomes highly energetic, which can damage or penetrate obstacles, which makes their use impractical. Both the long infrared and the microwave spectrum can be used for distance measurements but both come with drawbacks relating to their interaction with the environment. As part of the black body radiation, all objects emit infrared radiation with frequencies determined by their respective surface temperatures. This can lead to interference in these frequency ranges. The even longer waved far infrared radiation suffers from high absorption rates in atmospheric gases but can still penetrate some objects, making reliable detection above very short ranges impractical. Microwave radiation can be used for ranging [51] but suffers similar range constraints.

When using reflected waves of any kind, there are multiple ways to calculate the distance to the reflecting object. The simplest approach is the time of flight (ToF) method, where the time difference between emission of the initial wave and the detection of the reflected wave is used. This time delta is divided by two times the speed of the particular wave, resulting in the distance. This method relies on knowledge of the accurate speed of the wave. For electromagnetic waves, this is unproblematic since the speed of light is only marginally impacted by the low refraction index 1.0003 of air. The propagation of sound waves is highly dependent on temperature and further influenced by the gas composition and pressure. The

speed of sound c_{air} is directly proportional to the square root of the absolute air temperature T_{air} [20]:

$$c_{air} \approx \sqrt{T_{air}} \cdot 20.06 \; ms^{-1}K^{-\frac{1}{2}} \tag{2.2}$$

The difference between cold 0°C air and room temperature 20°C air is 11.9m/s or 3.5%. Equipping a sound based distance sensor with an air temperature sensor can therefore drastically increase its accuracy.

The influence of pressure and composition, most notably relative humidity, can be mostly ignored since the deviation is generally within 0.1m/s from the purely temperature derived speed. One other factor to be taken into account is the relative movement of the air itself, especially acceleration due to wind gusts, sensor movement, or, in the case of UAV applications, caused by the propellers.

Due to the high speed of light, ToF electromagnetic wave based sensors require an precise, high resolution timing methods capable of measuring fractions of a nanoseconds. A distance resolution of one centimeter requires a clock resolution of at least 0.03 ns.

When using optical sensors, making use of the parallax to calculate the distance based on the angle of the reflected light by placing the light emitter and receiver a known distance p apart. This method requires a sufficiently focused emission beam and the capabilities of precisely measuring the angle of the incoming light α. The distance can then be derived using the tangents of the incoming light's angle as seen in Equation 2.3.

$$d = \tan(\alpha) \cdot p \tag{2.3}$$

The precision of such a sensor depends on the distance between the light emitter and detector p and the resolution of the detector. The precision of detection also decreases with increasing ranges because the differential of the calculation rises sharply closer to 90° (Equation 2.4).

$$\frac{\mathrm{d}d}{\mathrm{d}\alpha} = \sec^2(\alpha) \cdot p \tag{2.4}$$

The use of the interference of overlapping waves for distance measurement is also possible [56]. Interferometry can measure distances with extreme precision but comes associated with high equipment costs. Only laser based methods can achieve the ranges required for obstacle detection, but the high sensor cost and size makes their use in this UAV scenario impractical.

To overcome the shortcomings of different sensors types a combination of can be used. This allows the user to take advantage of all used types with the trade-off of higher equipment and implementation costs. The sensors also have to be tested to not interfere with each other, and a solution for fusing the data has to be implemented.

Another type of sensor used for obstacle detection is cameras of visible or infrared light. They have the advantage of detecting a high resolution view of the environment, but deriving the distance of the detected obstacles can be complicated. Using multiple cameras to calculate the distance is a reliable but computationally hard

process [41]. The requirement for multiple synchronized high resolution cameras also restricts the application of this sensor configuration. Monocular depth estimation can not provide certainty about detected obstacles' distances. Traditionally, to achieve good distance estimation confidence, large machine learning models were required, but more recent developments, such as [46], show the potential to employ these techniques in lower power devices.

2.4 MAVLink

MAVLink (short for Micro Aerial Vehicle Link) is a communication protocol developed for the use in unmanned systems including MAVs and other robots. Since its release in 2013, it has been adopted as the standard for communication with open architecture drones. It specifies a binary data exchange format, comprehensive sets of messages and a number of microservices for the interaction between communication partners. There are 2 major MAVLink versions, version 1 which was released in 2013 and is still in use for older components and MAVLink 2 which has been adopted in 2017. Since then, version 2 has been used by all major systems. Version 2 comes with improvements that fix some shortcomings of MAVLink 1 and add additional features. It extends the length of the messages' ID from 1 to 3 bytes, adds additional fields to common messages and supports cryptographic message signing. To allow the support of additional future features, a mechanism of compatibility and incompatibility flags also exists. While both MAVLink versions are incompatible on a binary serialization level, all MAVLink 1 messages have a corresponding serialization in MAVLink 2.

Typically the communication happens between a GCS and one or multiple UAVs. The UAV continuously sends MAVLink messages containing its own state and sensor data, such as position data, flight mode and battery status. The GCS intern allows the monitoring and controlling of the UAV's mission by setting waypoints, starting a prepared mission, recording images and videos, and landing the UAV. Due to the open nature of MAVLink, a number of implementations exist that provide these functionalities by providing a full flight stack with firmware for UAV flight controllers and a GCS implementation. The most prominent being Ardupilot [5] with Mission Planner and PX4 [4] with PX4 Autopilot. The MAVLink protocol itself does not provide mechanisms for package routing, all packages are broadcast. While the protocol has package loss and duplication detection via packet numbering, there is no built-in support for retransmission of lost packages. For controlling key functions of the UAV, the command microservice is used. It adds improved package deduplication, target identification, and a mechanism to resend lost messages. All commands must be acknowledged by the target, indicating the command's success or failure. Addressing within a MAVLink network is done using system and component IDs. Each MAVLink system, such as a UAV or GCS, has one statically assigned system ID between 1 and 255. Each system may have up to 255 components (such as flight controllers or sensors) with the component IDs 1 to 255. All components

should announce their presence by periodically sending a heartbeat message that contains data about their component type and current status.

When controlling UAVs using MAVLink, an important mechanism for controlling them are their flight modes. These control the top-level behavior of the UAV and may restrict certain actions. A UAV in LAND mode will, for example, reject any takeoff commands. The UAV's flight mode can be changed using a command from the command microservice. When changing to certain modes, the flight controller may reject the new flight mode if the preconditions for that mode are not met. The initial flight mode the UAV will be in after it powers up is Stabilize. This mode is intended for operator controlled flight. The UAV will automatically stabilize itself by adjusting its pitch and roll angles. Autonomous flight is not possible in this mode, and any such commands will be rejected. The flight mode intended for autonomous flight and mission execution is Guided. This mode requires a precise GPS signal for the UAV to determine its position. The UAV will fly to global position waypoints that can either be loaded before takeoff or dynamically modified by the GCS during the flight based on external mission logic. When flying in Guided mode, an operator should be present supervising the active mission, ready to take over control by changing the flight mode to Stabilize and consequently landing the UAV safely. To get the UAV to return to above its takeoff position and land, the RTL flight mode can be used for both autonomous and operator controlled flight. The exact behavior in this mode depends on some parameters configured in the flight controller. First the UAV will ascend to a configured altitude, and then it will fly above either the closest rally point if one exists or its home position. Finally, the UAV descends to a configured final altitude. If the latter is set to zero, the UAV will land. Otherwise it will remain hovering. Since the UAV's home position is automatically set during takeoff, this ensures that when the home position is not modified and there are no rally points configured, that the UAV will automatically land at its takeoff position when set to this flight mode, as long as that the final RTL altitude is set to zero. In case a more immediate landing is required, for example, in case of damage to the UAV or a critically low battery, the Land mode can immediately land the UAV at its current position. This mode should be used cautiously since the current position may not be a valid landing spot, such as open water or forest, which can lead to the damage or loss of the vehicle. The AltHold and PosHold flight modes allow, as their names suggests, the UAV to keep a constant altitude or position regardless of external influences like wind. These modes can be useful in autonomous missions when the UAV is performing a task at a certain fixed altitude or position, such as a sensor measurement or an image or video recording. MAVLink standardizes a mechanism to configure UAVs using the parameter microservice. It implements an access protocol for reading and writing to a key-value storage in the UAV's flash memory. The key used is an up to 16 character long string, and the values can be floating point or integer number of different data types up to 32 bits long. Using the over 200 parameters in the typical flight controller, an accurate configuration and calibration adjusted to the UAV's mission, equipment, and environment is possible.

2.5 Expert Systems

The decision making process while controlling any aerial vehicle when executing any mission involves processing a large amount of constantly changing information and taking appropriate piloting actions. Quadrotor UAVs simplify this role greatly by not requiring the knowledge of control surfaces and their influence on the flight path. The ability to become stationary midair can also provide additional decision time in unexpected situations. Due to the lack of passengers and the comparatively low weight and cost of the vehicles, the consequences of mishaps are also reduced. Nevertheless, is automating this process a complex task, especially when having account for many potential situations. Validating that traditional sequential algorithms satisfy the specifications set for the system is a complex task that can often overlook unexpected edge cases. An alternative implementation strategy in such cases has been expert systems. These systems operate with a predefined set of rules on a dynamic knowledge base to derive output data.

CLIPS [55] is a tool developed by NASA's Johnson Space Center that that makes it possible to build such expert systems. The goal of its development from 1985 to 1996 was to provide an expert system framework not based on the LISP programming language due to portability and interoperability concerns. Therefore, CLIPS is implemented in C, for which compilers exist for virtually all systems that also provide a compatible ABI. Due to it being developed for the resource constraint hardware at the time and the advantages of modern compiler optimizations, this also allows for very high performance. Due to being designed with extensibility in mind, different input and output methods are supported, and the tool can be easily extended to support more features or different languages [39]. Using CLIPS allows the combination of rules based and object orientated programming paradigms via the CLIPS Object-Oriented Language COOL.

Facts are the basic components of the knowledge base of every CLIPS based expert system [2]. They are stored in a fact list and may be added, modified, or removed at any time. The addition of facts is done with the assert keyword and the deletion with the retract keyword; hence, these terms are used for these processes. There are two types of facts: ordered and unordered ones. Ordered facts are a series of data fields similar to a natural language sentence but may contain any data type. Unordered facts are always related to a fact template, which specifies a number of named fields such a fact has. These fields may have a fixed datatype and may have a default value. When creating an unordered fact, the name of the template used as well as the values for all fields that do not have a default value must be provided.

The main mechanism program logic of CLIPS is done is using rules. Each cycle the whole rule set is applied to the current fact list. Each rule defines a set of preconditions that control if a particular rule is fired. All conditions are applied to the current rule set, and if no condition evaluates false, the rule is fired. When that happens, a set of actions associated with that rule is executed. These can include the assertion of new facts, modification or retraction of existing facts, or functions with side effects such as text output or interaction with the environment. While the

order in which rules fire is deterministic, and it is possible to assign a priority for each rule to achieve full control over their order, this is often not desired since it turns the rule based system into a procedural one.

When handling events using a CLIPS based system, it is an important design consideration that the facts representing events should be retracted once they are handled to avoid being handled in the next cycle again.

For the AREIOM project [28], a CLIPS based expert system has already been developed [62]. For data communication with other components, it uses the IPC capabilities of the open source nanomsg library [19]. Both the addition of the facts in knowledgebase as well as well the forwarding of generated control commands are handled by seperate IPC sockets. The data input is handled by a separate fact feeder that sends new facts using the IPC mechanism. This allows different applications to interact with the EXS. Facts issued this way are assigned a timestamp field that tracks when they were issued. When duplicate facts of the same template are issued, this allows the retraction of the older ones to only have at most one of each at a time. The communication with the UAV is done using the custom `send-command` function that writes the command into a nanomsg communication buffer to be read by the MAVLink Abstraction Layer explored in the next section.

2.6 MAVLink Abstraction Layer

MAVLink provides libraries for multiple common languages, including C and Python. This provides message serialization and communication management but does not implement microservices. Application developers using the MAVLink protocol must implement the logic themselves. The success of sent messages is also not indicated in a unified way, so is the response to instantaneous actions like the change of the flight mode indicated in the response of the command microservice but the completion of movement commands must be derived manually by querying the UAV's position. To receive the message containing the current UAV position at the desired frequency, the corresponding message must be requested repeatedly.

To address these shortcomings and provide a higher level API, the MAVLink abstraction layer (MAL) [50] has been developed. Implemented in C++, it provides a higher level API that fully handles commands such as `Arm` or `MoveBy`. The MAL handles the request building and response handling and returns the command's success or failure in a unified way. To send commands to the MAL and use its responses, a vehicle must be selected. The interaction can be done using one of three implemented ones, which are the test, EXS, and CTH vehicles.

The test vehicle allows for interaction via a CLI and is intended for testing. The test vehicle supports the control commands required to fly the UAV as well as a number of predefined test missions. The supported commands are listed in Table 2.2. While it is possible to interact with it by connecting the standard input of the MAL to another process via system pipes, this is not a preferred option since it requires that the MAL and the command provider are started at the same time

and neither can be restarted without the other also being restarted. The EXS Vehicle allows for direct connections to the MAL with a CLIPS based expert system via a nanomsg IPC socket. The CTH vehicle allows a separate control handler component to communicate with the MAL using shared memory. This memory that can be written to to issue commands and read from to gather data about their outcome. The benefit of using a CTH as opposed to directly connecting is that other software components, such as monitoring programs, can connect to it. The EXS and CTH vehicles implement the same supported commands as the test vehicle with the exception of the predefined missions and the option to print debug data or reboot the flight controller.

Additionally, the RTL vehicle can be used to issue a return to launch command. This can be used as an alternative to manual operator controlled landing after an issue occurs. When used the RTL vehicle sets the UAVs flight mode to RTL which causes it to return to the its vertical home position or closest rally point and then descend slowly until it is landed.

Command	Description
`altHold`	Changes to the AltHold flight mode
`guided`	Changes to the Guided flight mode
`land`	Changes to the land Flight mode
`posHold`	Changes to the PosHold flight mode
`rtl`	Changes to the RTL flight mode
`stabilize`	Changes to the Stabilize mode
`arm`	Arm the UAV rotors
`disarm`	Disarms the UAV rotors
`kill`	Forces the UAV rotors to disarm
`takeoff`	Performs the UAV takeoff
`mission1` to `mission8`	Executes a predefined mission
`moveTo`	Move to a specified global position
`moveBy`	Move relative to the current position
`printDeviceData`	Prints the current output of the UAV
`rebootFlightController`	Reboots the Pixhawk flight controller

Table 2.2: Test Vehicle supported commands [50]

2.7 Position Data Distribution

The standard for distributing position data on Unix like systems, including Linux, Android and Mac OS, is gpsd [3]. The gpsd daemon service reads and decodes position data from connected sensors and makes it available to possibly multiple clients. The data is provided in a standardized JSON format via the fixed TCP port 2947. gpsd supports a wide range of sensors, formats, and protocols. The most commonly used ones are the text based NMEA 0813 [35] and the CAN bus compatible NMEA

2000 as well as a number of formats specific to sensor vendors. The gpsd project also implements a number of tools for reading the position data. This includes command line and graphical clients for monitoring the position data, a tool to simulate a position, and the libgps C library that can manage the communication with the gpsd daemon for reading position data. The gpsd daemon is not available on Windows.

3 Concept

For the design of a HOTL integrated copter obstacle avoidance system, the choice of hardware components to be used must be in accordance with the planned implementation design. Important considerations for the deployment on the flying UAV platform are weight and power usage, since both high weight and power consumption reduce the possible mission duration and flight performance negatively. Furthermore, the cost of the hardware used should be considered since high expenditures could also alternatively be spent on other hardware components that improve flight performance. This chapter describes the concept for the hardware and software to be used in the proposed system, the implementation plans, and the process to simulate and evaluate its performance. For the iterative evaluation of the new core component and the integrated system, a set of simulations is proposed that test the functionality while simulating different aspects of the hardware used. To save development time, all but the core software component uses existing and adjusted components that have been developed and evaluated previously. The system concept does not target any specific UAV but rather the size class of quad rotors that is capable of carrying the required hardware, with companion boards being the biggest constraint.

3.1 Hardware Selection

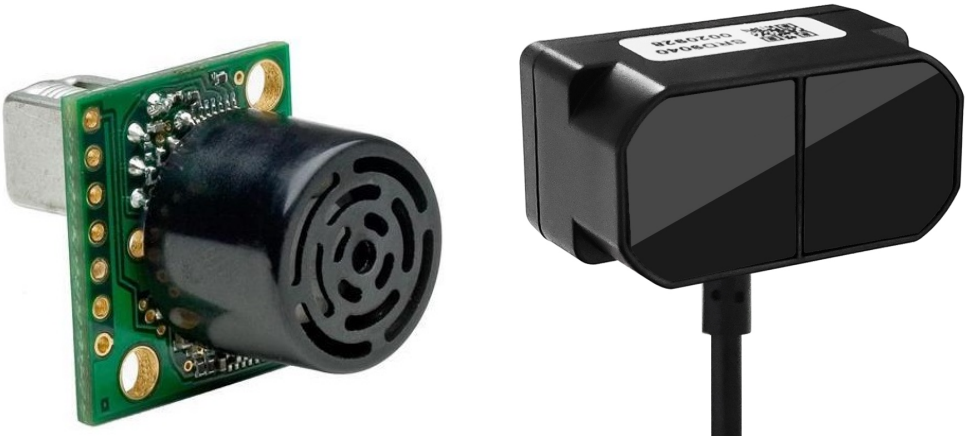
The UAV used carries a Jetson Nano companion board that can be used to control other aspects of the mission that require data processing. In addition to a Quad-core ARM CPU, it has a GPU that can be used for machine learning workloads. For the communication with external sensors, GPIO pins for UART, SPI, I^2S and I^2C communication are present. Since the Jetson Nano is powered by the system's main batteries, low power usage was a for its selection, being fulfilled by a maximum consumption of 10W. Despite this board being the development target and the only one tested, there are no restrictions that would prevent the use of a similar arm based computer running Linux as long as it has the required GPIO pins.

The selection of the sensors used for obstacle detection and their placement on the UAV is critical for the capability of the system to effectively avoid collisions while executing its mission. One important parameter before selecting and placing the sensors is the obstacle size. Thin hanging cables pose a problem in that regard since they are difficult to detect due to their size but can still cause the UAV to crash while also damaging the cable. For this reason the scope of the system should be restricted to only include obstacles of a minimum size. A value of 10cm has been

chosen, as this value includes the majority of static obstacles.

Moving obstacles are also excluded since the prediction of their movement is a complex problem that is not further explored here. Especially persons should not be in the mission area in a way that would require the UAV to actively avoid them. For the UAV a safe distance from its central position is selected, which it should always be kept obstacle free. This distance must encompass the size of the UAV, the distance around the propellers required to be free of obstacles for aerodynamic purposes, and a safety margin. The selection of this margin should take the precision of the GPS measurement, the impact of wind, the processing delays in the system, and the accuracy of the distance sensors into account.

To achieve long range obstacle detection to plan ahead while also having a wide detection angle to detect all obstacles close to the UAV, two kinds of sensors are used. LiDAR range sensors provide the long range detection capability, while ultrasonic sensors can utilize their wide detection cone to ensure the close environment of the UAV stays free of potential collision partners. The communication with both sensor types used should use the same electrical protocol, such as UART, I^2C or SPI, to simplify both the electrical and software implementation. Preferred are the bus protocols I^2C and SPI since they would allow to connecting multiple sensors to the same interface on the companion board. The sensors selected are the Maxbotix MB1212 ultrasonic sensor and the TFmini Plus LiDAR range finder, which both support I^2C (Figure 3.1).



(a) MB1212 [37] (b) TFmini Plus [11]

Figure 3.1: Distance sensors used

3.1.1 Maxbotix MB1212

Maxbotix offers a number of ultrasonic sound based range sensors for a wide array of applications, including UAVs. Their I2CXL-MaxSonar®-EZ/MB12X2 series [37] provides low cost range sensor products with a configurable I^2C interface. It operates at the inaudible frequency of 42 kHz while consuming only low amounts of power. With a size of 22 mm by 20 mm and a depth of 25 mm mounting can be done more flexibly. The low weight of 6 g also does not impact the UAV's flight characteristics.

Out of the available products in this series, the detection sensitivity varies based on the penultimate digit in the model number. The MB1202 sensor is very sensitive and can therefore detect smaller obstacles in a longer area and at a wider FOV. The MB1242 sits on the other end of the spectrum, having a narrow beam and low sensitivity. This allows it to be extremely tolerant to noise but reduces the detection range and fidelity. Since an LiDAR sensor is used in conjunction with the ultrasonic ones, the high accuracy and narrow beam are already accounted for. Just the noise tolerance is a desired feature of the lower sensitivity sensors, since for the UAV applications some level of sonic noise is to be expected from both the UAV and external sources. The best compromise is provided by the MB1212, which has a slightly reduced sensitivity. This results in a one meter shorter range and one meter slimmer beam compared to the MB1202 but also leads to the sensor being less prone to erroneous detections.

The range at which the MB1212 can detect obstacles depends on the supplied voltage, the size and material of the obstacle to detect, as well as the angle of the obstacle relative to the center line of the detection area. The sensors support 3.3 V to 5.5 V with better ranges at higher voltages. Since the results at 5 V are better than at lower voltages and the flight platform can provide it, this will be used. At this voltage the average current draw is 4.4 mA resulting in a low power consumption of 4.4 mW. Using the scope restrictions of obstacles being at least 10cm wide and assuming full sound reflectivity, the center line detection range is 4.9 m. The detection area is teardrop-shaped, originating at the sensor with a maximum FOV of 31 degrees. The beam widths depending on range for these conditions can be found in Table Table 3.1. Bigger obstacles may be detected up to 7.65m away.

Range in m	Beam width in cm	Range in m	Beam width in cm
0.3	36	2.7	240
0.6	72	3.0	254
0.9	94	3.3	248
1.2	120	3.6	238
1.5	142	3.9	224
1.8	170	4.2	204
2.1	202	4.5	150
2.4	210	4.8	60

Table 3.1: MB1212 detection area width

The data update frequency of the sensor family is recommended to be set to 10 Hz to avoid interference, but up to 40 Hz are supported for close range operations.

3.1.2 TFmini Plus

The Benewake TFmini Plus LiDAR sensor [11] produced by Benewake is a widely available one-dimensional LiDAR range sensor operating using 880 nm infrared LED light. It is capable of detecting obstacles 10 cm to 12 m away. It capitalizes on the advantages of the used LiDAR technology by having accurately detecting nonreflective obstacles with an low error margin of at most either 5 cm or 1%. This is done in a 3.6 degree cone, which reaches a radius of 38 cm at its maximum range. The sensor, which operates at 5 V, consumes an average of 85mW when in low power mode and up to 550mW when continuously measuring with the maximum frequency of 100 Hz. Its low weight of 11g makes it a good fit for weight constrained UAV applications, and the 19mm x 35mm x 21mm dimensions with integrated attachment points simplify placing it on the UAV. The TFmini Plus comes in multiple variants with UART and I^2C support. Its detections are resistant to the ambient light expected during outdoor missions. At the maximum detection range up to 70 klx, which corresponds to full vertical illumination by the sun without atmospheric interference, can be overcome.

3.1.3 Sensor Mounting

The placement of the sensor on the UAV platform depends on the mission requirements and on the available attachment points. The UAV platform has a central support structure containing the flight controller and companion board and four perpendicular beams protruding offset by 45 degrees that hold the motors and propellers.

One set of sensors of both sensor types is placed forward in the primary flight direction of the UAV. The long range LiDAR sensor can detect obstacles ahead of the copter and reroute efficiently, and the ultrasonic sensor ensures no obstacle is in

the flight path using its up to 2.5 m wide detection cone. The sensors tilt with the UAV, causing them to be pointed down during acceleration and up during braking. For the MB1212 with its high FOV and shorter range, this does not cause a problem, but the TFmini Plus readings could detect obstacles above and below the UAV and in turn miss those at its flight altitude. Depending on the magnitude of the tilt and the flight height, even the floor could be detected as an obstacle. This can be avoided by choosing a sufficiently high altitude.

To detect obstacles next to the UAV to more optimally plan alternative routes when the straight path is blocked, some partially side-facing LiDAR sensors should be used. The area of interest of these of sensors is not directly left or right of the UAV but rather left and right of the detected obstacle where the new path must be planned and the size of big obstacles must be detected. The beams holding the motors of the UAV are a good choice for this since their 45 degree angle provides the optimal balance between facing forwards and sideways, allowing them to detect obstacles in potential alternative routes. These two positions at 45 degrees left and right of the UAV heading can also be used to mount MB1212 sensors. The 45 degree separation ensures that the detection cones do not overlap, and the physical separation of the sensors themselves prevents them from interfering with each other. This placement allows for a high coverage of obstacles in the forward facing semi circle of the UAV when it is stationary and complete coverage when moving.

The property that the sensors are placed in pairs, each containing a TFmini Plus and an MB1212 is useful for a number of reasons. It simplifies the mounting mechanism, allowing for a combined design, provides redundancy in case one of the two sensors fails, and allows for a more unified sensor fusion implementation.

While this sensor layout design is optimal for obstacle detection in front of the UAV, other directions are not covered. The sides of the UAV usually get checked before the UAV reaches the current position, but the back, top and bottom are not checked. All height changes in the mission must therefore be checked manually for obstacles by a human to be obstacle-free. This includes not only altitude changes during the flight but also start and landing. When the mission is executed, the UAV must also always turn to face forwards. When one of the diagonal sensors faces forward the current flight path can be continued as long as the UAV turns to face fully forward.

All sensors are connected to the companion board via a shared I^2C bus. A passive I^2C connector board can be used to connect them all together. Since the combined maximum power draw of 3 TF Minis can exceed the power provided by the Jetson Nano's GPIO pins, they should be powered by the general 5V power that is available on UAV platform.

3.2 System Design

The chosen navigation approach is a sense and avoid concept. A navigation graph shall be generated around obstacles, and traditional pathfinding algorithms can

find the optimal path. This was chosen since it has a low base computation cost but provides reliable results and can incorporate a lot of proven algorithms into to achieve a novel concept.

The proposed system design includes a distributed architecture of components. This enables exchanging individual components without having to modify the whole system. It also provides a certain level of resilience. The system will not stay fully operational and able to execute missions in case of a component failure but depending on the failed component that failed it can be restarted or a manual takeover is possible. Due to the component separation, it can become possible to place them in different physical locations if the potential issues of network delay and data loss during transmission are further explored. This would allow the usage of more powerful hardware and easier oversight. The intended deployment, however, places all components on the UAV with all software components running the companion board. The hardware components used are the previously described MB1212 and TFmini Plus range sensors mentioned in subsection 3.1.3 and and Jetson Nano companion board.

The range sensors are connected in parallel to the I^2C GPIO pins of the Jetson Nano, and the flight controller is connected from its telemetry port to one of the Jetson Nano's USB ports. Figure 3.2 shows the mentioned hardware components and the software components running on the Jetson Nano companion board. The existing MAL and EXS components that will be modified for this implementation use nanomsg IPC between each other. The MAL opens the serial interface associated with the USB connection to the flight controller and performs the required MAVLink communications. It forwards the UAV status to the EXS and receives high level control control commands in return. These are processed, and the result is sent back to the EXS. The range sensors, which at are all configured with different I^2C addresses, are all connected to the same I^2C device. The COAS component acts as the bus master and reads all the sensors when required. The UAV requires a position sensor for its operation. This is typically a GPS receiver, but other global navigation systems or local localization systems can also be used. The flight controller fuses the sensors position together with data from its internal IMU to obtain an its own global position data including latitude, longitude and altitude as well as speed and heading. Caused by the periodically sent requests from the MAL, this data is forwarded to it for further distribution.

The current position data might be relevant to mission components not further explored here, for example, a video recording might want to attach a GPS track, or other measurements are performed at certain locations. To provide standardized access for all components, the MAL sends the position data to a gpsd service that can distribute it further to any number of clients. One of these clients is always the COAS, which uses libgps to access the data. Based on the acquired sensor data, the COAS can build an environment map with the surrounding obstacles and determine the current shortest path to the target position and transmit it to the EXS. This is done with the fact finder inside the COAS that sends CLIPS facts via nanomsg. There are three types of facts that are being transmitted: the facts

required to start a mission, the target position facts during the flight, and the mission completion fact that triggers a landing. Since the environment map of the COAS might constantly change due to previously unknown obstacles being detected or previously detected obstacles being disregarded as misdetections the shortest path to the target constantly changes. For this reason and because only one edge of the path is being transmitted at a time, the as position to fly towards must be constantly sent.

The COAS, which performs the core path planning and obstacle avoidance routines, uses a deliberate path planning approach. This was chosen since the efficient handling of static obstacles is more important than the ability to handle dynamic obstacles. These can be handled by manual intervention, especially because their unpredictability can not guarantee that automated systems would avoid them. Prioritizing reliability over more novel approaches, the path planning method used is based on long tested ideas. The obstacle detection generates a navigation graph, which is then solved for a shortest path by a pathfinding algorithm.

All four path pathfinding algorithms discussed in 2.2 should be tested to compare find the optimal choice for the considered problem. A dense navigation graph is generated using a geometric approach around all detect obstacles. This allows for the most optimal path to be found and improves the quality of the pathfinding evaluation.

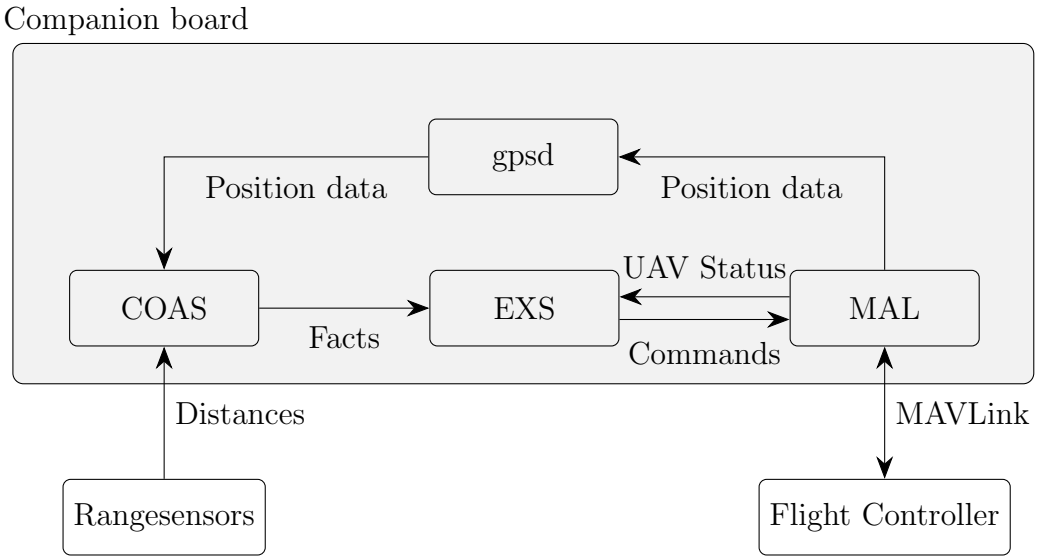


Figure 3.2: System Components

3.3 Simulation

To minimize the risk of damage to the hardware and allow for a simple development of the software components required, all system components should be validated in

simulations first. These can than be further refined to evaluate the performance of the implementations. All major flight controller software vendors, including Arducopter, provide simulation environments that allow developers to interact with a simulated UAV as if it were real. The existing MAL and EXS implementations were developed utilizing these simulations. Therefore, a simulation environment for the full system can be created with little effort. For the purpose of rapid iterative development, such a simulation is not ideal since it has quite a substantial startup time. The system design includes components exclusive to Linux based operation system, such as the MAL and gpsd. For the development process and demonstration purposes, supporting multiple operation systems is advantageous. Therefore a simulation which is implemented in exclusively within the COAS component while being operating system agnostic is also required.

3.3.1 Internal Simulation

The internal simulation can not rely on receiving position data via libgps from gpsd and can not read range sensor data. While its output may connect to an EXS, this would not be beneficial since the EXS can not communicate with an MAL under the restriction of this simulation being operating system independent. The output of the COAS must, instead of being sent to the EXS, be used to inform the data output of simulated sensor inputs. Figure 3.3 shows the data flow of the internal simulation. The simulation is configured using an initial position and a set of simulated obstacles. The test scenarios from 3.4 may be used for that purpose.

The target position output from the COAS pathfinding is used in conjunction with the current position to simulate UAV movement and update the current position as well as sending its value as simulated sensor inputs to the COAS main processing. Since all these processes are implemented within COAS, the data can be directly read from and written to memory without simulating communication protocols. The range sensor simulation uses the UAV's current simulated position and the knowledge of the location of the simulated obstacles to calculate the distance reading the range sensors, if placed as described in 3.1.3, would return. Due to this simulation being intended to test the basic properties of the COAS components algorithm with a more complex simulation providing data for the integration of the system, this simulation should be deterministic. The real range sensor sensors would provide measurements with certain levels of inaccuracy and imprecision. The simulated sensors do not need to replicate these properties. Localization systems such as GPS likewise provide only limited accuracy and may return position fluctuation on the scale of meters. This behavior should also not be replicated. These random behaviors could impact development since bugs would become hard to reproduce and the outcomes would not be deterministic even if correct, requiring performance evaluations to use bigger sample sizes to achieve statistical confidence.

Continuing the goal of making the simulation less complex, the position simulation is not required to implement higher integrals than the UAV's speed for its position. The UAV can be assumed to have infinite acceleration and turn speed as long as it

moves at a constant velocity along its designated path. One feature that this simulation can implement that would be complex to achieve for the full system simulation is that the simulation can be stopped at any time. The position simulation could set the UAV speed to 0, and all simulated aspects would stay constant. Additionally, the distance sensor input could be turned off for the UAV to reach positions that would otherwise not be visited to test edge cases in the implementation.

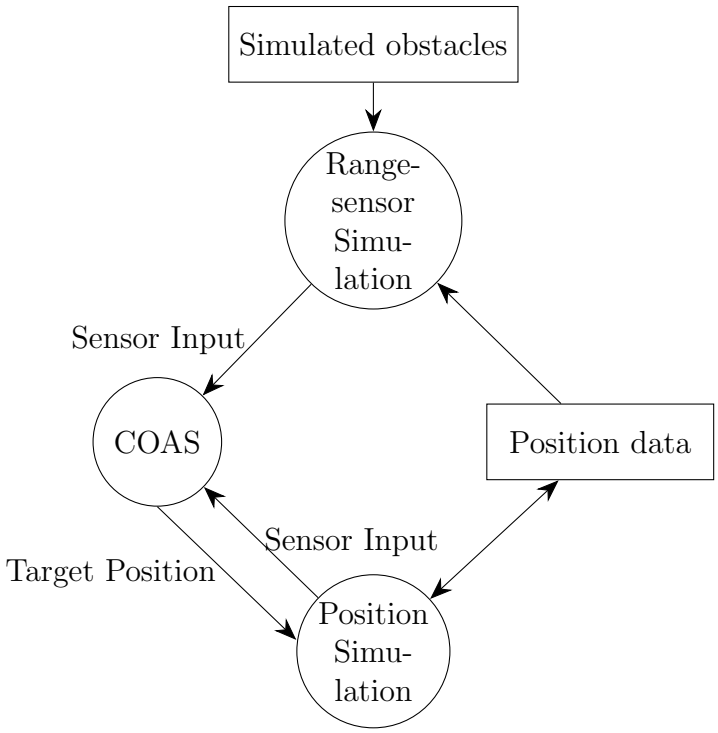


Figure 3.3: Internal Simulation data flow

3.3.2 Full System Simulation

The full evaluation of the completed system can only be achieved when it is fully deployed with a UAV in an obstacle environment. Failing to achieve that, all software components should be used with simulated inputs and outputs. The system concept proposes three hardware components on the UAV: range sensors, which can already be simulated with the internal simulation, the companion board, which acts as a generic Linux computer and the flight controller.

Ardupilot provides a Software in the loop (SITL) simulator that can run the flight controller on any PC directly to simulate a UAV. The SITL is built around the flight controller software that is compiled to to natively run on the target PC. The simulation of UAV flight is done by a physics simulation that connects to the flight controller. To interact with the SITL simulator, a MAVLink proxy is used that allows multiple connections to it at the same time. For monitoring, the simulator

connects to a satellite map of the copter's simulated position (Figure 3.4(a)) and its movement as well as a console window that shows key flight data (Figure 3.4(b)) including flight mode, altitude and airspeed. The windows also allow for limited control of the UAV, allowing the setup of specific situations to be simulated. For more complex operations, GCS software like Ardupilot Mission Planner can also be connected.

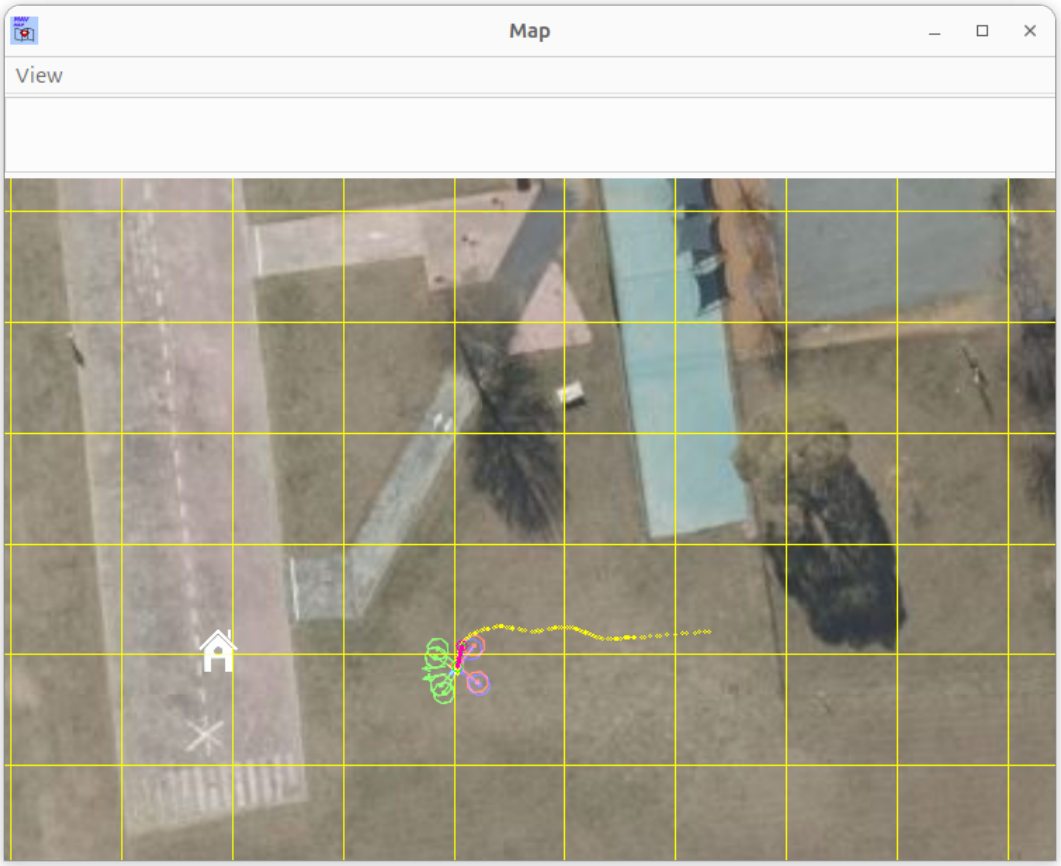
The physics simulation allows for the modeling of aspects that the internal simulation does not cover. It can implement limited turn speed and the UAV's inertia to validate if the reaction to newly discovered obstacles is sufficient or if the negative acceleration does not suffice to brake before reaching the obstacle. The integration of the MAL and EXS components is furthermore vital to not only ensure their evaluation but also COAS's operation for starting the mission with a takeoff, communicating with the MAL, and finalizing missions with a safe landing. Another benefit of this simulation is that it implements the processing and communication delay between the components expected in the deployed system. The nanomsg based communication between the COAS, EXS and MAL has a negligible delay. But the network communication between the flight controller and the MAL and the communications with gpsd might be delayed depending on the network implementation. The delay of the processing by the EXS depends on its rule evaluation rate and the flight controller transits its internal state only as often as it is requested by the MAL.

3.4 Test Scenarios

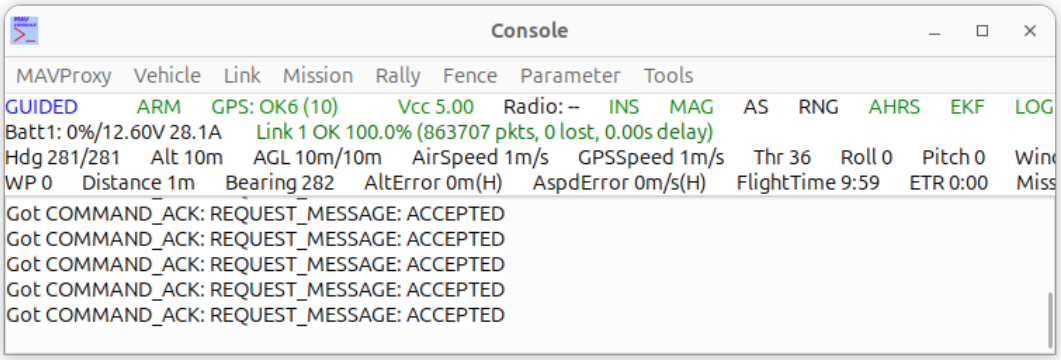
To evaluate the COAS pathfinding algorithms a number of obstacle configurations should be used. This set of scenarios as a whole should satisfy the following criteria. Firstly, it should verify the basic functionality of the system by demonstrating that the UAV can reach its target without collisions. For this , the UAV has to autonomously arm, take off, move to the target position, and then safely land and disarm while reporting its progress to the system. Secondly, some of the scenarios should be comparable in complexity to the expected real world applications. This includes a relatively low number of obstacles in a realistic pattern. For these scenarios, the discovered best path should be close to the theoretically best path. Thirdly, the execution time of these algorithms should be tested. For this, at least one scenario must be complex enough that the runtime performance becomes a relevant factor. Lastly, the scenarios should cover a diverse set of environments to include a large number of edge cases. This is necessary so that as many of the system's potential blind spots as possible may be discovered.

Four obstacle scenario templates will be used: "Empty", "Fins", "Turn around" and "Labyrinth". Each preset uses the origin $(0, 0)$ as its starting position but has a different target position. The "Empty" preset (Figure 3.5(b)) does not contain any obstacles and just the requires the UAV to move from the starting coordinates to the target position at 15.8m away at the grid coordinates $(15, 5)$. This template is used

to verify the general functionality of the implementation, as described in the first criterion, and its integration with the other software components. It only generates a start and target node and a single edge between them without the need for any rerouting due to obstacles. For this reason it does not provide any information about the algorithms' performance beyond this trivial use case and will therefore be excluded from further performance benchmarks. The "Turn Around" (Figure 3.5(b)) scenario forces the UAV to turn around after discovering the direct path to the target node to be a dead end. This is done with three rectangular obstacles. One blocks the direct path to the target, and the other two prevent simply moving around the obstacle. The "Fins" (Figure 3.5(c)) scenario contains four 10m wide and 0.3m thick walls spaced 3m apart perpendicular to the direct route from the start to target 15m away. Furthermore, three more of these parallel walls are beyond the target. This scenario is designed to test the system's capabilities to navigate in tight areas with the obstacles only 1.5m away from the path to the target. The "Labyrinth" scenario (Figure 3.5(d)) is a ten by ten grid labyrinth with the size of the square grid cells being 5m. The system is expected to guide the UAV through the labyrinth by finding a solution through it while turning around and rerouting multiple times. The optimal path only needs to visit 53 grid cells, but the system has to eliminate some wrong paths while finding the optimal route. This increases the number of unique grid cells by up to 28, with as many cells being visited twice. The approximate travel distance is 400 m. Since the system can not take advantage of knowing the grid structure of this setup, it also has to check if there are gaps in the grid walls that would provide a shorter route. Due to the effectively random nature of the already detected obstacles when reaching an intersection, the system might choose an optimal path rather than a dead end and significantly improve the travel time. But this outcome is not a property of the algorithms used and should therefore not be included in the evaluation. When running this setup using the obstacle data from a previous exploration, the system should find the optimal route is avoiding entering the labyrinth and going around. This, however, can be prevented by adding obstacles around the start and target node, forcing the only valid path to be through the labyrinth. This scenario does not represent an expected obstacle situation an UAV has to solve but is rather designed to stress test the performance of the algorithms involved with a large number of obstacles, waypoints, and paths generated.

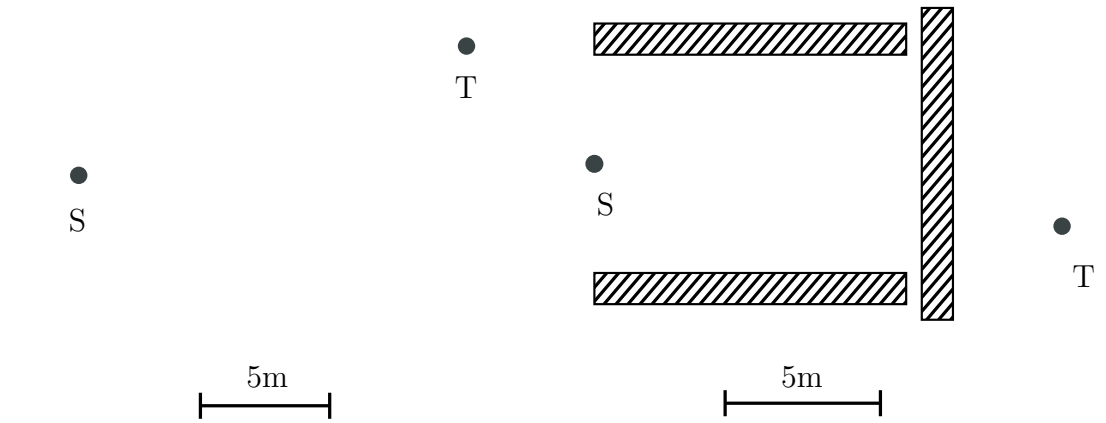


(a) Map

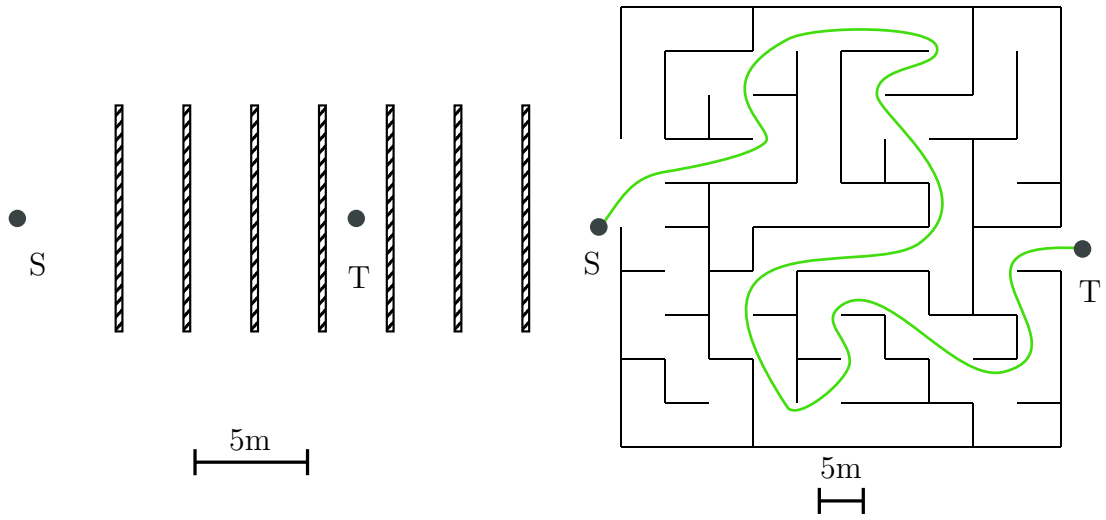


(b) Console

Figure 3.4: Ardupilot UAV simulator



(a) "Empty" (b) "Turn Around"



(c) "Fins" (d) "Labyrinth" with solution

Figure 3.5: Test scenarios with start position `S` and target position `T`

4 Implementation

The implementation is done using the hardware components selected to run on the Linux operating system on the Jetson Nano. While some components can run on other operating systems, targeting them is only an explicit goad for the internal simulation. The development of the COAS, MAL and EXS components has been done in parallel to earlier detect problems and ensure compatibility.

4.1 Copter Obstacle Avoidance System

The requirements for the implementation are cross-platform compatibility between the development environment and the deployment platform on the UAV's Jetson Nano and high performance. Both different operating systems, including Windows and Linux, as well as different processor architectures in x86 and ARM, must be supported. To fulfill these requirements, the implementation is done in a CMake based C++ project. C++ provides very high performance, only slightly being outclassed by C [42], while providing many modern ergonomic additions. Due to its interoperability with C, the compatibility of all required external libraries is guaranteed. The CMake build system supports cross platform project compilation and dependency management.

To ensure cross platform compatibility despite the use of dependencies that are exclusive to Linux operating systems and for interfacing with the position and range sensors, conditional compilation is used. The affected libgps and the linux i2c library are only included in the Linux build. The inclusion of the nanomsg communication library and the dependencies for the graphical user interface is similarly done on a per platform basis.

The COAS application, which is built as a single executable binary, is designed around three preemptively scheduled threads. The main thread performs the primary backend operation of the program, further elaborated on in section 4.1.1. The second thread is used to read the GPS sensor data. This has to be done in a separate thread since the position data provided by gpsd is written into a data buffer without synchronization. When reading data, the oldest data is read first. When not running the GPS data receiver in its own thread, a slowed down main thread could lead to old position data being read used or even the buffer filling completely causing gpsd to terminate its connection to it. Allowing the position data handler in a separate thread also improves the performance of the main thread since it does not have to wait for the GPS data to be read, potentially blocking until some is available. Since the position receiving thread will be blocked most of the time wait-

ing for new GPS data, the total computation cost of this optimization is negligible. The data is shared between the threads using a mutex to prevent data races.

To test, benchmark, and debug the implemented algorithms, an optional graphical user interface fronted is added. The user interface is optional and can be disabled when it is not required. It also runs in a separate thread from the main navigation logic to prevent performance impact on the Jetson Nano. The graphical interface, which can be disabled with a command line argument when it is not required to consume even less computational resources, is further described in section 4.1.12.

4.1.1 Backend

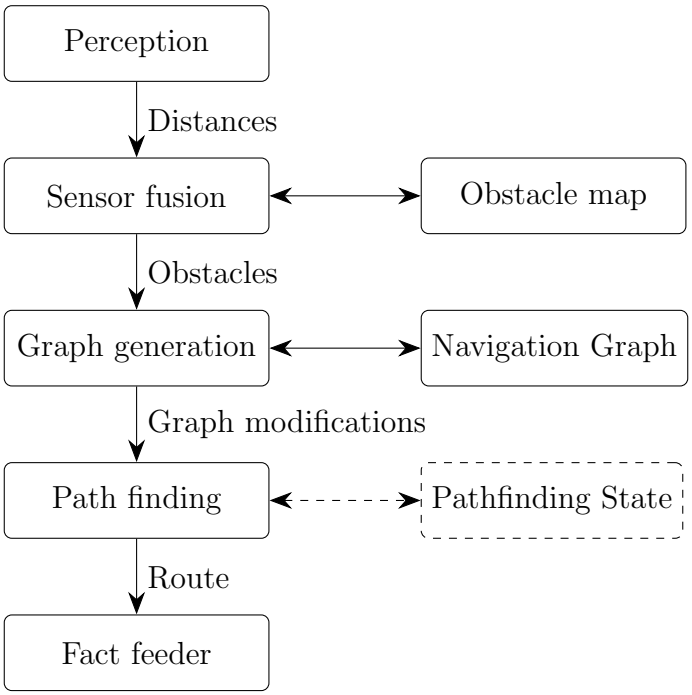


Figure 4.1: COAS data flow

The backend is the most important component of the COAS and, arguably, of the complete system. It performs the processing necessary for the UAV to avoid obstacles and navigate around them. It is built from a sequence of input, processing, and output operations that are performed periodically. The performance of the processing done by the COAS backend is critical to ensure obstacles are detected and avoided in real time. The data flow of the COAS backend is shown in Figure 4.1. It is at implemented to be reliant on the state of the previous iteration at every processing step. Only changes to the state are propagated through the processing steps. This reduces the computations of each module to be only a fraction of the total required computation in each iteration. The input sensor data read by the perception step (section 4.1.2) is used by the sensor fusion implementation, further

described in section 4.1.3, as the data becomes available. The persistent data of the obstacle map is used together with the distance data to generate new obstacles and remove old ones that are deemed to be faulty detections. These changes are written back into the obstacle map and then propagated to the waypoint generation. The waypoint generation algorithm is responsible for processing the added obstacles to modify the current navigation graph by adding new waypoints for the new obstacle, adding edges to them, removing existing edges that intersect new obstacles, and also reversing this process when obstacles are removed. It is elaborated upon in section 4.1.4. The changes to the graph, as well as a read-only reference to the graph, are then provided to the pathfinding implementation, which is described in detail in 4.1.5. Depending on the pathfinding algorithm used, the pathfinding is either operating on the full navigation graph or only on its changes compared to the previous iteration. The optimal path to the target position, which is output of the pathfinding algorithm is send provided to the fact feeder who outputs the data to the EXS.

Algorithm 1 shows the main processing loop of the backend without data flow. After reading the GPS position from the GPS receiving thread, the range sensors are read and obstacles generated. Then the waypoints are generated and the optimal path to the target is found. Since the graph generation and subsequent pathfinding processing step depend on the changes in the obstacle map they can be skipped if no new obstacles are detected or existing ones deleted. Finally the fact feeder sends the current target position to the EXS. This loop is repeated until the target is reached or the program is terminated.

Algorithm 1 COAS main loop

```
while not shouldStop do
    ReadGps()
    if DetectObstacles() > 0 then
        UpdateWaypoints()
        DoPathFinding()
    end if
    SendTargetPosition()
end while
```

4.1.2 Perception

There are two categories of data that the COAS uses as input for its processing. The first is position data that is received in a separate thread using libgps which connects to gpsd. This connection is initialized at the program's start, and if this fails, the program runs without GPS data. Therefore, the gpsd service should be prioritized to start first, which is frequently done by registering it as a system service to start on boot. Since Microsoft Windows does not support gpsd and libgps, there is no position data available when testing on Windows. The second category is distance

data read from the range sensors. The used sensor types with their I²C address and placement in clockwise rotation relative to the UAV's front heading direction can be seen in Table 4.1.

Sensor type	Placement relative to heading	I²C address (hexadecimal)
TF Mini Plus	-45°	0x71
TF Mini Plus	0°	0x72
TF Mini Plus	45°	0x73
MB1212	-45°	0x51
MB1212	0°	0x52
MB1212	45°	0x53

Table 4.1: Sensor Placement and hardware addresses

For the sensors connected via I²C, a driver had to be implemented that is capable of communicating with them. It is based on the Linux I²C bus interface implementation and therefore not compatible with other operating systems. The use of alternative dependencies for other operating systems would be possible, but since the deployment using the sensors takes place exclusively on the Jetson Nano, this is not required. The TFmini Plus uses a command protocol that allows applications to write commands to it [10]. Commands can be used to read the firmware version, reset the system settings, and configure the update rate and measurement resolution as well as the device address. Table 4.2 shows the frame format of the TFmini Plus used for command frames. These use a frame start marker and frame length to clearly identify which data belongs to the frame and an additive checksum to avoid corrupted data.

Offset	Field	Size in Bytes
0	Frame start marker 0x5A	1
1	Total frame length n	1
2	Command id	1
3	Command data	$n-4$
$n-1$	Checksum	1

Table 4.2: TFmini I²C Frame format

ID	Name	Request Parameters	Response
0x00	Read data frame	data format type	data frame
0x02	System reset n	0x60	outcome
0x03	Frame rate	16 bit frame rate	as request
0x05	Output format	data format type	as request
0x0B	Change I²C address	new I²C address	as request
0x35	Toggle low power mode	0: off, 1: on	as request

Table 4.3: Notable TFmini Commands

Since all sensors are connected to one I²C bus, their addresses are required to be different. To modify their device addresses, they use a specially defined I²C register that can be written to by connecting only one sensor at a time and addressing it with their default address. This can be done using a simple command line tool that was developed along side the COAS using the same device drivers. Since the LiDAR ranging method is fast, reading the distance from the TFmini Plus sensors can be synchronous. This is done by writing a `GET_DATA_FRAME` command to the I²C bus specifying the output format. The output as a 16bit integer with one millimeter resolution provides the highest precision without compromising on maximum detection range. After 10 milliseconds, a data frame can be read containing either a valid range measurement or an indication that no obstacle can be detected within its range of 12 000 mm.

4.1.3 Sensor Fusion

Algorithm 2 Obstacle detection function

```
function DetectObstacles()
    n ← 0
    for each s ∈ S do
        d ← S.ReadDistance()
        P* ← P_UAV + frompolar(d, s.angle + θ_UAV)
        for each o ∈ O do
            if overline(P_UAV P*) ∩ o ≠ ∅ ∧ |P*o| ≥ SAME_OBS_DIST then
                o.DecreaseConfidence()
                if o.confidence <MIN_CONFIDENCE then
                    RemoveObstacle(o)
                end if
            end if
        end for
        if d ≥ s.max_range then Continue
        end if
        if ∃o ∈ O : |oP*| ≤ SAME_OBS_DIST then
            o.IncreaseConfidence()
        else
            CreateObstacle(P*)
        end if
    end for
    return n
end function
```

The UAV is using its sensors to measure the distance to the closest obstacle in the respective directions the sensors are pointing. Using this along with the heading

direction obtained from the compass and position from the GPS module, the position of the obstacle detection can be calculated.

$position_{obstacle} = P_{UAV} + rot(s.angle + \theta_{UAV}) \cdot d$

Due to the sensors, especially the ultrasonic one, having a wide detection angle, this is not sufficient. To ensure no collisions happen, such detections must be considered to be in the whole detection area. For two dimensional calculations, this is a line that is perpendicular to the sensor direction, and for three dimensions, it expands into be a filled circle.

4.1.4 Graph Generation

Having derived an environmental map of detected obstacles to be able to use path planning algorithms, a navigation graph with waypoints and connecting weighted edges must be created. This is done in two steps. First, all required waypoints are placed, and then connecting edges are added. This process is done using Algorithm 3. The set of new obstacles is shown as O^*, and the existing ones as O. The edges are shown as the set E and the nodes of the graph as W.

Algorithm 3 Waypoints update

```
procedure updateWaypoints(O*)
    if W = ∅ then
        W ← P_0 ∪ P_t
    end if
    for all o* ∈ O*, w ∈ W do
        if |o*w| < CLEARANCE_RADIUS then
            W.remove(w)
        end if
    end for
    for all o* ∈ O*, e ∈ E do
        if |o*w| < CLEARANCE_RADIUS then
            E.remove(e)
        end if
    end for
    W* ← GenerateNewWaypoints(o*)
    GenerateNewEdges(W*)
    W ← W ∪ W*
end procedure
```

Each graph will start with two initial waypoints: the start point P_0 placed at the current position and the target P_t at the defined target position. Since the current position changes during the navigation process each time it is generated, it will be considered to be a new waypoint distinct from the previous starting position. On the contrary, the target position waypoint may be reused since it does not change its position. Obstacles may require the UAV to turn. Therefore waypoints shall be

generated around them. For two dimensional navigation using line obstacles, four waypoints are generated in pairs around the endpoints of the obstacles. In three dimensions, more waypoints are generated above and below the obstacle. They are spaced out depending on the safety margin demanded by the UAV, so moving around the obstacle in a rectangle is possible. Waypoints that are too close to any part of another obstacle are not generated since they could not safely be used. Furthermore, waypoints that would be closer than a defined minimum distance apart from each other are not generated to reduce the number of redundant waypoints. Since this may eliminate valid narrow paths between close obstacles, this value must be carefully chosen to trade off computational performance requirements and the ability to find paths in narrow spaces. The value used is 15cm. With the complete set of nodes, the edges can be generated. Algorithm 4 is used for this. This is done by checking for any possible connection of two nodes if the resulting edge intersects or gets closer than the safety margin to any obstacle. The computational cost of this operation is generally the predominant chunk of the computation time of the graph generation. Its worst case complexity is $\mathcal{O}(n^2 \cdot b)$ for n nodes and b obstacles since every connection of for every node with each other node has to be checked for up to each obstacle. With the nodes derived from the obstacles, this results in the cubic worst case complexity of $\mathcal{O}(b^3)$. The calculation time of this operation was increased by a factor of 2.38 when the order of the checks done was inverted to check the collision with newer obstacles first. The reason for this is that newer obstacles are closer to newer waypoints, increasing the likelihood that edges can be discarded early. When new obstacles are generated, all existing edges must also be checked, if they intersect with any of them, potentially removing them in the process. This has a time complexity of $\mathcal{O}(n^2 \cdot b^*)$ with b^* denoting the number of newly added obstacles. The intersection calculation itself checks if the line obstacle directly intersects the edge between the nodes, and if either of the endpoints is too close to the edge. Since nodes are not generated closer than the safety margin to obstacles, this guarantees that all generated edges adhere to the safety margin.

Algorithm 4 Edge generation

```
procedure GenerateNewEdges(W*)
    for all w ∈ W, w* ∈ W* do
        e* ← (w, w*)
        if ∀o ∈ O : |oe*| < CLEARANCE_RADIUS then
            E ← E ∪ e*
        end if
    end for
end procedure
```

4.1.5 Pathfinding

To compare the different pathfinding algorithms, the implementation uses a generic `doPathFinding` function that uses the desired algorithm based on the current user choice. This can be trivially done for the stateless algorithms like Dijkstra that take the current navigation graph as input and output the shortest path. But it requires some consideration for the iterative ones due to the D* family of algorithms storing and reusing data relating to the current path finding graph. These algorithms take the changes in the graph as input and use them to modify their internal state and output the current shortest path. Therefore, only updating the data associated with the current algorithm is not possible since, upon change in selection, the previous changes to the graph are not updated. Always updating all algorithms that require such updates would be possible but would make performance evaluation of the currently selected algorithm difficult. The implemented solution rebuilds the entire state when one of the D* algorithms is selected using the whole graph. This ensures that all data is accurate but leads to a long computation when the used algorithm is changed. Due to this potentially multi second computation in which the COAS is not performing any other actions, such a change should not be done when the UAV is in flight and near obstacles.

The two non-incremental, stateless algorithms implemented are Dijkstra and A*. Dijkstra provides a measure of the baseline level of performance for such an algorithm and is very unlikely to result in any mistakes due to its simplicity and long existence. A* is implemented as a high performance algorithm that is intended to provide the fastest results when operating on a new navigation graph. Due to the minor algorithmic modifications compared to Dijkstra, it is also expected to be reliable.

In the group of incremental search algorithms, D* and D* Lite are implemented. D* provides the baseline performance for this group and should show the potential of incremental algorithms. It is the most complex of the implemented algorithms and has also been tested in a comparatively low number of applications since it has been surpassed by other similar algorithms since its inception. One of those being D* Lite, which is intended to be an implementation that provides a very high performance while being less complex and thus easier to implement correctly and simpler to debug and test.

Each algorithm takes the current navigation graph and its delta compared to the previous execution as input. The graph is stored as a vector of PathNodes. Listing 4.1 shows the format of a PathNode with 3 local floating point coordinates and a list of neighbors stored by their index. Node indices are generally stored using the platform dependent unsigned integer type `size_t` that is used as index in array and vector operations. Its maximum value, also accessible by casing -1 is reserved to indicate a NULL value when a node does not exist. The start node is always stored first in the node list and therefore has the fixed node index 0, while the target node, which is always stored second uses the index 1. Since the start position changes with the UAV's movement all edges and relations to the start node have to be considered for reprocessing.

```
typedef struct {
  double x;
  double y;
  double z;
  vector<size_t> neighbours;
} PathNode;
```

Listing 4.1: PathNode struct definition

The used representation of the edges has the advantage that for any given node the set of edge and neighbors can be accessed in $\mathcal{O}(1)$ but requires that new edges must be duplicated as neighbors in both its ends. When a node or edge is removed, this also requires additional effort. To allow accessing a neighboring node directly by indexing into the nodes vector, the neighbors vector stores an index to it. This is required to deliver a high level of performance when iterating through neighbors but does not directly allow for the deletion of nodes. To solve this problem, nodes that would be deleted because they are either to close too an obstacle or were generated by an obstacle that was removed have all of their edges removed, a deletion flag set, and are left in the vector. This deletion flag will skip the computations that would be done when newly added waypoints check for edges to generate. As long as only a small fraction of all generated nodes gets deleted, this has negligible performance consequences. For long executions where these old nodes might become a liability, the whole graph has to be rebuilt and the internal state of D* and D*Lite algorithms has to be reset after a certain number of nodes generated is reached. The signatures of the respective path finding algorithms use same function arguments as shown in the D*Lite example in Listing 4.2.

```
vector<size_t> dStarLite(
  const vector<PathNode>& nodes,
  size_t new_nodes_count,
  const vector<pair<size_t, size_t>>& removed_paths
);
```

Listing 4.2: D*Lite path finding function signature

The `nodes` argument contains the list of nodes as described above. To signal the iterative algorithms which of the given nodes are new, the number of new nodes at the end of the nodes vector is provided. The stateless Dijkstra and A* algorithms ignore this value. All edges that got removed, including those that got removed due to one of their waypoints being deleted, are provided in a vector of a pair containing the two node indices the edge connected. This argument is also ignored by the stateless algorithms.

4.1.6 Dijkstra

Due to its an statelessness and low complexity Dijkstra algorithm is simple to implement. But due to the algorithm being less efficient than the other pathfinding algorithms, more optimization has to be done to achieve performance that allows for its use. The main optimization point is the choice of data structure to represent the list of open nodes. Initially this was implemented using the primitive C++ array vector and then as operation to get the next closest node the minimum of the distances was obtained using by linearly finding the minimum of the vector. This was chosen not only for the trivial implementation effort but also for the increased performance in low node count scenarios where the bad $\mathcal{O}(n)$ access time was compensated by the low base computation time. A more efficient $\mathcal{O}(log\ n)$ Fibonacci min heap was later used since the algorithm's performance in the Labyrinth scenario with over 3000 nodes was unsatisfactory compared to other algorithms with over 10 ms on average. After the improved heap was implemented the performance with 3000 nodes improved to 1.4 ms. The Fibonacci heap implementation used operates on a struct with two elements: the node index and its minimum path length. The heap comparisons are only done on the path length, but the index is used to identify nodes. To have an $\mathcal{O}(1)$ access to the current path length of a node, pointers to the nodes of the heap for all elements of the open list are stored in a vector indexable by the node index. For nodes that are not open, this value will be null. The memory footprint of the implementation is relatively big.

```
1 vector<int> predecessors(nodes.size(), -1);
2 vector<bool> unvisited(nodes.size(), true);
3 vector<Fnode<FEntry>*> fnodes(nodes.size(), NULL);
4 FibonacciHeap<FEntry> openList;
```

Listing 4.3: Datastructures used in the Dijkstra implementaion

The data structures used are shown in Listing 4.3 as they are initialized. The algorithm itself also uses a small amount of stack memory and the data structures used have some overhead even if empty. Since this M_0 value does not grow with node count and is comparatively small, it is not further take into account. The predecessors vector, which stores the next node on the shortest path, has a size of 8 bytes per node in the graph. The unvisited vector that keeps track of which nodes have already been expanded can take advantage of a C++ standard library optimization that packs 8 booleans in each byte in boolean vectors, taking up only one byte per 8 nodes. The fnodes vector, that stores the pointers to the heap nodes for each graph node, uses 8 bytes per graph node on 64-bit systems. On 32-bit systems this value would be halved. The open list head is stored in heap memory. For each node in the heap, 40 bytes have to be allocated in addition to the 16 bytes of memory used by the data stored in an `FEntry` struct itself. This brings the total memory consumption, depending on the node count n and the maximum number of heap entries m, to the value expressed in Equation 4.1 in bytes.

$$M_{dijkstra}(n, m) = M_0 + \lceil 16.125 \cdot n \rceil + 56 \cdot m \tag{4.1}$$

4.1.7 A*

The implementation of the A* algorithm, which is only small modification of the Dijkstra algorithm, is likewise relatively simple. The only major change is the modification of the ordering of the open list from the current path length from the start position to the sum of this length and the heuristic for the distance to the target position. Since all graph nodes have a Cartesian coordinates, the Euclidean distance can be used as a heuristic. The A* algorithm can leverage its algorithmic advantages against Dijkstra to perform on a similar level as it even without using the optimizations of an efficient open list data structure. Since it is not required to achieve acceptable performance, this optimization does not need to be applied. This helps reduce the memory footprint and the risk for bugs in the implementation. Instead, A* uses a C++ standard library vector that is kept sorted in descending order. The C++ standard sorting algorithm has a theoretical time complexity of $\mathcal{O}(n \cdot log\ n)$, but in practice the vector will always be almost sorted, resulting in a time complexity of $\mathcal{O}(n)$. Since the vector is sorted in descending order, removing the element with the shortest path can be done by popping the last element. This $\mathcal{O}(1)$ operation is done without having to move the rest of the elements or any other expensive memory operations, mitigating the effects of using a sub-optimal data structure when comparing time complexity.

To achieve better comparability between the algorithms, an A* implementation using the Fibonacci heap has been implemented and compared to the A* implementation using vector. In the evaluations used, the Fibonacci heap could not outperform the C++ standard vector. In smaller test scenarios, the higher initiation overhead and limited impact of the scaling of the time complexity produce the expect performance lead of the vector implementation. Even on the "Labyrinth" obstacle template with over 3000 nodes, it performed 27% percent worse in average pathfinding time. This surprising result is likely related to the Fibonacci heap's higher memory usage and related rate of cache misses, but further exploration of this unexpected outcome is should be done.

The memory footprint of the A* implementation in bytes is shown in 4.2. The data stored per node does not include a pointer into the open list saving 8 bytes but the A* algorithm requires 8 additional bytes per nodes to compared to Dijkstra to store the distance from the start node since it is not in the open list.

$$M_{A*}(n, m) = M_0 + \lceil 16.125 \cdot n \rceil + 16 \cdot m \tag{4.2}$$

The memory savings of not using a Fibonacci heap therefore amount to 8 bytes per graph node and 40 bytes per maximum open list size. To ensure that the A* algorithm implementation works as intended, a test suite has been built that checks for pathfinding mistakes and inconsistencies. The algorithm operates on 7 sets of nodes. Each pathfinding result must return an optimal path. The algorithm must also be stable, meaning that the result must also stay consistent between runs when multiple paths would be optimal. The tests have between two and and nine nodes and up to 12 edges. The optimal shortest path is between one and three edges long.

Additional, aa eighth test is done that checks the algorithm's behavior when no valid path can be found to reach the target. The algorithm must terminate, returning an empty path in such a case.

4.1.8 D*

The D* Algorithm is the most complex one implemented due to its design that keeps a state between runs and a lack of complexity optimizations. Its implementation takes roughly three times the lines of code compared to the stateless algorithms. Therefore, the presence of a test suite as well as extensive tests with the COAS's templates are required to ensure that the algorithm does not crash, eventually halts, and outputs a valid, optimal path. The test graphs used in the A* implementation are reused for this in addition to new test cases that test the functionalities on a changing graph. Tested are the removal of edges that previously were contained in the optimal path, the addition of new nodes, a combination of the previous two, the removal of nodes, and a 100 node 10 by 10 grid that adds a big obstacle severing multiple edges. The open list of D* is implemented the same as in A* using a C++ vector that is kept sorted, resulting in the same $\mathcal{O}(n)$ access time for removing the next node to expand. This is an even smaller problem compared to A* because D* can not naturally operate using a Fibonacci min heap. During the development a number of bugs could be eliminated using extensive testing for instance the algorithm could infinitly loop since the an set of 2 nodes seemed to improve each others optimal paths. This rare bug was caused by floating point rounding errors and prevented by adding a hard check that disallows the creation of loops in the tree of shortest paths. The earlier described issue of the iterative algorithms becoming out of synchronized due to not getting updates when not selected could also be identified during test with the full system. The memory requirement in bytes (Equation 4.3) of the implementation is worse than that of both stateless algorithms since two cost values need to be stored per node.

$$M_{D*}(n, m) = M_0 + 24 \cdot n + 8 \cdot m \tag{4.3}$$

Since D* keeps the state of the graph in storage between runs this memory stays allocated during the whole runtime of the program. The persistence of the memory and its larger size are likely to increase the number of cache misses when accessing it, leading to an increase in computation time.

The implementation of the D* pathfinding function, which has the same signature as in Listing 4.2 has to decide whether to perform an initial D* or an incremental one using existing data. An initial D*, which initializes the algorithm's state for all nodes, only has to be performed when there is no existing data, the data has been deleted due to a change in the algorithm used or when the node count decreased, indicating that the indexes used to identify nodes have been shifted. If this computationally expensive initialization does not have to be done the implementation only operates on the set of newly added nodes, which can be identified as the slice of

the given `new_nodes_count` length at the end of the nodes vector, and the removed paths. The introduction of additional nodes and associated edges connecting to the existing graph is processed by performing an edge cost decrease for all edges these nodes have with a given previous node weight of ∞. For removed egdes an cost increase operation is to ∞ is performed. This causes the algorithm to not use these edges since they can not be included in any finite paths.

4.1.9 D* Lite

The implementation of the D* Lite algorithm is done to achieve the best performance possible. The algorithm has utilizes both a heuristic and operates iterative granting it a sizable advantage over the other 3 implementations. These benefits alone are likely to make it the most performant and therefore most used algorithm when the COAS is used for UAV navigation. For this reason, the implementation is designed to achieve a result as fast as possible, not just faster than any of the other 3 implementations. Despite using more methods to reduce computation time than D* the optimizations in the algorithm's design produce a simpler implementation that is easier to maintain. Nevertheless, an extensive test suite is still required to ensure that the default algorithm operates correctly. The test used for evaluation D* can largely be reused by when changing the algorithms the optimal path is not stable when multiple options of equal length exist.

As described in section 2.2, D*Lite uses two elements for sorting the next node to process in the open list. For D* Lite the operation on the open list required are setting an elements sort value, checking if an element is already in the the list, remove an element by its node index and getting the current minimum key. It is implemented using the C++ standard map, as seen on line 8 in Listing 4.4.

```
1 class DStarLightOpenList {
2   public:
3   void set(size_t key, pair<double, double> value);
4   bool contains(size_t key);
5   void remove(size_t key);
6   pair<size_t, pair<double, double>> top();
7   private:
8   map<size_t, pair<double, double>> m;
9 };
```

Listing 4.4: D*Lite open list class definition

The `set`, `contains` and `remove` functions all have a time complexity of $\mathcal{O}(log\ n)$. The `top()` function has a time complexity of $\mathcal{O}(n)$.

The memory use of the D* Lite algorithm is distributed between the size of the rhs and g values. They take two times 8 bytes per node and the open list map. The open list implementation uses 24 bytes per entry to store data and 32 bytes as part of the data structure (Equation 4.4). The resulting memory usage of D*Lite is higher than both stateless algorithms but lower then D* in most graphs. This

memory persists for the whole use of the path finding algorithm and does not get freed after each execution.

$$M_{D*Lite}(n, m) = M_0 + 16 \cdot n + 56 \cdot m \tag{4.4}$$

4.1.10 Fact Feeder

The fact feeder component uses the computed current path as input, computes the next UAV target position, and consequently sends it to the EXS. The fact feeder operates on the COAS main thread at the last component each cycle. It has four states resulting from the Cartesian product of whether it is currently running and whether it should be running. Therefore, its state is either disabled, connecting, active, or closing. The connection to the EXS implementation is done using nanomsg IPC to send plain text CLIPS facts. To initiate missions, a set of facts further elaborated on in section 4.3 is used. The computations done by an active pathfinder are shown in Algorithm 5. The result of the pathfinding algorithm used is stored in a vector V of node indices to follow in the reversed order beginning with the target node and ending with the current start node. The penultimate node of the path vector will be used as the base for the next position to fly towards. Since the last element will always be 0, indicating the current node, and is not further needed, it will always be stripped from the list. Each time the fact feeder is executed, it checks if the UAV is sufficiently close to the next path node, and if this is the case, the last element from the current path to the target is removed so the next node can be reached. The distance check is based on the current UAV speed v_UAV and the time between executions of the fact feeder Δt. This threshold ensures that the UAV does not have to fully stop at each node. Occurrences when a single UAV update takes an unusually long time are handled by either the UAV waiting at the node until the delayed update with a new target position is sent or prevented by restricting Δt to no more than 200 ms for the purpose of this calculation. To ensure the node can still be marked as reached when the vehicle has already stopped close to it, an additional distance based check is in place that triggers at a distance of 0.2 m. The UAV speed used is likewise clamped to the range between 0.5 and 4 ms^{-1} for the purpose of this calculation.

The UAV's flight controller optimizes the flight speed based on the distance to its target position. When sending far away position, this causes the UAV to reach speeds at which the braking distance may exceed the range at which obstacles are safely detected. It also caused the UAV to follow a curved path instead of the intended straight one when turning a corner. Such a curved path is not cleared to be obstacle free and may therefore steer the UAV into obstacles. To avoid these problems, the UAV target position send is limited to be at most $d_{max} = 1\,\mathrm{m}$ from the current position away. Higher values up to 2 m have been tested to be safe, but they only result in a 17% higher travel speed while diminishing safety margins. This is achieved by checking the distance from the UAV position P_{UAV} to its the target position P_t. Should this distance exceed the defined maximum, a point d_{max} away

from the UAV in the direction of the UAV target is used instead. The current UAV target position is sent using the TargetPosition CLIPS fact template. This template requires the global position with latitude and longitude, the altitude relative to the takeoff position, and the desired UAV heading. Latitude and longitude can be calculated from the local coordinate system as seen in the equations 4.5 and 4.6 below.

$$latitude = \frac{x \cos\alpha + y \sin\alpha}{111195m} + latitudeOffset \tag{4.5}$$

$$longitude = \frac{-x \sin\alpha + y \cos\alpha}{111195m} \cdot \sin\frac{latitude \cdot \pi}{180} + longitudeOffset \tag{4.6}$$

The z-axis of the local coordinate system is directly used as altitude, and the heading h is from the target direction as seen in Algorithm 5 but must be converted to centidegrees.

Algorithm 5 Fact feeder: Target position transmission

procedure FACTFEEDER($V, P_{UAV}, P_t, v_{UAV}, \Delta t$)
  if $|P_{UAV}P_t| < v_{UAV} * \Delta t$ **then**
    $V \leftarrow V[0 \,..\, -1]$ ▷ Remove the last path element
  end if
  if $|V| \geq 1$ **then**
    $P_t \leftarrow V[-1]$
    if $|P_{UAV}P_t| > d_{max}$ **then**
      $P_f \leftarrow P_{UAV} + \overrightarrow{P_{UAV}P_t} \cdot \frac{d_{max}}{|P_{UAV}P_t|}$
    else
      $P_f \leftarrow P_t$
    end if
    $h \leftarrow$ ATAN2($\overrightarrow{P_{UAV}P_t}.x, \overrightarrow{P_{UAV}P_t}.y$)
    SENDTARGETPOSITION(P_f, h)
  end if
end procedure

If the pathfinding implementation could not find a finite path, the returned vector, which otherwise contains at least 2 elements, will be empty. The fact feeder will not send any data in such a case, and manual operator intervention is likely required. When the nanomsg based fact feeder can not be used for any reason, a backup solution is implemented and can be enabled at compile time by revealing a hidden UI window. It is designed to directly interface with the MAL and therefore does not provide the safety and reliability that the EXS brings when the default fact feeder is used. The implementation is done by appending commands into a text file that is designed to be piped into the command line interface of the test vehicle of the MAL. This alternative feeder should only be used for testing purposes since the EXS is required for safe flight operation and the test vehicle is not designed to be used in that capacity.

4.1.11 Simulation

The position simulation implementation derives the current position by moving the UAV toward the next target position that is also output by the fact feeder. The amount of the movement is derived from the configured flight speed and the time difference between iterations. Since infinite rotation speed is assumed, the UAV heading is set to be the target heading of the fact feeder.

For the simulation of the range sensors, a number of collision checks with rays emitted by the simulated sensors are done against the set of simulated obstacles. For the narrow beam TFmini sensor, only a single ray is used, while the MB1212 simulation uses 5 beams to cover the full detection cone. If one or more collisions between a ray and a simulated obstacle are calculated, the distance to the closest one is returned as the sensor reading. Otherwise an invalid distance is used.

4.1.12 User Interface

The DearImGui [18] library is used to create the GUI. The library provides the capability of creating simple, low overhead user interfaces in C++. Due to its design, which does not require the specification of a UI layout ahead of time but rather creates UI components as required on each frame, the UI can be iterated rapidly. The capabilities to create custom rendered elements are also present. DearImgui has implementations based on most common graphics stacks, including the DirectX, OpenGL and Vulkan renderers, as well as support for GLFW and Win32 platforms. For its wide compatibility, the backend based on OpenGL 2 and GLFW is used by default. The more modern OpenGL 3 render is also implemented and can be enabled using a command line argument. OpenGL 2 is used as the default since it is compatible with some X11 forwarding use cases that do not support OpenGL 3. Further renderers are not implemented since the existing one is supported on all required platforms, but this can be done with little effort if required. DearImgui's style is fully customizable, allowing for the configuration of all layout colors and elements. Its default styling is dark themed, with a light theme available. To accommodate use cases with different environmental lighting conditions, both are implemented and can be toggled using a button. The default font used by ImGui is a monospaced pixel font. To provide a more ergonomic UI a custom proportional anti-aliased font is used.

The GUI window opened by the DearImgui library has an internal window manager, which is used to create separate windows on the canvas of the operating system window. These subwindows are used to show different aspects of the system. The subwindows can be resized, moved around and collapsed to configure the user interface layout, which is stored in a configuration file to persist between application restarts.

The Environment Map (Figure 4.2(a)) displays the surroundings of the UAV as well as the current state of the pathfinding. It is always oriented, with an upward-facing north and has grid lines every 5 m in both directions. It shows the UAV,

its heading, all detected and simulated obstacles, all waypoints and the currently selected route to the target. The window also displays the current local position and speed at the top and the number of detected obstacles and generated waypoints at the bottom. Each aspect rendered can be enabled and disabled and the zoom level of the map can be configured to accommodate the requirements set by the the current mission area's dimensions. The current UAV position is displayed as a centered, big, lime-colored circle with an extruding line indicating the current heading. Waypoints are rendered as small red circles, with an option to render their waypoint indices. Waypoints that are marked as deleted as described in subsection 4.1.5 are not displayed, but counted towards the total waypoint count. Optionally, all edges between waypoints can also be displayed, but due to limitations within DearImgui that only allow up to 2^{16} vertices per window, this option may cause glitches in the UI when there are too many edges. The waypoints of the current route as well as edges connecting them are colored orange. The path that the UAV has already been taken is drawn as a line with the color transitioning from light blue for the oldest part to green for the most recently traveled part. This coloring is done to differentiate the movement when the UAV turned around and traveled some route multiple times. When using simulated obstacles, like those described in subsection 3.1.3, they are drawn as filled green rectangles. Detected obstacles are drawn in a shade of blue, with a lighter shade reflecting a higher confidence that the obstacle exists and a darker shade conversely indicating that the detected obstacle is more likely to be a misdetection.

The Sensor data window (Figure 4.2(b)) displays all currently used sensor data. All ranging sensors are visualized in a radar-like circular view. This can be configured to rotate with the current UAV heading or stay static. It has a close range mode where detections up to 4 m can be seen and a full range mode that shows all detections up to the TFmini's maximum range of 12 m. Data that falls out of the detection valid detection ranges for the respective sensor types that is discarded in the ranging obstacle avoidance logic is also not displayed by default but can be enabled to be shown. The data of the narrow beam LiDAR sensors is displayed as a line, while detections of the wider FOV ultrasonic sensors are shown as a cone. The detections are colored depending on their range and risk for collisions. Any detections over 4 m are light green, with measurements being displayed in yellow, orange, and eventually red, respectively, for each meter closer than 4 m. The current GPS position, if present, will also be displayed.

The Simulation control window (Figure 4.2(d)) can be used to configure the UAV simulation and some mission aspects. When the system is used without a real UAV to evaluate the algorithms' performance or with an externally simulated UAV that does not have range sensors, this can be configured here. The two main sensor simulations as well as the mission target position, are configured here.

When no external global position is provided or it should not be used, the GPS simulation is available. When enabled, this will simulate a UAV that moves according to the current path and returns its GPS position. This simulated UAV can be paused and continued at any time. The flight speed of this UAV, which defaults

to 1 ms^{-1} can also be modified. The range sensor simulation, which can be used together with and without the GPS simulation and simulates sensor readings based on a number of simulated obstacles, is configured with the positions and shapes of the obstacles to simulate and the local coordinate system offset. The obstacle scenarios defined in section 3.4 can also be loaded here using buttons. For each of the simulated obstacles, their axis-aligned bounding box can be configured in size and position for both axes. Additional obstacles can also be added, and existing ones can be removed. To visualize the obstacle positions externally, they can be exported as an SVG vector graphic. When using simulated obstacles together with real position data, the origin of the local coordinate system with the simulated obstacles can also be configured there to align with the UAV's global position. Resetting all detection data of the COAS, which should always be done when modifying the simulated obstacles and configuring the target position of the UAV's mission, is done in the simulation window too.

The System window (Figure 4.2(c)) can be utilized for configuring and monitoring the system's core tasks' performance. It shows performance metrics for the GUI's FPS and the program's UPS. The FPS of the GUI is shown in two separate values. The average frames per second (aFPS) measure the time elapsed since 120 frames were rendered to calculate the value, as seen in Equation 4.7. It is used to show the UI performance for the current state but dampens the impact of outliers. The immediate FPS (iFPS) is calculated using the reciprocal of the last frame time (Equation 4.8) and can be used to monitor performance outliers.

$$aFPS = \frac{120}{t - t_{-120}} \tag{4.7}$$

$$iFPS = \frac{1}{frametime} \tag{4.8}$$

Since the GUI uses VSync, the frame rate will be limited to the refresh rate of the monitor used, typically 60. A low FPS value might indicate a heavy load from a different mission component on the system's GPU or the full utilization of the CPU leading to the render thread slowing down. To assess the COAS's core task loop of obstacle detection, graph generation, and pathfinding, the UPS metric is used. The iUPS shows the immediate value of how fast this process is. It is computed similar to the iFPS as the reciprocal of the average of the two time differences between consecutive executions. The update rate is limited to 200UPS but will only reach this value in simulations since reading the sensor data alone takes over 10 ms. A low UPS can indicate high computational cost of the pathfinding and graph generation operations. When this occurs, the risk of collisions rises since the time the UAV moves without updated sensor readings rises. The FPS and UPS metrics are generally uncoupled since they run in separate threads, but due to both of them accessing the same data via a mutex lock, blocking might occur when the main thread is writing the data the UI thread requires for rendering.

A breakdown of the performance impact of the individual steps of core tasks can be derived from the WP and PF values. The WP value displays the number of

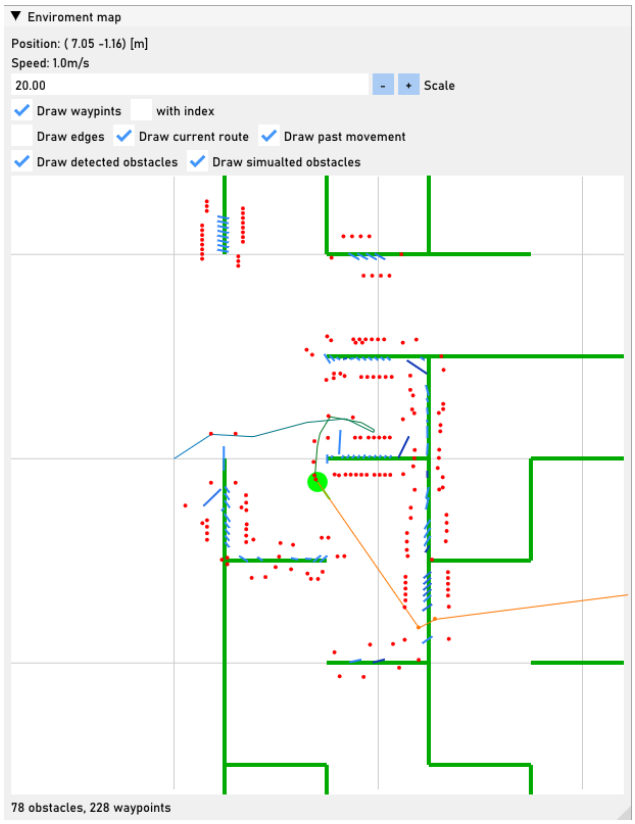
milliseconds that the last executed waypoint and edge generation (Algorithm 3) took, and the PF value shows the duration of the last execution of the pathfinding algorithm. Since waypoint generation and pathfinding are not executed in each loop iteration, these numbers persist over multiple update cycles until new obstacles are detected. The total execution time per iteration is also displayed. It includes the sum of the waypoint generation and pathfinding if it was executed in this respective cycle, the sensor reading and processing and the sending of a target position via the fact feeder. Changing the pathfinding algorithm to be used may be done in the System window. This is a computationally heavy update to the state of D* and D*Lite, which is further elaborated in section 4.1.1 and should be done with caution. The deletion of detected obstacles due to their confidence falling below the required threshold, as seen in 2 can also lead to an update cycle taking a disproportionately long time, which can exceed a second at 2000 waypoints in the navigation graph. When using a range sensor simulation, which does not have false positive detections, this can be disabled to provide better data about the performance of the tested algorithms. The toggle between GUI themes is placed at the bottom of the System window.

To control and monitor the operations of the fact feeder that sends data to the EXS, a dedicated window (Figure 4.2(e)) is used. It shows the current connection state of the fact feeder and allows to set the desired state, allowing the user to stop and start the it manually. The maximum distance of the target position sent to the fact feeder from the current position that is used to prevent the UAV from colliding with newly discovered obstacles due to its momentum, which is described in detail in section 4.1.10. It is also possible to resend the current target position if the last transmission did not get processed and the UAV has stopped moving. This window also shows the state of the mission. Initially it allows the user to start the mission using a button, then it displays that the mission is in progress, and finally it indicates that the mission has concluded successfully with the landing of the UAV.

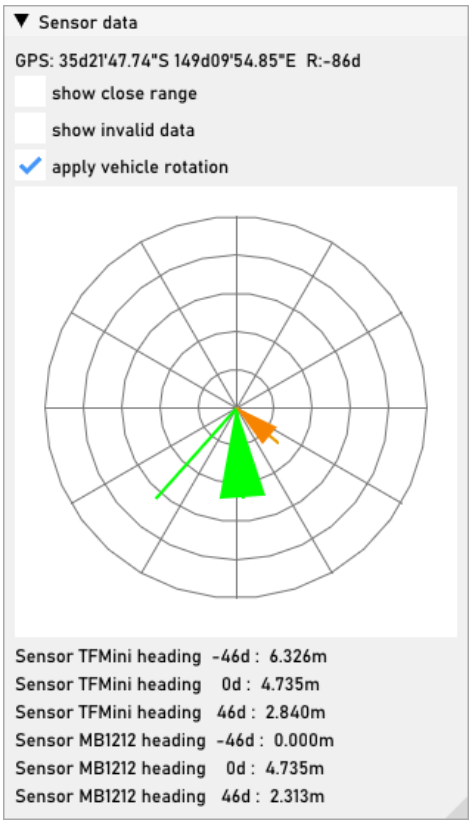
4.1.13 Command Line Arguments

For the deployment of the COAS on the UAV without a user manually taking over control, configuration and mission control must be achievable without GUI inputs. This is especially required since the deployment platform might not have a GUI enabled. This can be achieved using command line arguments. All available arguments can be seen in Table 4.4. Most have a shorthand that can be used instead of the long version.

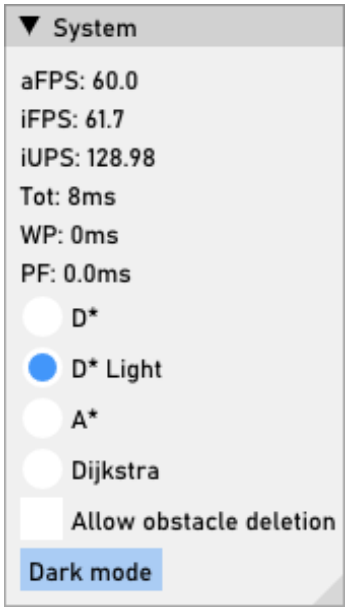
The `--nogui` argument will disable the output via the DearImgui based user interface. If this option is not specified but no desktop environment is active or the selected OpenGL version is not supported, the program will run as if it was specified. The OpenGL version, which implicitly defaults to OpenGL 2, can be explicitly set using the `--gl2` or `--gl3` command line arguments. Setting more than one OpenGL version or combining it with the `--nogui` may result in unexpected behavior and is not recommended.



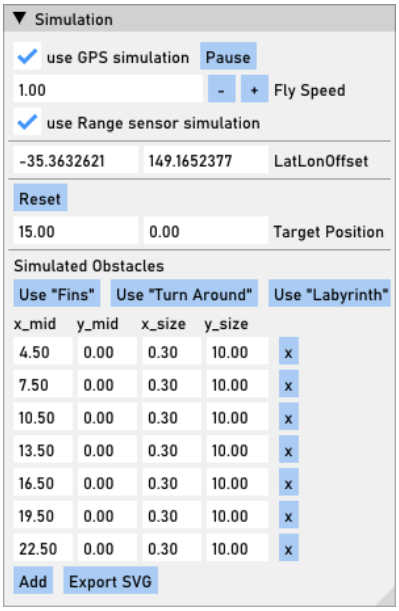
(a) Environment map



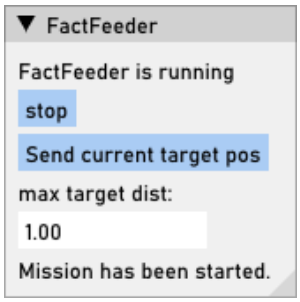
(b) Sensor data view



(c) System



(d) Simulation configuration



(e) Fact Feeder

Figure 4.2: COAS GUI windows

To start the fact feeder with the program, the `--start-fact-feeder` argument can be used. For the fact feeder to successfully start, the EXS must already be running. Otherwise the attempt to start it will fail, requiring it to be restarted manually. For automated testing or deployments when all system components should be started simultaneously but the takeoff should be delayed until everything is initialized, the `--takeoff-after` argument can be used with a number of seconds. This will cause the COAS to send the mission start command after the specified time after it itself initialized. The implementation is done using a separate thread to ensure it can proceed regardless of the current state of the main processing thread. This option should be used together with `--start-fact-feeder` since the command can not be sent without the fact feeder connected.

To load a predefined set of simulated obstacles from Figure 3.5 the `--template` argument can be used together with one of the 3 options: `turnaround`, `labyrinth` or `fins`. The empty preset is loaded by default when not specifying one. To select a different initial pathfinding algorithms other than the default D*Lite an `--algorithm` must be specified. The options available are `dijkstra`, `astar`, `dstar` and `dstarlight`. When using the COAS to steer a real UAV without a GUI, selecting the target position of the mission can not be done with the GUI. The `--target` argument should be used in this case. It takes the two floating point arguments x and y that specify the target position. When using an obstacle template, this does not have to be done since the templates always set their own target position, but it is still supported when the template is used with a differing target position. All command line arguments can be used in any combination with the limitation of the previously mentioned recommendation to only schedule a takeoff when the fact feeder is started and not specifying an OpenGL version if the GUI is not enabled.

Argument	Shorthand	Description
`--nogui`	`-X`	Start without GUI
`--gl2`	-	Use OpenGL 2 renderer
`--gl3`	-	Use OpenGL 3 renderer
`--start-fact-feeder`	`-F`	Automatically starts the fact feeder
`--takeoff-after`	`-L`	Starts takeoff timer
`--template`	`-T`	Loads an obstacle template
`--algorithm`	`-A`	Select pathfinding algorithm to use
`--target`	`-P`	Specify target position

Table 4.4: COAS command line arguments

4.2 Modifications to the MAVLink Abstraction Layer

To use the existing MAL together with the COAS and modified EXS, some minor modifications to it were required. The MAL has previously been developed with the proprietary Microsoft Visual Studio Solutions format. To build it on Linux platforms a remote connection from a Windows machine was required. To allow for

a more open development and direct compilation on the target system, the build system has been replaced with a CMake based approach. This approach is the de facto standard for C++ applications. It provides cross platform support allowing the compilation on the Linux target system and is also integrated in Visual Studio on Windows platforms. Using the native Linux GCC builds, some minor problems in the existing implementation became apparent. These did not appear when using the Microsoft C++ toolchains due to minor compiler differences. This includes mostly missing `#include` directives but also the absence from of some arguments in vehicle implementations. Notably, the beep command in the test vehicle was not functional. To allow the propagation of the GPS position from the UAV's sensor to the COAS in the test environment and potentially also in the fully assembled system in cases when the companion board does not get direct access to this data, an interface had to be created. Since the COAS receives its position data using libgps from the gpsd service, the MAL must provide the position to gpsd. While gpsd supports a number of formats and interfaces, using the ASCII based NMEA 0813 [35] format via a TCP connection is the simplest solution without any major drawbacks.

When the MAL is started with the EXS vehicle, a TCP server starts listening for inbound connections from gpsd. After a connection has been established, the MAL will forward any position data received in GLOBAL_POSITION_INT messages that periodically requests. This message contains the latitude and longitude with a resolution of 10^{-7} degrees, which is at most 11.1 mm. It also provides altitude in altitude in millimeter resolution and the UAV current heading in with centidegrees resolution.

The MAL uses IEEE 754 32-bit (single precision) floating point numbers to store the fields of received MAVLink messages. This is appropriate for most messages since most MAVLink fields are also 32-bit floats. Some fields, most notably those for global positions, use 32-bit signed integers. This allows for a higher resolution since only 23 bits of the float would be used in the significant. The highest possible longitude value of 180 degrees stored as $1.8 \cdot 10^9$ would be represented as $+(2^{23} + 5673892) * 2^7$, providing only a resolution of $128 * 10^{-7}$ degrees or 1.42 meters in the worst case. Since such a low resolution is not sufficient for precise flight, the MAL was modified to use 64-bit (double precision) floating point numbers that are capable of accurately representing signed integers up to 53 bits. There is a theoretical performance due to the doubling in number precision, but since most modern 64-bit systems only provide 64-bit FPUs, there should not be a noticeable difference.

Once a precise position has been acquired, it can be converted into NMEA messages. NMEA uses sentences to represent data. Each sentence start with a $ start marker followed by a identifier that describes the type of navigation system and sentence kind. Then a number of comma separated text fields follow before the sentence ends with a * end marker and a checksum. The sentences are separated using line breaks. For each position, two types of NMEA sentences are used RMC and GGA.

The recommended minimum sentence C (RMC) contains the minimal set of data required for positioning. This includes the longitude, latitude, altitude, the current

groundspeed and heading as well as the date and time of the positioning data and difference between magnetic and geographic north. Since the GLOBAL_POSITION_INT message does not contain a GPS time, the MAL's time is used instead. This could become an issue when the transmission between the flight controller and the MAL takes a long time but is sufficient as long as the MAL is running on the UAV's companion board. The information about the magnetic deviation is also not sent but since this is not used by any components, it is simply set to 0. In case this data would be required, it could be derived from a lookup of the world magnetic model [17] or a similar source.

The Global Positioning System Fix Data (GGA) sentence repeats core position data and also contains additional information about the quality of the position measurement. It indicates the kind of position fix, the number of satellites used, horizontal dilution of precision, geoid separation, and, optionally the age of the differential GPS and ID of a reference station. Since most of these values are not forwarded by the UAV flight controller and also not further used by the COAS, their values are set to sensible defaults. The type is a normal GPS fix with 10 satellites, the HDOP is the best possible 1.0, the geoid separation is 0 m and the remaining values are not set. Since NMEA 0813 is a text based format, the precision limitations of floating point numbers do not apply. The chosen resolution for latitude and longitude of 10^{-5} angular minutes represent at worst 18.6 mm.

4.3 Expert System

Since the existing expert systems ruleset is designed for the specific use case of microwaypoint generation and does not fulfill the requirement of the COAS, it has been heavily modified. Since waypoints, edges, pathfinding, and obstacles are handled by the COAS, all functionalities relating to it have been removed. The state of the UAV is added to the fact list by the MAL. It is contained in the CopterUpdate template template, which in turn has four copter fields, a data source, the property of the UAV, its value, and the timestamp of when the data was received from the flight controller. The properties can be found in Table 4.5. The CopterState property represents the arming of the UAV propellers. It starts and ends as Disarmed but must be Armed during any flight operation. The flight mode indicates the MAVLink flight mode described in section 2.4. For the operations of the EXS, only the Guided and Land modes are relevant, all other modes, when encountered during a mission, will always lead to an error. The LandedState indicates whether the UAV is on the ground, in the air, or transitioning between those two. The CopterAltitude informs the EXS about the current height of the UAV relative to its takeoff position, or if the UAV is on the ground, this value will be 0.

Property	possible Values
CopterState	Armed, Disarmed
FlightMode	Guided, Land and all other UAV flight modes
LandedState	OnGround, InAir, TakingOff, Landing
CopterAltitude	all floating point numbers

Table 4.5: UAV properties provided by the MAL

The EXS has multiple rules that ensure that only one fact for each of the UAV properties is present at once in the currently used fact list. These rules use a check for two distinct facts of the property, and if the rule fires, the older one gets redacted. To ensure that these rules that rely on time work, the LoopFact rule exist, which will always be asserted with the current time. To ensure there is only one such fact, there is also a rule that redacts all but the newest time.

During the mission it is important that the UAV properties stay within their expected values for the current mission state. For this purpose an ExpectedCopterState fact is issued that tracks the lists the expected values for the enumerated values CopterState, FlightMode and LandedState and the minimum valid height for CopterAltitude. For the enumerated values, multiple values may be expected at a given point in time. A set of four rules checks if the current properties are currently withing the expected boundires and if one detects a violation the system transitions into an error state an executes any error actions that are be defined in the fact list.

Once started, the EXS will create an UavState fact that controls the mission sequence. It sequentially progresses through values representing the state of the finite state machine in Figure 4.3. The fact starts with the value initializing. In that state, the EXS will check if all required facts to start the mission are set by the fact feeder. This includes an error action that is taken when an unexpected event or state occurs or a command sent to the MAL was not accepted. The mission's target position and the mission parameters "thresholds" must also be set for the mission to be considered fully loaded. Finally, a MissionLoaded fact with the value yes must exist.

When the conditions are fulfilled, the UavState is changed to MissionLoaded, and the preStart rule will be executed, setting the expected UAV state to disarmed, on the ground, and in the Guided flight mode. If one of those conditions is violated, the error action is executed and the mission is aborted. Otherwise the UavState is set to MALReady, and the threshold mission parameters are sent to the UAV via the MAL. If the thresholds are not rejected, which would trigger the error action, an arm request will be sent to the UAV, and its expected arm state is subsequently adjusted. This arm request might fail when the flight controller determines that the conditions for takeoff are not met. This could be due to not acquiring a precise position yet. After successfully arming the UAV, a request for it to set its home position is sent, and upon succes the UavState is set to SetHomeAccetpted.

With the UAV armed and ready to take off, the EXS sends the takeoff command with the configured takeoff height. The expected landed state of the UAV is modified

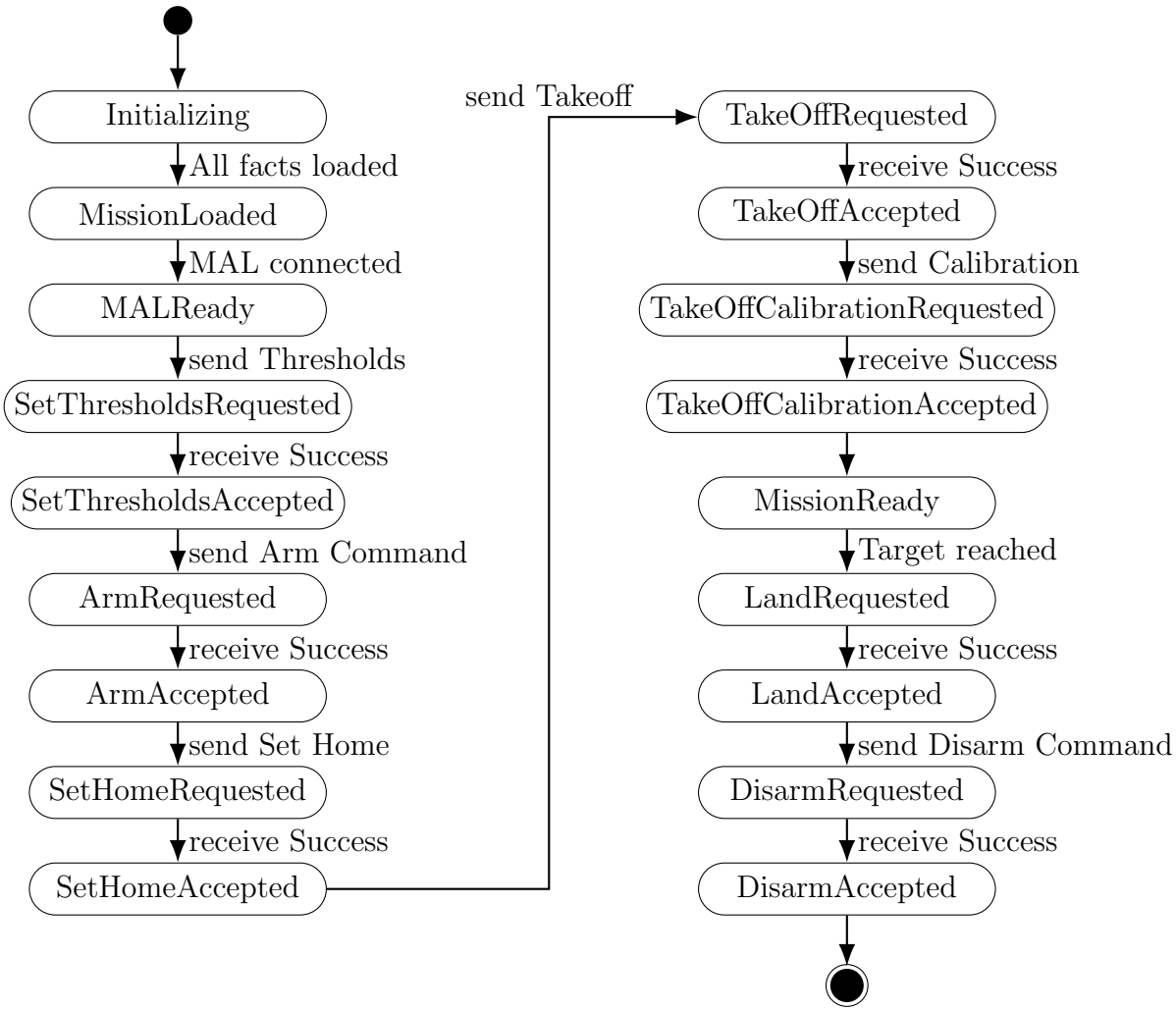


Figure 4.3: Simplified EXS state machine

to be one of OnGround, InAir and Taking off to account for the takeoff process, after its success it is restricted to be InAir. To ensure the UAV is at the desired position after takeoff, a move request relative to home with the (0,0) coordinates and the takeoff height is sent. This concludes the mission start, and the UavState is set to "MissionReady". From now on the UAV is able to act upon targetposition facts. When such facts are added from the COAS via its fact feeder, a movement request with the provided coordinates is sent to the UAV, and the fact is retracted. This is done until a targetReached fact is issued, asserted by the COAS. The EXS will then progress the UavState to LandRequested and send a message of the same name. If accepted, this causes the UAV to change flight mode to Land, which will land the UAV. hen the request was accepted and the UAV is landed, it is requested to disarm its propellers. This is done with a rule that modifies the UavState to DisarmRequested and sends a disarm request. Once the request succeeds and this

mission is completed, the UavState fact is retracted and the EXS can be terminated.

In addition to the regular values for UavState, three additional error states exists, that can be triggered from most other states when their conditions are met. The ErrorState, TimeViolatedState and AbortState are triggered when the EXS detects an error according to its current configuration. Once in such a state, the mission is aborted and can not be recovered without restarting the EXS. If the UAV's flight mode changes during flight, the UavState changes to the ErrorState unless the new flight mode is Stabilize. Since the Stabilize flight mode is used for manual flight, the system will assume that a human operator took over and consequently not send any commands to the UAV. This is done by changing the UavState to "StabilizeState".

For receiving data from the COAS fact feeder, a set of fact templates is defined that control and configure the operations of the EXS (Table 4.6).

Fact template name	Argument fields
ErrorAction	operations, maxAllowed
TimeAction	operationsAfterThreshold, threshold, addTime
ConstraintAbort	constName, applicationVal, operations
ConstSignalAbort	constName, val
Thresholds	horizontal, vertical, rotation, takeOffHeight, minimumHeight, tries
MissionLoaded	value
TargetPosition	latitude, longitude, altitude, heading
TargetReached	-

Table 4.6: Fact templates for EXS control and configuration

To ensure safe operation, the most important templates relate to error handling. ErrorAction rules are to configure the UavState "ErrorState". The maxAllowed parameter describes how many error can occur before the EXS transitions to this state and the operations list a array of strings that can will be send to the MAL when the state is reached. By default no errors are allowed, and the only error operation used is LandRequested. When the error state is reached, a message with the full current UAV state will also be logged before the mission ends. The TimeAction template configures the handling of a flight controller disconnect or MAL stall. As the ErrorAction a number of operations can be defined that will be sent to the MAL in case there is no new data for the amount of seconds specified with the addTime field. Using the threshold field, a number of such timeouts can be ignored. The default configuration for this template is a timeout of one second, triggering this error on its first occurrence and the landing and disarming of the UAV upon occurrence. When this is triggered, the UavState will also be changed to the "TimeViolatedState" error state.

If another system component, such as the COAS, wants to be able to trigger abortion of the current flight mission if it detects a fault that warrants this, the

ConstraintAbort template should be used. It to define a set of operations that occur when a matching ConstSignalAbort fact is asserted. The rules are matched when they have the same constName and a val field value identical to the applicationVal field value. When this mechanism executes, the UaVState will be set to "AbortState". The COAS defines such a template with a single land operation that it can trigger.

The Thresholds template is used to write some configuration data destined for the flight controller itself. The numerical horizontal, vertical, and rotation parameters specify the tolerances for the UAV's own detection if it has reached requested waypoints. They default to 0.1 m and 2.5° respectively. It is important that these values are lower then the once used by the fact feeder otherwise the UAV will stop moving as it categorizes the waypoint a reached while the COAS still waits for it to reach the waypoint. The takeOffHeight field configures the the height in meters which the UAV initially ascends during the start of the mission. It should be high enough to provide sufficient clearance above the ground for safe flight. Setting this value too high has the drawback that the takeoff will take longer, delaying the mission. When a higher flight altitude is required, it can be sent as a target waypoint instead. This value defaults to one meter. The minHeight field is responsible for ensuring ground clearance as well. It sets a minimum altitude value, which triggers an error when the UAV fails to reach it during the mission execution. The default minimum height is 10cm. To configure how often the EXS attempts to send a command to the MAL when it responds that the command got denied, the tries field of the Thresholds template is used. An error state will only occur if the same command gets denied the specified number of times, two by default.

To signal to the EXS that it should proceed with starting the mission, the MissionLoaded fact is used with the value field set to yes. If all other necessary facts are also present, the mission will start according to Figure 4.3. During the mission, the control over the UAVs target position is done using the accordingly named TargetPosition rule template. These facts can also be specified before the takeoff is completed, causing the UAV to immediately fly towards the specified location once it is in the air. It takes parameters for the target position and heading. The UAV will rotate to orient itself as requested while following the direct route to the specified location. This has the positive side effect of giving the sensors mounted on it the opportunity to record distance data in a greater range of rotations. When turning over 90°, however, the UAV could potentially fly into an area that might contain obstacles before having a forward facing sensor. This failure mode has not been observed during the extensive testing of the system, and should it occur, can be remedied by always turning the UAV when it reaches a waypoint until at least one sensor faces forward. The final fact received from COAS in a successful mission is a parameterless TargetReached fact. It indicates that the COAS has completed its mission and gives control to the EXS to land and disarm the UAV.

5 Results

To evaluate the performance of the implemented algorithms as well as the system as a whole, the implemented components are tested individually and integrated into the complete system. The testing and benchmarking of the COAS component that performs the most important task takes precedence. It is evaluated using both the internal simulation and the full system simulation. The functionality of the distance sensor drivers has been verified by connecting them to the GPIO pins of a Jetson Nano and verifying that the values read match the expected distances.

5.1 Internal simulation

The internal simulations verified the core capabilities. Each test scenario laid out in 3.5 is tested with all four implemented pathfinding algorithms: Dijkstra, A* D* and D*Lite. All tests using the "Empty" test scenario were complected successfully with the UAV moving the minimum required distance of 15.8m at the configured speed of 1ms^{-1} in 15.8s. The other scenarios are used to also provide quantitative results. The recorded data can be seen in the Tables 5.1 to 5.3. For each scenario each algorithm has been tested, and the average and maximum computation time for the pathfinding calculations and generation of the navigation graph have been recorded. For the relatively short "Fins" and "Turn Around" scenarios, a default flight speed of 1 ms^{-1} is used, and the longer "Labyrinth" scenario is done with 2 ms^{-1} to accelerate the process. For each test, the duration it took the UAV from start to finish is also shown. Using the flight speed that the UAV uses for the full simulation, the distance traveled can also be deduced.

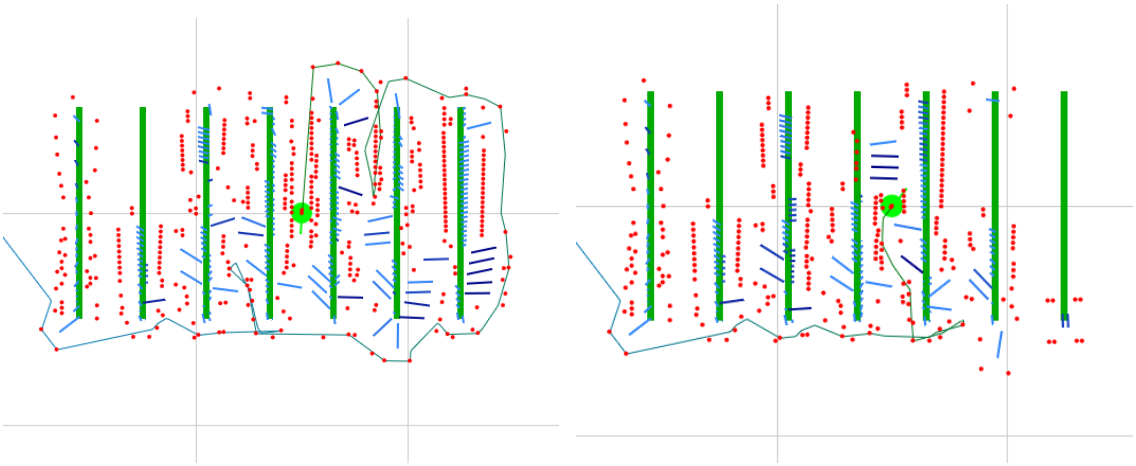
Since only the pathfinding algorithm differs, the graph generation time is expected to be similar. The number of detected obstacles and, closely related to it, generated waypoints should also not be significantly different between the algorithms used. All the algorithms are expected to return an optimal path of the same length, and in complex graphs such as the navigation graphs used, the probability of having more than one such path is exceedingly low. Nonetheless, the paths used differ significantly, as indicated by the differential in travel time. This can be observed to the highest degree in the "Fins" obstacle layout and to some degree in the "Labyrinth", while the fluctuations seen for the "Turn around" scenario are within the expected variance. The duration in these tests differs only by 0.34s at a standard deviation of 0.14s or 0.38% of the average. In the "Turn around" tests, the graph generation metrics and obstacle and waypoint counts are also not significantly different between multiple algorithms used. This allows for a more confident assessment of the data

recorded. But due to the very short computation times in the range of 100 µs all calculations fall in the category of having a negligibly short computation that is more than sufficient for the purpose of running pathfinding up to 200 times per second. Taking the conclusions from this data to more complex scenarios should also not be done without sufficient analysis since the time complexity of the different algorithms scale differently. Therefore, the conclusions of this scenario are that all algorithms implemented perform very highly, with the Fibonacci heap optimized Dijkstra being the fastest. What is also noteworthy is that the average time used to generate the navigation graph is eight to twelve times higher than the times used for pathfinding.

Algorithm	Pathfinding		Graph generation		Duration	Obstacles	Waypoints
	Max ms	Avg ms	Max ms	Avg ms	s	#	#
Dijkstra	0.05	0.03	0.93	0.33	37.82	144	315
A*	0.11	0.04	0.96	0.36	38.05	146	320
D*	0.11	0.03	0.96	0.35	38.09	145	318
D*Lite	0.12	0.05	0.96	0.35	38.16	146	316

Table 5.1: "Turn around" algorithm comparison

When examining the COAS as it simulates the "Fins" scenario, the cause of the high variance between the algorithms used gets revealed. As seen in Figure 5.1, the path taken to target can vary quite significantly. The reason that the Dijkstra



(a) Dijkstra (b) A*

Figure 5.1: Route taken in the "Fins" obstacle layout

run did not utilize the shorter path is that it detected some obstacles while the A* did not. This detection that does not always occur due to slight differences

in the approach to the position of the obstacle. The detected obstacle has been identified by the outer edge of the detection cone of an MB1212 sensor. The system then assumes the placement of the obstacle in the center of the detection area since it does not have information about the obstacle's real position and has to ensure that collisions are avoided. The initial trigger to that leads to the divergence is slight differences in the timing when the sensor is simulated and the UAV position is updated, which propagate into further differences increasing in magnitude until they are sufficiently large at a fork that they lead to a completely different route being planned. In further repeated executions, this test also reveals the possibility of the system not reaching the target at all when obstacles are detected in both routes accessing the target from the north and the south. While the probability for this failure to complete the mission depends on the simulation movement speed and can vary depending on minor changes in the template, further tests showed that it occurs in at least one in five runs.

Algorithm	Pathfinding		Graph generation		Duration	Obstacles	Waypoints
	Max ms	Avg ms	Max ms	Avg ms	s	#	#
Dijkstra	0.11	0.06	4.97	0.86	77.51	260	575
A*	0.17	0.03	1.92	0.54	32.47	165	386
D*	3.62	0.26	21.06	3.28	69.42	241	535
D*Lite	0.66	0.25	13.43	3.13	60.61	224	470

Table 5.2: "Fins" algorithm comparison

Due to the high variance displayed in the workload performed by the algorithms, the quantitative results of the "Fins" test can only provide limited reliable conclusions about the performance. The previously discovered fact that the pathfinding takes only a fraction of the time of the graph generation is reinforced nonetheless.

This test scenario has revealed a number of issues with the implementation that can cause problems. The first is the possibility of essentially random route selection based on the variance in sensor input and processing times. This in itself is not aa issue as long as all possible outcomes reach the target without collisions using a shortest route based on their relative knowledge of the environment. The more severe issue that, when combined with the first one, can lead to divergence in performance as seen in this test scenario is the creation of obstacles based on a ranging measurement at the edge of the MB1212's detection cone. These obstacles must be created in the way they are to ensure collisions are avoided but can lead to the blockade of routes that are safe to take based on the ground truth obstacle situation. Such obstacles may be removed if their confidence drops below the configured threshold if the sensors used can not detect it anymore, but this is unlikely to occur with the pathfinding as it is implemented. When an obstacle is detected, the pathfinding will create an alternative route around the obstacle, causing the UAV

to turn away from the suspected obstacle. The largely forward facing sensors will therefore not point in the direction of the obstacle anymore, making it impossible to invalidate its perceived existence. If all possible paths to the target are blocked, there is also no mechanism for the UAV to validate the existing obstacles for their validity. Possible improvements that could avoid, remedy, or eliminate these issues will be explored in section 6.1.

The large "Labyrinth" also suffers from the problem of the used route differing between runs, with the longest route taken being 86% longer than the shortest one. But in this case, this is in part expected since to the optimal path in a maze is designed to be nontrivial. To achieve more comparable results, each algorithm is tested in three separate runs, as seen in Table 5.3.

Algorithm	Pathfinding		Graph generation		Duration	Obstacles	Waypoints
	Max ms	Avg ms	Max ms	Avg ms	s	#	#
Dijkstra 1	3.06	0.76	152.75	30.16	285.81	1632	3785
Dijkstra 2	2.35	0.69	86.00	22.73	165.49	1245	3025
Dijkstra 3	2.50	0.68	112.76	27.00	246.72	1590	3638
A* 1	15.35	1.33	137.10	29.19	229.95	1533	3693
A* 2	5.85	0.83	100.13	24.60	196.34	1376	3243
A* 3	6.47	0.94	106.67	23.26	155.00	1262	3117
D* 1	3.69	0.68	101.00	24.45	153.25	1229	3082
D* 2	2.09	0.59	113.68	23.38	157.30	1221	2967
D* 3	6.93	0.66	99.87	23.86	186.51	1339	3191
D*Lite 1	2.31	0.36	111.29	22.09	183.85	1376	3231
D*Lite 2	9.43	0.36	90.57	23.51	191.29	1379	3227
D*Lite 3	4.53	0.38	128.81	24.88	155.70	1234	3077

Table 5.3: "Labyrinth" algorithm comparison over 3 independent runs

The maximum pathfinding time for all algorithms is volatile. The most reliable algorithm in that regard is Dijkstra, with its maximum computation time only fluctuating by 0.56ms and also being the lowest on average. In terms of average computation time, D*Lite is the clear best choice with a very stable value of 0.37ms but the algorithm has a high maximum computation time of 5.42ms, which is surpassed only by A*, which, even when excluding its first run that generated the most nodes out of all runs, has an average of 6.47ms maximum computation time. A* also has the worst average of the average time, with over one millisecond or 0.89ms when the first run is excluded. D*, despite theoretically being a worse implementation of its concept than D*Lite, produces the second best result in both average and maximum computation time of 0.63ms and 4.25ms, respectively. Since the maximum execution times of all the algorithms except Dijkstra are very unstable, this number can not be considered a significant result, especially considering that the D* executions all produced a low number of waypoints. From this data, it can be concluded

that this implementation of A* has no viable use case. It offers neither stability nor average high performance. As explained in section 4.1.7, the A* implementation does not utilize an efficient open list implementation since this slowed down its operation compared to the simpler approach. Numerous publications have shown that A* performs better in their test cases [15][23][27], which raises the question why it can not exploit its advantages here. The Labyrinth test scenario might not be ideal for A* since it uses the assumption that nodes closer to the target position tend to have a shorter path to the target, which is only not always true in these kinds of scenarios where backtracking is commonly required. The implementation of the heuristic might also play a role since the distance calculation has to solve a comparatively expensive square root.

To further explore the performance, Figure 5.2 compares the computation times of the Dijkstra, A*, and D*Lite algorithms in the "Labyrinth" obstacle scenario. To show an estimation of the computational effort required, the total number of waypoints generated is also shown. The three executions have been manually selected to terminate with a similar number of approximately 22000 of iterations. No D* execution is displayed since D* produced results very similar in shape to D*Lite but with a higher average. As discussed previously, not all iterations detect new obstacles, and therefore the pathfinding is only done in about 1200 iterations for each of the algorithms. Especially in sections where the UAV turns around in a dead end and subsequently moves through an area where all obstacles are already detected a high number of iterations without pathfinding can occur. This is visible around iteration 5000 and around iteration 10000. For the purpose of better readability, these sections are connected in the graph.

The plot reinforces the consistency with which Dijkstra solves the pathfinding problem. The lower number of generated waypoints during the beginning of the mission and the close distance to the target close to mission completion make it perform faster while otherwise taking around one millisecond. The A* algorithm performs better than Dijkstra in most iterations but has sections of very high computation cost. This could be related to the problem that the direction that A* favors in expanding nodes first does not lead to a path to the target but rather a dead end, wasting the computation time. A* design causes it to fully explore such sections of the graph when they are closer to the the target position then alternative possible routes. Since Dijkstra expands nodes solely on the path length from its start position, it does not face this issue. D*Lite has a very low computation time for the vast majority of the iterations, with an average computation time of 0.37ms, outperforming all other algorithms. In only 7.2% of iterations D*'s computation time exceeds the one of Dijkstra. But when doing so, its computation time can more than an order of magnitude higher than its average. These spikes in computation time that are likely occurring when D*Lite has to recalculate significant portions of its internal navigation state. This can be caused by a path segment used to calculate the optimal distance for a lot of notes intersecting with a newly discovered obstacle.

This more in depth exploration reinforces the conclusions found from the average and maximum computation times alone. Dijkstra repeatedly shows itself to be

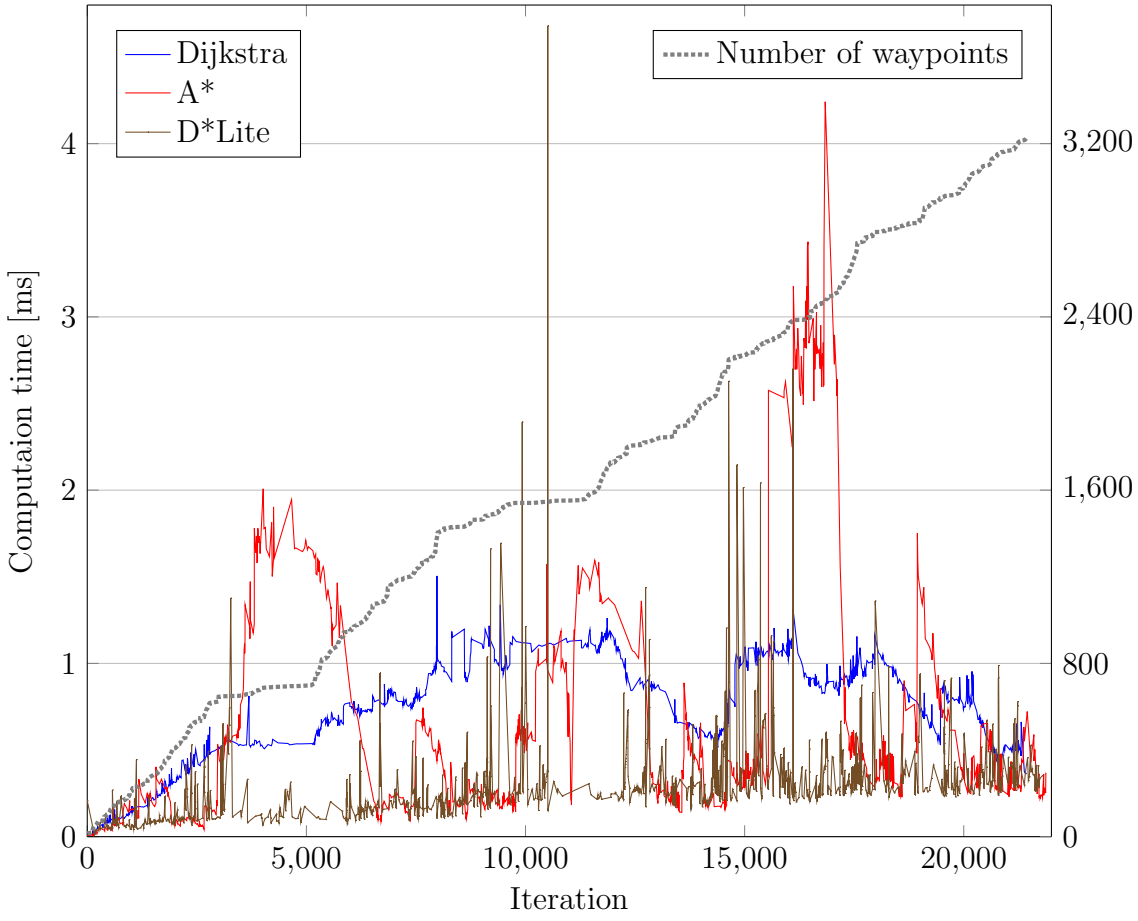


Figure 5.2: "Labyrinth" pathfinding time depending on iteration

the most reliable algorithm. The long periods of high computation time observed for A* back up the previously stated conclusion that it is not suitable for this kind of problem. D* Lite performs the best on average but has spikes of high computation. The here not reexamined D* while performing second to only D*Lite in average performance has the same issue with high computation time spikes and should therefore not get used over D*Lite when average performance matters while still having spikes too high to be considered as a consistent alternative.

Compared to the difference in pathfinding algorithms, the graph generation times in this scenario are again extremely high. The average pathfinding time over all twelve runs of 24.9s exceeds D*Lite's average by a factor of 67. With the maximum typically reaching around 100ms, which is high enough to seriously impact the system's operation. The resulting system update rate of under 10 per second has the potential to impact the detection of obstacles. With the travel speeds used this can not lead to frontal collisions but small, difficult to detect obstacles after a turn in the path traveled could become an issue.

The number of new waypoints each iteration varies. The theoretical maximum of 24, when four waypoints are generated for each of the six sensors, which can all detect separate obstacle, is never reached. The practical maximum is 12 when a new obstacle is detected in each sensor direction. Most frequently, two or three waypoints are generated. They represent 28.1% and 31.1% of the iterations with newly discovered obstacles. Comparatively, only 0.58% of executions generate eight or more waypoints. The graph generation performs the same edge creation algorithm for each new waypoint (compare Algorithm 4). If the system's update rate decreases, the distance traveled between each iteration increases causing the average number of obstacles detected to also grow. Since a higher number of obstacles further decreases the update rate, this can cause a feedback loop that worsens system performance. This runaway effect gets stopped once each iteration generates the practical maximum of 12 waypoints. The Labyrinth scenario is not designed as a scenario that models any expected occurrence in a real mission, therefore diminishing these concerns. To extrapolate the performance of the graph generation, Figure 5.3 can serve as a basis. The graph generation time per newly added waypoint is plotted depending on the already existing waypoints in the navigation graph.

The trend of a higher computation time per new waypoint with more existing waypoints is clearly visible. It is caused by the process of attempting to create an edge between all new and all old waypoints. On average the computation time per existing waypoint is 2.52 µs as indicated by the red affine regression line. The lower bound is approximately 1 µs and the upper one reaches 7.5 µs at 3000 existing waypoints. The strong divergence from the regression can be explained by the fact that an edge that collides with the first obstacle it is tested against has a significantly lower computation cost attached as opposed to one that does not collide with any obstacles but has to be checked against all of them. Some areas of the navigation map are laid out in a way where most new edges collide with more recently created obstacles, causing less computation time to be required. If the average of three waypoints created is applied to the upper bound, one can expect at most 22.5 ms at

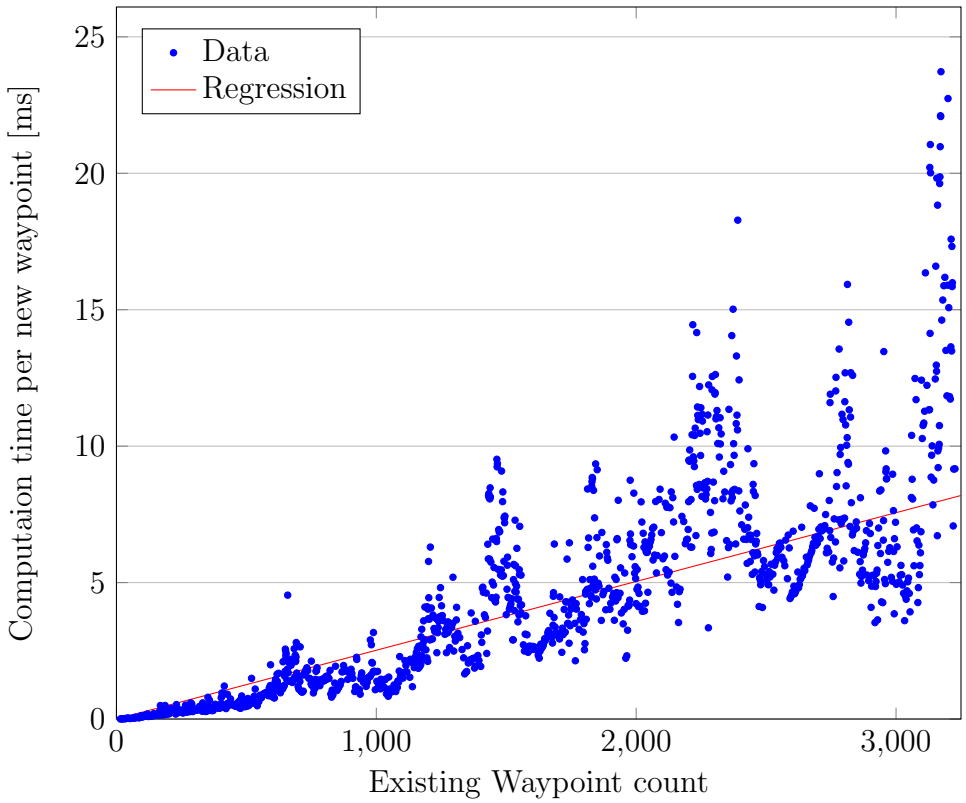


Figure 5.3: Graph generation time depending on graph size

1000 waypoints. This value, as well as the 24.9ms average for all Labyrinth graph generation operations, still represents a value that, when taking a 10ms sensor read time and 5ms for other computations, results in an acceptable 25Hz system update rate.

In conclusion, the internal simulations have shown that the system can perform its core operation, and the performance of the pathfinding algorithms is acceptable. But it also showed a number of issue including difficulties of fitting through relatively tight passages and performance problems in the graph and waypoint generation. There is no clearly best pathfinding algorithm, but the A* and D* implementations are strictly worse than the available alternatives. Which algorithm should be used depends on the mission layout and trade-off between average and worst case performance. The potential impacts of high computation time spikes could have will become be explored in the full system simulation.

5.2 Integrated System Simulation

As designated in the concept chapter, the fully integrated system simulation aims to evaluate the system as closely as possible without needing the system's hardware components.

For the full system simulation the five required software components—Ardupilot Copter Simulator, MAL, EXS, COAS and gpsd—must be started and run at the same time. The order of some of the components also matters. The MAL component that communicates with the UAV simulation via UDP can only be started once the latter has opened its communication channel. This can take up to 10 seconds. The UAV simulation also has to be manually set to the guided flight mode. The COAS component requires gpds to be running before it can start. Otherwise it will not attempt to connect to it. Gpsd can run without having received position data yet, in which case COAS will connect to it and proceed with its initialization once position data is available. The UAV simulation takes up to 30 seconds after being initialized before allowing takeoffs. The COAS is responsible for ensuring this delay is not violated. If the takeoff command is sent too early, the UAV will deny the takeoff attempt, and the EXS will transition into an error state.

For starting all the required programs and configuring them to work together in a reliable and reproducible fashion, a bash script is used. This script utilizes the powerful tmux tool that multiplexes multiple terminals into one terminal window. It also comes with strong automation support. The use of such a tmux script reduces the effort of starting the system simulation by a considerable margin and provides easier workflows when operating in an environment where opening more terminal windows is not trivial, such as a remote SSH connection. Figure 5.4 shows a terminal running the described simulation.

The tmux script navigates into the appropriate directories, waits as required, and executes the commands to start each of the components, like shown in Table 5.4 for the D*Lite algorithm test.

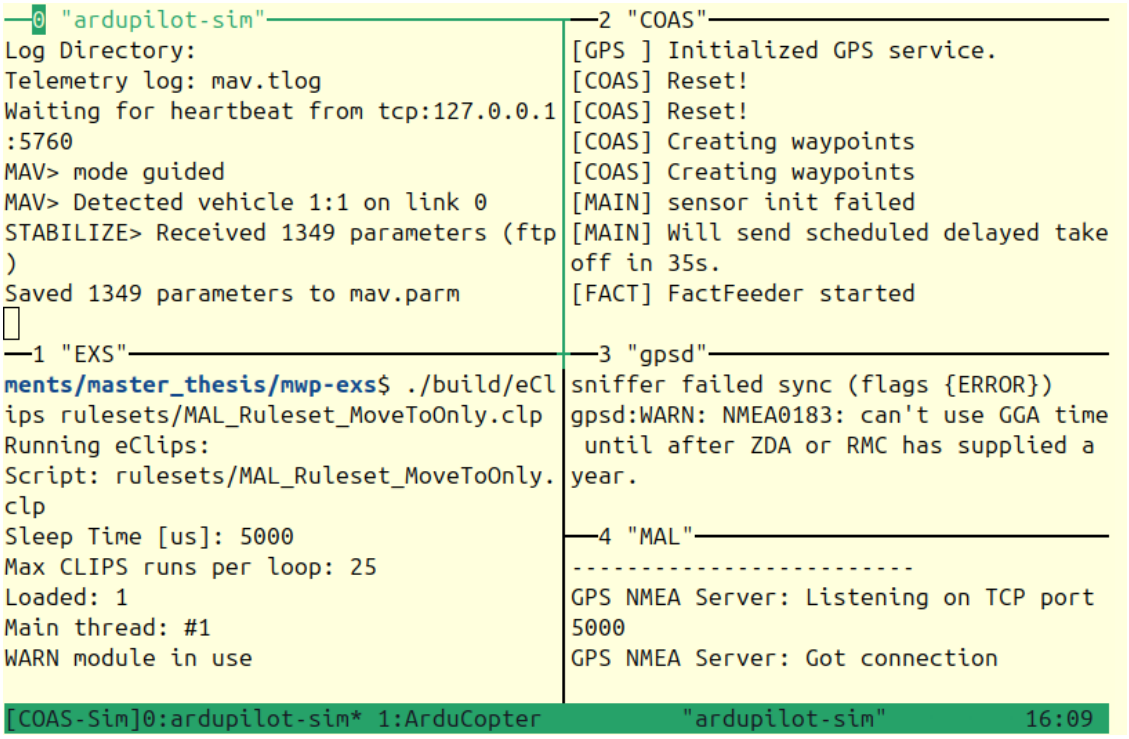


Figure 5.4: System simulation utilizing `tmux`

Component	Command line
COAS	`coas -T labyrinth -A dstarlight -F -L 35`
gpsd	`gpsd -D -N -n tcp://localhost:5000`
UAV simulation	`sim_vehicle.py -v copter --console --map`
MAL	`MAL -m udp -p 14550 -v exs -l info`
EXS	`eClips rulesets/MAL_Ruleset_MoveToOnly.clp`

Table 5.4: Command lines to start the full system simulation in the "Labyrinth" scenario for the D*Lite algorithm

The arguments used for COAS set up the test scenario, select the pathfinding algorithm to be used, start the fact feeder, and schedule a takeoff 35 seconds after initialization. The gpsd instance started is explicitly configured to run in the foreground and not in daemon mode so it can be stopped with all the other components, and it can log debug messages to the console. It is configured to connect to the MAL to receive NMEA position data via an open TCP port. The Ardupilot UAV simulation is started by specifying the copter vehicle type and that a map and console (Figure 3.4) should be opened. The MAL connects to the simulated flight controller using a UDP MAVLink connection, starts an expert system vehicle, and is instructed to log all events of the level info or higher. Finally, the expert system is started using the ruleset designed for communication with the COAS. After all components have been initialized, tmux sends `mode guided` to the simulated flight

controller to change the flight mode from Stabilize to Guided. When the 35s takeoff timer has expired, the COAS instructs the EXS to start the mission. After takeoff the performance of the COAS can be evaluated similarly as done in section 5.1. Once this is completed, the simulated UAV lands, and the simulation components can be stopped or reset to test another scenario. As with the internal test, the "Empty" obstacle template is tested first. Since all pathfinding algorithms have already shown their ability to solve this obstacle free scenario, they were not further evaluated. The focus has rather been on the other components and their communication. The UAV simulation, when directed by the COAS, EXS, and MAL, or alternatively manually by using the test vehicle of the MAL, showed that the UAV was capable of taking off to 10m in about 7s, flying precisely to a specified position, and landing safely. In this mission the UAV achieved a flight speed of $0.91 \mathrm{ms}^{-1}$ when configure with a maximum target distance of 1m. This test verified that the COAS can receive position data from gpsd and send facts to the EXS. The EXS itself has shown its own communication with the MAL and the functionalities of the adjusted ruleset. In addition to the successful EXS mission execution, a number of error states have also been verified. These include the failure to arm, take off, land, and disarm, as well as manual operator takeover. The MAL that was only slightly modified was capable of communicating with a flight controller and executing MAVLink commands. Its newly added function to send position data using the NMEA format to gpsd could be shown to be working. To ensure the system still works after delays in the mission execution, a test with the system sitting idle for 4 hours while connected before takeoff has been successfully done.

The "Turn around" scenario was tested, but the data did not show any results differing from the internal system test. All algorithms had an average pathfinding and execution time of under 50 µs and a maximum below 150 µs. Dijkstra performed best, followed by A* and D*Lite, with D* being the worst on average by 10 µs. Due to the low total time the comparison of the results can not give any significant conclusions.

The "Fins" test scenario, which has been shown to be problematic in section 5.1 has not been used with the full system simulation.

The most advanced test scenario is the use of the most complex "Labyrinth" scenario with the fully integrated system simulation. All four pathfinding algorithms have been reevaluated in this scenario. The results can be seen in Table 5.5. For all algorithms, the measured average flight speed was between 0.85 ms^{-1} and 0.86 ms^{-1}. The flight speed can reach up to 1.07 ms^{-1} on long, straight sections of the path but slows down near turns, leading to the lower average.

Algorithm	Pathfinding		Graph generation		Duration	Obstacles	Waypoints
	Max ms	Avg ms	Max ms	Avg ms	s	#	#
Dijkstra	1.46	0.55	148.60	18.01	381.53	1395	3484
A*	5.99	0.84	98.67	17.44	388.22	1390	3455
D*	19.05	0.66	108.36	16.86	411.79	1408	3408
D*Lite	10.20	0.34	96.34	16.99	483.16	1486	3550

Table 5.5: Labyrinth algorithm comparison in the full system simulation

Notable is that despite a substantial flight duration variation and consequently distance difference, the number of generated obstacles and waypoints stayed within 5% of the average of 1420 obstacles and 3474 waypoints. This suggests that all algorithms explored the same locations in the labyrinth but coincidentally turned around at different points in time. Due tu the limited turn rate of the UAV in this scenario, a higher portion of iterations is expected to detect new obstacles and therefore trigger graph generation and pathfinding, while the average number of new waypoints in these iterations is expected to be slightly lower. The lower flight speed of the simulated UAV in this simulation, especially when near waypoints of the current route, has the same expected consequences. These differences compared to the internal simulation are expected to favor the iterative algorithms because they scale with the number of changes, while the stateless ones only scale with total graph size, which is similar between the simulations.

The Dijkstra algorithm is once again shown to be the most consistent and even improves its performance compared to the internal simulation.

A* once again has the worst average performance, while also having worse maximum performance than Dijkstra, reinforcing that it is strictly worse than Dijkstra for the proposed obstacle templates.

Despite the theoretical advantages of the iterative algorithms in this scenario, their computation time does not improve compared to Dijkstra. D*, which had a better average performance than Dijkstra in the internal simulation gets outperformed by it in this simulation. The overall best average performance is still delivered by D*Lite but it improved less than Dijkstra. The worst iterations of the iterative algorithms are still their greatest issue. For D*Lite the maximum computation was 30 times longer than the average one.

The high computation time spikes have not been shown to result in problems for the system, but another problem that could cause them to become worse has been discovered. Observing the flight path of the UAV one rarely occurring problem is that when the UAV turns by over 90 degree while next to a wall it get too close to the obstacles detected. This occurs due to the momentum the UAV has when turning. The consequence of this is that the node representing the UAV's current position does not have any edges causing all pathfinding algorithms to fail finding any route. This is especially detrimental for D*Lite since it performs pathfinding beginning at the target node. The algorithm has to expand all nodes before it can

conclude that no path exists. Such an event can not cause the UAV to collide with any obstacles for the low flight speeds of up to 2 ms^{-1} used, but is a reason why higher speeds are not safe to use. If this occurs, the UAV will continue on its current trajectory to the next node, at which point the pathfinding algorithms can again find an optimal path and the mission can be continued.

The average graph generation time in this simulation is lower than the ones in the internal simulation. The main factor influencing this improvement is the previously mentioned tendency of having more graph generation iterations but fewer average new waypoints per iteration. The average graph generation time of 17.3 ms is 30.4% faster but the number of executions over on average three times longer full system simulation is also 93% higher. To improve the average performance of the system, slowing down the UAV is always possible and also improves safety, but this reduces effective mission range, increases battery consumption, and does not solve fundamental system drawbacks. The maximum graph generation time fluctuates but is in the same range as the values measured in the internal test. This can be explained by the fact that in both tests an iteration with a high number of new waypoints can occur while the number of already existing waypoints is near the maximum.

6 Discussion

The implementation has been shown to work in the simulation, fulfilling all primary objectives. As the simulations have shown, the UAV can reach its target without causing a collision and in a timely manner. The components work together without communication issues. The tested algorithms all can be used for this completed system but there they differ in performance metrics leading to different optimal choices depending on the uses case. For the best worst case performance, the oldest used algorithm, Dijkstra, performs the best, with all other implementations having spikes of high computation time in certain circumstances. The A* algorithm is outclassed by Dijkstra in all evaluated criteria in the simulations done. This result which contradicts other research results can be explored further in the future work. The iterative D* and D* Lite have low average computation costs but can have spikes far above their average value. They can exploit the advantages that their design provides but suffer when the graph changes significantly. The D* Lite algorithm beats D* in all significant tests. When selecting the algorithm to be used, the decision should be made between the stable but, on average, slower Dijkstra and the faster, and therefore more energy efficient, but varying D*Lite. The navigation graph generation has worked as intended, providing the input for the pathfinding algorithms, but has been shown to have the worst computation time of all system components. The focus on optimized pathfinding algorithms has caused it to receive less optimization work. The EXS has been proven to be a reliable mission control component, and in combination with the MAL it is capable of executing UAV missions. The use of multiple levels of simulation has been shown to be successful. The first, less time consuming simulation could identify a simulation scenario that could not provide reliable data. It already showed the indications for the result of the full simulation. Some minor problems with the system were also already identified in the first simulation, allowing for the processes in the second one to be adapted to contract them. The full test has shown the limitations related to the maximum flight speed and build a solid basis for tests with a real UAV.

6.1 Future Improvements

The biggest problem in evaluating the simulation results is the random variance in the path taken. One approach to circumvent this would be to run all pathfinding algorithms at the same time, collecting results on the exact same input data. This has the drawback of a test scenario where the whole computation takes longer than it would when only one algorithm is selected. An alternative would be to test using

fixed travel distances rather than real time. The use of test scenarios that have only one obvious optional path could remedy this problem. To further explore the shortcomings of A*, an obstacle scenario that is designed to utilize its strength could be designed. A mostly open environment with sparsely distributed obstacles and few dead ends could cause A* to show better results than Dijkstra.

To improve the performance of the graph generation, a number of approaches exist that still utilize a similar navigation graph. To limit the number of required collision checks, the length of edges could be restricted based on the local node density, or the number of maximum edges per node could be limited. These solutions can lead to an edge that would be part of an optimal path not being generated and should be used carefully. The partition of the navigation graph into multiple connected subgraph sections could optimize the collision checks by a large factor since the number of checked collisions could be limited when both ends are in the same or neighboring subgraphs.

The obstacle generation based on ranging measurements, which has been shown to sometimes create obstacles that block valid paths, could be designed in a way that these obstacles are split into multiple smaller ones that then in turn have an independent confidence level, causing some of them to be deleted individually. This would solve the problem with blocked paths that was occurring in the "Fins" obstacle template.

7 Conclusion

This thesis has shown that the chosen approach for implementing a path planning system using long-proven technologies such as expert systems, navigation graphs, and pathfinding algorithms can deliver highly performant, reliable, real time solutions that incorporate automation on a HOTL level. The modular system design and simulation backed development have produced a solution for autonomous UAV flight that can work using low computational requirements. The selected low cost sensors concept provides reliable obstacle detection. The integration with existing components in the MAL and EXS has also shown the flexibility and of such a approach. The continued use and improvement of existing software allows future applications to save development time. The incorporation of other, new and the improvement of implemented pathfinding algorithms allows to widen the scoped of the comparison work done. While D*Lite and Dijkstra have been shown to be the best for the tested scenarios other test scenarios, might lead to differing results. The usage of open standards for communicating with UAV flight controllers and for position data distribution makes this approach applicable for many types of missions while also allowing the exchange of individual components. The development process integrating simulations on multiple different levels shows not only makes gathering resulting data easier but also improves the reliability of the system.

Possible continued research steps included testing of the system in with data recorded during UAV and deployment in flight operation. Future development may include the design of new mission scenarios, incorporation of multiple UAVs, communication with other hardware, and integration with the existing hangar cloud concept developed at the TU Chemnitz [29].

Bibliography

[1] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985 pp. 1–20 (1985)

[2] Clips reference manual volume I. https://www.clipsrules.net/documentation/v642/bpg642.pdf (2025), [Accessed 29-06-2025]

[3] GPSd — Put your GPS on the net! — gpsd.gitlab.io. https://gpsd.gitlab.io/gpsd/ (2025), [Accessed 13-06-2025]

[4] Open Source Autopilot for Drones - PX4 Autopilot. https://px4.io/ (2025), [Accessed 30-06-2025]

[5] ArduPilot: ArduPilot. https://ardupilot.org/ (2025), [Accessed 30-06-2025]

[6] Azzabi, A., Nouri, K.: Path planning for autonomous mobile robot using the potential field method. In: 2017 International Conference on Advanced Systems and Electric Technologies (IC_ASET). pp. 389–394 (2017)

[7] Bagherian, M., Alos, A.: 3d uav trajectory planning using evolutionary algorithms: A comparison study. The Aeronautical Journal 119(1220), 1271–1285 (2015)

[8] Battseren, B., Harradi, R., Kilic, F., Hardt, W.: Automated Power Line Inspection. Technische Universität Chemnitz, Fakultät für Informatik (2020)

[9] Beesten, J., Braßel, H., Breuß, M., Fricke, H., Hardt, W., Heller, A., Kern, E., Khan Mohammadi, M., Lindner, M., Pfister, E., Schneidereit, T., Stuchtey, T., Yarahmadi, A.M., Zeh, T., Zell, S., Zügel, T.: RescueFly – Einsatz von dezentral stationierten Drohnen (Unmanned Aircraft Systems, UAS) zur Unterstützung bei der Wasserrettung in schwer zugänglichen und weitflächigen Gebieten. https://fis.tu-dresden.de/portal/files/55809353/BIGS-Studie-11-2024-WEB-final.pdf (Mar 2024)

[10] Benewake: Product manual of TFmini Plus. https://www.mouser.de/pdfDocs/TFminiPlusProductManual.pdf (2024), [Accessed 20-06-2025]

[11] Benewake: TFmini Plus LiDAR. https://en.benewake.com/DataDownload/index.aspx?pid=20&lcid=23 (2024), [Accessed 19-06-2025]

[12] Brinkman, M.P.: Applying uav systems in wildlife management. In: Proceedings of the Vertebrate Pest Conference. vol. 29 (2020)

[13] Burgos, E., Bhandari, S.: Potential flow field navigation with virtual force field for uas collision avoidance. In: 2016 International Conference on Unmanned Aircraft Systems (ICUAS). pp. 505–513 (2016)

[14] Çalışır, D., Ekici, S., Midilli, A., Karakoc, T.H.: Benchmarking environmental impacts of power groups used in a designed uav: Hybrid hydrogen fuel cell system versus lithium-polymer battery drive system. Energy 262, 125543 (2023)

[15] Candra, A., Budiman, M.A., Hartanto, K.: Dijkstra's and a-star in finding the shortest path: A tutorial. In: 2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA). pp. 28–32. IEEE (2020)

[16] Chakravarthy, A., Ghose, D.: Obstacle avoidance in a dynamic environment: a collision cone approach. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 28(5), 562–574 (1998)

[17] Chulliat, A., Macmillan, S., Alken, P., Beggan, C., Nair, M., Hamilton, B., Woods, A., Ridley, V., Maus, S., Thomson, A.: The us/uk world magnetic model for 2015-2020 (2015)

[18] Cornut, O.: Github - ocornut/imgui: Dear imgui: Bloat-free immediate mode graphical user interface for c++ with minimal dependencies. `https://github.com/ocornut/imgui` (2025), [Accessed 17-06-2025]

[19] D'Amore, G.: About Nanomsg. `https://nanomsg.org/` (2018), [Accessed 29-06-2025]

[20] Dean, E.: Atmospheric effects on the speed of sound. Citeseer (1979)

[21] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1(1), 269–271 (Dec 1959), `https://doi.org/10.1007/BF01386390`

[22] Ecke, S., Dempewolf, J., Frey, J., Schwaller, A., Endres, E., Klemmt, H.J., Tiede, D., Seifert, T.: Uav-based forest health monitoring: A systematic review. Remote Sensing 14(13), 3205 (2022)

[23] Elshaer, A., Elmanfaloty, R.A., Abou-Bakr, E., Elrakaiby, M., Saada, K.: Exploring algorithmic efficiency of a-star and dijkstra for optimal route planning in green transportation. International Journal of Intelligent Transportation Systems Research pp. 1–11 (2025)

[24] Everett, C.H.: Survey of collision avoidance and ranging sensors for mobile robots. Robotics and Autonomous Systems 5(1), 5–67 (1989), https://www.sciencedirect.com/science/article/pii/0921889089900419

[25] Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimization algorithms. In: 25th Annual Symposium onFoundations of Computer Science, 1984. pp. 338–346 (1984)

[26] Gageik, N., Benz, P., Montenegro, S.: Obstacle detection and collision avoidance for a uav with complementary low-cost sensors. IEEE Access 3, 599–609 (2015)

[27] Goyal, A., Mogha, P., Luthra, R., Sangwan, N.: Path finding: A* or dijkstra's? International Journal in IT & Engineering 2(1), 1–15 (2014)

[28] Harradi, R., Battseren, B., Heller, A., Hardt, W.: Areiom: adaptive research multicopter platform. In: IOP Conference Series: Materials Science and Engineering. vol. 1019, p. 012022. IOP Publishing (2021)

[29] Harradi, R., Heller, A., Roth, J., Hardt, W.: Mavlink uav hangar communication based on a cloud architecture. In: 2024 International Symposium ELMAR. pp. 301–305 (2024)

[30] Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)

[31] Huang, S., Teo, R.S.H., Tan, K.K.: Collision avoidance of multi unmanned aerial vehicles: A review. Annual Reviews in Control 48, 147–164 (2019), https://www.sciencedirect.com/science/article/pii/S1367578819300598

[32] Koenig, S., Likhachev, M.: Incremental A*. In: Dietterich, T., Becker, S., Ghahramani, Z. (eds.) Advances in Neural Information Processing Systems. vol. 14. MIT Press (2001), https://proceedings.neurips.cc/paper_files/paper/2001/file/a591024321c5e2bdbd23ed35f0574dde-Paper.pdf

[33] Koenig, S., Likhachev, M.: Fast replanning for navigation in unknown terrain. IEEE transactions on robotics 21(3), 354–363 (2005)

[34] Kwak, J., Sung, Y.: Autonomous uav flight control for gps-based navigation. IEEE Access 6, 37947–37955 (2018)

[35] Langley, R.: Nmea 0183: A gps receiver interfac standard. GPS world 6(7), 54–57 (1995)

[36] Majeed, A., Hwang, S.O.: Path planning method for UAVs based on constrained polygonal space and an extremely sparse waypoint graph. Applied Sciences 11(12), 5340 (2021), https://www.mdpi.com/2076-3417/11/12/5340

[37] MaxBotix: I2CXL-MaxSonar-EZ series. https://maxbotix.com/pages/i2cxl-maxsonar-ez-datasheet (2023), [Accessed 24-06-2025]

[38] Muchiri, G., Kimathi, S.: A review of applications and potential applications of uav. In: Proceedings of the Sustainable Research and Innovation Conference. pp. 280–283 (2022)

[39] Obermeyer, L., Miranker, D.P.: Clips++: Embedding clips into c++. In: NASA. Lyndon B. Johnson Space Center, Third CLIPS Conference Proceedings, Volume 1 (1994)

[40] Park, J.W., Oh, H.D., Tahk, M.J.: Uav collision avoidance based on geometric approach. In: 2008 SICE Annual Conference. pp. 2122–2126 (2008)

[41] Poggi, M., Tosi, F., Batsos, K., Mordohai, P., Mattoccia, S.: On the synergies between machine learning and binocular stereo for depth estimation from images: A survey. IEEE Transactions on Pattern Analysis and Machine Intelligence 44(9), 5314–5334 (2021)

[42] Prechelt, L.: An empirical comparison of seven programming languages. Computer 33(10), 23–29 (2000)

[43] Pérez-Carabaza, S., Scherer, J., Rinner, B., López-Orozco, J.A., Besada-Portas, E.: Uav trajectory optimization for minimum time search with communication constraints and collision avoidance. Engineering Applications of Artificial Intelligence 85, 357–371 (2019), https://www.sciencedirect.com/science/article/pii/S0952197619301411

[44] Rezwan, S., Choi, W.: Artificial intelligence approaches for uav navigation: Recent advances and future challenges. IEEE Access 10, 26320–26339 (2022)

[45] Rosen, S., Howell, P.: Signals and Systems for Speech and Hearing, vol. 29, p. 163. Brill, 2 edn. (2011), p. 163

[46] Saleh, S., Hasan, R., Battseren, B., Hardt, W., Ritter, M.: Optimizing monocular depth estimation for real-time edge computing platforms. In: 2024 International Symposium ELMAR. pp. 127–131. IEEE (2024)

[47] Satai, H.A., Zahra, M.M.A., Rasool, Z.I., Abd-Ali, R.S., Pruncu, C.I.: Bézier curves-based optimal trajectory design for multirotor uavs with any-angle pathfinding algorithms. Sensors 21(7), 2460 (2021)

[48] Stentz, A.: Optimal and efficient path planning for partially-known environments. In: Proceedings of the 1994 IEEE international conference on robotics and automation. pp. 3310–3317. IEEE (1994)

[49] Stentz, A., et al.: The focussed d* algorithm for real-time replanning. In: IJCAI. vol. 95, pp. 1652–1659 (1995)

[50] Stephan, M., Battseren, B., Tudevdagva, U.: Autonomous unmanned aerial vehicle development: Mavlink abstraction layer. In: International Symposium on Computer Science, Computer Engineering and Educational Technology (ISCSET-2020), Lauta Germany. pp. 45–49 (2020)

[51] Tang, L., Jia, X., Ma, H., Liu, S., Chen, Y., Tao, T., Chen, L., Wu, J., Li, C., Wang, X., et al.: Microwave absolute distance measurement method with ten-micron-level accuracy and meter-level range based on frequency domain interferometry. Sensors 23(18), 7898 (2023)

[52] Tjiharjadi, S., Razali, S., Sulaiman, H.A.: A systematic literature review of multi-agent pathfinding for maze research. Journal of Advances in Information Technology Vol 13(4) (2022)

[53] Wang, J., Li, Y., Li, R., Chen, H., Chu, K.: Trajectory planning for uav navigation in dynamic environments with matrix alignment dijkstra. Soft Computing 26(22), 12599–12610 (2022)

[54] Wu, Q., Chen, L., Liu, K., Lv, J.: UAV pathfinding in dynamic obstacle avoidance with multi-agent reinforcement learning. arXiv preprint arXiv:2310.16659 (2023)

[55] Wygant, R.M.: Clips — a powerful development and delivery expert system tool. Computers & Industrial Engineering 17(1), 546–549 (1989), `https://www.sciencedirect.com/science/article/pii/0360835289901216`

[56] Yang, S., Zhang, G.: A review of interferometry for geometric measurement. Measurement Science and Technology 29(10), 102001 (2018)

[57] Yaqot, M., Menezes, B.: The good, the bad, and the ugly: review on the social impacts of unmanned aerial vehicles (uavs). In: International Conference of Reliable Information and Communication Technology. pp. 413–422. Springer (2021)

[58] Yasin, J.N., Mohamed, S.A., Haghbayan, M.H., Heikkonen, J., Tenhunen, H., Plosila, J.: Unmanned aerial vehicles (uavs): Collision avoidance systems and approaches. IEEE access 8, 105139–105155 (2020)

[59] Yasin, J.N., Haghbayan, M.H., Heikkonen, J., Tenhunen, H., Plosila, J.: Formation maintenance and collision avoidance in a swarm of drones. In: Proceedings

of the 2019 3rd International Symposium on Computer Science and Intelligent Control. ISCSIC 2019, Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3386164.3386176

[60] Yu, Y., Tingting, W., Long, C., Weiwei, Z.: Stereo vision based obstacle avoidance strategy for quadcopter uav. In: 2018 Chinese Control And Decision Conference (CCDC). pp. 490–494. IEEE (2018)

[61] Yurtsever, E., Lambert, J., Carballo, A., Takeda, K.: A survey of autonomous driving: Common practices and emerging technologies. IEEE Access 8, 58443–58469 (2020)

[62] Zant, H.: Expert system-based embedded software module and ruleset for adaptive flight missions (2022)

[63] Zhang, J., Huang, H.: Occlusion-aware uav path planning for reconnaissance and surveillance. Drones 5(3), 98 (2021), https://www.mdpi.com/2504-446X/5/3/98

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-22-01 Henrik Zant, Reda Harradi, Wolfram Hardt, Expert System-based Embedded Software Module and Ruleset for Adaptive Flight Missions, September 2022, Chemnitz

CSR-23-01 Stephan Lede, René Schmidt, Wolfram Hardt, Analyse des Ressourcenverbrauchs von Deep Learning Methoden zur Einschlagslokalisierung auf eingebetteten Systemen, Januar 2023, Chemnitz

CSR-23-02 André Böhle, René Schmidt, Wolfram Hardt, Schnittstelle zur Datenakquise von Daten des Lernmanagementsystems unter Berücksichtigung bestehender Datenschutzrichtlinien, Januar 2023, Chemnitz

CSR-23-03 Falk Zaumseil, Sabrina, Bräuer, Thomas L. Milani, Guido Brunnett, Gender Dissimilarities in Body Gait Kinematics at Different Speeds, März 2023, Chemnitz

CSR-23-04 Tom Uhlmann, Sabrina Bräuer, Falk Zaumseil, Guido Brunnett, A Novel Inexpensive Camera-based Photoelectric Barrier System for Accurate Flying Sprint Time Measurement, März 2023, Chemnitz

CSR-23-05 Samer Salamah, Guido Brunnett, Sabrina Bräuer, Tom Uhlmann, Oliver Rehren, Katharina Jahn, Thomas L. Milani, Güunter Daniel Rey, NaturalWalk: An Anatomy-based Synthesizer for Human Walking Motions, März 2023, Chemnitz

CSR-24-01 Seyhmus Akaslan, Ariane Heller, Wolfram Hardt, Hardware-Supported Test Environment Analysis for CAN Message Communication, Juni 2024, Chemnitz

CSR-24-02 S. M. Rizwanur Rahman, Wolfram Hardt, Image Classification for Drone Propeller Inspection using Deep Learning, August 2024, Chemnitz

CSR-24-03 Sebastian Pettke, Wolfram Hardt, Ariane Heller, Comparison of maximum weight clique algorithms, August 2024, Chemnitz

CSR-24-04 Md Shoriful Islam, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Predictive Learning Analytics System, August 2024, Chemnitz

CSR-24-05 Sopuluchukwu Divine Obi, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Frontend for Agents in a Virtual Tutoring System, August 2024, Chemnitz

Chemnitzer Informatik-Berichte

CSR-24-06 Saddaf Afrin Khan, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Diagnostic Learning Analytics System, August 2024, Chemnitz

CSR-24-07 Túlio Gomes Pereira, Wolfram Hardt, Ariane Heller, Development of a Material Classification Model for Multispectral LiDAR Data, August 2024, Chemnitz

CSR-24-08 Sumanth Anugandula, Ummay Ubaida Shegupta, Wolfram Hardt, Design and Development of a Virtual Agent for Interactive Learning Scenarios, September 2024, Chemnitz

CSR-25-01 Md. Ali Awlad, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, AUTOSAR Software Component for Atomic Straight Driving Patterns, März 2025, Chemnitz

CSR-25-02 Billava Vasantha Monisha, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, Automotive Software Component for QT Based Car Status Visualization, März 2025, Chemnitz

CSR-25-03 Zahra Khadivi, Batbayar Battseren, Wolfram Hardt, Acoustic-Based MAV Propeller Inspection, Mai 2025, Chemnitz

CSR-25-04 Tripti Kumari Shukla, Ummay Ubaida Shegupta, Wolfram Hardt, Time Management Tool Development to Support Self-regulated Learning, August 2025, Chemnitz

CSR-25-05 Ambu Babu, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Retrieval Model based Backend of a Tutoring Agent, August 2025, Chemnitz

CSR-25-06 Shahid Ismail, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Generative Model based Backend of Tutoring Agent, August 2025, Chemnitz

CSR-25-07 Chaitanya Sravanthi Akula, Ummay Ubaida Shegupta, Wolfram Hardt, Integration of Learning Analytics into the ARC-Tutoring Workbench, August 2025, Chemnitz

CSR-25-08 Jörn Roth, Reda Harradi, Wolfram Hardt, Implementation of a Path Planning Algorithm for UAV Navigation, Dezember 2025, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz