ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Vision and Geometry Lab
Prof. Dr. Marc Pollefeys

Master Thesis

Obstacle Avoidance for Drones Using a 3DVFH* Algorithm

Spring Term 2018

Supervised by:
Dr. Lorenz Meier

Author:
Tanja Baumann

Declaration of Originality

I hereby declare that the written work I have submitted entitled

Obstacle Avoidance for Drones Using a 3DVFH* Algorithm

is original work which I alone have authored and which is written in my own words.[1]

Author(s)

Tanja	Baumann

Student supervisor(s)

Lorenz	Meier

Supervising lecturer

Marc	Pollefeys

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf`). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

[1] Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Contents

Preface

Unmanned Aerial Vehicles (UAVs) have many advantages over legged or wheeled systems. They are able to operate in a less cluttered environment up in the sky and they can achieve much higher velocities. Therefore, UAVs are valuable for a variety of tasks from consumer applications to search and rescue missions. The use of UAVs in commercial applications such as surveying or delivery has also increased significantly in the last few years. Special interest lies in autonomous UAV applications. Autonomous flight poses additional challenges as the trajectory has to be planned and crashes with obstacles must be avoided. Obstacle avoidance has been a very active field of research for many years. The algorithms have evolved and become more powerful. However, the problem is not yet solved conclusively and there is still a vast potential for improvement. This thesis is focused on the development of an obstacle avoidance algorithm for real-time flight on UAVs.

Abstract

Unmanned Aerial Vehicles (UAV) possess a vast potential for various autonomous applications such as surveying or delivery. When considering autonomous flight operations one of the most essential requirements certainly is a reliable obstacle avoidance mechanism. Obstacle avoidance is a very active field of research without any conclusive solutions so far. This thesis introduces the 3DVFH* obstacle avoidance algorithm suitable for real-time application on UAVs. The algorithm combines the ideas behind the previously presented 3DVFH+ and the VFH* algorithm with a novel memory strategy. The 3DVFH* algorithm computes obstacle avoidance maneuvers in a purely reactive manner without the need to build a global map of the environment. The memory strategy keeps track of previously seen obstacles by propagating the previous polar histogram to the current location. Various features, such as ground detection and safety mechanisms, have been implemented to increase the robustness and render application on UAVs possible. The 3DVFH* algorithm has been shown to effectively avoid obstacles in complex simulation scenarios exhibiting a look-ahead capability. The ground detection was shown to be able to keep a specified minimum distance to the ground. Real-world flight tests were performed by running the 3DVFH* algorithm on-board the Intel® Aero Ready to Fly drone equipped with only one forward facing camera. The drone was able to successfully avoid an obstacle in all of the 26 test flights.

Acknowledgements

I would like to thank Lorenz Meier for the supervision of this thesis. Without his guidance, resources and connections this project would not have been possible. I would also like to thank all the people from the Advanced Technology Labs AG who were always available to brainstorm and lend a hand wherever needed. Especially I would like to thank Christoph Tobler for providing his piloting skills and for joining me to defy wind and cold during the various hours of outdoor tests. Special thanks goes to Martina Rivizzigno, who has not only been supporting me throughout this thesis but also become a great friend. I am also deeply grateful to my parents who have always encouraged me to work towards my goals and supporting me though everything.

Symbols

Symbols

ϵ	elevation angle
ζ	azimuth angle

Indices

x	x axis
y	y axis
z	z axis
old	data from last computation cycle

Acronyms and Abbreviations

A*	Path planning algorithm (variant of Dijkstras algorithm using heuristics)
CVM	Curvature Velocity Method
DWA	Dynamic Window Approach
EKF	Extended Kalman Filter
ETH	Eidgenössische Technische Hochschule
FOV	Field of View
IVFH*	Variant of the VFH* algorithm
IMU	Inertial Measurement Unit
MPC	Model Predictive Control
MSV	Mobile and Static Vector Field Method
ROS	Robot Operating System
UAV	Unmanned Aerial Vehicle
VFH	Vector Field Histogram (an obstacle avoidance algorithm)
VFH+	Variant of the VFH algorithm
VFH*	Combination of the VFH algorithm with an A* algorithm

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAVs) have become increasingly present in research as well as in every day's life over the last years. They have many advantages over ground bound robots. For example, UAVs are able to operate in a less cluttered environment up in the sky and also have the ability to achieve much higher velocities. The use cases for UAV systems are very broad and reach from consumer applications to military and search and rescue missions in catastrophe areas. UAVs have also become increasingly present in commercial uses such as surveying or delivery. All of those applications have in common that they would benefit from or even rely on a dependable obstacle avoidance strategy. For consumer applications, often the pilots are inexperienced. The obstacle avoidance algorithm is therefore necessary to prevent damage to the vehicle as well as to avoid dangerous situations involving bystanders or foreign property. In commercial applications as well as for use in military or search and rescue missions, an increased amount of autonomy is required. An autonomous UAV flight is heavily reliant on an obstacle avoidance algorithm to be able to complete its mission without crashing into any objects.

Obstacle avoidance is a very active field of research as it is not an easy problem to solve. One of the most prominent issues is the limited compute power on board of a UAV. UAVs are usually built to be light and agile, therefore they have a very limited amount of payload and cannot carry large computing units. If a high-power computing unit is needed, the vehicle has to deal with increased power consumption which reduces the flight time. Also those UAVs need to be larger, which makes them more expensive and dangerous. However, there have been attempts to use a ground station for the computationally intensive calculations. This solution allows to have small agile UAVs and use a computationally intensive obstacle avoidance algorithm. On the other hand, it makes the UAV reliant on a stable and fast connection to the ground station, which often is not a realistic assumption. The ground station also increases the latency of the algorithm. Therefore, the goal would be to develop an algorithm which is feasible to execute on board. Another prominent difficulty in obstacle avoidance is the limited sensing capability of a UAV. As the payload of a UAV is limited, so is the amount and weight of the sensing units. Small and light sensors are required, which usually are very expensive if they shall produce good quality data. Especially in consumer products it is therefore not feasible to have high quality sensors. A further issue which complicates the obstacle avoidance problem is the imperfect state estimation of a UAV. UAVs do not have the possibility to use wheel encoders or similar strategies for state estimation. They usually rely on GPS, IMU and vision based algorithms to estimate their state which often is subject to significant uncertainties. The goal of this thesis is to develop a new obstacle avoidance strategy which is computationally simple enough to run on-board of a UAV and is able to cope with all those uncertainties and limitations.

Originally there have been two different approaches to obstacle avoidance. The first approach could be called global obstacle avoidance. It either has access to a map of the environment or builds it at runtime from sensor data. From this map a complete path to the goal is calculated using a path finding algorithm. Those global obstacle avoidance algorithms are usually too computationally expensive to run on-board of a regular size UAV. They require large systems or the use of a ground station. On the other hand there are the local or reactive approaches. They do not build a map but act on the sensor data gathered at the current time-step. This usually makes them unable to find the optimal path, however, those algorithms are fast enough to run on-board a UAV. A frequently used method for local obstacle avoidance is the Vector Field Histogram (VFH) [1, 2]. The work in this thesis will be based on the VFH method and will stretch it towards a global approach. A variant of the VFH algorithm called VFH* is used [3]. The VFH* algorithm builds a look-ahead tree and therefore combines the local method with A* path planner. The VFH* method was only used in 2D environments so far. We will extend it to the 3D space to make it usable for UAV applications. Also a memory will be added to the VFH method to keep track of previously seen obstacles. Those changes allow the use of some advantages inherent to global approaches without sacrificing the low computational effort of the algorithm. The goal of this thesis is to show successful obstacle avoidance in outdoor flights of a small UAV.

Chapter 2

Related Work

Obstacle avoidance algorithms have received a lot of attention over the last few years. But the foundations of obstacle avoidance research go back to the 90s. Where at the beginning the algorithms were only designed for robots moving in a 2D environment, but as the robotic hardware has evolved, recently many of them have been adapted to consider 3D environments. There exist two main categories of obstacle avoidance algorithms. On one hand there are the local or reactive approaches. Those algorithms do not build a map of the environment or save obstacle positions. They calculate the best reaction from the current sensor data. The advantage of local algorithms is that they have low computational cost, which is crucial to run on-board small robotic devices. However, as they do not plan a complete path they are unable to find the optimal path to the goal. Those algorithms are also prone to get stuck in local minima. On the other hand there are the global approaches. Those algorithms have access to a map of the environment which is already precomputed or they build the map themselves while encountering obstacles. Often path planning algorithms, such as dynamic programming of A* algorithms, are used to determine the best path from the given map. Those global algorithms are less prone to get stuck in local minima as they consider all the available information about the terrain. In the case where the whole environment is mapped they might even yield an optimal solution. However, global methods require a lot of computation power. Goerzen et al. [4] give an overview over the most common obstacle avoidance algorithms, which is focused mostly on global approaches. They further distinguish between algorithms which take kinematic and dynamic constraints of the robotic platform into account and those which do not. In the next paragraphs research in global and local obstacle avoidance algorithms is summarized.

2.1 Global Obstacle Avoidance

Most global obstacle avoidance algorithms either have access to a map of the environment or build a map from the acquired sensor data. This map is then searched for the optimal path to the goal using a path finding algorithm. Heng et al. [5] use a stereo camera to acquire a point-cloud of the environment, which is stored in a Octomap [6]. This Octomap is used to calculate a search tree from predetermined motion primitives of the UAV. An Anytime Dynamic A* (ADA*) algorithm is then used to find the best path. Flight tests with the algorithm running on-board a UAV succeeded. Hwangbo et al. [7] use an A* algorithm on a given 3D map of the environment to calculate a first coarse path for a fixed-wing UAV. This path is then refined using local area of the map and building a search tree from precomputed

motion primitives. Fraundorfer et al. [8] use an UAV for autonomous mapping and exploration of an unknown environment. The UAV is equipped with a stereo camera from which an occupancy map is built on a ground station using SLAM with loop closure and re-localization. A frontier algorithm determines the new goal for the UAV and the VFH+ algorithm is used to navigate around obstacles. Mohammadi et al. [9] use a similar method in 2D to find a path for a surgical needle in a 2D environment. The map is given by the X-Ray image and the space is tessellated into hexagonal cells as hexagons provide smoother paths then squares. This grid is searched for the optimal path using an A* algorithm. Stachniss and Burgard [10] use an A* algorithm to find a path for a ground robot in 2D space. On this path a subgoal is defined and another A* algorithm is used to search the five dimensional configuration space, providing a look-ahead not only in position, but also in velocity and orientation. This algorithm has been compared to the Dynamic Window Approach (DWA) introduced by Brock and Khatib [11]. The DWA algorithm determines the feasible velocity space such that only velocities are considered with are reachable in the next timstep and allow the robot to stop in front of obstacles. The look-ahead tree in [10] enables the robot to slow down before an opportunity to turn comes up, whereas the DWA overshoots and misses the turn because it does not reduce the velocity beforehand.

There have also been studies using Extended Kalman Filters (EKF) to track distinct obstacles to be able to cope with dynamic scenes. The final path is then calculated by applying dynamic programming [12] or by drawing a collision cone around the obstacles and limiting the velocity space to vectors on the outside [13]. Sasiadek and Duleba [14] use a two layer approach, where first a coarse path is planned and then individual obstacles are traced at a safe distance.

Another method to build search trees uses probabilistic properties in the hope of reducing the computational cost of the algorithm. Hrabar [15] draws a probabilistic road-map into the occupancy grid by randomly adding points and connecting them if a path between them exists. This tree is then searched using a D* Lite algorithm which does not recalculate the path at every time-step but only updates it when new information becomes available. Kuffner and LaValle [16] use two random trees in a given map, one grown from the start position and one from the goal. Those trees are expanded alternatively until they can be connected. The simulation returns a feasible path in position and orientation space.

A further family of global avoidance algorithms are based on optimization techniques. Deits and Tedrake [17] use mixed-integer programing to plan smooth UAV trajectories in simulation. Shim et al. [18, 19] use a Model Predictive Control (MPC) approach. The cost-function was designed as a potential field and only the closest obstacle was considered in the optimization. In real-world tests, the algorithm ran on a large UAV with 20 kg payload, which was able to avoid tents on a field which were spaced at a distance of 10 m.

2.2 Local Obstacle Avoidance

The most common strategy for local obstacle avoidance is the Vector Field Histogram (VFH) introduced by Borenstein and Koren [1]. A sonar is used to generate a 2D occupancy grid of the environment, which was then mapped to 1D polar histogram. From that histogram free directions can be extracted. An improvement by Ulrich and Borenstein [2] called VFH+ considers the maximum turn radius of the robot and safety margins. Based on the VFH+ algorithm there have been

many adaptions and variants. Song and Huang [20] use an optical flow algorithm to generate the polar histogram and adapts safety margins around obstacles to their distance. Yamauchi [21] also uses distance dependent safety margins around obstacles. The Mobile and Static Vector Field Method (MSV) [22] is an adaption to VFH+ which is able to cope with moving obstacles by estimating their velocity from consecutive histograms. Another variant is the VFH* introduced by Ulrich and Borenstein [3]. This algorithm combines the advantage of the A* planning algorithm with the local properties of the VFH+ algorithm. A look-ahead tree is built where at every node the VFH+ algorithm is executed to find feasible directions. The tree is then searched for the best path using the A* algorithm. There have been attempts to enhance the VFH* algorithm to take moving obstacles into account. Jie et al. [23] developed the IVFH* algorithm which uses spring like forces on the nodes of the search tree to adapt them to moving obstacles. Babinec et al. [24, 25] determine the velocity of obstacles from several subsequent states and use this knowledge while building the search tree. So far all VFH algorithms have been used in a 2D environment for ground bound robots, however, Vanneste et al. [26] have developed the 3DVFH+ algorithm. This algorithm cannot be counted as a completely local algorithm as it builds a map of the environment in the form of an Octomap [6]. From this Octomap a 3D occupancy map is built by considering only voxels in a bounding box around the vehicle. From the 3D occupancy map a 2D polar histogram is built in which openings are detected and evaluated to find the best direction for moving in a 3D environment.

Some recent studies have also been using machine learning approaches to determine the correct avoidance maneuver from sensory input. There has been work to combine the VFH method with machine learning and fuzzy logic [27, 28]. An early version of a fuzzy logic method influenced by potential fields was presented by Zavlangas et al. [29]. On the other hand, Ross et al. [30] learn the correct control input directly from sensor data and expert demonstrations. A continued study [31] uses receding horizon control to evaluate a preselected set of feasible trajectories on a cost map, instead of the learnt purely reactive control.

Another popular method used for local obstacle avoidance is the potential field method, which is first introduced by Khatib [32]. There have been various adaptions to this method over the years especially to find good potential functions and avoid local minima. Waydo [33] used stream functions and considered the velocity of obstacles for the generation of the potential field in a 2D application. Mac et al. [34] used the potential field method to navigate a UAV through an indoor environment.

There are some further propositions for local obstacle avoidance such as the strategy by Oleynikova et al. [35]. Obstacle positions are determined by converting the disparity map into a column-accumulated U-map. In this U-map bounding boxes are fitted around distinctive areas. Those boxes are then converted to ellipses and projected into the 2D space at the current height of the UAV and inflated by a safety radius. Flight tests showed that the UAV was able to avoid sparse obstacles when choosing the waypoints to lie outside the ellipses. Another family of computationally cheap algorithms for local obstacle avoidance in 2D are the Bug algorithms. A comparison of different Bug algorithms can be found in [36]. Buniyamin et al. [37] developed the PointBug algorithm which was able to outperform other Bug algorithms, especially in environments with little sharp edges and turns. An early method for local obstacle avoidance is the Curvature-Velocity Method (CVM) by Simmons [38]. CVM achieves real-time performance by approximating how far a robot could travel on a given curvature before hitting an obstacle. Additional

velocity constraints can be imposed based on the physical limitations of the robot.

2.3 Impact on this Thesis

From all the published research on obstacle avoidance very few showed successful real-world tests on UAVs. Successful flight tests have been performed using MPC methods [18, 19], a global approach using an Octomap and A* algorithm [5], machine learning [30, 31] or Potential field methods [34]. However, most of the discussed local approaches are designed for application in 2D environments only. Even though there has been quite excessive and promising research on historgramic methods, they were not yet applied to UAVs in three dimensions. The 3DVFH+ algorithm [26] has only been tested in simulation. All the other adaptions to the historgramic approach such as building a look-ahead tree have not yet been extended to a 3D environment. As a consequence there is huge unused potential in histogramic methods for UAV flight. The goal of this thesis is to combine the VFH* algorithm with the 3DVFH algorithm and extend it with additional features to achieve a smooth UAV flight with dependable obstacle avoidance behavior.

Chapter 3

Approach

3.1 The 3DVFH Algorithm

The obstacle avoidance algorithm implemented for this thesis is based on the 3DVFH algorithm introduced by Vanneste et al. [26]. Vanneste et al. build a global map of the environment in the form of an Octomap. From this global map local information in a bounding box around the UAV is extracted to perform the histogramic obstacle avoidance. Therefore, this obstacle avoidance strategy lies somewhere in between global and local approaches. Having access to a global map has the main advantage that the algorithm remembers previously seen obstacles which might not be in the FOV anymore. But building a global map also introduces a lot of computational overhead. Therefore, in this thesis the 3DVFH method is implemented as a purely local algorithm without building a global map. The global map is replaced by direct usage of the 3D point-cloud provided by the stereo camera and a computationally less expensive memory strategy is developed to mitigate the inherent issues of a local approach.

From the cropped point-cloud information, a 2D polar histogram is built as shown in figure 3.1 . For every 3D point in the cropped point-cloud the azimuth and elevation angle with respect to the UAV position are calculated. The point is then placed in the corresponding histogram bin. In the primary polar histogram, each cell holds the number of how many 3D points fall into its sector. In this implementation the primary polar histogram is not masked for minimum turn radii as slow flight is assumed and the drone is able turn on the spot. The primary polar histogram is converted into a binary polar histogram by comparing the point count in each cell with a threshold. To consider the UAV size as well as a minimum distance to the obstacles, histogram cells inside a safety margin around occupied cells are also considered as blocked. The size of the this safety margin is dependent on the obstacle distance as further discussed in chapter 3.3.3.

A cost function is evaluated for all free cells in the resulting histogram. The cost function contains a goal orientation term and a smoothing term. The goal orientation term compares the evaluated direction to the goal direction. The smoothing term compares the evaluated direction to the direction chosen in the last time-step. The cost function is discussed in more detail in chapter 3.3.4. The histogram cell with the lowest cost will be chosen as the direction for movement. In the case when there is no obstacle present the histogram consists only of unoccupied cells and the UAV is allowed to go faster. In this case the waypoint is chosen to lead directly towards the goal instead of determining it from the histogram. This reduces the

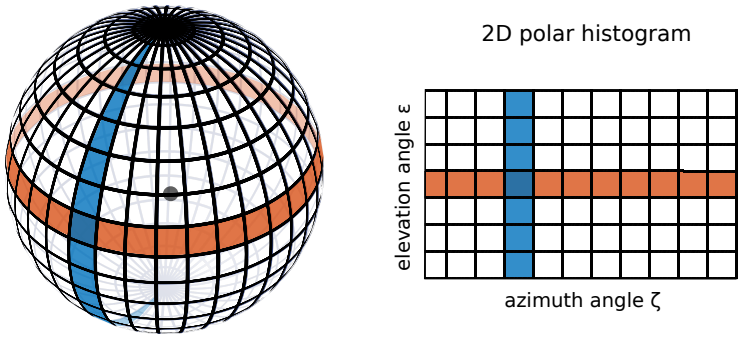


Figure 3.1: *The elevation and azimuth angle with respect to the UAV position are calculated for every 3D point. The points are then mapped to the corresponding 2D histogram bin.*

computation time, as there is no need to evaluate the cost function for every free cell, and it helps avoiding the effects of discretization.

3.2 Building a Memory for 3DVFH

The implemented 3DVFH algorithm is purely local and reactive, which means that the algorithm has only access to the data of the current time-step and in proximity to the drone. One of the biggest drawbacks of local avoidance strategies is, that they do not consider any data or action from previous time-steps. This often leads to unstable behavior and local minima. A movement in one direction causes another to be favorable because the previous data is discarded and the drone has no knowledge about the previously seen part of the obstacle. Especially flying along bigger obstacles such as walls is therefore impossible using a local VFH method. To mitigate those problems the 3DVFH algorithm has been enhanced to include a memory of previously seen objects. This memory is built in accordance with the polar histogram principle of the 3DVFH algorithm, which allows to memorize obstacles without the need to store large datasets. The strategy consists of four steps. The final histogram of every calculation is stored, such that it is available in the subsequent time-step. In a first step, the data from this old histogram is projected into 3D points. In a second step a memory histogram is built from those re-projected points at half the resolution. This memory histogram is then up-sampled to the regular resolution. In a third step the new histogram of the current time-step is calculated from the new depth map data. Finally, the new histogram and the memory histogram are fused together into the combined histogram of the current time-step. This combined histogram will be used to determine possible flight directions and to navigate around obstacles. This strategy is shown schematically in figure 3.2. In the following paragraphs the individual steps towards the combined histogram are explained in more detail.

3.2.1 Re-projection of Histogram into 3D Points

The memory approach uses data from the last time-step to enhance the histogram built at the current time-step. The stored old data, however, is in histogram form. The old histogram is a polar representation of the obstacles seen at the previous time-step. As the angular direction of an obstacle depends on the location of the drone, the old histogram is bound to the old location at which it was built. To use the data of the old histogram for the calculations in the current time-step, it

needs to be converted into a histogram at the current drone location. This can be achieved by re-projecting the occupied cells in the old histogram into 3D points and then building a memory histogram from those 3D points. To get the re-projected points, the old histogram is scanned for occupied cells. Each of those occupied cells is characterized by an elevation angle ϵ and an azimuth angle ζ. In addition to those angles, a distance is stored in the distance layer of the old histogram and an age in the age layer. If the age of the occupied cell does not exceed a defined limit, the cell is re-projected into four 3D points corresponding to the corners of the cells. This process is shown in figure 3.3. The elevation and azimuth angles of the four corner points can be calculated by adding/subtracting half the angular resolution α of the histogram from the elevation and azimuth angles of the occupied cell as shown in equation 3.1. Those four points can then be re-projected into 3D space using equation 3.2.

$$\begin{aligned} \epsilon_1 &= \epsilon + \frac{\alpha}{2} & \zeta_1 &= \zeta + \frac{\alpha}{2} \\ \epsilon_2 &= \epsilon - \frac{\alpha}{2} & \zeta_2 &= \zeta + \frac{\alpha}{2} \\ \epsilon_3 &= \epsilon + \frac{\alpha}{2} & \zeta_3 &= \zeta - \frac{\alpha}{2} \\ \epsilon_4 &= \epsilon - \frac{\alpha}{2} & \zeta_4 &= \zeta - \frac{\alpha}{2} \end{aligned} \tag{3.1}$$

$$\begin{aligned} p_{i_x} &= pos_{old_x} + d \cdot cos(\epsilon_i \frac{\pi}{180}) sin(\zeta_i \frac{\pi}{180}) \\ p_{i_y} &= pos_{old_y} + d \cdot cos(\epsilon_i \frac{\pi}{180}) cos(\zeta_i \frac{\pi}{180}) \\ p_{i_z} &= pos_{old_z} + d \cdot sin(\epsilon_i \frac{\pi}{180}) \end{aligned} \tag{3.2}$$

3.2.2 Building a Histogram from 3D Points

To achieve a histogram representation of the environment the 3D points first need to be converted into a 2D primary polar histogram where every cell holds the number of points lying in the corresponding sector. For every 3D point p_i the elevation angle ϵ and azimuth angle ζ seen from the current position *pos* of the drone is calculated according to equations 3.3, 3.4. After the angles of the 3D point are determined, they need to be converted into histogram indices. The calculated angles are in the range $\zeta \in [-180, 180]$ and $\epsilon \in [-90, 90]$. First those angles need to converted into positive ranges $\zeta \in [0, 360]$ and $\epsilon \in [0, 180]$. From those positive angles the histogram indices can be calculated according to equations 3.5, 3.6. Where β is the positive angle (here ϵ or ζ) which needs to be converted and α is the resolution of the histogram.

$$\zeta = \frac{180}{\pi} \cdot atan2\big(p_x - pos_x, p_y - pos_y\big) \tag{3.3}$$

$$\epsilon = \frac{180}{\pi} \cdot atan\left(\frac{p_z - pos_z}{\sqrt{(p_x - pos_x)^2 + (p_y - pos_y)^2}}\right) \tag{3.4}$$

$$\beta_{temp} = \beta + \Big(\alpha - \big(\beta \% \alpha\big)\Big) \tag{3.5}$$

$$\beta_{index} = \frac{\beta_{temp}}{\alpha} - 1 \tag{3.6}$$

This procedure is used twice in the process of getting the combined histogram for navigation: To get the memory histogram from the re-projected points and to calculate the new histogram from the point-cloud provided by the stereo camera. In both cases the primary polar histogram is calculated in the same way. However, the thresholds for conversion to a binary histogram are different as the total 3D point count largely differs. Also the cell size of the memory histogram is larger as it is built at half the resolution. For the new histogram, a bin in the binary layer is set to occupied if at least one 3D point falls into it. The corresponding value in the distance layer is set to the mean of the distances of all points which fall into that bin. The value in the age layer is always set to zero, as this histogram is generated from new data. The memory histogram on the other hand is generated at half the resolution of all the other histograms and then up-sampled. A cell in the binary layer is considered as occupied, if at least six points fall into it. This threshold needs to be chosen with care. If every cell of the old histogram is split into four 3D points, then six is the only admissible value. If the threshold is chosen lower than six, the memory histogram has more occupied cells than the old histogram. This causes the combined histogram to converge to a completely occupied histogram. However, if the threshold is chosen too large, the memory histogram will have less occupied cells than the old histogram. The memory effect would be minimized and especially the borders of obstacles would be neglected. This effect is visualized in figure 3.4. So the optimal choice for the threshold is the largest number which does not lead to a convergence to a full histogram. For the distance layer, the same approach as for the new histogram is used. The distance value is therefore defined a the mean distance of all the points in the bin. The age value is also determined as the mean age of all points in the bin. After the memory histogram is created at half the resolution, it is up-sampled to match the full resolution of the new histogram. The resolution change makes the method less susceptible to discretization errors.

3.2.3 Combining Memory Histogram and New Histogram

In the last step, both the memory histogram and the new histogram need to be combined to a final histogram. The final histogram is used for navigation and stored for the next calculation. To combine both histograms, two different regions are distinguished: The cells which lie inside the current FOV of the drone versus the cells outside the FOV. The histogram region lying inside the FOV can be calculated from the yaw and pitch angle of the drone and the sensor specifications. The Realsense camera used has a vertical FOV of 46°and a horizontal FOV of 59°. It is possible that the new histogram has occupied cells outside the FOV, because the FOV is chosen in a very conservative way. It marks the area where the new data is trusted, not only that occupied cells contain obstacles but also that empty cells are really free. On the other hand unoccupied cells outside the FOV usually are caused by a lack of data rather than knowledge about a free cell. For the histogram cells inside the FOV the new histogram is copied and therefore given priority over the memory histogram. For all the cells lying outside the FOV, an OR operation is used for the binary layer. The age and distance layer will be taken from the new histogram, if the corresponding cell is marked as occupied. Only in the case where the cell is occupied only in the memory histogram, also the age and distance layer values are copied from there. An example for this histogram combination can be seen in figure 3.5.

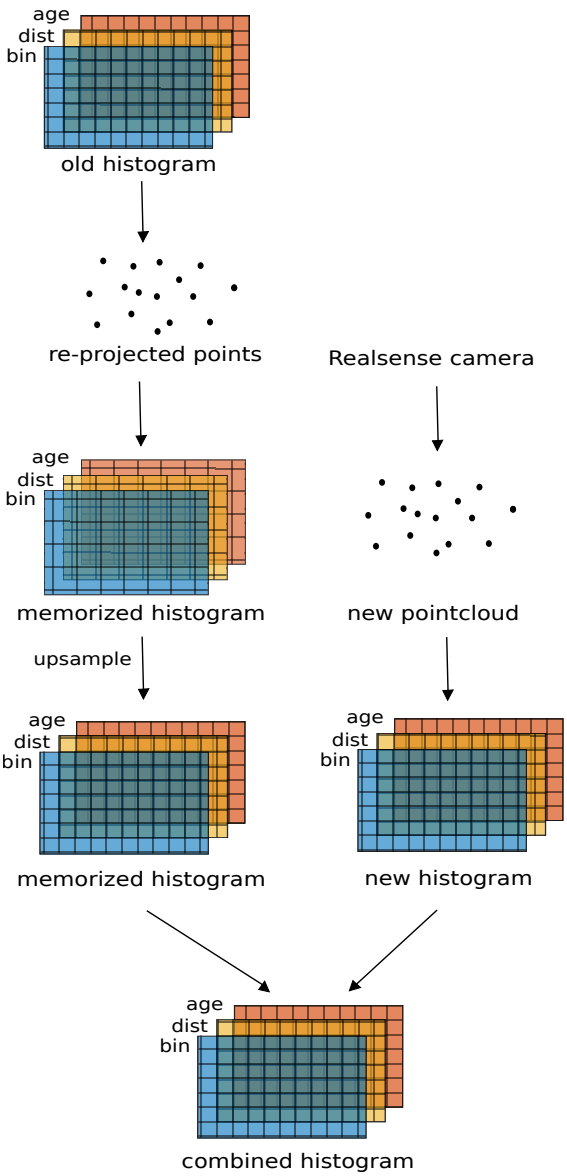


Figure 3.2: *Strategy to include previously seen data into the current histogram. The original 3DVFH algorithm uses the new histogram only, while the memory approach enhances the algorithm by combining the new histogram with propagated data from the last time-step.*

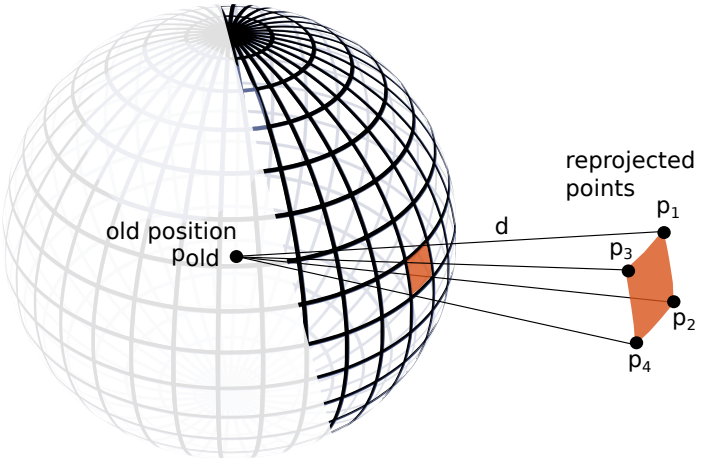


Figure 3.3: *Each occupied cell in the old histogram corresponds to an obstacle at the corresponding elevation and azimuth angle. Using the stored distance d the occupied cell can be re-projected into its four corner points in 3D space to approximate the location of the original obstacle.*

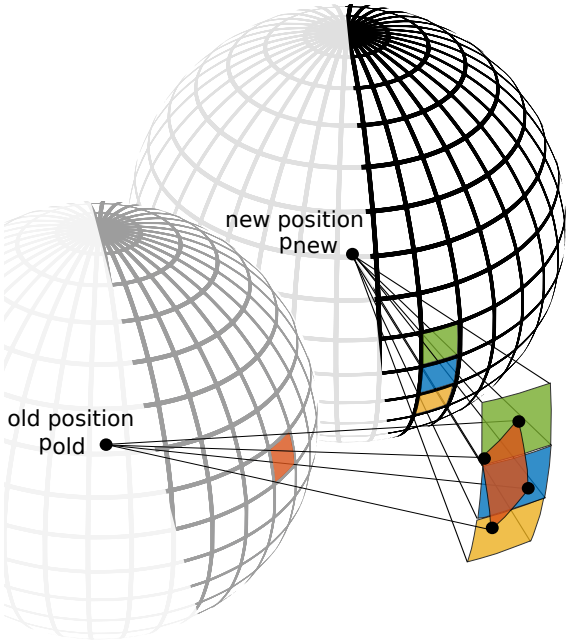


Figure 3.4: *The threshold for the minimum number of points which make up an occupied cell needs to be chosen carefully. It could lead to no memory effect if chosen too large or to convergence to a completely full histogram if chosen too small. In this figure, if the threshold is chosen at one point, the memory histogram would have three occupied cells. If the threshold is chosen at two, the memory histogram would have one occupied cell. And for all thresholds larger than two the memory histogram would be empty.*

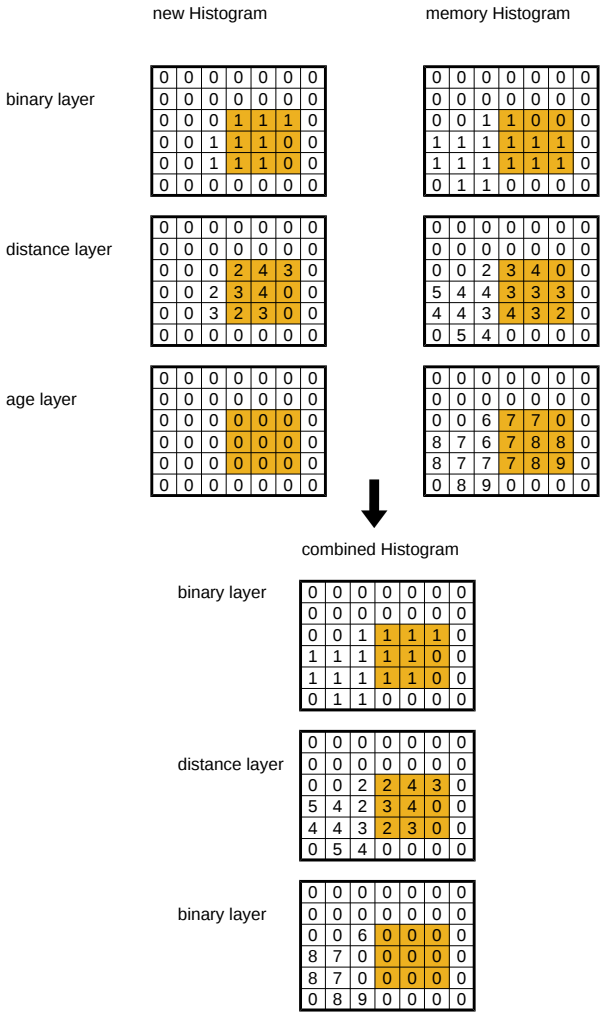


Figure 3.5: *An example on how to combine the new histogram and the memory histogram. The blue area marks the FOV.*

3.3 Adaptions for Safety and Corner Cases

The 3DVFH algorithm is of a theoretical nature and has never been implemented on a physical system. To be used in practice a few adaptions and safety measures are necessary. An inherent drawbacks of all histogramic approaches is their directional nature. The histogram gives information over the direction of obstacles but it has no knowledge about their distances. As already mentioned in chapter 3.2, the histogram of the memory 3DVFH algorithm additionally owns a distance layer. This layer is necessary to store the distances to be able to re-project the histogram cells to the correct 3D location. However, this distance layer also has a large potential for adaptions to the algorithm. Additionally, when flying with a UAV there should be some safety measure to prevent crashes for cases where the obstacle avoidance fails.

3.3.1 Move only in Directions Inside the FOV

Especially difficult situations arise, when two options have very similar cost. In this case, the algorithm might switch between the two options multiple times. Those situations occur often when an obstacle lies straight between the drone and the goal. In this case, avoiding the obstacle to the right might be just as good as avoiding it to the left. The movement choice to the left as well as the right side will still have a forward component. If the drone chooses to go to the right side, it will move forward and yaw to the right simultaneously. Because of the yaw motion the FOV changes and new obstructed areas on the right of the obstacle become visible. This causes the left direction to be more attractive and the drone will change its direction to the left. Again a new part of the obstacle becomes visible due to the change of the FOV, which again causes the drone to change directions. If the algorithm switches between the the two options too often, the drone gets closer and closer to the obstacle and will eventually crash as shown in figure 3.6.
A measure which significantly reduces the risk of crashing is, if movement is only allowed into directions inside the FOV. If the next waypoint lies outside the FOV, the drone has to hover and yaw until the waypoint enters the FOV. In the situation pictured in figure 3.6 the drone would first hover and turn to the right all the while acquiring data from the right, previously unknown, part of the obstacle. If including the information from the right obstacle part passing on the left becomes more attractive, the drone will continue to hover and yaw to the left. In this case all parts of the obstacle, which are visible from the current position of the drone, have

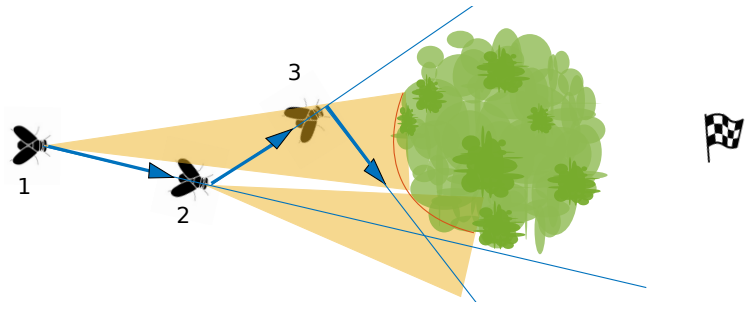


Figure 3.6: *The yellow area marks the FOV of the UAV. Once a direction is chosen, new parts of the obstacle become visible (red part of the obstacle is what is known at the position 2) and cause the other direction to be favorable. The forward component in movement direction leads to crash if the the controller alternates between two choices.*

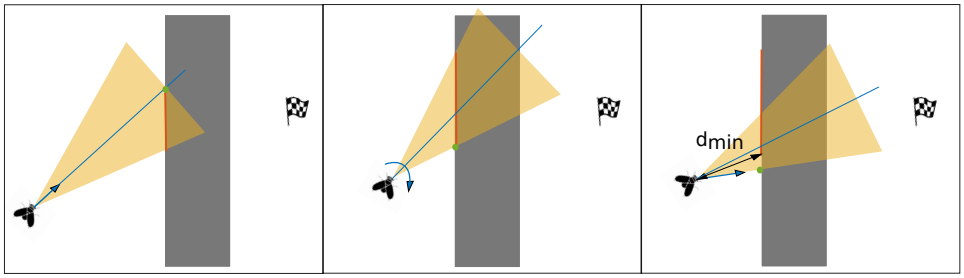


Figure 3.7: *In the leftmost figure, the drone is flying along a long obstacle. For large obstacles, the drone will come quite close due to the forward component of the flying direction. If the obstacle is too long on the left side, a change of direction becomes more attractive and the drone hovers and yaws to the right as shown in the middle. If the obstacle lies closer than the minimum perception distance, a part of the obstacle will not be seen and the drone flies straight into it.*

been perceived and stored in the memory. Therefore, the drone will chose the better direction from this information, turn and move. In real-world tests the drone might turn more than once in each direction due to the noise in the point-cloud. This feature does not solve the problem of switching between possibilities but eliminate the risk of crashing emanating from it.

3.3.2 Safety Measure: Back-off

In a few corner cases, the drone might come very close to obstacles due to the limited FOV of the camera. The used Realsense camera also has a minimum distance d_{min} for perception, which lies around 20 cm. Obstacles which lie closer than this minimum distance will not be perceived at all. In real-world scenarios the minimum distance might be larger, obstacles closer than 50cm are seen rather unreliably depending on noise and lighting conditions. Figure 3.7 shows one of the most common critical situations. In the left figure the drone is flying along the wall. The initial decision was to avoid the obstacle to the left. The drone has already been flying along the wall to the left for some time. Due to the forward component in the moving direction, the drone has come quite close to the obstacle. Now if the wall is very long on the left side, avoiding it to the right side becomes more attractive and the drone will change its direction. It will hover and and yaw to the right. If the wall lies closer than d_{min} the drone will not seen this part of the wall. This would cause the drone to fly straight into the obstacle. These holes in the point-cloud are an inherent problem of the sensor equipment of the drone and cannot be prevented. The risk of crashing, however, can be significantly reduced by tracking the closest part of the existing point-cloud. Blind spots are usually accompanied by other parts of the obstacle, which lie a bit further from the sensor and can be seen in the point-cloud. Those parts of the point-cloud are usually at a distance close to d_{min}. If such close points are detected, the drone needs to move backwards, such that the possibility of holes in point-cloud can be reduced. Moving backwards is suboptimal, as the drone has no sensors at the back. But the risk of crashing is much higher if the drone keeps going forward. The holes in the point-cloud usually lie in very attractive directions, such that the probability of choosing to go there is quite high. The space behind the drone on the other hand is most likely to be free, as the drone came from that direction. Therefore, if some points in the point-cloud lie closer than a specified distance, the drone will back-off from that point by 1 m in the x-y plane.

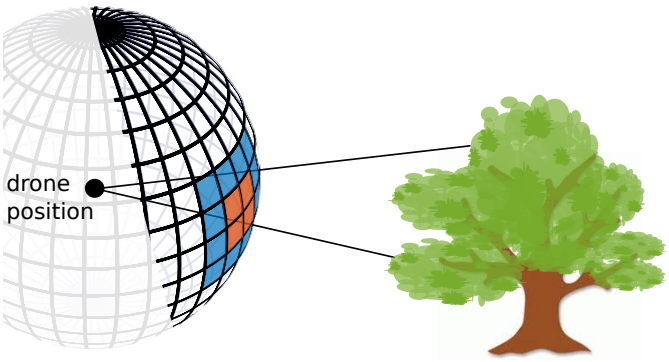


Figure 3.8: *The occupied histogram cells are marked in red and cells inside the safety margin are marked in blue. All colored cells are not eligible as candidate directions.*

3.3.3 Safety Margin in Histogram Dependent on Obstacle Distance

The occupied cells in the histogram are enlarged by a safety margin. Some adjacent cells are also marked as blocked such that those directions cannot be chosen as candidates. In figure 3.8 the blocked cells are marked in red and the additionally discarded cells in the safety margin in blue. If the safety margin is set too high, the drone cannot pass gaps between obstacles anymore or in the worst case cannot make any progress towards to the goal. If the margin is set too low, the risk of crashing is increased. In proximity to obstacles the safety has priority over progress and an increased safety margin should be chosen. Whereas when the obstacles are far away a smaller margin is beneficial. To take that into consideration, two different margins have been implemented. The distance of the closest obstacle determines, which size of safety margin is chosen. Every change of the margin size often introduces a change in the desired direction as usually the desired direction lies next to blocked cells. To avoid oscillations, a hysteresis is implemented for switching between the differently sized margins.

3.3.4 Adaption of Cost-Parameters Dependent on Progress

After the histogram is built, all directions which are neither blocked nor inside the safety margin are considered as candidate directions. The cost of every direction is evaluated using a cost function and the best candidate direction is chosen for the movement. The cost function considers how close the candidate direction is to the goal direction and how close it is to the selected direction of the last time-step. The first criterion of goal orientation is split into three different parts: Yaw difference Δ_{yaw}, pitch difference upwards Δ_{pitch_up} and pitch difference downwards Δ_{pitch_down}. To calculate the difference in yaw and pitch, the candidate direction is projected to a point at the same distance as the goal. As shown in figure 3.9, the yaw difference is determined to be the distance from the projected point to the goal in the x-y plane, whereas the pitch difference is the distance in z-direction. Those terms are weighted and combined into the goal cost c_{goal} according to equation 3.7. Similarly the candidate direction is compared to the last selected direction to introduce smoothing. The smoothing cost term is also composed of a yaw difference Δ_{yaw} and a pitch difference Δ_{pitch} as stated by equation 3.8. The total cost is then determined as the weighted sum of the smoothing cost and the goal cost (equation 3.9).

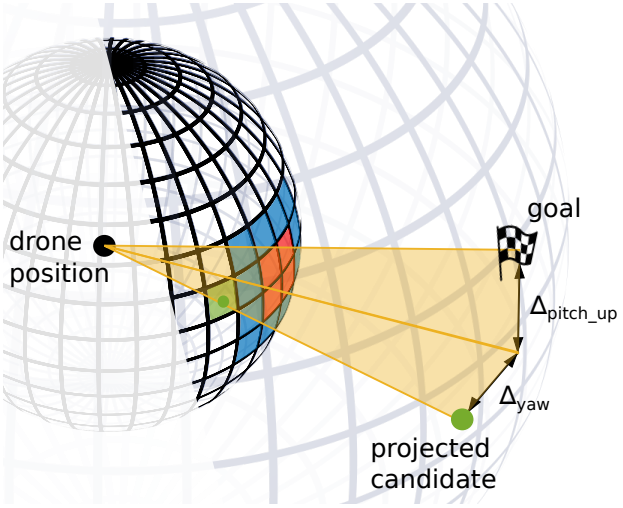


Figure 3.9: *The occupied histogram cells are marked in red and cells inside the safety margin are marked in blue. The active candidate is marked in green and projected to the same distance as the goal. The distances between the projected candidate and the goal determine the values used to calculate the goal cost of the active candidate.*

$$c_{goal} = \Delta_{yaw}(g,p) + k_{up} \cdot \Delta_{pitch\_up}(g,p) + k_{down} \cdot \Delta_{pitch\_down}(g,p) \tag{3.7}$$

$$c_{smooth} = \Delta_{yaw}(p_{old},p) + \Delta_{pitch}(p_{old},p) \tag{3.8}$$

$$c_{tot} = k_{goal} \cdot c_{goal} + k_{smooth} \cdot c_{smooth} \tag{3.9}$$

Where: g: goal position
p: projected candidate direction
p_{old}: projected old candidate direction
k: weight factors

The weight factors k_{goal} and k_{smooth} describe how the goal oriented behavior is rated in comparison to the smoothness of the path. The weight factors k_{up} and k_{down} are introduced to distinguish and rate the different avoidance paths. The factor k_{down} is set significantly higher than the other ones to discourage searching for a path underneath obstacles. The parameter k_{up} is also set high, because we favor a path around an obstacle compared to flying over the obstacle. A suggestion for the choice of cost parameters can be found in table 3.1. If the parameters k_{up} and k_{down} are chosen to be higher than one, the drone will favor flying around obstacles instead of flying over or underneath. This corresponds to the desired behavior but it has the drawback, that the drone flies endlessly along a wall (maybe back and forth) to find a gap but it never rises if there is another choice available. It is therefore beneficial to allow rising if trying to fly around the obstacle is inefficient. This suggests that the progress towards the goal should be monitored to decrease the rising cost k_{up} if there is no progress towards the goal. To track the progress rate towards the goal, the trend in distance d to the goal over time is monitored. The slope s of this curve at the time-step n can be calculated using equation 3.10. The slope values are filtered using a moving average (MA) filter with a window size of n_{MA}. A strongly negative slope at the current time-step indicates good progress towards the goal, whereas a slope with a zero or positive value means no progress. A critical value

s_{krit} for the slope is chosen as well as the boundary values on the cost parameter k_{up}. If the current filtered slope value is below s_{krit} the cost factor k_{up} is increased with a rate of Δ_{inc} per time-step until it reaches the maximum value k_{up_max}. If the current filtered slope lies over the critical value, the cost factor is reduced Δ_{dec} per time-step until it reaches the minimum value k_{up_min}. The cost parameter k_{up} is therefore calculated according to equation 3.11. In table 3.1 the parameter values used for this study are summarized.

$$s[n] = \frac{\Delta d}{\Delta t} = \frac{d[n] - d[n-1]}{t[n] - t[n-1]} \tag{3.10}$$

$$k_{up}[n] = \begin{cases} k_{up}[n-1] + \Delta_{inc}, & \text{if } s[n] < s_{krit} \text{ and } k_{up}[n-1] < k_{up\_max} \\ k_{up}[n-1] - \Delta_{dec}, & \text{if } s[n] > s_{krit} \text{ and } k_{up}[n-1] > k_{up\_min} \\ k_{up}[n-1], & \text{otherwise} \end{cases} \tag{3.11}$$

Parameter	value
k_{goal}	2.0
k_{smooth}	1.5
k_{down}	4.0
k_{up_max}	4.0
k_{up_min}	0.75
Δ_{inc}	0.3
Δ_{dec}	0.2
s_{krit}	-0.0007
n_{MA}	50

Table 3.1: *Suggestions for the choice of the cost parameters*

3.3.5 Sphere Avoidance - Minimum Distance to Obstacles

The 3DVFH algorithm is not able to guarantee a minimum distance to the obstacles. It does change direction to avoid the obstacles but it keeps moving closer unless all cells in the front half of the histogram are occupied. This behavior is even more prominent when the point-cloud is noisy and unstable. Some of the point clusters suddenly disappear and the drone moves forward because the directions are indicated as free. When the cluster reappears the drone will react and change its direction but at that point it has already come closer to the obstacle. To prevent getting too close to obstacles an additional feature has been added to the 3DVFH algorithm. A minimum distance to any obstacle r_{sphere} is specified. A larger distance r_{sphere_cloud} is chosen to determine all considered points from the point-cloud. All points lying inside the radius r_{sphere_cloud} around the UAV are used to calculate the obstacle center c_{sphere}. This center is taken to be the mean of all the considered points from the point-cloud. Around this center a sphere is drawn with the minimum distance r_{sphere} as radius. This sphere is used as an additional constraint and it cannot be entered by the UAV. If in one time-step, the point-cloud does not contain enough points to calculate the center of gravity (specified minimum number of points n_{sphere}) the old sphere will be remembered. Eventually, if the sphere exceeds the maximum age it will be discarded. The calculation of the sphere is visualized in figure 3.10.

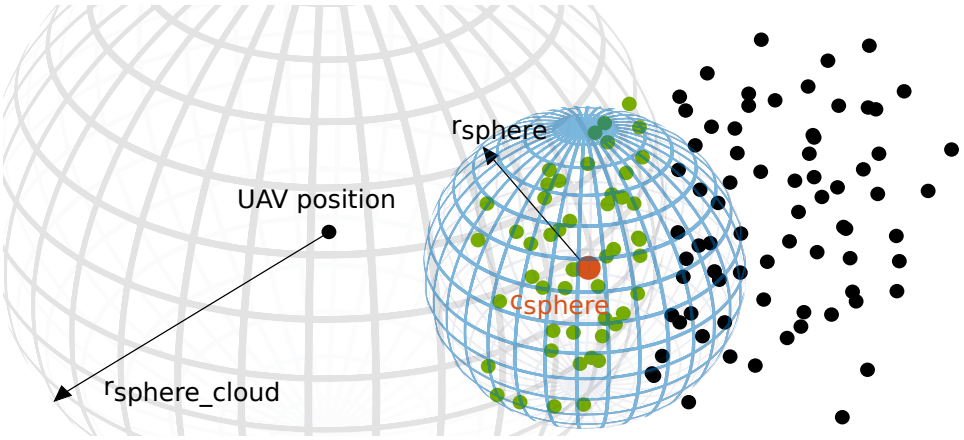


Figure 3.10: *To ensure a minimum distance to the obstacles a sphere is fitted around the closest part of the obstacle. The green points lie inside the radius to be considered as close points and their mean position is taken as the sphere center. The blue sphere cannot be entered by the UAV.*

The sphere avoidance constraint interferes with the avoidance algorithm only at the very end of the pipeline. The next waypoint wp_{orig} is calculated either from the histogram (if there are points in the point-cloud) or does lead directly towards the goal (if there is no perceived obstacle). After the desired waypoint has been calculated, the sphere avoidance checks it for validity and adjusts it if necessary. The existing sphere is enlarged by a factor k_{hyst} to implement a hysteresis in the influence of the sphere. All waypoints wp_{orig} lying inside the radius $k_{hyst} \cdot r_{sphere}$ will be modified by the sphere avoidance feature. If a waypoint lies inside the hysteresis sphere, it will first be modified in the z-component. The temporary waypoint wp_{temp} has the same x- and y-components as the original waypoint. The z-component is calculated according to equation 3.12. This shift in height causes the waypoint to lie closer to the equator. The point will later be projected to the sphere surface and points closer to the equator are more in accordance to avoid obstacles by flying around instead of over or under.

$$wp_{temp_z} = \begin{cases} wp_{orig_z} + 0.25|wp_{orig_z} - c_{sphere_z}|, & \text{if } wp_{orig_z} < c_{sphere_z} \\ wp_{orig_z} - 0.25|wp_{orig_z} - c_{sphere_z}|, & \text{if } wp_{orig_z} > c_{sphere_z} \end{cases} \tag{3.12}$$

From this shifted waypoint wp_{temp}, the sphere waypoint is calculated by projecting it onto the sphere surface. Here, two cases are distinguished: Either, the temporary waypoint lies inside the actual avoidance sphere, or it lies in the hysteresis shell. The projection radius r_{proj} will be determined according to equation 3.13.

$$r_{proj} = \begin{cases} r_{sphere}, & \text{if } |wp_{temp} - c_{sphere}| < r_{sphere} \\ |wp_{temp} - c_{sphere}|, & \text{otherwise} \end{cases} \tag{3.13}$$

The sphere waypoint wp_{sphere} is the projection of the temporary waypoint onto a sphere around the center-point with a radius of r_{proj}. The final waypoint can then be calculated according to equation 3.14.

$$wp_{final} = \begin{cases} wp_{sphere}, & \text{if } |wp_{temp} - c_{sphere}| < r_{sphere} \\ (1-f)wp_{sphere} + f \cdot wp_{orig}, & \text{if } r_{sphere} \leq |wp_{temp} - c_{sphere}| < k_{hyst} \cdot r_{sphere} \\ wp_{orig}, & \text{otherwise} \end{cases} \tag{3.14}$$

$$\text{Where: } f = (|wp_{temp} - c_{sphere}| - r_{sphere})/(k_{hyst} \cdot r_{sphere} - r_{sphere})$$

The final waypoint wp_{final} will be used as a control input. A further advantage of the sphere avoidance constraint is, that the adaption of the waypoint is also performed when there are no obstacles ahead and the original waypoint would suggest to go straight to the goal. This causes the drone to remember previously seen obstacles, even if the noise in the point-cloud temporarily causes the histogram to be empty. Table 3.2 shows some suggestions for the tuning of the sphere avoidance parameters.

r_{sphere}	2.5m
r_{sphere_cloud}	3.5 m
n_{sphere}	20
k_{hyst}	1.3
age_{sphere}	100 iterations

Table 3.2: *Suggested parameters for sphere avoidance*

3.4 Ground Detection

Ground detection is an important problem when attempting autonomous flight in unknown environments. The drone needs to be able to keep a minimum distance to the ground to avoid unintentional touch downs. One problem regarding ground detection with the given setup described in chapter 4.1 is, that the drone will never be able to see what is lying right underneath it due to the limited vertical FOV. Therefore, the actual distance to the ground is unknown. However, the ground in front of the drone is visible if the flight height is high enough. This information can be stored in an internal height map, which can be used once the drone is above the previously seen ground area. This enables the drone to keep a certain distance to the ground and even to look ahead and increase its height beforehand if moving towards a higher ground.

3.4.1 Building an Internal Height Map

To detect the ground, the point-cloud is cropped with respect to a second bounding box. A minimum flight height h_{min} is specified and the ground box is designed such that the ground will be detected before the drone reaches the minimum flight height. The design of the ground box is shown in figure 3.11. The angle γ specifies the vertical FOV of the camera. The box is designed around the drone, such that the drone is always located in the middle of the upper surface. The dimensions of the box are calculated according to the equation 3.15. Using those dimensions for the bounding box, the ground can be well seen when it lies at the specified minimum height.

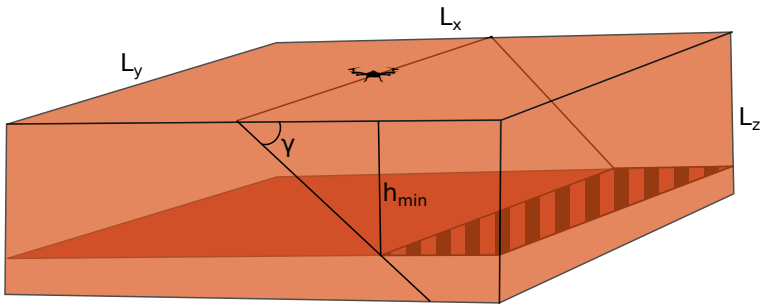


Figure 3.11: *Dimensions of the ground box used to crop the point-cloud.*

$$\begin{aligned} L_z &= 1.5 \cdot h_{min} \\ L_x &= 2 \cdot \frac{L_z}{tan(\gamma)} \\ L_y &= 2 \cdot \frac{L_z}{tan(\gamma)} \end{aligned} \tag{3.15}$$

The cropped point-cloud is fed into a RANSAC algorithm, which estimates a horizontal plane from the data. The RANSAC algorithm requires an angle difference prior $\Delta\phi$, which defines the maximal deviation of the plane model from the horizontal, and the maximum inlier distance d_{inlier_max} to the plane. The resulting plane is assumed to be valid if the cropped point-cloud is larger than a minimum cloud size n_{cloud_min} and a specified minimum fraction p_{cloud_min} of the cropped cloud are inlier to the fitted plane model. A suggestion for the choice of the ground detection parameters can be found in table 3.3.
The RANSAC algorithm returns the plane coefficients $[a, b, c, d]$ of the fitted plane model (3.16) if the calculated plane is valid. From those plane coefficients, the closest point p_{plane} to the drone on the plane and the perpendicular distance d_{plane} of the plane to the drone can be calculated using the equations 3.17, 3.18.

$$ax + by + cz + d = 0 \tag{3.16}$$

$$d_{plane} = -\frac{ap_x + bp_y + cp_z + d}{a^2 + b^2 + c^2} \tag{3.17}$$

$$p_{plane} = [p_x + d_{plane}a, p_y + d_{plane}b, p_z + d_{plane}c]^T \tag{3.18}$$

Parameter	value
n_{cloud_min}	160
p_{cloud_min}	0.7
$\Delta\phi$	20°
d_{inlier_max}	0.1

Table 3.3: *Suggestions for the choice of the ground detection parameters*

A rectangular patch is estimated from the inlier by searching for the maximum and minimum x- and y-positions in the inlier cloud. The boundary coordinates $[x_{min}, x_{max}, y_{min}, y_{max}]$ are saved together with the height of the patch. The height of the patch can be calculated from the height of the drone and the distance d_{plane}

to the patch. From all those saved patches, a height memory of the environment is built. When a new patch is found, first the height memory is searched for a patch with similar height. If it overlaps with the new patch, they are assumed to be the same surface and are saved as one entry with bounding values encompassing both patches. If no overlapping patch with similar height is found, the new data is saved as an individual patch. This knowledge about the height of the terrain can either be used as an individual controller in the case where there are no additional obstacles ahead, or it can be combined with the histogram approach to control height and obstacle avoidance simultaneously. In the following paragraphs both use cases are discussed.

3.4.2 Height Control without Additional Obstacles

When there are no obstacles in sight (the polar histogram has no occupied cells) the drone will move directly towards the goal. The direction from the drone to the goal is computed and scaled to fit a specified velocity. This target vector is then used to generate the next waypoint. To avoid collisions with the ground, the knowledge from the internal height map can be used to determine the correct flight altitude.

When the drone is approaching a ground patch stored in the internal height map, it has to start rising early enough to reach the required height once it is over the patch. The maximum rising angle e_{max} of the drone is set to half the vertical FOV, which for the Realsense is 24°. If rising at a steeper angle, a collision-free path cannot be guaranteed. Therefore, the drone needs to start adjusting its height as soon as it enters a margin m in front of a patch. This margin is dependent on the minimum distance to ground h_{min} , the difference of height between the UAV and the ground patch as well as on the rise angle. It can be calculated using the equation 3.19.

$$m = \frac{p_z - h_{patch} - h_{min}}{tan(e_{max})} \tag{3.19}$$

Once the drone has entered an area of a patch enlarged by the margin m in flight direction, the margin is set to be fixed and the drone is considered to be above the concerned ground patch. A continuously adapting margin would lead to oscillations, as the margin decreases as the drone height increases. When the drone is over an obstacle, the minimum flight height is calculated by adding the specified minimum height over ground h_{min} to the obstacle height h_{patch}. The drone altitude is forced to the required value by assigning one of the following three states:

1. drone height is too low
2. drone height is high enough, but very close to minimum flight height
3. drone height is high enough and not close to minimum flight height

In case 1 the new waypoint is set in the x-y direction of the goal, but such that the drone will be rising at an angle e_{max}. In case 2 the drone is not allowed to sink any further. Therefore, the suggested waypoint towards the goal is only modified if it would lead the drone downwards. In case 3. the original waypoint directly towards the goal is kept. To avoid oscillations, hysteresis between all three cases are implemented.

3.4.3 Height Control in Combination with Histogram

Analog to the obstacle free case, the first step is to determine whether the drone is over one of the patches. The calculation of the margin is performed as described

in equation 3.19 but considering the discretization of the histogram in the desired rise angle. The same three cases are distinguished. In the first case, where the drone is flying too low, all histogram cells which lie at an elevation angle lower than e_{max} are blocked such that they are no longer valid candidates. This means the drone has no other possibility than to rise at an angle e_{max} in a direction where the corresponding histogram cell is free. In the second case, all candidate cells at an elevation angle lower then zero are discarded such that the drone cannot choose to drop in altitude. For the third case, the candidates are determined from the histogram without interference of the height controller.

3.5 Look Ahead: 3DVFH* with memory

The VFH* algorithm was introduced by Ulrich and Borenstein [3] for navigation of a ground robot in two dimensions. It allows to look ahead into the future and evaluate possible movements not only for the current time-step but also for the subsequent ones. This algorithm is a combination of the polar histogram approach and an A* search algorithm. The basic principle is to repeat the histogram step at all possible future positions of the vehicle in order to build a search tree. This tree can be searched using the A* algorithm to find the best possible path. The present implementation is an extension of the VFH* algorithm to a 3D environment.

3.5.1 Algorithm Structure

The structure of the 3DVFH* algorithm is depicted in figure 3.12. The algorithm includes all the additional features discussed in chapter 3.3 as well as the memory. The 3DVFH* algorithm also starts using the new complete point-cloud from the camera. Next it will update the cost parameters according to the progress made. As discussed in chapter 3.3.4, this enables the drone to rise if no way can be found around the obstacle. Then the old histogram is used to re-project the occupied cells into the 3D space and generate a set of 3D points. The first histogram is calculated as a combination of the re-projected points and the data from the new point-cloud. This histogram is saved as the old histogram for the next time-step. In the 3DVFH algorithm we would now calculate the next waypoint from this histogram only. But the 3DVFH* algorithm builds a search tree to propose different movement possibilities. This tree has its root at the current UAV position. This position is entered in the tree as the first origin and the tree cost function as well as the heuristic for the first node are set. From there, new nodes are entered into the tree structure until a specified number N of nodes has been expanded.

To expand a node, the complete point cloud is first cropped around the current node position. The points of the cropped point-cloud will be checked for their distance to the drone same as for the *Back-off* feature discussed in chapter 3.3.2. As the current node is just a proposal of a possible movement, it does not require a reaction if it is too close to an obstacle. But if the node is too close to an obstacle, its cost is set to infinity. Instead of expanding it, the tree will be searched for the next best node. To expand a node, the FOV at its position is calculated. The histogram is built as a combination of the histogram of the cropped point-cloud and the histogram of the re-projected points. To combine those two histograms the FOV of the drone at the node position is needed. The histogram is down-sampled to half the resolution to reduce the number of candidate directions and make them more distinct. The candidate directions are extracted and evaluated according to the same cost function as used for the 3DVFH algorithm. To reduce the branching factor of the search tree not all candidates are added as nodes. They are added best-first in the order

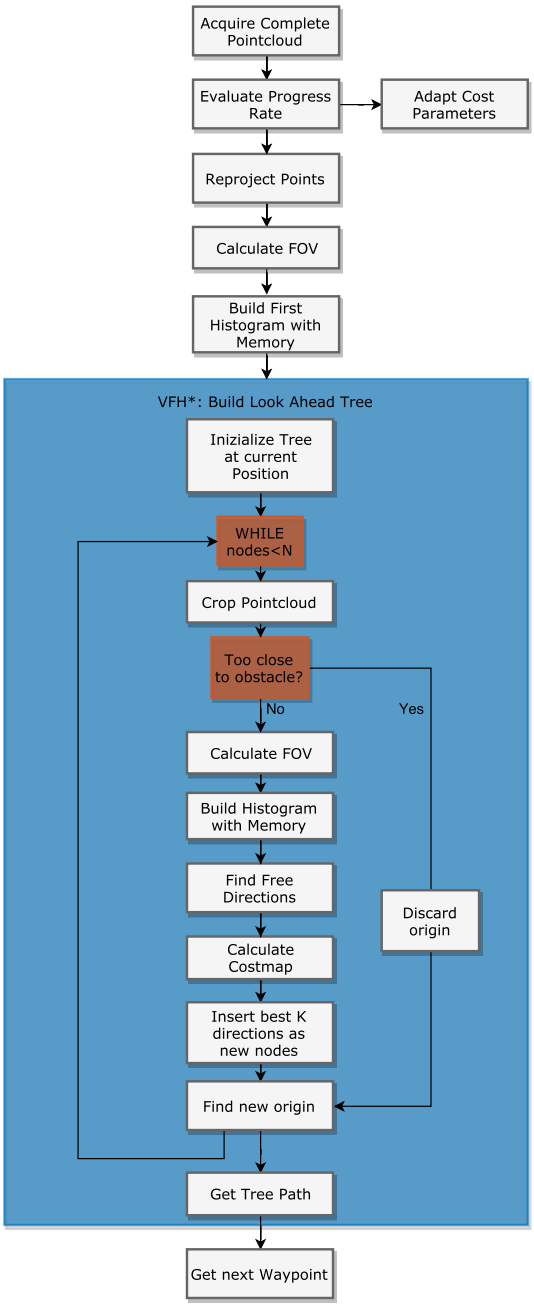


Figure 3.12: *Structure of the VFH* part of the algorithm. Building the search tree is achieved by repeating the histogram approach for every proposed future position of the drone.*

of their cost. A candidate is only added if from the same origin no close node was added before. The maximum branching factor can be specified by the parameter b_{max}. Every added direction produces a new node at a distance of L_{nodes} from the current origin. The tree cost function and heuristic function are evaluated for the new nodes to assign the total cost to every node. Then the whole tree is searched for the node with the least total cost. The node with the lowest cost is expanded next if the limit N of expanded nodes is not yet reached. If N nodes are expanded,

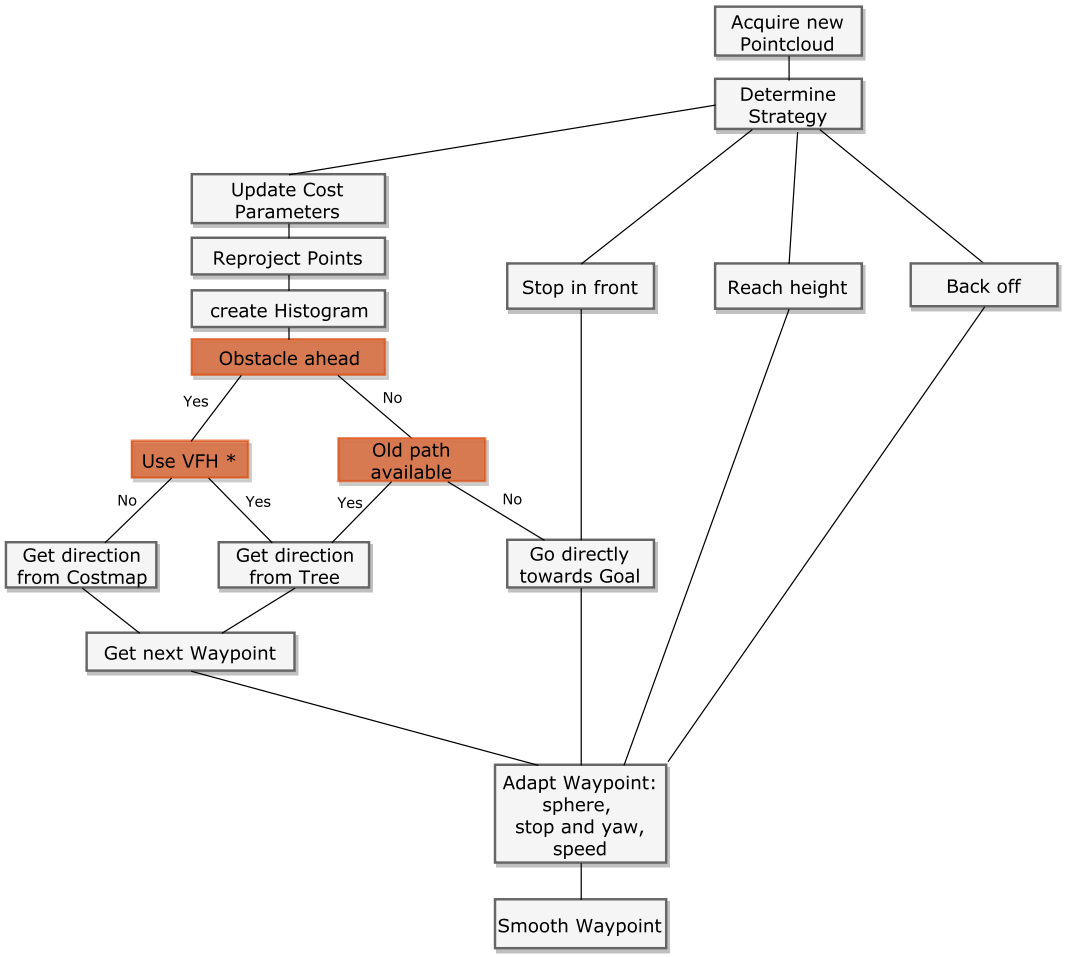


Figure 3.13: *Logic of the whole algorithm including all the discussed features.*

the node with the least cost corresponds to the end of the tree path. The best path through the tree will be back-traced from the end node to find the best direction for movement.

The architecture of the whole avoidance algorithm with all the features is shown in figure 3.13. The logic of the algorithm from the acquisition of the point-cloud to the output of the next waypoint is displayed. Note that if there is no obstacle ahead in the current time-step the previously calculated path is reused. Whether the previously calculated path is still valid is determined from the age of the path as well as the current drone location compared to the suggested tree path. If the drone seems to be on a previously calculated tree path, the path will be followed to the end instead of going directly towards the goal. In cases where the point-cloud contains a lot of noise and might even be empty for some time-steps, the drone will stay on the calculated path and not swerve towards the goal. The choice of the parameters is mostly a compromise between computation time and accuracy of the calculation. The parameters need to be tuned on the specific hardware to ensure valid computation times.

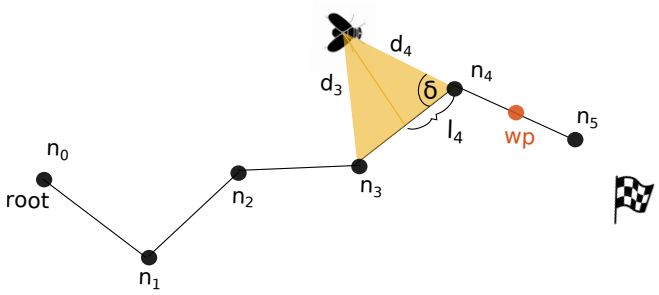


Figure 3.14: *Extraction of the next waypoint from an previously constructed tree path.*

3.5.2 Reuse Old Path

The path calculated from a search tree in a previous time-step can be reused if it suffices certain criteria. If the path has expired its maximum age, it will be discarded. Otherwise the distances to the closest two nodes of the path are calculated. If the closest node is equal to the end of the path, the path is discarded as its end is reached. Similar, if the drone is too far from the closest node. A visualization of the calculation for following a previously computed path is shown in figure 3.14. If the old path can be reused, the angle δ to the next node n_i can be calculated using equation 3.20. In the schematic the node n_i corresponds to the node 4, which is the node lying further along the path from the two closest ones. The length L_{nodes} corresponds to the distance between every two nodes. The distance l_i can be calculate from the angle δ according to equation 3.21. From this length the fraction p of the path, which the drone has traveled between node i and $i-1$ can be calculated. Using this value the new waypoint can be calculated from the node positions i and $i+1$ as described by equation 3.23. For $p=0$ the drone is very close to node $i-1$ and its new way point will be node i, whereas for $p=1$ the drone is very close to node i and its waypoint will be node $i+1$. Those calculations ensure a smooth movement when following a precomputed path.

$$cos(\delta) = \frac{L_{nodes}^2 + d_i^2 - d_{i-1}^2}{2L_{nodes}d_i} \tag{3.20}$$

$$l_i = L_{nodes}cos(\delta) \tag{3.21}$$

$$p = \frac{l_i}{L_{nodes}} \tag{3.22}$$

$$wp = (1-p)n_i + pn_{i+1} \tag{3.23}$$

3.5.3 Cost and Heuristic Functions

The A* search requires a cost function and a heuristic function to rate every node. Both of those functions are designed to use angular differences instead of comparing positions. The cost function is composed of four different cost terms: target cost, yaw cost, path smoothing cost and tree smoothing cost. The target cost describes if the direction of the node is into the direction of the goal. In other words it is calculated from the angular difference between the tree branch to the current node and the direction towards the goal. The yaw cost is determined as the angular distance between the current yaw of the drone and the the tree branch to the node. The path smooth cost forces the paths inside the tree to be smooth. It is calculated from the angular distance between the current tree branch and the tree

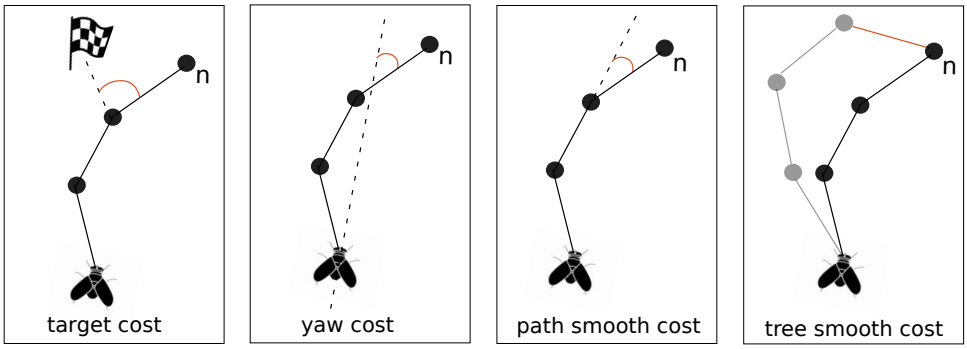


Figure 3.15: *Visualization of the different cost terms. The angle or distance marked in red determines the value of the corresponding cost.*

branch to the origin of the node. The last cost term is not an angular term but a comparison of trees from different time-steps. It compares the current node with the node at the same depth in the last chosen path. It needs to be scaled in a way that the magnitude of this term fits to the angular costs. The different cost terms are visualized in figure 3.15. The different cost terms are weighed with factors k to determine the importance of each criterion. The cost function is written in equation 3.24. The sum of the individual weighted cost terms is subject to scaling. The scaling factor is dependent on the depth of the node. Nodes at higher depth are less penalized by cost. The discount factor λ needs to be less than one. This depth dependent scaling is necessary to avoid trajectories which stop shortly before an obstacle instead of searching for a way.

$$c_n = \lambda^{depth_n} \cdot (k_{target}c_{target} + k_{yaw}c_{yaw} + k_{path}c_{path} + k_{tree}c_{tree}) \tag{3.24}$$

The A* algorithm also demands a heuristic function which estimates the cost of the cheapest path from a node n to the goal. For a function to be admissible as a heuristic, it must never overestimate the cost to reach the goal. For simplicity we use the target cost of the current node as a heuristic.

Chapter 4

Results

4.1 Experimental Setup

For this project the Robot Operating System (ROS) is used as a framework [39]. ROS provides libraries and tools to design and coordinate robot software and is completely open-source. In ROS different processes are coordinated by running them in different nodes. ROS provides an interface for message passing between different nodes as well as a visualization tool called RViz. The drone is operated using the open-source PX4 framework [40]. Our setup consists of the following ROS nodes:

- Camera Node
- Stereo Processing Node (in simulation only)
- UAV Node (Mavros)
- Local Planner Node

The camera node runs the camera driver and acquires the images from the camera. In simulation, a stereo processing node then generates a depth map from the camera images. The physical camera does this computations internally and directly outputs the depth map. The Mavros node is a communication driver for autopilots. It acquires information, such as state information, from the PX4 autopilot using the MAVlink protocol. The local planner node runs the whole planner algorithm and publishes the calculated setpoints. Those are then sent to the autopilot over the Mavros node.

4.1.1 Simulation Setup

The simulation tests are done using the open-source 3D simulation environment Gazebo. Gazebo provides a robust physics engine which is able to detect collisions between the robot and the environment. It also allows the use of sensor models, which provide the necessary sensor data. Gazebo is compatible with ROS, which means that for example the artificial sensor data can be published as a ROS topic. This means that from the view point of the local planner node, everything stays the same, whether the algorithm is used with simulated sensor data or data coming from a real camera. To simulate the UAV we use a model of the 3DR Iris Quadcopter [41]. This drone does not have an integrated camera, therefore we use a model of a stereo pair which is attached to the drone and provides the necessary simulated images. An additional stereo processing node is used to convert the stereo images into the required depth map.

4.1.2 Real World Platform

For the outdoor tests we use the Intel® Aero Ready to Fly drone [42] shown in figure 4.1. The Intel® Aero drone weights 865 g without the battery and has a maximum speed of $15\,\mathrm{m\,s^{-1}}$. It contains a compute board with an Intel® Atom™ x7-Z8750 processor on which the proposed algorithm will be running. The drone is also equipped with an Intel® Realsense™ (R200) camera [43]. The Realsense camera internally calculates a depth map from the stereo images and publishes this map as a 3D point-cloud with a frequency of approximately 28 Hz. It has a horizontal FOV of 59° and a vertical FOV of 46°.

4.1.3 Local Planner Node

The local planner node runs the local planner algorithm and handles the communication to the other nodes. It receives vehicle state information from the Mavros node and the 3D point-cloud from the camera node. The planner algorithm then calculates the next setpoints and publishes them such that the Mavros node can send them to the PX4 autopilot. During the obstacle avoidance tests the drone is in off-board mode. This means that the autopilot directly uses the sent setpoints to navigate. The setpoints have to be published at a minimum rate of 2 Hz for the autopilot to stay in off-board mode. Depending on the size of the search tree in the 3DVFH* algorithm it is possible that the calculation of the planner is not fast enough to meet this rate criterion. Also, it has been observed that occasionally the Realsense camera fails to publish the point-clouds fast enough. If the autopilot receives setpoints too low in rate an error message appears and the drone switches back to position mode. To avoid this situation it is assured that the planner always sends setpoints. This is achieved by running two different threads in the local planner node. One thread runs the planner algorithm and the other thread handles the communication with the other nodes and keeps track of the timing. The planner algorithm is executed whenever a new point-cloud is provided. The communication thread runs with a rate of 10 Hz to be sure to meet the minimum setpoint rate. If time since the last published setpoint exceeds a specified value, this thread will provide alternative setpoints until the planner is ready. Figure 4.2 shows the basic tasks of the communication thread. It first receives the messages from the

Figure 4.1: *The Intel® Aero Ready to Fly drone used for outdoor testing*

other nodes and stores the small messages locally until it can update them in the planner. As the point-cloud requires a lot of memory, it is copied directly to the planner as soon as possible instead of storing it locally. To avoid race conditions, it is important that the planner information is only updated when the algorithm is not running. After the incoming messages are handled, the communication thread determines if it is necessary to provide an alternative setpoint. In our setup the variable *t_max* is set to 0.25 s. If the planner takes longer to provide the next setpoint, the communication thread provides an interim setpoint using data from the last planner cycle if possible. If the 3DVFH* is running, the previously planned path is used to determine the interim waypoint. If the 3DVFH algorithm is used, the last calculated setpoint is repeated. If there is no information available, e.g. in the before the planner has ever run, the current position is fed as a setpoint. If the planner cannot provide a new setpoint for a time of *t_land* (in this implementation 15 s) then the drone will go into landing mode.

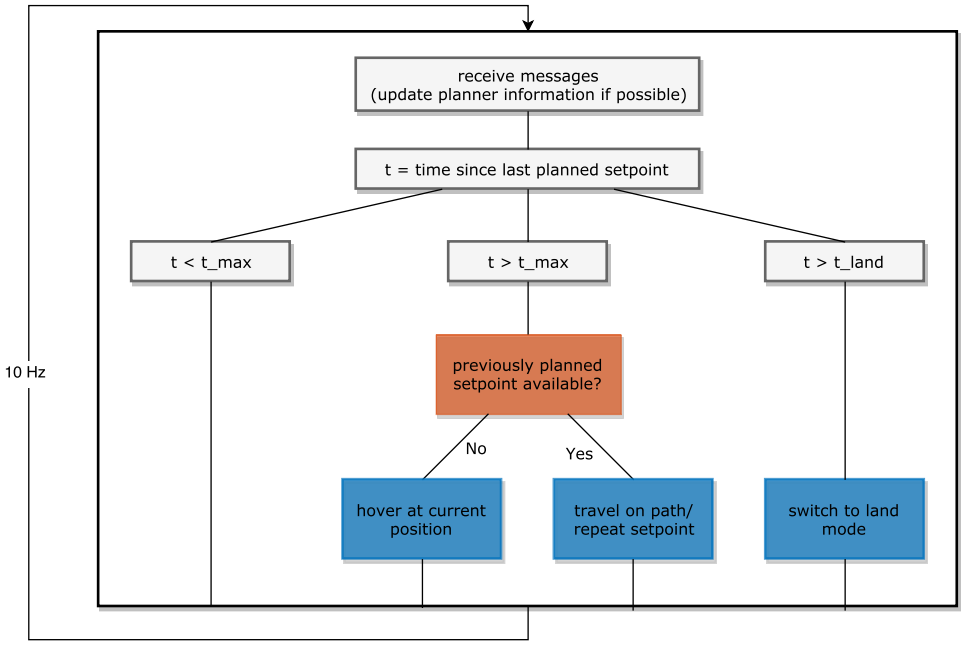


Figure 4.2: *Work-flow of the communication thread: It receives messages from the other nodes and updates the planner thread. If the planner does not produce setpoints at the required rate, this thread will provide setpoints from previously calculated cycles or from estimation data.*

4.2 3DVFH with Memory

The benefit of the memory addition to the 3DVFH algorithm has been tested using the simulation setup. Figure 4.3 shows the simulation environment. For visualization purposes, simple rectangular obstacles have been chosen. Those obstacles are high enough, such that the drone will search its way around the obstacles instead of surpassing them. The drone is started at the marked position and will move to five randomly chosen goal positions. For each goal, five runs have been performed using the algorithm with memory and five runs without the memory. The back-off safety measure described in chapter 3.3.2 was enabled to avoid collisions as much as possible. The sphere avoidance as well as the ground detection were disabled. Also the standard 3DVFH was used without any look-ahead functionality.

Figure 4.4 shows the mean travel times to each goal. For all goal locations the algorithm with memory was faster to reach the goal. The main reason for this finding is, that the original algorithm often gets stuck in locations where passing left or right seems equally good from the current visual information. As soon as the drone turns in one direction, the parts of the obstacle on the other side disappear from the FOV, which in turn makes the other direction more attractive. As the drone is only allowed to move in directions inside the FOV, it will stay in one place while turning. Due to the lacking memory, the algorithm has no knowledge about what it has seen before and it will hover on the spot and turn back and forth. Therefore, this indecisive behavior increases the time to goal. The algorithm with memory however will remember the previously seen parts of the obstacle and keep going into the decided direction. Only if the obstacle is very large in that direction, the drone might turn back to find another way.

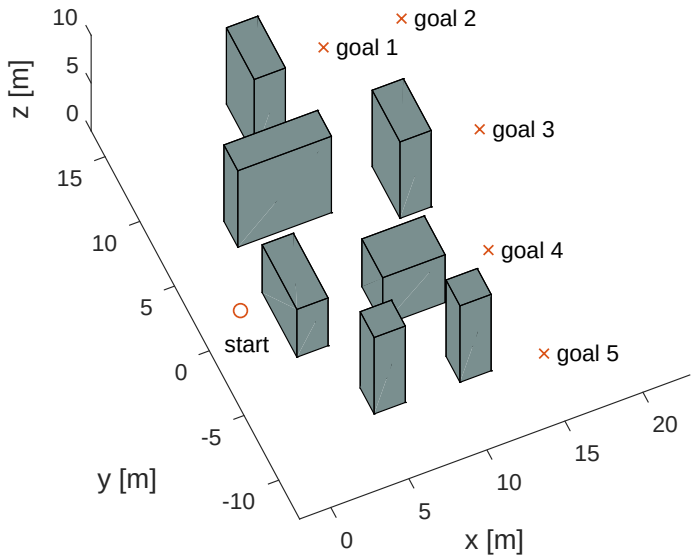


Figure 4.3: *Simulated world with rectangular obstacles. The drone will always start in the same location and move to five randomly chosen goals.*

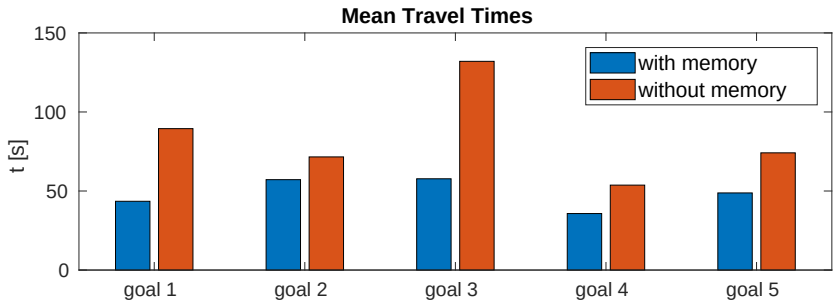


Figure 4.4: *Mean travel times for each goal location. the times for the original algorithm are shown in red, whereas the times for the memory algorithm are shown in blue.*

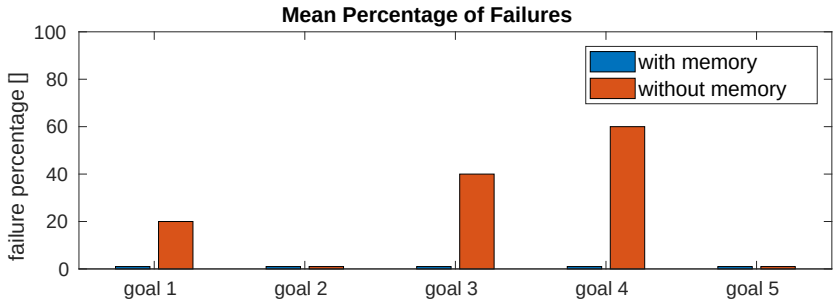


Figure 4.5: *Percentage of failure for each of the goal locations. The memory algorithm never failed to reach the goal, whereas the original algorithm failed in 24% of all runs.*

The statistic in figure 4.5 show the percentage of failed runs for each of the goals. The algorithm with the memory never failed to reach the goal. However, the original algorithm failed in 24% of all the runs. The failure cases were mainly due to position drift when the drone was stuck in an indecisive behavior. Either the drone lost height and landed on the ground, or it drifted upwards and would have surpassed the obstacle. Both of those cases were treated as failure.

Figure 4.6 shows how often the back-off safety measure was needed to avoid collisions. Those numbers can be understood as the amount of failures that would have happened if the measure was deactivated. It is obvious, that the original algorithm largely depends on this safety feature. The main reason for this dependency lies in the same issue as discussed before with the travel times and failures. When the drone cannot decide which direction to go and yaws a lot, it is prone to the failure case shown in figure 3.7. The more the drone changes direction without making any real progress, the closer it comes to the obstacle. The back-off safety measure prevents crashes for this exact failure case.

Figure 4.7 shows the trajectories of the drone towards goal 1. The memory algorithm manages to travel on a smooth path towards the goal. The original algorithm on the other hand gets stuck at a position where an obstacle is directly between the drone location and the goal. When the drone yaws to change direction, the obstacles points closest to the drone come into view and the drone has to back-off to maintain a safe distance. Due to the continuous yawing motion, also the estimation of the drone drifted more then usually. This is the reason why the trajectories of the

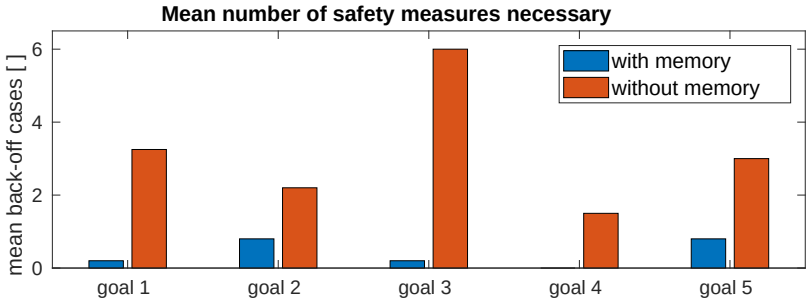


Figure 4.6: *Number of situations, where the back-off safety measure was needed to prevent collisions with the obstacles.*

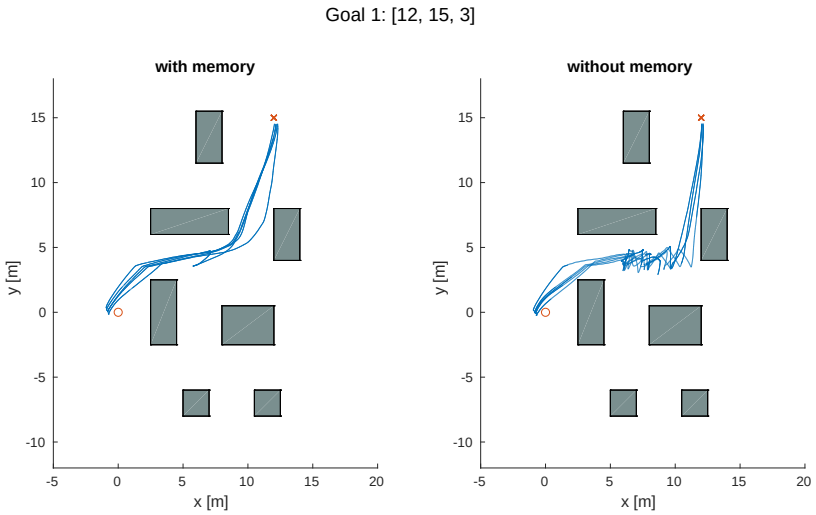


Figure 4.7: *Trajectories of the drone flying towards goal 1. The left figure shows the paths taken by the 3DVFH with memory, whereas the right figure shows the trajectories of the 3DVFH algorithm without memory.*

original algorithm pass the last gap farther to the right. This difference is made up purely by position drift and cannot be observed when looking at the real relative position of the drone to the obstacle. The trajectories for all the other goals can be found in the appendix A.1.1.

4.3 Ground Detection

4.3.1 Simulation Results

The obstacle avoidance with ground detection is tested in a simulated world. The world is designed such that both application cases discussed in chapter 3.4 are tested. The highest obstacle on top of the first terrain step is detected as an obstacle by the VFH algorithm, which causes occupied cells in the polar histogram. Around this obstacle, the ground detection will therefore be combined with the histogram approach. The rest of the terrain does not contain any obstacles, which will be seen by the VFH algorithm. This allows the ground detection to act without any interference from the VFH algorithm. The following figures show the flight course of the simulated drone from different perspectives. In figures 4.8, 4.9 the memory 3DVFH is used without ground detection, whereas in figures 4.10, 4.11 the ground detection enhances the algorithm. The specified minimum height h_{min} is shown as a red line. In the first run without ground detection, the drone flew too low in several occasions. Once it even hit the edge of the second terrain step while decreasing height. It is apparent that the use of ground detection helps in keeping the requested minimum flight height. The internal height map constructed for the height control is visualized in figures 4.12, 4.13. The patches fit quite well with the real ground topography.

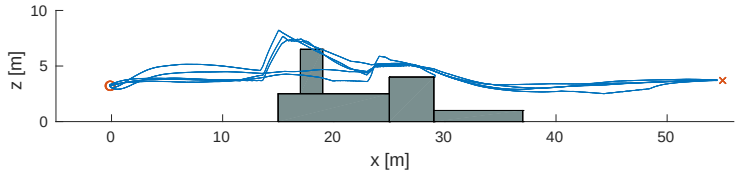


Figure 4.8: *Obstacle avoidance without ground detection. Seen in the x,z-plane*

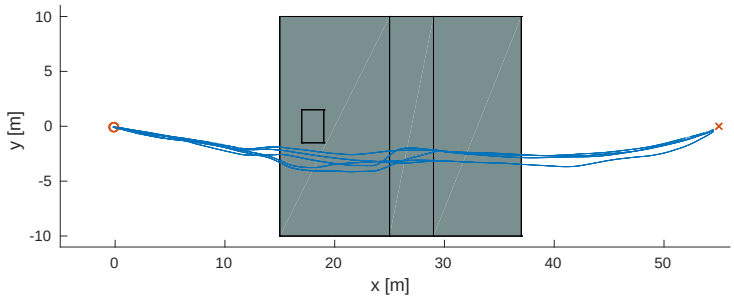


Figure 4.9: *Obstacle avoidance without ground detection. Seen in the x,y-plane*

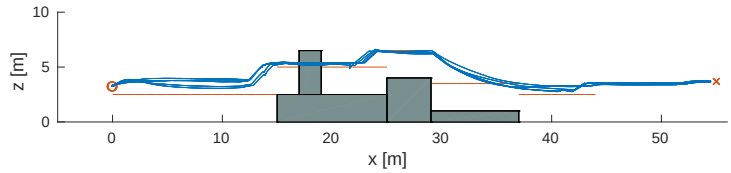


Figure 4.10: *Obstacle avoidance using ground detection. Seen in the x,z-plane*

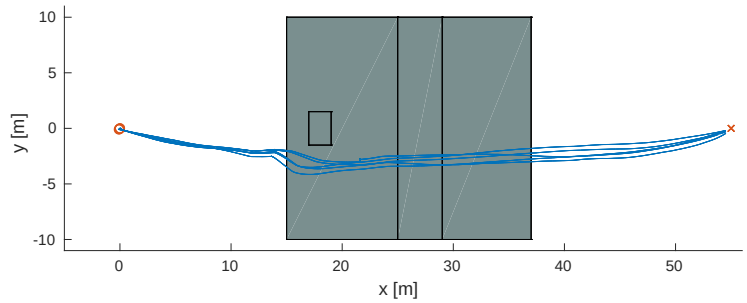


Figure 4.11: *Obstacle avoidance with ground detection. Seen in the x,y-plane*

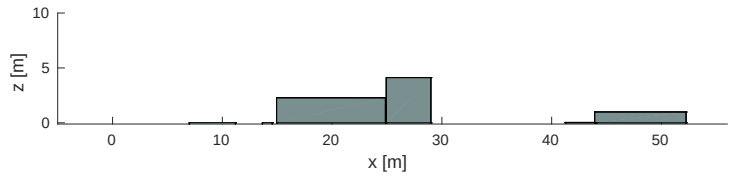


Figure 4.12: *Ground as estimated by the drone. Visualization of the patch memory in the x,z-plane*

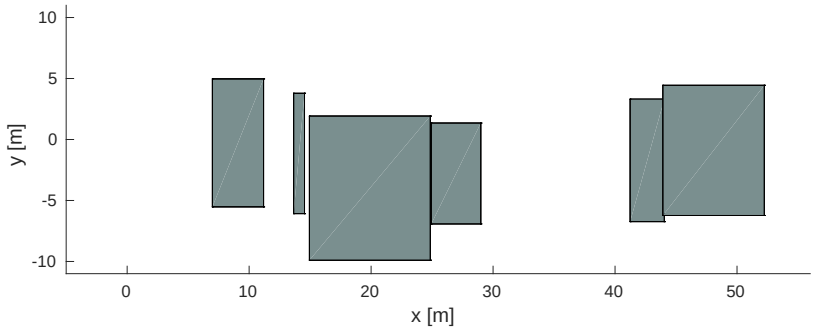


Figure 4.13: *Ground as estimated by the drone. Visualization of the patch memory in the x,y-plane*

4.3.2 Real -World Ground Map

Handheld tests have been performed to evaluate the quality of the internal ground map. The drone was carried over an environment to see if the drone catches the most important ground features. Figure 4.14 shows the internal map built while traveling up the flat concrete stair shown in figure 4.15. The internal ground map shows an approximation of the ground which is not entirely accurate. However, this representation of the ground would be able to provide correct height constraints as to limit the minimum flight height of the drone. The map does not contain any holes, where the drone could come too close to the ground. Issues would rather lie on the opposite side, such that the drone maps obstacles as ground. In this case the drone would over fly the object instead of searching for a way around. This behavior is not optimal, however it does not lead to collisions as the top surface of the obstacle is known.

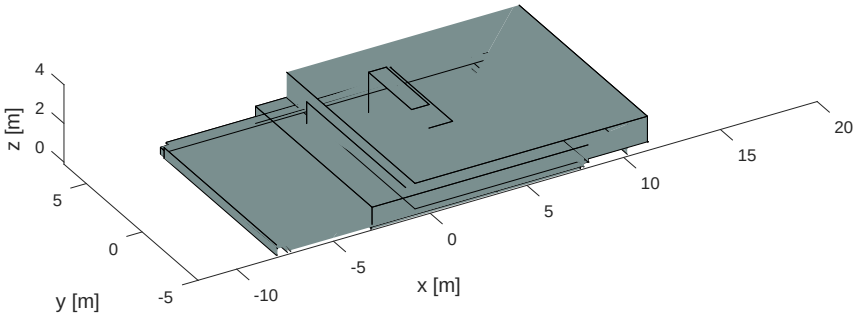
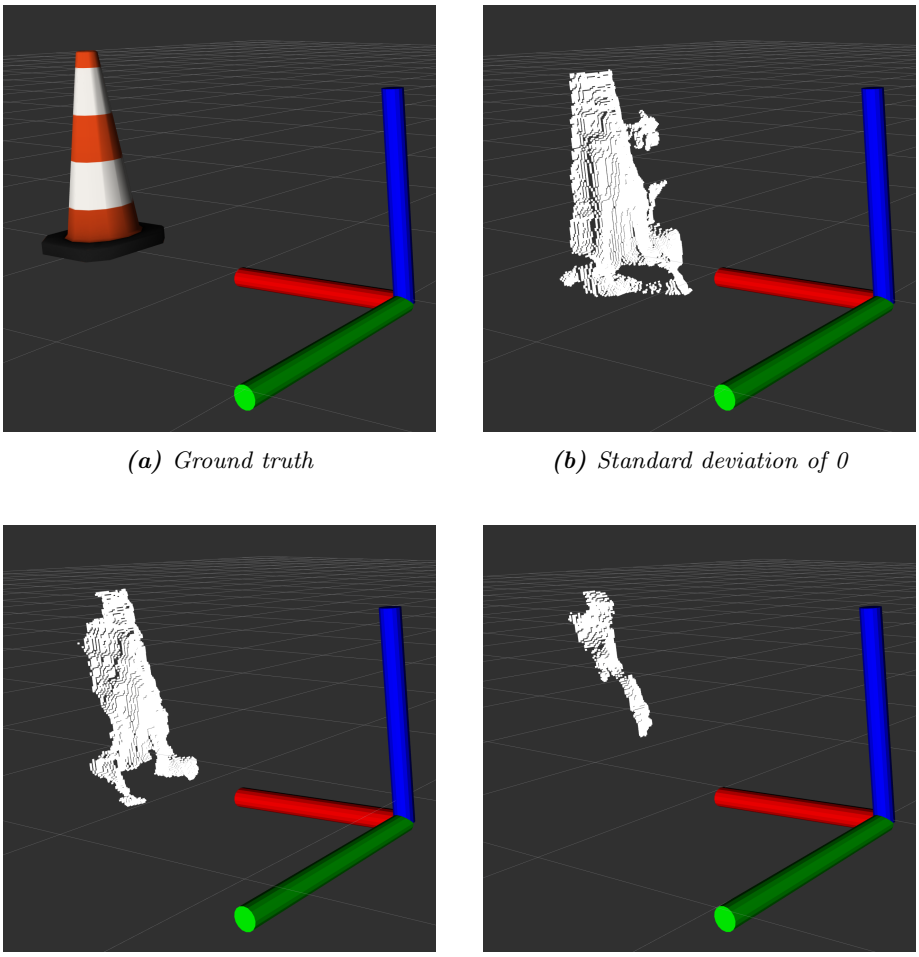


Figure 4.14: *Ground as estimated by the drone while being carried up a concrete stair.*

Figure 4.15: *Ground truth of the terrain which was used to build the internal height map.*

4.4 Sphere Avoidance

In simulation we have the access to perfect sensor data, whereas in reality sensor outputs are always subjected to noise. In figure 4.16 simulated point-clouds for different noise standard deviations are shown. As we also use a simulated stereo camera even the point-cloud without noise will be imperfect due to stereo processing errors. With increasing noise standard deviation, large parts of the obstacles disappear from the point-cloud. The point-cloud also gets increasingly unstable and differs a lot between adjacent time-steps. To show the benefit of the sphere avoidance in these cases simulation tests were performed in an environment with highly irregular obstacles to imitate real world trees (see figure 4.17) . For each strategy (with and without sphere avoidance) 15 runs were performed to the same randomized goals. This test was repeated for different standard deviations. The results of those simulations are summarized in figure 4.18. It can be concluded, that with increasing standard deviation of the noise, the strategy without sphere fails more and more often. The sphere avoidance on the other hand seems to be able to cope with the increasingly noisy point-cloud.



(a) *Ground truth*

(b) *Standard deviation of 0*

(c) *Standard deviation of 0.01*

(d) *Standard deviation of 0.03*

Figure 4.16: *Point-cloud of an exemplary object using the stereo camera and different noise standard deviation values. The coordinate system marks the position of the drone, where the stereo camera points into the red x-axis.*

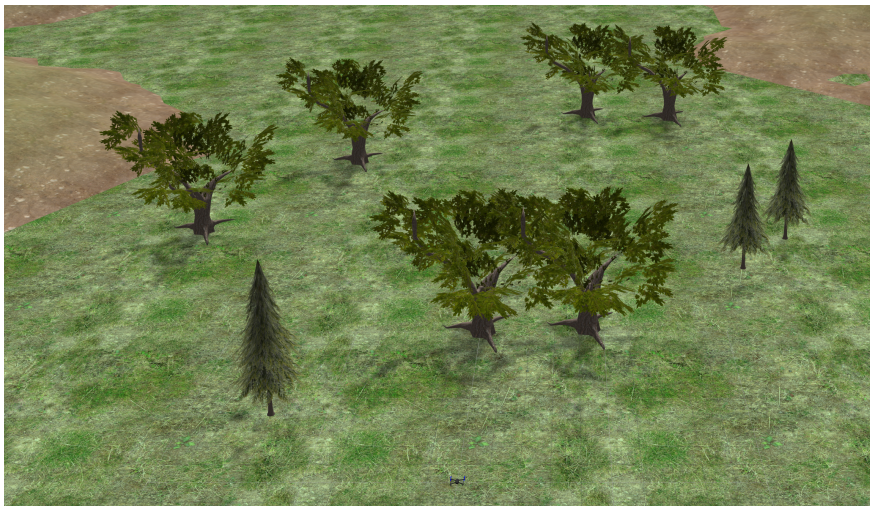


Figure 4.17: *Gazebo visualization of the testing environment. The oak trees have relatively sparse branches which makes the point-cloud highly irregular.*

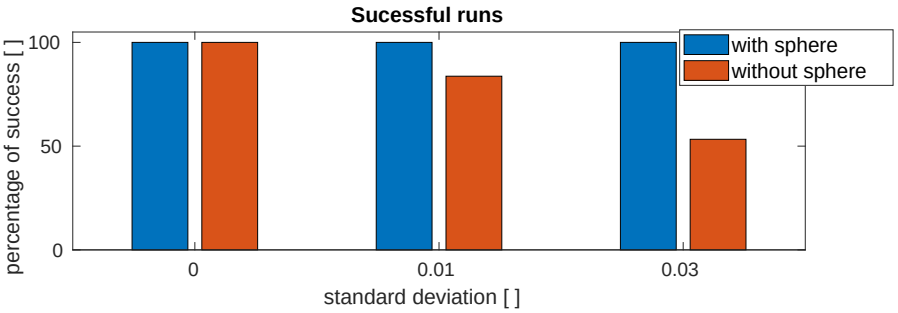


Figure 4.18: *Success rate of the different algorithms in the simulated tree world. For each standard deviation value* 15 *runs were performed with sphere avoidance and* 15 *without. The goal were chosen randomly but for both strategies the same goals were used.*

4.5 3DVFH* with memory

4.5.1 Simulation Results

To evaluate the advantages of the 3DVFH* algorithm over the 3DVFH algorithm both have been tested in simulation in the same environment. The simulated world consists of an L-structure, where the differences between the algorithm become apparent. For both strategies 10 runs have been performed. The trajectories of the different algorithms can be seen in figure 4.19. The 3DVFH chooses to avoid the obstacle to the right and therefore takes the longer path. This is due to the fact, that the planner only sees the narrow part of the obstacle for which passing on the right side is more efficient. It does only see broad part of the obstacle when is has already traveled far enough that a change of direction would even lead to a longer way to the goal. The 3DVFH* on the other hand builds a look-ahead tree where the complete point-cloud is re-cropped at the position of every node. The nodes on the right side therefore consider also the broad part of the obstacle and the branches would have to grow around it. They therefore have a higher cost than

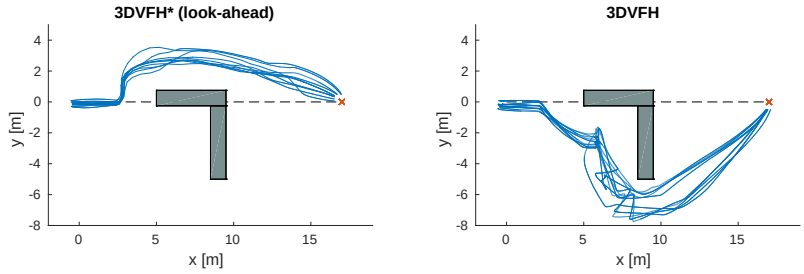


Figure 4.19: *Trajectories of the 3DVFH* and the 3DVFH algorithm in avoiding an L-shaped obstacle. The dashed line indicates the direct path to the goal which is marked with a red cross.*

the branches to the left side. As a consequence the cheapest path takes the drone to the left side of the obstacle. In figure 4.21 the decision process of the 3DVFH* is visualized. Taking a closer look at the shape of the trajectories, it can already be seen in figure 4.19 that the 3DVFH algorithm struggles with the concave obstacle. The back-off safety measure was invoked once in 50% of the runs. Once the broad part of the obstacle comes in sight the drone tries to change direction. Only after it has turned to the left and the wall of the narrow part becomes visible, it continues to pass the obstacle to the right. The 3DVFH algorithm therefore looses time not only by the inefficient path choice (passing the obstacle to the right) but also by indecision. To quantify the difference in smoothness of the trajectories, the jerk has been calculated. The magnitude of jerk is depicted in figure 4.20. It becomes apparent that the 3DVFH* produces smoother trajectories. The magnitude of jerk between the two strategies differs by a factor of more than two.

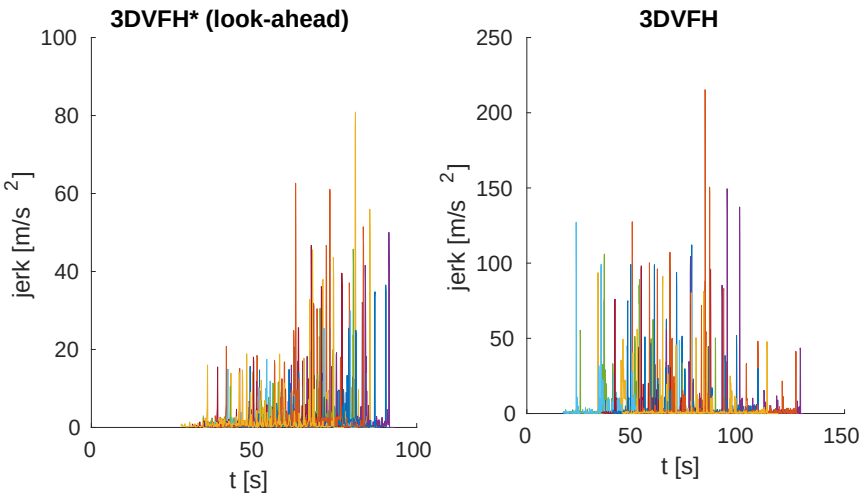


Figure 4.20: *Magnitude of jerk over time for all the tests runs. The different colors correspond to different runs. Ten runs have been performed for both algorithms*

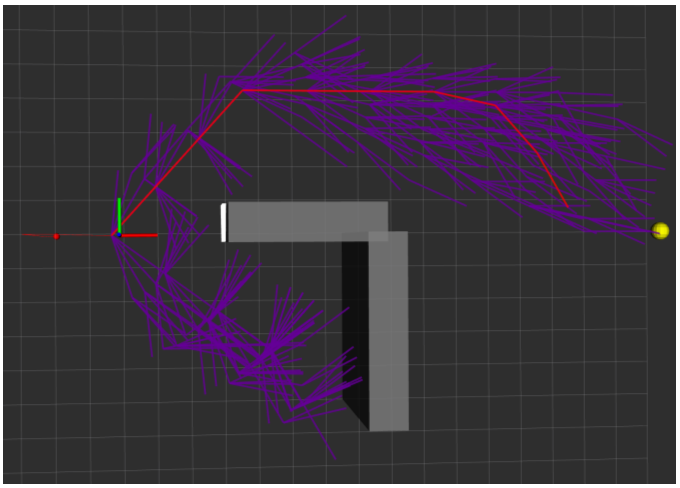


Figure 4.21: *Visualization of the look-ahead tree in the 3DVFH* algorithm. The coordinate system marks the body frame of the drone and the camera is aligned with the red x-axis. The goal is marked with a yellow sphere. The white surface is the part of the obstacle which can be seen when the point-cloud is cropped around the current position of the drone. At every node the whole point-cloud is cropped with respect to the node position, which is why the nodes on the right stop being expanded. They would have to avoid the broad part of the obstacle further to the right, which make them more costly than the paths to the left. The red line corresponds to the cheapest path through the search tree.*

4.5.2 Outdoor Flight Results

The 3DVFH* algorithm with memory was evaluated on the Intel Aero drone. For safety reasons the tests were performed on an open field using a cardboard obstacle depicted in figure 4.22. The obstacle has a side length of approximately 0.5 m and a height of 3 m. The algorithm was run using all the discussed adaptions except for the ground detection. The ground detection was disabled as the ground was flat and the chosen goal high enough such that the drone does not run any risk of ground collision. The parameters used in the test flight are listed in table 4.1. The test scenario consists of a given goal, which lies behind the obstacle. The drone autonomously takes off from ground and flies to the goal position by maneuvering around the obstacle. When the goal is reached the drone stops and hovers at the goal position. The test set consists of 26 runs. The individual runs start at slightly different initial positions and three different goals are given. The drone successfully avoided the obstacle in all the 26 test flights. The trajectories of the individual flights are shown in figure 4.23.

Parameter	value
bounding box size (x, y)	10 m
bounding box size (z)	2 m
sphere avoidance radius	2.5 m
max re-projection age	50 iterations
number of expanded nodes	10

Table 4.1: *Parameters used for outdoor testing*

Figure 4.22: *Cardboard obstacle on an empty field used for testing the 3DVFH* algorithm. The colored duct tape provides features for the stereo camera and therefore improves the visibility of the object.*

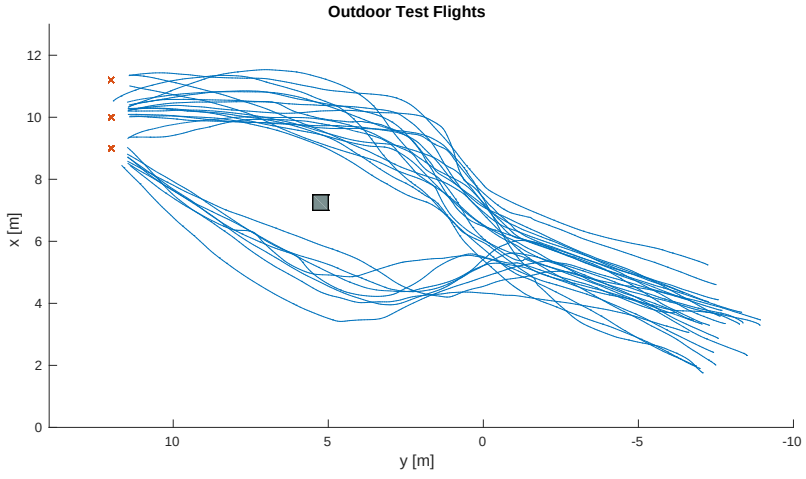


Figure 4.23: *Trajectories of the 26 test flights. All flights started at slightly different initial positions and lead to three different goal positions marked in red. The grey box visualizes the cardboard obstacle.*

It was found that the runtime of the algorithm largely depends on the core temperature because the clock-rate is throttled as the core temperature rises. Figure 4.24 shows the running time of the obstacle avoidance algorithm on the Intel Aero compute board using the parameters in table 4.1. The drone was stationary in an outdoor location where the air temperature was measured to be 5 °C. It can be shown that the algorithm runs significantly faster in the beginning, where the core temperature was still low. As the core temperature converges to 58 °C, the running time converges to a mean of 367.82 ms. For this test the algorithm continuously runs in the obstacle avoidance mode for a total time of 45 min. In a real flight the mean compute-time would be significantly lower, as the there are many iterations where there is no obstacle present and the drone can fly directly towards the goal or along a previously computed path.

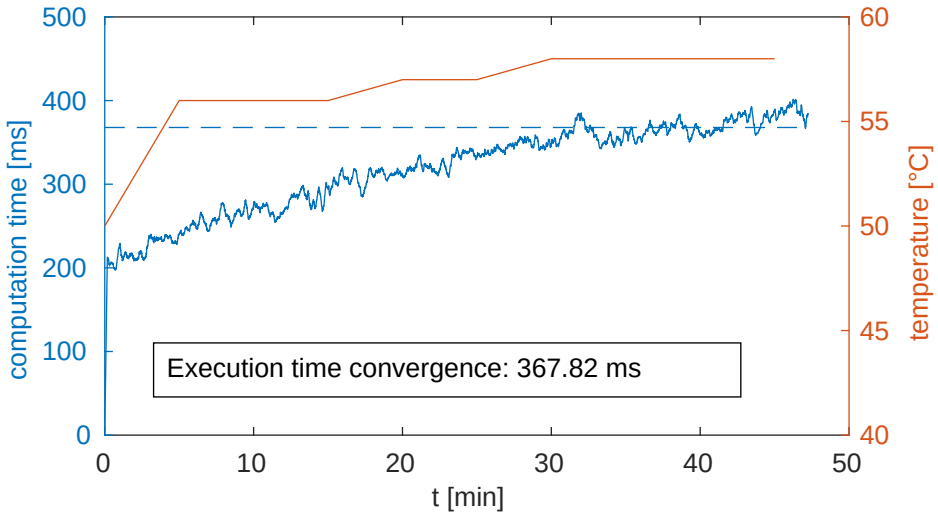


Figure 4.24: *Running time of the algorithm during obstacle avoidance in a benchmark test performed outdoor where the algorithm ran for approximately* 45 min. *The air temperature was about* 5 °C. *The red graph shows the core temperature of the CPU and the dashed blue line marks the convergence value of the algorithm running time in the given scenario.*

Chapter 5

Conclusion

This thesis introduces the 3DVFH* obstacle avoidance algorithm suitable for real-time application on UAVs. The algorithm combines the ideas behind the 3DVFH+ [26] and the VFH* [3] algorithm with a novel memory strategy. The introduced memory strategy is shown to be effective for outdoor flight as well as complex simulation scenarios. Therefore, in contrast to the 3DVFH+ algorithm, the 3DVFH* algorithm is not dependent on a global map of the environment which reduces the computational cost of the algorithm. The VFH* algorithm was extended to a 3D environment and the advantages of the look-ahead functionality were reproduced in the 3DVFH* algorithm. Due to this look-ahead capability, the 3DVFH* algorithm outperformed the 3DVFH+ in more complex scenorios. A computationally simple method for ground detection using only data from a forward facing camera was implemented. The ground detection method was shown to be able to maintain a minimum distance to the ground. Various features have been implemented to increase the robustness of the suggested approach and render application on UAVs possible. From the conducted outdoor flight tests it can be concluded that the 3DVFH* algorithm is able to avoid obstacles robustly and produces smooth flight behavior. This is the first study to provide a thorough evaluation of UAV flight using a 3D histogramic method.

5.1 Outlook

The performance of the 3DVFH* is severely limited by the available sensor information. For this study the algorithm was implemented on a simple hardware with one forward facing camera. The safety as well as the flight speed could be increased by using additional sensors to enlarge the field of view.

The 3DVFH* offers various possibilities for further improvement by consideration of the drone dynamics. Those constraints could be added to the cost function of the look-ahead tree by extending it to include not only positions but also velocities. The nodes could be interconnected by splines instead of straight lines, such that the maximum velocity for each segment can be calculated. The best trajectory would in this case be a trade-off between length of the path and the velocity at which it can be executed.

The simulation experiments suggested a vast potential of the 3DVFH* algorithm for complex environments where the performance largely benefits form the look-ahead functionality. It is therefore planned to test the algorithm in more challenging outdoor environments.

Bibliography

[1] J. Borenstein and Y. Koren, "The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.

[2] I. Ulrich and J. Borenstein, "VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1572 – 1577, 1998.

[3] I. Ulrich and I. Borenstein, "VFH*: Local obstacle avoidance with look-ahead verification," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2, pp. 1572–1577, 2000.

[4] C. Goerzen, Z. Kong, and B. Mettler, "A survey of motion planning algorithms from the perspective of autonomous UAV guidance," *Journal of Intelligent and Robotic Systems*, vol. 57, no. 1-4, pp. 65–100, 2010.

[5] L. Heng, L. Meier, P. Tanskanen, F. Fraundorfer, and M. Pollefeys, "Autonomous obstacle avoidance and maneuvering on a vision-guided MAV using on-board processing," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2472–2477, 2011.

[6] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, 2013.

[7] M. Hwangbo, J. Kuffner, and T. Kanade, "Efficient Two-phase 3D Motion Planning for Small Fixed-Wing UAVs," *IEEE International Conference on Robotics and Automation (ICRA)*, no. April, pp. 10–14, 2007.

[8] F. Fraundorfer, L. Heng, D. Honegger, G. H. Lee, P. Tanskanen, and M. Pollefeys, "Vision-Based Autonomous Mapping and Exploration Using a Quadrotor MAV," *Intelligent Robots and Systems (IROS)*, pp. 4557–4564, 2012.

[9] A. Mohammadi, "A New Path Planning and Obstacle Avoidance Algorithm in Dynamic Environment," *Electrical Engineering (ICEE)*, vol. 22nd Irani, pp. 1301–1306, 2014.

[10] C. Stachniss and W. Burgard, "An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments," *IEEE International Conference on Intelligent Robots and Systems (IROS)*, vol. 1, pp. 508–513, 2002.

[11] O. Brock and O. Khatib, "High-speed navigation using the global dynamic window approach," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 1, pp. 341–346, 1999.

[12] A. Ess, B. Leibe, K. Schindler, and L. Van Gool, "Moving obstacle detection in highly dynamic scenes," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 56–63, 2009.

[13] Y. Watanabe, A. Calise, and E. Johnson, "Vision-Based Obstacle Avoidance for UAVs," *AIAA Guidance, Navigation and Control Conference*, pp. 1–11, 2007.

[14] J. Z. Sasiadek and I. Duleba, "3D local trajectory planner for UAV," *Journal of Intelligent and Robotic Systems*, vol. 29, no. 2, pp. 191–210, 2000.

[15] S. Hrabar, "3D path planning and stereo-based obstacle avoidance for rotorcraft UAVs," *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pp. 807–814, 2008.

[16] J. Kuffner and S. LaValle, "RRT-connect: An efficient approach to single-query path planning," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2, pp. 995–1001, 2000.

[17] R. Deits and R. Tedrake, "Efficient mixed-integer planning for UAVs in cluttered environments," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 42–49, 2016.

[18] D. Shim, H. Chung, H. J. Kim, and S. Sastry, "Autonomous Exploration In Unknown Urban Environments For Unmanned Aerial Vehicles," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, no. August, pp. 1–8, 2005.

[19] D. H. Shim, H. Chung, and S. S. Sastry, "Conflict-free navigation in unknown urban environments," *IEEE robotics & automation magazine*, pp. 27–33, 2006.

[20] K. Song and J. Huang, "Fast optical flow estimation and its application to real-time obstacle avoidance," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, pp. 2891–2896, 2001.

[21] B. Yamauchi, "The Wayfarer modular navigation payload for intelligent robot infrastructure," *Unmanned Ground Vehicle Technology VII, Proceedings of International Society for Optics and Photonics*, vol. 5804, no. 781, pp. 85–96, 2005.

[22] B. You, J. Qiu, and D. Li, "A novel obstacle avoidance method for low-cost household mobile robot," *IEEE International Conference on Automation and Logistics, ICAL*, pp. 111–116, 2008.

[23] D. Jie, M. Xueming, and P. Kaixiang, "IVFH*: Real-time Dynamic Obstacle Avoidance for Mobile robots," *International Conference on Control Automation Robotics & Vision (ICARCV)*, pp. 7–10, 2010.

[24] A. Babinec, M. Dekan, F. Duchoň, and A. Vitkoa, "Modifications of VFH navigation methods for mobile robots," *Procedia Engineering*, vol. 48, pp. 10–14, 2012.

[25] A. Babinec, F. Duchoň, M. Dekan, P. Pásztó, and M. Kelemen, "VFH*TDT (VFH* with Time Dependent Tree): A new laser rangefinder based obstacle avoidance method designed for environment with non-static obstacles," *Robotics and Autonomous Systems*, vol. 62, no. 8, pp. 1098–1115, 2014.

[26] S. Vanneste, B. Bellekens, and M. Weyn, "3DVFH+: Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap," *MORSE 1st International Workshop on Model-Driven Robot Software Engineering*, 2014.

[27] B. I. Kazem, A. H. Hamad, and M. M. Mozael, "Modified Vector Field Histogram with a Neural Network Learning Model for Mobile Robot Path Planning and Obstacle Avoidance," *International Journal of Advancements in Computing Technology*, vol. 2, no. 5, pp. 99–110, 2010.

[28] A. H. Hamad and F. B. Ibrahim, "Path Planning of Mobile Robot Based on Modification of Vector Field Histogram using Neuro-Fuzzy Algorithm," *International Journal of Advancements in Computing Technology*, vol. 2, no. 3, pp. 1–10, 2010.

[29] P. G. Zavlangas, S. G. Tzafestas, and K. Althoefer, "Fuzzy Obstacle Avoidance and Navigation for Omnidirectional Mobile Robots," *European Symposium on Intelligent Techniques*, pp. 14–15, 2000.

[30] S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert, "Learning monocular reactive UAV control in cluttered natural environments," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1765–1772, 2013.

[31] D. Dey, K. S. Shankar, S. Zeng, R. Mehta, M. T. Agcayazi, C. Eriksen, S. Daftry, M. Hebert, and J. A. Bagnell, "Vision and learning for deliberative monocular cluttered flight," *Field and Service Robotics*, vol. 113, pp. 391–409, 2016.

[32] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *Autonomous robot vehicles*, pp. 500–505, 1985.

[33] S. Waydo, "Vehicle motion planning using stream functions," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 2, pp. 2484–2491, 2003.

[34] T. T. Mac, C. Copot, A. Hernandez, and R. De Keyser, "Improved potential field method for unknown obstacle avoidance using UAV in indoor environment," *IEEE International Symposium on Applied Machine Intelligence and Informatics*, pp. 345–350, 2016.

[35] H. Oleynikova, D. Honegger, and M. Pollefeys, "Reactive avoidance using embedded stereo vision for MAV flight," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 50–56, 2015.

[36] J. Ng and T. Bräunl, "Performance comparison of Bug navigation algorithms," *Journal of Intelligent and Robotic Systems*, vol. 50, no. 1, pp. 73–84, 2007.

[37] N. Buniyamin, W. A. J. Wan Ngah, N. Sariff, and Z. Mohamad, "A simple local path planning algorithm for autonomous mobile robots," *International journal of systems applications, Engineering & development*, vol. 5, no. 2, pp. 151–159, 2011.

[38] R. Simmons, "The curvature-velocity method for local obstacle avoidance," *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 4, no. April, pp. 3375–3382, 1996.

[39] "Robot Operating System," http://www.ros.org/.

[40] L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6235–6240, 2015.

[41] "3DR IRIS Quadcopter," http://www.arducopter.co.uk/iris-quadcopter-uav.html.

[42] "Intel Aero Ready to Fly Drone," https://click.intel.com/intel-aero-ready-to-fly-drone.html.

[43] "Intel Realsense R200," https://software.intel.com/en-us/realsense/previous.

Appendix A

Appendix

A.1 Additional Results Graphics

A.1.1 Memory 3DVFH Simulation

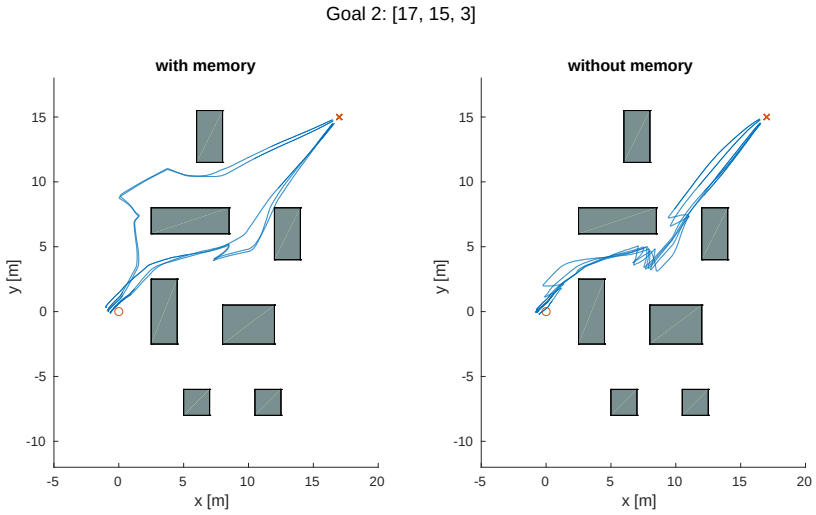


Figure A.1: *Trajectories of the drone flying towards goal 2. The left figure shows the paths taken by the 3DVFH with memory, whereas the right figure shows the trajectories of the 3DVFH algorithm without memory.*

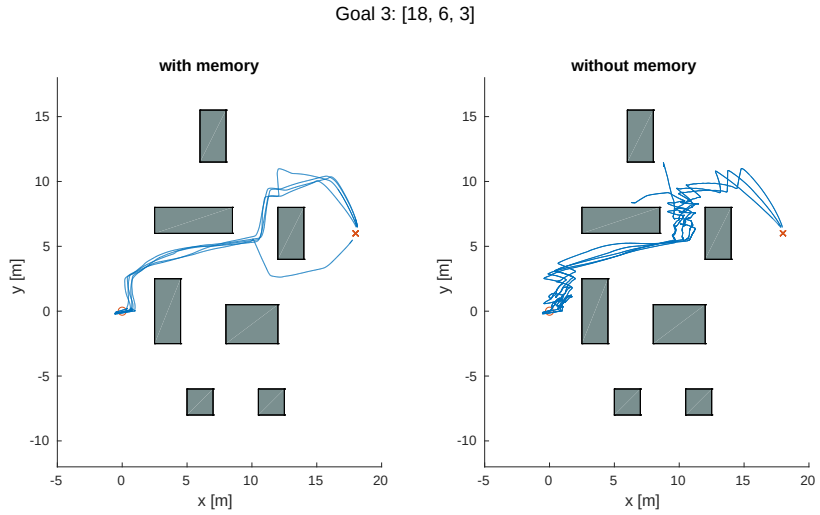


Figure A.2: *Trajectories of the drone flying towards goal 3. The left figure shows the paths taken by the 3DVFH with memory, whereas the right figure shows the trajectories of the 3DVFH algorithm without memory.*

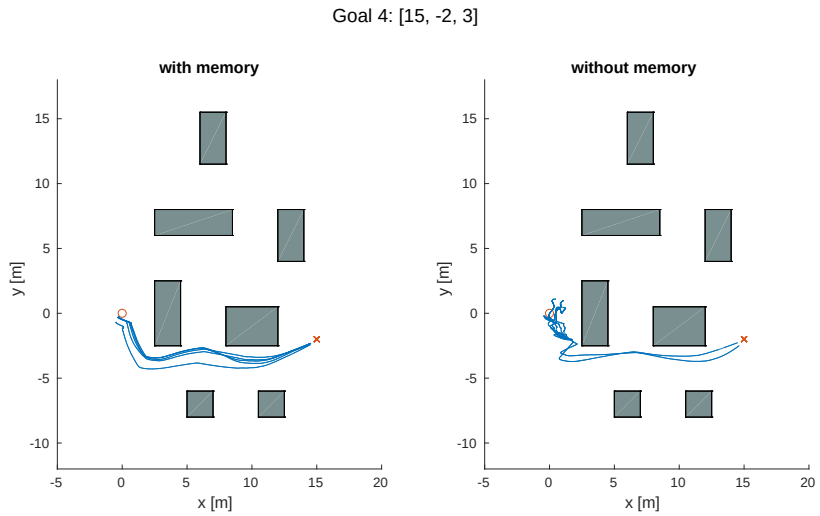


Figure A.3: *Trajectories of the drone flying towards goal 4. The left figure shows the paths taken by the 3DVFH with memory, whereas the right figure shows the trajectories of the 3DVFH algorithm without memory.*

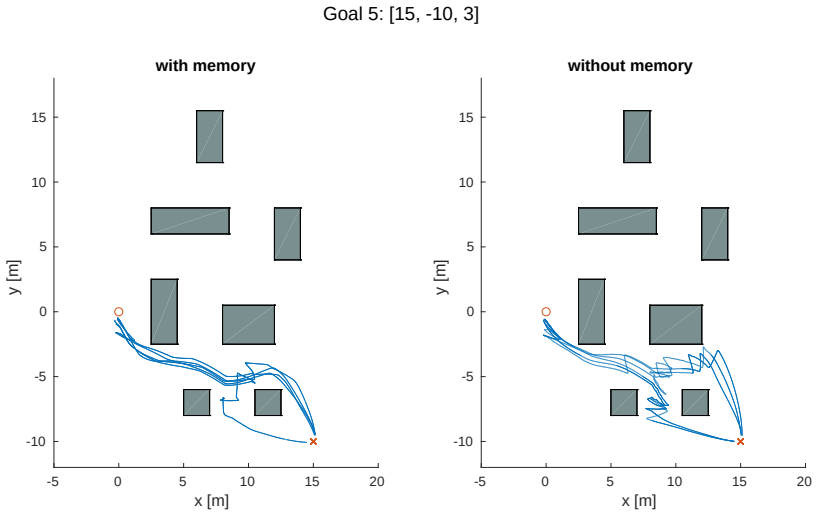


Figure A.4: *Trajectories of the drone flying towards goal 5. The left figure shows the paths taken by the 3DVFH with memory, whereas the right figure shows the trajectories of the 3DVFH algorithm without memory.*