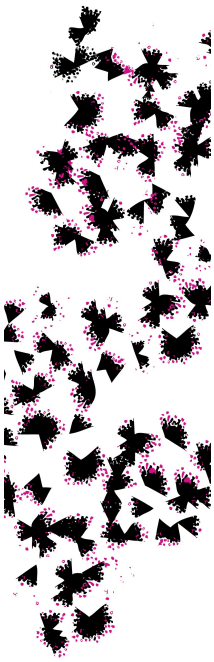

Waypoint navigation with obstacle avoidance for MAV's

R. (Ramon) Jansen

BSc Report

Committee:
D. Dresscher, MSc
Dr.ir. J.F. Broenink

August 2016

028RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Inspection of large plants and infrastructures done by humans takes time and can be dangerous. Inspection sites can be hard to reach. The use of MAV's has become popular over the last years and has made these kinds of inspections easier. However, controlling an MAV requires a skilled pilot.

During this project a system will be designed for an MAV that is equipped with an inertial measurement unit (IMU) and a stereocamera. The goal of this system is to enable the MAV to fly itself to a desired inspection site. An inspection expert will be able to do the inspection from the ground without the need of piloting skills.

The system is divided in several parts. The goal of this assignment is to write an algorithm that can be used to plan a path for an MAV and that is capable of planning the fastest route from its current location to a certain waypoint while avoiding any obstacles. This algorithm will be tested in a simulation program and not on the hexacopter itself. The inputs will be a 6 DoF pose estimation of the current pose and a 3-dimensional waypoint location. The output will be an array of 3-dimensional waypoints which form the path to the goal. This output is then used as an input for the controller.

An algorithm based on the A*-algorithm was designed to find the shortest path from the current location of the MAV to a destination while avoiding any obstacles. The first implementation did cause some problems. With some optimization, the final algorithm has provided us with good results on our own PC's. However, the results received from the challenge stated that the system had not settled during the testing by EuRoC. The overall score for the path-planning part of the challenge was satisfactory.

Even though the algorithm did function satisfactory given the time that was available, there is still room for improvement. Some experiments have been done with adding a second algorithm as an extra safeguard but was not added. This will reduce the chance of a collision. Secondly, the algorithm sometimes creates an illogical path where a straight line would be sufficient. This indicates an error in the algorithm and should be looked into.

Acknowledgments

I would like to thank Douwe Dresscher for letting me be a part of this team, for supervising me during my assignment, and, together with Geert Folkertsma, for supporting and thinking with the team during the challenge. I would also like to thank Matteo Fumagalli for coordinating team LEO during the challenge and for putting a lot of effort in this challenge. And of course I also want to thank the other members of my team Hengameh, Evytar, Roald and Mohamed. It was great to be a part of such a motivated team that as determined to get everything working no matter what, and it was really satisfying to get some good results.

Table of contents

1. Introduction

1.1 Context

Inspection of large plants and infrastructures done by humans takes time and can be dangerous. Inspection sites can be hard to reach. The use of MAV's has become popular over the last years and makes this a whole lot easier. However, controlling an MAV requires a skilled pilot.

Since an MAV is a naturally unstable platform and will be used inside as well as outside, the designed control system has to be robust since MAV's are easily destroyed. The system has to be able to independently fly the platform around without the need of interference by a person such that an inspection expert can completely focus on the inspection.

1.1.1 Project structure

This challenge is part of the European Robotics Challenges (EuRoC) and is carried out by team LEO. LEO is a collaboration of regional corporations and universities that want to transmit their skills and knowledge in service robotics around the world. The challenge is divided into three stages:

1. Simulation stage: the teams have to use the provided simulation environment to test their algorithms for localization, mapping, controlling and navigation. The fifteen best teams will team up with end users and technology developers and come up with a fifteen page proposal.
2. Realistic labs: the five best teams of the previous stage will participate in an end user-driven task which will be aimed at showcasing customizability under realistic conditions.
3. Field test: the best three teams of the previous stage will do real-life experiments at an end user site to test their solution. After this stage, the winner of EuRoC is determined.

The simulation stage is carried out in a simulated environment provided by EuRoC. Two Ubuntu virtual machines are provided by EuRoC, one challenger virtual machine and one simulation virtual machine, with ROS (Robot Operating System) combined with all needed packages and Gazebo pre-installed, refer figure 1.1. The simulation virtual machine contains a model of the MAV which is the same for all participants and is not to be changed. The solutions can be programmed in the client virtual machine.

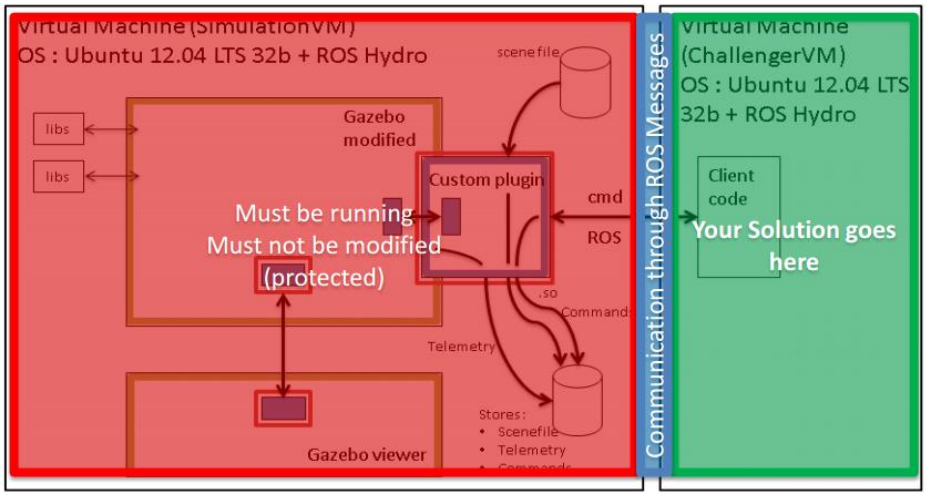

Figure 1.1: Simulation virtual machine and challenger virtual machine

The simulation stage is divided in two tracks with each two different tasks. Each track is divided in two teams, one for each task. The two teams within a track have to have good communication, since ultimately they will depend on each other's data. Each track has a small team of students from Saxion and University of Twente working on it accompanied and supported by PhD-students and professors of the RaM-group. Below follows the division of the simulation stage:

Track 1

I. Task 1: Localization of the MAV

The goal of this task is localizing the MAV and tracking its movement to estimate its position at all times as accurately as possible with the use of two generic 6-DoF pose sensors and an IMU.

II. Task 2: Environment mapping

The goal of this task is mapping the environment and all obstacles within as accurately as possible with use of a virtual-inertial SLAM-sensor.

Track 2

III. Task 3: Hovering control

The goal of this task is designing a controller for the MAV to assure stable hovering in situations with no disturbances as well as in situations with constant and random disturbances.

IV. Task 4: Path planning and navigation

The goal of this task is planning the fastest route from the current location of the MAV to a certain waypoint while avoiding possible obstacles and navigating the MAV with the use of an OctoMap provided by EuRoC.

1.1.2 Track 2 structure

Figure 1.2 shows the structure of track 2 and the dependencies of the different tasks. As stated before, track 2 is divided in two teams; one for task 3 and one for task 4. It is important for both teams to have good communication since both tasks depend on each other's data as seen in figure 1.2.

Below a short description of the connection between each fraction of track 2 is given.

The simulation provides real time acceleration data which can be requested at all times. The Kalman filter uses this acceleration data provided by the simulation to estimate the current position and angle of the MAV.

For path planning and navigation, a 6 DOF vector containing the current state of the MAV, a target waypoint and an occupancy map of the environment is received as inputs. The target waypoint is provided by the simulation software as a 3D waypoint. An Octomap provides occupancy information about the environment with a resolution of 0.25m.

The controller receives a 3D waypoint array from the path planner. This data is processed together with the current position data from the Kalman filter and ultimately the six rotors of the MAV are controlled by sending a 6 DOF vector to the simulation software.

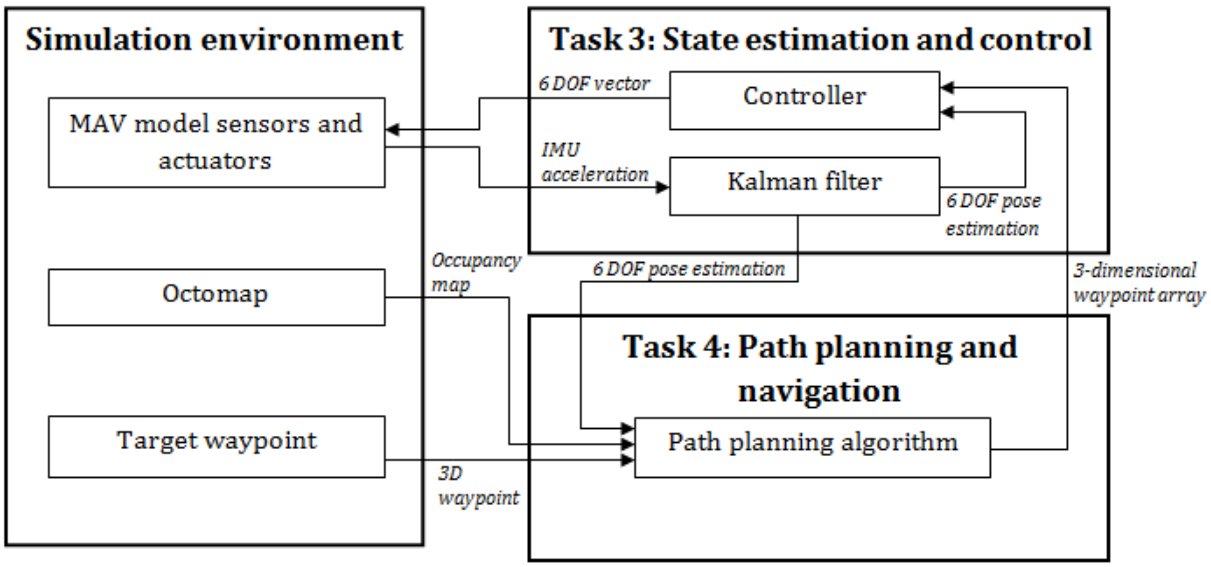

Figure 1.2: Block diagram of track 2

1.1.3 Thesis assignment

This assignment contributes to path planning and navigation. The goal of this assignment is to write an algorithm that can be used to plan a path for an MAV and that is capable of planning the fastest route from its current location to a certain waypoint while avoiding any obstacles.

Task 1 till task 4 are all divided in different subtasks. For each of these subtasks with increasing complexity, points can be scored based on several criteria in the final evaluation. For task 4, the subtasks are stated in table 1.1.

Subtask description	Criteria	Maximum score
Subtask 4.1: Waypoint navigation The goal is to plan a path from a starting point to a random waypoint. There will be no obstacles present.	Settling time	2
	Accuracy	2
	Energy efficiency	2
Subtask 4.2: Switching sensor input The goal is to plan a path from a starting point to a random waypoint while one of the sensor inputs is failing. Again there will be no obstacles present.	Settling time	3
	Accuracy	3
	Energy efficiency	3
Subtask 4.3: Navigation with obstacle avoidance The goal is to plan a path from a starting point to a random waypoint while avoiding collisions with obstacles.	Settling time	5
	Accuracy	5
	Energy efficiency	5

Table 1.1: Task 4 scoring table

1.2 Goals

The goal of this project is to develop a path planner that is exploring the design space provided by EuRoC and provides a collision-free path to a random waypoint within this design space.

The first part of this assignment is focused on finding background information about path planning algorithms and comparing them to select the algorithm best suited for this project.

The second part is the implementation of the selected path planning algorithm.

1.3 Approach

The focus is on scoring in every subtask, refer table 1.1. After having a basic functioning algorithm which is able to complete subtasks 4.1 and 4.2, the focus will be on the obstacle avoiding algorithm so that there will be a score on every subtask. After being able to score points on every subtask, improvements then can be made to the algorithm in order to achieve better scores.

1.4 Report structure

Chapter 2 treats the comparison between six possible and most suitable path planning algorithms as a result of a literature study. The comparison is made based on several requirements.

Chapter 3 shows the design and implementation of the final algorithm.

Chapter 4 discusses the results of intermediate tests and the final results of the challenge based on the criteria shown in table 1.1 and table 3.7.

Chapter 5 evaluates the final algorithm and gives conclusions and recommendations for future development.

2. Analysis

As stated in chapter 1, the first goal is to at least score points on every subtask. The primary requirements for achieving this within the given time are stated below. The algorithm should

- always finds a collision-free path, a single collision will result in 0 points for subtask 4.3.
- be easy to implement, so that quick results are obtained from within the given time, an so that there is probably time left to improve the algorithm. Please not that the scoring on this requirement is somewhat subjective.
- function well with at least a 3 DoF platform, i.e. is able to provide 3D vectors for movement along the x, y and z-axis.

The secondary requirements which would provide for a good algorithm are:

- It has to be efficient in calculating a path in a 3 dimensional environment. Fast computation of a path means that the total time needed for path planning and navigation is lower. This is one of the scoring criteria of EuRoC, refer table 1.1 (settling time).
- It should always try to find the shortest/quickest path to the target. The MAV will reach its destination quicker, this results in a lower settling time as well. Also, this is more energy efficient.
- It should aim for a smooth flight, no stuttering or difficult maneuvers if not necessary. The more straight the path is, the less maneuvering is needed. Maneuvering costs energy, so less maneuvering means more energy efficiency.

For designing and developing a suitable algorithm for subtask 4.3, a comparison is made between six obstacle avoiding algorithms. In this section, the working of each algorithm will be described briefly and the algorithm is evaluated based on the requirements stated above. Then a comparison between these six algorithms is made, refer section 3.2.7, and one is chosen as the base of the final algorithm.

2.1 Dijkstra's shortest path algorithm

This algorithm will expand outwards equally in every direction as can be seen in figure 3.1. It will put the neighboring nodes of the current node in a list. Each node will get a score depending on the cost of travelling to that node from the starting node. Then, the cheapest node is chosen and that will become the new 'current node'. The old current node is set as the parent of the new current node. When there are multiple nodes with the same score, they are all examined.

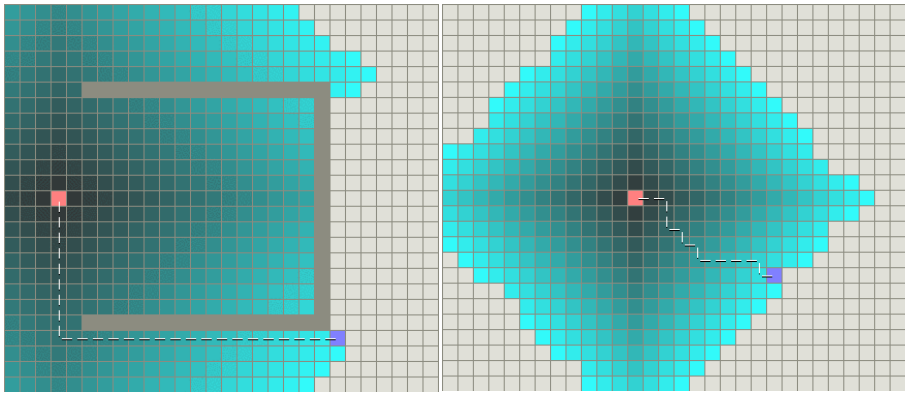

Figure 3.1 a) Expansion of Dijkstra's algorithm with b) and without obstacle. The red tile is the starting point, the purple tile is the goal. (Source: Stanford, A* algorithm)

The score of a node is based on the previous node, or its 'parent'-node. When a node can be reached via a cheaper path, that node will get a new parent. Let's say that all paths have equal costs and node1 has node2 as its parent. Node3 is examined, and has node1 as its neighbor. But the score of node3 is lower than the score of node2. That means that node1 can be reached quicker via node3. So the algorithm will reset node1's parent to node3. When two possible parents have the same score, one of them will be chosen at random. After all, both paths will most likely have the same length. [*Correll, Nikolaus (2011)*]

When the algorithm has reached the goal, the path will be found simply by noting each nodes parent working backwards starting at the goal.

Requirements	**Rating**
Collision-free path	Yes, when forbidden areas are well stated
Implementation difficulty	**Easy**/Medium/Hard
3 DoF functionality	Will work, but mostly used in 2D
Other criteria	
Path finding speed in 3D environment	**Slow**/Medium/Fast
Shortest path	Yes
Smooth flight	Grid-based, limited directions

Table 3.1: Dijkstra's shortest path

Dijkstra's algorithm will always find a collision-free path if the coordinates of the obstacles, the forbidden areas, are provided correctly. It is easy to implement and will always find the shortest path available due to the fact that it will check a wide amount of possibilities.

Because this algorithm expands in all directions, it is very slow in finding a path in 3D environments. This is also called the curse of dimensionality. Dijkstra's algorithm is mostly used in 2D environments. But when adding a third dimension, the number of computations needed to find a path increases significantly. For example, if we have a 2D design space of 10x10 nodes, there is a total of 100 possible nodes. If a third dimension is added, the total number of nodes increases by another factor 10, so that the total number of nodes is 10x10x10=1000 nodes. This generally means that to find a path, the number of computations needed also increases by a factor 10 in this example.

2.2 Best-first search algorithm

This algorithm does not use the distance from the starting node, but uses an estimate value called 'heuristic' to calculate the cost for each node. This heuristic value can be calculated in different ways depending on the application, these are visualized in figure 3.2.

The red line shows the Manhattan distance between point A and point B. The difference in the x-direction and the difference in the y direction between point A and point B are summed up. For three dimensions, the difference in the z-direction is added.

The orange line shows the diagonal distance. Diagonal movement is allowed. A diagonal step usually has a value of 1.4 times the step size ($\sqrt{1+1} \approx 1.4$).

Figure 3.2: Heuristic value determination

The green line shows the Euclidean distance. This is the absolute distance between point A and point B, and is calculated by:

$$H = \sqrt{x^2 + y^2}$$

or by:

$$H = \sqrt{x^2 + y^2 + z^2}$$

for a three dimensional grid. [*Red blob games, Heuristics*]

Figure 3.3 shows how greedy best-first search will develop and what path it calculates in the same configuration as showed with Dijkstra's algorithm in the previous chapter. In this configuration, the Manhattan-distance is used to calculate the heuristic value.

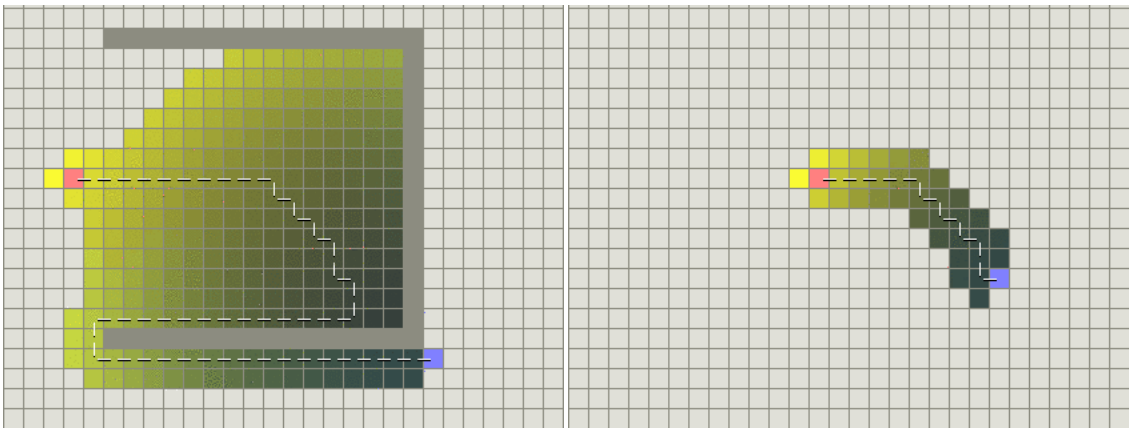

Figure 3.3 a) Expansion of the greedy best-first search algorithm with b) and without obstacle. The red tile is the starting point, the purple tile is the goal. (Source: Stanford, A* algorithm)

The algorithm works the same way as Dijkstra's, each node will have the heuristic value as its score. The neighboring nodes of the current node are stored in a list with their score. The cheapest one is chosen, that node will become the next 'current node', with the current node as its parent. If a node has two possible parents with the same score, one of them will be chosen as the parent since both resulting paths will have the same length. The algorithm stops when the goal is found or when no possible path is found. In the former case, the path is reconstructed by noting each nodes parent working backward from the goal node. [*Stanford, A* algorithm*]

Requirements	**Rating**
Collision-free path	Yes, when forbidden areas are well stated
Implementation difficulty	**Easy**/Medium/Hard
3 DoF functionality	Will work, but mostly used in 2D
Other criteria	
Path finding speed in 3D environment	**Slow**/**Medium**/Fast
Shortest path	Not guaranteed
Smooth flight	Grid-based, limited directions

Table 3.2: Best first search

The principle of this algorithm is the same as Dijkstra's algorithm with just minor differences and is also easy to implement. Compare the figure 3.3a) with 3.1a). As can be seen immediately, the path found by best-first search is longer than the path found with Dijkstra's algorithm. This is because each node is scored based on its distance from the goal instead of the distance to the starting point. Therefore, the closer a node is to the goal, the lower the score. The algorithm thus tends to go straight to the goal, and when an obstacle is found the path is redirected around this obstacle. With Dijkstra's algorithm, all nodes on the same radius from the starting point have the same score and thus a better path is found.

Compare 3.3a) with 3.2a). This algorithm checks a lot less nodes than Dijkstra's algorithm when no obstacle is in the way. This is because each node is scored based on its distance from the goal instead of the distance to the starting point. Since the nodes going towards the goal will have a lower score than the nodes away from the goal, the algorithm will always try to expand in the direction of the goal. Dijkstra's algorithm does not have information about the direction of the goal and thus expands in all directions.

Best-first search is more efficient when no obstacles are obstructing the ideal path. When this ideal path is obstructed, a less optimal path is found than with Dijkstra's algorithm.

2.3 A-star

This algorithm combines the best from the two algorithms discussed above. It takes into the account the heuristic distance to the goal as well as the distance from the starting point. This way, the algorithm will always find a shortest path. That is, when a path is available and with respecting the angle restrictions.

The algorithm starts with examining the neighbors of the current node. Each node will get two scores: one for heuristic and one for travelling cost from the starting point to that node. Let's call the heuristic score H, the travel cost score G. The total score F for a certain node is then:

$$F = H + G$$

The closer a node is to the goal, the lower the heuristic score H. The closer a node is to the starting point, the lower the travel cost score G. The nature of the algorithm depends on which score has a bigger weight. If for example the H score has a bigger weight (by for example incrementing with 5 points for every step), the nature of this algorithm is more like best-first search, while if the G score has a bigger weight the nature is more like Dijkstra's algorithm.

In figure 3.4, the expansion of the A-star algorithm is shown. Figure 3.4 a) shows the expansion with an obstacle between the starting point and goal. Figure 3.4 b) shows the expansion when no obstacle is present between the starting point and the goal.

[*Red blob games (2009), Introduction to A**], [*Winands, Mark (2004)*]

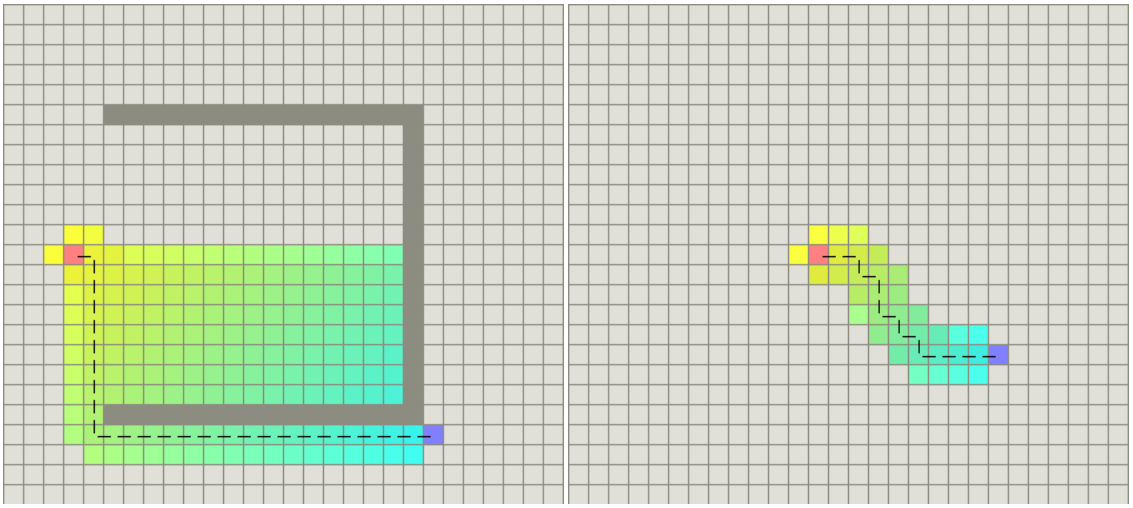

Figure 3.4 a) Expansion of the A-star algorithm with b) and without obstacles. The red tile is the starting point, the purple tile is the goal. (Source: Stanford, A* algorithm)

The working of this algorithm is as follows: all the neighbors are stored in a list, after which the node with the lowest total score F is chosen. When two nodes have the same lowest score, one of them is chosen at random as the parent since both resulting paths will have the same length. The current node is set to be this nodes parent. This continues until the goal is reached or when no path is found. That path is generated by tracing back each nodes parent, starting from the goal.

Requirements	**Rating**
Collision-free path	Yes, when forbidden areas are well stated
Implementation difficulty	**Easy**/**Medium**/Hard
3 DoF functionality	Will work, but mostly used in 2D
Other criteria	
Path finding speed in 3D environment	Slow/**Medium**/Fast
Shortest path	Yes
Smooth flight	Grid-based, limited directions

Table 3.3: A-star

A-star is a combination of Dijkstra's shortest path algorithm and best-first search algorithm. Benefits of both algorithms can be seen in figure 3.4. It will always find the shortest path to the goal like Dijkstra's (and unlike best-first search), but it will only scan nodes in the direction of the goal if possible like best-first search (and unlike Dijkstra's). The former guarantees that the shortest path is found, the latter makes the algorithm more efficient. This algorithm is a little more complex and is somewhat harder to implement than Dijkstra's or best-first search since it is a combination of the two algorithms.

2.4 Potential fields

This algorithm creates virtual forces that are acting on the object. The goal has acts as an attractive force on the object, and obstacles will act as repulsive forces. It can be compared to a marble rolling down the graph shown in figure 5.

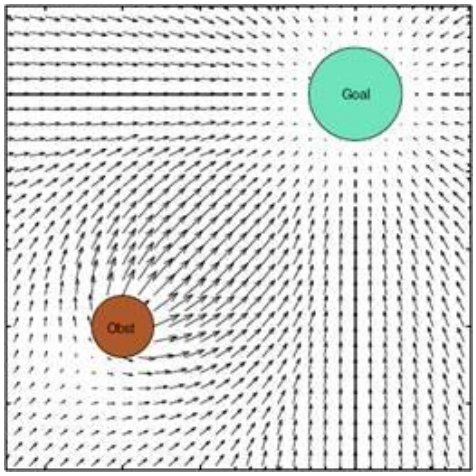

Figure 3.5: Visualization of a potential field (Source: Safadi, H. (2007))

The highest point is the starting point, the lowest point is the goal. When the marble rolls down this hill, it will automatically go around the pillars and get to the goal without crashing directly into the pillars. Another analogy would be a magnet being pulled towards the goal while being repulsed by obstacles, which will have the same polarization as the magnet.

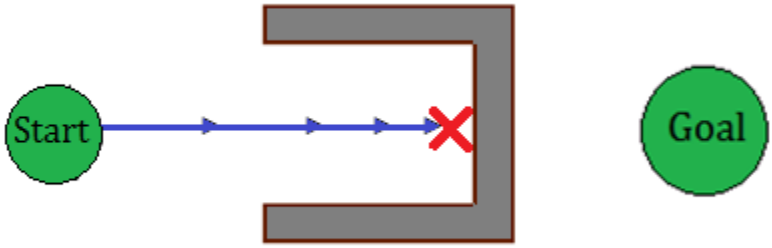

Figure 3.6: Visualization of a potential field with local minimum

The advantage of potential fields is that it can be used to navigate around obstacles during flight as a dynamic path planning algorithm, since the MAV will automatically be repulsed by obstacles before hitting them.

However, there is one big downside: while the previous types of algorithms will always find a path when there is one available, potential field can fail to find a path. This is because the algorithm can get stuck in a local minimum, for example the one shown in figure 3.6. This can be countered by adding random backward movement when the algorithm is stuck, but no distinction can be made between a local minimum and the absence of a path.

[*Safadi, H. (2007)*], [*Vaščák, J. (2007)*], [*Slideshare, Dynamic Path Planning*]

Requirements	**Rating**
Collision-free path	Not always finding path due to local minima
Implementation difficulty	Easy/Medium/**Hard**
3 DoF functionality	Yes
Other criteria	
Path finding speed in 3D environment	Slow/Medium/**Fast**
Shortest path	Not guaranteed
Smooth flight	Not guaranteed

Table 3.4: Potential fields

Potential fields are very well suited for 3D environments. It can even be used as a dynamic path planner, meaning that it can also avoid obstacles and find a path in an unknown environment while flying around. This will come in very handy in a lot of cases, since the complete environment is often unknown. Even when it has to plan a path before flight, it will be very fast in determining a path.

However, the big issue with potential fields is local minima. Therefore it is not always guaranteed that a path is found, even though one is available. If a path is found, it is not guaranteed to find the shortest path. It seemed to be more complex to implement, and few implementations are found on the internet or in books.

2.5 Harmonic potential fields

As stated in section A.4, potential fields can suffer from local minima which results in the algorithm failing to find a path. Harmonic potential field method is also an artificial function based on harmonic functions, which overcomes the limitations of potential field methods. Harmonic functions are solutions to the Laplaces equation (eq. A.1), the so-called harmonic equations (hence the name harmonic potential fields). The most important property of harmonic functions is that they are free from local minima. The core idea of this method lies in creation of only one minimum in the working environment i.e, the global minimum which is represented by the goal. If the goal is represented by a global minimum and no other minimum exists in the environment then the robot will arrive at the goal location always. Harmonic potential fields provide a solution to this.

$$\nabla^2 f = \Delta f = \frac{df^2}{dx} + \frac{df^2}{dy} + \frac{df^2}{dz} = 0 \qquad \text{(A.1)}$$

In equation A.1, f is a scalar function and in this case describes the space in which has to be navigated. The goal is to find a function f for which the divergence of the gradient is zero everywhere except at goal. The gradient of f is in the direction of the goal, since the goal has an attractive force. The divergence of is a measure for sources and sinks within the space described by f. If the divergence of the gradient of f is zero everywhere except for the goal, then there is certainly no local minimum. The goal will be a global minimum, and this way a path is always found if one is available.

Some nice papers about implementations using this method can be found, see references stated below.

[*Daily, Robert and David M. Bevly (2008)*], [*Masoud, Ahmad A. (2008)*]

Requirements	**Rating**
Collision-free path	Yes
Implementation difficulty	Easy/Medium/**Hard**
3 DoF functionality	Yes
Other criteria	
Path finding speed in 3D environment	Slow/**Medium**/**Fast**
Shortest path	Yes
Smooth flight	Yes

Table 3.5: Harmonic potential fields

Harmonic potential fields are guaranteed to always find a path if there is one available, this will also be the shortest path. It will be a little bit slower than normal potential field algorithms since more calculations have to be done. Opposing to normal potential fields, harmonic potential fields cannot be used for dynamic in-flight obstacle avoidance and path planning since the whole environment needs to be known in order to calculate the harmonic potential field.

The problem with harmonic potential fields is that the algorithms are often very complex and there is very little information to be found about this path planning method. Because of this, it will be a risk to try and implement a working harmonic potential field algorithm before the deadline.

2.6 Rapidly exploring random tree search algorithms

This algorithm is based on quick (random) exploration of a certain configuration space by filling it up with a 'tree'. The example in figure 3.8 start at the top left corner, this is the 'base' of the tree. It will continue to grow until it has found the goal, which is circled in green.

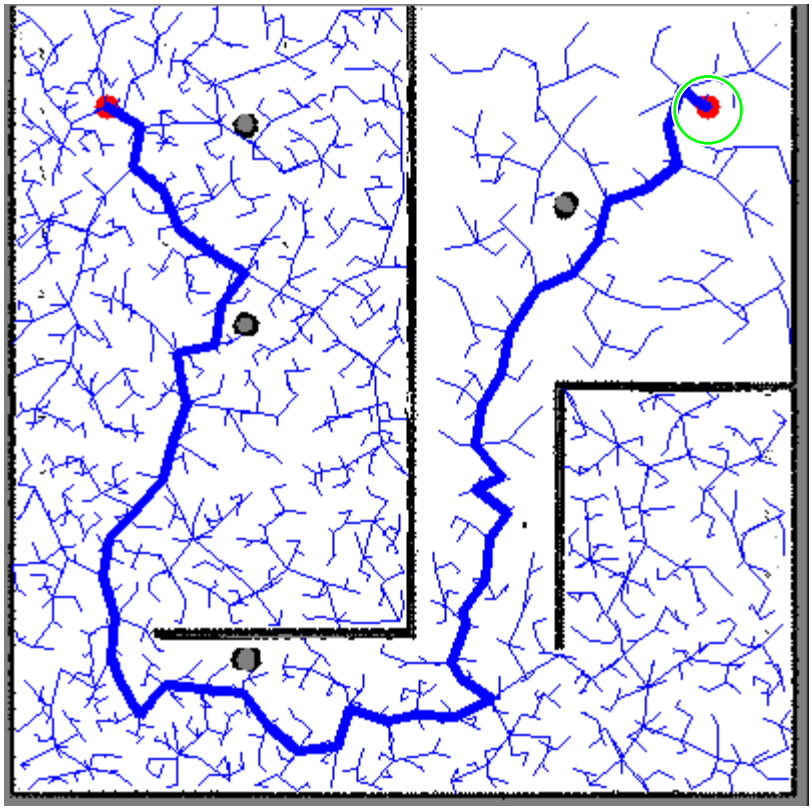

Figure 3.8: Example of the result of a Rapidly Exploring Random Tree algorithm

This kind of algorithm is quick and suitable for high-dimensional problems while the tree will explore the configuration space randomly. This means that this algorithm may find the goal just as quick in a large configuration space as in a smaller configuration space. The increment value that is used (so in figure 4 the distance between two dots) has a great influence on the working of the algorithm. When the increment value is too large, the search will be quick, but obstacles may cause the algorithm to fail. When this value is too small, it will take longer to cover a certain area.

The downside of this algorithm that the chance of finding the best path is very small. This is already clear in figure 4. There are no obstacles between the starting point and the goal, so the optimal path would be a straight line. In complex environments, the path may become even less efficient. For example, when a path could have been planned between two obstacles, the tree may only have found a path around the obstacles.

[*Rodriguez, S., Xinyu Tang, Jyh-Ming Lien, Nancy M. Amato, An Obstacle-Based Rapidly-Exploring Random Tree*]

Requirements	**Rating**
Collision-free path	Yes, if a path is found
Implementation difficulty	Easy/**Medium**/Hard
3 DoF functionality	Yes
Other criteria	
Path finding speed in 3D environment	Slow/Medium/**Fast,** depending on step size
Shortest path	Almost never
Smooth flight	Almost never

Table 3.6: Rapidly exploring random tree

Rapidly exploring random tree algorithm is probably the quickest way to generate a path to the target. It works very well in 2D as well as in 3D environments.

The downside is that this algorithm will not always find a path, even if one is available. This is because the expansion of this algorithm is random, and when all branches hit a dead end, the algorithm will conclude that no path is available. When a path is found, it will practically never be the shortest/quickest path and also will not provide a smooth path. It also doesn't seem easy to implement.

2.7 Comparison

In table 3.7 on the next page a total scoring is given for all algorithms. The scores used for the requirements are 0, 2 or 6 points. The scores used for the other criteria are 0, 1 and 3 points, since these criteria are less important than the requirements. Again, please note that the scoring for 'Implementation difficulty' is somewhat subjective.

2.8 Decision

From Table 3.7 is concluded that A-star is the best algorithm for our purposes. It guarantees a collision-free path, can be made to work in 3D environments and the probability of a working algorithm before the deadline is high. It seems to be quick enough for use in this challenge, will always find the shortest path and also provide a smooth flight.

	Dijkstra's shortest path	Best-first search	A-star	Potential fields	Harmonic Potential fields	Random tree search
Collision-free path	*6*	*6*	*6*	*2*	*6*	*2*
Implementation difficulty	*6*	*6*	*6*	*0*	*0*	*2*
3 DoF functionality	*0*	*2*	*2*	*6*	*6*	*6*
Path finding speed in 3D environment	*0*	*1*	*1*	*2*	*2*	*2*
Shortest path	*2*	*1*	*2*	*1*	*2*	*0*
Smooth flight	*2*	*1*	*2*	*1*	*2*	*0*
Total score	**16**	**17**	**19**	**12**	**18**	**12**

Table 3.7: Scoring table for comparing algorithms based on requirements

3. Design and implementation

This chapter will show the process of developing the path planning algorithm with obstacle avoidance. An algorithm based on A-star is designed and implemented. Finally, improvements are made, leading to the final version of the algorithm. The full implementation of the algorithm is found in appendix B.

3.1 Design

For designing the algorithm, a pseudo code implementation is made based on research done on A-star. A flowchart of this pseudo code is found in figure 3.1. The pseudo code is found in appendix A.

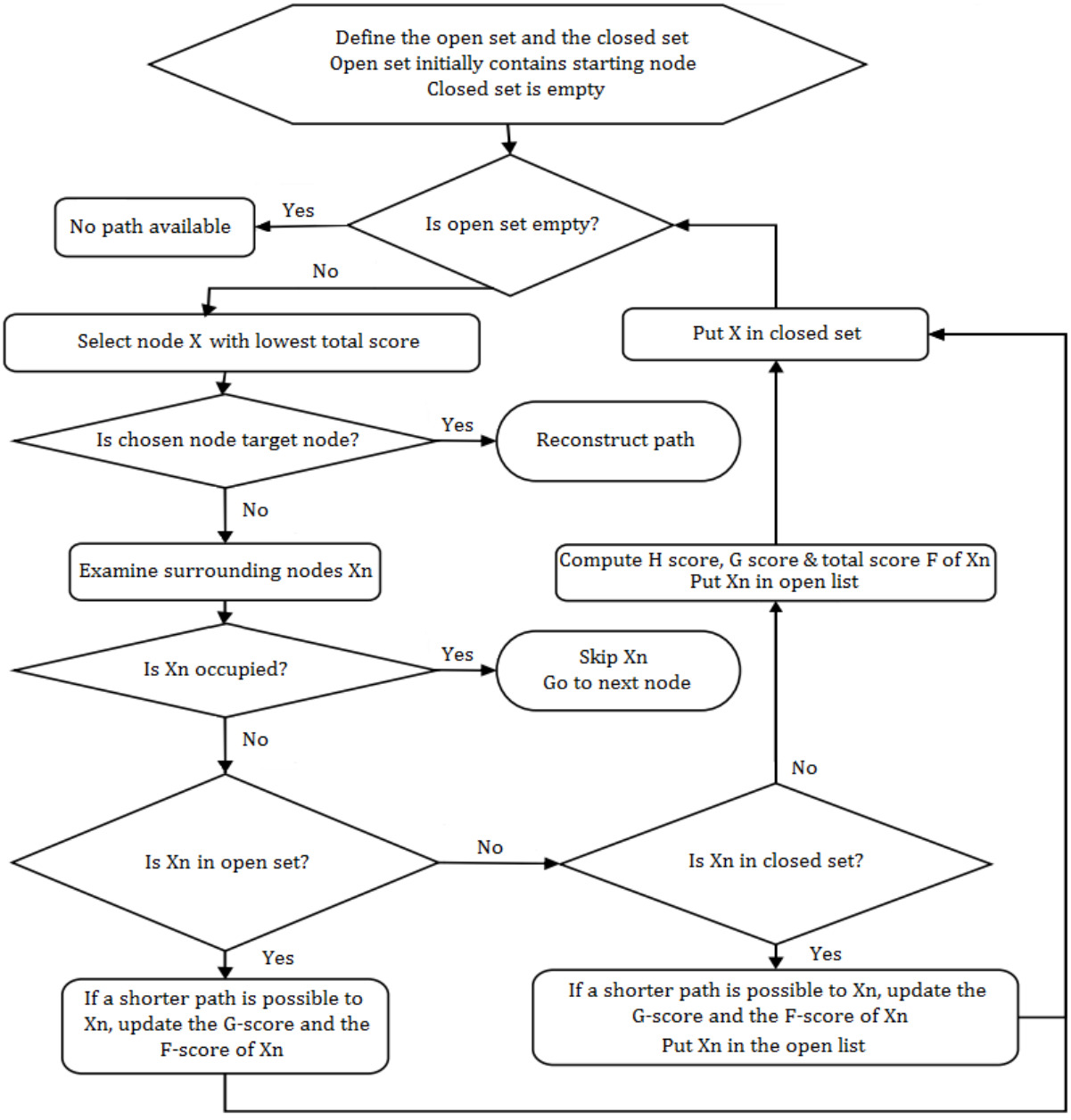

Figure 3.1: Flowchart A-star algorithm

3.2 Implementation

This section explains the implementation of the final algorithm. First, an implementation is made based on the flowchart found in figure 3.1. Due to problems with this implementation, improvements are made. These are described and added to the flowchart.

3.2.1 Basic A-star implementation

Refer figure 3.1. On initiation, the open set and the closed set are defined. The open set contains the nodes that are examined at a certain time by the algorithm, excluding nodes that belong to an obstacle. The closed set contains all nodes that have been examined already by the algorithm. Initially, the open set contains the starting node and the closed set is empty.

As long as the open set is not empty, the open set is checked for the node with the lowest F score, let's call this node X. Initially, the open set only contains the starting node and thus this becomes node X. If the node with the lowest F score is the target node, the algorithm stops and the path is reconstructed. Else the neighboring nodes of X, let's call them Xn, are each examined, refer figure 3.2. The numbers represent the distance to node X of each node Xn. First, each node Xn is checked for occupancy.

If Xn is occupied, this node is skipped and the next node Xn is examined. An occupied node can never be part of the final path.

If a neighboring node has not yet been examined before, the F-score of this node has to be calculated. The G-score of this node is calculated by: $G_{Xn}=G_X$ + *distance to X*. The H-score of each node Xn is calculated by using the Manhattan distance to the goal, refer section 2.2. This is the easiest method to implement, and is sufficient to meet the requirements stated in chapter 2.

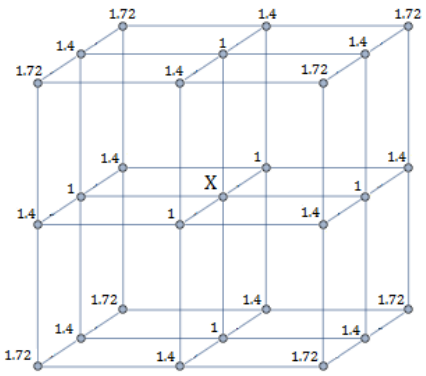

Figure 2.2: Nodes examined around node X. The numbers represent the distance to node X of each node.

It is also possible that a neighboring node Xn has already been examined before, but a shorter path to this node is found. This is true if the G-score of that node Xn that has just been calculated is lower than the G-score that was calculated on previous examination. In this case, the G-score and the F-score of this node Xn are updated and the node is put in the open set.

If all nodes Xn have been given a score and are put in the open set, node X is put in the closed set. This way the algorithm will know that this node already has been the center node X. After this, the open list is again checked for nodes and the process described above is repeated. If a path is found, each node contributing to the path is sent to the controller one by one. If the MAV gets within a distance of 0.1 m to the current presented node, the next node is presented. This way, the MAV will remain a constant speed during flight.

When testing this algorithm, it became clear that there were two issues. The first issue is seen in figure 3.3a. The path is planned alongside the walls, which would result in a collision of the MAV. By testing whether the error came from the algorithm or the Octomap, it was found out that the nodes queried for occupancy in the octomap are not in the center of a block, but at one of the corners, refer 3.3b. This means that for instance when the green node is queried, the Octomap will return unoccupied because the block belongs to the red node. Therefore, if a node is said to be unoccupied, there is a chance that this node is on the edge of an occupied node. In the simulation this will result in a collision. Section 3.2.2 provides a solution to the wall hugging issue.

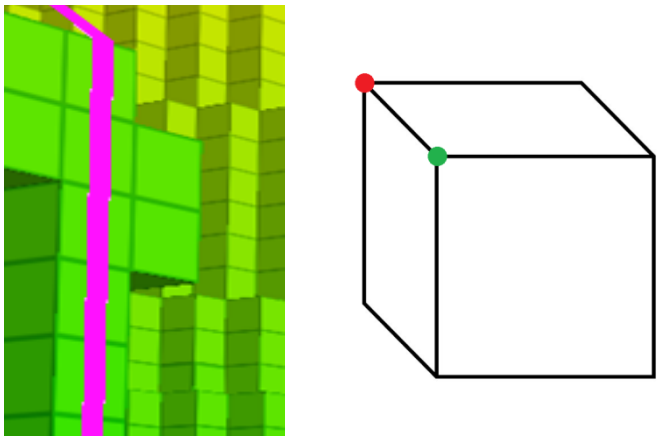

Figure 3.3 a) Wall hugging issue b) Node position in Octomap

The second issue was that the algorithm sometimes would plan an illogical path where a straight line would be sufficient, refer figure 3.4. The red line is a 2D example of what such an illogical path might look like. The green line shows the optimal path between point A and point B. This only occurs when a path is planned close to an obstacle. The cause of this is unknown.

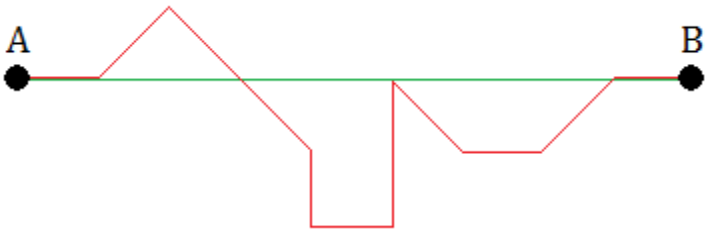

Figure 3.4: 2D example of an illogical path between point A and B. The green line is the optimal path.

3.2.2 Wall hugging issue

As stated in section 3.4, the algorithm would plan a path that has virtually no distance from the obstacles which would definitely result in a collision. In order to solve this, the 3x3x3 matrix is resized to 7x7x7. This adds two nodes in all directions that is evaluated. For every next node this 7x7x7-matrix is formed and the occupancy of all surrounding nodes is queried and stored.

For choosing the next node, the original 3x3x3 matrix is still evaluated but now one extra step is added. A 5x5x5-matrix around every node Xn of the 3x3x3-matrix is evaluated, refer figure 3.5 for a 2D example. The current center node is X, the 3x3x3-matrix is formed by nodes Xn and the 5x5x5 matrix used for the extra step is formed by nodes B. For a node Xn to be stored as a possible successor, all the B-nodes around this node Xn and of course the Xn-node itself should be unoccupied. This assures that the is at least a distance of two nodes between the MAV and an obstacle.

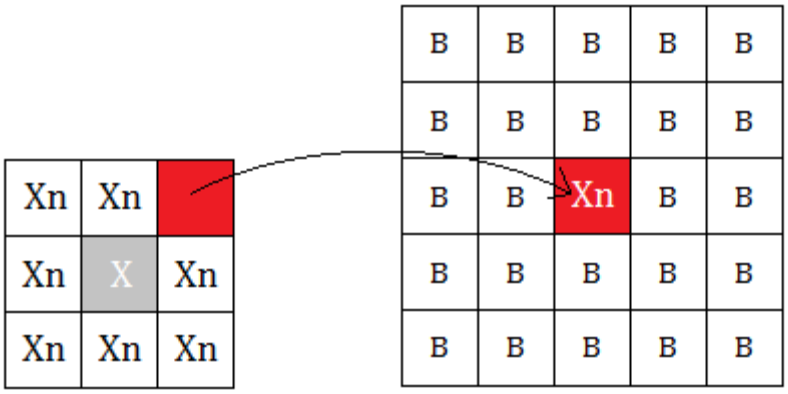

Figure 3.5: Node checking procedure

This solution was also tested with a 5x5x5-matrix and even though this also solved the wall hugging issue, this would not leave a big margin of error. A 7x7x7-matrix provides a safety margin.

3.2.3 Illogical path issue

The cause of this issue has not been found. Since this issue does not cause any significant problems and does not occur every time, the choice was made to leave this issue as it is and focus on optimizing the algorithm for better results.

3.2.4 Removing straight-line nodes

Since a PID-controller is used for the control, it can be assumed that when waypoint is received by the controller, it will fly the MAV to that waypoint in a straight line. This means that if there is a straight line in the path, all the nodes between the beginning and the end of this line can be discarded and only the end-node has to be sent to the controller. After the path is created, the path is checked for straight lines and in-between nodes are removed if possible.

Since the output of a PID-controller depends on the error presented at the input, this will increase the speed of the MAV along these straight parts of the path, and thus the MAV will reach its goal significantly faster, refer figure 3.6.

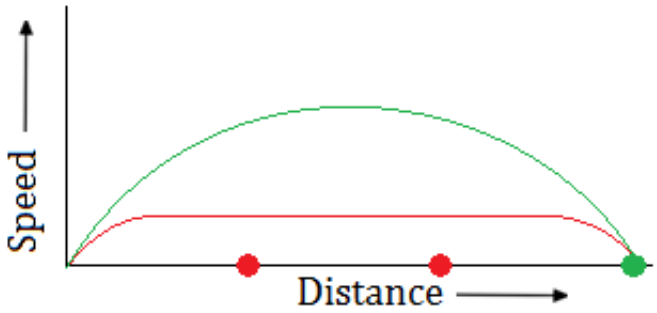

Figure 3.6: Speed of the MAV with (green line) and without removing straight line nodes (red line)

3.2.5 Optimizing the path

As stated before, the algorithm used has only a limited choice of directions, that is only straight or diagonal. Therefore the path is often not as direct as it could be. Especially in tasks 4.1 and 4.2, where the optimal path is always a straight line. Instead of a straight line, a less optimal path like the one in figure 2.1 is often created.

In order to optimize this, there is a function available within the Octomap. This function draws a straight line between a given start- and end point and will check whether the line is interrupted by an obstacle or not. However, this function only checks for occupancy along this line. The dimensions of the MAV are not taken into account. Therefore, this function is only used to determine whether a shorter path is available between the starting point and the target node. If no obstacles are found, all in-between nodes are discarded and only the target node is presented to the controller.

For tasks 4.1 and 4.2 this will always result in just the target node being published to the controller since there are no obstacles present. This decreases the time in which tasks 4.1 and 4.2 are completed, and might result in a better score due to lower settling time.

4. Evaluation

In this chapter the results of task 4 will be discussed. The last section will show the overall results of team LEO in the European Robotics challenge.

4.1 Evaluation

By using the 7x7x7-matrix, the wall hugging issue is permanently solved. The time needed for the algorithm to find a path has increased due to the extra nodes that have to be processed. Though the total settling time of the system, i.e. from receiving a target waypoint to reaching the target waypoint, is still within the benchmarks of EuRoC and thus the final scoring for settling time has not changed.

Concerning the issue of generating an illogical sub-optimal path, this issue still remains since the cause has not been found. This issue has not caused big problems and did not have significant influence on performance of the algorithm and thus the focus was on optimizing the algorithm for better results.

By checking the path for straight lines and removing all the nodes between the beginning and the end of this line, the MAV reaches its destination quicker and the flight is smoother as expected. The controller is handling this very well and this has surely improved the algorithm.

By immediately checking if a straight line is possible between the start and the goal, the settling time in tasks 4.1 and 4.2 has decreased as expected. The algorithm just publishes the target waypoint, the controller makes sure that the MAV gets to this target in a straight line and settles again when the target is reached.

Since the function used within the Octomap to optimize the path only checks the nodes along a line (so 1D), it is not safe to assume that the optimized path is obstacle-free. The function can plan a new path too close past obstacles without this being checked since there is no margin, this will result in a collision. Therefore, this function is only used once to check if a straight line is possible between start and target. Though this has not shown any issues, theoretically the chance of a collision is still there. Looking for another solution is advised.

An additional issue was also found in the algorithm, namely that not all waypoints generated by the path planner were received by the controller. Finding the source of this issue has taken some time, but eventually this was solved by adding a small delay between publishing the waypoints to the event handler. This was thanks to Geert Folkertsma, who came up with this solution.

4.2 EuRoC results

4.2.1 Task 4 results

In table 5.1 the results of the final algorithm are shown. The left column shows the results of the tests performed on our own PC's, the right column shows the results received from EuRoC.

As stated before, the algorithm is tested in a 3D environment along with the controller implemented in task 3. So both systems have to work well together in order to get good results. The algorithm is tested by EuRoC in within the same Octomap as was provided by EuRoC, so testing the algorithm on our own PC's should give a good idea of how well the algorithm performs.

	Results on own PC	Results from EuRoC
***Task 4.1**: Waypoint navigation (no obstacles)*		
Accuracy	1.5/2	0.5/2
Settling time	1.5/2	1.5/2
Energy-efficiency	1.5/2	0.5/2
***Task 4.2**: Sensor failure (no obstacles)*		
Accuracy	2/3	2/3
Settling time	2/3	2/3
Energy-efficiency	2/3	2/3
***Task 4.3:** Navigation with obstacle avoidance*		
Accuracy	3.5/5	0/15 (system did not settle)
Settling time	1.5/5	
Energy-efficiency	3.5/5	

Table 5.1: Results final algorithm

As can be seen in table 5.1 the results received from EuRoC are quite different than the results that were obtained on our own PC's. It was already discovered that the results of the simulation may differ depending on how fast the used PC is. Assuming that the algorithm was tested by EuRoC on a faster PC than the ones we used, this might explain the difference in some of the results.

Also, the system did not settle is task 4.3 when evaluated by EuRoC. No collision was reported by EuRoC, but one explanation might be that the algorithm failed to find a path to the goal. Though this has never happened when testing on our own PC's.

4.2.2 Team LEO results

In order to keep track on where each task stands, a status table was set up in which intermediate testing scores were posted. The last known results before handing in the complete solution for every task can be found in table 5.2. In table 5.3 the results of the European Robotics Challenge for team LEO can be found.

Perception		**Control**	
Task 1: localization	**Task 2: Mapping**	**Task 3: Hovering**	**Task 4: Navigation**
1.1: 5/8	*2.1:* 5/8	*3.1:* 6/9	*4.1:* 4.5/6
1.2: 6.5/10	*2.2:* 5/10	*3.2:* 3/4.5	*4.2:* 6/9
1.3: 5/8	*2.3:* 5/8	*3.3:* 5/9	*4.3:* 8.5/15

Table 5.2: Last known intermediate test results for each task

Perception		Control	
Task 1: localization	**Task 2: Mapping**	**Task 3: Hovering**	**Task 4: Navigation**
1.1: 3/8	*2.1:* 3/8	*3.1:* 4.5/9	*4.1:* 2/6
1.2: 3.5/10	*2.2:* 3.5/10	*3.2:* 1.5/4.5	*4.2:* 6/9
1.3: 3/8	*2.3:* 3/8	*3.3:* 4/9	*4.3:* 0/15

Table 5.3: Team LEO results European Robotics Challenge

The results on our own PC's were obviously better than the results received from EuRoC. Despite that, these results were good enough for rank 12 out of 35 in the EuRoC challenge which of course is a great achievement. The fifteen best teams proceeded to the next round, and thus team LEO as well.

5. Conclusions and recommendations

5.1 Conclusions

The goal of this assignment was to ultimately implement a working algorithm for navigation with obstacle avoidance. Choosing A-star as a base has proven to be a good choice. After the changes and optimizations described in chapter 3, the algorithm has always provided a collision-free path in a satisfactory short time. Due to the creation of an illogical path at some points, the path created has not always been the smoothest and shortest, but this did not result in a significant decrease in performance. The results from our own pc's show that the goal of this assignment is met, but optimization in terms of robustness is advised.

The issue of creating an illogical path has not been solved during the project. Together with the result from EuRoC where the system did not settle, this shows that the algorithm is not entirely bug-free.

5.2 Recommendations

For future research and development, some improvements are suggested.

1. Adding a local potential field as an extra safeguard while flying is suggested to decrease the risk of collision even further.
2. The function used within the Octomap to optimize the path only checks the nodes along a line (so 1D), it is not safe to assume that the optimized path is obstacle-free. The function can plan a new path too close past obstacles without this being checked since there is no margin, this will result in a collision. Looking for another solution is advised.
3. It is suggested to check the algorithm further for robustness. Illogical paths together with the system not settling when tested by EuRoC show that there are still some issues that need some work.
4. For 3D-environments, one might want to look at harmonic potential fields. This algorithm takes more effort to implement due to the mathematics involved, but if this algorithm is implemented correctly, it might prove to perform better than A-star. Especially in a 3D environment.

Appendix A: Pseudo code for A-star

```
function A*(start, goal)
  closedSet := {}                                   // The set of nodes already evaluated.
  openSet := {start}        //The set of currently discovered nodes still to be evaluated. Initially, only
                              the start node is known.
  cameFrom := the empty map //for back tracking the path
  gScore := map with default value of Infinity      // For each node, the cost of getting from the start
                                                       node to that node.
  gScore[start] := 0 // The cost of going from start to start is zero.
  fScore := map with default value of Infinity      // For each node, the total cost of getting from the
                                                       start node to the goal.
  fScore[start] := heuristic_cost_estimate(start, goal)  // For the first node, that value is completely
                                                                                          heuristic.
  while openSet is not empty{                       // If there are still nodes in the open set
    current := the node in openSet having the lowest fScore[] value
    if current = goal {
      return reconstruct_path(cameFrom, current)
      end algorithm
    }
    openSet.Remove(current)
    closedSet.Add(current)
    for each neighbor of current {
      if neighbor in closedSet {
        if currently calculated neighbor F score is lower than saved neighbor F score {
          Recompute H score, G score and F score
          Put neighbor in open set
        }
      }
    tentative_gScore := gScore[current] + dist_between(current, neighbor)  // The distance from
                                                                              start to a neighbor
```

```
        if neighbor not in openSet   {
            openSet.Add(neighbor)                                   // Discover a new node
         } else if tentative_F-score >= F-score[neighbor] {
            continue                                                // This is not a better path.
        }
         // This path is the best until now. Record it!
         cameFrom[neighbor] := current
         gScore[neighbor] := tentative_gScore
         fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)
     }
  }
   return failure

function reconstruct_path(cameFrom, current) {
   total_path := [current]
   while current in cameFrom.Keys:  {
      current := cameFrom[current]
      total_path.append(current)
  }
 return total_path
}
```

Appendix B: Implementation final algorithm

```
#include "euroc_track_2_nodes/euroc_path_planner.h"

#include <string>
#include <iostream>
#include <algorithm>

#include <visualization_msgs/Marker.h>

#define DEBUG 1
/*
 * Coordinate system: double coordinate[3] = {x, y, z};
 *
 * field is a 7x7x7 matrix that keeps track of the occupancy probability of
the nodes surrounding the current position
 * field[1][2][3]
 * TODO: update values
 * 1: x-axis (-0.5, -0.25, 0, +0.25 + 0.5)
 * 2: y-axis (see x-axis)
 * 3: z-axis (see x-axis)
 * Where numbers in () represent the difference with the center of the field.
The difference is 0.25 meters, because this is the resolution of the
 * octomap, thus each node is 0.25 meters apart.
 */

// Constructor.
Pathplanner::Pathplanner(std::string path)
  :tree(path)
{

}

// Destructor.
Pathplanner::~Pathplanner()
{

}

// Initialize variables.
bool Pathplanner::init(std::string pub_topic)
{
  ROS_INFO("Initializing pathplanner");

  ros::NodeHandle node_handle;

  std::string file_path;

  waypoint_pub = node_handle.advertise<mav_msgs::CommandTrajectory>(pub_topic, 10);
  marker_pub = node_handle.advertise<visualization_msgs::Marker>("marker", 10);

  // Set the old goal (init position).
  old_goal.x = 0;
  old_goal.y = 0;
  old_goal.z = 1;
```

```
54
55      // Set dif to 0.
56      old_goal.dif_x = 0;
57      old_goal.dif_y = 0;
58      old_goal.dif_z = 0;
59
60      // An array to help with shifting the field. This array contains the difference
61      //between nodes as described at the start of this file.
62      dif[0] = -0.25;
63      dif[1] = 0.0;
64      dif[2] = 0.25;
65
66      step[0] = -0.75;
67      step[1] = -0.5;
68      step[2] = -0.25;
69      step[3] = 0.0;
70      step[4] = 0.25;
71      step[5] = 0.5;
72      step[6] = 0.75;
73
74      ROS_INFO("euroc_path_planner initialized");
75      return true;
76  }
77
78  // Handle incoming waypoints.
79  void Pathplanner::wpCallback(const mav_msgs::CommandTrajectoryConstPtr &wp)
80  {
81      Node start, goal;
82
83      // New path, so clear list.
84      closed_list.clear();
85      illegal_list.clear();
86
87      // Get goal position.
88      goal.x = wp->position[0];
89      goal.y = wp->position[1];
90      goal.z = wp->position[2];
91
92      // Start planning from the previous goal.
93      start = old_goal;
94      start.g = 0;
95      start.f = 0;
96
```

```
97    // Plan initial path.
98    bool success = planPath(start, goal, closed_list);
99    ROS_INFO("Generated path from [%f|%f|%f] to [%f|%f|%f]. Total nodes traversed
100   (start and end included): %d \tCost: %f", start.x, start.y, start.z,
101   closed_list.back().x, closed_list.back().y, closed_list.back().z, closed_list.size(),
102   closed_list.back().g);
103
104   if(success)
105   {
106     int old_size = closed_list.size();
107
108     // Optimize path. Cost may be incorrect.
109     optimizePath();
110     ROS_INFO("Path has been optimized, %d nodes were removed.
111     New path consists of %d nodes. Total cost: %f", old_size - closed_list.size(),
112     closed_list.size(), closed_list.back().g);
113
114     old_size = closed_list.size();
115
116     // Clean path
117     cleanPath(closed_list);
118     ROS_INFO("Path cleaned, %d nodes were removed. New path consists of %d nodes",
119     old_size - closed_list.size(), closed_list.size());
120
121     // old_goal is set so that it will function as new start when a new waypoint is
122     //received.
123     old_goal = goal;
124
125     // Set dif to 0.
126     old_goal.dif_x = 0;
127     old_goal.dif_y = 0;
128     old_goal.dif_z = 0;
129
130
131     ROS_INFO("Publishing path.");
132     publishWaypoints();
133   }
134
135   // Draw path for rviz visualization. Only for debug.
136   if(DEBUG)
137     drawPath(closed_list);
138 }
139
140 bool Pathplanner::planPath(Node start, Node& goal, std::vector<Node>& path)
141 {
142   double dx, dy, dz;
143   Node q;
144
145   // Calculate difference between goal and start. Has to be positive, hence the fabs().
146   dx = fabs(goal.x - start.x);
147   dy = fabs(goal.y - start.y);
148   dz = fabs(goal.z - start.z);
149
150   // Set h, g and f. f and g are 0 because this is the first node.
151   start.h = dx + dy + dz;
152
153   // Push the start node to the open list.
154   open_list.push_back(start);
155
156   // End not reached.
157   bool end = false;
158
159   // Initialize the occupancy field with the start node as center.
160   initField(start);
```

```
161
162   // While there are successors available and the end has not been reached.
163   // TODO: Fix deadlock that occurs when there are no successors but the end has not
164   //been reached.
165   while(!end)
166   {
167     if(!open_list.empty())
168     {
169       // Find the successor with the lowest f value.
170       q = findNextNode();
171
172       // Adjust the occupancy field to the new center.
173       initField(q);
174
175       // Generate the successor node.
176       end = generateSuccessors(q, goal, path);
177
178       // Push q to the closed list.
179       path.push_back(q);
180
181       // If the goal is reached push the goal to the closed list.
182       if(end)
183         path.push_back(goal);
184     }
185     else
186     {
187      // TODO: Solve path
188      path.pop_back();
189      end = generateSuccessors(path.back(), goal, path);
190      ROS_INFO("No successors could be created. Stepping back one node");
191 //     ROS_INFO("No path found. Check rviz for path visualization");
192 //     return false;
193     }
194   }
195   return true;
196 }
197
198 void Pathplanner::optimizePath()
199 {
200   std::vector<Node>::reverse_iterator it;
201   std::vector<Node>::iterator cur_it;
202   std::vector<Node> path;
203
204   Node cur, goal;
205   double old_goal_g;
206
207   cur_it = closed_list.begin();
208   cur = *cur_it;
209
210   while(cur != closed_list.back())
211   {
212     for(it = closed_list.rbegin(); it != closed_list.rend(); it++)
213     {
214       goal = *it;
215
216       // No faster path possible, go to the next node in the closed list.
217       if(goal == cur)
218       {
219         Node new_cur = *(it - 1);
220 //        new_cur.g = new_cur.g - old_goal_g + cur.g;
221         cur = new_cur;
222         break;
223       }
224
```

```
225         // Limit for castRay() (otherwise the ray will continue to the edge of the map).
226         double distance = sqrt(pow(goal.x - cur.x, 2) + pow(goal.y - cur.y, 2)
227         + pow(goal.z - cur.z, 2));
228
229         // castRay() requires the usage of octomap::point3d.
230         octomap::point3d start_point(cur.x, cur.y, cur.z), goal_point(goal.x, goal.y,
231         goal.z), obstacle(0, 0, 0);
232
233         // Check if a direct path between current and goal exists.
234         if(!tree.castRay(start_point, goal_point, obstacle, false, distance))
235         {
236           // castRay() also returns false (no occupied node hit) on unknown space.
237           //Check if unknown space was hit.
238           if(tree.search(obstacle) != NULL)
239           {
240             old_goal_g = goal.g;
241
242             // Plan a new path between current and goal.
243             planPath(cur, goal, path);
244
245             // Remove goal from closed list, otherwise it will result in a duplicate
246             //node in closed_list (with incorrect cost).
247             closed_list.erase(--(it.base()));
248
249             // Remove old path between cur and goal and insert the new path.
250             closed_list.erase(cur_it, --(it.base()));
251             closed_list.insert(cur_it, path.begin(), path.end());
252
253             cur = goal;
254
255             break;
256           }
257         }
258       }
259     }
260   }
261
262   void Pathplanner::cleanPath(std::vector<Node> &path)
263   {
264     std::vector<Node>::iterator it;
265     Node prev, cur, next;
266
267     for(it = path.begin(); it != path.end(); )
268     {
269       cur = *it;
270       // Skip the first node and last node, since you will be checking one node before
271       //(it - 1) and after (it + 1) the current position (it).
272       if(it != path.begin() && it != path.end())
273       {
274         next = *(it + 1);
275         // Check if the difs (x, y, z) of the nodes are equal, if they are, remove the
276         //current node (it) from the closed list and set
277         // the parent of the next node (it + 1) to the previous node (it - 1)
278         if(prev.dif_x == cur.dif_x && prev.dif_x == next.dif_x &&
279            prev.dif_y == cur.dif_y && prev.dif_y == next.dif_y &&
280            prev.dif_z == cur.dif_z && prev.dif_z == next.dif_z)
281         {
282           next.parent = &prev;
283           path.erase(it - 1);
```

```
284         }
285         else
286           it++;
287       }
288       else
289         it++;
290
291       prev = cur;
292     }
293   }
294
295   bool Pathplanner::generateSuccessors(Node parent, Node& goal, std::vector<Node> path)
296   {
297     // Before generating new successors, clear the open list (removing the successors of
298     //the previous q).
299     open_list.clear();
300
301     for(int x = 0; x < 3; x++)
302     {
303       for(int y = 0; y < 3; y++)
304       {
305         for(int z = 0; z < 3; z++)
306         {
307           // Check if the successor node is occupied.
308           if(field[x][y][z] <= 0.1192 && !checkNeighbours(parent, x, y, z))
309           {
310             // Create successor and set the fields.
311             Node successor;
312
313             successor.parent = &parent;
314
315             successor.x = parent.x + dif[x];
316             successor.y = parent.y + dif[y];
317             successor.z = parent.z + dif[z];
318
319             // Check if the node has already been created or used before moving on.
320             //Successor can't be equal to parent.
321             if(!nodeAlreadyUsed(successor, path) && successor != parent)
322             {
323               successor.h = fabs(goal.x - successor.x) + fabs(goal.y - successor.y)
324                             + fabs(goal.z - successor.z);
325               successor.g = sqrt(fabs(dif[x]) + fabs(dif[y]) + fabs(dif[z])) + parent.g;
326               successor.f = successor.h + successor.g;
327
328               successor.dif_x = x;
329               successor.dif_y = y;
330               successor.dif_z = z;
331
332               if(successor == goal)
333               {
334                 goal = successor;
335                 return true;
336               }
337
338               open_list.push_back(successor);
339             }
340           }
341         }
342       }
343     }
344     return false;
345   }
```

```
346
347  Pathplanner::Node Pathplanner::findNextNode()
348  {
349    Node smallest;
350    std::vector<Pathplanner::Node>::iterator it;
351
352    it = std::min_element(open_list.begin(), open_list.end(), smallest);
353    smallest = *it;
354
355    open_list.erase(it);
356
357    return smallest;
358  }
359
360  bool Pathplanner::nodeAlreadyUsed(const Node& node, std::vector<Node> path)
361  {
362    if(std::find_if(open_list.begin(), open_list.end(), node) != open_list.end())
363      return true;
364    else if(std::find_if(illegal_list.begin(), illegal_list.end(), node) !=
365          illegal_list.end())
366      return true;
367    else if(std::find_if(path.begin(), path.end(), node) != path.end())
368      return true;
369    else
370      return false;
371  }
372
373  void Pathplanner::publishWaypoints()
374  {
375    std::vector<Node>::iterator it;
376    mav_msgs::CommandTrajectory waypoint;
377
378    for(it = closed_list.begin(); it != closed_list.end(); it++)
379    {
380      Node node = *it;
381
382      waypoint.position[0] = node.x;
383      waypoint.position[1] = node.y;
384      waypoint.position[2] = node.z;
385
386      waypoint_pub.publish(waypoint);
387    }
388  }
389
390  void Pathplanner::printField(const Node& center)
391  {
392    for(int x = 0; x < 7; x++)
393    {
394      for(int y = 0; y < 7; y++)
395      {
396        for(int z = 0; z < 7; z++)
397        {
398          std::cout << "(" << center.x + step[x] << "|" << center.y + step[y] << "|" <<
399          center.z + step[z] << "|" << field[x][y][z] << ") ";
400        }
401      std::cout << std::endl;
402      }
403    std::cout << std:: endl;
404    }
405  }
406
```

```
407  void Pathplanner::drawPath(std::vector<Node> path)
408  {
409    visualization_msgs::Marker line;
410
411    line.header.frame_id = "map";
412    line.header.stamp = ros::Time::now();
413    line.ns = "line";
414    line.action = visualization_msgs::Marker::ADD;
415    line.pose.orientation.w = 1.0;
416
417    line.pose.position.x = 0;
418    line.pose.position.y = 0;
419    line.pose.position.z = 0;
420
421    line.id = 0;
422    line.type = visualization_msgs::Marker::LINE_STRIP;
423    line.scale.x = 0.05;
424
425    line.color.r = 1.0;
426    line.color.a = 1.0;
427
428    line.lifetime = ros::Duration(0);
429
430    std::vector<Node>::iterator it;
431    for(it = path.begin(); it != path.end(); it++)
432    {
433      geometry_msgs::Point p;
434      Node n = *it;
435      p.x = n.x;
436      p.y = n.y;
437      p.z = n.z;
438
439      line.points.push_back(p);
440    }
441    marker_pub.publish(line);
442  }
443
444  void Pathplanner::initField(const Node& center)
445  {
446    for(int x = 0; x < 7; x++)
447      for(int y = 0; y < 7; y++)
448        for(int z = 0; z < 7; z++)
449          field[x][y][z] = tree.search(center.x + step[x], center.y + step[y],
450                                       center.z + step[z])->getOccupancy();
451  }
452
453  bool Pathplanner::checkNeighbours(const Node center, int step_x, int step_y, int step_z)
454  {
455    for(int x = step_x; x < step_x + 5; x++)
456      for(int y = step_y; y < step_y + 5; y++)
457        for(int z = step_z; z < step_z + 5; z++)
458          // Make sure the coordinates are in the field (between 0 and 6).
459          if(x > -1 && y > -1 && z > -1 && x < 7 && y < 7 && z < 7)
460            // If the field entry has higher occupancy probability then the minimum the
461            //node should be considered occupied.
462            if(field[x][y][z] > 0.1192)
463              return true;
464
465    // This point is only reached if none of the neighbours was occupied.
466    return false;
467  }
468
```

Appendix C: Other experiments

C.1.1 First solution to direct path planning

As can be seen from table 1.1, subtasks 4.1 and 4.2 are about waypoint navigation without obstacle avoidance. Basically, this means going from point A to point B as straight as possible.. The algorithm treated this chapter has not been used in the final solution, since the final algorithm used for obstacle avoidance was also capable of planning a direct path to a given waypoint.

In order to make sure the algorithm will complete subtasks 4.1 and 4.2, a set of demands is determined. The algorithm has to:

- Determine the direction of the target waypoint
- Find the best path towards the target waypoint
- Deliver the right information to the controller

The first goal was to have a working algorithm that plans a path to a target and delivers the right information to the controller. When this base algorithm is working, improvements can be made.

The first design is based on a 3D-grid. Only discrete steps can be made within this grid, the step size can be set with one parameter. The step size was initially set to 1m. The pseudo code of this algorithm can be found at the end of this appendix.

The algorithm will check the retrieve data about position of the MAV from the simulation tool itself as a (x,y,z)-vector. Next, the coordinates of the target are retrieved in the same format and compared to the current position of the MAV. As long as the goal is not reached, a new subwaypoint is created and stored in the array that will contain the final path.

The coordinates of this subwaypoint depend on whether the difference between the current position of the MAV and the goal in the x-, y- and z-direction is zero or not. If the x-value of the previous waypoint is not equal to the goal's x-coordinate, the x-coordinate of the previous subwaypoint is incremented or decremented by the step size. Same goes for the y-coordinate and the z-coordinate.

When the goal is reached, that is when the created subwaypoint has the same coordinates as the target waypoint, the path is published to the controller. The subwaypoints will be published one at a time. When the MAV gets within 0.25m of the published waypoint, this waypoint is discarded and the next waypoint is published, until the target waypoint is reached.

A resulting path created by this algorithm is shown in figure 2.1. The blue line shows the path the MAV will follow. Each intersection with a red line represents the coordinate of a sub-waypoint. The starting position of the MAV is the coordinate given by the vector (1,1,2). The coordinates of the target is given by the vector (9,6,4).

The sub-waypoints that were created by the algorithm and represented by the red and blue line intersections are respectively (2,2,3), (3,3,4), (4,4,4), (5,5,4), (6,6,4), (7,6,4), (8,6,4) and (9,6,4).

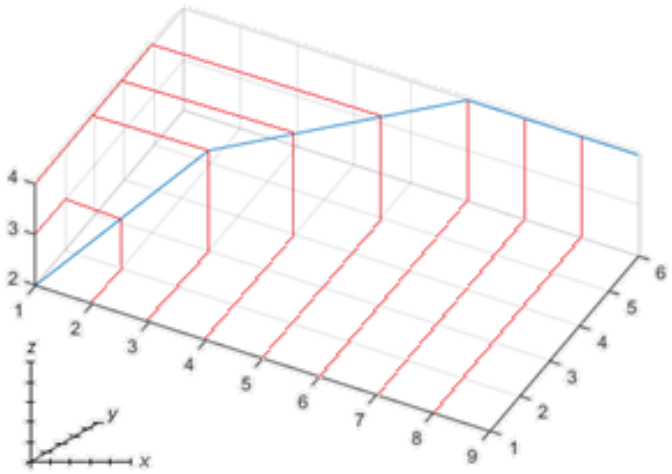

Figure 3.1: Resulting path of the direct path planning algorithm

The reason why this path is not a straight line (which would be the optimal path) is that the idea was to combine this algorithm with the algorithm used for obstacle avoidance. An Octomap of a power plant was provided by EuRoC, and because it wasn't sure yet whether the Octomap should be queried discretely (so by respecting the resolution) or if every possible coordinate could be queried for occupation. Therefore, this way was chosen as a safe base to start with. This algorithm would compute a path, and every sub-waypoint would be checked for any occupation. If a sub-waypoint is occupied, in other words an obstacle is found, the obstacle avoidance algorithm would take over.

C.1.2 Direct path planning pseudo code

Check position: while goal is not reached {

If(difference between previous x and goal x is not 0){

New x=Increment or decrement previous x by step size

}
If(difference between previous y and goal y is not 0){

New y=Increment or decrement previous y by step size

}
If(difference between previous z and goal z is not 0){

New z=Increment or decrement previous z by step size

}
Store subwaypoint (New x, New y, New z)
}

Publish path

C.1.3 Direct path planning algorithm

```
#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

struct Chkpts {
    double x;
    double y;
    double z;
    double seq;
    double yaw;
} check;

int main () {

    double x_pos, y_pos, z_pos, x_goal, y_goal, z_goal;
    cout<<"It is only allowed to fill in coordinates equal to or bigger than 0 0 0."<<endl;
    cout<<"Enter the starting coordinates x y z: ";

    // //starting position can be retrieved from the sensors/localization
    cin>>x_pos>>y_pos>>z_pos;
    cout << "X-start: " << x_pos << endl << "Y-start: " << y_pos << endl <<
    "Z-start: " << z_pos << endl;

    // Listen to firefly/waypoint (mav_msgs/ControlTrajectory) and use the published waypoint
    cout << "Enter goal coordinates x y z: ";

     cin>>x_goal>>y_goal>>z_goal;
     cout << "X-goal: " << x_goal << endl << "Y-goal: " << y_goal << endl << "Z-goal: "
     << z_goal << endl;

     // Calculate the difference between start and goal on each axis
     double x_dif=x_goal-x_pos;
     double y_dif=y_goal-y_pos;
     double z_dif=z_goal-z_pos;

     double k=0;
     vector<Chkpts> points;

```

```
41      // As long as difference is not zero, so the goal is not reached...
42      while(x_dif!=0 || y_dif!=0 || z_dif!=0){
43          k++;
44          check.seq=k; //keeping track of sequence numbers to make sure waypoints are sent in
45                       //right order
46          check.yaw=0;
47
48
49
50         // If x_dif is not zero, the next waypoint will have a component in the x-direction
51         if(x_dif!=0){
52             x_pos=x_pos+(x_dif>0)-(x_dif<0);       //should be incremented by the step/grid
53                                                    //size, now set to 1
54             x_dif=x_goal-x_pos;
55             check.x=x_pos;
56         }
57
58         if(y_dif!=0){
59             y_pos=y_pos+(y_dif>0)-(y_dif<0);       //should be incremented by the step/grid
60                                                    //size
61             y_dif=y_goal-y_pos;
62             check.y=y_pos;
63
64         }
65
66         if(z_dif!=0){
67             z_pos=z_pos+(z_dif>0)-(z_dif<0);       //should be incremented by the step/grid
68                                                    //size
69             z_dif=z_goal-z_pos;
70             check.z=z_pos;
71         }
72
73         // Add the new waypoint to the list
74          points.push_back(check);
75      }
76
77         // Waypoints are written to a text file that can be plotted by Matlab, this will be
78         //changed to publishing the waypoints so the controller receives them
79         vector<Chkpts>::iterator it;
80
81      ofstream myfile;
82      myfile.open("Checkpoints.txt",ios::out);
83
84      for(it=points.begin(); it!=points.end(); it++){ //read out vector
85        Chkpts ch = *it; //it doesn't have the struct properties. So we make a new object of
86                         //Chkpts which will have the struct properties.
87        cout <<ch.seq<<", "<<ch.x <<", "<<ch.y <<", "<< ch.z <<", "<< ch.yaw << endl;
88
89        //write data to Checkpoints.txt
90        myfile <<ch.x <<", "<<ch.y <<", "<< ch.z << endl;
91
92      }
93      myfile.close();
94
95      //we now have a vector (of structs) with all the checkpoints the hexacopter has to fly
96      //towards (vector 'points')
97      cin.ignore();
98      cin.get();
99
100
101 return 0;
102 }
103
```

C.2.1 Adding a local potential field

As some kind of extra safeguard, a local potential field has been added for in-flight obstacle avoidance, refer Appendix C.3. If for some reason the MAV would come to close to an obstacle, a force opposite to the direction of the obstacle will be applied. To give an image of what this 'force field' around an obstacle looks like, refer to figure 4.2. The closer the MAV gets to the obstacle, the bigger the repulsive force.

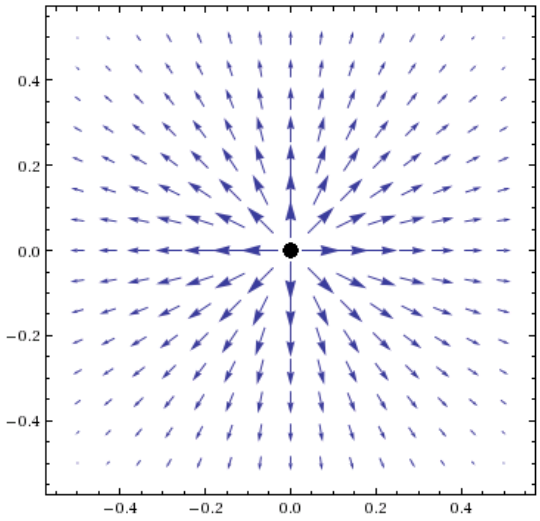

Figure 4.2: Force field around obstacle as seen by the MAV

To achieve this, a 7x7x7-matrix is present around the MAV when it is following the path. The matrix is shifted each time the next node is published. The new nodes are checked for occupancy, and if a node is found to be occupied the repulsive force is calculated as function of the distance to the MAV. This force is then added to the output of the PID-controller. The resulting force vector is sent to the MAV in the simulation, which will process this force vector and move according to it.

C.2.2 Local potential field algorithm

```
257  void Control::calculateRepulsion(Node center, double &fxz, double &fyz, double &fzz
258  {
259    // Array with the distance from center values.
260    double shift[7];
261
262    // Vector with occupied nodes.
263    std::vector<Node> occupied;
264
265    // Fill the shift array with values from -0.75 to 0.75.
266    for(int i = 0; i < 7; i++)
267      shift[i] = -0.75 + i * 0.25;
268
269
270    // Fill the occupied vector with all the occupied nodes surrounding the center.
271    for(int x = 0; x < 7; x++)
272    {
273      for(int y = 0; y < 7; y++)
274      {
275        for(int z = 0; z < 7; z++)
276        {
277          if((center.x + shift[x]) > 0.0 && (center.y + shift[y]) > 0.0 &&
278             (center.z + shift[z]) > 0.0)
279          {
280            // Check if the neighbour node is occupied. If it is, add it to the
281            // occupied vector.
282            if((tree.search(center.x + shift[x], center.y + shift[y],
283                center.z + shift[z])->getOccupancy()) > 0.1192)
284            {
285  ROS_INFO("FOUND");
286              Node neighbour;
287
288              neighbour.x = center.x + shift[x];
289              neighbour.y = center.y + shift[y];
290              neighbour.z = center.z + shift[z];
291
292              occupied.push_back(neighbour);
293            }
294          }
295        }
296      }
297    }
298
299    // Start calculating the repulsive forces.
300    std::vector<Node>::iterator it;
301
302  //  if(occupied.size() > 0)
303  //    ROS_INFO("Occupied nodes; %d", occupied.size());
304
```

```
305     for(it = occupied.begin(); it != occupied.end(); it++)
306     {
307       double dx, dy, dz, fInit, magnitude;
308       Node neighbour;
309
310       neighbour = *it;
311
312       fInit = 3;
313
314       dx = neighbour.x - center.x;
315       dy = neighbour.y - center.y;
316       dz = neighbour.z - center.z;
317
318       magnitude = sqrt(pow(dx, 2.0) + pow(dy, 2.0) + pow(dz, 2.0));
319
320       fx = fx + ((-fInit * dx) / magnitude) * (0.75 - magnitude);
321       fy = fy - ((-fInit * dy) / magnitude) * (0.75 - magnitude);
322       fz = fz + ((-fInit * dz) / magnitude) * (0.75 - magnitude);
323     }
324   }
325
```

References

Carsten, Joseph (2006), 3D Field D*: Improved Path Planning and Replanning in Three Dimensions
http://ri.cmu.edu/pub_files/pub4/carsten_joseph_2006_1/carsten_joseph_2006_1.pdf

Correll, Nikolaus (2011), Introduction to pathplanning for robotics
http://correll.cs.colorado.edu/?p=965

Daily, Robert and David M. Bevly (2008), Harmonic Potential Field Path Planning for High Speed Vehicles
http://www.nt.ntnu.no/users/skoge/prost/proceedings/acc08/data/papers/0383.pdf

Masoud, Ahmad A. (2008), Planning with Discrete Harmonic Potential Fields
http://cdn.intechopen.com/pdfs-wm/5366.pdf

Panati, S., Bayanjargal Baasandorj, Kil To Chong (2015), Autonomous Mobile Robot Navigation Using Harmonic Potential Field
http://iopscience.iop.org/article/10.1088/1757-899X/83/1/012018/pdf

Prestes, Edson, Silva Jr., Paulo M. Engel, Marcelo Trevisan, Marco A.P. Idiart (2002), Exploration method using harmonic functions
http://inf.ufrgs.br/~prestes/publicacoes/RAS02.pdf

Red blob games (2009), Introduction to A*
http://www.redblobgames.com/pathfinding/a-star/introduction.html

Red blob games, Heuristics
http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Rodriguez, S., Xinyu Tang, Jyh-Ming Lien, Nancy M. Amato, An Obstacle-Based Rapidly-Exploring Random Tree
https://parasol.tamu.edu/publications/download.php?file_id=525

Safadi, Hani (2007), Local Path Planning using Virtual Potential Field
http://www.cs.mcgill.ca/~hsafad/robotics/

Siegwart, R., Autonomous Mobile Robots
http://slideplayer.com/slide/5784565/

Slideshare, Dynamic Path Planning
http://www.slideshare.net/dare2kreate/dynamic-path-planning

Stanford, A* algorithm
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Vaščák, Ján (2007), Navigation of Mobile Robots Using Potential Fields and Computational Intelligence Means
https://uni-obuda.hu/journal/Vascak_9.pdf

Winands, Mark (2004), Informed Search in Complex Games, chapters 4,5 & 7
https://dke.maastrichtuniversity.nl/m.winands/documents/informed_search.pdf