Frustumbug: a 3D Mapless Stereo-Vision-based Bug Algorithm for Micro Air Vehicles

Ruben Meester

Delft University of Technology

TU Delft

Frustumbug: a 3D Mapless Stereo-Vision-based Bug Algorithm for Micro Air Vehicles

by

Ruben Meester

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on January 18th, 2023.

Student number:	4453964
Project duration:	March 2021 – January 2023
Thesis committee:	Prof. Dr. G.C.H.E. de Croon
	Dr. ir. C.J.M. Verhoeven
	Dr. ir. C. de Wagter
	Ir. T. van Dijk

TU Delft

Preface

Dear reader,

Before you lies my master thesis called “Frustumbug: a 3D Mapless Stereo-Vision-based Bug Algorithm for Micro Air Vehicles”. Since drones are getting smaller and lighter, I have been researching efficient and lightweight autonomous path planning solutions for drones.

I sincerely hope that the value of my work is recognised by someone who has the interest and resources to focus on future work. The results have validated the method and I am proud to see how well the algorithm performs. Now it is time to solve for the very few cases where the drone does not reach its target, and to brainstorm about how to best extend its sensing capabilities to improve performance, while keeping the added computational expenses low.

I would like to direct a massive thank you to my PhD supervisor ir. Tom van Dijk for his help in this project full of new challenges for me. Normally supervisors are given a bottle of wine at the diploma ceremony, but Tom deserves the whole crate. He has been very approachable and is willing to put his own research aside to help me. Also, I would like to thank my supervisor Prof. dr. Guido de Croon for his guidance and critical mindset during our meetings. He is a great example for always being in a good mood, whilst living an extremely busy life.

It is a great relieve to be able to say that I can end my master thesis on a high note. The topic has been exciting, especially since I started working with the real drone. I enjoyed the problem-solving approach and the room for creativity. But as you know, the only reason that these highs exist, is because there have been lows. These were especially present during the pandemic lockdowns. I am glad to see that the TU Delft is focusing more and more on mental health, but perhaps it should just be less of a taboo in general.

I have tried many different locations to study during my thesis, and I would like to say that the connections I have made with the people in room 2.56 mean a lot to me and gave me motivation to continue studying, day in, day out.

Megan, I am excited now that we have both finished our MSc thesis. I look forward to the coming year, full of new experiences and who knows where we will be. I value your support during my low days, and your happiness during my high days.

Via this way, I would also like to thank my parents for their interest, support and motivation during my studies. I came to Delft without knowing anything about student life, and they simply trusted that it would turn out alright. I am grateful to be given this chance.

Thank you to all the friends I have made over the past few years: housemates, Duikboot, Tenniphil and VSV friends. You have made these past years extremely memorable and I look forward to keeping in touch.

For now, I would like to thank you for your interest in my work, and I hope you will enjoy reading.

R.S. Meester
Delft, January 2023

Contents

List of Figures

List of Tables

List of Abbreviations

ABC	Artificial Bee Colony
ACO	Ant Colony Optimisation
BA	Bat Algorithm
BF	Boundary Following
BFO	Bacterial Foraging Optimisation
BFO	Best-First Optimisation
BFT	Boundary Following - Turning
BFW	Boundary Following - Waypoint
BIP	Binary Integer Programming
BM	Block Matching
BT	Birchfield and Thomasi
BVP	Boundary Value Problem
CCW	Counter-clockwise
CEG	Convex-Edges-Graph
CMU	Carnegie-Mellon University
CS	Cuckoo Search
CSG	Constructive Solid Geometry
CT	Census Transforms
CW	Clockwise
DE	Differential Evolution
DT	Decision Tree
ELAS	Efficient Large-Scale Stereo Matching
FA	Firefly Algorithm
FoV	Field of View
FSM	Finite-State Machine
GA	Genetic Algorithm
GPS	Global Positioning System
HS	Harmony Search
HS	Harmony Search
IMU	Inertial Measurement Unit

IWO	Invasive Weed Optimisation
K-PRM	K-Nearest Probabilistic Road Map
LETG	Local-ϵ-Tangent-Graph
LIDAR	Light Detection and Ranging
MA	Memetic Algorithm
MI	Mutual Information
MILP	Mixed Integer Linear Programming
MtG	Motion to Goal
MtW	Motion to Waypoint
NCC	Normalized Cross-Correlation
PRM	Probabilistic Road Map
PSO	Particle Swarm Optimisation
RADAR	Radio Detection and Ranging
RRG	Rapidly-exploring Random Graph
RRT	Rapidly-exploring Random Tree
RT	Rank Transforms
SA	Simulated Annealing
SAD	Sum of Absolute Differences
SB	Scanning Boundary
SC	Scanning Climb
SCE	Shuffled Complex Evolution
SDB	Scanning Descent Backwards
SDE	Scanning Descent Either
SDF	Scanning Descent Forwards
SFLA	Shuffled Frog Leaping Algorithm
SG	Scanning Goal
SGBM	Semi-Global Block Matching
SONAR	Sound Navigation and Ranging
sPRM	Simplified Probabilistic Road Map
SPS-St	Slanted Plane Smoothing Stereo
SSD	Sum of Squared Differences

SUMT	Sequential Unconstrained Minimization Technique
SW	Scanning Waypoint
TTC	Time-To-Contact
VD	Voronoi Diagram
VFF	Virtual Force Field
VFH	Vector Field Histogram
VG	Visibility Graph
WC	Waypoint Climb
WD	Waypoint Descent
WR	Waypoint Reverse

Introduction

The drone market is going through a massive economic growth, which can be translated into billions of dollars (Aswini et al., 2018). Researchers are highly interested in developing efficient on-board obstacle avoidance algorithms, since there is no room for large on-board computers. Solutions exist, and are currently on the market. Both DJI and Skydio have drones which use multiple stereo camera pairs to generate computationally expensive 3D depth, colour and heat maps. Skydio needs over 256 GPU cores to handle over a trillion operations per second. Since these drones are small, a relative high weight ratio is allocated to the equipment needed to plan its path. A simpler solution could decrease some of the weight. Decreasing weight is advantageous for the battery life for (e.g.) industrial inspection or package delivery, but it is crucial for applications like interplanetary missions, which are taking place present-day. For example, Mars' first UAV "Ingenuity" landed on the planet in February 2021 and has made 18 flights up to December 2021. Any weight that can be saved does not need to be transported millions of kilometres.

Problem statement

The goal of this literature study is to explore multiple distance measurement sensors and map-based and mapless path planning algorithms to create a new 3D static obstacle avoidance algorithm, which can navigate a small drone from outdoor GPS coordinate A to B, while minimising deviation from the nominal path and minimising computational resources.

Research questions

To find an answer to the problem statement, it has been divided up into three research questions.

Research question 1: Which distance measurement sensor would be a good candidate to obtain the relevant information about the environment?

To avoid obstacles, the drone needs sensors that will give it information about its environment. Therefore, the first research question looks at the different types of sensors, and compares their advantages and disadvantages. Furthermore, it investigates the necessary post-processing of the incoming data.

Research question 2: Which existing method, or combination of existing methods, is the most advantageous to find an obstacle free path?

Sub-question 2.1: How to avoid obstacles?

Sub-question 2.2: How to avoid local minima?

Sub-question 2.3: How to limit computational expenses?

Sub-question 2.4: How to find paths that resemble global optimal solutions?

Since this question is rather broad, it has been broken down into four sub-questions. Firstly, how to avoid obstacles? In other words, once the drone realises there is an obstacle in its way, how does it pick the next waypoint? Secondly, how to avoid local minima? There exists the possibility to get trapped by obstacles, so the drone will have to find its way out. Thirdly, how to limit computational expenses? Fourthly, how to find paths that resemble global optimal solutions? The last two have an interesting relationship, since generally the computationally more intensive algorithms produce better results. Running computationally heavy algorithms on a small drone's processor will result in a low FPS.

Two drones that were often used in research at the TU Delft are the Parrot ARDrone 1.0 (De Croon et al., 2013, De Croon et al., 2014) and the Parrot ARDrone 2.0 (van Hecke et al., 2018, Remes et al., 2013). In recent years, the focus has been shifted to smaller drones which use a JeVois stereo camera, running on a 1.34 GHz ARM A7 quad-core processor, which has 256 MB RAM ("JeVois Smart Machine System", n.d.). Although it is not yet certain whether stereo cameras are the way to go, these give rough estimates of representative values for small processors used and will be taken as a reference in this study.

Research question 3: Is it necessary to create a map of the environment?

Sub-question 3.1: In what ways can obstacles be represented?

Sub-question 3.2: In what ways can the workspace be represented?

A map provides more information about the environment than a local percept, but requires more memory to be stored and more computation to be built. This question will be broken down into two sub-questions. Firstly, in what ways can obstacles be represented? Secondly, in what ways can the workspace be represented? These so-called environment representations are used by map-based path planning algorithms.

Assumptions

The assumptions for this literature study are:

- The environment is outdoor.
- The drone has to avoid static obstacles only
- GPS signal is available.
- The target is known and static.
- Maze-like environments are not considered.
- No a priori information about the environment is known to the drone.
- Flights take place during daylight.

Organisation

This final product consists of two parts: a scientific paper and a literature study. The scientific paper presents the implementation and results. Since the literature study is slightly outdated, there is a chance that conclusions or implementation proposals have changed. If so, the scientific paper is leading. The literature study answers the research questions: chapter 1 will dive into the world of sensing. After, chapter 2 explains different methods for analysing and processing this sensor information. An interesting topic here is whether or not to store a map which represents both obstacles and the workspace. Multiple map-based path planning algorithms exist, which will be presented in chapter 3. Aside from that, it is also possible to plan a path without any environment representation. These mapless algorithms are discussed in chapter 4. Subsequently, chapter 5 explains how the proposed algorithm is going to be implemented, and finally chapter 6 presents the conclusion.

I

Scientific Paper

Frustumbug: a 3D mapless stereo-vision-based bug algorithm for micro air vehicles

R.S. Meester [*], G.C.H.E. de Croon [†], T. van Dijk [†]
Control and Simulation, Faculty of Aerospace Engineering
Delft University of Technology, The Netherlands

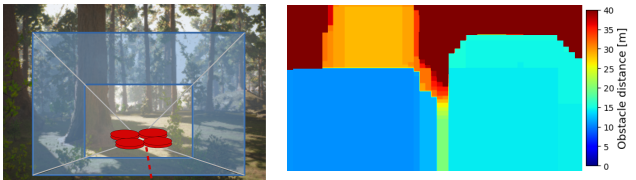
Abstract

We present a computationally cheap path planning algorithm for drones in 3D, by combining stereo vision with bug algorithms. Obstacle avoidance is important, but difficult for robots with limited resources, such as drones. Stereo vision requires less weight and power than active distance measurement sensors, but typically has a limited Field of View (FoV). In addition, the stereo camera is fixed on the drone, preventing sensor movement. For path planning, bug algorithms require few resources. We base our proposed algorithm, Frustumbug, on the Wedgebug algorithm, since this bug algorithm copes with a limited FoV. Since Wedgebug only focuses on 2D problems, the Local-ϵ-Tangent-Graph (LETG) is used to extend the path planning to 3D. Disparity images are obtained through an optimised stereo block matching algorithm. Obstacles are expanded in disparity space to obtain the configuration space. Furthermore, Frustumbug has an improved robustness to noisy range sensor data, and includes reversing, climbing and descending manoeuvres to avoid or escape local minima. The algorithm has been extensively tested with 225 flights in two challenging simulated environments, with a success rate of 96%. Here, 3.6% did not reach the goal and 0.4% collided. Frustumbug has been implemented on a 20 gram stereo vision system, and guides drones safely around obstacles in the real world, showing its potential for small drones to reach their targets fully autonomously.

1 Introduction

The drone market keeps growing, with applications ranging from industrial inspection to search-and-rescue and package delivery. Since most of the applications have to be performed autonomously, autonomous obstacle avoidance is of the essence. Current solutions for this combine multiple sensors and are computationally expensive and memory intensive. An efficient, lightweight solution would require less energy from the battery, increasing the flight time. Moreover, the size and weight of on-board processors can be reduced, allowing tiny drones to navigate autonomously to a goal. Ideally, this solution makes decisions that have an explanation based on logic, instead of randomly generating obstacle-free paths.

[*]MSc Student
[†]Supervisor



(a) Drone's viewing frustum

(b) Depth image obtained from information inside frustum

Figure 1: Creation of the 3D viewing frustum. Drone represented by the red structure. Red dotted line indicates the travelled path. Depth image includes a safety margin around obstacles.

Over the past few decades, several distance measurement sensors have been studied. To allow for on-board path planning, vision-based sensors are the only promising candidate when looking at size and weight and power [Aswini et al., 2018]. To limit the computational resources used by the drone, a mapless method is preferred over a map-building method. Since outdoor flight is assumed, building maps of large, complex environments requires high resources.

Ideally, the generated path should resemble the globally shortest path. Therefore, the drone's ability to move up or down to avoid obstacles needs to be exploited, hence this paper focuses on path planning in three dimensions. 3D mapless vision-based methods exist, but are often combined with probabilistic roadmaps [e.g. Matthies et al., 2014 and Lee et al., 2021] or machine learning algorithms [e.g. Doukhi and Lee, 2022 and Grando et al., 2022]. Since there is a certain amount of randomness in these methods, the choice of paths can not always be explained, and may not always lead to the goal. Furthermore, these methods tend not to include strategies for escaping local minima, or use randomly generated sub-optimal paths to escape [e.g. Matthies et al., 2014 and Yu et al., 2018].

Bug algorithms are used for robots with limited computational resources, since they plan their path using very little computation and memory. Another advantage is that these finite-state machines provide a strategy for escaping local minima. Wedgebug [Laubach and Burdick, 1999] is a two-dimensional bug algorithm that solved the problems for robots with a small Field of View (FoV), by rotating the sensor when more information is required. However, to create an accessible algorithm for the many drones and cameras without that capability, and to limit the necessary moving components, the algorithm should be designed without the require-

ment of camera rotation. When stuck in a local minimum, Wedgebug will follow the obstacle boundary until an intermediate waypoint is found which decreases the distance to goal [Laubach and Burdick, 1999]. However, in 2D an obstacle can only be avoided in two directions: left or right. In 3D, the number of directions to avoid an obstacle increases to infinity, since any location on the obstacle boundary can be chosen. 3DBug [Kamon et al., 1996] reduces the necessary calculations by discretising the continuous obstacle contour into a set of points. However, 3DBug assumes infinite sensor range and a 360 degrees FoV. Furthermore, it has been tested on relatively simple environments where the locally best decisions mostly lead to the globally best paths.

The goal of this study is to create a 3D path planning algorithm, which can navigate a drone to a target while minimising deviation from the nominal path using limited resources. We contribute to the existing literature by: (i) extending Wedgebug to operate in 3D, (ii) introducing additional navigation states for avoiding and escaping local minima, (iii) designing a novel waypoint selection method to increase robustness to noisy range sensor data, (iv) extensively testing the proposed algorithm in simulation and real-world, using a very light-weight stereo vision system. We will call it Frustumbug, as the frustum can be seen as the 3D extension of a wedge. Figure 1 shows how the viewing frustum is shaped (Figure 1a), along with its corresponding configuration space (Figure 1b). Our contributions take the next step towards a publicly available self-contained path planning package and allows for future extensions, since each path planning decision follows from logical decision captured in a decision tree, and each branch can be altered individually. Assumptions include outdoor flight during daylight, where GPS signal is available. Furthermore, the target and obstacles are assumed static and maze-like environments are not considered. The remaining part of the paper is structured as follows: first, section 2 shows the related work and section 3 explains how the chosen approach is implemented. Then, section 4 presents the experimental results from both simulation and real world testing. This paper then discusses the results in section 5 and concludes in section 6.

2 Relevant Work

The relevant work is divided into two sections: we first look at literature on distance measurement sensors to optimise our sensing, followed by literature on avoidance algorithms to complete the path planning package.

2.1 Sensing

Vision-based sensors are suitable for outdoor environments, since they usually have enough texture to be captured with a camera [Matthies et al., 2014]. Deep/Machine learning algorithms are not considered for obtaining depth information, since these methods consist of multiple convolutional layers, which require large matrices to be stored and many computations to be made [e.g. Doukhi and Lee, 2022]. Moreover, parameters of a neural network result from training the algorithm for a longer period of time. If it shows inconsistent results for repetitive image texture, or low-textured regions, the parameters can not be easily adapted to improve the result. It would have to be trained again on an improved dataset, or with different hyperparameters.

Vision-based sensing can be done with one, two or more cameras. Monocular vision in obstacle avoidance is generally paired with optical flow methods. These have their drawbacks in obtaining disparity images compared to stereo vision: a larger image area needs to be searched to find matching pixels, estimates on position and attitude changes need to be included in the calculation and it is not able to sense distances in the direction of travel, since there is no optical flow in the focus of expansion [van Dijk, 2020]. Following the argument above, stereo vision is preferred, since the added weight and required power are small; they can be as lightweight as 4 grams and be run on 168 MHz microprocessors [McGuire et al., 2017]. To limit the computational resources, a solution using more than two cameras is not considered.

Stereo matching can be done either local or global, and sparsely or densely. Although global methods tend to generate better results in low-texture environments, it comes at a computational cost [Liu et al., 2020]. Furthermore, outdoor environments during daylight have enough texture for local methods. Sparse methods only calculate the disparity for interesting features, which decreases the computational cost. However, this would leave large gaps in the disparity image, which are unwanted. Therefore, a local dense stereo matching algorithm will be used to create the disparity image. Possible stereo matching algorithm candidates can be compared by their run-time, used platforms, performance and code availability from common benchmarks KITTI [Geiger et al., 2012] and Middlebury [Scharstein and Szeliski, 2002]. The most promising candidates are Block Matching (BM) [Scharstein and Szeliski, 2002] and Semi-Global Block Matching (SGBM) [Hirschmuller, 2005], due to their fast execution time and performance in outdoor environments [Lyrakis, 2019]. In addition, it was concluded that BM is preferred over SGBM, since it is faster and does not have the tendency to fill the sky with incorrect large disparity values [Lyrakis, 2019].

Block Matching can be performed using various metrics for similarity, pre-/post-filters and block sizes. The Sum of Absolute Differences (SAD) consists of relatively simple calculations, resulting in a fast algorithm [Patil et al., 2013]. Filters are used to delete outliers, which can be caused by occlusions or untextured or repetitive image regions. These filters will be discussed in more detail in subsection 3.1.

2.2 Avoidance

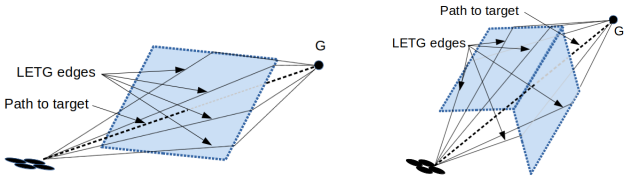
After the obstacles have been observed by the stereo camera, a path needs to be planned around them for avoidance. Path planning can be done by using either map-based or map-

less methods. Map-based methods either start with a complete map of the environment (global map), or build them using sensors (local map), here referred to as map-building [Elmokadem and Savkin, 2021]. Mapless methods do not utilise this previously gathered information and show reflex-like behaviour based on current sensor data [Elmokadem and Savkin, 2021]. Since complete maps of outdoor environments are hardly ever available, map-based approaches which require a global map are unrealistic for outdoor path planning.

There is a trade-off in the decision for a map-building or mapless method. The objective of this research is to minimise deviation from the nominal path using limited resources. Map-building methods require more computational resources and memory, whereas mapless methods make reactive decisions using a relatively small amount of processing of the sensor data [Elmokadem and Savkin, 2021]. However, when stuck in a local minima, a mapless method does not use any information of previously sensed obstacles. It needs to scan the obstacles again and it could fail to identify an escape path already known to the map-building method. Hence, map-building methods could lead to more optimal paths. Nevertheless, only a small amount of dead ends is expected in outdoor environments, since obstacles can also be avoided by passing over them. Therefore, a mapless approach is chosen to limit computational resources.

Bug algorithms are computationally cheap mapless path planning algorithms, due to their simple reasoning in finding a way to the goal. For example, Bug2 [Lumelsky and Stepanov, 1986] draws a main line, or *M-line*, from start- to goal position and follows this line until an obstacle is encountered. The obstacle boundary is then followed until the *M-line* is met, and the motion to goal is resumed. Wedgebug [Laubach and Burdick, 1999] is of interest because it uses a range sensor with a small FoV, and we also have limited FoV vision. It scans a 'wedge' to see if the goal is safe, or if an intermediate waypoint needs to be set. In case there is no solution inside the wedge, it will scan an adjacent wedge to obtain more information. This algorithm was used to move a planetary rover horizontally, i.e. in 2D. Extending it to 3D is necessary to utilise the drones ability to move vertically. 3DBug [Kamon et al., 1996] is a bug algorithm that extends path planning to 3D by constructing a Local-ϵ-Tangent-Graph (LETG) which discretises the obstacle boundary. This is necessary since any obstacle boundary is a line with an infinite number of points. 3DBug assigns a finite number of points to the boundary, and creates the LETG, which consists of the lines connecting the current- and the goal position to these waypoints, as shown in Figure 2. Figure 2a shows an example for a 2D plane and Figure 2b for a concave obstacle.

A reactive approach requires each disparity image to be used individually for path planning. Since the assumption that the robot can be modelled as a point robot does not hold, a safety margin needs to be included. This can be done with low computational costs and little memory by expanding the

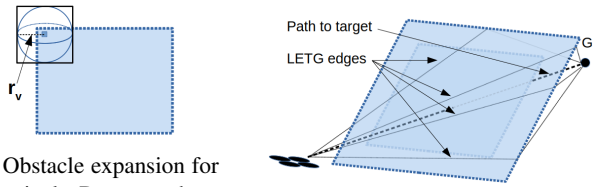


(a) 2D plane blocking the path

(b) Concave obstacle blocking the path

Figure 2: LETG edges used by 3DBug for an obstacle blocking the drone's path to goal, represented by the thicker line.

obstacles in disparity space to construct the configuration space (C-space) [Matthies et al., 2014], as presented in Figure 3. Each pixel in the disparity image is translated into its world coordinates and a sphere of a predefined expansion radius is drawn around it. An example for one pixel is shown in Figure 3a. Next, a rectangle which hides the sphere behind it is drawn just in front of the sphere, to simplify calculations. The area captured by each rectangle obtains a depth value equal to the pixel's original depth value minus the expansion radius. Using the C-space, the drone can be modelled as a point and it can safely choose an obstacle-free pixel in the disparity image, as shown in Figure 3b.



(a) Obstacle expansion for one pixel. Draw a sphere with expansion radius r_v and capture in a rectangle.

(b) C-space after obstacle expansion

Figure 3: Obstacle expansion in disparity space using expansion radius r_v, creating the drone's C-space.

3DBug makes two assumptions that do not hold: The range sensor has perfect readings and all obstacles can be modelled as polyhedral obstacles. The stereo camera will not provide perfect readings, which makes it difficult to model the obstacles as polyhedral. As a solution, a boundary tracing algorithm can be used to find the obstacle contours in the disparity image. The Moore-Neighbor tracing algorithm is often used, since it is fast and easy to implement [Reddy et al., 2012]. After creating the contour, the point that represents the locally shortest path can be calculated. Since a boundary tracing could identify the (unsafe) concave edge in Figure 2b as safe, sudden points are searched within the image. These are defined in Pointbug [Buniyamin et al., 2011] as large sudden changes in depth readings. Hence, the sudden points will ensure that the drone chooses a waypoint around the obstacle instead of into it.

If the drone is stuck in a local minimum, it needs to apply a strategy to escape. Wedgebug halts the robot and scans

additional wedges to find an intermediate waypoint [Laubach and Burdick, 1999] and goes into boundary following mode. Here, the robot skirts the contour of the obstacle and leaves the boundary whenever the goal direction is free of obstacles. Other escaping strategies are proposed by potential field methods, for example introducing virtual obstacles that push the drone away [Lee and Park, 2003] or using rotational forces to steer around the obstacle [Sfeir et al., 2011]. However, the literature focuses on solutions for 2D problems, and 3D mapless obstacle avoidance algorithms tend not to mention they are getting stuck in local minima [e.g. Lee et al., 2021 and Oleynikova et al., 2015]. Due to the limited FoV of stereo vision also in the vertical direction, situations in which the drone cannot instantly see a vertical escape route can definitely occur.

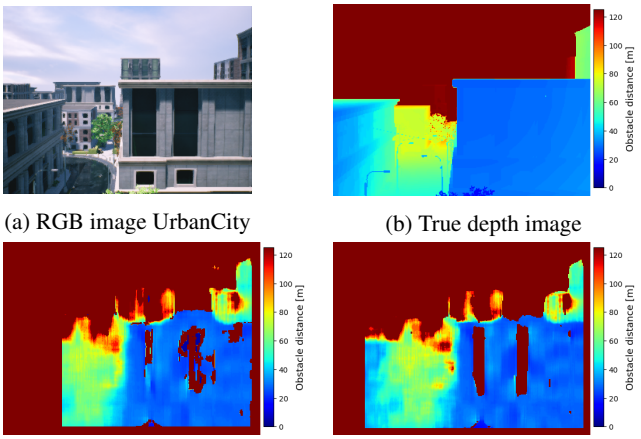
3 Method

The method section first discusses the sensing, followed by the intermediate waypoint selection method and the finite-state machines. The latter is divided into the description of the Wedgebug states, changes in these states and the additional states for avoiding and escaping local minima.

3.1 Sensing

The rectified stereo image pair is fed to the stereo matching algorithm to generate the disparity image. The depth in metres can be obtained by multiplying the stereo base by the focal length, and dividing by the disparity in pixels. Figure 4 shows the result of the stereo BM algorithm using SAD. The RGB image in Figure 4a is taken from simulated environment UrbanCity[1]. Its corresponding true depth image is shown in Figure 4b. The generated depth image shown in Figure 4c shows the result using BM's default parameters. To improve the result, a grid search is performed to find the parameter combination giving the best disparity images. For this, a dataset of 100 images with ground truth depth values is generated from the open-source simulation software AirSim[2]. To quantitatively compare the performance of the parameter combination, each disparity image was evaluated for completeness, accuracy and noise. The parameters are for block size, texture filter, uniqueness ratio filter, speckle filter, left-right consistency check. An extra filter has been added in an attempt to further reduce noise: morphological opening. The method and results are described in Appendix B. The generated depth image using these optimised parameters is shown in Figure 4d.

The next step is to expand the obstacles in disparity space, to create the C-space. As suggested by the authors, look-up tables for obstacle expansion coefficients are generated preflight for efficiency [Matthies et al., 2014]. The method is shown in Figure 5. Figure 5a is a small cutout of the optimised parameters' depth image showing two pixels as example of obstacle expansion. The width of r_v has to be at least the half the width of the drone to include a large enough safety margin. Expanding every pixel of an image with a pixel resolution of 240x320, results in the C-space shown in Figure 5b. To further improve performance, pixels are only expanded if their depth value is below a threshold, and the images are downscaled. In the example of Figure 5c, a depth threshold of 30 meters is used, speeding up the runtime by a factor 2.5. Figure 5d shows the image downscaled by a factor 100, yielding a 24x32 resolution, and is 110 times faster than the complete C-space. The thresholded and downscaled image shown in Figure 5e is 210 times faster than the complete C-space, with satisfactory results. Figure 5f shows what happens to the C-space if the BM parameters are not optimised: the noisy pixels on the nearby building cover the entire FoV after obstacle expansion, making path planning impossible. More examples can be found in Appendix A.



(a) RGB image UrbanCity

(b) True depth image

(c) Depth image using BM with default parameters

(d) Depth image using BM with optimised parameters

Figure 4: True- and stereo Block Matching depth images, belonging to an RGB image from simulated environment UrbanCity.

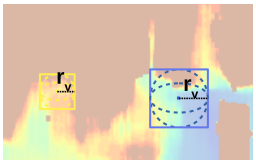
The best parameters for a dataset are not necessarily the best parameters for each individual image. For example, some images could have highly repetitive textures and would need different values for the uniqueness ratio parameter. Therefore, if the drone is not able to move due to poor stereo matching, Frustumbug will change the uniqueness ratio temporarily, in an effort to escape the current location. Results can be found in Appendix C.

3.2 Waypoint selection

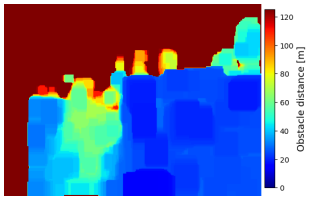
Wedgebug and 3DBug are both finite-state machines (FSM). They set intermediate waypoints to avoid obstacles, and move to the goal position when possible. The basis of the avoidance strategy is shaped by Wedgebug.

The novel waypoint selection method, shown in Figure 6, finds the locally best waypoint based on its sensor data. The

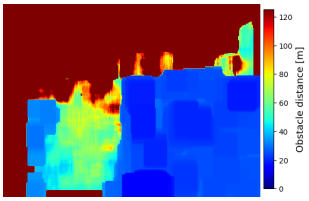
[1] https://www.unrealengine.com/marketplace/en-US/product/urban-city
[2] https://microsoft.github.io/AirSim/



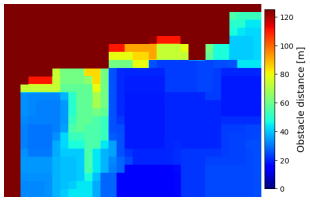
(a) Expansion radius r_v



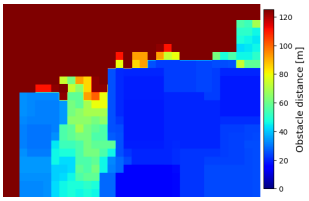
(b) Complete C-space



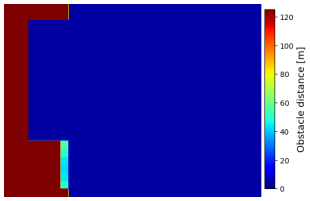
(c) Thresholded C-space (2.5x faster)



(d) Downscaled C-space (110x faster)



(e) Thresholded and Downscaled C-space (210x faster)



(f) Thresholded and Downscaled C-space default parameters

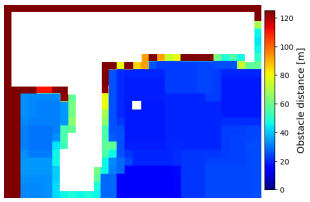
Figure 5: Comparing C-space efficiency measures results

goal- or current waypoint pixel must always be located inside the FoV, else the drone will hover and scan. The goal pixel is shown in white in the middle of each image. In this example, the goal is blocked by an obstacle at around 30 meters and a waypoint with a clearance of at least 35 meters will be selected. The first step is to identify the 'safe' pixels, as shown by the white mask in Figure 6a. Second, the mathematical erosion of this mask is performed, yielding the eroded mask in Figure 6b. Since the obstacle expansion in disparity space is extremely sensitive to noise, as shown in Figure 5f, its expansion radius can not be too large. The mathematical erosion is performed to ensure a large enough safety margin, in a noise-robust manner. Third, the edge of the eroded mask is calculated, shown in Figure 6c. To get the locally shortest path, the obstacle needs to be avoided with the smallest deviation possible. Hence, the best waypoint will be located on the edge.

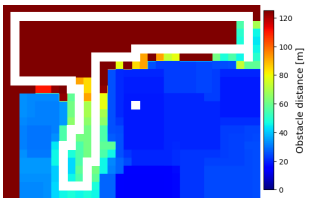
Since the Moore-Neighbor Tracing Algorithm requires a starting point for each boundary it traces, the novel method is preferred because it is able to find multiple safe areas. In addition, it makes it simpler to include an extra safety margin. Since a mask also makes no distinction between waypoints around or into concave obstacles, the fourth step is to look for sudden points. These are defined as pixels that have neighbouring pixels have a change larger than 20% in their depth value, and are shown in Figure 6d.

The last step is to find the best waypoint according to a cost function. For this, the obstacle boundary is discretised to pixel resolution: each pixel indicates a new possible waypoint. The cost function minimises the pixel distance to the sudden points within the eroded mask (if they exist) and the goal pixel, shown in Figure 6e. Wedgebug places the waypoint at a finite sensor range [Laubach and Burdick, 1999]. Frustumbug places the waypoint beside the obstacle it is avoiding, to allow for goal scanning upon arrival.

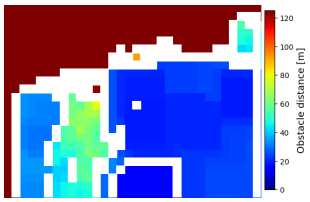
(a) Safe pixels mask



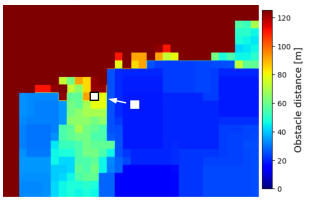
(b) Eroded mask



(c) Eroded mask edge



(d) Sudden points



(e) Selected waypoint

Figure 6: Waypoint selection procedure

3.3 *Finite-state machine - Wedgebug states*

Similar to Wedgebug and 3DBug, Frustumbug is also a finite-state machine. A simplified version is shown in Figure 7. The Wedgebug FSM is shown by the blue, green and yellow shapes, which are connected by solid lines. Frustumbug expands the FSM by adding reversing, climbing and descending states, indicated by the dashed grey lines. These can be activated from motion to goal, motion to waypoint or boundary following. The states with a solid boundary indicate a moving robot, whereas the dotted boundaries indicate a stationary robot, used for scanning. For the drone, stationary means hovering.

Wedgebug uses state ***Motion to Goal*** (*MtG*) to move the robot directly towards the goal. If the goal is reached, the algorithm halts. If an obstacle is detected, a new waypoint will be searched using the method described above. If found, state *MtG* transitions to state ***Motion to Waypoint*** (*MtW*). If not, it transitions to state ***Scanning Waypoint*** (*SW*). State *MtW* moves the robot to the waypoint and checks if state

MtG is possible when the waypoint is reached. If the path to the waypoint becomes unsafe because of a previously unseen obstacle, a new waypoint is searched. If this waypoint is not found, state *MtW* transitions to state *SW*. In other words, state *SW* is activated when there are no safe waypoints visible in the current wedge. The robot halts and scans additional wedges to find a new waypoint that can still decrease its distance to the goal. If no such waypoint can be found, state *SW* transitions to state ***Scanning Boundary*** (*SB*). State *SB* keeps scanning additional wedges until a new waypoint is found. If none are found, the goal is deemed unreachable. Otherwise, state *SB* transitions to state ***Boundary Following*** (*BF*), which moves the robot around the obstacle boundary until a transition to state *MtG* is possible, or until there exists a leaving point: a waypoint which has a smaller distance to the goal than the previously visited coordinates, making a transition to state *MtW* possible [Laubach and Burdick, 1999]. The algorithm stores the direction in which the obstacle boundary is being followed (CW/CCW) to prevent backtracking. If the goal is reachable, Wedgebug guarantees global convergence [Laubach and Burdick, 1999].

3.4 Finite-state machine - Changes to Wedgebug states

Since Frustumbug obtains depth information from imperfect sensor readings, it is not able to identify independent obstacle boundaries. In other words, it is not sure whether it has been following the boundary of the same obstacle, or if it has jumped to different ones (e.g. it is hard to identify which branch belongs to which tree in a dense forest). This causes Frustumbug to lose its global convergence guarantee. Moreover, if an unseen obstacle appears within a short time after transitioning from state *BF* into state *MtW*, it cannot be said with certainty if this is the same obstacle whose boundary it was just following, or if it is a new obstacle. If it is the same, it should follow the boundary in the same direction (CW/CCW). Therefore, the positive direction around an obstacle is remembered until a distance is travelled which exceeds a threshold, currently implemented as 20 meters.

Frustumbug is designed for a fixed stereo camera, hence it can not rotate its sensors like Wedgebug. Whenever it has to scan an extra wedge, it will rotate the drone around the vertical axis (yaw). To minimise the chance of crashing into obstacles while the camera is facing away from the direction of motion, this is only performed when hovering stationary. If the drone is moving, it has to come to a stop first. To transition from state *MtW* to state *MtG*, the drone first needs to stop and scan the goal. This is done via the state ***Scanning Goal*** (*SG*). Furthermore, it required a design change for state *BF*, since it cannot scan the goal direction while moving along the boundary. Frustumbug divides state *BF* into two states: ***Boundary Following - Waypoint*** (*BFW*) and ***Boundary Following - Turning*** (*BFT*). When state *SB* has found a waypoint, it transitions to state *BFW*, which moves the drone to the waypoint. Upon arrival, there is a transition to state *BFT*, which

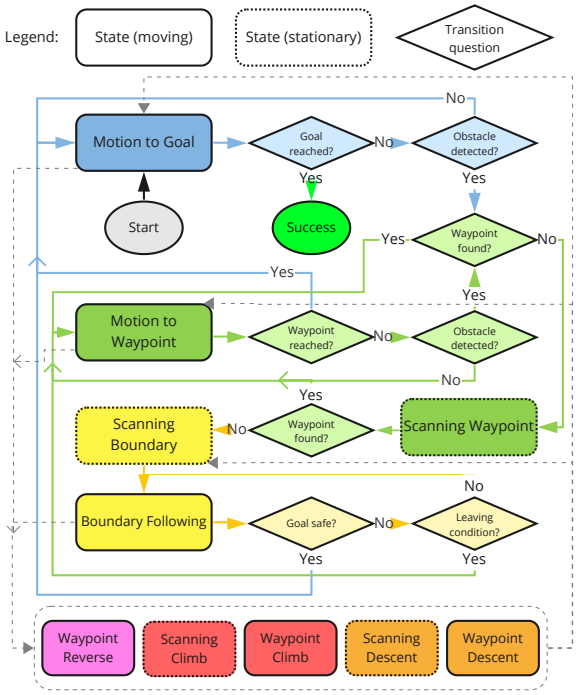


Figure 7: Simplified Frustumbug FSM

rotates the drone to scan the goal and/or the obstacle boundary to plan a new waypoint. The distance between the waypoints results from a trade-off between stopping too often, and missing opportunities where the goal direction was safe. Unfortunately, states *BFW* and *BFT* can lead to long paths if they miss safe goal direction opportunities and the continuous stopping, rotating and going can be time-demanding.

To decrease the frequency in which states *BFW* and *BFT* are activated, state *SW* is allowed to scan for waypoints that do not necessarily decrease the distance to goal, as long as they are within a 180 degrees FoV when facing the goal. Since Frustumbug already lost its global convergence guarantee due to being unable to identify independent obstacle boundaries, this rule is implemented to improve the overall performance. Further improvements are made by allowing state *SW* to always scan a left wedge and a right wedge, unless a fixed direction around the obstacle has been specified. This is implemented to prevent missing out on good opportunities to avoid the obstacle. Wedgebug would stop scanning as soon as it finds a waypoint in the left or right wedge [Laubach and Burdick, 1999].

To prevent looping around the same obstacle, Wedgebug has implemented a loop detection: when a location on the obstacle boundary is visited twice, a loop is detected [Laubach and Burdick, 1999]. A two-dimensional loop might not be found in a three-dimensional environment, since the shape of an object is not always constant over height, and the altitude of the drone can change during state *BFW*. Since Frustumbug

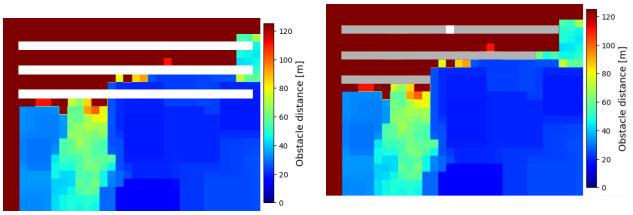
can not clearly distinguish independent obstacle boundaries, it is possible that it will follow the boundary of multiple obstacles. Hence, even if a loop is found, the manoeuvre can be time consuming. Therefore, state *MtG* is activated by state *BFT* once the *M-line* is crossed. Instead of using the *M-line* from start- to goal position, Frustumbug defines this line as the line between the start of state *SB* and the goal position, since the *M*-line does not play a role in the other states.

3.5 Finite-state machine - Additional states

To further decrease the frequency in which states *BFW* and *BFT* are activated, Frustumbug introduces the possibility to reverse the drone backwards along previously visited GPS points using state ***Waypoint Reverse*** (*WR*). This state looks for escape points that would steer the drone around the obstacle in an earlier stage, using different parameter settings. Perhaps it did not see the obstacle before due to a stereo mismatch. Just in case it is mismatched again, the obstacle location is remembered as a single pixel, which is redrawn in the incoming images while reversing. One pixel is enough since it will be expanded in disparity space to create the C-space. State *WR* is only activated if the previously visited GPS coordinates are located on a straight line, since the drone is going there blindly and turning radii can be different than on the way there.

Next, a distinction has been made regarding obstacle size when reaching local minima: the C-space could be blocked by a farther away large obstacle (note that these pixels are still unsafe) or a nearby smaller obstacle which caused the obstacle expansion to cover the entire FoV, similar to what happened in Figure 5f. In case of a small obstacle, it is preferred to reverse and avoid using state *WR*, or to find a new waypoint using state *SW*. However, for a large obstacle, a new waypoint may not be found by states *WR* or *SW*. Instead, the altitude is increased in an effort to avoid the obstacle by going over it.

The process for selecting the best waypoint for climbing is shown in Figure 8. Due to the limited FoV it is not possible to move straight up, hence the drone will climb using a small flight path angle. To limit the required memory and computation when scanning its surroundings, only three pixel rows are checked, shown in white in Figure 8a. In the 32x24 image, these are rows 3, 6 and 9. The waypoint will not be selected in the top rows (0, 1 and 2), since it would move out of FoV quickly when the drone pitches forward to move, and obstacle presence could not be checked. For each white row, the best pixel's column is stored if the number of safe pixels exceeds the threshold for a safe climb. The best pixel's column is the column with the largest distance from any obstacle in that row. Figure 8b shows the safe pixels in grey, as well as the best pixel in white. If a safe pixel exists in the highest row, it is chosen first, followed by the middle row and then the bottom row. Preference is given to the climbing directions that are closer towards the goal direction.



(a) The three scanned rows in white

(b) Waypoint for quickest and safest climb in white. Grey shows obstacle-free waypoints

Figure 8: Method for selecting the best waypoint for a climbing manoeuvre

The goal direction is always scanned first for a possible waypoint to climb. If not found, state ***Scanning Climb*** (*SC*) is activated. It will first scan the 180 degrees in the goal direction to find a suitable climbing waypoint. If no waypoints are found, it will scan the remaining 180 degrees. If a waypoint is found, state ***Waypoint Climb*** (*WC*) will move the drone towards this waypoint. Else, state *SB* is activated. A climbing manoeuvre always consists of at least 2 segments to minimise the deviation from the nominal path: halfway the desired height difference, a new waypoint will be searched. It is possible that Frustumbug uses more than 2 segments to obtain the desired height difference if a previously unseen obstacle blocks the waypoint in state *WC*. A new segment is then added, using the method described above.

The climbing manoeuvre is created to allow the drone to avoid obstacles by flying over them, followed by a descending manoeuvre. Therefore, the drone is instructed to keep flying at a higher altitude until the obstacle is passed. Since individual obstacle boundaries can not be distinguished, this is implemented as a minimal distance to be flown towards the goal at higher altitude. Once arrived, the goal direction is always scanned first for a possible waypoint to descend. If not found, state ***Scanning Descent Forwards*** (*SDF*) is activated. The method is similar to selecting climbing waypoints, but rows 14, 17 and 20 are scanned for descent.

State *SDF* only scans the 180 degrees FoV towards the goal. This state is normally used after the obstacle has been passed, because descending backwards could send the drone back to the wrong side of the obstacle. If no waypoint for descent is found, the drone will keep flying towards the goal at the same altitude for a few metres and transitions to state *SDF* again. Else, state ***Waypoint Descent*** (*WD*) will move the drone towards this waypoint. Upon arrival, the goal direction will be scanned to see if the goal is inside the FoV. If not, state ***Scanning Descent Backwards*** (*SDB*) will be activated, which will scan the 180 degrees in the opposite direction of the goal. If a waypoint exists here, state *WD* is activated to execute the path. Else, state *SDF* is activated, which will try to find a waypoint to descend the drone towards the goal.

In case the drone is located nearby and above the goal, or if state *BFT* noticed that the goal is located below its FoV, it

has no clear preference between states *SDF* or *SDB*. Hence, state ***Scanning Descent Either*** (*SDE*) is designed such that it will first scan in goal direction. If no waypoint is found, it will continue to scan until a waypoint is found or until a full revolution is completed, similar to what state *SC* does for climbing. A note to this: if the drone realises it is above an obstacle (e.g. a roof), it will keep moving towards the goal until this obstacle is passed. It attempts to identify this situation by checking if the pixels blocking the goal have a roughly linearly increasing depth value, since this is usually the case when above buildings or large trees.

In total, the drone can adapt to 14 different states on its way to the goal position. A complete overview of each state individually can be found in Appendix G, as well as the possible state transitions. We are aware that these FSMs can come across as quite elaborate. However, in complex environments there are many possible failure modes which need to be accounted for, and we would like to present the states in detail to keep a clear overview of the drone's behaviour during flight. Note that each path planning decision can be changed easily, allowing for quick user-preferred changes or upgrades.

4 Experimental Results

The proposed algorithm has been tested both in simulation and on a real drone. First, results from simulation environments UrbanCity and Forest are presented, followed by the the real world results. A number of start- and goal positions are generated for both UrbanCity and Forest environments. The generated paths are stored, from which the success rate and the path length can be obtained.

4.1 Simulation UrbanCity

UrbanCity is an environment built in Unreal Engine 4. Open source project UnrealCV included several commands to interact with the environment. The environment simulates a urban city corner and contains buildings, trees, roads and street signs. Start- and goal positions are chosen throughout the environment: 11 on ground level and 3 on top of buildings. These 14 locations yield 182 possible paths to be travelled. Since UnrealCV is only able to set static camera poses, drone dynamics are not included. To navigate through the environment, the camera location is moved by 1 metre in the direction of the waypoint each time. Only the yaw angle is changed, not pitch nor roll.

In total, 177 out of 182 goals were reached successfully, leading to a success rate of 97.3 %. Figure 9 shows all the successful paths. Each path has the same color as its goal, however since there are only 7 colours for 14 goals, every color appears twice. The 5 unsuccessful paths are shown in Figure 10, of which the first two are shown in Figure 10a. Here, the crosses represent the starting points, the dots are where it failed and the stars are the goals. The black line used state *WC* to avoid an obstacle at higher altitude near the end. Upon arrival, the goal was still inside FoV near the bottom, but was

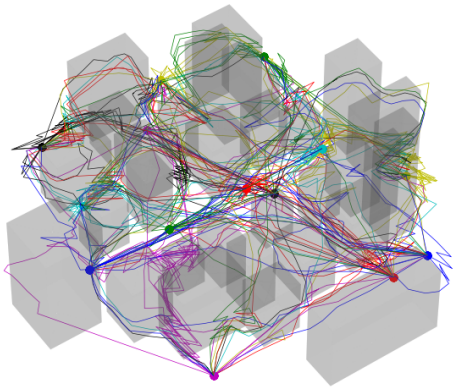
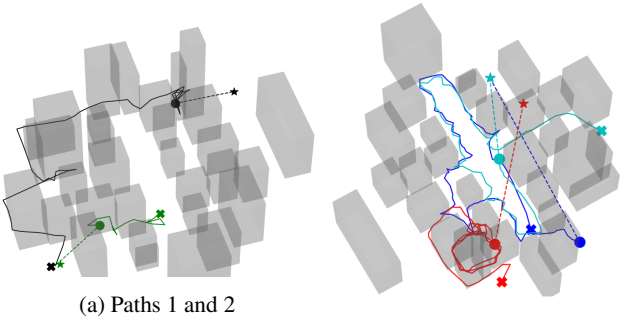


Figure 9: All successful paths in UrbanCity. The paths are colour coded similar to their goal position.

blocked by the top of the building, hence state *MtW* was activated. A small distance later, the goal went out of FoV and state SDE was activated, which found a waypoint in the direction it just came from: in the front of the building. The drone got stuck in this loop until a timeout was reached. The green line collided with a building, due to a temporary change in value for the uniqueness ratio parameter, as explained in the last paragraph of subsection 3.1. After a turn, the drone was facing a different obstacle, which could not be detected using the temporarily increased uniqueness ratio. However, without this change in the uniqueness ratio parameter, dozens of simulations got stuck. Corresponding onboard RGB images and depth maps are shown in Appendix C.



(a) Paths 1 and 2

(b) Paths 3, 4 and 5

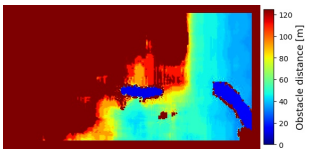
Figure 10: All unsuccessful paths in UrbanCity. A cross represent a starting point, a star represent the goal and the dot is where the simulation failed.

The three other unsuccessful flights, shown in Figure 10b, all ended in state *BFT* or *BFW* at the timeout. The blue and cyan lines could not find a path in between the buildings and continued to follow the boundary of a growing group of obstacles. The red line initially starts boundary following almost from the start, but gets stuck in a loop around one of the higher buildings. All three did not cross their respective

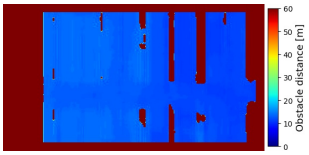
M-lines. They initially did not start climbing because they were looking into small alleys, meaning that the minimum percentage of nearby pixels was not reached.

(a) RGB image light pole

(b) RGB image building



(c) Depth image light pole



(d) Depth image building

Figure 11: Examples of nearby (light pole) and farther away (building) obstacles, both blocking the C-space. The light pole would trigger a transition to either state *SW* or *WR*, depending on the current state. The building would trigger a transition to state *SC*.

In subsection 3.5 a distinction between small and large obstacles was made, which would both have the entire FoV covered after obstacle expansion. Figure 11 shows an example to explain both possibilities using on-board data. Figure 11a shows a nearby light pole and Figure 11b shows a farther away building. This is supported by their depth maps: Figure 11c shows only a few nearby pixels in blue, whereas Figure 11d shows a large obstacle. After obstacle expansion, the nearby pixels cover the complete FoV for both. To transition to state *SC*, the percentage of nearby pixels in the depth image needs to exceed a threshold (currently 80 percent), else there will be a transition to either *SW* or *WR*, depending on the current state. For Figure 11a, state *SW* was activated, because the camera had just turned and there was no straight line segment, so the transition to state *WR* was not allowed. For Figure 11b, state *MtG* transitioned into state *SC*.

The path lengths are compared to their default path length. Note that the default path is not equal to the global optimal path. Since it is complicated to get the globally shortest path out of UrbanCity, the default path length is defined by the path length that is obtained if the drone would simply climb high enough, fly over all obstacles, and descend to goal, using the FoV restricted climbing manoeuvre. On average the path lengths are 75.8 % of their default path lengths, meaning that Frustumbug has found an average shortcut of 24.2 %. Results are shown in Figure 12. All 6 flights exceeding 115 % have activated states BFW and BFT during flight.

4.2 Simulation Forest

The open-source simulator AirSim is developed by Microsoft and contains an outdoor environment called

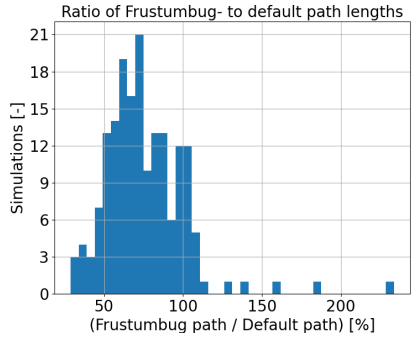


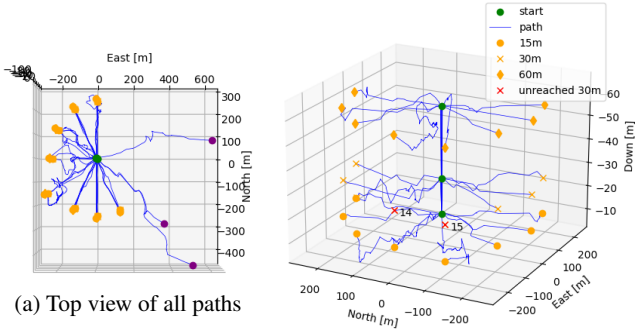
Figure 12: Frustumbug's path lengths histogram

Forest. It includes many trees, branches, bushes, rocks and hills, and is shown in Figure 13. The green dot indicates the starting position, and the 8 orange dots indicate the goal positions. Some of the goals are located near trees, but all are reachable. The paths are flown at 3 different altitudes, and from low to high altitude and vice versa, resulting in 40 paths. The area with the purple dots consists of fewer trees and more hills, hence it is used to test Frustumbug's ability to deal with ground elevation changes. In total, 43 paths are generated. The arrows indicate that the waypoint location is outside of the image. Drone dynamics are included in this simulation, making it more representative for outdoor flight.

Figure 13: Forest environment. Green dot represents the start position, the orange and purple dots show all the goal positions. Arrows indicate they are off the map.

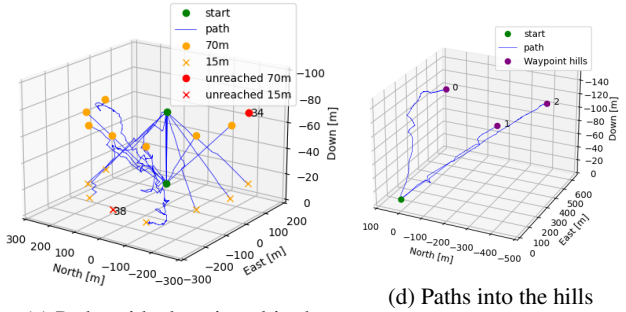
39 out of 43 goals were reached successfully, leading to a success rate of 90.7 %. Figure 14 shows all the successful paths and Figure 15 shows the 4 unsuccessful paths. Figure 14a starts with the top view. Note that the left side of this plot is the most cluttered part of the environment, with a large number of trees located close together. Figure 14b shows the 24 paths where start- and goal position have the same altitude. Two goals (14 and 15) are not reached, both in the cluttered area. Figure 14c shows the 16 paths that go from low to high altitude and vice versa. Here, two goals (34 and 38) were not reached, of which one was located in the cluttered area. Figure 14d shows the three waypoints far into the forest, in the

area containing hills. All three were reached by the proposed algorithm. More data, including onboard images during the flight, are presented in Appendix E.



(a) Top view of all paths

(b) Paths at constant altitude



(c) Paths with changing altitude

(d) Paths into the hills

Figure 14: All successful flights in Forest

Ideally, the generated path lengths are compared to the globally shortest path lengths. However, calculating these would require accurate obstacle data, which is not available. Furthermore, the climbing and descending method presented in UrbanCity would yield unrealistically long default paths, since the trees are generally not avoided by climbing over them. Therefore, the paths had to be analysed qualitatively. It was found that any deviation it took from the nominal path was supported by obstacle presence.

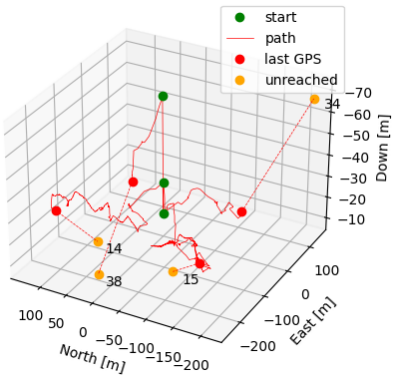


Figure 15: Four unsuccessful paths in Forest

The four unsuccessful paths are caused by three different reasons. The paths shown in Figure 15 belong to the red markers in Figure 14b and Figure 14c. Paths 34 and 38 both got stuck for the same reason: states *BFW* and *BFT* did not find an escape point in any of the wedges. Path 14 was working its way around the trees, but got stopped by a timeout. Path 15 transitioned many times from state *BFT* to state *MtW*, and from state *MtW* back to *SB* (followed by *BFW* and *BFT*). This was because the positive direction around the obstacle boundary was forgotten by the time the transition to *SB* happened. In other words, the threshold explained in subsection 3.4 was too low for this run, since it kept going around the same group of obstacles in different directions, until the timeout was reached.

4.3 *Real World*

For real world testing, two JeVois[3] cameras were connected to one ARM-A7 processor, responsible for the navigation. The setup is shown in Figure 16, where the stereo camera is shielded in tin foil to block the electromagnetic radiation.

Figure 16: Experimental setup: JeVois stereo camera mounted to the bottom of the drone.

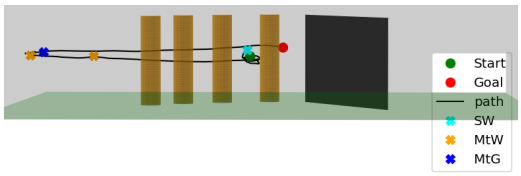
The testing is performed in the Cyber Zoo facility at the faculty or Aerospace Engineering at the TU Delft. The drone obtains accurate position and orientation estimates via the motion capture system Optitrack. In the experiment shown in Figure 17, obstacles are placed in a V-shape to create a local minimum.

Figure 17: Real world testing environment: Cyber Zoo facility at the TU Delft. Obstacle are positioned to create a local minimum.

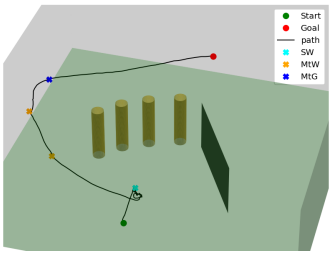
The results are shown in Figure 18, where Frustumbug has created a safe path around the obstacles. Figure 18a uses

[3]http://jevois.org/

a low elevation angle to show that the height is approximately constant throughout the flight. Figure 18b shows a better perspective of the generated path. The path starts at the green dot, and the drone moves to goal using state *MtG*. Once the obstacle is detected, the drone stops and hovers, because a waypoint is not found. State *SW* (cyan cross) starts scanning for a waypoint, and on the second left scan, a waypoint is found and the drone moves there. At both the orange crosses an obstacle (the Cyber Zoo edge) is detected and a new waypoint is found while in state *MtW*. At the dark blue cross, the waypoint reached and state *MtG* moves the drone towards the goal. A waypoint above the obstacles was not found due to the limited vertical FoV. More results, including on-board images and decision making, are included in Appendix F.



(a) Low elevation angle



(b) High elevation angle

Figure 18: Result of a Cyber Zoo flight test. State *SW* finds a waypoint around the group of obstacles. State *MtW* steers clear of the Cyber Zoo edge. State *MtG* moves the drone towards the goal.

Another experiment, where the drone only had to avoid one obstacle, is shown in Figure 19. It starts at the green dot in state *MtG*, and the obstacle is detected at the orange cross. However, in this experiment, a waypoint is immediately found, hence no transition to state *SW* is necessary. State *MtW* moves the drone towards the waypoint. Upon arrival, the state transitions to *MtG* to complete the last segment to goal. Note that these real world tests do not test all the possible state transitions. Since the simulated environments have already proven that the algorithm can reach its goal through challenging environments, the main focus of the real world tests is to show its feasibility on resource limited systems.

5 Discussion

The results have shown that Frustumbug is able to reactively plan its path from start- to goal position with an acceptable path length, using limited computation and memory.

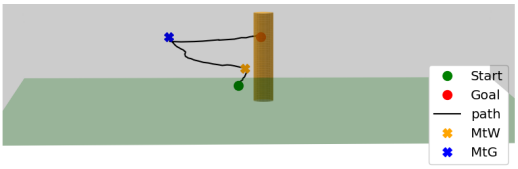


Figure 19: Result of a simple Cyber Zoo flight test. In state *MtG*, the obstacle is detected and a waypoint is found. State *MtW* moves the drone to the waypoint and a transition to state *MtG* follows.

Previous research had various simplifications and limitations. 3DBug assumed that the range sensor has perfect readings over a 360 degrees FoV and that all obstacles could be modelled as polyhedral. In addition, it was only tested in simulation. Other studies on reactive path planning in 3D did not explicitly mention escaping local minima or used randomly generated sub-optimal paths to escape. Frustumbug has been demonstrated in complex, realistic environments. It has found solutions for the assumptions that do not hold in these environments, and escapes local minima by taking logical decisions. The strengths of the proposed algorithm are the high success rate and the small size, weight and power. Furthermore, every path planning decision is made according to a large decision tree. This means that any choice made by the algorithm can be examined afterwards and changed individually. Moreover, the stereo matching parameters can be altered mid-flight if the incoming image shows signs of low texture or highly repetitive texture.

The limitations of the algorithm come in twofold, which will be discussed along with a few recommendations.

First, the choice of using a stereo camera with a small FoV limits the drone's local environment information. For example during boundary following, the drone has to stop often to scan extra wedges. Furthermore, it results in small climbing and descending angles in states *WC* and *WD*. Increasing the FoV is a possible solution, but comes with issues. Increasing the FoV while keeping the same image resolution will decrease the focal length, which decreases the largest depth that can be measured. The stereo base could be increased to counteract, but for small drones this is not ideal. To keep a constant focal length, the image size needs to be increased, but this will increase the computation. Another solution is to allow separate rotation of the stereo camera, similar to Wedgebug. However, this would introduce more moving components that can fail, and puts more requirements on drones which would like to use Frustumbug. A third solution would be the use additional lightweight sensors, for example an HC-SR04 SONAR. Since it has an extremely small FoV it could not be used for normal path planning, but it could scan to see if there is an obstacle above the drone. If not, the FoV limited climbing manoeuvre can be avoided. Pointing a stereo camera up would be excessive, since detailed infor-

mation about obstacles above the drone is not required. The SONAR requires only 15 mA (JeVois: 800 mA), but adds an extra weight of 8.7 gram per sensor [4].

Second, the c-space expansion could hide any escape points in narrow streets, or local areas with a high obstacle density. Future work could focus on running two instances of the c-space algorithm, with both a small and large expansion radius. The small expansion radius could be used temporarily to escape the situations described above. However, the larger expansion radius is the default for path planning, to keep a big enough safety margin from obstacles for changes caused by wind or GPS inaccuracy.

Further improvements can be found when looking at the results. The black path in Figure 10a failed because it did not identify that it was above an obstacle that needed to be passed, as described in subsection 3.5. In this particular flight, the drone was squeezed between two buildings of different heights, and no linearly increasing depth values were found in the direction to the goal. Future work could focus on designing an improved version of this function. The same rationale applies to the green path: it failed because the trick to ignore repetitive texture caused it to not recognise another building. Future work could use image characteristics to have a variable parameter set for the stereo matching algorithm. Some on-board examples are shown in Appendix C.

Looking at Figure 10b, improvements can be made in the decision for state transition. For example, state *SB* is activated when the obstacle is relatively small (and state *SW* could not find a waypoint), but once state *BFW* or *BFT* realises that the obstacle is in fact quite large, the state should switch to state *SC* and not stay in *BFW* or *BFT*, as was seen in Figure 10b. The same figure also shows a red path getting stuck in a loop around the same building, hence implementing something equivalent to a 3D loop detection seems useful after all.

6 Conclusion

Frustumbug is a cheap, lightweight three-dimensional path planning package for small drones. It reaches more than 90% of its goals in complex environments, is robust to noisy range sensor data and runs smoothly on a 20 gram stereo vision system with limited memory and computation. Furthermore, it has been extensively tested, both in simulation and real-world.

Our paper takes a step towards a publicly available path planning package. Since it does not rely on heavy processing power, it will be less demanding for the battery and onboard computer, increasing valuable flight time.

References

[1] N Aswini, E Krishna Kumar, and SV Uma. Uav and obstacle sensing techniques–a perspective. *International Journal of Intelligent Unmanned Systems*, 2018.

[2] Larry Matthies, Roland Brockers, Yoshiaki Kuwata, and Stephan Weiss. Stereo vision-based obstacle avoidance for micro air vehicles using disparity space. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 3242–3249. IEEE, 2014.

[3] Junseok Lee, Xiangyu Wu, Seung Jae Lee, and Mark W Mueller. Autonomous flight through cluttered outdoor environments using a memoryless planner. In *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1131–1138. IEEE, 2021.

[4] Oualid Doukhi and Deok Jin Lee. Deep reinforcement learning for autonomous map-less navigation of a flying robot. *IEEE Access*, 10: 82964–82976, 2022.

[5] Ricardo Bedin Grando, Junior Costa de Jesus, Victor Augusto Kich, Alisson Henrique Kolling, and Paulo Lilles Jorge Drews-Jr. Double critic deep reinforcement learning for mapless 3d navigation of unmanned aerial vehicles. *Journal of Intelligent & Robotic Systems*, 104 (2):1–14, 2022.

[6] Yang Yu, Wang Tingting, Chen Long, and Zhang Weiwei. Stereo vision based obstacle avoidance strategy for quadcopter uav. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 490–494. IEEE, 2018.

[7] Sharon L Laubach and Joel W Burdick. An autonomous sensor-based path-planner for planetary microrovers. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 1, pages 347–354. IEEE, 1999.

[8] Ishay Kamon, Ehud Rivlin, and Elon Rimon. 3dbug: A three-dimensional range-sensor based globally convergent navigation algorithm. Technical report, Computer Science Department, Technion, 1996.

[9] Tom van Dijk. Self-supervised learning for visual obstacle avoidance. Technical report, Micro Air Vehicle Lab (MAVLab), TU Delft, mar 2020. Technical report.

[10] Kimberly McGuire, Guido De Croon, Christophe De Wagter, Karl Tuyls, and Hilbert Kappen. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters*, 2(2):1070–1076, 2017.

[11] Hua Liu, Rui Wang, Yuanping Xia, and Xiaoming Zhang. Improved cost computation and adaptive shape guided filter for local stereo matching of low texture stereo images. *Applied Sciences*, 10(5):1869, 2020.

[12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3354–3361. IEEE, 2012.

[13] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1):7–42, 2002.

[14] Heiko Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814. IEEE, 2005.

[15] Alexios Lyrakis. Low-cost stereo-based obstacle avoidance for small uavs using uncertainty maps. 2019.

[16] Suyog Patil, Joseph Simon Nadar, Jimit Gada, Siddhartha Motghare, and Sujath S Nair. Comparison of various stereo vision cost aggregation methods. *International Journal of Engineering and Innovative Technology*, 2(8):222–226, 2013.

[4] https://www.adafruit.com/product/3942

[17] Taha Elmokadem and Andrey V Savkin. Towards fully autonomous uavs: A survey. *Sensors*, 21(18):6223, 2021.

[18] V. Lumelsky and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063, 1986. doi: 10.1109/TAC.1986.1104175.

[19] P Rajashekar Reddy, V Amarnadh, and Mekala Bhaskar. Evaluation of stopping criterion in contour tracing algorithms. *International Journal of Computer Science and Information Technologies*, 3(3):3888–3894, 2012.

[20] Norlida Buniyamin, W Wan Ngah, Nohaidda Sariff, Zainuddin Mohamad, et al. A simple local path planning algorithm for autonomous mobile robots. *International journal of systems applications, Engineering & development*, 5(2):151–159, 2011.

[21] Min Cheol Lee and Min Gyu Park. Artificial potential field based path planning for mobile robots using a virtual obstacle concept. In *Proceedings 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003)*, volume 2, pages 735–740. IEEE, 2003.

[22] Joe Sfeir, Maarouf Saad, and Hamadou Saliah-Hassane. An improved artificial potential field approach to real-time mobile robot path planning in an unknown environment. In *2011 IEEE international symposium on robotic and sensors environments (ROSE)*, pages 208–213. IEEE, 2011.

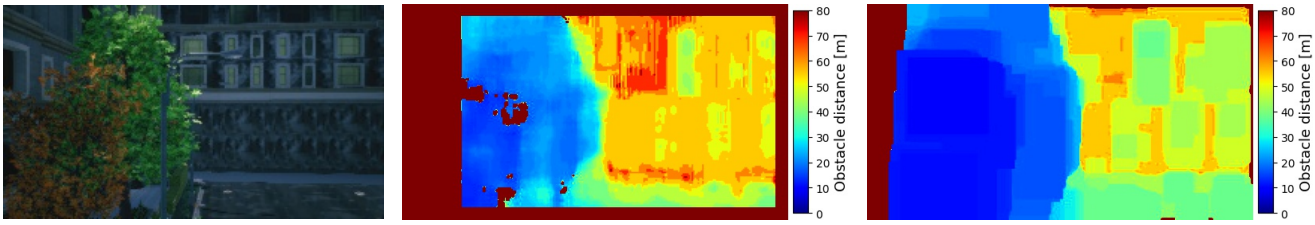
[23] Helen Oleynikova, Dominik Honegger, and Marc Pollefeys. Reactive avoidance using embedded stereo vision for mav flight. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 50–56. IEEE, 2015.

APPENDIX A C-SPACE THRESHOLDS

To show the effect of the thresholds used for creating the C-space, a few examples are shown below. These four images are chosen since they have both nearby and farther away obstacles.

For each figure, subfigures (a), (b) and (c) show the RGB image, the generated disparity map using BM with optimised parameters and the complete C-space (i.e. without thresholding or downscaling). Subfigures (d), (e), (f) and (g) show thresholds at 10, 20, 30 and 40 metres respectively. Any pixels with depth values larger than these threshold will not be expanded in disparity space. The time in between the brackets indicates runtime, averaged over 3 runs. Note that the thresholded images are also downscaled.

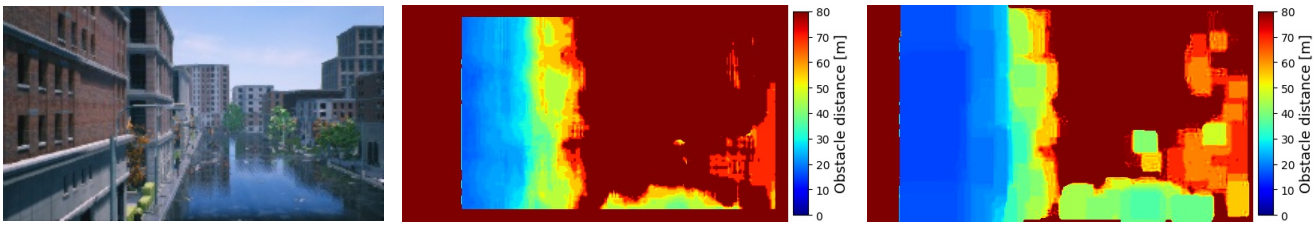
Since some of Frustumbug's states require obstacle information at more than 20 metres (e.g. *SC*), the threshold needed to exceed 20. The selected threshold is 30 meters, since obstacles at a distance between 30 and 40 meters distance are not used by Frustumbug's reactive planning.



(a) RGB image nearby tree (b) Depth image (c) Complete C-space (257 msec)

(d) Threshold at 10m (11 msec) (e) Threshold at 20m (23 msec) (f) Threshold at 30m (25 msec) (g) Threshold at 40m (28 msec)

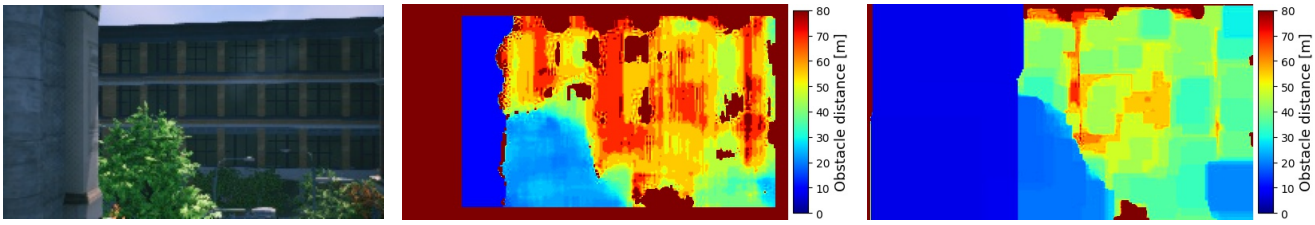
Figure 20: An example from on-board footage, used to show the influence of the obstacle expansion threshold. It shows two nearby trees, and a farther away building. Any pixels with depth values above the threshold are not expanded to improve runtime. Frustumbug uses a threshold of 30m.



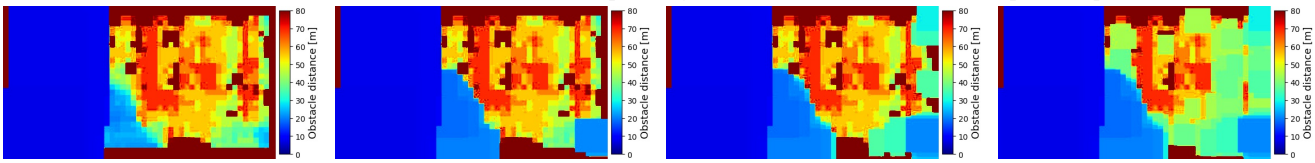
(a) RGB image buildings along road (b) Depth image (c) Complete C-space (212 msec)

(d) Threshold at 10m (17 msec) (e) Threshold at 20m (19 msec) (f) Threshold at 30m (20 msec) (g) Threshold at 40m (22 msec)

Figure 21: An example from on-board footage, used to show the influence of the obstacle expansion threshold. It shows the side of a building, slowly disappearing in the distance. Any pixels with depth values above the threshold are not expanded to improve runtime. Frustumbug uses a threshold of 30m.

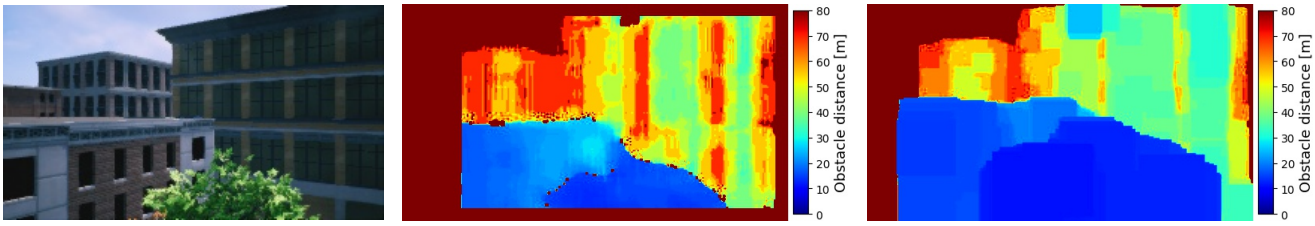


(a) RGB image nearby tree and building (b) Depth image (c) Complete C-space (224 msec)

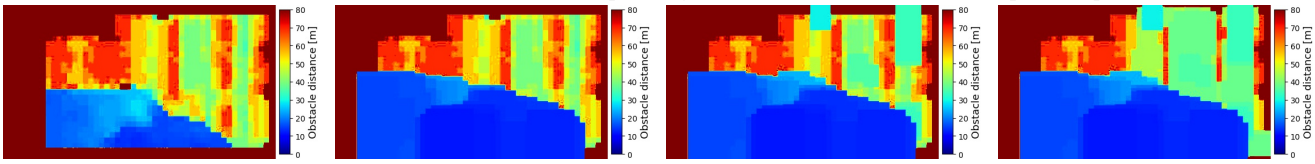


(d) Threshold at 10m (20 msec) (e) Threshold at 20m (26 msec) (f) Threshold at 30m (26 msec) (g) Threshold at 40m (30 msec)

Figure 22: An example from on-board footage, used to show the influence of the obstacle expansion threshold. It shows a nearby building and tree, and a large building farther away. Any pixels with depth values above the threshold are not expanded to improve runtime. Frustumbug uses a threshold of 30m.



(a) RGB image above tree, near building (b) Depth image (c) Complete C-space (225 msec)



(d) Threshold at 10m (12 msec) (e) Threshold at 20m (21 msec) (f) Threshold at 30m (23 msec) (g) Threshold at 40m (29 msec)

Figure 23: An example from on-board footage, used to show the influence of the obstacle expansion threshold. It shows the drone passing over a tree and a building, with another large building in the background. Any pixels with depth values above the threshold are not expanded to improve runtime. Frustumbug uses a threshold of 30m.

APPENDIX B OPTIMISING STEREO BLOCK MATCHING PARAMETERS

The parameters used in the stereo block matching algorithm are the following:

- MinDisparity (MD) - minimum possible disparity.
- NumDisparities (ND) - maximum possible disparity. A high value allows nearby obstacles to be matched, but it increases computation.
- BlockSize (BS) - sliding window size. High values give smooth, but possibly inaccurate images. Low values give detailed, but possibly noise images.
- SpeckleWindowSize (SWS) - maximum group size for pixels to still be considered a speckle.
- SpeckleRange (SR) - maximum range between pixel values to be considered the same group.
- Disp12MaxDiff (LR) - maximum allowed difference in the left-right consistency check.
- PreFilterType (PT) - either NormalizedResponse or XSobel
- PreFilterSize (PS) - pre-filtering size for NormalizedResponse
- PrefilterCap (PC) - clip value if outside range [-cap, cap]
- TextureThreshold (TXT) - minimum texture required within one BlockSize. Relates quadratically to BlockSize.
- UniquenessRatio (UR) - a high ratio means the best disparity match must be a unique match, i.e. SAD(d) >= SAD(d) $\cdot$ (1 + UR/100) for any other disparity d within NumDisparities
- SmallerBlockSize (SBS) - Unused/not defined in source code
- ROI1 and 2 - Region Of Interest left and right image

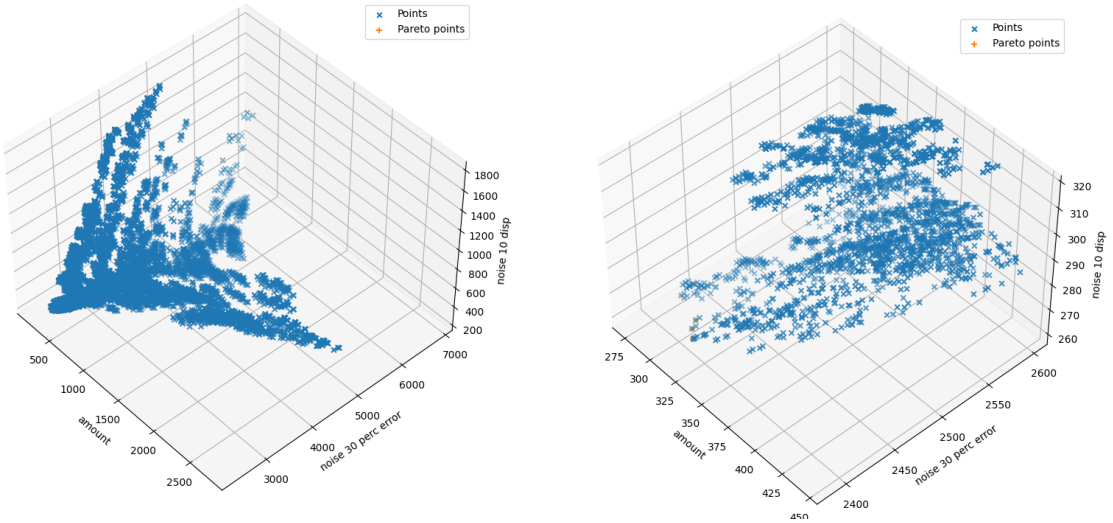
In addition, morphological opening (OP) has been added, in an effort to further reduce the noise in the image. Grey-scale opening consists of two stages: first the image is eroded, then dilated. The window size used for these operations is the parameter that will be optimised. Table 1 shows for each parameter its default value, whether it is included in the optimisation, what values are used to test the parameter and the implemented.

Table 1: Default and optimised values for each stereo BM parameter

	Default	Optimised	Tested range	Implementation
MD	0	No	-	0
ND	64	Yes	-	32
BS	21	Yes	[5-21]	17
SWS	0	Yes	[0-150]	150
SR	0	Yes	[0-14]	14
LR	-1	Yes	[-1-10]	14
PT	XSobel	No	-	XSobel
PS	9	No	-	9
PC	31	No	-	31
TXT	10	Yes	[0-5780]	1156
UR	15	Yes	[0-45]	2
SBS	0	No	-	0
ROI	(0,0,0,0)	No	-	(0,0,0,0)
OP	0	Yes	[0-7]	0

A grid search is used to test all parameter combinations on a dataset containing 100 images. These images are taken from the Forest environment and include a mix of obstacles: trees, branches, bushes, rocks, ground, etc. Each parameter combination generates a disparity image, which is expanded to create the C-space. The generated C-space is then compared to the C-space resulting from ground-truth depth data. The generated C-space is evaluated on completeness, accuracy and noise. Firstly, completeness is quantified by the amount of pixels that did not get a disparity value, from which the true disparity value is higher than 2 pixels. Secondly, accuracy is quantified by the amount of pixels that give a depth estimate within a 30% error. Note that pixels with a disparity value lower than 2 pixels are not considered here either. Using a 60 degrees FoV, 256x144 resolution and a stereo base of 0.25m, a 2 pixel disparity gives a depth of 27.6m. A 30% error within this depth range could be crucial, whereas inaccurate depth values of above 27.6m have small consequences to the algorithm. Thirdly, noise is quantified by the amount of pixels that were given a disparity error of more than 10 pixels. For all three metrics, only the pixels inside the stereo BM frame are considered. This frame excludes the first 31 columns, since stereoBM can not match these due to the NumDisparities being equal to 32. In addition, it excludes half the blocksize around the whole contour of the image. All three objectives are minimised. Testing all possible parameter combinations would roughly take a couple of years to run. Instead, each variable is assigned a small array of values. For example, SR would be given the array [3, 5, 7, 10], and SWS the array [55, 75, 95, 115].

The results of the first run are shown in Figure 24. Figure 24a shows the result of all 11520 parameter combinations. Each data point shows the average value of the parameter combination from 100 images. Since all objectives need to be minimised, the best solutions appear on the left side of the graph. The pareto front forms a V-shape, stretching from this corner both up and to the right. Figure 24b shows a more detailed zoom of the left corner.

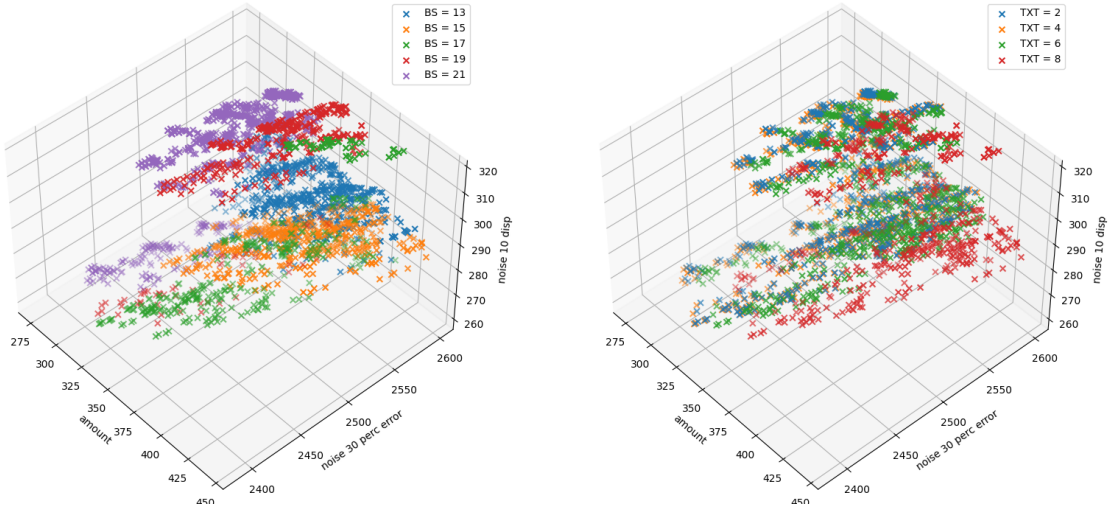


(a) All 11520 parameter combinations. The best combination is found in the left corner, where all objectives are minimised.

(b) Zoom of the best parameter combinations in the left corner.

Figure 24: Results of each stereo BM parameter combination averaged over 100 images

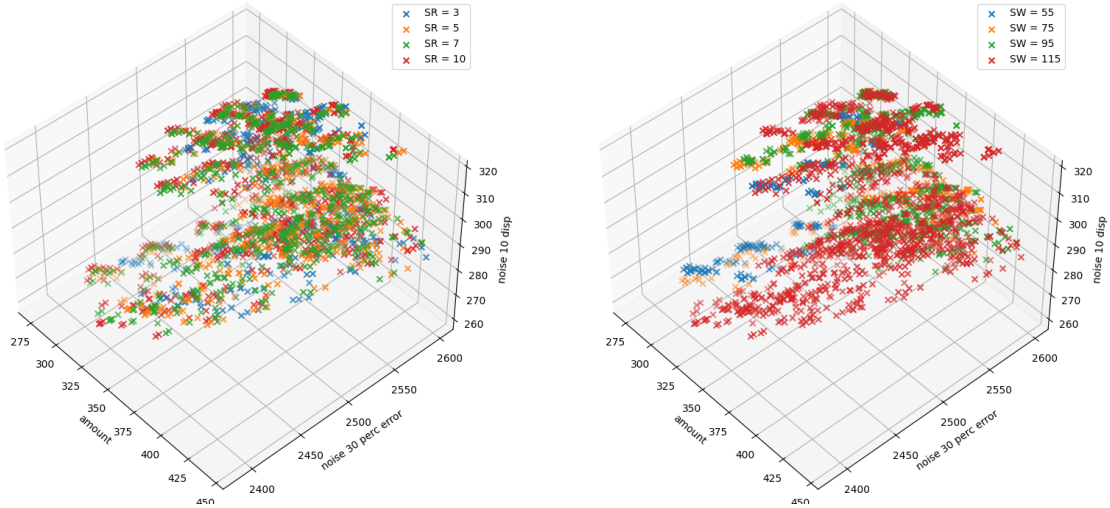
The zoom of the best parameter combinations is analysed in more detail. Figure 25a shows how BS influences the results. From BS 13 to 17 there is an increase in performance, which stagnates around BS 17 and 19. The best results of BS 21 show a slightly improved completeness, but perform worse in accuracy and noise. Figure 25b shows how TXT influences the results. A TXT of 8 shows the worst results, followed by a TXT of 6. There is no clear difference between the other two. The actual values used for the texture threshold are equal to TXT x BS^2. This is because a TXT of 50 at a BS of 5 had the same influence as a TXT of 450 at a BS of 15.



(a) BlockSize (b) TextureThreshold

Figure 25: Results showing influence of BlockSize and TextureThreshold. The best parameters are located on the left, where all the values on all three axes are minimised.

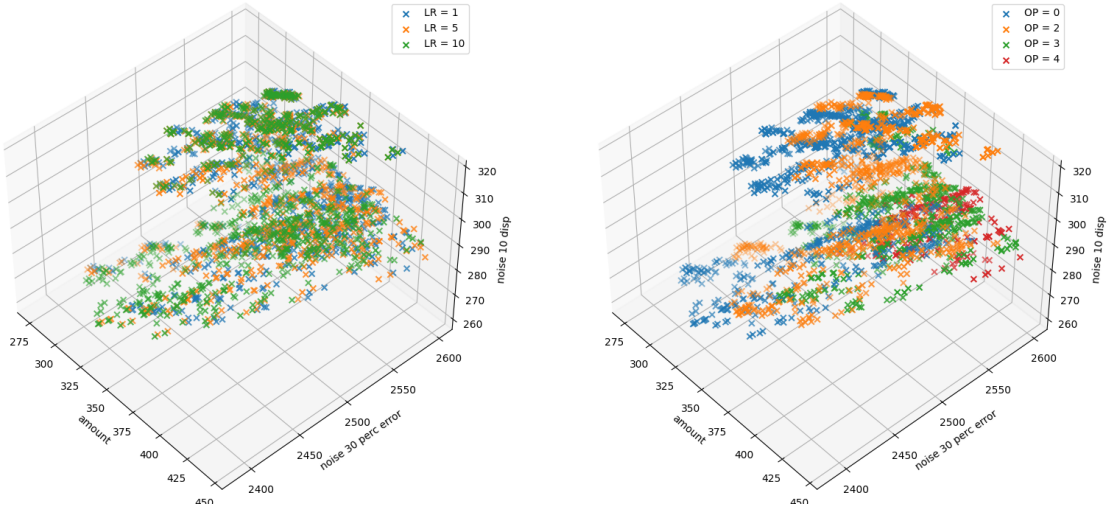
The same process takes place for the other parameters. From Figure 26a it can be concluded that values 3 and 5 perform worse for SR than 7 and 10, but it might be that there exists a value above 10 that could improve performance. Figure 26b shows a preference towards higher SWS values, although there are also a few combinations using 55 and 75 that still show good results.



(a) SpeckleRange (b) SpeckleWindowSize

Figure 26: Results showing influence of SpeckleRange and SpeckleWindowSize. The best parameters are located on the left, where all the values on all three axes are minimised.

Lastly, the results are shown for LR and OP in Figure 27. Figure 27a shows that a higher LR values yields better performance and Figure 27b shows that any other value than 0 for OP yields worse performance.



(a) Left-Right consistency check

(b) Morphological Opening

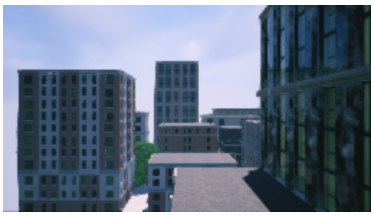
Figure 27: Results showing influence of Left-Right consistency check and Morphological Opening. The best parameters are located on the left, where all the values on all three axes are minimised.

From these observations, a subsequent run slightly adjusts the values of the parameters in promising directions. The final set of parameters used for the stereo block matching algorithm , as presented in Table 1, resulted after 6 iterations.

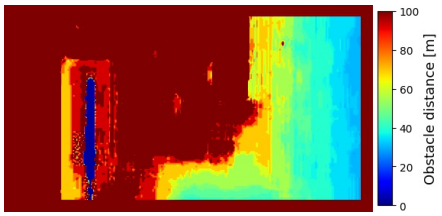
APPENDIX C VARIABLE BM PARAMETERS

In a few specific cases in environment UrbanCity, the drone gets stuck due to incorrect stereo matching. The examples presented here show obstacles with repetitive texture. It is included to show that changing the Uniqueness Ratio threshold can get the drone out of tricky situations, and to show that it does not have the perfect response to every possible situation. In future work, a variable parameter set could be optimised based on the characteristics inside an image (e.g. repetitive texture, low texture or obstacle size).

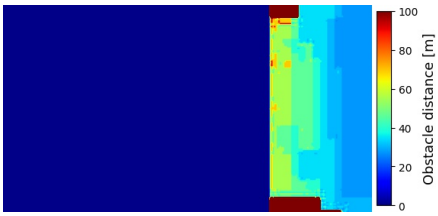
Figure 28 shows what the majority of the results looked like: the noise is removed and path planning is made possible again. However, Figure 29 shows that for some cases, it can delete many pixels. This can make it difficult to check if the goal direction is safe in state *MtG*, or to find new waypoints for state *MtW*, since it requires the new waypoint to have a disparity value assigned to it. On the other hand, an example where the Uniqueness Ratio threshold of 105 is still too low is shown in Figure 30. The approach did not completely solve the problem, however the drone was able to find a waypoint. Figure 31 shows an example where the Uniqueness Ratio threshold was temporarily changed, but the drone turned and did not identify the obstacle and crashed.



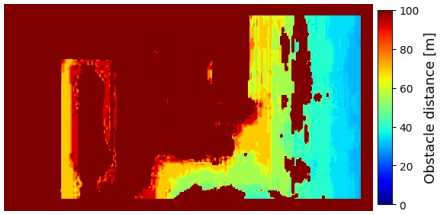
(a) RGB repetitive texture building 1



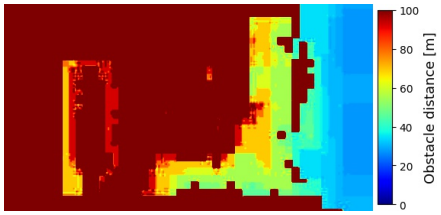
(b) Depth image (UR = 2)



(c) C-space (UR = 2)



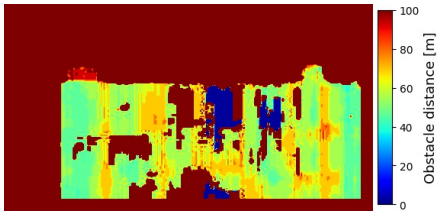
(d) Depth image (UR = 105)



(e) C-space (UR = 105)

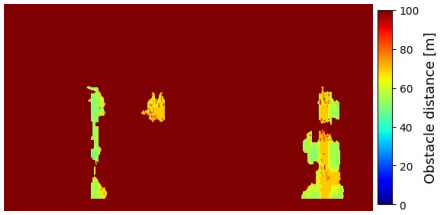
Figure 28: Increasing the Uniqueness Ratio (UR) removes the incorrect stereo matches, and allows the drone to escape the current location.

(a) RGB repetitive texture building 2

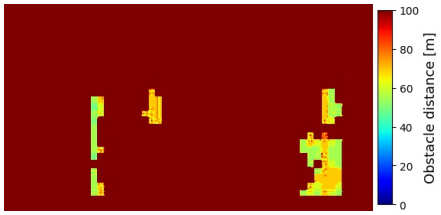


(b) Depth image (UR = 2)

(c) C-space (UR = 2)

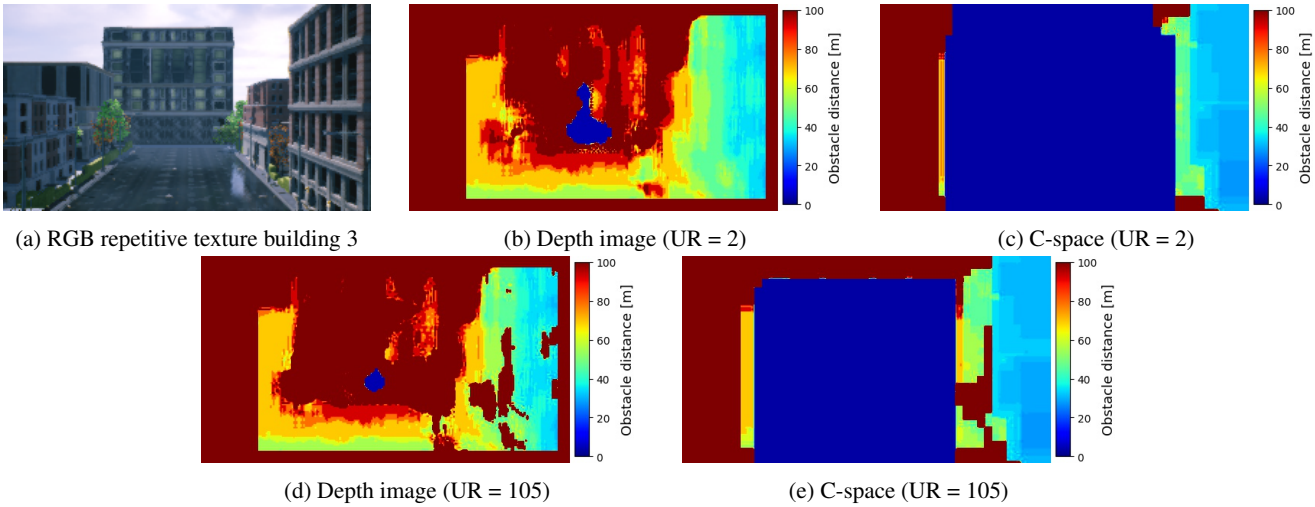


(d) Depth image (UR = 105)



(e) C-space (UR = 105)

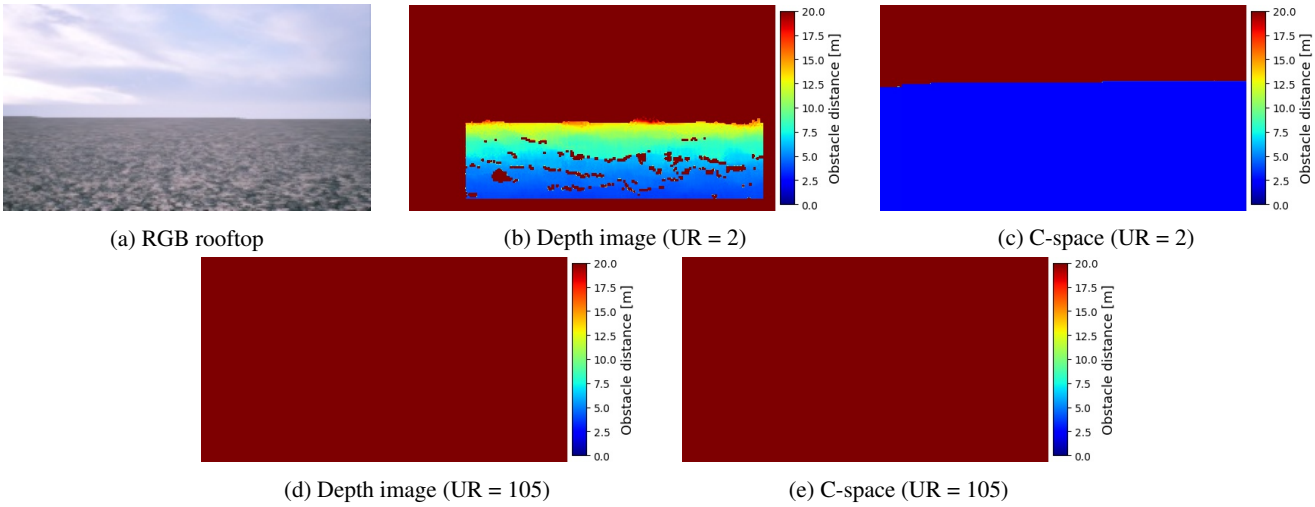
Figure 29: Increasing the Uniqueness Ratio (UR) removes the incorrect stereo matches. In fact, it removed many pixels since their uniqueness ratio was small.



(a) RGB repetitive texture building 3 (b) Depth image (UR = 2) (c) C-space (UR = 2)

(d) Depth image (UR = 105) (e) C-space (UR = 105)

Figure 30: Example showing that increasing the Uniqueness Ratio (UR) does not always solve the problem of incorrect matches.



(a) RGB rooftop (b) Depth image (UR = 2) (c) C-space (UR = 2)

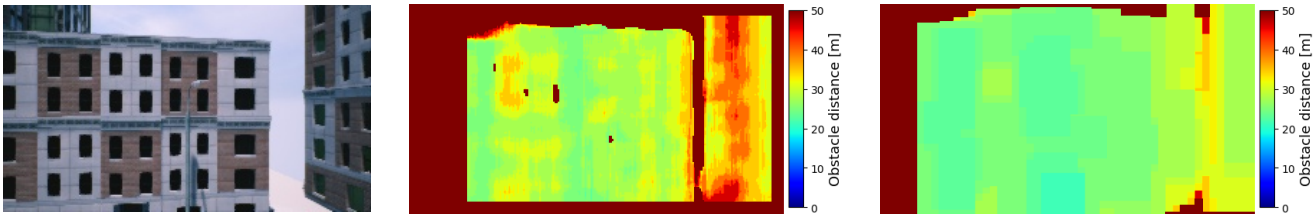
(d) Depth image (UR = 105) (e) C-space (UR = 105)

Figure 31: Example of the effects on a disparity image with low Uniqueness Ratio (UR) overall. All estimates were deleted from the depth image, and the drone was told the path to goal was safe. This led to a collision.

APPENDIX D URBANCITY RESULTS

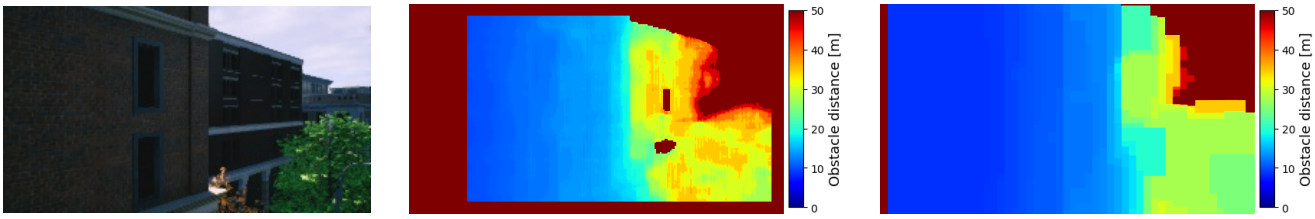
Many paths have been travelled through UrbanCity and a large amount of data has been collected. This appendix is included to show the strengths of the stereo matching through six good results, but also to reflect on its limitations by showing two acceptable and two unsatisfactory results.

The first six examples show good stereo matches for buildings, trees, streetlights and electric cables. The buildings in Figure 32 and Figure 33 are well matched and it has no problems regarding nearby obstacles, as shown in Figure 34. Furthermore, it can match small objects like the leaves and branches in Figure 35, the streetlight in Figure 36 and the cable in Figure 37.



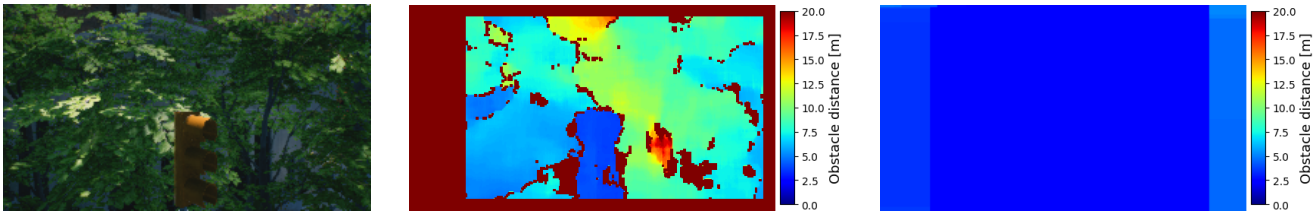
(a) RGB building 1 (b) Depth image (c) C-space

Figure 32: Good example: correct depth image of a building, with most of the pixels matched.



(a) RGB building 2 (b) Depth image (c) C-space

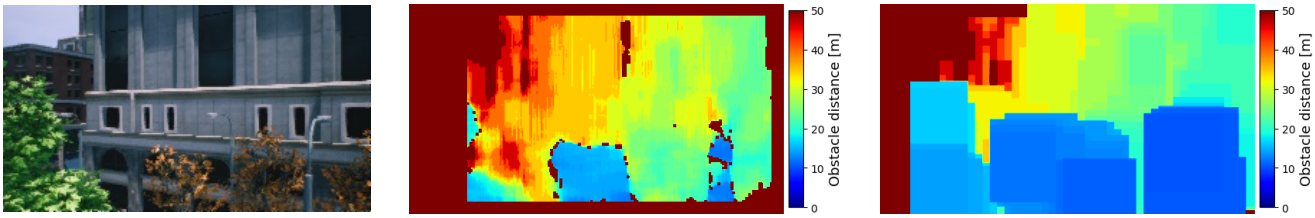
Figure 33: Good example: correctly matched building, even though the texture and brightness are low.



(a) RGB tree 1 (b) Depth image (c) C-space

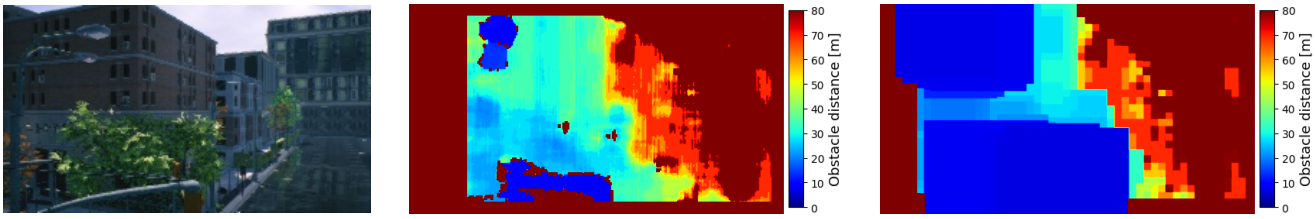
Figure 34: Good example: the stereo matching algorithm has no problems with nearby obstacles, and the branches and leaves are matched correctly.

Next, there are three examples that show satisfactory results for stereo matching. By this is meant that the initial depth image may not be perfect, but it is compensated for by the obstacle expansion in disparity space, or it is not a direct threat to a safe flight. Figure 38 shows an image with large patches of low-texture, especially near the bottom, which are not matched. Note that it initially does not match the window, but the reflection in the window. Figure 39 shows a traffic light which is not completely visible in the depth image. However, enough texture has been matched for it to block the entire FoV in the C-space.



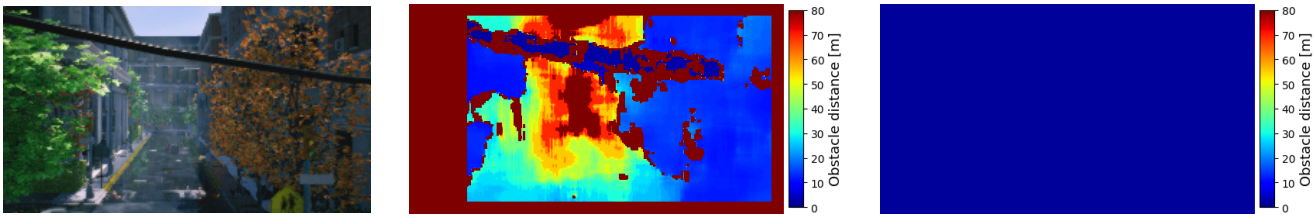
(a) RGB tree 2 (b) Depth image (c) C-space

Figure 35: Good example: good stereo matches for both the trees and the building.



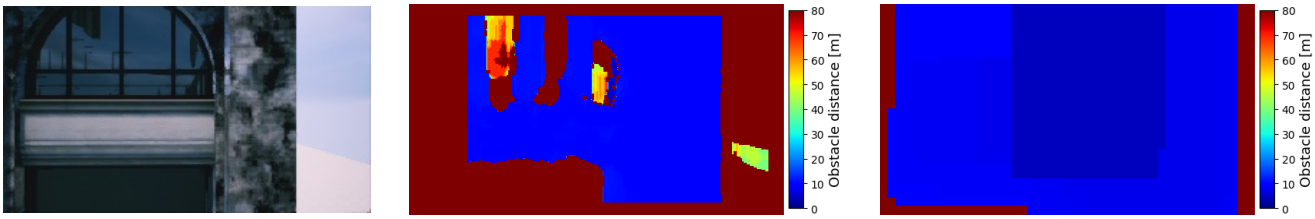
(a) RGB streetlight 1 (b) Depth image (c) C-space

Figure 36: Good example: small obstacles like streetlights are identified and marked unsafe.



(a) RGB cable (b) Depth image (c) C-space

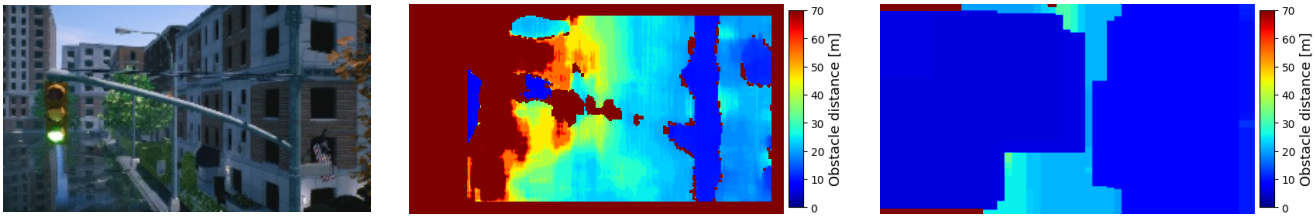
Figure 37: Good example: cables are noticed in time.



(a) RGB low texture (b) Depth image (c) C-space

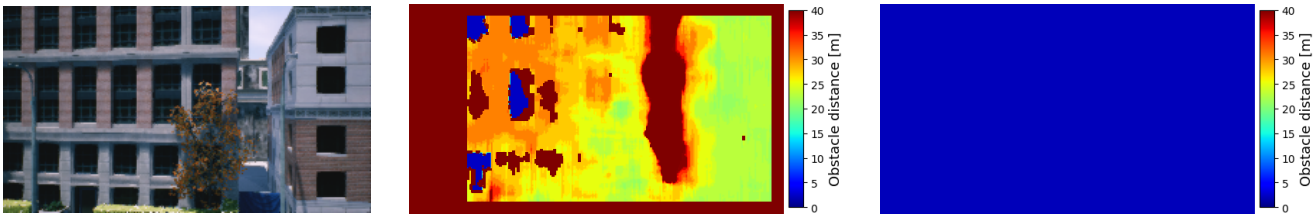
Figure 38: Satisfactory example: not all low-textured pixels are matched, but after obstacle expansion there are no more gaps.

Lastly, two on-board images are presented to show bad stereo matches. Most of the bad matches were caused by repetitive texture. This was already shown in Appendix C, and yet another example follows in Figure 40. Moreover, there same cable from Figure 37 is shown, this time from a farther distance. Although the depth image still matches a tiny part of the cable, it indicates that the cable should not get a lot thinner.



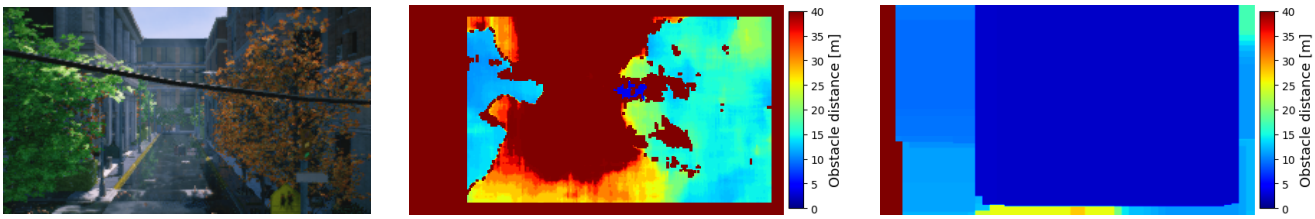
(a) RGB traffic light (b) Depth image (c) C-space

Figure 39: Satisfactory example: not all pixels belonging to the traffic light are matched, but it is enough to cover the FoV in the C-space



(a) RGB repetitive texture (b) Depth image (c) C-space

Figure 40: Unsatisfactory example: Repetitive texture is often identified as a large disparity, leading to incorrect C-spaces.



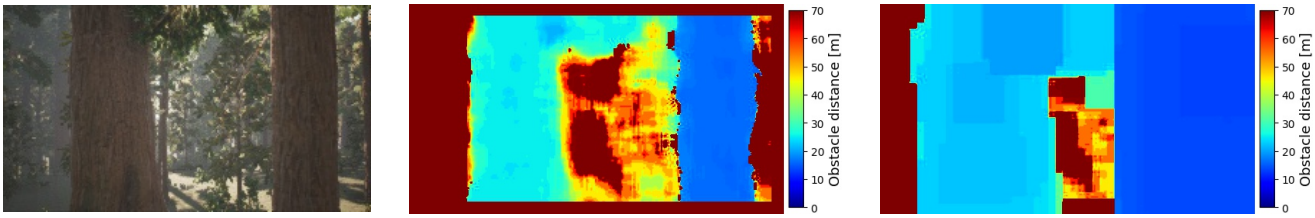
(a) RGB cable 2 (b) Depth image (c) C-space

Figure 41: Unsatisfactory example: The cable almost goes unnoticed. If it would get thinner, the drone might not notice it until it is very close.

APPENDIX E FOREST RESULTS

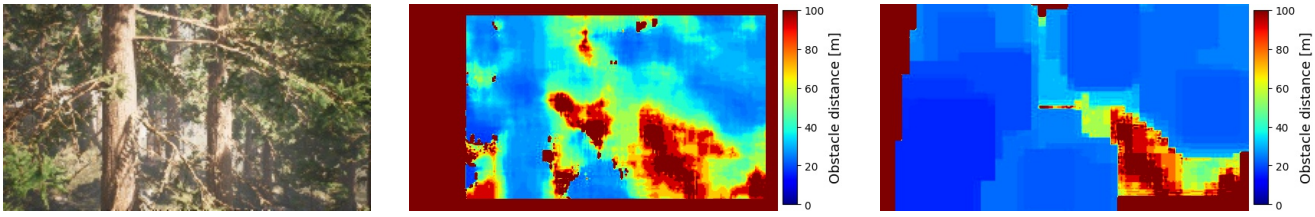
This appendix shows a number of examples coming from on-board footage of the drone flying through the Forest (AirSim) environment. Three good, and three unsatisfactory results are presented.

Compared to UrbanCity, Forest tends to have highly textured images and repetitive textures are uncommon. The result is that stereo matching performs well. Figure 42 shows that wide trunks are matched easily. Figure 43 and Figure 44 show the performance for larger branches. In general, the obstacles are matched correctly and are avoided without problems.



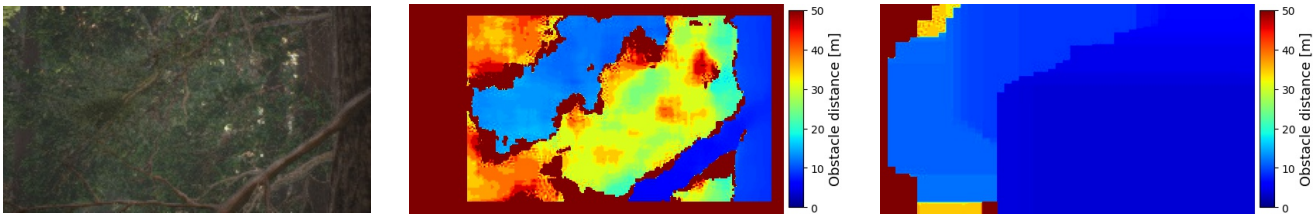
(a) RGB tree trunks (b) Depth image (c) C-space

Figure 42: Good example: tree trunks are matched correctly, as well as the branch in the top-middle of the image



(a) RGB large branches (b) Depth image (c) C-space

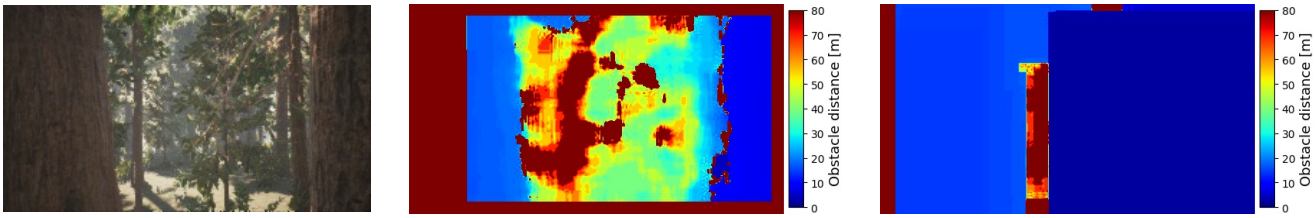
Figure 43: Good example: large tree branches are identified and the C-space makes sure the drone navigates around them



(a) RGB nearby branches (b) Depth image (c) C-space

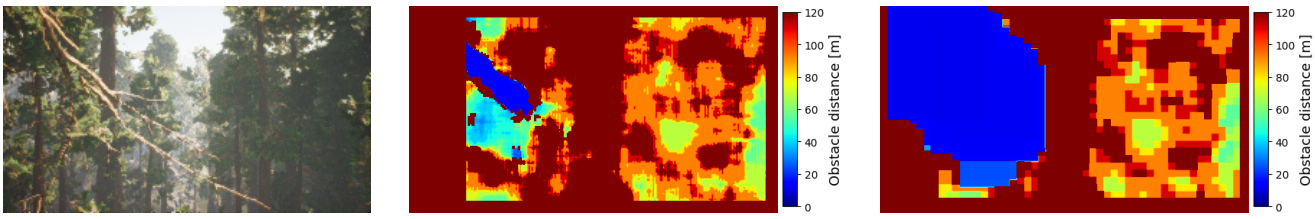
Figure 44: Good example: nearby branches are matched correctly, and the path is marked unsafe.

However, there are also examples of less successful matches. Figure 45 shows an example where a small section of the tree in the bottom right is matched incorrectly, leading to a large disparity which covers the FoV after obstacle expansion, possibly caused by a small patch of repetitive texture. This type of mismatch is uncommon in Forest. Next, there are two examples where matching branches was not successful. For Figure 46 only a small part of the branch was detected. The branch in Figure 47 is incorrectly matched, since the estimated depth is around 30 metres while the true depth is less than 10 metres. This is likely to be caused by the blocksize of the algorithm: a smaller blocksize has a higher chance of detecting small branches, but generally gives a more noisy depth image.



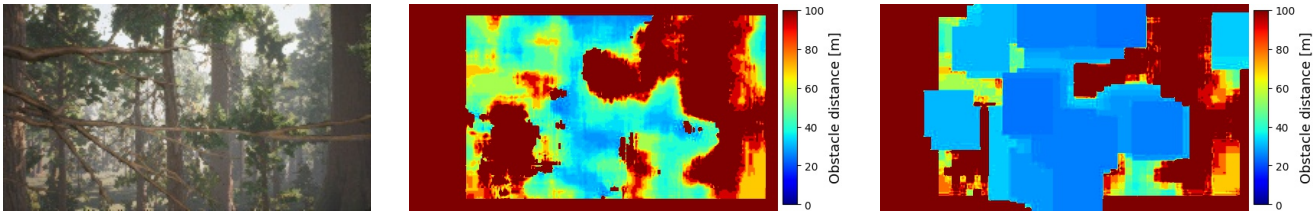
(a) RGB mismatch (b) Depth image (c) C-space

Figure 45: Unsatisfactory example: mismatch on the tree trunk on the bottom right causes the FoV to be covered after obstacle expansion.



(a) RGB small branch 1 (b) Depth image (c) C-space

Figure 46: Unsatisfactory example: only a small part of the branch has been matched correctly. It is not enough to steer the drone around it safely.

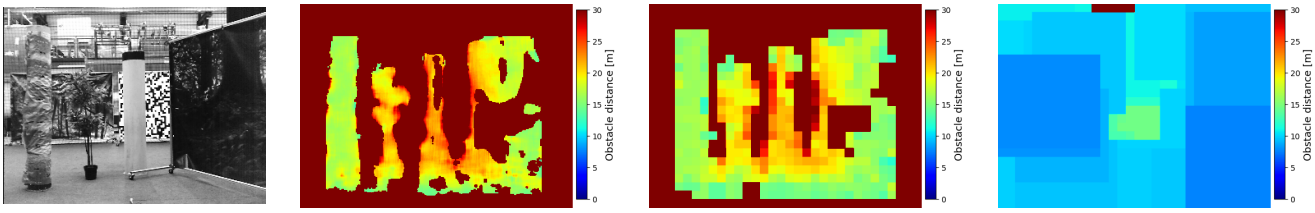


(a) RGB small branch 2 (b) Depth image (c) C-space

Figure 47: Unsatisfactory example: the branch is not matched and the path is incorrectly labelled as safe.

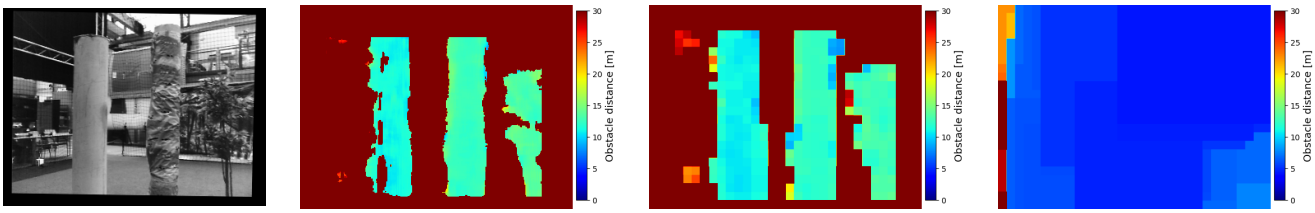
APPENDIX F CYBER ZOO RESULTS

This section shows 7 images which summarise the first path created by Frustumbug in the Cyber Zoo experiment. Note that for finding a waypoint, the 'safe' pixels in the C-space still have to be mathematically eroded, as explained in subsection 3.2. Therefore, there are no safe waypoints found in C-spaces such as Figure 48d and Figure 49d.



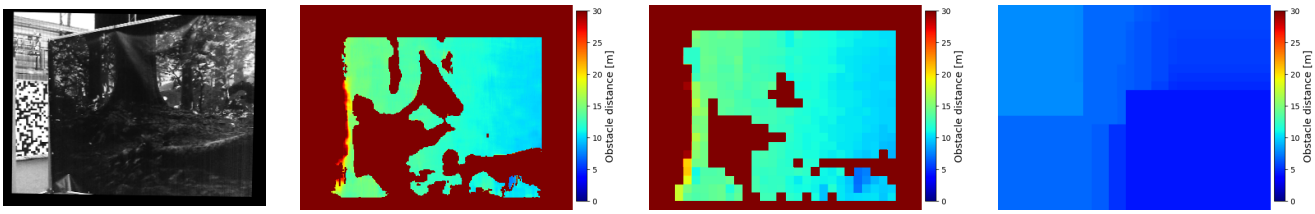
(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 48: Obstacle depth drops below threshold, path is unsafe and no waypoint is found in the C-space. State *MtG* transitions to state *SW*.



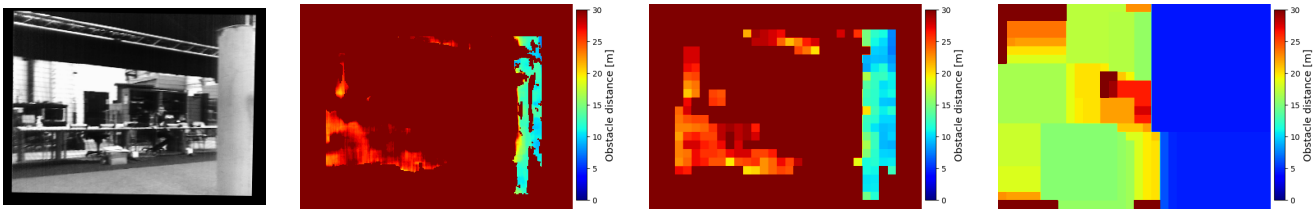
(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 49: State *SW* first scans left. Obstacle blocks the path and no waypoint is found.



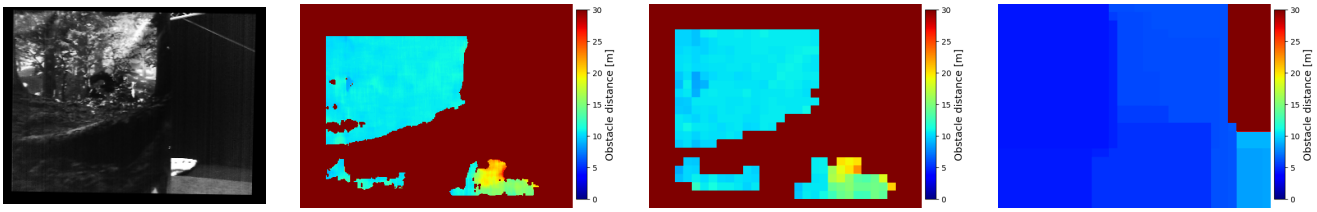
(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 50: State *SW* scans right. Obstacle blocks the path and no waypoint is found.



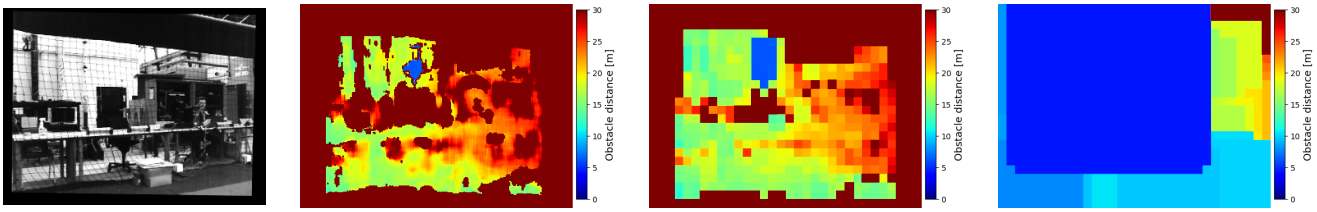
(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 51: State *SW* scans further to the left. Obstacle is detected, but a safe waypoint around it is found.



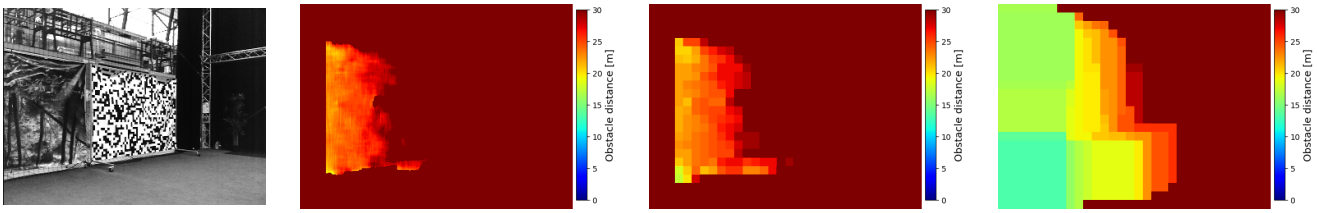
(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 52: To make sure it does not miss out on opportunities, state *SW* also scans the right wedge. Obstacle is detected, and a safe waypoint is found on the right-hand side. Since this waypoint yields a (locally) longer path to goal than the waypoint found on the left, it is discarded. The drone turns left again and transitions to state *MtW*.



(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 53: In state *MtW*, the Cyber Zoo safety net is matched incorrectly, likely because of repetitive texture. A waypoint on the right-hand side is found. This happens once more in this experiment.



(a) Greyscale image (b) Depth image (c) Depth image downscaled (d) C-space

Figure 54: The drone has arrived at the waypoint, and state *MtG* scans for a safe path to goal and executes the last segment of the path.

APPENDIX G FINITE-STATE MACHINES FOR ALL STATES

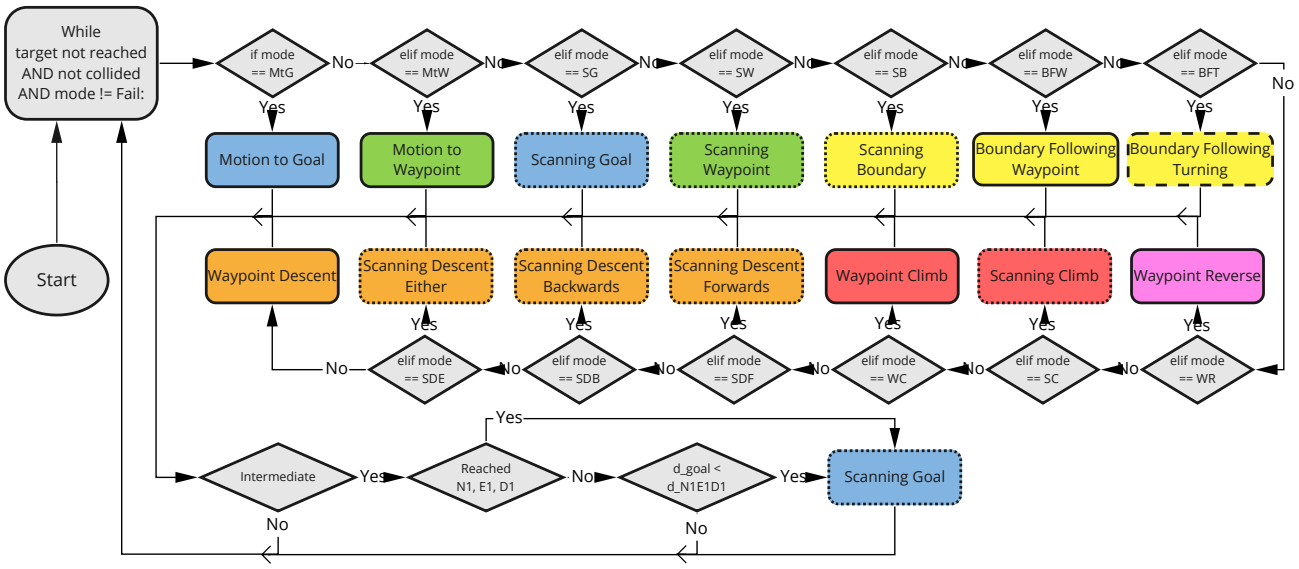


Figure 55: Main python file

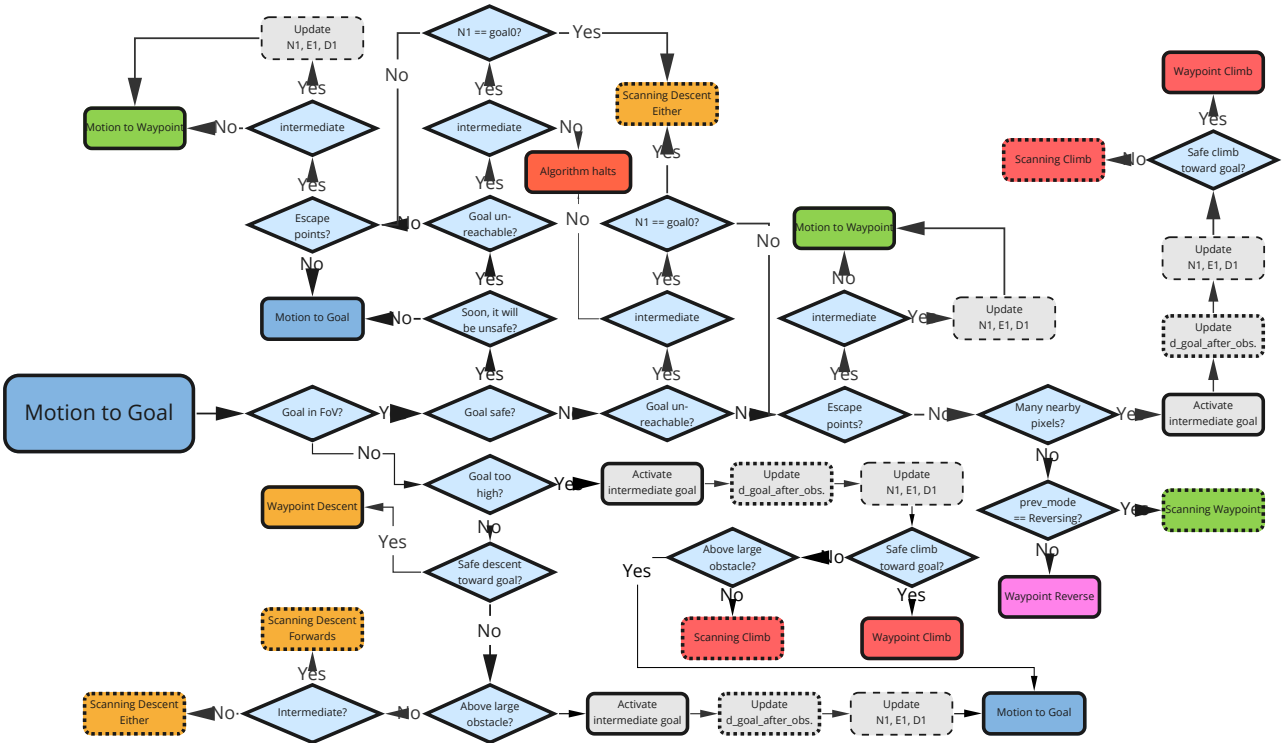


Figure 56: Motion to Goal

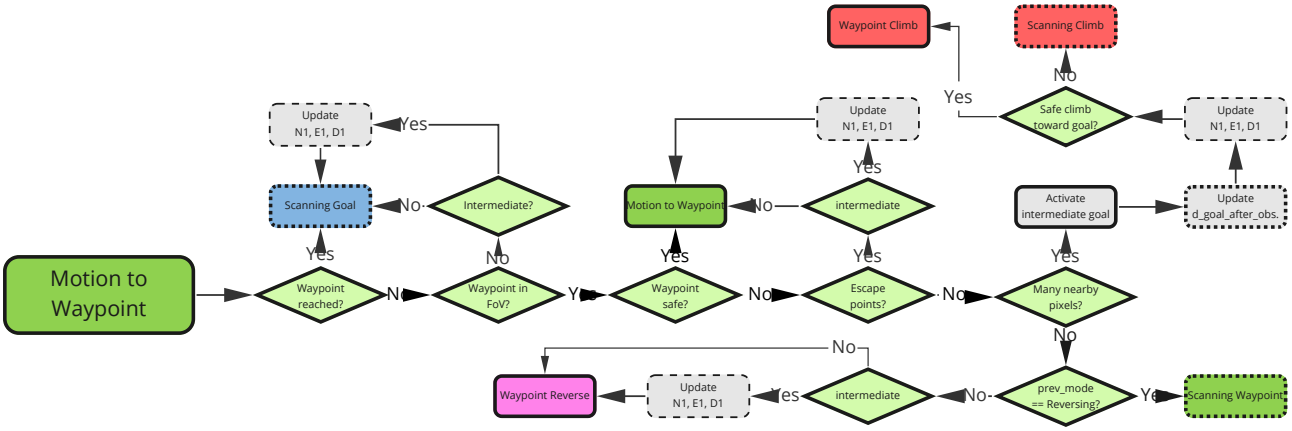


Figure 57: Motion to Waypoint

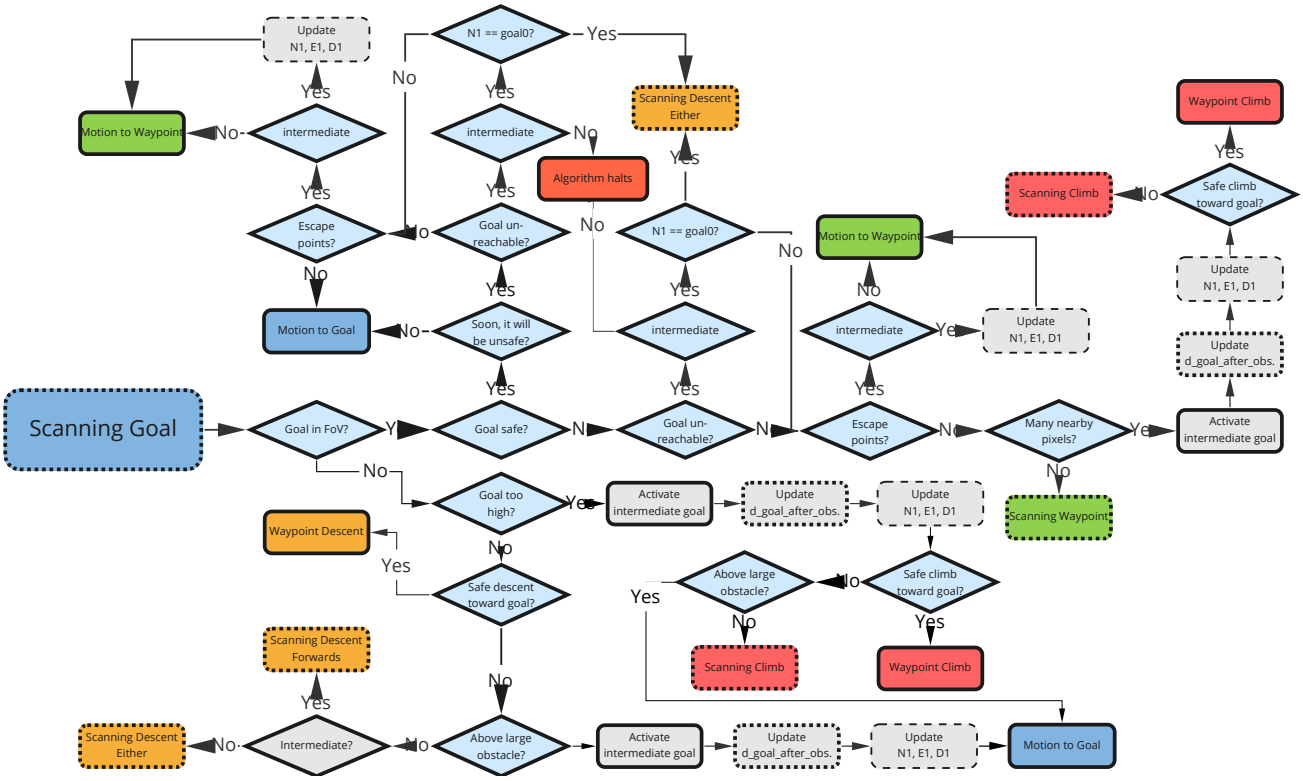


Figure 58: Scanning Goal

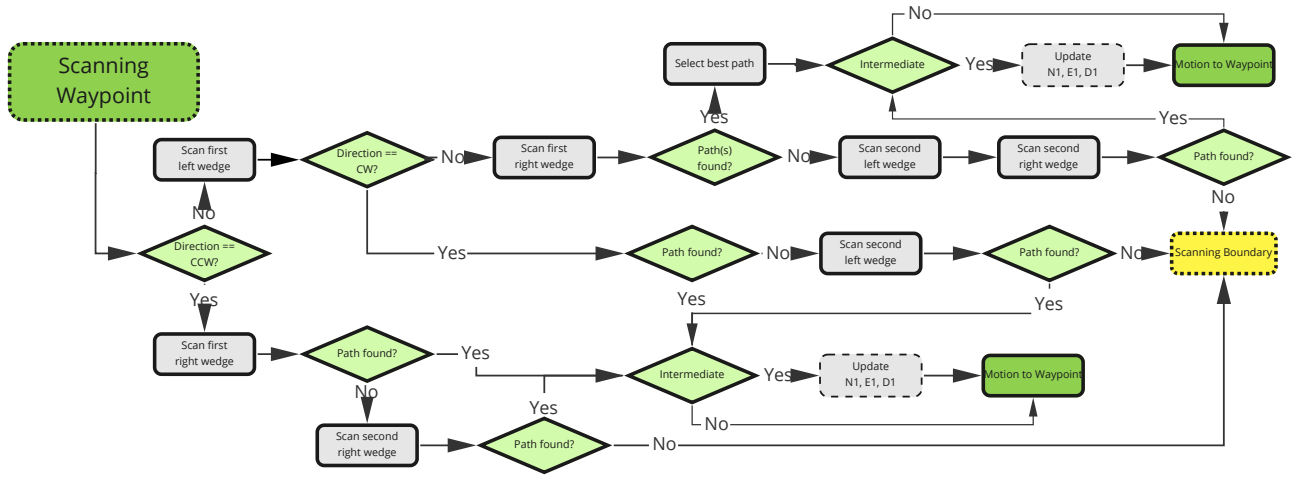


Figure 59: Scanning Waypoint

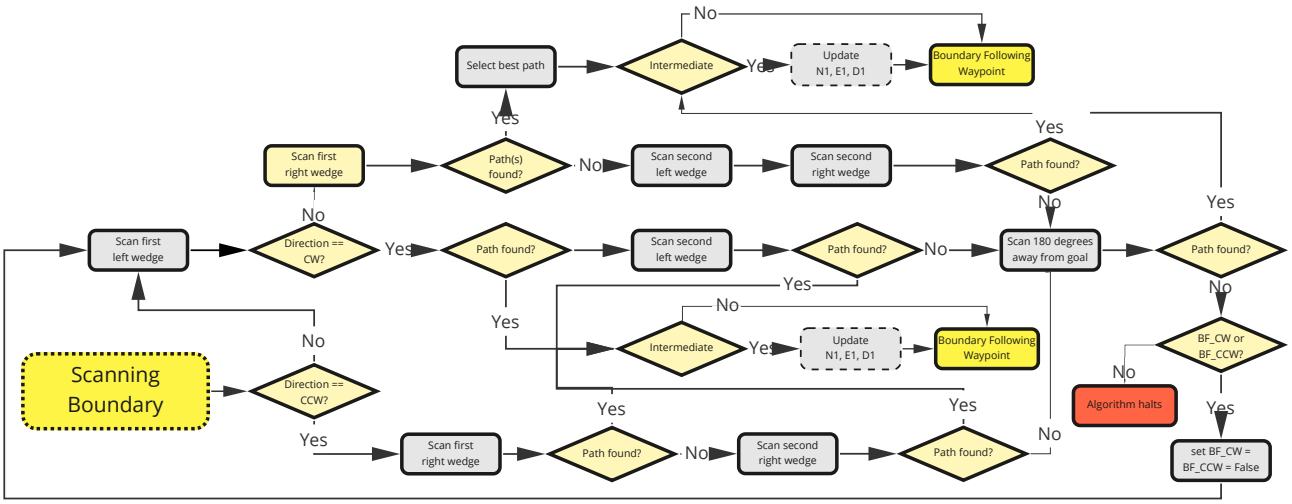


Figure 60: Scanning Boundary

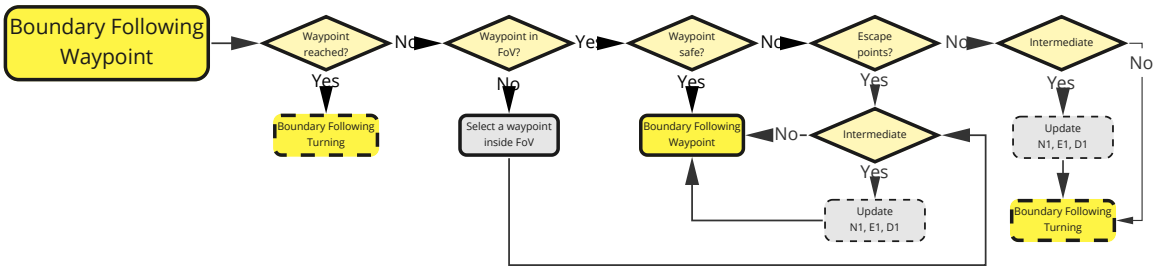


Figure 61: Boundary Following - Waypoint

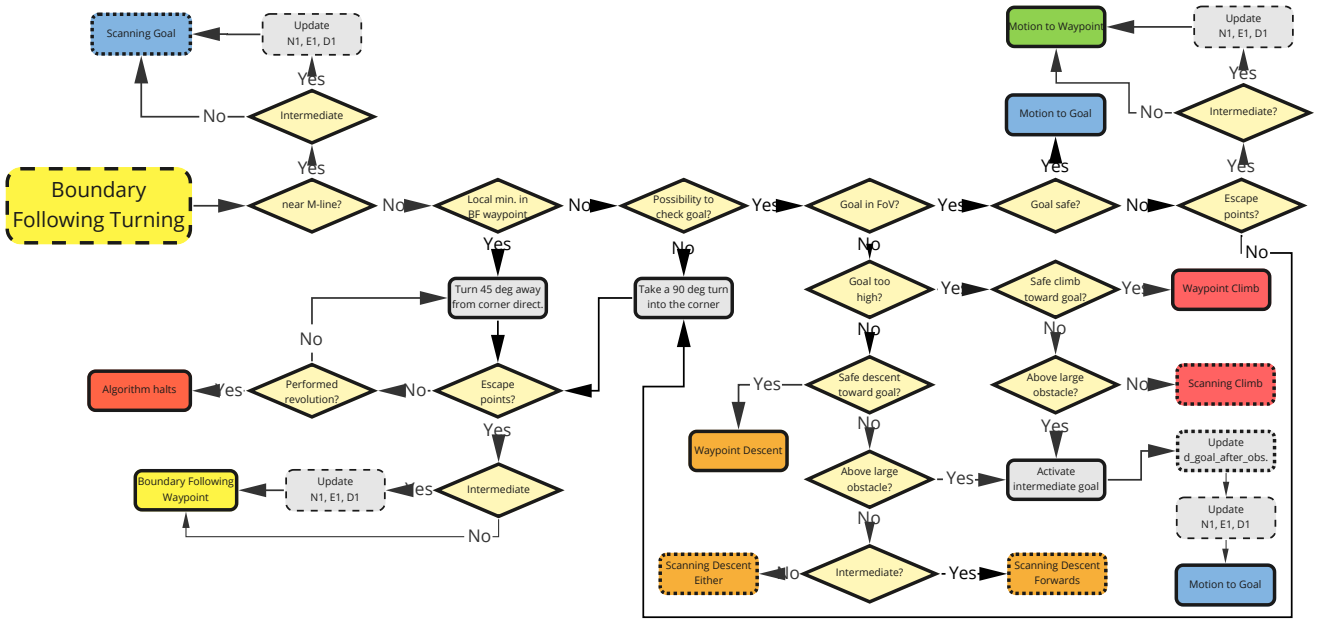


Figure 62: Boundary Following - Turning

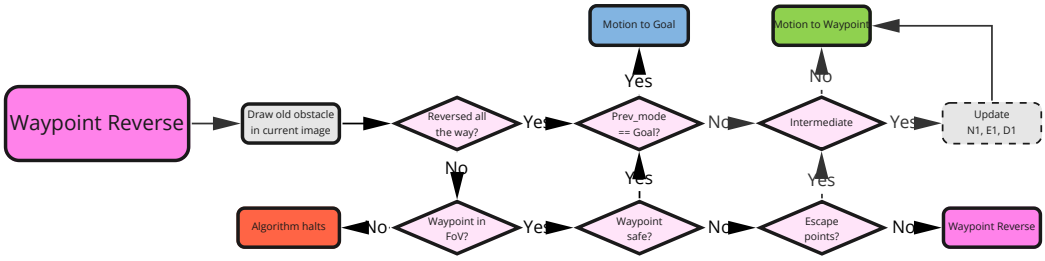


Figure 63: Waypoint Reverse

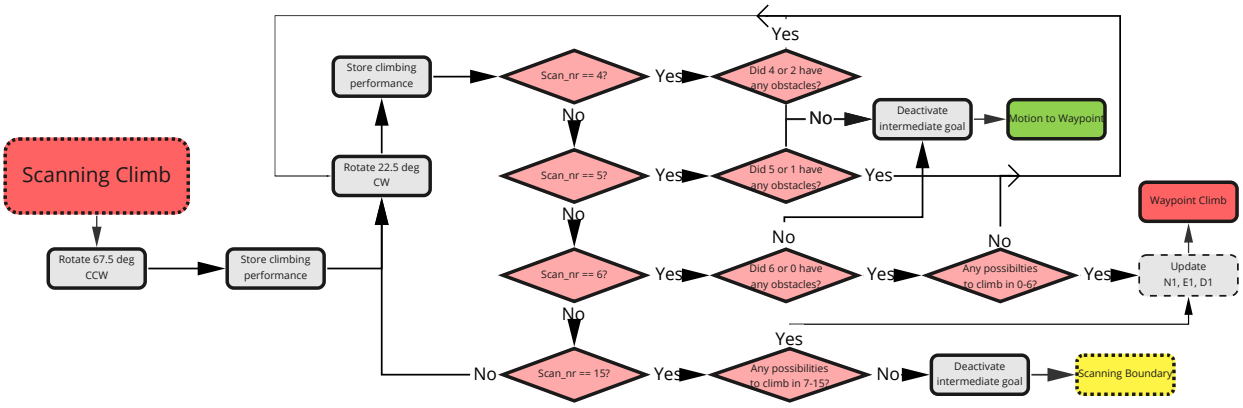


Figure 64: Scanning Climb

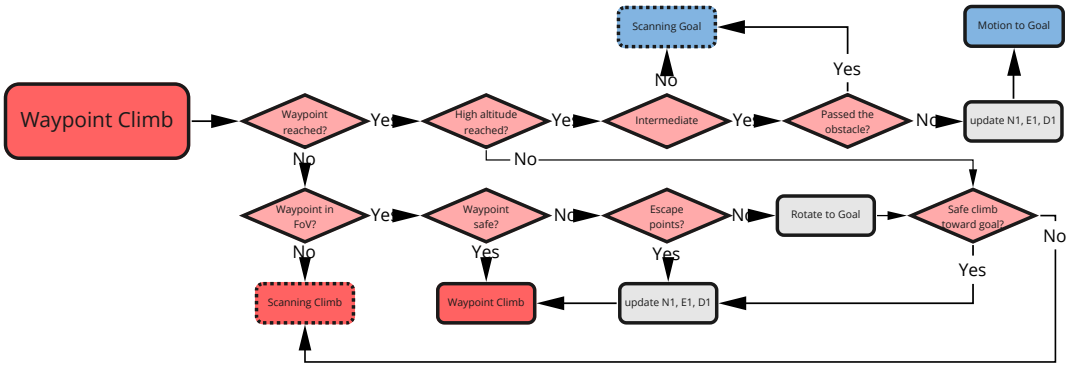


Figure 65: Waypoint Climb

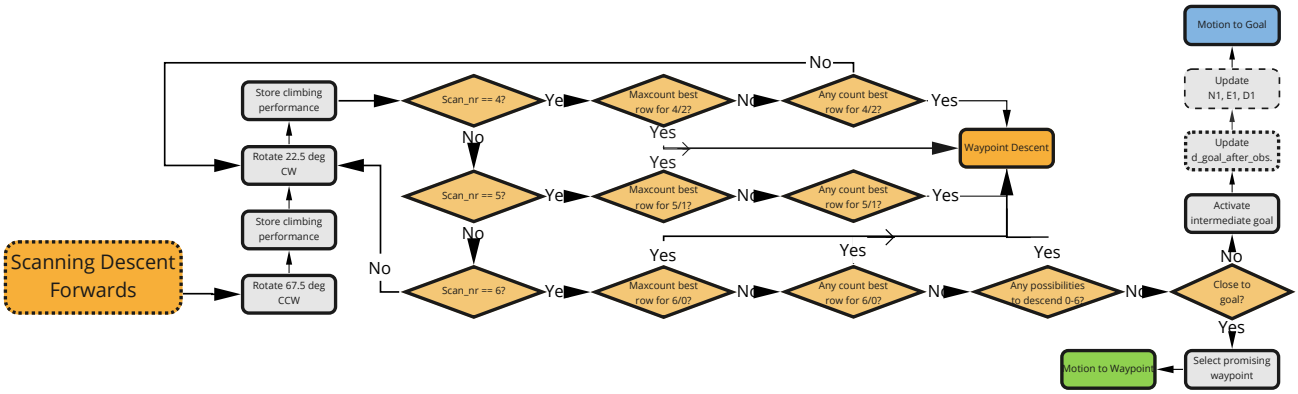


Figure 66: Scanning Descent Forwards

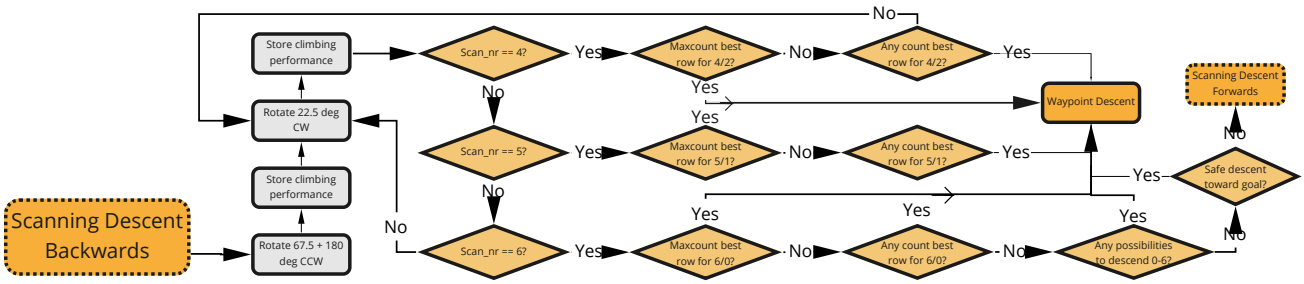


Figure 67: Scanning Descent Backwards

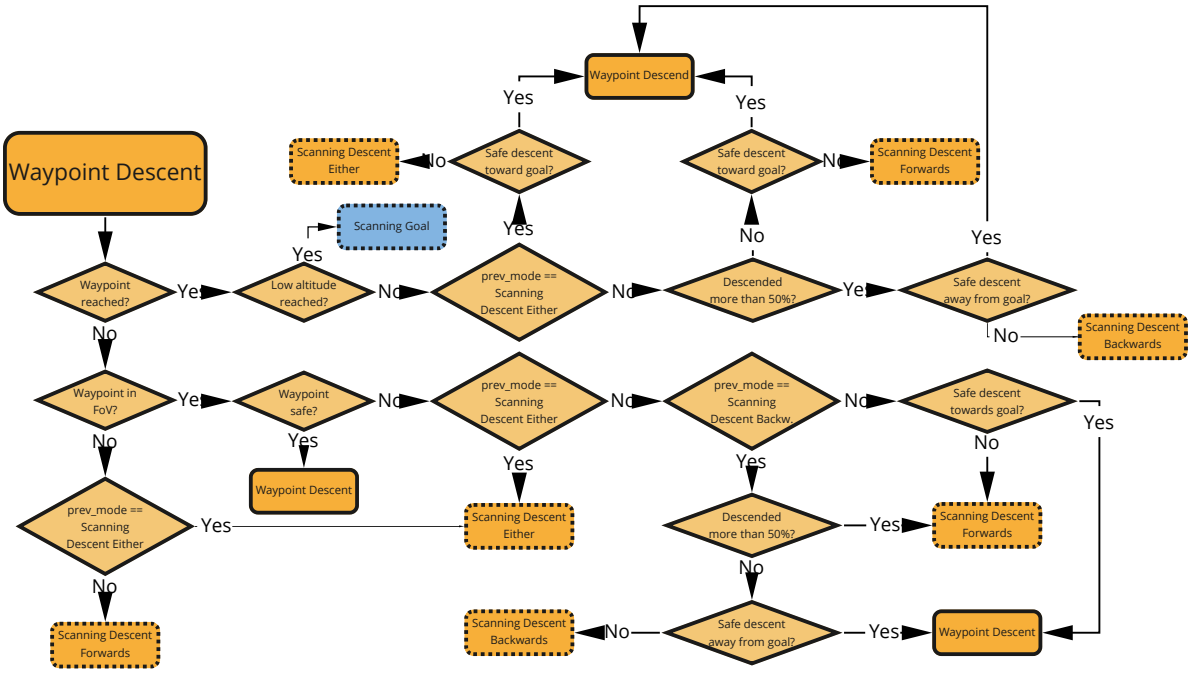


Figure 68: Waypoint Descent

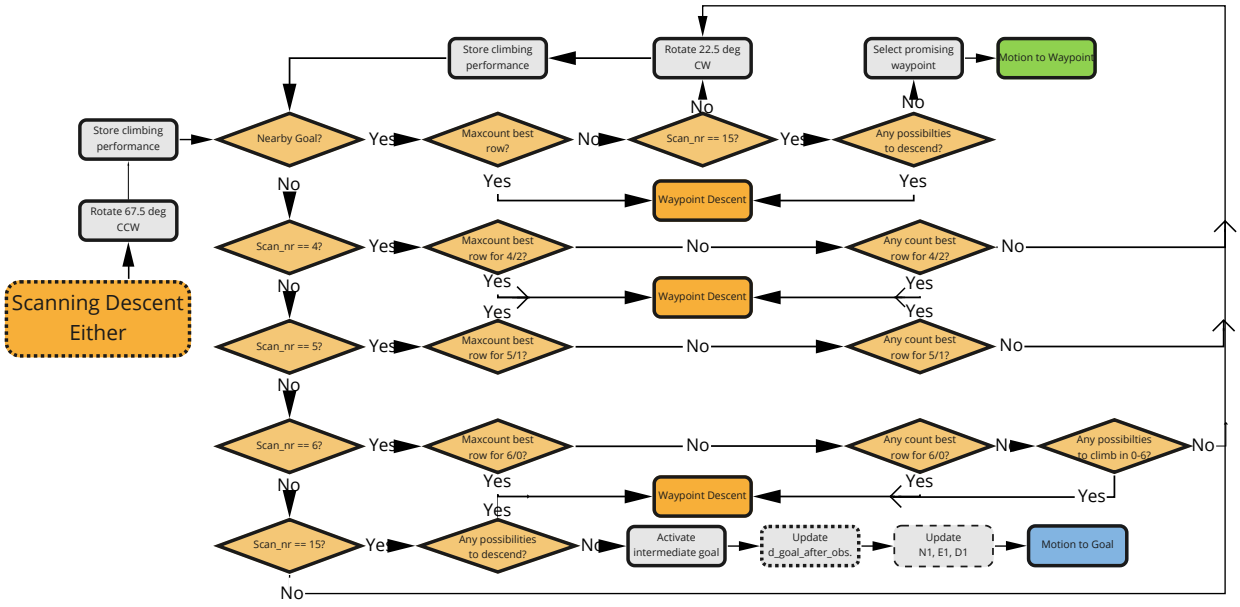


Figure 69: Scanning Descent Either

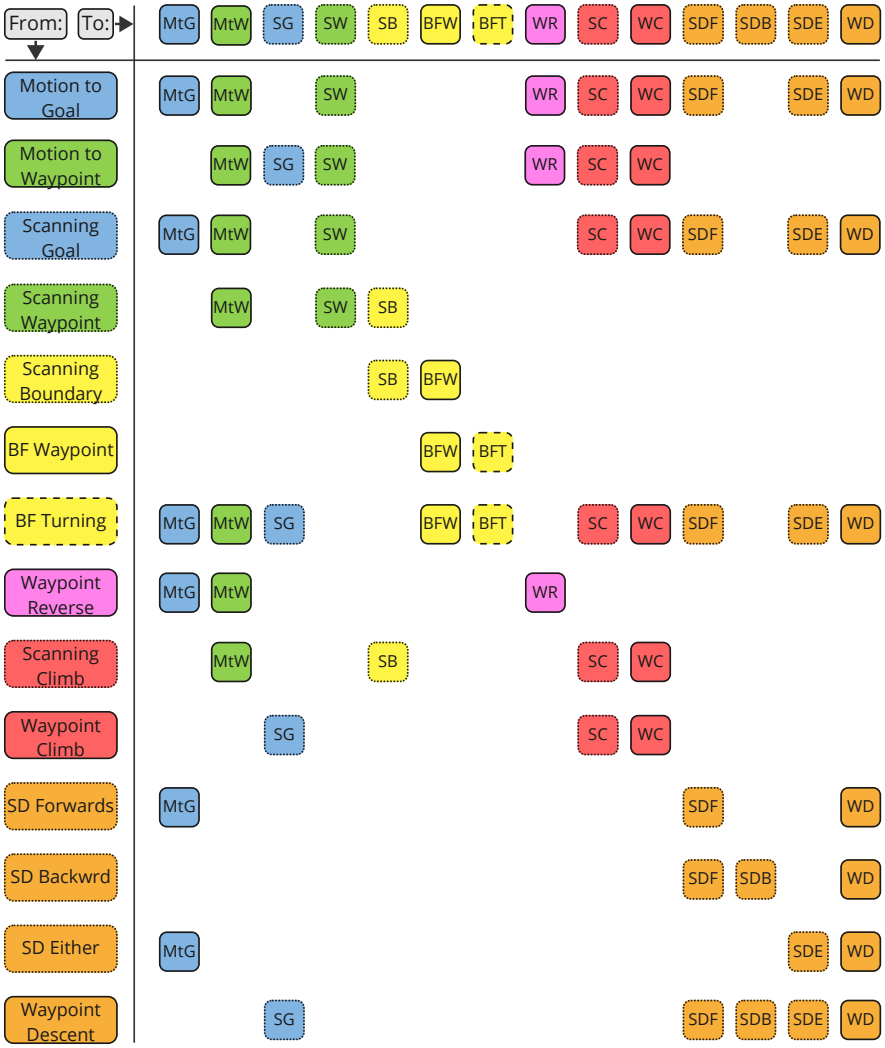


Figure 70: Overview of the allowed state transitions

II

Literature Study
previously graded under AE4020

1
Sensing

Drones need multiple sensors in order to fly. For example, the Inertial Measurement Unit (IMU) measures accelerations and angular velocities, and the Global Positioning System (GPS) gives the position, as well as the velocity. With these sensors, both the attitude and location of the drone is known. However, autonomous drones will need distance measurement sensors to obtain the necessary information about their environment. The size, weight and power consumption of these sensors on small drones must be minimised.

1.1. Distance measurement sensors

Many different distance measurement sensors exist. This section holds eight common sensors used for autonoumous drones: SONAR, LIDAR, RADAR, Infrared, Time-of-Flight cameras, structured light, mono camera, stereo camera and multi camera. (Fiorenzani, 2017, Lyrakis, 2019). Nyasulu et al. (2017) performed a survey on low-cost sensing solutions for drones in an outdoor environment, and concluded that there is no clear winner which outperforms all the other sensors. Every sensor comes with its advantages and disadvantages, hence a more detailed exploration is provided in this section.

- Sound Navigation and Ranging (SONAR) emits sound waves and calculates the distance based on the time it takes before the echo returns. It tends to be inaccurate, since the echo of the sound wave bounces off many surfaces. For round obstacles like a tree, the wave will not reflect back to the drone when it hits the side of the tree (Nyasulu et al., 2017). Furthermore, these sensors only have a range around 5 meters, making fast flight difficult.

- Light Detection and Ranging (LIDAR) applies the same theory, but works with a laser which sends out pulses. LIDAR is more accurate than SONAR, but it comes at the cost of a heavier system. It has been proven to work on a small drone by Shen et al. (2011). They combine a laser scanner with a Microsoft Kinect to obtain a 3D map for a computationally constrained drone. However, their drone is able to store 1 GB RAM, compared to the 256 MB used in this study. Furthermore, LIDAR does not give information about the obstacle's shape, unless some signal processing is done, leading to large computational expenses (Nyasulu et al., 2017)

- Radio Detection and Ranging (RADAR) is another application of the theory, which uses radio waves. A RADAR gives very good estimates of the distance to the obstacles (Aswini et al., 2018). Unfortunately, like LIDAR, these sensors are heavy and cannot be carried by small drones. Moses et al. (2011) attempted to build a RADAR system on top of a Parrot AR drone to detect other nearby drones. Shown in Figure 1.1, the system is rather bulky, weighs 230g and decreased the performance of the drone regarding velocity, flight time and wind resistance.

- Infrared proximity sensors send out waves in the infrared spectrum and measure the intensity of the light when it has bounced off a surface. To calculate the distance to an obstacle, the change in intensity of the light is used (Adarsh et al., 2016a). To prevent the sunlight from interfering with the infrared signal, Rusu et al. (2010) send out these waves at 38.5 kHz, and only look for received signals at around this frequency. A disadvantage is that the sensor can be inaccurate, for example when detecting a non-smooth surface like wood (Adarsh et al.,

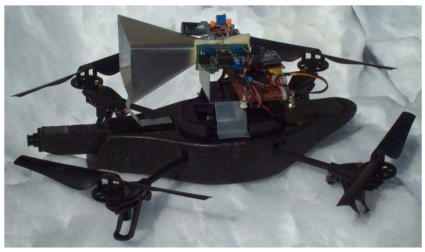


Figure 1.1: Radar system on top of Parrot AR drone (Moses et al., 2011)

2016b). Furthermore, like SONAR, infrared waves are not returned when they hit curved obstacles. Lastly, these sensors are not appropriate for outdoor environments, since they have a short range of about 2 meters (Nyasulu et al., 2017).

- A Time of Flight camera sends out pulses or a continuous wave of light, and waits for the reflected light to hit the lens. In case of a pulse this reflection has a time difference, and in case of a continuous wave it has a phase difference. Either of these methods can be used to obtain the depth and intensity information of all pixels at the same time (Moeller, 2016). There are three main disadvantages. First, the reflected light gets scattered inside the camera if an obstacle gets too close. Second, concave obstacles might cause multiple reflections, leading to erroneous readings. Third, sunlight directly into the camera saturates the pixels and no useful information can be obtained anymore.

- Structured light can be seen as LIDAR, but without the use of scanning. For LIDAR, either the camera, object or light needs to be in motion. Structured light is similar to stereo vision, however one of the cameras is replaced by a projector which sends out a structured light pattern. The camera observes this pattern on an obstacle and is able to form a 3D reconstruction. The advantages of a structured light camera over a stereo camera is that it can still give depth estimates when the visual texture is low (Bachrach et al., 2012). Disadvantages of structured light are that it also has troubles with reflections and transparent obstacles, and that the range depends on the projector's performance.

- A mono camera takes a 2D picture of the environment. Cameras are accurate and can provide a high resolution. Moreover, Aswini et al. (2018) state that vision-based sensors are the only promising option for a distance measurement sensor when looking at computational power, size, weight and cost. Furthermore, outdoor environments usually have enough texture to be captured with a mono or stereo camera (Matthies et al., 2014). Obstacle depth can be obtained from a mono camera in two different ways: either by using one image, or multiple images with a different time stamp. (Deep) Machine learning algorithms can be trained to map a single RGB image onto a depth map. The downside is that these algorithms needs many images to train on. Keltjens et al. (2021) train their monocular depth estimation algorithm with 18,000 images, and their environment is limited to a relative small indoor simulated environment. For outdoor environments there is an extreme variety of different surroundings. A dataset representing the complete variety will first have to be created, which will likely take more than just a few thousand images.
  Another option is an algorithm that uses optical flow. These algorithms use at least two images, and translate the differences into image disparity. Image disparity is a measure of how much an obstacle has moved between the two images, which is inversely related to the distance. Optical flow yields a flow divergence, which is defined as the velocity divided by the distance. The inverse gives a unit in seconds, which is the Time-To-Contact (TTC), or: the time it takes for the drone to hit the obstacle. Since GPS data is known, the velocity can be

obtained, as well as the distance to the obstacle.

- For a stereo camera, the depth of an obstacle can be calculated through simple triangulation (de Croon, 2021). The base b of the cameras is known, and the focal length f can be calculated using the FoV. Dividing the base by the disparity d, and multiplying this with the focal length will give depth values, as shown in Equation 1.1. To clarify, Figure 1.2 shows how these variables are related (van Dijk, 2020). Disparity information can be obtained from a stereo image matching algorithm. Since these algorithms will not yield perfect disparity images due to noise, they likely need to be post-processed before a path planning algorithm can be applied.

$$z[m] = \frac{B * f}{d} \tag{1.1}$$

- Multi-camera layouts can provide reliable depth maps, since multiple images can be taken from different angles at the same time (Sanchez-Rodriguez and Aceves-Lopez, 2018). The more cameras used on the drone, the smaller the errors in depth estimation. Unfortunately, as expected, this comes at the cost of extra weight and heavier computations (Lyrakis, 2019).

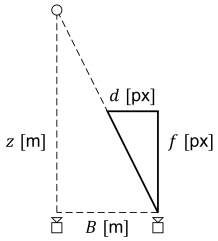


Figure 1.2: Depth estimation using stereo vision (van Dijk, 2020)

To conclude, either a mono or stereo camera will be used to give the drone the information it needs to plan its path, due to their high range, low weight and relatively low required computational resources. Since outdoor flights during daylight are assumed, there will be enough texture to be captured by a camera. The next section will discuss the differences between the two methods. A path planning algorithm will use either the local percept directly, or use a map created by one or multiple percepts, as shown in Figure 1.3. The rest of this literature study will discuss both possibilities. The chapters in the figure at the arrows refer to the chapters in this report that deal with the topic.

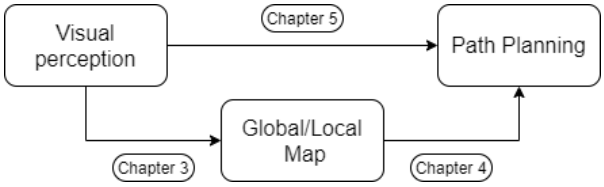


Figure 1.3: Two types of path planning: map-based and mapless

1.2. Visual perception

A camera captures the environment by taking 2-dimensional RGB images. For a mono camera, disparity can be obtained using optical flow algorithms. For a stereo camera, a stereo image matching algorithm is needed. In this section, these two methods will be compared. Both methods can have either sparse or dense approaches.

Sparse: Sparse approaches only look at a small amount of recognisable features inside an image, and perform their calculations on this small portion of pixels. For optical flow these features can be corners for example, and for stereo vision highly textured regions are of interest. Since not all pixels are matched between the images, the resulting disparity images will consist of large gaps (van Dijk, 2020).

Dense: Dense algorithms will give each pixel in the image a disparity value, which yields more available information for path planning algorithms. Since large gaps of information need to be avoided, the dense approach is the most suitable for drone sensing. Dense approaches for both mono- and stereo camera will be discussed.

1.2.1. Mono camera

Optical flow uses vectors as an indication of how much a pixel moved from one image to the next. Pixels with large vectors likely belong to nearby obstacles, while pixels with smaller vectors are likely to be farther away obstacles. Optical flow gives information about the flow divergence, which can be combined with GPS data to obtain estimates for the distances.

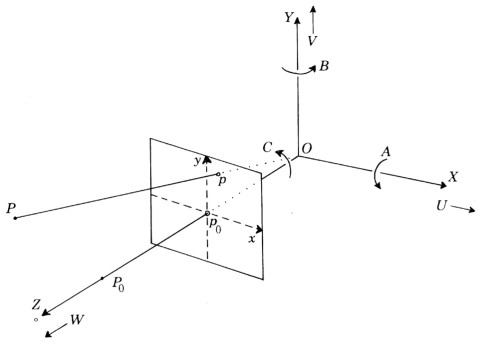


Figure 1.4: Image coordinate system (Longuet-Higgins and Prazdny, 1980)

The coordinate system used in the following calculations is shown in Figure 1.4. Here, the image coordinates x and y are the normalised coordinates of X and Y, since they are divided by Z and multiplied by the focal length. When there is movement between two images, x and y will have a derivative unequal to zero. These derivatives can be calculated, and are a function of velocities U, V and W, rotations A, B and C, depth Z and normalised image coordinates x and y. They are presented in Equation 1.2 and Equation 1.3 (de Croon, 2016). The values of $\dot{x}$ and $\dot{y}$ can be found for every pixel using dense optical flow algorithms. An example of this is Farnebäck's method, which only requires two sequential images (Farnebäck, 2002). For every pixel, it takes into account a few neighbouring pixels and tries to approximate their values using a second degree polynomial function. Using the parameters of this function, the method finds an estimation of the displacement of the pixels between the two images.

$$\dot{x} = -\frac{U}{Z} + x\frac{W}{Z} + Axy - Bx^2 - B + Cy \tag{1.2}$$

$$\dot{y} = -\frac{V}{Z} + y\frac{W}{Z} - Cx + A + Ay^2 - Bxy \tag{1.3}$$

Since GPS signal is available, the velocities U, V and W are known. Furthermore, the IMU holds information about the angular velocities in pitch, yaw and roll, which are represented by A, B and C.

The normalised image coordinates can simply be read from the image and the optical flow can be found using methods like Farnebäck. Therefore, the only unknown is depth Z, which can be calculated by rewriting the equations, shown by Equation 1.4 and Equation 1.5. In case a pixel has both optical flow in x and y, an average of the two calculated depth values can be used.

$$Z = \frac{-U + xW}{\dot{x} - Axy + Bx^2 + B - Cy} \tag{1.4}$$

$$Z = \frac{-V + yW}{\dot{y} + Cx - A - Ay^2 + Bxy} \tag{1.5}$$

Optical flow has three main drawbacks. Firstly, if there are small inaccuracies in GPS and IMU data compared to the actual velocities and rotations, these will accumulate, since many are needed for the depth calculation. Secondly, a larger area has to be searched to find a matching pixel neighbourhood, compared to stereo vision. This is due to the fact that stereo vision knows that the matching pixel in the second image can be found on the epipolar line (van Dijk, 2020). For a stereo camera which has both cameras pointing in the same direction, assuming a rectified stereo image pair, this line is a horizontal line. Thirdly, and most importantly, optical flow algorithms are not able to sense depth in the direction of flight. The drone is flying towards the point of divergence, which is an image coordinate in which no optical flow is present, since there is no disparity.

Due to these drawbacks, the stereo based algorithms are preferred. Stereo cameras can be very lightweight, for example the 4 g stereo camera used by K. McGuire et al. (2017), which also includes a microprocessor. The small bit of extra weight from the second camera is preferred over the drawbacks mentioned of a mono camera using optical flow.

1.2.2. Stereo camera

The second option is to use a stereo camera, to obtain two images at the same time from a different position. Dense stereo matching methods combine these two images into one image which contains disparity values for each pixel. A distinction can be made between local and global approaches (van Dijk, 2020).

Local: The local approach only looks at disparity values around a pixel within a small window. resulting in relatively fast computation. However, it is less accurate than the global approach.

Global: The global approach performs an optimisation which takes all disparity values into account. It includes a smoothness term, which makes sure that pixels that are located around the same area obtain similar disparity values. This creates better results, especially in environments with low texture, but it comes at a computational cost.

Since an outdoor environment during daylight is assumed, there is enough texture for the local approach. Therefore, a local dense approach will be used to create a disparity map. Lyrakis (2019) performed a state-of-the-art literature search on local dense stereo algorithms. These algorithms were tested on two famous datasets: KITTI and Middleburry. The most promising candidates were Block Matching (BM), Semi-Global Block Matching (SGBM), Efficient Large-Scale Stereo Matching (ELAS) and Slanted Plane Smoothing Stereo (SPS-St). These were then tested on the outdoor scenes shown in Figure 1.5(a) and Figure 1.6(a), as well as an indoor scene shown in Figure 1.7(a). ELAS and SPS-St mixed up background and foreground for the outdoor scene, and failed to produce a meaningful disparity image for the indoor scene. Furthermore, their runtime is a disadvantage: ELAS is 5

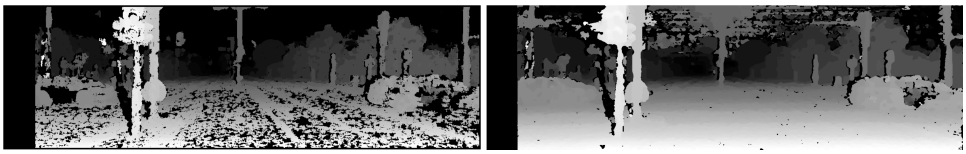
times slower than BM, and SPS 31 times. Therefore, only results from BM and SGBM will be shown in this study.

Block Matching (BM): BM is a local dense approach which computes disparity for every pixel. Excluded here are the first few columns equal to the number of maximum disparity, and a frame around the picture equal to half the blocksize, rounded down. It draws a so-called aggregation window around a pixel, and compares this to a similar sized window in the other image. This comparison is done by using the intensity values within this window, where multiple classes of metrics exist. Lyrakis (2019) mentions Normalized Cross-Correlation (NCC), Sum of Squared Differences (SSD), Sum of Absolute Differences (SAD), Rank Transforms (RT) and Census Transforms (CT). The result will be a cost value given to a range of disparities for a pixel, and the disparity with the lowest cost will be chosen, since it likely belongs to the true disparity of the original pixel. Lyrakis (2019) found an efficient implementation of Kurt Konolige, who compared intensity values using the Sum of Absolute Differences (SAD). The SAD is expected to be fast, since its calculations are relatively simple.

Semi-Global Matching (SGM): SGM is a mix between a local and global approach. The necessary computations are comparable to local algorithms, yet its performance is slightly under that of global methods. For every pixel it computes the disparity value using a Birchfield and Thomasi (BT) or Mutual Information (MI) metric (Hirschmuller, 2005). Although these methods are computationally more expensive than the ones mentioned for BM, they perform better under illumination differences, reflections and radiometric differences between the images. Instead of using an aggregation window like BM, SGM uses a set of 1D paths in 8 or 16 different directions, and sums the cost of all these paths for a certain disparity for a pixel. Semi-Global Block Matching (SGBM) is an algorithm that is available in the OpenCV library, which uses a combination of SGM and an aggregation window.

The first comparison between the two is shown in Figure 1.5. Using BM, the algorithms has difficulties to find exact matches for the road patches, as seen in Figure 1.5(b). Since multiple sections of the road look alike, the repetitive texture creates many outliers for disparity values. BM deals with these by simply deleting them out of the final disparity image (Lyrakis, 2019). SGBM results, shown in Figure 1.5(c), show better results for the road. However, it tends to fill the sky with incorrect large pixel disparities.

((a)) KITTI image 1



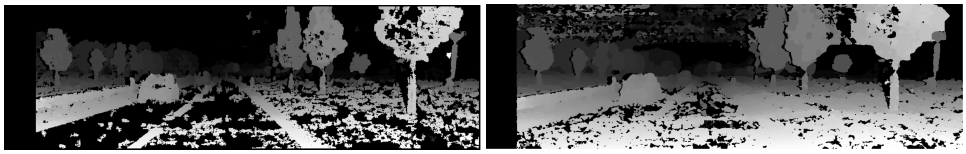
((b)) BM ((c)) SGBM

Figure 1.5: Disparity images generated for outdoor scene 1 (Lyrakis, 2019)

A similar observation can be made in Figure 1.6, where Figure 1.6(c) shows pixel disparities in the sky, top left of the image. BM has a lower performance in the low-textured areas, as seen in Figure 1.6(b), however it does not produce as many outliers in areas where there are no obstacles present.

Figure 1.7 shows an indoor image, to introduce a variety of environments. Again, it is clear from

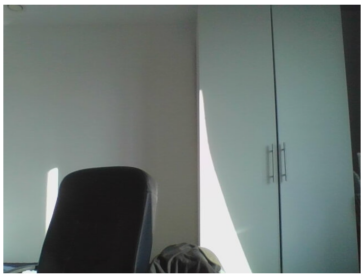
((a)) KITTI image 2



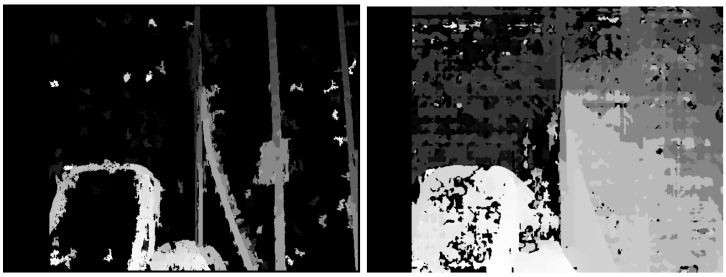
((b)) BM ((c)) SGBM

Figure 1.6: Disparity images generated for outdoor scene 2 (Lyrakis, 2019)

Figure 1.7(c) that SGBM performs better in low textured area. Two assumptions in chapter were that the flights take place during daylight, and in an outdoor environment. Since it is quite unlikely that the drone will be hovering above low-textured roads for cars, the BM algorithm will be suitable for this study. Furthermore, many of the holes in the disparity map will be filled using a C-space expansion, which will be explained in section 1.5.



((a)) Indoor image



((b)) BM ((c)) SGBM

Figure 1.7: Disparity images generated for indoor scene (Lyrakis, 2019)

1.3. Filters

Occlusions, untextured or repetitive image regions create challenges for the creation of a disparity map (van Dijk, 2020). Imperfect disparity maps mean that not every pixel will have the correct value. Furthermore, also aliasing, bluriness, image distortions, reflections and radiometric differences like image noise or vignetting have a negative effect on the resulting disparity map (Lyrakis, 2019). However, since they are independent of the chosen stereo algorithm, so they will not be discussed further in this study.

Multiple filters exist that try to find a solution for getting rid of outliers in the disparity map. Three post-processing filters that are also included in the OpenCV StereoBM function are a texture

filter, a uniqueness ratio filter and a speckle filter.

Texture filter: A texture filter gets rid of disparity values that were calculated for a region in the original image with low texture. The higher the threshold for this filter, the more disparity values will be discarded.

Uniqueness ratio filter: A uniqueness ratio filter checks if there are similar cost values for different disparity values. If the costs of the best and second best pixel match are similar, this would mean that the solution is not unique and the disparity value will be discarded.

Speckle filter: This filter tries to get rid of the speckles, or: pixels with high disparity values which are not so representative for the area around it. If there is a larger blob with pixels of similar high values, they will not be discarded.

Another post-processing filter, not included in the StereoBM function parameters, is the Left-Right consistency check. Here, a disparity image is created from the point of view of both cameras. If the disparity values for the same pixel differ between the images by more than a predefined threshold, the disparity values are either set to zero or another predefined value. This consistency check is useful for identifying occlusions (Lyrakis, 2019).

Setyawan et al. (2018) use another post-processing filter: a semi-global weighted least squares filter. The method shows good results in their paper, where it was tested on images with low variance noise. It does not present any results on images with large noise values which creates speckles. Since it roughly averages the values around the noise it is expected that this method will incorrectly increase the pixel values in the neighbourhood of the noisy pixel.

To conclude, all of these filters try to solve different types of the previously mentioned challenges for the creation of a disparity map. It cannot be said with certainty that one will be better than the other. Therefore, a careful parameter tuning of each of the filters will need to be performed to draw conclusions on their performance. Here, a multi-objective optimisation approach is needed, since the performance can be translated into three components: How good are the estimates? How many pixels obtain disparity values? And how fast can these computations be performed? More of this is discussed in section 5.1.

1.4. Uncertainty

Parameter selection of the previously mentioned filters is a time-consuming process, and the ideal parameters are different depending on the environment. Therefore, we could use an uncertainty prediction, where the uncertainty is "the expected value of the error between disparity estimation and actual disparity" (Lyrakis, 2019). 13 different features, including matching costs and disparity values, were used to obtain an estimate for this uncertainty. Some of these features are representative for the amount of texture, or noise for example. This resulted in a Decision Tree (DT), trained on the KITTI dataset, with Grid Search finding the best hyperparameters. The uncertainty values are used to filter the disparity map, and also to check the (un)certainty of collisions and obstacle-free escape points. A more detailed explanation is given in section 4.4. Using uncertainty maps gives more correct collision warnings (True Positives) than the classical BM or SGBM methods (Lyrakis, 2019).

Although this method is created to avoid parameter selection of the previously mentioned filters, still careful hyperparameter selection is a must for the DT. The images from KITTI focus on

autonomous driving, meaning that many will include a low-texture asphalt road. Drones are able to fly above a higher variety of services, therefore the DT would need to be trained again, using a dataset which includes a larger variety of images. Obtaining and training on such a dataset is a very time consuming approach, and the large variety within the dataset could cause a low performance estimator.

Instead of using all 13 features mentioned by Lyrakis (2019), a method to avoid retraining of the DT could be to simply take one feature, and base the estimation of the uncertainty on that. For example feature 11, representing the smoothness by looking at the average disparity difference between the current pixel and the pixels in its neighbourhood. In the KITTI dataset used, this feature had a Pearson correlation of 0.81 and a Spearman correlation of 0.74 with the disparity error. These correlations look at the linear and monotonic relationship between the variables respectively (Lyrakis, 2019). Using only feature 11 to create the uncertainty map may not give the best possible estimate, but it could be a satisfactory approximation. An approach for testing these hypotheses is provided in section 5.1.

1.5. Drone's dimensions

The drone is wider than the base of its cameras. In other words, given the disparity image from the stereo camera, a path planning algorithm could plan an obstacle-free path according to the cameras, yet the drone would still crash due to its propellers. This section will explain a configuration-space (c-space) expansion, along with some preliminary results.

1.5.1. C-space expansion

The drone's dimensions need to be taken into account, either when creating the disparity maps or when planning the path. The first option can be done with low computational costs and little memory, and has been implemented by Matthies et al. (2014), where the obstacles are expanded into disparity space. In short, obstacles near the drone will appear slightly larger since the drone needs to make a larger turn around it, such that the propellers will not hit the obstacle.

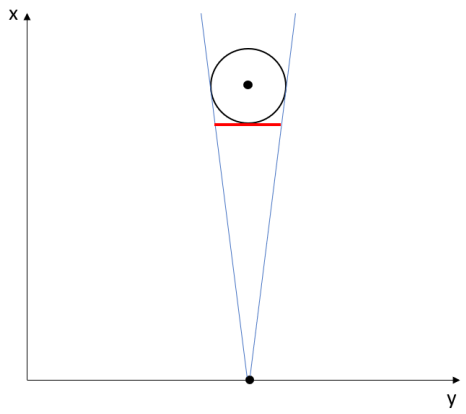


Figure 1.8: Drone's dimensions sketch (adapted from Matthies et al., 2014)

A sketch of the method is shown in Figure 1.8. This is a top view of the situation, where x denotes the direction straight ahead, and y is east of the drone. The bottom dot is the drone itself, and the dot in the middle of the circle is a pixel where the drone has encountered an obstacle. The circle drawn here is the 2D representation of the sphere that is created around this pixel, with a predefined expansion radius. This sphere creates the buffer around the obstacle that is needed for the drone to safely move around it. In the next step, a red line is constructed which represents the rectangle

that covers the complete area of the sphere from the camera's point of view. In fact, it will cover slightly more than the sphere would, since the correct approach would be to use ellipses. However, the rectangle method only needs an operation in horizontal and vertical direction, which results in a lower computational cost (Matthies et al., 2014).

1.5.2. Preliminary results

Figure 1.9 shows how this c-space expansion looks on a disparity image. Figure 1.9(a) is a disparity image of a nearby tree, with some of its branches, as well as some other trees in the distance. Figure 1.9(b) shows the result after expanding all the pixels. The tree looks wider and the branches cover a larger area. This was expected since the drone is not able to fly close to the original tree nor through the branches.

The only downside is that generating the c-space expansion takes a couple of seconds on a 256x144 image, even with the recommended look-up tables of the original paper (Matthies et al., 2014). However, the original paper implemented the method in C/C++, and the larger runtime in Python was not representative for the one in C/C++. Unfortunately, some testing will need to be done in Python as well, so if this problem remains, it is possible to perform the c-space expansion only on a selection of pixels. Figure 1.9(c) shows the result after applying the expansion to only 1/25 of all pixels. It only looks at every 5th column and row. This creates a slightly more inaccurate disparity image, but allows for real-time implementation in Python. Another option would be to alter the clockspeed of the simulation software used.

((a)) Disparity image tree ((b)) C-space expansion ((c)) C-space expansion (1/25)

Figure 1.9: Disparity image along with its c-space expansion

1.6. Discussion

The distance measurement sensor that will be used is a stereo camera, since the environment is assumed outside and during daylight, hence it has enough texture. Methods that use sound or radio waves tend to have difficulties with curved boundaries, and are bulky compared to the stereo camera.

The two cameras allow quick depth perception by comparing the two images through a local dense approach called BM. BM is preferred over SGBM since it does not produce as many outliers in low-textured areas like the sky, plus it performs faster calculations. BM compares two aggregation windows between the two images based on pixel intensity, and returns a value resulting from an evaluation metric. SAD is expected to yield the fastest calculations due to its simplicity, compared to other metrics like NCC, SSD, RT and CT. Testing this is left for further thesis work. It could be that the differences in pixel intensities between the two images is too large for correct matching due to radiometric differences or reflections. In that case, a BM implementation using BT or MI will need to be tested, since they perform better under these circumstances.

Regarding the filters, the combination of parameters yielding the best results needs to be found.

The performance can be expressed by the quality of the estimates, the amount of estimates and the time it takes to calculate them. The filters that will be considered are: texture, uniqueness ratio and speckle filter, as well as an Left-Right consistency check.

If, after careful tuning of these parameters, the disparity image remains noisy, the uncertainty map will be created. Two methods were proposed: the first one is to train a DT on a new dataset, which includes a larger variery of images. The second option is to base the calculation of the uncertainty on one feature only, which is the average disparity difference between a pixel and its surroundings.

Lastly, the drone's dimensions are taken into account by expanding in disparity space. Every pixel is translated into its world coordinates, and a sphere of a predefined radius is drawn around it. Next, a rectangle is drawn between the drone and the circle, which covers the area of this circle. In this way, the drone can be modelled as a point and it can safely choose a pixel with a small disparity value to travel to.

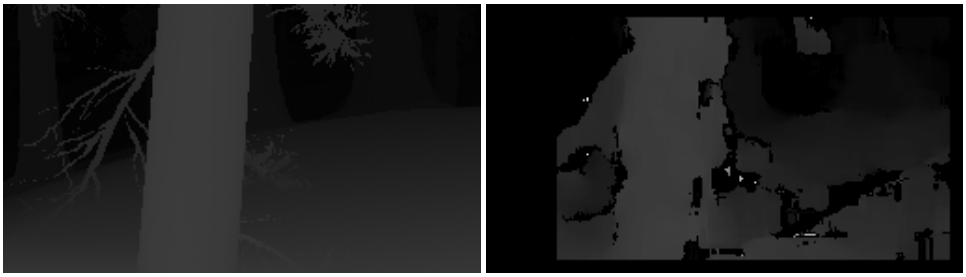
2

Environment Representation

In this chapter, methods to store information about the environment will be discussed. The previous chapter has shown methods on how to create disparity maps, which can be used to build maps, where both obstacles and the workspace can have their own representation. Such a map is built up from multiple local percepts over time, where a local percept is a disparity image captured by the stereo camera at one time instant. If the map has been built completely, and the position of all the obstacles is known, this map is called a global map.

2.1. Obstacle representation

Figure 2.1 shows a comparison between a true disparity image, and the disparity image obtained from the Block Matching algorithm. Here, the drone flew through a forest and encountered a tree. The BM parameters still need to be tuned to obtain better results than shown in Figure 2.1(b). Suppose that improving the parameters leads to a result closer to Figure 2.1(a). This image can either be used directly for planning the next waypoint, or to start building a map of the environment, which is then used to plan the next waypoint.



((a)) True disparity image ((b)) Block Matching result

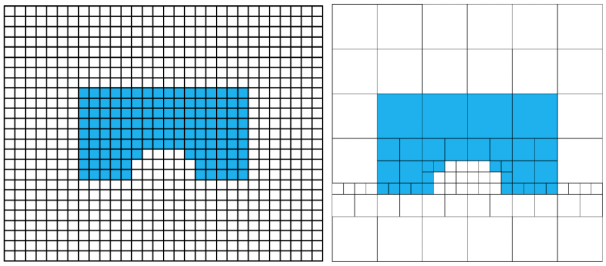
Figure 2.1: Differences between the true disparity and stereo camera disparity calculation

The different types of maps can be divided into three classes: Image space maps, discretised space maps and continuous space maps (van Dijk, 2020). The image space maps use one image only, and plan a next waypoint using the information of the depth of each pixel. Since it does not build a map using multiple measurements, it is considered a mapless method in this study. Discretised- and continuous space maps build a 3D map, using the information obtained through multiple images. The advantageous of such a map is that the drone is less likely to get stuck in local minima, since it memorises what the environment looks like and how it can get out or avoid them. The disadvantage of these maps are the extra computational expenses that are paired with it.

2.1.1. Discretised space

In a discretised space map, the environment is divided into many cells, and each cell is checked for obstacle presence. These cells can by either shaped by a grid, or a cell tree. These methods are also referred to as occupancy grids. A grid uses cells of identical size, as shown in Figure 2.2(a), and the performance is dependent on the size of these cells. The smaller the cells, the more precise it can model the environment, but the more computationally demanding the calculations become. Furthermore, larger cells are sensitive to noisy data, since they will be marked as occupied if only one data point falls within. A cell tree uses different sizes for the cells, as shown in Figure 2.2(b). If there is no obstacle present, cell tree uses relatively large cells. When an obstacle is present, these cells are divided into 4 smaller cells to highlight details. This process can be repeated multiple times.

Figure 2.2 shows the 2D top view representation for simplicity. The map is extended into 3D in a similar way.



((a)) Grid ((b)) Cell tree

Figure 2.2: Obstacle representations in discretised space (adapted from Shin and Chae (2020))

2.1.2. Continuous space

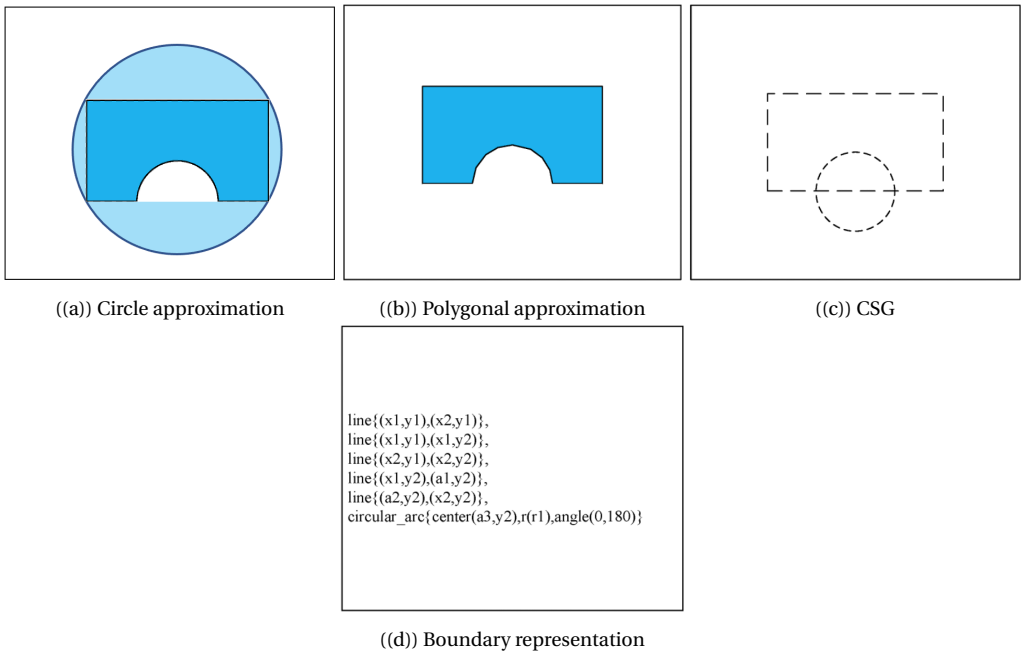
Continuous space maps do not divide the environment into a grid of cells, but assign coordinates to the location of obstacles. The typical form of this is a point cloud, which is a large collection of all the 3D points found where an obstacle is present, as shown in Figure 2.3. Since, geometrically speaking, points do not have a volume, small spheres need to be drawn around the points to allow path planning algorithms to avoid them.

Figure 2.3: Obstacle representation: Point cloud

Since point clouds have a high computational complexity (van Dijk, 2020), it might be wiser to locate the boundaries of an obstacle and represent those in a map. There are different approaches for this. Obstacles can be approximated by spheres, as shown in Figure 2.4(a). This will introduce an extra safety margin around the obstacles. However, this might not be necessary if the disparity space expansion from section 1.5 is used. The boundaries could also be represented using polyhedrons, as shown in Figure 2.4(b). This method creates a polyhedron from multiple polygons, which are straight line surfaces. Instead of using straight line surfaces, Constructive Solid Geometry (CSG) uses a combination of shapes to represent the obstacles, as shown in Figure 2.4(c). Furthermore, the boundaries of an obstacle can be represented by their vertices and links, as shown in Figure 2.4(d).

2.1.3. Discussion

These are all the different obstacle representations that will be used for the workspace representations in the next section. The end of that section will provide a short discussion on which types are favourable over others, and if it is worth diving deeper into path planning algorithms that use these types of representations.



((a)) Circle approximation ((b)) Polygonal approximation ((c)) CSG

((d)) Boundary representation

Figure 2.4: Obstacle representations in continuous space (adapted from Shin and Chae (2020))

2.2. Workspace representation

The workspace is defined by the reachable configuration space, where the configuration space is the space of all possible configurations that the drone can achieve (Lynch, 2017). This section will discuss different workspace representations. The mapping of the workspace is assumed to be performed over multiple measurements. This yields two different possibilities: creating a discretised space map or a continuous space map.

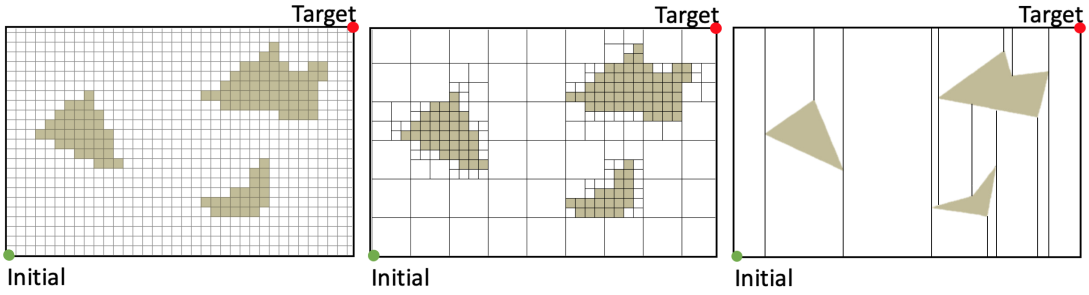
2.2.1. Discretised space

When using occupancy grids for obstacle representation, a similar representation is given to the workspace, as shown in Figure 2.5(a). Every cell is checked for obstacle presence, meaning that the cells that do not contain obstacles can be seen as the configuration space. For the grid representation, the cell size is known beforehand and does not need to be changed. A disadvantage is that there will always be a trade-off between computational expenses and ability to represent the workspace in detail. Therefore, a larger volume stored for the workspace can not be used with a high accuracy (Dubey et al., 2017), unless computational expenses are allowed to increase. Perrollaz et al. (2012) used such occupancy grids to plan an obstacle-free path for a vehicle on the road. Since cars are unlikely to drastically change in altitude, only 1.8 m of height was included in the grid, as well as a depth and width of 20 m. Their method runs on average in 150 ms, on a 3.4 GHz processor with 8 GB RAM. Since the drone has less than half of that processing power, and it needs more than 1.8 m of height difference to be included in the grid, this method is computationally too expensive, since it will result in only a few fps.

The cell tree method tries to improve computational expenses, by only assigning smaller cells in areas where detailed information is necessary, as shown in Figure 2.5(b). This results in fewer cells, hence fewer path possibilities, making it easier for path planning algorithms. The stereo camera has good ability to see distances far ahead, but it tends to be noisy. The noise will unnecessarily break up many of the large cells into smaller cells. Therefore, the fact that the camera can see depth far away is not utilised to its potential.

Other than grid based and cell tree occupancy grids, the workspace can be divided into cells using exact cell decomposition, as shown in Figure 2.5(c) (Patle et al., 2019). Here, the workspace is divided into multiple cuboids stacked behind each other from the drone's view. In case of obstacle presence within a cell, it is divided into smaller polyhedral cells.

A cell without an obstacle is called a pure cell. Concatenating adjacent pure cells leads to a path from initial to target position. The disadvantage of occupancy grids is that every cell needs to be constantly checked for obstacle presence. This lead to a high computational complexity (van Dijk, 2020). Compared to a continuous space map, the number of feasible paths is only reduced if the possible waypoints are discretised. Else, the drone would still be able to fly anywhere within a cell. However, even if the waypoints are discretised, for example to the centre of each cell, the planned path jumps from the centre of one cell to another, causing unnecessary movement.



((a)) Grid ((b)) Cell tree ((c)) Exact CD

Figure 2.5: Discretised workspace representations (adapted from Patle et al. (2019))

2.2.2. Continuous space

In the continuous space map, the obstacles can be represented by either a point cloud or a boundary representation. Initially the workspace will consist of all possible coordinates that do not include the coordinates of the obstacles. This means that the obstacle-free part of the map is simply an empty space. One possibility is to leave it as an empty space and perform path planning, another is to generate nodes and links inside this space to create a large roadmap. This roadmap will consist of many 1D lines, and can be created in different ways.

Empty continuous space

After obtaining the obstacle representation, no computation is necessary to build this workspace representation. The downside is that there are infinitely many possible paths within this large obstacle-free space, since any coordinate can be part of the path. Therefore, the necessary path planning algorithm might require more computation, compared to other workspace representations.

Roadmap continuous space

The workspace can also be represented using nodes and links, also referred to as the Roadmap, Retraction, Skeleton, or Highway approach (Masehian and Sedighizadeh, 2007). Using nodes and links, the number of possible configurations is reduced to a network of lines, restricting the drone's motion. Having a more restricted motion space will result in a lower computational cost, since fewer configurations are being considered. However, some roadmaps might require to be smoothed, since the drone is not able to follow the straight line segments perfectly (Shin and Chae, 2020). It can fly

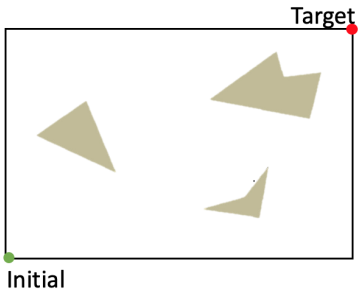
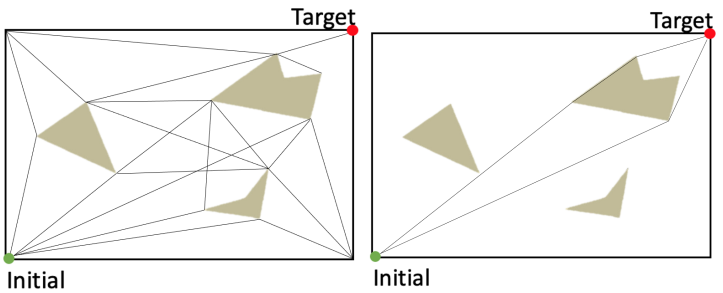


Figure 2.6: Empty continuous workspace (adapted from Patle et al. (2019))

past the nodes within a certain radius, but if the next link is a tight turn, the drone might hit an obstacle if it flies a small distance farther by taking a wider turn. Furthermore, the fact that not every coordinate can be traversed anymore might lead to paths that are not globally optimal. There are different methods for deciding where to locate the nodes and links in the workspace, of which the following five will be discussed: Visibility Graph, CEG, Voronoi Diagram, PRM and RRT.

Visibility Graph: Figure 2.7(a) shows the Visibility Graph (VG) method. Here, links are used to connect the mutually visible vertices of the obstacles with each other, as well as with the initial and target position. This will create a large network of possible obstacle-free paths. In 2D the method seems rather straightforward, but in 3D with local perception some implications arise. For example, it is difficult to select the corner of an obstacle if the obstacle is a tree and you only see the front part of it. Furthermore, the back side of the obstacle is not visible to the drone unless it turns around, so it is difficult to select a vertex which can be connected to other obstacles farther away. Also, this method travels from obstacle corner to obstacle corner, which is not necessarily the most direct path to the target.

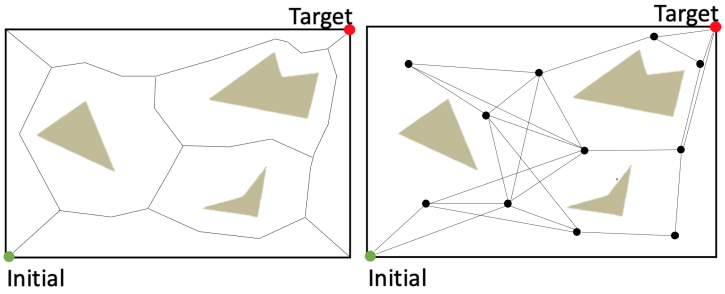


((a)) Visibility graph ((b)) Convex-Edges-Graph

Figure 2.7: Workspace roadmaps visibility graph and CEG (adapted from Patle et al. (2019))

Convex-Edges-Graph: Whereas VG looks at combining obstacle vertices, the Convex-Edges-Graph (CEG) only looks at the obstacle which blocks the direct path to the target, as shown in Figure 2.7(b). It draws its nodes anywhere along the obstacle boundary. For simplicity, the figure only shows the side edges. After, it creates links from the drone's current position towards these nodes. The difficulty here lies in the tracking of the obstacle boundary, but the advantage is that the path is shorter than for the VG. This is because the drone does not have to travel to the vertex of an obstacle, but can pass it anywhere along the boundary. CEG has only been applied for reactive approaches (Kamon et al., 1999). In other words, the node selection was based on the current best solution, not taking into account other obstacles. The advantage is that no graph search algorithm is necessary in this reactive approach.

Voronoi Diagram: To include a safety margin from the obstacles, the Voronoi Diagram (VD) was introduced (Patle et al., 2019). It is shown in Figure 2.8(a). The links drawn in the graph have an equal distance to the obstacles around it. This means that if the drone moves along this line, it will keep maximum possible distance from the obstacles and boundary of the map, since the obstacles outside the map are unknown. There are also rectilinear voronoi diagrams, used for path planning in only 8 directions (Masehian and Sedighizadeh, 2007). A disadvantage of the VD is that the chosen route has a large deviation from the global optimum, since it follows these equidistant lines.



((a)) Voronoi diagram ((b)) PRM

Figure 2.8: Workspace roadmaps Voronoi diagram and PRM (adapted from Shin and Chae (2020))

PRM: The next option, shown in Figure 2.8(b), shows the Probabilistic RoadMap (PRM). Nodes are constructed randomly throughout the obstacle-free space and are connected using a local planner when the distance between them is smaller than a certain metric (Kavraki et al., 1996). The local planner checks a number of equally spaced coordinates between the two vertices and if none result in a collision, the path is considered obstacle-free. The nodes are clustered into connected components. To prevent infinite connected components, a new randomly generated vertex is not connected to another vertex if they share the same connected component (Kavraki et al., 1996). This also avoids unnecessary computations (Karaman and Frazzoli, 2011). The algorithm ends once the desired number of nodes is reached. PRM is probabilistically complete, but not asymptotically optimal: as the number of nodes approaches infinity, the probability that the method finds a path, if one exists, approaches 1, but it will not necessarily give the optimal path (Karaman and Frazzoli, 2011).

The simplified PRM (sPRM) includes the initial location of the robot as a vertex, and allows for a link between nodes that have the same connected component. This results in an asymptotically optimal algorithm, but it comes at a computational cost. Furthermore, there are different PRM versions based on the choices made when selecting new connections (Karaman and Frazzoli, 2011). The k-nearest PRM (K-PRM), as the name implies, chooses the k-nearest neighbouring vertices to the one being considered. The value for k can be chosen by the user, but for any k the method is not asymptotically optimal. Variable-radius PRM decreases the radius used when searching for new vertices as the total number of vertices increases. In this way, first the so-called highways are created to travel across the map quickly, followed by the smaller roads to find the exact location of the target. PRM* also makes use of this variable-radius, and combines it with the property of sPRM that allows links between all vertices. Since PRM* has many edges, it will create smooths paths, but it is computationally more expensive. Lazy PRM* decreases the computational expenses, by only activating the local planner on nodes and edges that are part of the current best solution, found by node-based algorithm A* (Lundin and Dath, 2018). Other extensions of PRM exist, but the main takeaway is that PRM generates nodes randomly and tries to link them together.

The advantage of PRM is that the generation of nodes is fast, since there is no need to place

nodes exactly on obstacle boundaries or equidistant lines. PRM can generate short paths, with lower memory requirements than RRT (Lundin and Dath, 2018). RRT will be explained in the next paragraph. Lazy PRM* only uses half of the memory that PRM uses, and produces similar paths. However, PRM also has a few disadvantages. Firstly, PRM produces many different possible paths, but does not choose one. A graph search algorithm is needed to plan a path. Heuristic algorithms like A* are able to do so fast, so this is not a large problem (Hart et al., 1968). Secondly, since the map is constantly being updated with incoming information about new obstacles, the graph search algorithm needs to be rerun constantly. Now, with the arrival of D* Lite this disadvantage has been mitigated, since many pre-calculated paths are stored in memory, and only the links that changed weight are updated (Karur et al., 2021). But thirdly, the random generation of points results in a path that is likely to not represent global optimum. Fourthly, when traversing from one point to another, the direction of the drone is constantly changing. Ideally, the drone should rotate as little as possible to obtain images that are in the direction of flight. Furthermore, movement should be limited since there is no visual perception to the side of the drone.

RRT: Lastly, RRT generates a slightly different roadmap from PRM. When a new random node is generated, it is simply connected to the nearest node only. There is no need to check for connected components of other nodes. This means that, once a node is within a certain threshold from the target position, there is only one path possible, starting from the initial position. Therefore, no node-based path planning (graph search) algorithm is required to search the nodes, and RRT can be regarded as both a workspace representation and a map-based path planning algorithm. This is a large advantage over PRM, which needs to rerun a graph search algorithm whenever new information about the environment becomes available. An example of an RRT construction is shown in Figure 2.9.

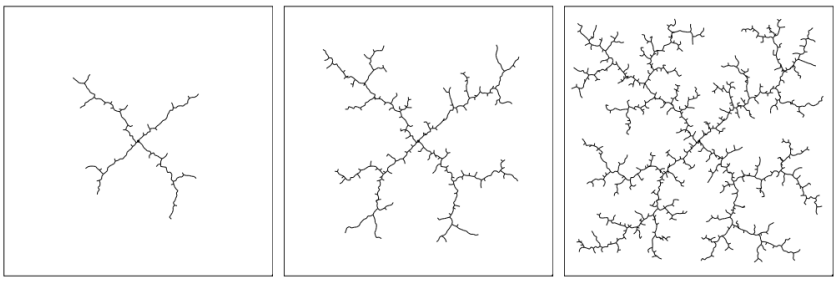


Figure 2.9: Example of RRT construction (LaValle et al., 1998)

Even though the roadmap creation is different, RRT has similar characteristics to PRM: it is probabilistically complete, and not asymptotically optimal (Karaman and Frazzoli, 2011). Furthermore, since RRT only looks at one connection with another node, non-holonomic and kinodynamic constraints can be introduced (LaValle et al., 1998). The randomly generated node attempts to connect to the nearest node. If this connection is not possible, due to either obstacle presence or other constraints, a new node is generated on the line between the nearest node and the random node. This calculation of this new node is done by a so called state transition equation, which is a differential equation which includes current state and inputs. It can be solved with either Euler integration or Runge-Kutta for example. PRM has no such possibility, since one node is sometimes connected to multiple other nodes, making it impossible to account for these constraints from multiple directions.

Another advantage is that the generation of random nodes is heavily biased towards the unexplored region, which results in a uniform coverage of the workspace (Lundin and Dath, 2018).

However, a disadvantage is that the method can get trapped in environments where there is only a small opening to pass through (Karaman and Frazzoli, 2011). This is a similar problem for PRM, since both methods generate nodes randomly, and the chance that one node perfectly finds a tiny opening in between obstacles is small. For example, RRT and PRM would have trouble finding a way out of a wasp catcher, which is an object designed for easy entry but difficult escape.

Dynamic Domain RRT thought of a way to get out of these small openings, by assigning a maximum radius for the connection to another node (L. Yang et al., 2016). New nodes can only be connected if they are located within radius r from a nearby node. Thus, the process of generating random nodes is repeated until a node lies within this radius of another node (Yershova et al., 2005). If the connection succeeds, then the value of r is set to infinity. Once a connection fails, the radius will be set to a default value r. In the example of the wasp catcher, more nodes and links will be created inside. This will increase the chance that a node is created near the opening, which will increase the change that it connects to a node outside of it. The downside is that the path will not necessarily be optimal, nor smooth. Furthermore, in outdoor navigation the chance that the drone will get trapped in these wasp catcher-like situations is small.

Another disadvantage of RRT, similar to PRM, is that the generated roadmap will consist of many short path sections, resulting in a drone which is constantly changing direction. To improve the result quality, RRT* was introduced (Karaman and Frazzoli, 2011). To explain how RRT* works, first Rapidly-exploring Random Graph (RRG) needs to be explained. The difference between RRG and RRT is that RRG does not stop after finding a connection with the nearest node, but tries to connect to any other node within radius r. Thus, RRG and RRT have the same vertices, but the connections in RRT are a subset of the connections in RRG. The size of r decreases with increasing amount of generated nodes. The extra connections in RRG compared to RRT causes the graph to be undirected, meaning that it might contain cycles. This will create a complex roadmap that has multiple paths from initial to target position, like PRM. Now, RRT* tries to find the most promising connections within RRG (Karaman and Frazzoli, 2011). For every node, it only keeps the link which provides a path to initial position with the smallest distance travelled. This creates a directed graph, since all the longer paths are discarded from memory. Unlike RRT, RRT* is asymptotically optimal, and it has the same complexity in time and space.

These sampling based methods are powerful since they are able to find paths through cluttered, maze-like, environments. RRT* produces more optimal paths in terms of deviation from nominal path than PRM, but it comes at a computational cost. RRT* tends to quickly take up more than 200 MB RAM, even in simple environments (Lundin and Dath, 2018). But the main disadvantage for these sampling based methods is that the nodes are generated randomly, and are unlikely to represent global optimum. This is only the case for RRT* when the number of nodes reaches infinity. A probabilistic method would be good for a maze-like situation, where small openings need to be found far away from the drone. However, these maze-like situations are not very representative for outdoor environments like a forest or a neighbourhood, where the advantages of probabilistic methods are not utilised to its potential.

2.2.3. Discussion

Table 2.1 summarises the advantages and disadvantages of the different environment representations mentioned in the paragraphs above. Firstly, the empty workspace representation does not alter any of the workspace after the obstacles have been represented. This saves some computation compared to the other methods, but a path planning algorithm might have more trouble finding a path, since there are infinitely many. To examine this, the next chapter will discuss some common

path planning algorithms used in this type of map.

Table 2.1: Comparing the different workspace representations mentioned above

	Advantages	**Disadvantages**
Empty	• Only obstacle representation	• Infinitely many feasible paths
Occupancy	• Less memory needed than point cloud • Ability to connect adjacent free cells	• Computationally expensive • Number of feasible paths not reduced
Grid	• Cell size known	• Performance depends on cell size
Cell tree	• Fewer cells than grid	• Sensitive to noise
CD	• Fewer cells than cell tree	• Harder to locate
Roadmap	• Workspace reduced to 1D lines	• Requires memory storage • Travels from obstacle to obstacle
VG	• Fewer nodes than PRM, RRT	• Need to find obstacle borders
CEG	• Shorter paths than VG	• Need to find obstacle borders
VD	• Safety margin from obstacles	• Need to find obstacle borders • Larger deviation from nominal path
PRM	• Fast and memory is minimal	• Randomness of nodes is suboptimal • Gets stuck with small openings
RRT	• Creates one path to target • Introduces nonholonomic constraints	• Randomness of nodes is suboptimal • Gets stuck with small opening

Secondly, the occupancy grids require more computation, since every pixel's world coordinate is checked in which cell they are located. The advantage is that many individual points can be represented by one large cell, which decreases the required memory. CD has the disadvantage that the obstacle contours need to be tracked precisely before the cell can be drawn. Since the incoming data is noisy, this will be a difficult process and the grid or cell tree methods will be preferred. The cell tree method will most probably have fewer cells than the grid method, but extra bytes need to be included to define the current size of the cells. Even if the total required memory is less than the grid method, extra computations are needed to find out whether or not the cells should be split into smaller cells. Lastly, the occupancy grids do not take great advantage of the fact that a stereo camera can see large depths, due to the sensitiveness of the noise.

Thirdly, the roadmaps require more computation than both other methods, but the result is that the workspace is reduced to a network of 1D lines. The CEG, VG and VD all need to precisely track obstacle borders, which is difficult in noisy data. However, their nodes are placed in a logical way, and not through random node generation as in PRM and RRT. VD creates a path with a large deviation from the nominal path, since it tries to go around obstacle with the largest margin possible. To limit the deviation, either VG or CEG is preferred. When comparing these two, CEG finds shorter paths than VG. They use the same principle of connecting obstacles, but CEG uses the nodes that have the smallest deviation from the nominal path. When comparing the two probabilistic methods, RRT uses a factor 1.4 more memory on average than PRM (Lundin and Dath, 2018). However, it finds a solution most of the time in the order of a few hundreds of a second, while PRM needs tens of seconds and sometimes even a second. Another advantage of RRT is that the drone's dynamics can be taken into account in the generation of the RRT graph. However, RRT is not asymptotically optimal and RRT* is computationally expensive. These methods show their potential in a maze-like environment, but in an outdoor environment it is preferred to choose waypoints that are not randomly generated, since the escape points can be seen from the current point of view in most cases.

In conclusion, empty workspaces are computationally cheap compared to the others and are in-

teresting for path planning purposes. These can consist of either point clouds or, ideally, boundary representations. Occupancy grids are computationally expensive and are more sensitive to noise, since one noisy data point can mark a complete cell as occupied. Regarding roadmaps, CEG seemed the most promising. However, the performance of this algorithm also depends on the quality of the disparity image. The same goes for the boundary representations for the empty workspaces. The discussion on CEG and image noise will be continued in subsection 4.3.3.

3

Map-based path planning

This section holds path planning algorithms which use maps to create a path. Graph searching algorithms will not be discussed in detail since the previous chapter concluded that RRT will be favoured over PRM, and even RRT is unlikely to be chosen as the best algorithm due to its generation of random waypoints. Furthermore, CEG was preferred over VG and VD, and the best node for CEG can be calculated using a cost function. The remaining map-based path planning algorithms can be divided into two categories: optimisation under constraints and potential field methods. These methods can be used for the empty workspaces discussed before. Any promising algorithms that use global maps will need a discussion on whether it is possible and advantageous to run these algorithm on local maps.

Machine learning methods will be discussed in chapter 4, since they are mostly limited to reactive approaches in the literature, with or without a target location. The existing machine learning methods are either not applicable because they focus on graph searching methods (e.g. Chen and Chiu, 2015)), or are computationally expensive.

Before discussing the different path planning algorithms, a comment has to be made about the missing common benchmark to evaluate these algorithms (van Dijk, 2020, Goerzen et al., 2010, Lyrakis, 2019). Many authors create their own testing environment and demonstrate the performance of their algorithm. Lately, efforts have been made to produce these types of benchmarks (e.g. Nous et al., 2016). However, the path planning algorithms discussed in this literature study do not use this evaluation metric and due to time constraints it is not possible to implement and evaluate every single algorithm. In the rest of this study, the path planning algorithms will be compared based on their relative computational expenses and performance. Here, performance can for example be translated into path length, probability of collision or the tendency to get stuck in local minima. Later, in section 5.6, it will be explained how the performance evaluation metrics used by Nous et al. (2016) will be used to evaluate the proposed algorithm.

3.1. Optimisation under constraints

The goal of these algorithms is to plan an optimised path from initial to target position, while taking certain constraints into account. These can be dynamic constraints which restrict movement of the drone, or constraints including the location of obstacles which need to be avoided. The purpose of this cost function could be to minimise distance travelled or energy consumed (Masehian and Sedighizadeh, 2007). Using optimisation methods, an optimal solution is obtained for this cost function. This section holds the following methods: mathematical programming, curve based, gradient-based search, heuristic and meta-heuristic algorithms.

3.1.1. Mathematical Programming

In mathematical programming, kinematic constraints model the environment, and dynamic constraints model the robot. These dynamic constraints normally satisfy velocity and acceleration bounds, but can also be lower level, like torque constraints on the propellers. Mathematical programming uses the empty workspace, where obstacles are represented using the boundary representation for the obstacles. The planned path consists of straight line segments between the nodes. Most methods use either splines or polynomial approximations to calculate the actual trajectories

of the drone.

K. Culligan et al. (2007) have been successful in testing their variable time-step Mixed Integer Linear Programming (MILP) method in an indoor environment. Their method is based on a paper that uses an iterative MILP, which tries to find a path without including obstacle knowledge, and only after that it checks whether the planned path has hit an obstacle. If it does, the path is re-planned, and after a few iterations the obstacle-free path is found. The variable time step means that waypoints nearby are calculated with a small time-step difference, and it increases towards farther away points. This is in line with the available information about the environment, which is better near the drone. The solution is calculated within a second on a 2 GHz processor, with 1 GB RAM (K. F. Culligan, 2006).

The authors are able to perform linear programming, since all of their obstacles are assumed to be box-shaped. This means that their constraints are as simple as: x should be larger than or equal to value a, and x should be smaller than or equal to value b, and similarly for y and z. For multiple obstacles, multiple constraints can be added in a similar fashion. For other convex shapes it should still be possible to write their constraints in terms of x, y and z, but for concave obstacles the MILP method does not hold. This means that for obstacles like a tree, where the top part is wide and the bottom part is thin, this method will not work. To counteract this problem, an obstacle (e.g. a tree) could be divided into two box shaped obstacles for the top and bottom. However, an extra algorithm would be needed to divide all concave obstacles into multiple boxes. Also, there would be a trade-off for deciding how many boxes per obstacle are allowed. More boxes can approximate the obstacle closely, but increases the computation needed for the MILP constraints. Fewer boxes might still allow for a fast algorithm, but will define some of the obstacle-free space as occupied. Even though MILP is an optimisation technique, due to the mentioned implementation difficulties the algorithm is likely to plan paths with a larger deviation from obstacles due to the concave obstacle approach, leading to a longer path.

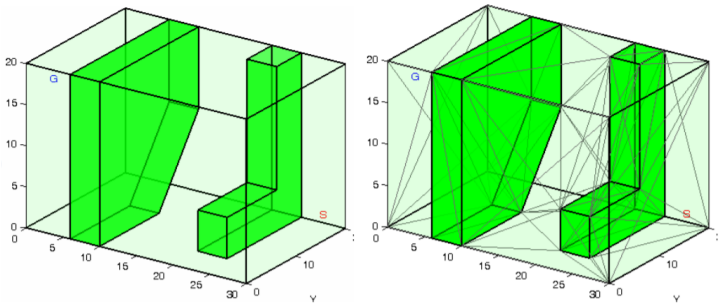
Miller et al. (2011) have found a way to find a path locally using a mathematical programming method called Binary Integer Programming (BIP) (Wu et al., 2019). The method is designed for a UAV to avoid threats. These threats may include missiles, detection by enemy sensors or obstacle collision. A Boundary Value Problem (BVP) is formed to calculate the optimal path. However, in 3D cluttered environments the number of constraints increases rapidly, which results in this being a computationally expensive algorithm (L. Yang et al., 2016).

According to L. Yang et al. (2016), five different papers attempted to improve the high time complexity in 3D. Two of these use a fleet of drones which interact with each other, hence not applicable to this study. The third paper avoids obstacles using MILP (Richards and How, 2002). However, this paper was published 5 years before the previously mentioned variable time-step MILP and does not improve on any high time complexity. Furthermore, it only deals with rectangular obstacles in 2D. The paper uses a 1 GHz processor with 256 MB RAM. For a relative simple environment with three rectangular obstacles within the next 10 meters, the algorithm takes 11 seconds on average, with outliers up to 68 seconds. Hence, this algorithm is not a viable option for a small drone.

The fourth paper introduced several techniques to limit the computational effort of MILP, such as a receding horizon framework and multiple timescales (Ma and Miller, 2006). With receding horizon, the next waypoint is moved closer than the target position, to reduce the computations needed. Furthermore, there are two timescales: a long and a short one. The long timescale plans the path broadly for the next 10 time steps at 0.5 second intervals, and the short timescale plans the path precisely for the next 10 time steps at 0.2 second intervals. The paper calculates the long timescale

on a 1.8 GHz processor, in the order of 1 second. The short timescale is run on a 200 MHz processor, in the order of 5 seconds. This means that it takes 5 seconds to calculate the detailed path that will be flown in the next 2 seconds. The paper mentions that, although it does not meet the required time, it is "only" one order of magnitude away. The drone used in this study has a processor of 1.34 GHz, which would need to be used for both the short and long timescale, which will likely not result in more than 1 fps on the drone.

The fifth used Binary Integer Programming (BIP) and create a 3D Delaunay triangulation of a part of the configuration space with multiple obstacles (Masehian and Habibi, 2007). An example of 3D Delaunay triangulation is shown in Figure 3.1, which divides the workspace into tetrahedrons. A tetrahedron is a polyhedron with 4 faces. The tetrahedrons inside the obstacles are removed from the set of all tetrahedrons, and a path is planned using the free tetrahedrons that are left in the set. The method is promising when using a global map, since solutions are found in a few hundreds of a second. Since it is difficult to precisely track the corners of obstacles, for local maps it is hard to define the location of the tetrahedrons. This method could be used for occupancy grids, but then the resulting path would be a zigzag of 90 degree turns through space, and the algorithm will not have any benefit over probabilistic roadmaps or trees.



((a)) Workspace example ((b)) 3D Delaunay Triangulation

Figure 3.1: 3D Delaunay Triangulation of the workspace (Masehian and Habibi, 2007)

3.1.2. Curve based

Curve based algorithms use polynomial equations to plan a path, taking into account constraints for kinematics, safety and optimum path. The difference with Mathematical Programming is that for curve based algorithms the paths are based on polynomial equations, while for mathematical these are straight line segments, followed by a smoothing algorithm which can curve based, splines and the like.

The curve based algorithms started with Dubins car (LaValle, 2006). A car has a minimal turning radius which has to be taken into account in path planning. This means that all segments of the path from initial to target position have a minimal turning radius, such that the car can execute the path. The idea has been further expanded for drones, by including altitude and wind, as well as clothoid arcs to create smoother paths (Triharminto et al., 2013). A clothoid is similar to an arc, but the radius of curvature changes linearly along its length. The term is sometimes referred to as Cornu Spiral or Euler Spiral (Dai and Cochran, 2009). Triharminto et al. (2013) also mention Phytagorian Hodograph (PH), which is a similar method to Dubins. A PH curve makes use of hodographs to describe the velocity. The useful property of these functions is that they can easily be integrated to obtain path length, which can be used to evaluate the performance of the algorithm.

The curve based algorithms have two applications. The first one is path planning from initial position to target, and the second one is to smooth the paths produced by other algorithms. Due to their simplicity, Dubins method is the current most used method to smooth a path. It can for example take turning radius and rate of climb limitations into account (De Filippis et al., 2012).

A disadvantage for the curve based algorithms is that they are created for static environments, and have difficulty adapting to dynamic environments (Triharminto et al., 2013). Due to their high time complexity, they are not considered as a viable option. Perhaps in a later stage they can be applied to smooth the path generated by another algorithm, but splines are also a cheap option.

3.1.3. Gradient-based search algorithms

Gradient-based search methods are optimisation techniques that use the gradient of the current solution to find the direction in which better solutions might be. Examples are the bisection method, Newton's method, golden section search, Frank-Wolfe and Sequential Unconstrained Minimization Technique (SUMT) (Hillier and Liebermann, 2014). In map-based path planning, it can defined how good the path is in terms of a cost function, yet the cost function can not be evaluated for a certain path to obtain a gradient.

However, these gradient-based methods can be used for solving for a system with non-holonomic constraints (Divelbiss and Wen, 1993). Inequality constraints were introduced that reduce all paths from initial to target position to the feasible paths, based on kinematic constraints. The authors defined a constraint equation that is a non-linear function of the control inputs, for which an optimum is found using Newton's method. Solving these equations using gradient-based search is a computationally expensive process, and therefore the paper assumes an environment where the obstacles are known.

To improve on the path length, Duleba and Sasiadek, 2003 extends the previous method by including an energy term which needs to be optimised. This energy term is related to the energy needed to control the robot, and is a complex function which looks at the energy of the harmonics of the trajectories which are represented in Fourier basis. Unfortunately this method only shows the theoretical approach and does not perform experiments. Moreover, the method is in 2D and it does not even avoid obstacles. It is only able to calculate a trajectory from point A to B, and mentions that the previously mentioned paper is able to avoid obstacles, so an extension to the current paper is to include that. However, the previous algorithm was already computationally expensive by itself, hence the combination will not give a faster solution. Furthermore, for these gradient-based search algorithms it is possible that they end up at a local optima (Duleba and Sasiadek, 2003). The next section on heuristic algorithms try to include methods that avoid these local optima, and improve the computational efficiency.

3.1.4. Heuristic algorithms

Heuristic methods are methods that try to discover the optimum by trial and error. Here, global optima are not always guaranteed, but the solutions are found quickly and can be of good quality. Examples of heuristic methods are A*, Nelder-Mead, machine learning and fuzzy logic. The latter two are discussed in chapter 4, since they are generally used as mapless path planning algorithms.

The A* algorithm is used to find the shortest path in roadmaps (Hart et al., 1968). In addition to the well cited Dijkstra algorithm (Dijkstra et al., 1959), A* adds a heuristic estimation for the cost of an arc. In an addition to the original weight of the arc to a node, a cost estimation from that node to the target node is included in the function. This particular example of a heuristic method can in fact guarantee an optimal path from initial to target node. However, as stated at the start of this chapter,

neither PRM nor visibility graph will be used, so no graph searching algorithm will be necessary.

The Nelder-Mead method, also called the downhill simplex method, is an optimisation technique that does not require the gradient of a function. For an n-dimensional problem, it creates a simplex with (n+1) vertices (Nelder and Mead, 1965). For example, for a 2D problem, it uses a simplex with 3 vertices, which is a triangle. This means that the method will evaluate the function at the three vertices of the triangle. The initial location of this triangle in the data is important for the final result. To find a more optimal solution, the method will flip the triangle over the side which connects the two best solutions, in the hope to find a better solution on the other side. Depending on how good this next corner point is, the triangle can reshape or flip in other directions. In path planning terms, a solution can represent a path, which has obtained a value through a cost function. If two paths are good, and one path is bad, the algorithm is interested in finding a path that is nearer to the two good solutions. The disadvantage of this method is that it also tends to get stuck in local optima, since the triangle will decrease in size once one of the corners gets close to a local optima.

3.1.5. Meta-heuristic algorithms

Meta-heuristic algorithms are optimisation algorithms which try to find a balance between looking for better solutions in the local area, and looking for other global solutions at random. In this way, they minimise the chance of getting trapped in local optima. The term meta-heuristic means "beyond heuristic" (X.-S. Yang, 2010a).

The optimisation problem that these meta-heuristic algorithms try to solve is the same as for the previous optimisation under constraints methods: find the path with the lowest cost function value from initial to target position, while keeping into account kinematic and dynamic constraints. The kinematic constraints mean that the drone needs to avoid obstacles, and the dynamic constraints make sure that the path is within the drone's velocity and acceleration limits. These algorithms have different metaphors to represent the path: It can be a chromosome, an ant in a colony, a particle in a swarm, a bee in a bee colony, and so forth. Each metaphor holds the information of the path that has been planned from the initial to the current position. A cost function gives a value to the chromosome, the ant, the particle and the bee. The difference with mathematical programming is that these methods try to find the shortest path iteratively, rather than minimising a certain cost function mathematically.

A well known example of a meta-heuristic algorithm is Tabu Search (TS). It memorises the previously selected solutions in a Tabu list, which means that it is forbidden to choose them again (Hillier and Liebermann, 2014). These solutions can be local optima. The method allows for a step in the direction of a worse path, so that it has more chance in finding global optima somewhere else.

A subcategory of the meta-heuristic algorithms are the nature-inspired meta-heuristic algorithms. These algorithms try to mimic biological, physical or sociological processes, which can be used to find the previously mentioned balance. For example, Simulated Annealing (SA) looks at the physical annealing process of metal or glass. First the material is melted at high temperatures, after which it is slowly cooled down until it reaches a desirable state. In path optimisation language, it means that the chance to select a new path randomly is high at the start, and slowly decreases. In this way, the chance to avoid local optima and end up at the global optima is increased. When the temperature is low, a new path will only be chosen when it proves to be better (Hillier and Liebermann, 2014).

Since animal behaviour is studied in large scale in the field of path planning, a separate sub-

category of the nature-inspired meta-heuristic algorithms has been created: bio-inspired meta-heuristic algorithms. These algorithms try to mimic the biological and sociological processes of animals and insects, and are referred to as Evolutionary Algorithms (EA). These EA may look at how animals or insects behave regarding the paths they take from their nests to a food source, and how these paths are optimised throughout time using communication.

For clarification, Figure 3.2 shows the subcategories in blue, and some example algorithms in grey. Each column of blue boxes represents a deeper level of subcategory.

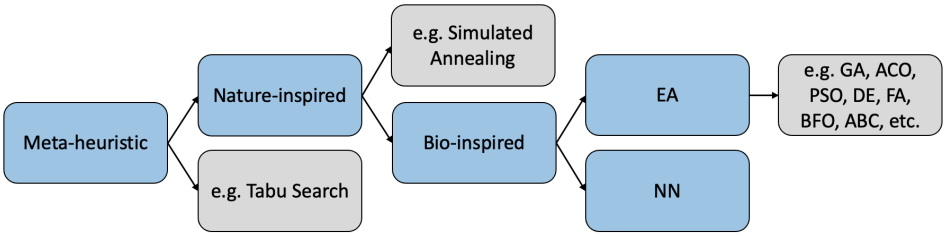


Figure 3.2: Subcategories of meta-heuristic algorithms

Some examples of EAs will be given below. They are also referred to as "Natural optimisation algorithms" (Haupt and Ellen Haupt, 2004). Also the definition of a memetic algorithm is included. A short conclusion on these meta-heuristic algorithms is provided at the end of this section.

Evolutionary Algorithms

In this section, the most popular EAs are highlighted. This selection is based on the frequency at which they are mentioned in the literature. The algorithms contain general optimisation methods and can be applied to path planning. A short explanation for each EA is given, where the link is made between the biological or sociological process and path planning.

- **Genetic Algorithm (GA).** Each path is represented by a chromosome, and this chromosome has a certain fitness value based on a cost function (Zhao et al., 2018). The fitter chromosomes are used for creating the next generation, where mutations and random chance changes their properties. (example by Samadi and Othman, 2013)
- **Ant Colony Optimisation (ACO).** Each path is represented by the path that an ant walks from its nest to a food source (L. Yang et al., 2016). On their walk, ants release a trail of a chemical factor called pheromone. This trail can be sensed by other ants and degrades over time. A shorter path can be travelled more often in the same time period than a longer path, hence the shorter path will contain a thicker trail of pheromone and will be followed by the other ants.
- **Particle Swarm Optimisation (PSO).** Each path is represented by a particle, which again represents a bird in a large swarm of birds. The birds fly from their nest to a food source, and their performance is being tracked (Kennedy and Eberhart, 1995). Each particle knows its own best location, as well as the swarm's best location (Zhao et al., 2018). The next iteration for a path takes into account these two locations, as well as its own velocity.
- **Differential Evolution (DE).** Each path is represented by a vector, and this method has similar properties to GA in the sense that it likes to use the terms population initialisation, mutation, crossover and selection (Storn and Price, 1997). However, it is all a bit more mathematically

complex, and even though it uses terms like mutation, it is a differential mutation with mathematics behind it which are far from something that looks like mutation. Larger vectors are used to explore the free space quickly, followed by smaller vectors to find the target precisely.

- **Firefly Algorithm (FA).** Each path is represented by a firefly, where the brightness of the firefly depends on a cost function (Gandomi et al., 2011). They send signals using flashing patterns. All fireflies are attracted to each other, and the attractiveness depends on the brightness and distance between the fireflies. Thus, when a firefly represents a short path, the cost function will be low, its brightness will be high and it will attract many other fireflies to join him in that direction.

- **Bacterial Foraging Optimisation (BFO).** Each path is represented by a bacterium, which can inform other bacteria if they are in a nutrient area or if they are in the lesser nutrient area through signals (Passino, 2002). Bacteria with enough food can split into 2 parts, and the ones without will die. This means that the local search around short paths is encouraged, and longer paths are discarded.

- **Artificial Bee Colony (ABC).** Each path is represented by a bee, which tries to find a food source (Karaboga, 2005). If a food source is found, they send more of their fellow bees there, which implies the local search near a short path. After some time, the food source is empty and the bees will look for another food source. The short paths belonging to all food sources are remembered.

- **Cuckoo Search (CS).** Each path is represented by an egg laid by a cuckoo. The cuckoos place their eggs in nests of other birds, which are often from other species (X.-S. Yang and Deb, 2009). If the egg represents a short path, the egg will create new generations, meaning that there will be a local search around this short path. The forming of new generations depends on both a cost function and random luck: if the host bird decides to throw the egg out of the nest, the path that was being considered is removed from memory.

- **Bat Algorithm (BA).** Each path is represented by a bat, which can communicate to other bats using echolocation (X.-S. Yang, 2010b). If a bat represents a short path, it will signal that to the other bats. The method is comparable to PSO, but the local search is improved by changing the rate and loudness of the echolocation.

- **Harmony Search (HS).** Each path is represented by an instrument, or a chord, and all paths currently stored in memory form the harmony. The method is based on the improvisation of music players (Geem et al., 2001). There are three different ways to obtain a new path for in the harmony. Firstly, it can be assigned randomly. Secondly, it can be assigned out of memory, where good values are stored. Thirdly, the value looks at values close to the previous value.

- **Invasive Weed Optimisation (IWO).** Each path is represented by individual weeds, spreading through a field (Mehrabian and Lucas, 2006). The number of seeds that an individual can produce depends on a fitness function: A high fitness results in a high seed production and a low fitness results in a low seed production. The seeds will fall near the parent with a standard deviation, allowing for local search. An individual with a high fitness can be seen as a path with low value for the cost function.

A Memetic Algorithm (MA) makes use of multiple EAs, or combines EAs with other algorithms, with specific attention to the population-based metods (Moscato et al., 2004). MAs were introduced to address the drawback of EAs of getting stuck in local optima. An example of this is the Shuffled Frog Leaping Algorithm (SFLA). SFLA combines PSO with Shuffled Complex Evolution (SCE) (Eusuff and Lansey, 2003). It uses PSO to search for local optima, and SCE to combine the information

from all the searches to get to a global optima. SCE is an optimisation method created by Duan et al. (1993) to reach global optima, by combining multiple methods that have shown to be able to converge to global optima.

More meta-heuristic bio-inspired path planning algorithms based on metaphors can be found in the literature, but they all stem from the same principle: Look at biological behaviour, try to mimic that using an algorithm and apply it to path planning. Unfortunately, meta-heuristic bio-inspired algorithms are prone to get stuck in local optima (Mac et al., 2016). Despite their initial popularity, the most common bio-inspired algorithms were barely mentioned in scientific papers published after 2014 (Zhao et al., 2018). Furthermore, these algorithms mainly focus on optimising the current best solution, which takes many iterations and is computationally expensive (Triharminto et al., 2013). For every iteration, a new path needs to be generated and sent through a cost function. The drone is not able to see behind obstacles, therefore it is expected that these algorithms need to be rerun often, since new information might consists of additional obstacles in the way.

3.2. Potential Fields

The next algorithms described make use of a potential field within the map to plan a path from initial to target position. The target sends out an attractive force to the drone, while the obstacles give a repulsive force. The resultant of these forces defines the direction of travel. The first method discussed is the Virtual Force Field, followed by the Vector Field Histogram.

3.2.1. Virtual Force Field

The idea behind the potential fields method was first mentioned by Khatib and Mampey (1978), and later extended and implemented on a robot with actual sensing capabilities by Borenstein and Koren (1989), and they named it the Virtual Force Field (VFF) method. In their paper, the authors use a grid map to represent the environment. This process is called tessellation (Radmanesh et al., 2018). The center of each cell in the grid is used for the calculations. Here, obstacles are represented using certainty grids, where a higher cell value means a higher certainty of obstacle presence. The higher the certainty, the higher the repulsive forces towards the robot. This method could also be applied to empty workspaces with point clouds or other obstacle representations (e.g. by Kaldestad et al., 2014). The VFF method works as long as the obstacles are sending out some form of repulsive force to the drone.

The authors found four significant drawbacks of the VFF method, which are independent on the implementation (Koren, Borenstein, et al., 1991). Firstly, the most cited drawback, the robot is prone to get stuck in local minima. Secondly, VFF has difficulties travelling through obstacles close together, for example the posts of a door. Thirdly, the robot oscillates when an obstacle suddenly pushes the robot away from the target. Fourthly, the robot oscillates when travelling through narrow passages. The authors mention that many other papers on this topic focus mainly on simulation results, and do not take these unsolvable implementation problems into account. Some papers use real robots, but move them at slow travelling speeds which hide these drawbacks. Hence, the authors stepped away from the PF method and turned to the Vector Field Histogram method (VFH).

However, since the creation of the VFH method in 1990, many other papers have tried to overcome the previously mentioned drawbacks of VFF. For example, Sfeir et al. (2011) solves for drawbacks two, three and four, by introducing a new repelling and rotational force. These rotational forces steer the robot around an obstacle, instead of away from it. Still, they can not mathematically proof that the robot will not get stuck in local minima. Kim (2009) thought of a method that lets the

robot know when it is in a local minima, so that it can apply a different strategy to find its way out. This happens when four conditions are satisfied: the resultant force equals zero, two or more obstacles cause this, the goal is not reached, and the location of the robot not changing. Their method works in simulation, but has not been tested in real life. A similar approach was implemented by Antich and Ortiz (2005), who performs an obstacle boundary-following technique to escape local minima. The virtual obstacle approach also helps to get the robot out, by generating a virtual obstacle on the location of the local minima, so that the robot is repelled from here (Lee and Park, 2003). However, all three methods do not avoid getting trapped in local minima, they only tell the robot how to get out once it is in. Normally, one of the advantages of a mapping is that the robot should be able to avoid local minima. With the mapping that potential field uses, memory is being wasted since it does not avoid local minima any better than a mapless (reactive) method would. The different strategies used here to get out of local minima could still be used in a reactive approach. This will be discussed in section 5.2.

3.2.2. Vector Field Histogram

Borenstein and Koren (1990) proposed the Vector Field Histogram (VFH) method, which can be used by a robot to avoid obstacles at high speeds and move through narrow corridors. In the original paper, the environment is divided into equal cells which form a grid, and every cell is assigned a certainty value. However, this method could also be applied to empty workspaces with point clouds or other obstacle representations. Furthermore, the grid is in 2D, hence it does not suffer as much from the computational expenses as the 3D grid map does. Figure 3.3 shows what the VFH method looks like when dividing the active grid into 16 sectors of equal angle. Note that this figure is simply used as an example, since the actual paper uses 24 sectors. To obtain the polar graph on the right, the method needs to find in which direction the obstacles are present, for which vectors are used. The magnitude of a vector is a function of the distance of the obstacle, as well as the certainty value. The direction of the vector is the angle between the obstacle and the positive x axis, so the angle is defined similar to the unit circle. In this way, all sectors obtain a value for the likelihood of an obstacle being present. Sectors in the polar graph which have a value under a certain threshold are considered as free sectors. The free sector closest to the target direction is chosen.

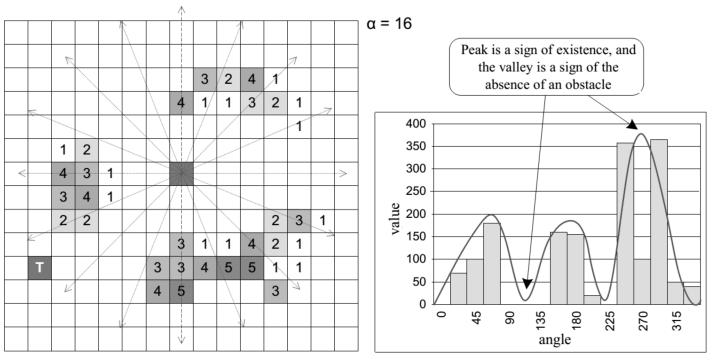


Figure 3.3: VFH method (Jahanshahi and Sari, 2018)

The authors expanded on their idea by incorporating the robot size, among with its kinematic constraints in VFH+ (Ulrich and Borenstein, 1998). VFH+ introduces the binarised polar histogram. In this histogram, sectors with values under a certain threshold A are given a 0, and sectors above a certain threshold B are given a 1. When a sector value is between A and B, the previous iteration is used. Furthermore, in the real world the robot is not able to move in every direction as indicated in Figure 3.3. For example, to move to the left, first a turn with a certain radius has to be initiated

until the desired heading angle is obtained. During this turn the robot might hit obstacles that were not foreseen. Hence, sectors which seemed promising, but will result in hitting an obstacle during the turn are also blocked in the so-called masked polar histogram. Lastly, a new cost function is defined for choosing a sector. It is a function of the distance to the target, as well as the distance to that sector due to the orientation of the robot and the distance to the target of the previous sector.

Two years later the authors expanded further by including future prediction in their method, now called VFH* (Ulrich and Borenstein, 2000). It looks at all free sectors individually and predicts future positions based on choosing a free sector. After creating a tree of possible directions, the A* algorithm is applied to find the shortest path. The methods assumes 360 degrees SONAR readings, but could still work with a smaller FoV, since the most interesting directions lay in the direction of the target. A disadvantage is that the method is designed for a two dimensional problem, where a polar histogram is created for the top view. Creating a polar histogram for every small angle of pitch will yield many graphs, many possible directions and many cost function evaluations. In 2D, the method is already computationally expensive, making real-time applications difficult (Samir, 2014).

A solution for the 3D problem could be to design a reactive VFH approach, where the grid is not applied to the top view, but to the disparity image of the drone. Then, every cell needs to be checked for certainty values and the best cell according to the cost function is chosen. However, with noisy disparity images, these cells need to be small to prevent a large number of false positives. Furthermore, these certainty values are calculated using the Carnegie-Mellon University (CMU) method (Borenstein and Koren, 1990), which require the robot to remain stationary while taking multiple sensor readings. Fortunately, a faster approach has been performed by Lyrakis (2019), since a drone travels fast and should not stop for sensor readings. Instead of assigning the certainty of obstacle presence, it assigns the uncertainty of the disparity values to each cell, which has been discussed in section 1.4.

3.3. Discussion

This chapter on map-based path planning has presented two different categories: optimisation under constraints and potential field methods. Optimisation under constraints was divided over five subsections, and the potential field methods were captured by the VFF and VFH.

Regarding optimisation under constraints, and more particularly, mathematical programming: methods using MILP or BIP either require much computation, or precise maps where the complete environment has been partitioned into tetrahedrons. Curve based algorithms also have a high time complexity, and are mainly used when global maps are available. Gradient-based search algorithms are not very common in path planning, since they tend to find local optimal solutions. Heuristic algorithms try to find the global optima quicker by introducing a heuristic, but still these algorithms are prone to yield local optima. Meta-heuristic algorithms try to minimise the chance of getting a local optima, which comes at a computational cost. Each iteration needs to generate a path and evaluate it with a cost function. Since information about the environment is constantly updated, this makes it an expensive local approach.

The VFF method is prone to get trapped in local minima, or at least get caught in it for a short time period. Also extensions on this method exhibit similar problems. A reactive approach would be a cheaper solution, while having the same challenges. The VFH seems promising in 2D, but yields implementation difficulties in 3D. A solution could be to perform VFH as a reactive method, and use the uncertainty of the disparity values, much like the approach presented by Lyrakis (2019), presented in section 1.4.

4

Mapless path planning

While building maps can be helpful for avoiding local minima and obtaining global optima, it can be a computationally expensive process. Furthermore, a map would create most extra added value if the drone flies through the same area multiple times and remembers previously seen obstacles. If the drone is travelling through the local areas quickly without coming back, the added value is limited. Looking at Figure 4.1, the grid map does not contain any extra information to what the stereo camera can see. Taking either one of the two possible routes can be done with a simpler algorithm, then to first create a grid map and use a path searching algorithm. Maps are in favour when the drone needs to find its way out of a local minima, or when it is stuck in a large maze with only one way out. Since maze-type environments are unlikely in outdoor situations, a reactive mapless approach might be enough for path planning. Local minima would need to be escaped or avoided using different modes during flight, as will be explained in section 5.2.

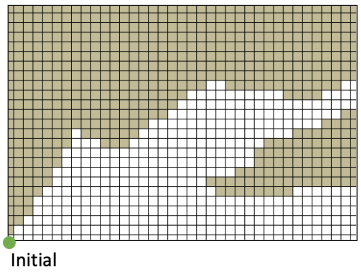


Figure 4.1: Local grid map

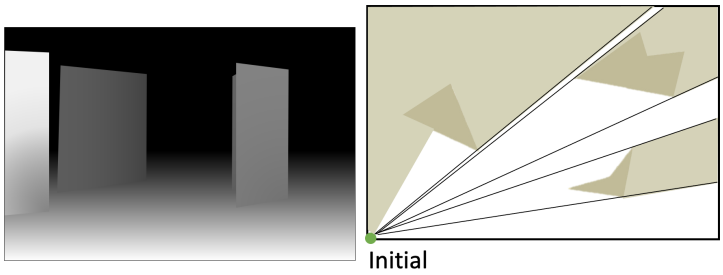
For a reactive approach, the local percept is used as environmental representation. This is also referred to as an image space map (van Dijk, 2020). An example is shown in Figure 4.2(a), where the resulting disparity image produced by the stereo camera is shown. The information stored in this image is used to build the previously mentioned maps, but could also be used directly for path planning. The task is as simple as selecting a pixel, but the question is: which one? Realistically, due to stereo matching challenges, the disparity image will contain more noise than shown here.

The disparity image could be used to track the edges of the obstacles, as shown in Figure 4.2(b). This form of ray tracing can be extended to 3D, where the boundaries of the obstacles are traced, in a similar fashion to CEG. Using this method, only a few possible heading angle ranges are given to the drone. However, the process of finding these boundaries might be a difficult process due to noise.

This chapter will present multiple mapless path planning algorithms, along with their advantages and disadvantages and will discuss their findings at the end. The methods discussed are fuzzy logic, machine learning and bug algorithms, which can all be related to bio-inspired algorithms, since they try to mimic human or animal behaviour.

4.1. Fuzzy Logic

Fuzzy Logic (FL), introduced by Zadeh (1965), tries to mimic human "if-then" reasoning when faced with incomplete knowledge. Since humans are able to navigate without any direct calculations, it is interesting to study this field. It is used for situations when the information is vague, or unclear,



((a)) Disparity image

((b)) Ray tacing (adapted from Patle et al. (2019))

Figure 4.2: Path planning in image space

hence the term fuzzy. The information is first sent through a fuzzification layer. After, the "if-then" rules are applied and a defuzzifier is used to give an output value (Patle et al., 2019). FL can be used for making decisions, but also for control and pattern recognition. It is a popular method since it gives insights into the reason behind the choices made.

FL can be categorised under bio-inspired algorithms, since it tries to mimic humans. The most common implementation of FL algorithms is mapless, where it uses range sensor data to obtain input. For example, Pandey et al. (2014) created a fuzzy logic controller for their two-wheeled robot to avoid obstacles. Their range sensor readings are categorised under either close, medium or far away, based on predefined thresholds. These categories cause the input data to be fuzzy, since none of them can tell exactly where the obstacles are. A large look-up table was created with actions that need to be taken under certain conditions of the range sensor. Another paper used FL to steer a quadrotor around obstacles, using 12 sensors for omni-directional sensing (Wahyunggoro, Cahyadi, et al., 2016. These sensor readings are compared to each other, and the result of the fuzzification layer is that the results only know if they are larger or smaller than other sensor readings. Using comparison operators, the method gives a heading angle for the drone as output.

The disadvantage of these methods is that every possible action needs to be included in some sort of a look-up table of if-then rules. For the two-wheeled robot the inputs are obstacle distance on the front, left and right side, and the outputs are the motor speed for the left and right wheel separately (Pandey et al., 2014). This already gives a table of 27 fuzzy rules. For a drone, it is not as simple as turning one propeller faster or slower. It will need outputs for yaw, pitch and roll, as well as accelerate or decelerate. To avoid this becoming a large list, the quad-copter paper discretised the output in heading angles of 45 degrees. Being limited by these 45 degrees will result in a sub-optimal path. Also, their FL method needs an additional potential field method to steer the drone to the target. Furthermore, their method is applied in 2D, since the altitude of the drone is assumed constant. In 3D, more sensor data needs to be used as input, since now also the height and altitude of obstacles needs to be taken into account. Let's assume that the sensor now reads in 9 directions, in a grid from top left to bottom right. The if-then list expands with power 3, meaning that for 3 inputs it contains 27 rows, and for 9 inputs it contains 729 rows. Moreover, FL is not always able to apply the best rules to a particular situation. It is a difficult process to select the necessary rules that include every possible situation (Zhao et al., 2018).

According to Zhao et al. (2018), methods based on GA and FL are the most used algorithms for online path planning. However, the main contribution to that is that the method used to be extremely popular in the past. Since 2012, FL has been hardly ever mentioned in the literature on path planning algorithms. Since it is a difficult process to select the appropriate rules, FL is often

integrated with a neural network. In this way, the human reasoning is combined with the learning ability that a neural network has. This results in a neural network that gets the sensor data readings and target direction as input, and gives, for example, steering angles as output (Mohanty and Parhi, 2013). The look-up table of FL has therefore been replaced by the nodes and weights of the network, that try to map the input to the appropriate output. The combination of the two give better results than using the individual algorithms, but it comes at a computational cost (Mac et al., 2016). An example is the Adaptive Neuro-Fuzzy Inference System (ANFIS), which uses a 6 layer neural network and gives a steering angle as output (Singh et al., 2009). But even after extensive training and careful hyperparameter tuning, the path that this methods chooses is still sub-optimal. The planned path sometimes even heads straight towards an obstacle, before taking the same path back. Since it is preferred to have a predictable method that takes logical steps, and can explain why it takes certain steps, neuro-fuzzy methods will not be used.

4.2. Machine Learning

Machine learning methods can be used for robot path planning in three different ways. Firstly, it can be used for building a map. For example, Luo and Yang (2008) build a 2D map for a cleaning robot, which is interested in a complete coverage of the available area. The map is divided into small squares and triangles to make sure that all the area is covered. The authors call their method real-time, yet are not confident enough to mention anything about their run times or robot speed. Furthermore, since building maps with machine learning mostly focuses on full coverage methods, it will not be used in this study. Secondly, machine learning methods can be used for reactive avoidance. This means that the target location is not given as an input, and the only goal is to not hit obstacles. Since the task in this study is to find a path to the target, the third machine learning method is preferred, which finds a path to the target while avoiding obstacles. The rest of this section will discuss machine learning-based path planning methods.

Machine learning algorithms can be subdivided into three types of learning: Supervised, unsupervised and reinforcement learning (Brownlee, 2019). Supervised learning algorithms learn from many samples, where the input is mapped onto a desired output. Unsupervised learning does not have these examples, and is normally used to cluster data. It can look for similarities in the data and summarise it. Reinforcement learning is not given a large set of data either. Instead, it is given rewards or punishments for performing actions in certain states. It remembers the action it took for a state, and if it was punished, it will try something different next time.

Artificial neural networks are a common feature of machine learning algorithms. A neural network can be seen as a bio-inspired algorithm, since it mimics the neural activity of the brain. The structure of the network can be defined by the user. Many nodes are interconnected, each with its own weight, passing on information to an activation function. The output state can be a heading angle, or a velocity for example. A neural network is able to generalise well and can learn complex and non-linear relations within the information presented (Patle et al., 2019).

4.2.1. Supervised Learning

Supervised learning algorithms are common for path planning. For example, S. X. Yang and Meng (2000) use a neural network approach to guide their robot through a dynamic environment with a moving target. In a static environment they obtain the global optimum solution. Disadvantage is that the method assumes a constant speed for both the robot, which is not representative for real life situations (Mac et al., 2016). A few years later, Janglová (2004) came up with a method that uses

two neural networks: one to find the obstacle free space, and one to find the path within this space. The first network uses the information given by the distance sensors and the second network uses the output of the first network, as well as the direction of the target as input. Main disadvantage is that a lot of human intervention was needed during the training process to help the robot learn to take the correct path and many samples were needed to train the network. To limit the amount of training samples, Singh and Parhi (2011) thought of training their neural network on 200 typical scenarios only. There are four inputs to the network: obstacle distance on the left, front and right, as well as the heading angle of the target position. With less computation time, the algorithm is able to find a path to the target in a unknown environment (Mac et al., 2016). It should be said that the range sensors only have a range of 25 cm, and the robot moves at a maximum speed of 0.5 m/s, so the algorithm does not get large amounts of data and uses roughly half a second to safely plan a path.

In a more recent example, Tai et al. (2016) created a deep network that is able to translate a depth image into a steering angle. However, deep neural networks are not ideal for systems that require low computation. The paper used their method in a 2D environment, and on average the algorithm took 0.25 sec to generate output. In 3D this computational time is expected to increase rapidly, since the degrees of freedom go from 3 to 6. Since the depth images will contain noise, the calculations must be performed quickly to obtain certainty about a particular direction. Therefore the deep neural networks will not be suitable.

The list of supervised path planning algorithms can be extended, but the main drawbacks remain. In general, supervised learning algorithms need many samples plus a large variety within this data (Mac et al., 2016). Another possibility is a self-supervised learning algorithm, which creates the data while it is learning from it. However, that would only solve for the time consuming data collection step. The output of these algorithms will remain unpredictable, and global optimal solutions cannot be guaranteed (Mac et al., 2016).

4.2.2. Unsupervised Learning

Unsupervised learning algorithms are not common in path planning applications. Since they are not told what is right and wrong, it is difficult, if not impossible, to create a path from initial to target position. There are instances where authors tried to apply it directly for reactive path planning. For example, Hirose et al. (2017) use unsupervised learning to check the incoming images for last-minute obstacle collision, on a robot with one camera. This is redundant, since a drone with a stereo camera can already obtain this type of information. Normally, unsupervised learning is used to cluster data, or to find points of interest within the data. These type of algorithms need to be combined with another algorithm to plan a path, leading to higher computational expenses.

As a sidenote, a paper by Muniz et al. (1995) claims that it performs unsupervised learning, while in fact it performs reinforcement learning.

4.2.3. Reinforcement Learning

One type not yet discussed is reinforcement learning. This type tries to learn the best actions to be taken under certain situations, saved as states. Q-learning is a subset of reinforcement learning, which is of particular interest because it does not need to remember a model of the environment, but simply remembers what actions need to be taken for the states. However, for an unknown, three-dimensional environment the number of states quickly approaches infinity. To solve this, a smart way to reduce the states had to be introduced. For example, Jaradat et al. (2011) thought of mimicking human reasoning when faced with an unknown environment: the human does not care about its own location as much as the relative distance to the obstacles. It managed to decrease the

states to a couple of heading directions and train the model on those. After careful tuning, it took 75 different training scenarios to obtain a 98% success rate on reaching the target position. Still it is unclear what happened in the 2% where it did not meet the target. This percentage increased with an increasing number of obstacles. When there would be 13 obstacles, this would have increased to 24.4% of not achieving the target position. Since these methods require a lot of training, and still show unpredictable behaviour, they are not considered suitable for this study.

4.3. Bug Algorithms

Bug algorithms look at the behaviour of bugs, and can therefore be considered a bio-inspired algorithm. They perform path planning in unknown environments, where the direction to the target position is known. Furthermore, they require little computation due to their simple reasoning in finding a way to the target. The bug algorithm can be divided into two categories: angle-bugs and M-line bugs, which can be seen in Figure 4.3 (K. N. McGuire et al., 2019). Their names originate from the choice of path when leaving an obstacle. Some of these use range sensors to steer the robot in a different direction, before it has been in contact with the obstacle. These can be classified under the subcategory of range-bugs. The bug algorithm family was the first to guarantee global convergence for path planning in an unknown environment, in 1986 (Ng, 2010).

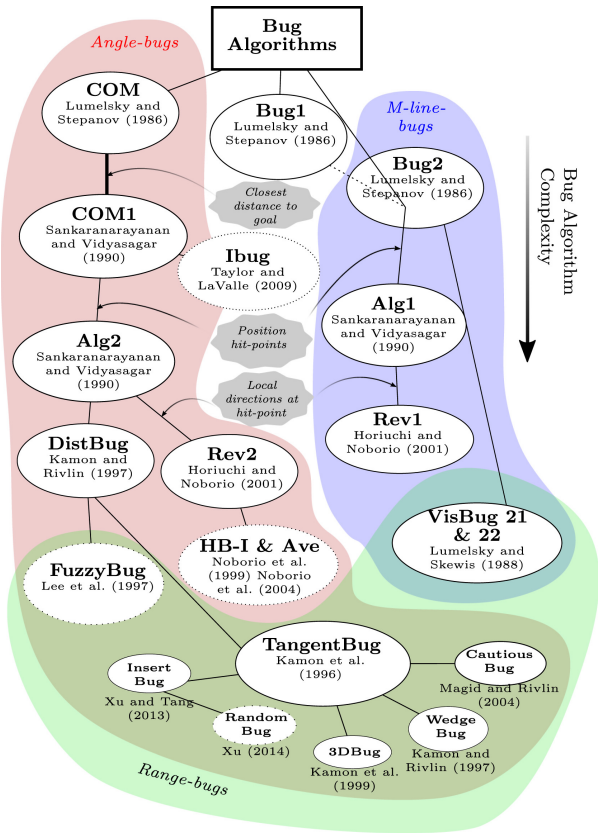


Figure 4.3: Classifying Bug algorithms (K. N. McGuire et al., 2019)

The following six assumptions hold for bug algorithms: First, the geometry and position of obstacles are unknown. Second, every obstacle has a finite area it covers. Third, the robot is considered a point object. Fourth, the location of the target and the objects do not change. Fifth, the robot has perfect localisation property and sixth, it has perfect sensors. The first assumption is in line with this study. For the second, even though it might cover a finite area, it could still block the complete

FoV. The third is covered by the c-space expansion, explained in section 1.5. The fourth assumption holds since this study considers only static obstacles. The fifth holds since the drone has GPS data available. The sixth does not hold, since the stereo camera produces noisy disparity images. Approaches to reduce the noise are discussed in section 5.1.

Before going into the different types of bug algorithms, the very first bug algorithm will be explained. In Bug1, the robot is able to travel towards the target, and along obstacle boundaries (Lumelsky and Stepanov, 1986). Once it senses an obstacle with its contact sensor, it defines a hit point. From there, it will perform a full revolution around the obstacle, while trying to find the point with the smallest distance towards the target. After returning to the hit point, the robot will return to the point that is closest to the target, via the shortest route. The method is non-heuristic, since a certain worst case path length is guaranteed (Sankaranarayanar and Vidyasagar, 1990). The advantage of bug1 is that it guarantees to reach its target, The disadvantage is that the chosen path is rather long, since the robot performs a revolution around every encountered obstacle. Furthermore, Bug1 does not work for dimensions greater than 2 (Sankaranarayanar and Vidyasagar, 1990).

4.3.1. M-line bug algorithms

An M-line, or main line, is a virtual line between the initial and target position, and was introduced by Bug2 (Lumelsky and Stepanov, 1986). Once the robot meets an obstacle, it will move along the obstacle boundary until a point on the virtual line is met. From there, the robot will continue its way towards the target. Where Bug1 is more conservative, Bug2 is more aggressive and can be more efficient, however the distance for Bug2 can also be larger when start and target points are placed in a nontrivial way (Lumelsky and Stepanov, 1986). An alternative method created by Noborio et al. (1997) changes Bug2 by providing a random selection of turn, instead of a constant left or right turn when hitting an obstacle. The difference is small, but the authors claim that on average the algorithm performs better.

Alg1 modifies the constant turn version of Bug2 (Sankaranarayanan and Vidyasagar, 1990). Once it encounters a point that it has already visited, and therefore also the boundary behind it, it changes direction and tries to find a new leave point closer to the target. Although it has a higher bound, it is faster on average. The worst case path of Alg1 cannot be improved any further, when using the M-line method. Rev1 is based on Alg1 and allows for an alternative switching between left and right turns (Horiuchi and Noborio, 2001). Some simulations results show that it can produce shorter paths, but there is no mathematical proof of this.

The authors of VisBug take most of their inspiration from their previously created algorithm Bug2 (Lumelsky and Skewis, 1988). In addition, it has local sensing information from a stereo camera or a range sensor, hence the term vision is abbreviated in the name. It can guide the robot through shorter paths by cutting corners, meaning that if the robot sees that it will hit an obstacle soon, it can already start turning to the edge of the obstacle. Visbug-21 is claimed to be never outperformed by Bug2. The sensors scan a limited area, which is computationally more efficient than scanning 360 degrees. The VisBug-22 algorithm has a more opportunistic approach. It can deviate from the path that Bug2 would tell it to take by making use of promising opportunities. These opportunities can be paths that first lead back to the M-line, before heading towards the target position. The paths are normally shorter than Bug2, but better performance can not be guaranteed for VisBug-22.

The main disadvantage of these M-line bug algorithms is that they have to return to the M-line every time they pass an obstacle. Even with some visual feedback, the method tends to take unnecessary turns. Another approach would be to directly head towards the target position once an

obstacle is cleared. This idea is adopted by the angle bug algorithms.

4.3.2. Angle bug algorithms

The angle bug algorithms leave the obstacle boundary once they see that the path towards the target is clear. The first algorithm to be discussed, Com, got its name from being a "common sense algorithm", since it makes sense to go straight to the target when possible (Sankaranarayanar and Vidyasagar, 1990). It performs less boundary following, relieving some of the work of the contact sensors. The disadvantages are that it can sometimes create extremely long paths for simple environments, caused by drifting to infinity or getting stuck in a loop. The first is addressed by Comr, which only leaves an obstacle boundary when this leave point is within a disc. This disc is drawn around the target, and its radius is equal to the previously best visited location's distance to the target. The second is addressed by Com1, which added another rule: the distance from the leave point to the target must be smaller than the distance of x to the target, for all x visited by the robot prior to the leave point. An infinite loop can now only occur if the target is trapped by a boundary.

One of the disadvantages of Com1 is that, in some environments, it tends to visit previously visited path many times more. Alg2 prevents this by reversing the direction of travel once it encounters a previously visited hit or leave point. Bug algorithm Gen is a generalised version of Alg2, which does not necessarily use the euclidian metric, but any metric for distance (Sankaranarayanar and Vidyasagar, 1990). Furthermore, Rev2 is comparable to Rev1, but now it is applied to Alg2, meaning: the direction of turns is alternated (Horiuchi and Noborio, 2001). Again, no mathematical proof on better performance is given but simulations show that it has the capability of outperforming Alg2.

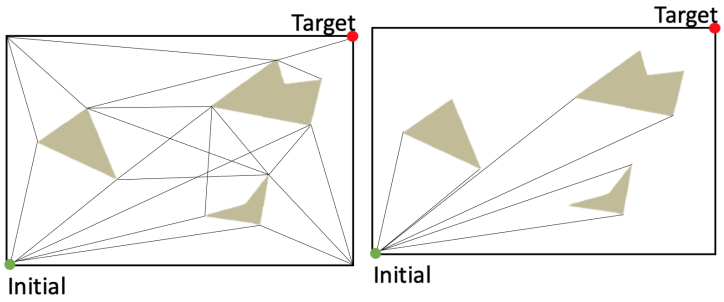
The last angle bug algorithm without a range sensor is HD-I, together with a small improvement which resulted in Ave, and DistBug. Noborio et al. (1999) created HD-I, which is a mix between the Best-First (BF) and Class 1 algorithm. Class 1 is an algorithm that the author has claimed to be his own work, yet it is an exact replica of the Com1 algorithm, which was released two years before Class 1 (Noborio, 1992). Furthermore, BF is an algorithm that searches the best options within a graph. A* is an example of such an algorithm. Ave is an extension of HD-I, which uses the previously visited nodes to decide which boundary following direction is best. A year later, the same author created NH-I and NH-II (Noborio et al., 2000). NH-I is similar to HD-I, but heuristic methods are introduced in both BF and Class 1. NH-II is a combination of Depth-First and Class I. However, section 2.2 concluded that roadmaps like PRM will not be used since other methods are preferred, so none of these algorithms will be used.

4.3.3. Range bug algorithms

Figure 4.3, created by K. N. McGuire et al. (2019), shows a small mistake: DistBug does in fact use range sensor data to plan its path, and should be inside of the green area (Kamon and Rivlin, 1997). While following the boundary of an obstacle, it checks whether the direction of the target position is obstacle free. In addition, they use local information to choose for either a left or right turn and they bound the search area to further improve performance. The method is robust to noise in the distance measurements and has been tested on real robots. It was mentioned that the implementation was easier than VisBug, since DistBug uses the range data directly instead of first modeling the local environment. Its performance is comparable to Alg1 and Com1, but for relatively simple environments, DistBug has greater performance. The remaining disadvantage is that it still adheres to boundary following.

TangentBug

TangentBug is another bug algorithm, created by the same authors as DistBug. In fact, it was published a year earlier. It uses range sensors to obtain information about the distance and heading of the obstacles ahead. If the path towards the target has obstacles, the robot will choose the path where the distance from the current to the target position is the smallest (Kamon et al., 1998). Global convergence is guaranteed, since it has a theoretical worst case path length. Once a loop is encountered, TangentBug will switch to boundary following mode to prevent making the same mistake twice (Ng, 2010). The algorithm uses a local tangent graph, which is a simpler form of the visibility graph as shown in Figure 4.4. It only contains the lines that mark the edges of the obstacles. The authors tested TangentBug against Visbug, and over 900 runs TangentBug was a factor 3 better on average. These runs included convex-shaped obstacles, mazes and indoor office-like environments.



((a)) Visibility graph ((b)) Local Tangent Graph (LTG)

Figure 4.4: Visibility graph versus LTG (adapted from Patle et al. (2019))

A disadvantage of this algorithm is the computational power required. The robot uses 16 sonar sensors and 16 infrared range sensor, each with 22.5 degrees of resolution. Creating a 360 degrees LTG is computationally heavy in real-time (Q.-l. Xu, 2014). Another disadvantage is that the method is applied in 2D, whereas path planning for drones will require 3D.

EgressBug

EgressBug claims to have better performance in path planning than both Bug2 and TangentBug (Guruprasad, 2011). The paper states that TangentBug can enter an infinite loop in a nontrivial environment. Once such a loop is encountered, TangentBug concludes that the target is unreachable. EgressBug improves on this by letting the robot know once it has performed a loop. It then will change the direction of the wall following mode to egress (or: leave) the loop, in a similar fashion to Alg1 and Alg2. Due to its drawbacks, TangentBug will not be used. However, in case the chosen algorithm performs some kind of wall-following manoeuvre, EgressBug shows that it is wise to change the direction of travel once the robot has encountered a loop. On the other hand, loops are highly unlikely in 3D since the drone also has the possibility to change altitude.

InsertBug

InsertBug also builds on the idea of TangentBug, but places the waypoints at a safe distance from an obstacle's boundary to prevent collisions (Q.-L. Xu and Tang, 2013). However, when implementing a disparity space expansion, as explained in section 1.5, this safety distance from obstacle boundaries is already taken care of. Furthermore, InsertBug uses a vector notation to describe the planned paths, which makes the path easier to follow and is computationally lighter, according to the authors. However, their method is mostly computationally lighter since it scans 180 degrees, compared to the 360 degrees of TangentBug. Furthermore, every new waypoint has a minimum distance from the previous, whereas TangentBug is constantly looking for better waypoints every time it obtains more range information. Their implementation of TangentBug is also questionable, since they state

that the algorithm has a range sensor of 300 pixels, yet it still behaves almost like a contact sensor version, where it only changes direction after it has been in contact with the obstacle. The benefit of the vector notation is that the difference between two waypoints can be described in terms of a rotation angle and movement distance, creating simple commands that can be send to the drone to follow the path. However, the simulation software used for this study, discussed in section 5.4, uses the carrot following algorithm to get to its waypoints. Therefore, the exact navigation commands are already dealt with.

RandomBug

Since the influence of InsertBug in the world of bug algorithms was minimal, being cited by only 4 different researchers over the past 9 years, the same author tried again with a different algorithm 2 years later: RandomBug. As the name might have suggested, it creates multiple random waypoints once it encounters an obstacle, and chooses the best one (Q.-l. Xu, 2014). The best waypoint can for example be defined as being the closest to the target position. Since a safety margin is taken into account upon selecting a new waypoint, it is similar to InsertBug. A disadvantage is that the constant generation of random waypoints causes the path to consist of many, arguably unnecessary, sharp turns. The advantages that the author mentions are similar to InsertBug: the vector form is good, it requires a smaller storage space and keeps a safe distance to obstacles. However the path lengths are worse than TangentBug, and also worse than InsertBug. Being cited by only 2 different researches this time, it can be concluded that also RandomBug will not be the chosen algorithm.

Wedgebug

Laubach and Burdick (1999) address the drawback of computational expenses of the TangentBug algorithm. This was a result from the limitation of the planetary rover they wanted to use it for. The rover has limited range sensing, and can only detect obstacles within a small wedge, formed by the FoV and the range. They use a local version of the local tangent graph. The local tangent graph is called local since it only measures obstacles nearby, and the local version of it means that it only senses in a small FoV. If an obstacle is blocking the view of the complete wedge, the robot scans another wedge in an adjacent direction by turning the robot. Wedgebug tries to limit the amount of necessary scans, to avoid unnecessary robot movement. The algorithm is a promising candidate for the path planning algorithm to be used on a drone. The challenge that remains is extending this 2D algorithm into 3D.

3DBug

3DBug extends TangentBug into 3D, by defining a new, efficient data structure called the Convex Edges Graph (CEG) (Kamon et al., 1999). The CEG also has been discussed in section 2.2. It stores the convex hull of obstacle edges, as well as the current and target position. It plans the path around these obstacles by setting waypoints, called focus points F. The algorithm can obtain the global optimum for relatively simple environments. They categorised the algorithm under the bug algorithm family, since it uses local sensor data directly, without losing the guarantee of global convergence. In 3DBug, the next set of possible waypoints are connected to only one obstacle. Compared to the visibility graph, this is more memory efficient, since the visibility graph tries to connect all vertices with each other. The authors simulated both options and found that 3DBug needed 3.3 nodes on average from initial to target position, while the visibility graph needed 32.4 nodes on average. Furthermore, the compact data structure is able to represent a certain 3D environment with 7 obstacles in 7 nodes and 9 edges, while the numbers for the visibility graph are 620 and 118912 respectively. Another advantage for 3DBug is the simplicity of deciding unreachability of a target. It explores the surface of one obstacle, while an A* algorithm in a visibility graph would need to search through all possible nodes before concluding unreachability.

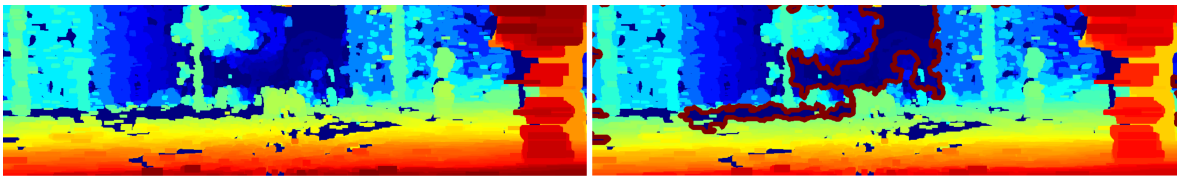
The algorithm makes a number of assumptions to get to its results. First, it assumes that all obstacles in the environment can be modelled as polyhedral obstacles. Second, a perfect range sensor with no maximum range is used. Third, the range sensor can sense in all directions. Fourth, the range data, which includes both vertices and edges, is perfectly translated into 3D coordinates. The second, third and fourth do not hold, since the range data contains noise, has a small FoV as well as a limited range. Furthermore, the noise makes it difficult to detect the precise contours of an obstacle, breaking assumption one. Fortunately, an algorithm similar to 3DBug has been developed that is able to cope with these limitations. Instead of finding obstacle boundaries, it finds escape point boundaries. These are the boundaries between areas with obstacles, and areas without. The algorithm will be discussed in the next section.

the first to guarantee global convergence for path planning i

4.4. Obstacle avoidance using uncertainty maps

The idea behind uncertainty maps has been sketched in section 1.4. The uncertainty values are used to filter the disparity map, to check the (un)certainty of collisions and to check the (un)certainty of the obstacle-free escape points. This section explains the process of finding these escape points, as well as selecting the best one. This results in local path planning in image space.

Escape points are essentially obstacle-free image pixels. To clarify, escape points are only calculated once the drone realises it is close to colliding with an obstacle. The Moore-Neighbor Tracing algorithm is used to find these points, because of its efficiency compared to Square Tracing algorithm, Radial Sweep, Theo Pavlidis algorithm and Dr Kovalevsky algorithm (Lyrakis, 2019). Its average runtime is 0.1 msec on a 1241x376 disparity image. An example is shown in Figure 4.5, where Figure 4.5(a) shows the disparity image after the c-space expansion, and Figure 4.5(b) shows the escape points with the dark red boundary (Lyrakis, 2019).



((a)) C-space disparity image ((b)) Escape points (dark red)

Figure 4.5: Escape point selection from disparity image (Lyrakis, 2019)

The selection procedure of the best pixel was done according to the cost function shown in Equation 4.1 (Lyrakis, 2019). Here, w_i represents weights and d the distances. The subscript cp refers to the distance to the escape point, g is the distance from the escape point to the goal, pp is the distance between the escape point and the previously calculated escape point. Lastly, $cert$ refers to the average uncertainty of the pixels neighbouring the escape point. The runtime needed to calculate the escape point with the lowest cost takes 0.22 msec on average for the 1241x376 disparity image.

$$Cost(p) = w_1 * d_{cp} + w_2 * d_g + w_3 * d_{pp} + w_4 * cert_p \tag{4.1}$$

The benefit of this method compared to 3DBug is that it does not necessarily need to track the complete obstacle boundary of the obstacle in front of the drone, it only tracks the boundaries which are of interest for planning new escape points. Furthermore, it is not necessary for obstacles to be represented as polyhedral objects, which makes the method more realistic and faster. Even if the uncertainty map would get rid of most of the noise in the disparity image presented for 3DBug, obstacle boundaries would still be hard to find since they are incomplete or inconsistent. For example,

after the c-space expansion, some pixels of nearby obstacles expand more than others due to small errors, as shown by the pole on the right in Figure 4.5.

4.5. Discussion

This chapter has presented multiple mapless path planning algorithms, along with their advantages, disadvantages and perhaps opportunities. It has been concluded that Fuzzy Logic is not chosen, due to the difficulties in selecting the best rules for particular situations and in designing rules that include every imaginable situation.

Machine learning has numerous disadvantages. Firstly, supervised learning requires many samples, plus a wide variety within the data. Furthermore, these algorithms remain unpredictable and global optima cannot be guaranteed. Secondly, unsupervised learning is only used to find patterns in the data, after which another path planning algorithm is still required to actually plan the path, increasing the computation. Thirdly, reinforcement learning will not be used, since even after careful tuning of the parameters, the algorithm still shows unpredictable behaviour.

Afterwards, this chapter dove into the world of bug algorithms, which were categorised under M-line-, angle- and range bug algorithms. The range bugs were the most interesting, since they do not adhere to following boundaries as much as the others. In particular, the computationally efficient implementation of TangentBug presented by the WedgeBug algorithm seemed a good opportunity. A 3D implementation of TangentBug was performed by 3DBug. Combining WedgeBug and 3DBug into a 3D local planning algorithm with limited sensor range and limited FoV looked promising. However, the assumption that all obstacles can be modelled as polyhedral obstacles is quite tricky in noisy image conditions.

To avoid finding all these obstacle boundaries, Lyrakis (2019) looks for escape points using a Moore-Neighbour Tracing algorithm. The best escape point is then chosen according to a cost function, and all of this happens in less than half a millisecond for a 1241x376 disparity image. The only challenges that remain are finding parameters for BM which result in the best disparity maps, avoiding local minima and checking the path for the drone's dynamic constraints. These are explained in section 5.1, section 5.2 and section 5.3.

5
Implementation

This chapter touches upon some of the subjects which were not discussed in detail before. This includes how to find the best parameters for the BM algorithm, how to avoid local minima and how to smooth the path. Furthermore, this chapter presents the method in which the resulting algorithm can be tested.

5.1. Selecting BM parameters

As discussed in section 1.3, BM comes with a set of parameters that can tune the disparity map. The parameters of interest for tuning are: SADWindowSize, numDisparities, textureThreshold, uniquenessRatio, speckleRange and speckleWindowSize. If any improvement is noticed by changing any of the other parameters, they will be included as well (i.e. preFilterType, preFilterSize, preFilterCap, minDisparity, roi1, roi2, disp12MaxDiff and dispType). As discussed in section 1.6, if the disparity image remains noisy after tuning the parameters, the uncertainty map will be created.

5.1.1. Optimisation method

Now, is it possible to avoid manually changing every single parameter and writing down performance results to find the optimal combination? Luckily, yes. Optimisation algorithms exist that can give an insight into the pareto-front of multiple objective functions. Since it is tricky to assign a weight factor representing the relative importance of objective functions, NSGA-III takes into account all of the objective functions and finds non-dominated solutions. A non-dominated solution is a solution that is not dominated by any other solution. When comparing two solutions, the performance of one is said to dominate the other one if the values of all objective functions are at least equal, and one is larger (Deb and Jain, 2013).

The performance of the disparity image can be tested by the number of pixels that have an error greater than 2 pixels, or the amount of pixels that are given a value, or the time it takes to run the algorithm with the current parameters. The power of NSGA-III is that outperforms other optimisation methods when the number of objective functions increases, to between 3 and 15. Furthermore, the solutions are in some sense connected to a set of equally spaced reference points. This prevents all solutions from moving to a local/global optimum, and maintains the spread in the solutions that is desired for representing a pareto-front.

5.1.2. Preliminary results

To test the performance of the NSGA-III algorithm for this study, a small example was used. Here, only two parameters were being given as input: SADWindowSize and textureThreshold. Three objective functions were created: px2_perc refers to the amount of pixels that have a disparity error of 2 pixels or smaller, values_perc refers to the amount of pixel values and time_taken measures the runtime for the set of parameters. The objective function px2_perc has been chosen to represent errors of 2 and smaller (instead of 3 and larger) to allow for better connection with the reference points. The results are shown in Figure 5.1. Note that the axes are scaled to be consistent with the reference points used, so the values in the graph don't represent the real values. For completeness, they can be transformed to their true values using Equation 5.1, Equation 5.2 and Equation 5.3. The values used in these equations result from the attempt to spread the population data between 0 and 1 on the different axes.

$$px2\_perc\_true[-] = \frac{px2\_perc}{5} + 0.8 \tag{5.1}$$

$$values\_perc\_true[-] = values\_perc \cdot 0.7 \tag{5.2}$$

$$time\_taken\_true[seconds] = \frac{time\_taken}{600} + 0.0004 \tag{5.3}$$

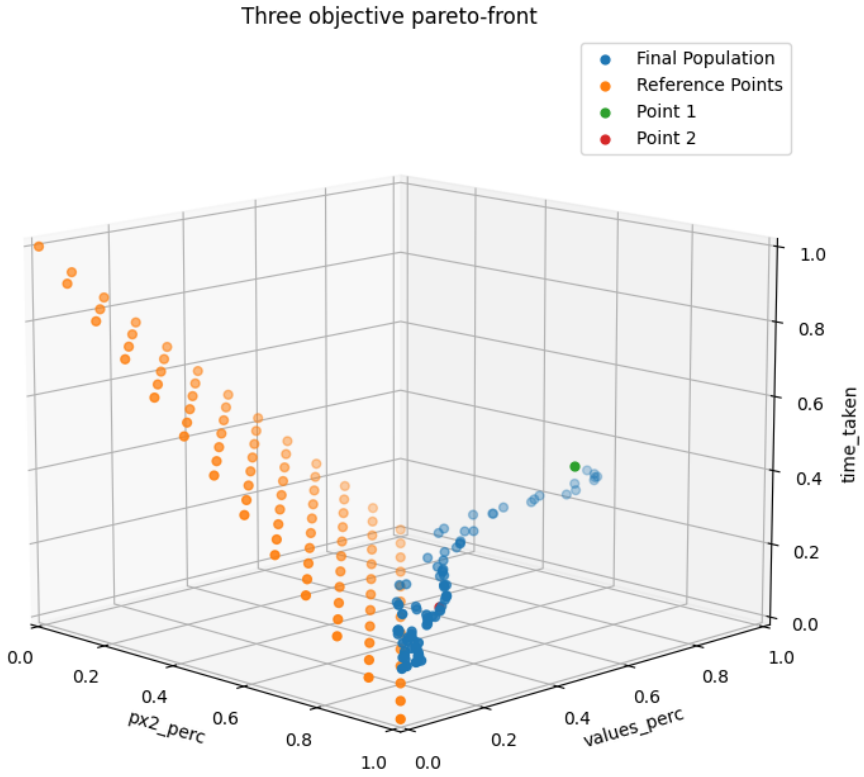
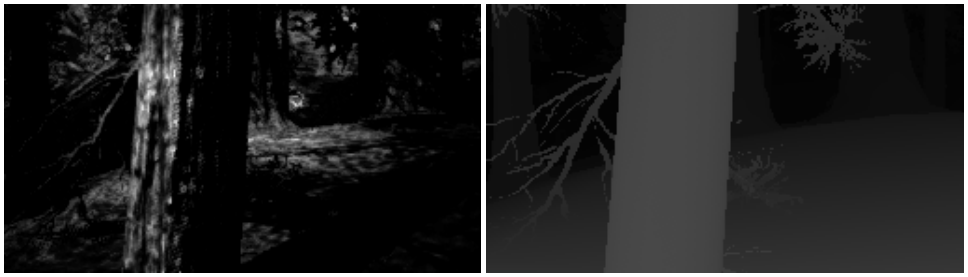


Figure 5.1: NSGA-III preliminary results for finding BM parameters

The orange dots represent the reference points used in the optimisation. The blue dots represent the population of 92 members, after going through 20 generations. Each member consists of an array of 2 values: one for the SADWindowSize and one for the textureThreshold. The number of members roughly equals the number of reference points, as recommended by the original paper (Deb and Jain, 2013). The number of generations in the paper is 400. Here, 20 is used since it gives a quick indication of how the algorithm performs, and the most optimal solution is not yet the main objective. Finding optimal parameters for population size, number of generations and crossover/mutation probabilities is left for further thesis work. For this, the Grid Search tuning algorithm used by Lyrakis to find the best hyperparameters can be used.

When the number of pixels that are given a value (values_perc) increases, the number of pixels with a disparity error of 2 or smaller decreases (px2_perc). In other words: the more values being estimated, the more mistakes being made. Another observation is that the more values being estimated, the more time it takes to run. To give an insight into these population members, two are highlighted: Point 1 (green) belongs to the member with the highest values_perc value, and Point 2 (red) belongs to the member which is the median from all values_perc. For this preliminary study, the disparity image of only one stereo image was calculated. The left grayscale image of a stereo pair

is shown in Figure 5.2(a), and its corresponding true disparity image is shown in Figure 5.2(b). Point 1 and 2 are taken as an example. The pareto-front allows the user to decide the ideal combination of different objective function values.



((a)) Left image of stereo pair ((b)) True disparity image

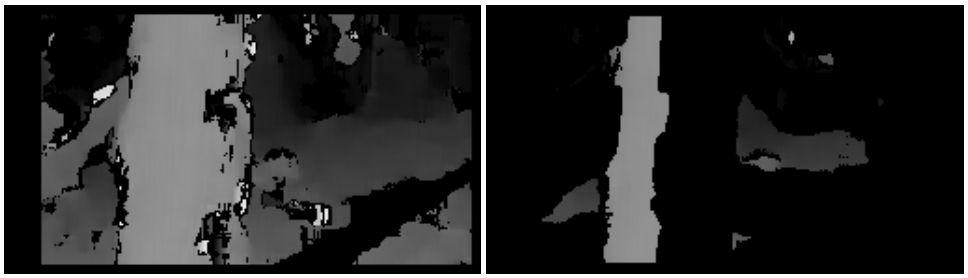
Figure 5.2: The left image together with its disparity image

Table 5.1 shows the parameter values of the two points, as well as the resulting objective function values. The disparity images created by the parameters of Point 1 and Point 2 are shown in Figure 5.3(a) and Figure 5.3(b) respectively. The parameters belonging to population member Point 1 generate a disparity image where many pixels are given a value, although the pixels with errors equal to or smaller than 2 drops to 92.3%. The parameters of point 2 only create 16 % of pixel values, but they perform slightly better: 97.5%.

Table 5.1: Parameters and results for Point 1 and 2

	SADWindowSize	**textureThreshold**	**px2_perc**	**values_perc**	**time_taken**
Point 1	11	21	92.3 [%]	61.8 [%]	0.986 [ms]
Point 2	13	3970	97.5 [%]	16.0 [%]	0.723 [ms]

As a sidenote, BM will always create a black frame around the disparity image. Most of the left column is due to the numDisparities. The value used here is 32. Since the algorithm looks for pixel disparities between 0 and 31 pixels, it is not able to draw any conclusions about the first 31 pixels, since it is not possible to examine their complete range of possible disparities. The rest of the frame equals half the SADWindowSize, rounded down. This is because the block scans through the image, and assigns values to the central cell. This means that for a SADWindowSize of 11 (Point 1), it is not able to give values to any of the outer five row and columns, since the complete block is not defined in that area. Hence, for a SADWindowSize of 11 the maximum attainable percentage for a 256x144 image is 78.2%. For a SADWindowSize of 13 (Point 2) this drops to 76.3%.



((a)) Point 1 ((b)) Point 2

Figure 5.3: Disparity images created through BM using both points' parameters

What is left for further thesis work is to include more BM parameters into the NSGA-III optimisation algorithm, and decide which solution in the pareto-front will be the preferred solution.

5.2. Avoiding local minima

Since the mapless method by Lyrakis (2019) is chosen, the famous problem of local minima needs to be addressed. To clarify, with a local minima it is meant that the drone is stuck in a position which is not the target. For example, with potential field methods it might happen that the cumulative repulsive force vector of the obstacles is equal and opposite to the attractive force vector from the target. The net force is zero and the drone might think that it has arrived at the target position.

While avoiding local minima sounds ideal, it is not very realistic. Since a local approach is used, the drone will not be able to spot a local minima before it has entered it. Solving for local minima has been of particular interest for potential field methods, since this was one of their main drawbacks. Some solutions discussed in subsection 3.2.1 included rotational forces, virtual obstacles, virtual targets and boundary following. The main idea is that the robot entered a different mode, once it has realised that it was stuck in a local minima.

Since the PF method itself is not used, virtual obstacles will not be considered, also since a newly generated obstacle nearby the drone will block the complete FoV due to the c-space expansion. However, the idea behind the other three options could still be implemented.

Instead of a virtual target, a new waypoint could be set on one of the already visited GPS locations, to review the decision taken a few steps ago. In addition, it is already known that the path from current location to the previous is safe, so the drone could even fly backwards while keeping a forward view, trying to find better escape points. The rotational forces could be translated into a rotational movement of the drone. There are two possibilities for this. Lyrakis (2019) chose the simple method, which included the yawing motion of a stationary drone, which allowed for safe further scanning of the environment. Another option is to fly the drone in spirals, which allows it to move up or down. These spirals are necessary since the drone does not know what is directly above or below. Boundary following methods performed in the literature are on 2D maps. A similar approach can be designed for 3D, where the drone can also move up or down while following the boundary, a bit like the spiralling approach.

The exact implementation and the discovery of possible advantages and disadvantages is left for further thesis work.

5.3. Path smoothing

Every time an escape point is selected, the drone changes its attitude and heads towards this point. Due to the dynamics of the drone, there is a small delay and the drone does not exactly follow the straight line segment between the escape point and the location of the drone at the time where this escape point was calculated. In the worst case scenario, this deviation is large enough to cause an obstacle collision.

In subsection 3.1.2, some curve based path-smoothing algorithms were discussed. The most used one is Dubins method due to its simplicity, and it can take turning radius and rate of climb into account. Another cheap option that is worth researching is the use of splines. Splines are functions dedicated to a small section of a curve (in this case: the path). Each point on the resulting Dubins curve or spline can then be checked for obstacle collision. An overview of other popular path smoothing techniques is presented by Ravankar et al. (2018). Since the importance of path

smoothing is yet unknown, the exact implementation is left for further thesis work.

5.4. AirSim

The open-source simulator used is developed by Microsoft and is called AirSim: Aerial Informatics and Robotics Simulation [1]. The simulator is meant for AI research and is built on Unreal Engine to generate different environments. One of the outdoor environments created by AirSim is a forest. This will give the algorithm relatively simple obstacles to avoid, yet it should watch out for low hanging branches and leaves. Once the drone is able to successfully avoid obstacles in this forest, other environments could be put to the test as well.

5.5. Real drone

Once the algorithm successfully avoids obstacles in the simulated environments, it can be put to the test on an actual drone. The simulation uses Python APIs to communicate with the drone. On the real drone the C/C++ programming language is used, hence the Python code must be translated.

5.6. Performance evaluation

As stated in the introduction of chapter 3, there is no common benchmark for evaluating path planning algorithms. To address this problem, Nous et al. (2016) created metric for both detection and avoidance performances. To obtain level playground for all the possible simulation environments, they are described using quantitative metrics.

5.6.1. Detection

The performance of the detection algorithm is dependent on the environment. Obstacles can only be detected under certain conditions. The environmental quantitative metrics used here are distance, reflectivity, illumination, texture and transparency. It is possible to find bounds for these metrics in which obstacles are detected well. Since reflectivity and transparency cause problems, no matter what stereo algorithm is chosen (Lyrakis, 2019), they will not be used for evaluating the proposed algorithm. Furthermore, chapter stated that the environment is assumed to be well illuminated and textured. Therefore, the main interest in this study is the distance metric.

To calculate the performance of detection, depth estimations can be compared to true depth values in simulation for a range of distances. This comparison can consist of average error and variance. A threshold for allowable error can be set to obtain the operational conditions in which the detection method can be used.

5.6.2. Avoidance

The performance of the avoidance algorithm is also dependent on the environment. Examples used by Nous et al. (2016) are traversability, collision state percentage and average avoidance length. The first is related to the obstacle density, the second to the workspace (or: achievable states) and the third to the relative size of the obstacles to the drone. Since the simulated environment used is a forest, different sections of the forest can be used to obtain a range of values for the metrics.

The bounds on the quantitative environmental metrics can be found by calculating the performance of the avoidance algorithm for different values. The performance will be quantified by the success rate and the path optimality (Nous et al., 2016).

[1] https://microsoft.github.io/AirSim/

6

Conclusion

This report has presented a rich history of path planning for robots, and has been applied to path planning for drones where possible. A computationally cheap sense and avoid algorithm eliminates the need for heavy sensing equipment and large on-board computers. The search for a high performance distance measurement sensor led to a stereo camera, which answers research question 1. It has relatively low computational requirements, and it has good accuracy since the outdoor environment is highly textured. The drone's dimensions are included in the disparity map through obstacle expansion in disparity space.

The purpose of research question 2 was to address the trade-off between keeping computational expenses low and avoiding obstacles, as well as avoiding local minima and finding paths that resemble global optimal solutions. Research question 3 also plays a particularly interesting role in this: is it necessary to create a map of the environment? The quick answer to this is "no", since mapless path planning algorithms exist. But, since these reactive methods tend to suffer from local minima and local optima, perhaps a computationally cheap map could improve on that. Storing obstacles as polyhedral objects made mappings relatively cheap. These could be used for either a roadmap-like Convex-Edges-Graph, or simply an empty workspace. However, practical implementations arise since the disparity images are noisy. Because of this, the obstacles will not have clear boundaries and polyhedral objects that represent these obstacles will need many faces if they were to be drawn accurately, increasing the computation.

These practical implementations are not the only downside of map-based path planning algorithms. Assuming there is a way to approximate obstacles as polyhedral, there are two main approaches: optimisation under constraints and potential field methods. The first has relatively high computational requirements, while still being vulnerable to local optima. The second tends to get stuck in local minima, even though it spent computational resources on building a map. Although the first is less likely to get caught by local minima, these situations are often caused by obstacles that are located in a nontrivial way. Since it is unlikely that these situations happen often, a mapless path planning algorithm is chosen.

Once the drone gets close to an obstacle, it will calculate a new waypoint according to a cost function formulated by Lyrakis (2019). This function finds the locally best solution. Finding sequential locally best solutions will result in the global optimal solution. This is not true when local minima are encountered, for which a different flight mode will need to be used. The available literature on this topic is limited, but either spiralling, boundary following or returning to a previously visited GPS point are possible solutions. These options will need to be evaluated in further thesis work. Furthermore, parameters for creating the disparity image will need to be optimised. In the end, the algorithm will be evaluated using the metrics used by Nous et al. (2016). The depth estimates will be compared to true depth values, and the avoidance strategy will be tested for a range of values for obstacle density and size.

The algorithm will be tested in the open-source simulator AirSim, where outdoor scenes can be simulated. These tests can conclude whether any post-smoothing of the planned path is necessary. If time allows, once the algorithm has been tested in numerous simulations, it can be tested on a real drone. Once those tests are passed, a 3D obstacle avoidance algorithm has been successfully implemented, which can navigate a drone from outdoor GPS coordinate A to B, while staying close to its nominal path and keeping computational expenses low.

Bibliography

Adarsh, S., Kaleemuddin, S. M., Bose, D., & Ramachandran, K. (2016a). Performance comparison of infrared and ultrasonic sensors for obstacles of different materials in vehicle/robot navigation applications. IOP Conference Series: Materials Science and Engineering, 149(1), 012141.

Adarsh, S., Kaleemuddin, S. M., Bose, D., & Ramachandran, K. (2016b). Performance comparison of infrared and ultrasonic sensors for obstacles of different materials in vehicle/robot navigation applications. IOP Conference Series: Materials Science and Engineering, 149(1), 012141.

Antich, J., & Ortiz, A. (2005). Extending the potential fields approach to avoid trapping situations. 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 1386–1391.

Aswini, N., Kumar, E. K., & Uma, S. (2018). Uav and obstacle sensing techniques–a perspective. International Journal of Intelligent Unmanned Systems.

Bachrach, A., Prentice, S., He, R., Henry, P., Huang, A. S., Krainin, M., Maturana, D., Fox, D., & Roy, N. (2012). Estimation, planning, and mapping for autonomous flight using an rgb-d camera in gps-denied environments. The International Journal of Robotics Research, 31(11), 1320–1343.

Borenstein, J., & Koren, Y. (1989). Real-time obstacle avoidance for fast mobile robots. IEEE Transactions on systems, Man, and Cybernetics, 19(5), 1179–1187.

Borenstein, J., & Koren, Y. (1990). Real-time obstacle avoidance for fast mobile robots in cluttered environments. Proceedings., IEEE International Conference on Robotics and Automation, 572–577.

Brownlee, J. (2019). 14 different types of learning in machine learning [Retrieved on September 24th, 2021].

Chen, Y.-W., & Chiu, W.-Y. (2015). Optimal robot path planning system by using a neural network-based approach. 2015 international automatic control conference (CACS), 85–90.

Culligan, K., Valenti, M., Kuwata, Y., & How, J. P. (2007). Three-dimensional flight experiments using on-line mixed-integer linear programming trajectory optimization. 2007 American Control Conference, 5322–5327.

Culligan, K. F. (2006). Online trajectory planning for uavs using mixed integer linear programming (Doctoral dissertation). Massachusetts Institute of Technology.

Dai, R., & Cochran, J. E. (2009). Path planning for multiple unmanned aerial vehicles by parameterized cornu-spirals. 2009 American Control Conference, 2391–2396.

De Croon, G., Ho, H., De Wagter, C., Van Kampen, E., Remes, B., & Chu, Q. (2013). Optic-flow based slope estimation for autonomous landing. International Journal of Micro Air Vehicles, 5(4), 287–297.

De Croon, G., Gerke, P. K., & Sprinkhuizen-Kuyper, I. (2014). Crowdsourcing as a methodology to obtain large and varied robotic data sets. 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, 1595–1600.

De Filippis, L., Guglieri, G., & Quagliotti, F. (2012). Path planning strategies for uavs in 3d environments. Journal of Intelligent & Robotic Systems, 65(1), 247–264.

Deb, K., & Jain, H. (2013). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. IEEE transactions on evolutionary computation, 18(4), 577–601.

de Croon, G. (2016). Derivation of optical flow formulas for course ae4317.

de Croon, G. (2021). Computer vision for micro air vehicles iii [Retrieved on October 18th, 2021].

Dijkstra, E. W., et al. (1959). A note on two problems in connexion with graphs. Numerische mathematik, 1(1), 269–271.

Divelbiss, A. W., & Wen, J. (1993). Nonholonomic path planning with inequality constraints. Proceedings of 32nd IEEE Conference on Decision and Control, 2712–2717.

Duan, Q., Gupta, V. K., & Sorooshian, S. (1993). Shuffled complex evolution approach for effective and efficient global minimization. Journal of optimization theory and applications, 76(3), 501–521.

Dubey, G., Arora, S., & Scherer, S. (2017). Droan—disparity-space representation for obstacle avoidance. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 1324–1330.

Duleba, I., & Sasiadek, J. Z. (2003). Nonholonomic motion planning based on newton algorithm with energy optimization. IEEE transactions on control systems technology, 11(3), 355–363.

Eusuff, M. M., & Lansey, K. E. (2003). Optimization of water distribution network design using the shuffled frog leaping algorithm. Journal of Water Resources planning and management, 129(3), 210–225.

Farnebäck, G. (2002). Polynomial expansion for orientation and motion estimation (Doctoral dissertation). Linköping University Electronic Press.

Fiorenzani, T. (2017). How do drones work? part 12 - distance measurement sensors [Retrieved on September 13th, 2021].

Gandomi, A. H., Yang, X.-S., & Alavi, A. H. (2011). Mixed variable structural optimization using firefly algorithm. Computers & Structures, 89(23-24), 2325–2336.

Geem, Z. W., Kim, J. H., & Loganathan, G. V. (2001). A new heuristic optimization algorithm: Harmony search. simulation, 76(2), 60–68.

Goerzen, C., Kong, Z., & Mettler, B. (2010). A survey of motion planning algorithms from the perspective of autonomous uav guidance. Journal of Intelligent and Robotic Systems, 57(1), 65–100.

Guruprasad, K. (2011). Egressbug: A real time path planning algorithm for a mobile robot in an unknown environment. International Conference on Advanced Computing, Networking and Security, 228–236.

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

Haupt, R. L., & Ellen Haupt, S. (2004). Practical genetic algorithms.

Hillier, F. S., & Liebermann, G. J. (2014). Operations research. Oldenbourg Wissenschaftsverlag.

Hirose, N., Sadeghian, A., Goebel, P., & Savarese, S. (2017). To go or not to go? a near unsupervised learning approach for robot navigation. arXiv preprint arXiv:1709.05439.

Hirschmuller, H. (2005). Accurate and efficient stereo processing by semi-global matching and mutual information. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2, 807–814.

Horiuchi, Y., & Noborio, H. (2001). Evaluation of path length made in sensor-based path-planning with the alternative following. Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164), 2, 1728–1735.

Jahanshahi, H., & Sari, N. N. (2018). Robot path planning algorithms: A review of theory and experiment. arXiv preprint arXiv:1805.08137.

Janglová, D. (2004). Neural networks in mobile robot motion. International Journal of Advanced Robotic Systems, 1(1), 2.

Jaradat, M. A. K., Al-Rousan, M., & Quadan, L. (2011). Reinforcement based mobile robot navigation in dynamic environment. Robotics and Computer-Integrated Manufacturing, 27(1), 135–149.

Jevois smart machine system [Retrieved on November 22nd, 2021]. (n.d.).

Kaldestad, K. B., Haddadin, S., Belder, R., Hovland, G., & Anisi, D. A. (2014). Collision avoidance with potential fields based on parallel processing of 3d-point cloud data on the gpu. 2014 IEEE International Conference on Robotics and Automation (ICRA), 3250–3257.

Kamon, I., Rimon, E., & Rivlin, E. (1998). Tangentbug: A range-sensor-based navigation algorithm. The International Journal of Robotics Research, 17(9), 934–953.

Kamon, I., Rimon, E., & Rivlin, E. (1999). Range-sensor based navigation in three dimensions. Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C), 1, 163–169.

Kamon, I., & Rivlin, E. (1997). Sensory-based motion planning with global proofs. IEEE transactions on Robotics and Automation, 13(6), 814–822.

Karaboga, D. (2005). An idea based on honey bee swarm for numerical optimization (tech. rep.). Citeseer.

Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. The international journal of robotics research, 30(7), 846–894.

Karur, K., Sharma, N., Dharmatti, C., & Siegel, J. E. (2021). A survey of path planning algorithms for mobile robots. Vehicles, 3(3), 448–468.

Kavraki, L. E., Svestka, P., Latombe, J.-C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE transactions on Robotics and Automation, 12(4), 566–580.

Keltjens, B., van Dijk, T., & de Croon, G. (2021). Self-supervised monocular depth estimation of untextured indoor rotated scenes. arXiv preprint arXiv:2106.12958.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. Proceedings of ICNN'95-international conference on neural networks, 4, 1942–1948.

Khatib, O., & Mampey, L. (1978). Fonction decision-commande d'un robot manipulateur. Rep, 2(7), 156.

Kim, D. H. (2009). Escaping route method for a trap situation in local path planning. International Journal of Control, Automation and Systems, 7(3), 495–500.

Koren, Y., Borenstein, J., et al. (1991). Potential field methods and their inherent limitations for mobile robot navigation. ICRA, 2, 1398–1404.

Laubach, S. L., & Burdick, J. W. (1999). An autonomous sensor-based path-planner for planetary microrovers. Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C), 1, 347–354.

LaValle, S. M., et al. (1998). Rapidly-exploring random trees: A new tool for path planning.

LaValle, S. M. (2006). Planning algorithms. Cambridge university press.

Lee, M. C., & Park, M. G. (2003). Artificial potential field based path planning for mobile robots using a virtual obstacle concept. Proceedings 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003), 2, 735–740.

Longuet-Higgins, H. C., & Prazdny, K. (1980). The interpretation of a moving retinal image. Proceedings of the Royal Society of London. Series B. Biological Sciences, 208(1173), 385–397.

Lumelsky, V., & Skewis, T. (1988). A paradigm for incorporating vision in the robot navigation function. Proceedings. 1988 IEEE International Conference on Robotics and Automation, 734–739 vol.2. https://doi.org/10.1109/ROBOT.1988.12146

Lumelsky, V., & Stepanov, A. (1986). Dynamic path planning for a mobile automaton with limited information on the environment. IEEE Transactions on Automatic Control, 31(11), 1058–1063. https://doi.org/10.1109/TAC.1986.1104175

Lundin, L., & Dath, C. (2018). Comparing different sampling-based motion planners in multiple configuration spaces.

Luo, C., & Yang, S. X. (2008). A bioinspired neural network for real-time concurrent map building and complete coverage robot navigation in unknown environments. IEEE Transactions on Neural Networks, 19(7), 1279–1298.

Lynch, K. (2017). 2.5 task space and workspace [Retrieved on October 21st, 2021].

Lyrakis, A. (2019). Low-cost stereo-based obstacle avoidance for small uavs using uncertainty maps.

Ma, C. S., & Miller, R. H. (2006). Milp optimal path planning for real-time applications. 2006 American Control Conference, 6–pp.

Mac, T. T., Copot, C., Tran, D. T., & De Keyser, R. (2016). Heuristic approaches in robot path planning: A survey. Robotics and Autonomous Systems, 86, 13–28.

Masehian, E., & Habibi, G. (2007). Robot path planning in 3d space using binary integer programming. International Journal of Mechanical System Science and Engineering, 23, 26–31.

Masehian, E., & Sedighizadeh, D. (2007). Classic and heuristic approaches in robot motion planning-a chronological review. World Academy of Science, Engineering and Technology, 23(5), 101–106.

Matthies, L., Brockers, R., Kuwata, Y., & Weiss, S. (2014). Stereo vision-based obstacle avoidance for micro air vehicles using disparity space. 2014 IEEE international conference on robotics and automation (ICRA), 3242–3249.

McGuire, K., De Croon, G., De Wagter, C., Tuyls, K., & Kappen, H. (2017). Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. IEEE Robotics and Automation Letters, 2(2), 1070–1076.

McGuire, K. N., de Croon, G., & Tuyls, K. (2019). A comparative study of bug algorithms for robot navigation. Robotics and Autonomous Systems, 121, 103261.

Mehrabian, A. R., & Lucas, C. (2006). A novel numerical optimization algorithm inspired from weed colonization. Ecological informatics, 1(4), 355–366.

Miller, B., Stepanyan, K., Miller, A., & Andreev, M. (2011). 3d path planning in a threat environment. 2011 50th IEEE Conference on Decision and Control and European Control Conference, 6864–6869.

Moeller, T. (2016). What is time-of-flight? – vision campus [Retrieved on October, 10th 2021].

Mohanty, P. K., & Parhi, D. R. (2013). Controlling the motion of an autonomous mobile robot using various techniques: A review. Journal of Advance Mechanical Engineering, 1(1), 24–39.

Moscato, P., Cotta, C., & Mendes, A. (2004). Memetic algorithms. In New optimization techniques in engineering (pp. 53–85). Springer.

Moses, A., Rutherford, M. J., & Valavanis, K. P. (2011). Radar-based detection and identification for miniature air vehicles. 2011 IEEE international conference on control applications (CCA), 933–940.

Muniz, F., Zalama, E., Gaudiano, P., & Lopez-Coronado, J. (1995). Neural controller for a mobile robot in a nonstationary environment. Proceedings of 2nd IFAC Conference on Intelligent Autonomous Vehicles, 279–284.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. The computer journal, 7(4), 308–313.

Ng, J. S. (2010). An analysis of mobile robot navigation algorithms in unknown environments (Doctoral dissertation). University of Western Australia.

Noborio, H., Maeda, Y., & Urakawa, K. (1999). Three or more dimensional sensor-based path-planning algorithm hd-i. Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289), 3, 1699–1706 vol.3. https://doi.org/10.1109/IROS.1999.811723

Noborio, H. (1992). A sufficient condition for designing a family of sensor-based deadlock-free path-planning algorithms. Advanced robotics, 7(5), 413–433.

Noborio, H., Yamamoto, I., & Komaki, T. (2000). Sensor-based path-planning algorithms for a non-holonomic mobile robot. Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113), 2, 917–924.

Noborio, H., Yoshioka, T., & Tominaga, S. (1997). On the sensor-based navigation by changing a direction to follow an encountered obstacle. Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS'97, 2, 510–517.

Nous, C., Meertens, R., De Wagter, C., & De Croon, G. (2016). Performance evaluation in obstacle avoidance. 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 3614–3619.

Nyasulu, T. D., Du, S., van Wyk, B., & Tu, C. (2017). Comparison study of low-cost obstacle sensing solutions for unmanned aerial vehicles in wildlife scenery. 2017 3rd IEEE International Conference on Computer and Communications (ICCC), 1072–1076.

Pandey, A., Sonkar, R. K., Pandey, K. K., & Parhi, D. (2014). Path planning navigation of mobile robot with obstacles avoidance using fuzzy logic controller. 2014 IEEE 8th international conference on intelligent systems and control (ISCO), 39–41.

Passino, K. M. (2002). Biomimicry of bacterial foraging for distributed optimization and control. IEEE control systems magazine, 22(3), 52–67.

Patle, B., Pandey, A., Parhi, D., Jagadeesh, A., et al. (2019). A review: On path planning strategies for navigation of mobile robot. Defence Technology, 15(4), 582–606.

Perrollaz, M., Yoder, J.-D., Nègre, A., Spalanzani, A., & Laugier, C. (2012). A visibility-based approach for occupancy grid computation in disparity space. IEEE Transactions on Intelligent Transportation Systems, 13(3), 1383–1393.

Radmanesh, M., Kumar, M., Guentert, P. H., & Sarim, M. (2018). Overview of path-planning and obstacle avoidance algorithms for uavs: A comparative study. Unmanned systems, 6(02), 95–118.

Ravankar, A., Ravankar, A. A., Kobayashi, Y., Hoshino, Y., & Peng, C.-C. (2018). Path smoothing techniques in robot navigation: State-of-the-art, current and future challenges. Sensors, 18(9), 3170.

Remes, B., Hensen, D., Van Tienen, F., De Wagter, C., Van der Horst, E., & De Croon, G. (2013). Paparazzi: How to make a swarm of parrot ar drones fly autonomously based on gps. International Micro Air Vehicle Conference and Flight Competition (IMAV2013), 17–20.

Richards, A., & How, J. P. (2002). Aircraft trajectory planning with collision avoidance using mixed integer linear programming. Proceedings of the 2002 American Control Conference (IEEE Cat. No. CH37301), 3, 1936–1941.

Rusu, C., Birou, I., & Szöke, E. (2010). Fuzzy based obstacle avoidance system for autonomous mobile robot. 2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 1, 1–6.

Samadi, M., & Othman, M. F. (2013). Global path planning for autonomous mobile robot using genetic algorithm. 2013 International Conference on Signal-Image Technology & Internet-Based Systems, 726–730.

Samir, N. (2014). Collision avoidance and simple path planning for autonomous robotic exploration.

Sanchez-Rodriguez, J.-P., & Aceves-Lopez, A. (2018). A survey on stereo vision-based autonomous navigation for multi-rotor muavs. Robotica, 36(8), 1225–1243.

Sankaranarayanan, A., & Vidyasagar, M. (1990). A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. Proceedings., IEEE International Conference on Robotics and Automation, 1930–1936.

Sankaranarayanar, A., & Vidyasagar, M. (1990). Path planning for moving a point object amidst unknown obstacles in a plane: A new algorithm and a general theory for algorithm development. 29th IEEE Conference on Decision and Control, 1111–1119.

Setyawan, R. A., Sunoko, R., Choiron, M. A., & Rahardjo, P. M. (2018). Implementation of stereo vision semi-global block matching methods for distance measurement. Indonesian Journal of Electrical Engineering and Computer Science (IJEECS), 12(2), 585–591.

Sfeir, J., Saad, M., & Saliah-Hassane, H. (2011). An improved artificial potential field approach to real-time mobile robot path planning in an unknown environment. 2011 IEEE international symposium on robotic and sensors environments (ROSE), 208–213.

Shen, S., Michael, N., & Kumar, V. (2011). 3d indoor exploration with a computationally constrained mav. Robotics: science and Systems, 2.

Shin, H., & Chae, J. (2020). A performance review of collision-free path planning algorithms. Electronics, 9(2), 316.

Singh, M. K., & Parhi, D. R. (2011). Path optimisation of a mobile robot using an artificial neural network controller. International Journal of Systems Science, 42(1), 107–120.

Singh, M. K., Parhi, D. R., & Pothal, J. K. (2009). Anfis approach for navigation of mobile robots. 2009 International Conference on Advances in Recent Technologies in Communication and Computing, 727–731. https://doi.org/10.1109/ARTCom.2009.119

Storn, R., & Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. Journal of global optimization, 11(4), 341–359.

Tai, L., Li, S., & Liu, M. (2016). A deep-network solution towards model-less obstacle avoidance. 2016 IEEE/RSJ international conference on intelligent robots and systems (IROS), 2759–2764.

Triharminto, H. H., Prabuwono, A. S., Adji, T. B., Setiawan, N. A., & Chong, N. Y. (2013). Uav dynamic path planning for intercepting of a moving target: A review. FIRA RoboWorld Congress, 206–219.

Ulrich, I., & Borenstein, J. (1998). Vfh+: Reliable obstacle avoidance for fast mobile robots. Proceedings. 1998 IEEE international conference on robotics and automation (Cat. No. 98CH36146), 2, 1572–1577.

Ulrich, I., & Borenstein, J. (2000). Vfh/sup*: Local obstacle avoidance with look-ahead verification. Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), 3, 2505–2511.

van Dijk, T. (2020). Self-supervised learning for visual obstacle avoidance (tech. rep.) [Technical report]. Micro Air Vehicle Lab (MAVLab), TU Delft.

van Hecke, K., de Croon, G., van der Maaten, L., Hennes, D., & Izzo, D. (2018). Persistent self-supervised learning: From stereo to monocular vision for obstacle avoidance. International Journal of Micro Air Vehicles, 10(2), 186–206.

Wahyunggoro, O., Cahyadi, A. I., et al. (2016). Quadrotor path planning based on modified fuzzy cell decomposition algorithm. Telkomnika, 14(2).

Wu, Q., Shen, X., Jin, Y., Chen, Z., Li, S., Khan, A. H., & Chen, D. (2019). Intelligent beetle antennae search for uav sensing and avoidance of obstacles. Sensors, 19(8), 1758.

Xu, Q.-l. (2014). Randombug: Novel path planning algorithm in unknown environment. The Open Electrical & Electronic Engineering Journal, 8(1).

Xu, Q.-L., & Tang, G.-Y. (2013). Vectorization path planning for autonomous mobile agent in unknown environment. Neural Computing and Applications, 23(7), 2129–2135.

Yang, L., Qi, J., Song, D., Xiao, J., Han, J., & Xia, Y. (2016). Survey of robot 3d path planning algorithms. Journal of Control Science and Engineering, 2016.

Yang, S. X., & Meng, M. (2000). An efficient neural network approach to dynamic robot motion planning. Neural networks, 13(2), 143–148.

Yang, X.-S. (2010a). Nature-inspired metaheuristic algorithms. Luniver press.

Yang, X.-S. (2010b). A new metaheuristic bat-inspired algorithm. In Nature inspired cooperative strategies for optimization (nicso 2010) (pp. 65–74). Springer.

Yang, X.-S., & Deb, S. (2009). Cuckoo search via lévy flights. 2009 World congress on nature & biologically inspired computing (NaBIC), 210–214.

Yershova, A., Jaillet, L., Siméon, T., & LaValle, S. M. (2005). Dynamic-domain rrts: Efficient exploration by controlling the sampling domain. Proceedings of the 2005 IEEE international conference on robotics and automation, 3856–3861.

Zadeh, L. (1965). Fuzzy sets. Information and Control, 8(3), 338–353. https://doi.org/https://doi.org/10.1016/S0019-9958(65)90241-X

Zhao, Y., Zheng, Z., & Liu, Y. (2018). Survey on computational-intelligence-based uav path planning. Knowledge-Based Systems, 158, 54–64.