DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2018

3D obstacle avoidance for drones using a realistic sensor setup

THOR STEFANSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

3D obstacle avoidance for drones using a realistic sensor setup

THOR STEFANSSON

Master in Computer Science
Date: August 26, 2018
Supervisor: Patric Jensfelt
Examiner: Joakim Gustafson
Swedish title: Hinderundvikande i 3D för drönare med en realistisk sensoruppsättning
School of Electrical Engineering and Computer Science

Abstract

Obstacle avoidance is a well researched area, however most of the works only consider a 2D environment. Drones can move in three dimensions. It is therefore of interest to develop a system that ensures safe flight in these three dimensions. Obstacle avoidance is of highest importance for drones if they are intended to work autonomously and around humans, since drones are often fragile and have fast moving propellers that can hurt humans. This project is based on the obstacle restriction algorithm in 3D, and uses OctoMap to conveniently use the sensor data from multiple sensors simultaneously and to deal with their limited field of view. The results show that the system is able to avoid obstacles in 3D.

Sammanfattning

Hinderundvikande är ett utforskat område, dock för det mesta har forskningen fokuserat på 2D-miljöer. Eftersom drönare kan röra sig i tre dimensioner är det intressant att utveckla ett system som garanterar säker rörelse i 3D. Hinderundvikande är viktigt för drönare om de ska arbeta autonomt runt människor, eftersom drönare ofta är ömtåliga och har snabba propellrar som kan skada människor. Det här projektet är baserat på Hinderrestriktionsmetoden (ORM), och använder OctoMap för att använda information från många sensorer samtidigt och för att hantera deras begränsade synfält. Resultatet visar att systemet kan undvika hinder i 3D.

Contents

Chapter 1

Introduction

Autonomous navigation for robots that move in the plane has been researched since the appearance of mobile robots in the late 60s [22]. This has resulted in numerous techniques to avoid obstacles in 2D, see section 2.1. UAVs were developed throughout the 20th century to be used in warfare [38]. More recently their use has expanded to other applications such as aerial photography, surveillance, mapping and product deliveries. These vehicles have traditionally been remotely controlled or relied on GPS coordinates. However, with the recent development of multirotor UAVs the possibility of flying in more cluttered environments than the open sky and even in GPS denied environments such as indoors has opened up. For these vehicles to operate autonomously, obstacle avoidance in 3D is needed.

1.1 Objective and scope

The objective of this thesis is to design a system that can avoid obstacles safely in a 3D GPS denied environment. Furthermore, the system is to be designed to work with a realistic sensor setup. More specifically the setup consists of a 3D camera, a lidar and simpler distance sensors. The environment is previously unknown, so that it needs to be discovered by the sensors.
The system receives information from the sensors and a goal point and gives velocity commands to the drone so that it flies to the goal point without colliding with obstacles. This means the system assumes a higher level path planner that feeds it with way points, a positioning system so we always know the position of the drone, and that there is

a low level controller seeing to that the drone moves with the desired velocity within reasonable deviations due to inertia. To limit the scope, the system will be evaluated in simulation.

1.2 Research Question

The research question of this project is the following: How can we design a system for avoiding obstacles and allow safe motion of a UAV in real world 3D environments?

1.3 Outline of thesis

Here the structure of the rest of the thesis is described. In chapter 2 the background of the thesis is described, 2D and 3D obstacle avoidance methods and a 3D mapping method. The chosen obstacle avoidance method, ORM in 3D, is described in some detail. Chapter 3 describes the method unique to this thesis; modifications to the ORM in 3D and how we can use it with our sensor setup. Chapter 4 outlines the experimental setup. It goes over hardware and software specifications as well as parameter values and the different test cases designed to show the performance of the system. Chapter 5 shows the results for each test case. In the discussion, chapter 6, the results of each test case are discussed as well as the advantages and limitations of the method. In chapter 7 the conclusions of the work are summarized and potential future work is described.

Chapter 2

Background

2.1 2D obstacle avoidance methods

2.1.1 Potential field methods

The method of artificial potential fields was first introduced by [18] in 1985 and works by creating a potential field where the obstacles have a repelling force and the destination has an attractive force. Then the robot moves like a particle in a potential field towards the goal while staying away from obstacles.
In 1988 the method of virtual force fields (VFF) was developed by combining certainty grids for representation of the obstacles and potential fields [2, 3]. This method was more suitable for noise in sensor data than the original potential field method because it uses a certainty grid for the environment representation and allowed continuous motion of the robot [2]. In 1991, the authors that developed VFF described the drawbacks of the potential field methods and why their new method of vector field histogram was superior [20]. The drawbacks of potential field methods are: trap situations due to local minima in the potential field, no passage between closely spaced obstacles even though the robot fits through, and oscillations in narrow passages or in presence of obstacles [20]. To eliminate the problem of local minimas, [19] used harmonic potential functions.

2.1.2 Vector field histogram (VFH)

In 1990 the authors of VFF developed the method of vector field histogram (VFH) [4]. The VFH works in the following way: A cartesian occupancy grid is constructed to map the obstacles. Next, the space around the robot is divided into polar sections, where the obstacle density in each direction from the robot is the sum of the cartesian occupancy grid that falls within its limits. From this, a polar histogram is constructed and a direction for the robot is selected where the probability of an obstacle is lower than a threshold.
This method was further improved by the creation of VFH+ [33] and VFH* [34]. The VFH method has the tendency to cut corners, which has been solved in VFH+ [33]. However the VFH+ can not direct the robot towards obstacles, making it badly suited for cluttered environments [23]. These methods are good at dealing with noisy sensor data because of their probabilistic representation of obstacles [22]. However they have the problem of tuning an empirical threshold, which needs to have a different value when navigating between close obstacles than when in open spaces [24].
The VFH and VFH+ have been extended to 3D and experiments performed in simulation [40] [1].

2.1.3 Dynamic window approach (DWA)

This method was developed in 1997 by [14] and works in two steps. In the first step a set of possible control commands is generated from the dynamics of the vehicle and the location of the obstacles. Then an objective function is maximized so it favors directions close to the goal direction, that points towards obstacle-free space and a fast velocity.
This method was generalized further with the global dynamic window approach [5]. The field dynamic window approach (FDWA) was developed to tackle the problem of DWA of crashing into obstacles that are close to but not directly on its trajectory [28].
A big advantage is that DWA directly selects a control command based on the dynamics of the vehicle. However, like the potential field methods, this method does not solve the problem of getting trapped inside a U-shaped obstacle [24].

2.1.4 Velocity obstacles

The velocity obstacles method was developed in 1998 by [13]. This method generates a set of possible commands from the dynamics of the vehicle just like DWA, except that it takes into account the velocities of the obstacles. It is therefore suitable in scenarios where the obstacles can move. The authors propose a number of heuristic methods to choose a direction [13]. This method was integrated with LQG control by [35]. It was extended to deal with multiple robots simultaneously by [7].

2.1.5 Nearness diagram (ND)

The method of nearness diagram navigation was first published in 2000 by [23]. It works in the following way: First, two nearness diagrams are constructed, similar to the polar diagram used in VFH. These diagrams are called the PND (Nearness Diagram from the central Point) and the RND (Nearness diagram from the Robot). Which means that the PND shows the distance from the center of the robot to each obstacle while the RND shows the distance from the robot edge in each direction to the obstacle in that direction. From the PND we get "valleys", directions in which there are no close obstacles and are sufficiently wide for the robot to pass through. We choose the valley that is closest to the direction of the goal location. From the RND we define if the obstacles are within a security distance of the robot. We get five different cases in the end, depending on: if obstacles are within security distance of the robot, if the goal location is within the selected valley and if the valley is wide or narrow. For each case there is a different equation for selecting the direction for the robot to move in.
This method was further developed by the creation of ND+ [25]. These methods solve many of the problems of previously mentioned methods, such as trap situations due to U shaped obstacles, oscillations, and parameter tuning since only one parameter needs to be tuned for the ND. However, this method does not use all the obstacle information available and therefore tends to have suboptimal trajectories in open spaces [21]. Therefore, one of the authors of ND created the obstacle-restriction method in 2005 [21]. The smooth nearness-diagram navigation (SND) was developed in 2008 as an evolution of ND+ to generate smoother motion by considering all obstacles surrounding the robot [11].

2.1.6 The obstacle-restriction method (ORM)

The obstacle restriction method developed in [21] works in the following steps. The first step is to define potential subgoals that are either in the direction of an edge of an obstacle, at a distance farther than the robot diameter or at the middle point between two obstacle points that are angularly beside each other relative to the robot. Then, it is checked whether the goal location can be reached, and if not, the closest reachable subgoal is selected. To check if a goal can be reached is done in the following way: All the obstacle points are expanded to circles with the robot radius. A line is drawn from the position of the robot to the goal and extended on both sides by the radius of the robot to form a rectangle. The goal is then reachable if the rectangle is not blocked completely by the circles. See figure 2.1.

The next step is to find the feasible set of directions. For each obsta-

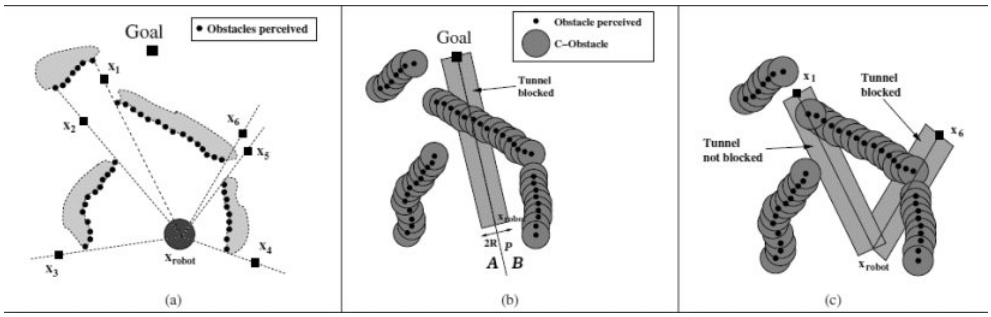


Figure 2.1: (a) shows the subgoals. In (b) we can see that the tunnel to the goal is blocked. In (c) x_1 is selected. Figure copied from [21].

cle we find the sets S_1 and S_2. S_1 represents the directions on the side of the obstacle that are not feasible, i.e. the side further away from the goal. S_2 represents an exclusion area around the obstacle so the robot will not run into it. See figure 2.2 (a)(b)(c). The complete set of non-desirable directions for all the obstacles is then the union of all the S_1 and S_2 sets, S_{nD}, and the set of desirable directions, S_D, is the complementary of S_{nD}. The boundaries of S_{nD} are called the closest left bound ϕ_L^{max} and the closest right bound ϕ_R^{max}.Then the direction of motion is decided from:

1. If $S_D \neq \emptyset$ and $\pi_{target} \in S_D$ then $\pi_{sol} = \pi_{target}$.
2. If $S_D \neq \emptyset$ and $\pi_{target} \notin S_D$ then the left bound ϕ_L^{max} or right bound ϕ_R^{max} of S_{nD} is selected depending on which is closer to the target direction.

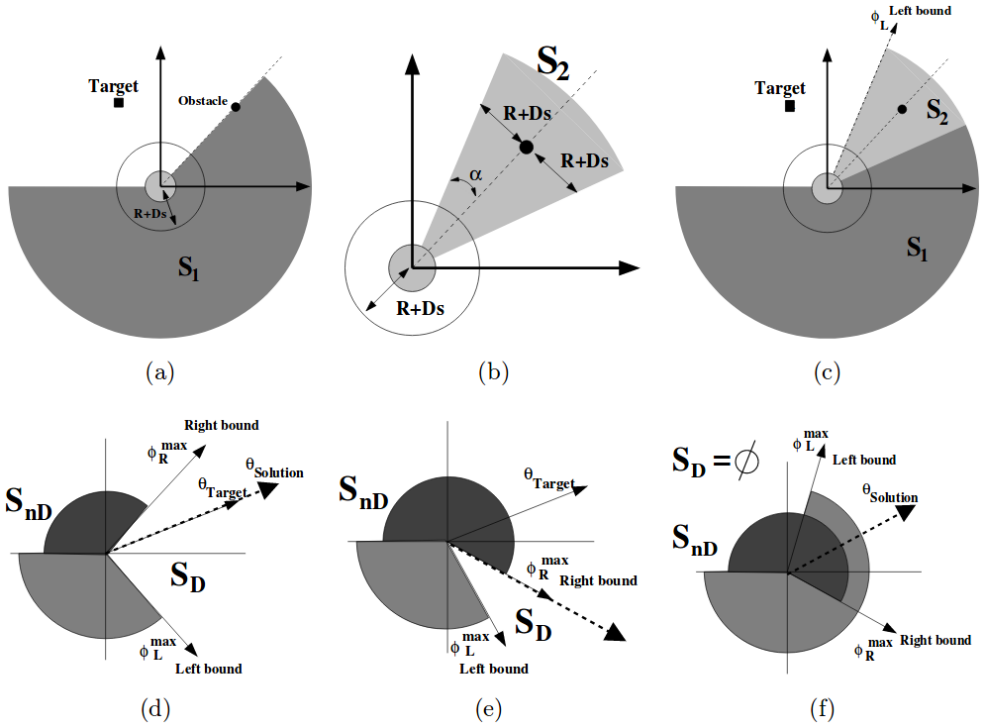


Figure 2.2: (a)(b)(c) show the creation of sets S_1 and S_2. (d)(e)(f) show the different selection of directions 1, 2, 3 respectively. Figure copied from [21].

3. If $S_D = \emptyset$ then $\pi_{sol} = \frac{\phi_L^{max} + \phi_R^{max}}{2}$ is selected as the target direction. See figure 2.2 (d)(e)(f).

Many of the advantages of the ORM compared with other methods are the same as for the ND; it does not have the problem of local minimas, oscillations, or parameter tuning [21]. Furthermore, the ORM performs better than the ND in open spaces [21]. The ORM was extended to 3D by [36] in 2006 and is described further in section 2.2.1.

2.2 3D obstacle avoidance methods

This section covers some of the research made for obstacle avoidance in 3D that has not already been mentioned. ORM in 2D was used to avoid obstacles with a remotely controlled UAV by [10].
A popular method to use for UAV obstacle avoidance involves opti-

cal flow [6]. This method is especially common to use for micro aerial vehicles(MAV) [12]. Since they require small and power efficient sensors, a camera is a good choice and therefore an optical flow method that can utilize it for navigational purposes [41]. Comparison between optical flow used by insects and applications for collision avoidance for drones can be found in [30]. In optical flow, a vision sensor, e.g. a camera, mounted on a robot is used. Optical flow assumes that the difference between two consecutive image frames in a video is mostly generated with the translation and rotation of the robot. When we know that translation and rotation, and the distance between the same point in two consecutive image frames, it's possible to calculate the distance to that point. A typical usage of optical flow for obstacle avoidance is to move the robot so that the optical flow on each side is balanced [39, 41]. The main problem with this method is detecting obstacles close to the direction of motion, since they will have little optical flow.
[8] use potential fields to avoid obstacles, and harmonic functions to avoid local minimas. [31] use a layered approach, with their reactive obstacle avoidance component using a potential field, and a number of parameters being set by observing the flight of a human operator. They used a 3D laser scanner and represent the environment with a probabilistic occupancy grid. They managed to complete over 700 successful runs, flying a helicopter in an outdoor environment, 5-11m above ground with speeds up to 10 m/s.

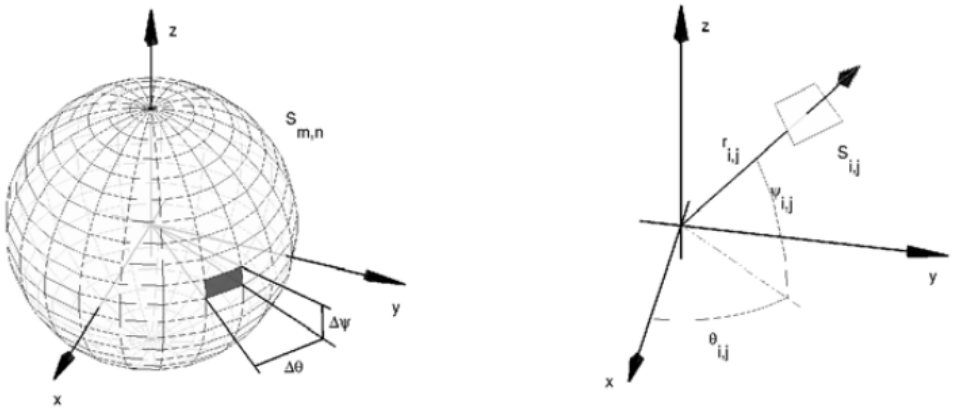
[29] approximate obstacle shapes with ellipsoids to perform 3D obstacle avoidance for fixed wing UAVs. [15] use a laser distance sensor along with GPS and inertial sensors to create a grid map of the environment and then use path planning based on Bézier curves to avoid obstacles. [37] converted the robotic path planning problem to the Semi-infinite Constrained Optimization Problem. That allowed them to perform path planning and obstacle avoidance taking into account the shape of the robot and the obstacles without the use of C-space.

2.2.1 The obstacle-restriction method (ORM) in 3D

The ORM, described in section 2.1.6 was extended from two to three dimensions by [36] in 2006. The work in this thesis builds on this 3D version of ORM and brings it from a theoretical study into a real world implementation. The method is described in some detail below.

Representation of environment

In order to represent the distance to an obstacle in all different directions the robot can move in, spherical coordinates are used. A sphere around the robot is constructed and divided into m x n sectors, represented as a matrix $M_{m,n}$. The azimuth angle with range $\theta \in (-\pi, \pi]$ is divided into n sectors and the elevation angle with range $\psi \in [-\frac{pi}{2}, \frac{pi}{2}]$ is divided into m sectors. See figure 2.3.



(a) (b)

Figure 2.3: (a) Shows how the space surrounding the robot is divided into angular boxes of size $\Delta\theta$ by $\Delta\Psi$. (b) A single sector in the sphere with location defined by spherical coordinates $\theta_{i,j}, \Psi_{i,j}, r_{i,j}$, where the distance r is the closest distance to an obstacle that falls within this sector. Figure copied from [36].

For each obstacle point we then calculate to which sector of the spherical matrix it belongs and store the lowest distance for each sector in the matrix.

Choosing a target direction

This spherical matrix is used to locate potential sub goals in the same way as in 2D ORM:
1) In the middle point between two adjacent obstacle distances in the

spherical matrix; who are farther apart than the robot diameter.
2) At the edge of an obstacle in the spherical matrix at a distance greater than the robot diameter.
The next step is to check if the goal can be reached. If not, the closest reachable subgoal is chosen as a target point. This is done by forming a tunnel from the robot location to the goal location with radius equal to the robot radius and checking if the tunnel is blocked or not in configuration space. See figure 2.4.

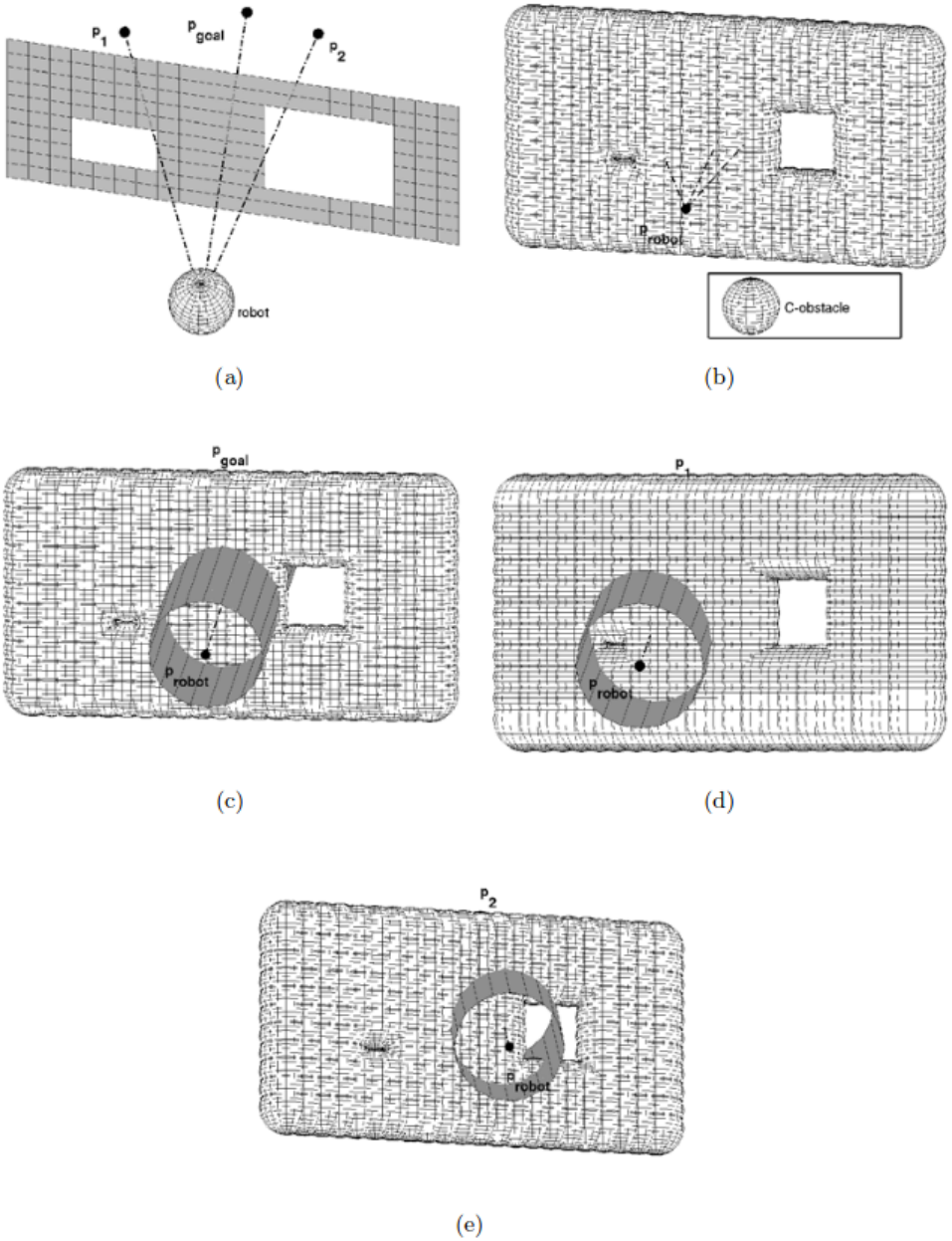


Figure 2.4: Shows how the reachability of points is estimated. (a) Lines drawn from the robot location to the location of the goal and two sub goals. (b) Configuration space is constructed by enlarging each obstacle point by the robot radius. (c) - (d) The tunnels to the goal point and subgoal p_1, are blocked and therefore those are not reachable. (e) The tunnel to sub goal p_2 is not blocked, so it is reachable. Figure copied from [36].

Subspaces

After a target point has been chosen from the goal point or sub goals, the next step is to decide in which direction to move. First we divide the directions into four quadrants:
Let the vectors (e_x, e_y, e_z) be the unitary vectors of the axis in the robot frame, the vector u_{targ} be the unitary vector in the target direction and the point $p_0 = (0, 0, 0)$ the origin.
Then we let the plane A be defined with $[p_0, e_x, e_z]$, B with $[p_0, e_z, u_{targ}]$, and C with $[p_0, e_y, u_{targ}]$. Then we let n_A, n_B and n_C be the normals to these planes, given by: $n_A = e_y$, $n_B = e_z \times u_{targ}$ and $n_C = u_{targ} \times e_y$. Then the quadrants are defined as seen in figure 2.5.

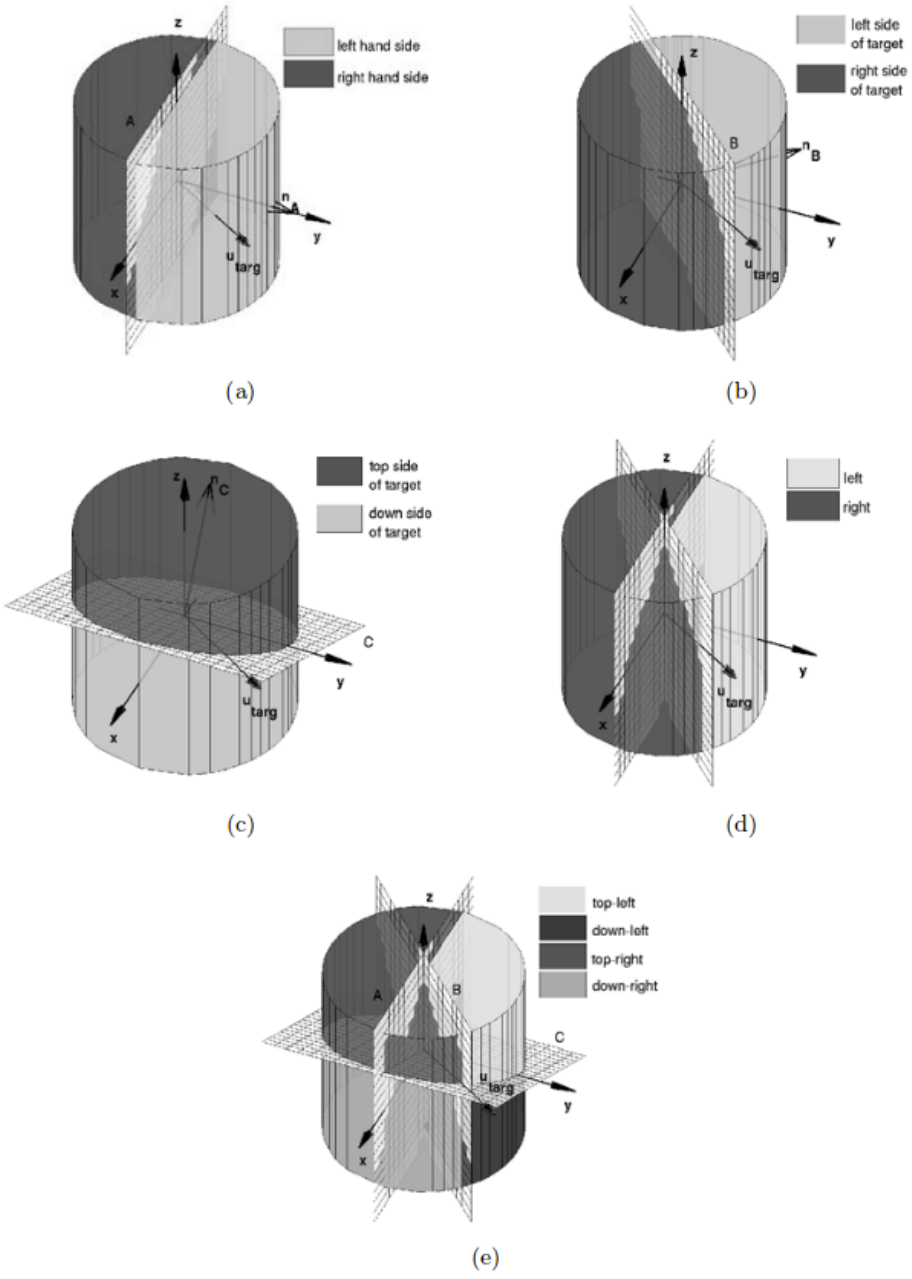


Figure 2.5: Shows how the space around the robot is divided into four quadrants. (a) Plane A with normal n_A marks the left and right hand side of robot. (b) Plane B with normal n_B marks the left and right hand side of the chosen target point. (c) Plane C with normal n_C marks the upper and lower side of the target with respect to the robot. Figure copied from [36].

Motion constraints

The next step is to define motion constraints for each obstacle point. Similarly to the ORM in 2D, the motion constraint for an obstacle is defined as the union of two subsets S_1 and S_2, that form a set of non desirable directions of motion. As in the 2D ORM, S_1 represents the directions on the side of the obstacle that are not feasible and S_2 an area around the obstacle.
S_1 is created with three planes: A and B mentioned earlier, and D defined by $[p_0, u_{obst}, u_{targ} \times u_{obst}]$, with normal $n_D = (u_{targ} \times u_{obst}) \times u_{obst}$.
Then we define D^+ as follows:
Given a vector v, we say that $v \in D^+$ if $n_D \cdot v > 0$.
Then S_1 is constrained by these spaces, so it is the intersection of the left quadrants and D^+ if the obstacle point is in the left quadrants:

$$S_1 = (TL \cup DL) \cap D^+, if u_{obst} \in (TL \cup DL) \tag{2.1}$$

Or the intersection of the right quadrants and D^+ if the obstacle point is in the right quadrants:

$$S_1 = (TR \cup DR) \cap D^+, if u_{obst} \in (TR \cup DR) \tag{2.2}$$

See figure 2.6.

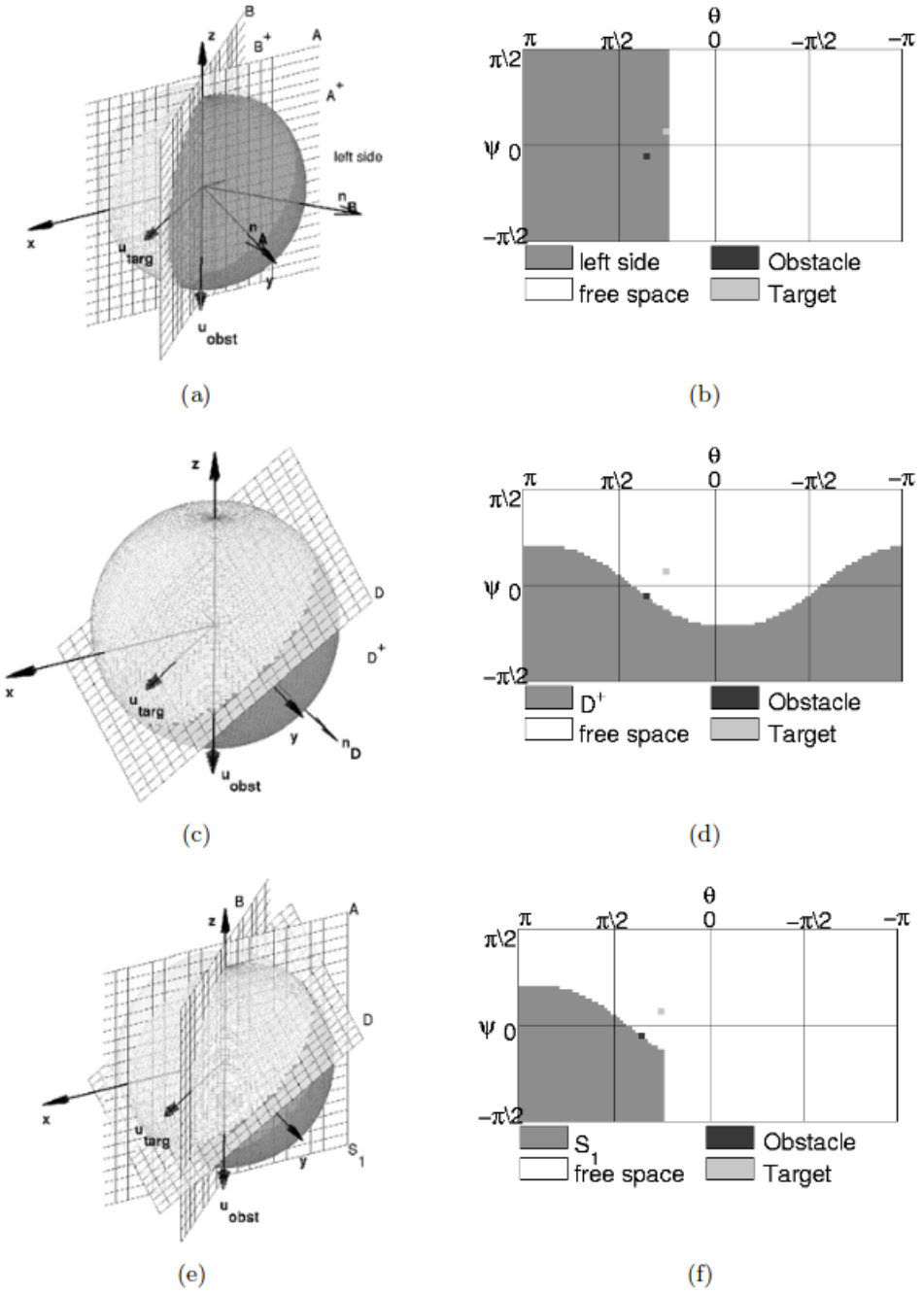


Figure 2.6: Shows the construction of S_1. (a) $A^+ \cup B^+$. (b) D^+. (c) $(A^+ \cup B^+) \cap D^+$. Figure copied from [36].

S_2 is an exclusion area around an obstacle defined in the following

way: $S_2 = \{u_i | \gamma_i \leq \gamma\}$, where

$$\gamma_i = arccos(\frac{u_i \cdot u_{obst}}{||u_i|| \cdot ||u_{obst}||}) \tag{2.3}$$

and

$$\gamma = \alpha + \beta, 0 < \gamma < \pi \tag{2.4}$$

with α and β given by:

$$\alpha = |atan(\frac{R + D_s}{d_{obst}})| \tag{2.5}$$

and

$$\beta = \begin{cases} (\pi - \alpha)(1 - \frac{d_{obst} - R}{D_s}) & \text{if } d_{obst} \leq D_s + R \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

where R is the radius of the robot, D_s is the safety distance, and d_{obst} the distance to the obstacle, see figure 2.7. The final set of motion constraints for an obstacle is then the union of its S_1 and S_2, see figure 2.8.

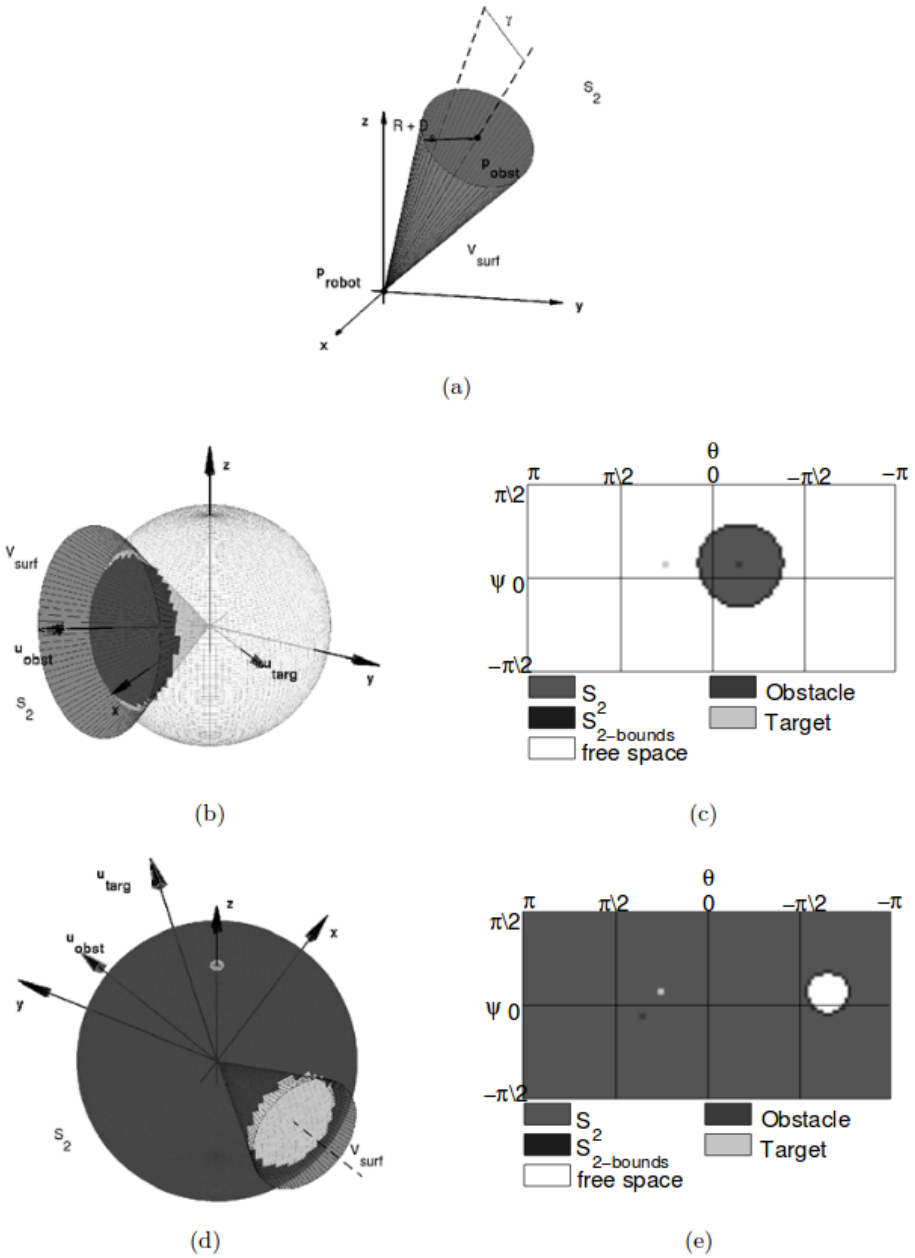


Figure 2.7: Shows the construction of S_2. (a) The boundary of S_2 is $R + D_s$ at the obstacle distance, when $d_{obst} > D_S + R$. γ is the angle between the center of the S_2 cone and its boundary. (b) - (c) Shows how S_2 is discretized into a matrix representation. (d) - (e) Here we are very close to the obstacle point, $d_{obst} < D_S + R$ and then $\gamma = \alpha + \beta$. Figure copied from [36].

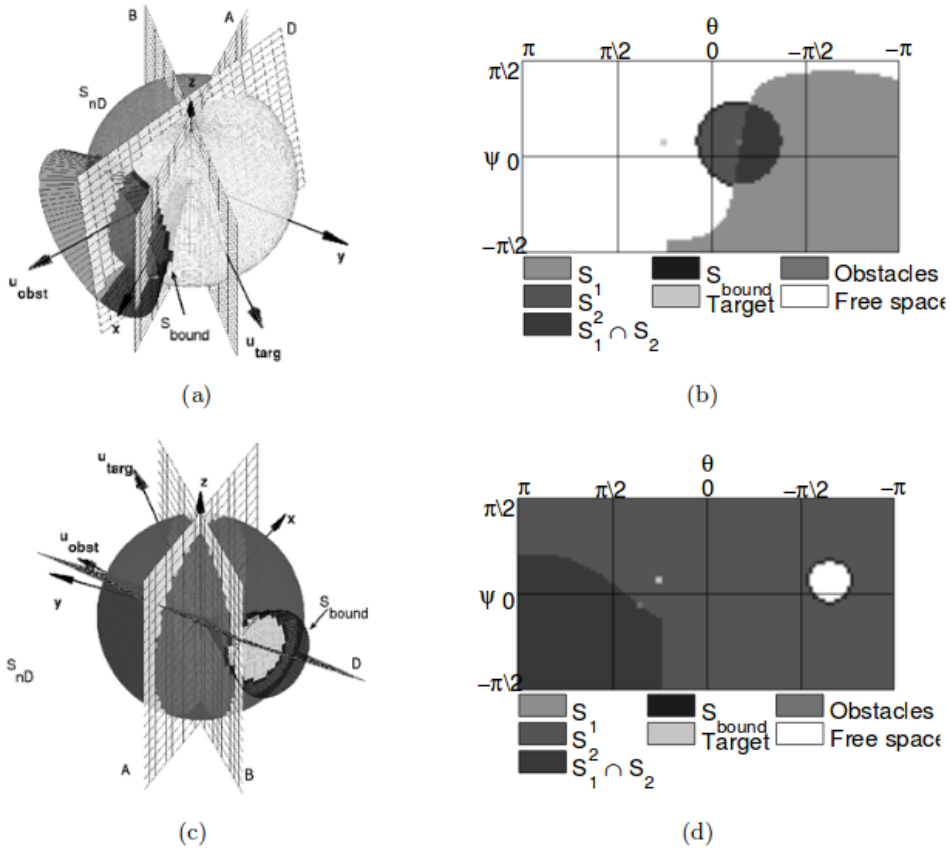


Figure 2.8: Shows the combination of S_1 and S_2. The boundary of S_2 that does not belong to S_1 forms S_{bound}. (a) - (b) Here $d_{obst} > D_S + R$ and $\gamma = \alpha$. (c) - (d) Here $d_{obst} < D_S + R$ and $\gamma = \alpha + \beta$. Figure copied from [36].

Selecting the direction of motion

The next step of the method is to select the direction of motion for the robot. The motion constraint for a single quadrant of the space is defined as the union of the motion constraints for all obstacle points that fall within that quadrant. So if the motion constraints for a single obstacle p_i in the top left (TL) quadrant is

$$S_{nD}^{TL,i} = S_1^i \cup S_2^i \tag{2.7}$$

then the set of motion constraints for the TL quadrant is

$$S_{nD}^{TL} = \cup_i S_{nD}^{TL,i} \tag{2.8}$$

We also have that

$$S_1^{TL} = \cup_i S_1^{TL,i} \tag{2.9}$$

and

$$S_2^{TL} = \cup_i S_2^{TL,i} \tag{2.10}$$

We call the union of motion constraints for all quarters

$$S_{nD} = S_{nD}^{TL} \cup S_{nD}^{TR} \cup S_{nD}^{DR} \cup S_{nD}^{DL} \tag{2.11}$$

Then the set of desired directions becomes

$$S_D = \{\mathbb{R}^3 \backslash S_{nD}\} \tag{2.12}$$

A set of bounding directions is calculated for each quadrant and is the boundary of S_2 that is not in S_1. Thus, for TL: if we call the boundary of S_2^{TL}, $S_{2-bound}^{TL}$ then we get

$$S_{bound}^{TL} = \{p | p \in S_{2-bound}^{TL} \text{ and } p \notin S_1^{TL}\} \tag{2.13}$$

The dominating direction for each quadrant is chosen in the following way:
1) If there are no free directions, $S_D = \emptyset$, we choose the direction in S_{bound} in each quadrant that has the smallest angle relative to the robot direction.
2) If there exist free directions, $S_D \neq \emptyset$, we choose the direction in S_{bound} in each quadrant that has the smallest angle relative to the target direction.
Depending on the number of quadrants where the target direction belongs to the motion constraint; the final chosen direction is calculated in different ways:
1) The target direction belongs to the set of free directions, $u_{targ} \in S_D$. Then the final direction of motion is chosen as $u_{sol} = u_{targ}$.
2) The target direction belongs to the motion constraint of one quadrant $G \in \{TL, TR, DR, DL\}$. Then the final direction of motion is chosen as the dominating direction of that quadrant, $u_{sol} = u_{dom}^G$.
3) The target direction belongs to the motion constraint of two quadrants $G_1, G_2 \in \{TL, TR, DR, DL\}$. Then the final direction of motion is chosen as the medium value of the dominating directions of the two quadrants, $u_{sol} = \frac{u_{dom}^{G_1} + u_{dom}^{G_2}}{2}$. See figure 2.9
4) The direction belongs to the motion constraint of three quadrants $G_1, G_2, G_3 \in \{TL, TR, DR, DL\}$. Then we first calculate the medium

direction of the quadrants that are anti symmetric (i.e. TL and DR or TR and DL):

$$u_{dom}^{G_1-G_2} = \frac{u_{dom}^{G_1} + u_{dom}^{G_2}}{2} \tag{2.14}$$

and then calculate the medium direction of $u_{dom}^{G_1-G_2}$ and the remaining set G_3 as the chosen direction:

$$u_{sol} = \frac{u_{dom}^{G_1-G_2} + u_{dom}^{G_3}}{2} \tag{2.15}$$

5) The direction belongs to the motion constraint of all four quadrants. First the medium values of the anti symmetric quadrants are computed:

$$u_{dom}^{TL-DR} = \frac{u_{dom}^{TL} + u_{dom}^{DR}}{2} \tag{2.16}$$

and

$$u_{dom}^{TR-DL} = \frac{u_{dom}^{TR} + u_{dom}^{DL}}{2} \tag{2.17}$$

Now, let plane ε be the plane defined by $[p_0, u_{dom}^{TL-DR}, u_{dom}^{TL} \times u_{dom}^{DR}]$ and F similarly by $[p_0, u_{dom}^{TR-DL}, u_{dom}^{TR} \times u_{dom}^{DL}]$. They have normals

$$n_\varepsilon = (u_{dom}^{TL} \times u_{dom}^{DR}) \times u_{dom}^{TL-DR} \tag{2.18}$$

$$n_F = (u_{dom}^{TR} \times u_{dom}^{DL}) \times u_{dom}^{TR-DL} \tag{2.19}$$

Finally,

$$u_{sol} = n_\varepsilon \times n_F \tag{2.20}$$

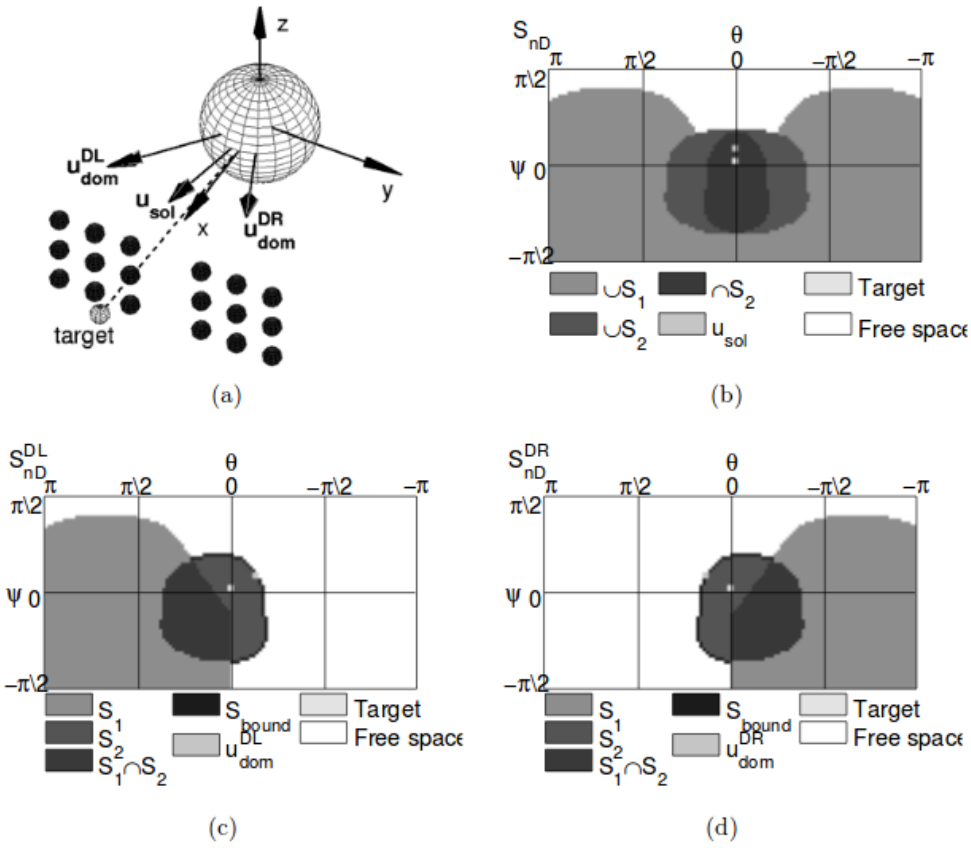


Figure 2.9: Shows how we choose the final direction of motion when the target direction belongs to the motion constraints of two subspaces. (a) Obstacles located in front of the robot, so that the target direction belongs to the motion constraints of the down right (DR) and (DL) subspace. The dominant directions of those subspaces, u_{dom}^{DL} and u_{dom}^{DL}, and the average of those two directions, u_{sol}, are shown. (b) The combination of the motion constraints for the subspaces. We can see that u_{sol} is higher than the target direction. However, u_{sol} is still within the set of S_2 of both subspaces. (c) The motion constraints and dominant direction for the DL subspace. (d) The motion constraints and dominant direction for the DR subspace. Figure copied from [36].

Velocity calculation

In [36] it is assumed the robot can rotate in any direction but that it can only fly in that one forward direction. With these constraints the

translational velocity is calculated as

$$v = \begin{cases} v_{max} * (\frac{\frac{pi}{2} - |\theta|}{\frac{pi}{2}}) & \text{if } d_{obst} > D_s + R \\ v_{max} * \frac{d_{obs} - R}{D_s} * (\frac{\frac{pi}{2} - |\theta|}{\frac{pi}{2}}) & \text{if } d_{obst} \leq D_s + R \end{cases} \tag{2.21}$$

where θ is the angle between the current robot direction e_x, and the new direction of motion u_{sol}:

$$\theta = arccos(\frac{u_{sol} \cdot e_x}{||u_{sol}|| \cdot ||e_x||}) \tag{2.22}$$

Thus, the translational velocity is negatively proportional to the angle the robot needs to turn. The translational velocity is 0 if $\theta > \pi/2$.
The angular velocity is calculated as

$$\omega = \omega_{max} * \frac{\theta}{\frac{\pi}{2}} \tag{2.23}$$

Thus, the angular velocity is proportional to the angle the robot needs to turn.

2.3 OctoMap

Since the UAV that the method is developed for in this thesis does not have sensor coverage in all directions, OctoMap was chosen to represent the environment. OctoMap is a probabilistic 3D mapping framework based on octrees and uses probabilistic occupancy estimation [17]. It is easy to combine information from multiple sensors with OctoMap and since the updating of the map is done in a probabilistic fashion it can deal with noise in sensor data and a dynamic environment [17].

Chapter 3

Method

Since the method of ORM in 3D, which was described in section 2.2.1, is mostly followed; only changes to the original work by [36] will be described in this section. The largest difference is that in the work by [36] the distance to obstacles in all directions is known, but in this work we use the method with a more realistic sensor setup.

3.1 Sensors to OctoMap

The robot that this project is developed for is equipped with a depth camera facing forward, a lidar that has a 270° field of view (FOV), sonars with one facing up and the other down and a TeraRanger laser distance sensor facing down, see figure 3.1. Further specifications concerning the sensors can be seen in table 3.1

As mentioned before, since the robotic platform that is used in this thesis does not have sensor information in all directions, OctoMap [17] is used to represent the environment, see section 2.3. The information from the lidar, the TeraRanger and the depth camera are published to OctoMap, in such a way that their maximum range is taken into account. This means that values outside of their maximum range only result in free space in OctoMap up until the maximum range, but not occupied space, see figure 3.2.

The sonars on the top and bottom of the drone have a relatively wide FOV of 55° and do not contribute to the occupied space in OctoMap since they could otherwise mark space as occupied that is only partially so, see figure 3.3. Instead, the information from the sonars is used directly in the spherical matrix as described in the next section.

Physical sensors			
Sensor	Model number	Field of view	Range
Depth camera	Primesense Carmine 1.09	45° vertical 54° horizontal	0.35 - 3m
Lidar	UST-20LX	270°	0.06 - 20m
Sonars	SRF08	55°	0.03 - 6m
Laser distance sensor	TeraRanger One	3°	0.2 - 14m

Table 3.1: Main characteristics of the physical sensors. The data sheets can be found in [27], [16], [9], [32].

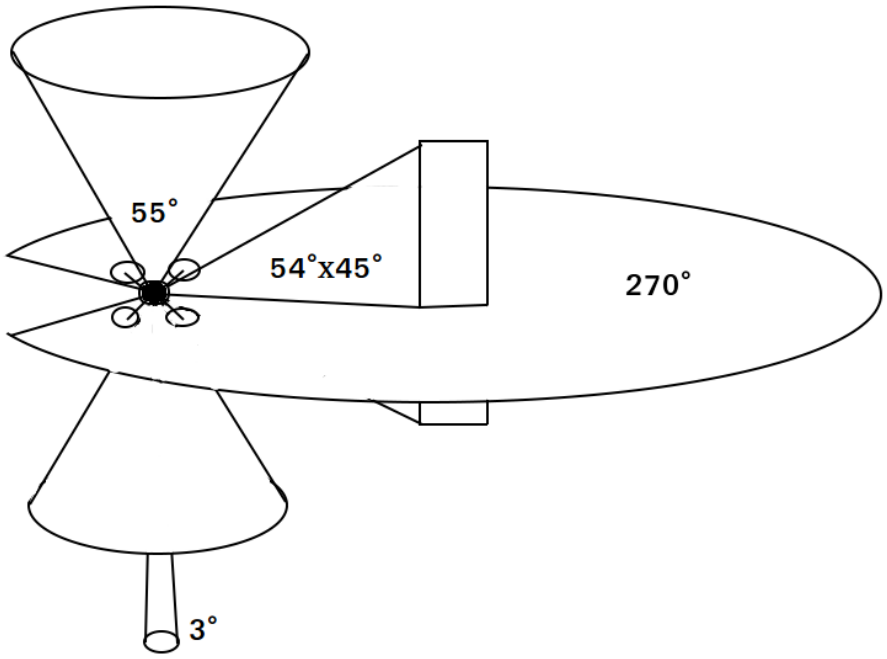


Figure 3.1: The sensor setup of the drone, along with the field of view of each sensor. A 3D camera with a FOV of 54° horizontal and 45° vertical. A lidar with a 270° FOV. Sonars facing up and down with 55° conical FOV. A laser distance sensor facing down with a 3° FOV.

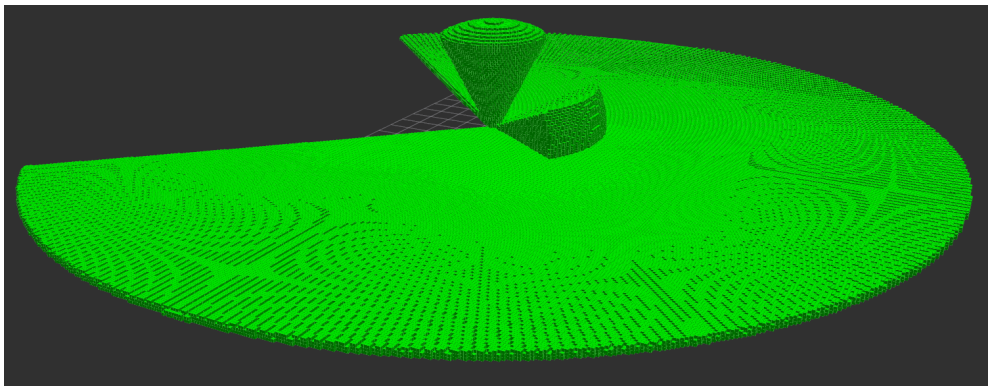
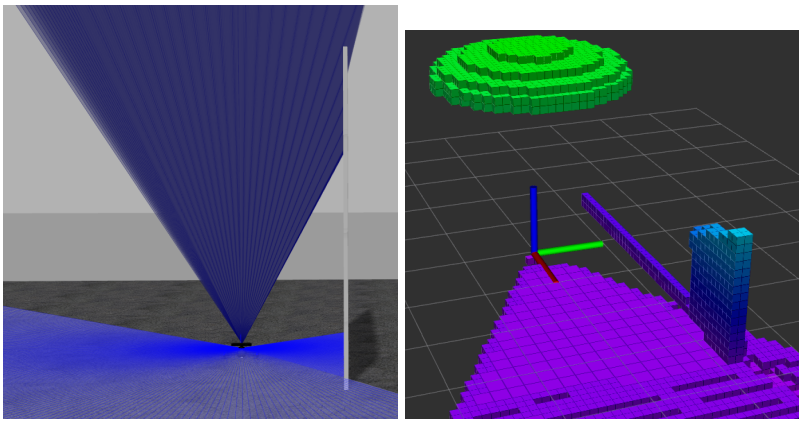


Figure 3.2: The free space in OctoMap, when there is no obstacle in the way except the floor beneath the robot. Then the space is marked as free up to the maximum range of the sensors.

3.2 Construction of the spherical matrix

The spherical matrix that is used to locate potential subgoals, described in 2.2.1 and figure 2.3, is constructed by getting information from OctoMap in a similar way as is described in [1]. The feature of OctoMap to loop through all nodes within a bounding box is used to find all occupied voxels within a cube around the robot. Then only the voxels that are within a sphere with diameter equal to the side length of the cube are considered to give a rotation invariant representation of the environment. This means essentially that we have a single maximum distance in all directions for adding to the spherical matrix. We finally find in which sector of the spherical matrix each occupied node belongs to and store the shortest distance of all nodes that fall within a certain sector in the spherical matrix. If an occupied node is large enough or close enough to the robot it can belong to more than one sector in the spherical matrix.
The range from the sonars is added straight to the spherical matrix since the sonars do not contribute to the occupied space in OctoMap as described in the previous section. However, to get rid of the effect described in figure 3.3, it is only added when we have not already added that obstacle to OctoMap. Therefore, if we have already detected an obstacle in the FOV of the sonar within 0.15m from the sonar reading,



(a) (b)

Figure 3.3: What happens when the sonars can mark space as occupied. (a) A wall is placed to the right of the drone. (b) The whole cap is marked as occupied at the range detected by the sonar.

it is not added. The reason for allowing up to 0.15m difference is because of the OctoMap resolution and the discretization effects of the spherical matrix.

3.3 Locating the subgoals

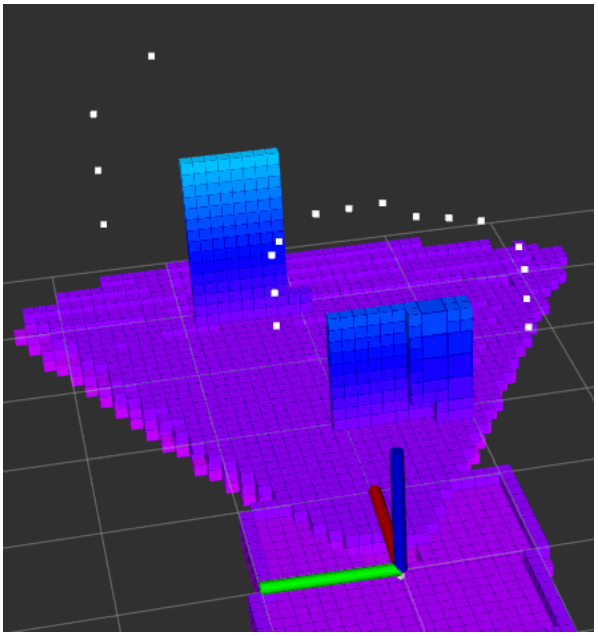


Figure 3.4: Screenshot from RViz that shows how potential subgoals are located.

Figure 3.4 shows an example of locating potential subgoals. If we locate the subgoals as in section 2.2.1, we can get subgoals incorrectly located on the floor or on a surface. This happens when we have an incomplete map of the environment, so subgoals can be located on the edge of the known section of the floor or surface. It can also happen because of the discretization effect of the spherical matrix.
If the distance difference between two contiguous points is larger than the robot diameter, a subgoal is located between them. This can happen between two points on the same plane if the angular resolution of the spherical matrix is not very high. To remove these false positive subgoals, all subgoals located very close to the floor are removed as well as all subgoal which have coordinates that are marked as occupied in OctoMap. Otherwise the subgoals are located as described in section 2.2.1.

If the goal point is not reachable the closest reachable subgoal is chosen as a target point as in [36].

3.4 Method for checking if a point is reachable

A different method for checking if a point is reachable is used than in [36] since that algorithm was found to have logical errors. In short, the method needs to create a cylinder between the robot and the target point with radius equal to the robot radius, r_{robot}, and check if it is completely blocked or not. Perhaps the most obvious logical error of the algorithm described in [36] is that it does not take into account obstacle points that are located outside of the cylinder but are inside of the cylinder when the configuration space (C-space) is created, see figure 3.5. When C-space is created, all obstacle points are enlarged to a sphere with radius equal to r_{robot}. This means that if they are within r_{robot} distance from the cylinder a part of the obstacle sphere will fall within the cylinder. That means that all points that are within $2 * r_{robot}$ distance from the cylinder center line need to be considered.

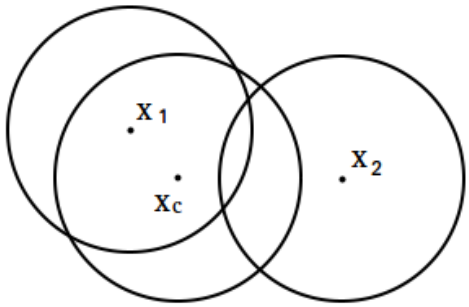


Figure 3.5: x_c is the center of the cylinder from the robot to the target. The cylinder has radius equal to the robot radius. x_1 and x_2 are obstacle points, which are enlarged by r_{robot} in C-space. The method developed by [36] considers x_1 as an obstacle but not x_2 even though it blocks the cylinder partially.

In this project, a discretized C-space is constructed using all obstacle points within $2 * r_{robot}$ distance from the cylinder center and then a search algorithm is used to check if the cylinder with radius equal to the robot radius is fully blocked or not. Obstacle points here are occupied nodes in the constructed OctoMap. To find the relevant points, we loop through a bounding box first, covering the area around the

cylinder from the robot to the target point with radius equal to the robot diameter. Then all points that are within that cylinder are considered and the relevant nodes in the C-space marked as occupied. If no obstacle point is found within the cylinder, the point is marked directly as reachable. Otherwise we use the search algorithm, that can be found in Appendix B, to check if the cylinder is blocked or not.

3.5 Selecting the direction of motion

Selecting the direction of motion follows mostly [36], with a few exceptions. First of all, an obstacle point is defined as a sector in the polar matrix with a distance d_{obst} smaller than the sum of the distance to the target point d_{targ} plus the robot radius r_{robot} : $d_{obst} < d_{targ} + r_{robot}$. This is not mentioned in [36] but something similar was probably used.
Second, a different safety distance is used depending formula being used. Now we define $D_{s,\alpha}$ and $D_{s,\beta}$ as the safety distances used in the calculation of α and β respectively, see formulas 2.5 and 2.6. By looking at formulas 2.3 to 2.6, notice that γ is the angle that defines a circle of directions around the obstacle point direction that should be avoided. A direction that is on this boundary around the obstacle points is chosen as a desirable direction, which means that when $d_{obst} > D_{s,\beta}$, $\gamma = \alpha$ and by following this direction the robot should be exactly $R + D_{s,\alpha}$ from the obstacle point when the robot is closest to it, see figure 3.6. Now if $D_{s,\alpha} = D_{s,\beta}$, and the robot flies a little inside

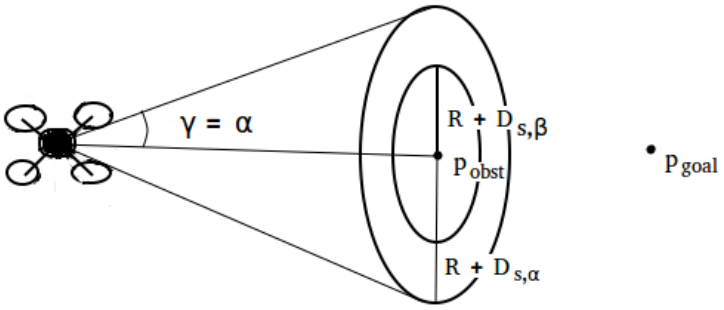


Figure 3.6: Shows the exclusion area S_2 around an obstacle point. A direction on the boundary of the cone with center in p_{obst} and radius $R + D_{s,\alpha}$ is chosen as the direction of motion. If we get within $R + D_{s,\beta}$ of p_{obst} we will have a direction change for the drone.

of the safety distance from the obstacle, γ will increase by a factor of β and the robot suddenly needs to turn farther away from the obstacle being avoided. Therefore, it can be beneficial to have $D_{s,\alpha} > D_{s,\beta}$ to avoid sudden changes in the robot heading when it gets close to an obstacle.
Sometimes, when the robot is very close to a wall for example, the motion constraint of a quadrant can cover the whole set of possible directions, so no direction of motion is free. In this case the dominant direction for that quadrant is calculated as the opposite of the average of all directions that are within the safety distance of the robot in that specific quadrant:

$$u_{dom}^{G} = -\frac{\sum_i u_{close,i}^{G}}{n_{G,close}} \tag{3.1}$$

where $G \in \{TL, DL, TR, DR\}$, $u_{close,i}^{G} \in G$ is a direction in which the distance is shorter than $D_{s,\beta}$ and $n_{G,close}$ is the number of such directions in quadrant G.
Because of the limited field of view of the 3D camera, the robot is constrained to always turn towards the direction that it is about to fly in. The constraint is that if the robot is more than 10° away from the selected direction in the planar field, it only turns and translational velocity is zero. When choosing the value of this parameter it is necessary to have the FOV of the 3D camera in mind, which is 54° horizontally in our case. This affects the choice of the dominant direction of each quadrant. In the original work by [36], the dominant direction is chosen as the direction on the boundary closest to the current robot direction if there are no free directions, but the direction closest to the target direction otherwise. Since we do not move the robot forward if there is a large angle difference between its heading and the chosen direction, it is unnecessary to choose a direction close to the robot direction and therefore the direction closest to the target direction is always chosen.
In the original work [36], when the target direction belongs to the motion constraints of all four quadrants, a number of cross product calculations are performed to choose the final direction of motion, see equations 2.14 to 2.20. This can however give a strange solution because of the properties of the cross product. As an example, from a simulation run we get the following numbers:

$$u_{dom}^{TL} = (0.933013, -0.25, -0.258819)$$

$$u_{dom}^{TR} = (0.756468, -0.147922, 0.42544)$$
$$u_{dom}^{DR} = (0.683013, 0.683013, 0.258819)$$
$$u_{dom}^{DL} = (0.880037, -0.139384, 0.45399)$$

and finally

$$u_{sol} = (-0.0243931, 0.0019842, -0.0153574)$$

Here we can see that the dominant directions of all subspaces are located in front of the robot but the final chosen direction of motion is backwards.

Therefore, when the target direction belongs to the motion constraints of all four quadrants, we simply calculate u_{sol} as the average of all four dominant directions:

$$u_{sol} = \frac{u_{dom}^{TL} + u_{dom}^{TR} + u_{dom}^{DR} + u_{dom}^{DL}}{4} \tag{3.2}$$

This is of course not a perfect solution either but produced better results in our experiments.

3.6 Final motion computation

The final motion computation is based on formulas 2.21 to 2.23. However equation 2.22 is changed so theta is the angle between the robot and the chosen direction of motion *in the xy-plane*, since the UAV can only turn by the vertical axis. As mentioned before, the robot is only allowed to move forward when the target direction is within 10 ° of the current robot direction in the xy-plane. If the target direction is within these limits, the robot moves directly in the target direction while also turning to face it completely. Since the UAV can only turn in the xy-plane, we do not turn it if the chosen direction is almost directly above or below the robot.

Just like with using different safety distances depending on equation being used when selecting the direction of motion, having a different safety distance when deciding on robot speed was found to be beneficial. While a low speed is sometimes necessary when moving close to obstacles, a higher speed can be desirable when the robot is far away from obstacles. In order to speed up the total time it takes the robot to reach its goal while having sufficiently low speed to traverse between

close obstacles, a longer safety distance was chosen for the velocity computation. A longer safety distance allows for a higher maximum speed.
Lastly, we have a critical safety distance $D_{s,crit}$, which means that if the closest obstacle is within $D_{s,crit} + R$ distance from the robot, we simply fly slowly in the opposite direction of the closest obstacle.

Chapter 4

Experimental setup

This chapter describes the experimental setup.

4.1 The physical UAV

The UAV this project was developed for has six propellers, a PrimeSense depth camera facing forward, a Hokuyo lidar with around 270 field of view (FOV), sonars facing up- and downwards and a teraranger laser facing downwards. Further specifications can be found in table 3.1 and [27], [16], [9], [32].

4.2 Testing environment

Simulation was constructed with Gazebo, with the code written in C++ for ROS and rviz used for visualization. The real sensors have noise that needs to be accounted for when moving from simulation to the real world. In simulation, the sensors always detect the surfaces their rays land on which is not always true in reality. The specifications for the hardware used can be seen in table 4.1

Simulation hardware specifications	
Memory (RAM)	32 GiB
Processor	Intel Core i7-7700K CPU @ 4.20GHz * 8
Graphics card	GeForce GTX 1050 Ti
OS type	64-bit

Table 4.1: The simulation hardware specifications

Simulation parameters	
Algorithm parameters	
α - Safety distance, $D_{s,\alpha}$	0.8 m
β - Safety distance, $D_{s,\beta}$	0.7 m
Velocity safety distance, $D_{velocity}$	1.2 m
Critical safety distance, $D_{s,crit}$	0.07 m
Maximum velocity	0.4 m/s
OctoMap resolution	0.1m
Spherical matrix resolution	6°
UAV parameters	
Mass	1.5kg
Radius	35cm
Inertia, ixx	0.0348
Inertia, iyy	0.0459
Inertia, izz	0.0977

Table 4.2: The simulation parameters

4.3 Parameters used in implementation

Parameters used for the physical sensors are listed in table 3.1. The difference between them and the simulated sensors is that the sonars, the depth camera and the TeraRanger send sensor data of maximum five meters to OctoMap in the simulation. Other parameters that affect the result of the simulation are listed in table 4.2. In the simulation, the UAV itself has a diameter of 70 cm and is around 15 cm tall, counting the sensors. In the code, the UAV is approximated as a sphere with diameter of 80 cm to simplify the calculations and to always ensure a little distance from the obstacles.

4.4 Test cases

This section describes the different test cases used to evaluate the performance of the system.

Figure 4.1: Shows the set up of test case 0. No obstacles are present.

4.4.1 Test case 0: No obstacles

In this test the robot needs to fly straight a 10m distance, 2m above the ground. See figure 4.1. This means the closest obstacle to the robot is the ground around 2 meters away which is farther than the velocity safety distance. Therefore, the robot should fly to the goal point with the maximum velocity.

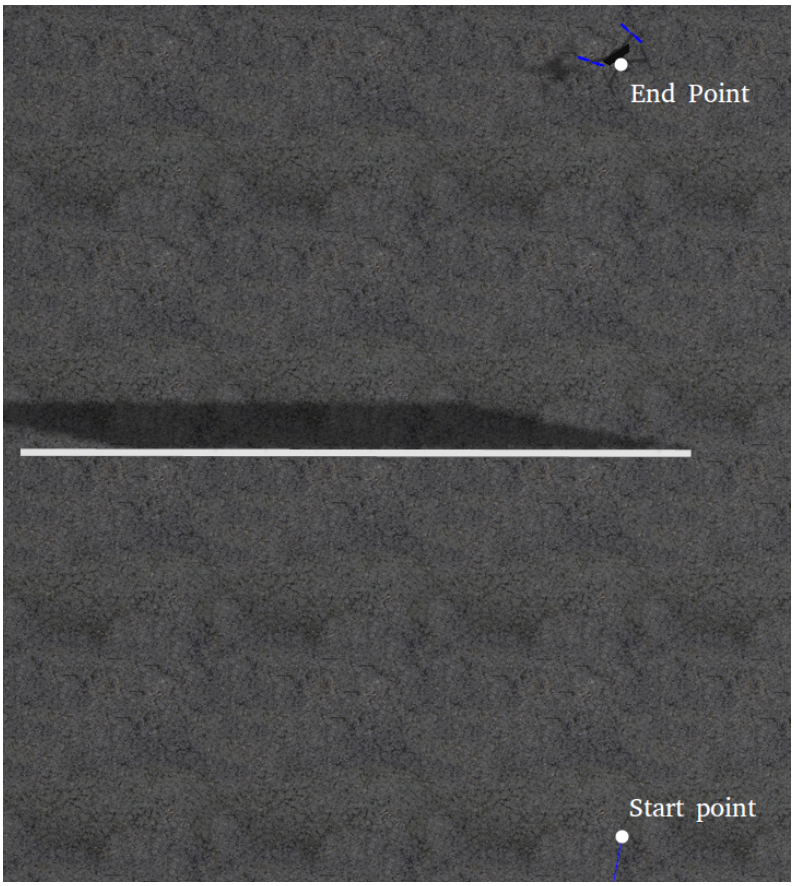


Figure 4.2: Shows the set up of test case 1. The robot needs to fly around the wall.

4.4.2 Test case 1: Simple wall

In this test case, a simple wall is placed in front of the robot to the left, so that the robot needs to turn to the right to avoid colliding with the wall. The wall has a height of four meters and a width of five meters. See figure 4.2. This test case is designed to show the basic functionality of the obstacle avoidance system.

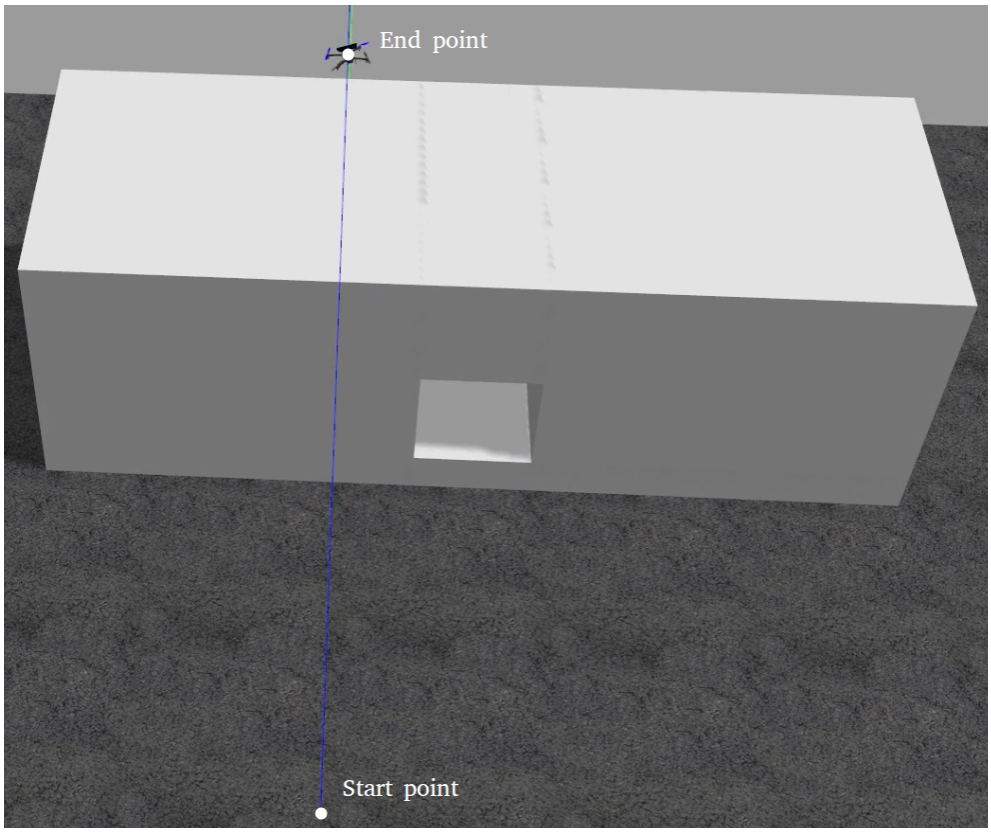


Figure 4.3: Shows the set up of test case 2. The robot needs to fly through a narrow tunnel.

4.4.3 Test case 2: Narrow passage

Here a three meter thick wall, with a tunnel of size 1.2 by 1.2 m, is placed between the robot and the goal point. It is placed so that the robot needs to turn a little and fly through the tunnel to reach the goal point. See figure 4.3. This test case is designed to show the ability of the method to traverse through narrow spaces.

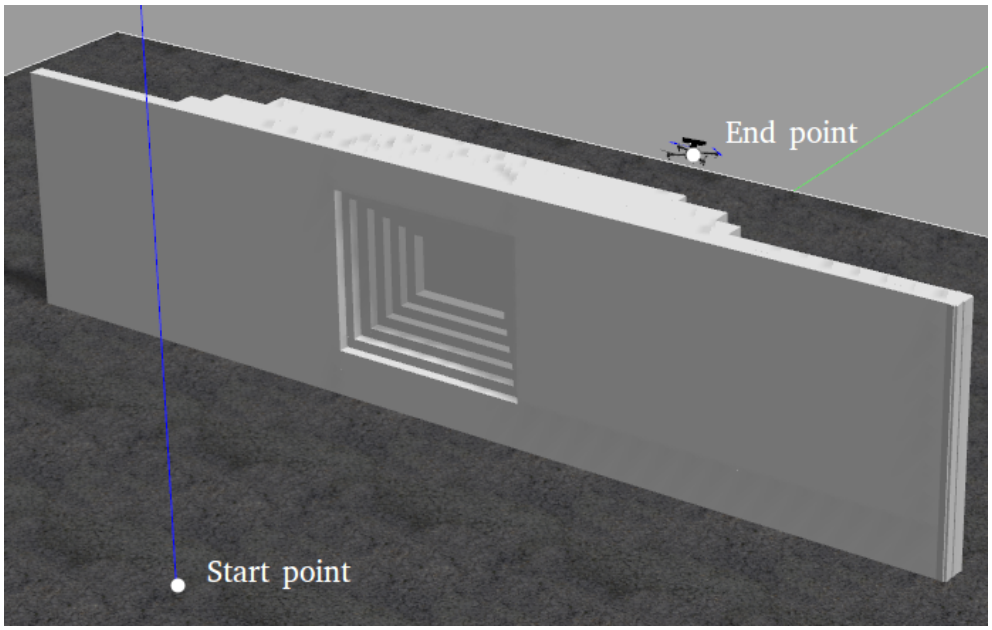


Figure 4.4: Shows the set up of test case 3. The robot needs to fly over the wall.

4.4.4 Test case 3: Trap

In this case a wide wall with a cavity that could potentially act as a local minima for some obstacle avoidance algorithms is placed between the robot and the goal point. The wall is 2.5 meters high. The robot needs to fly over the wall to reach its goal point. See figure 4.4. This test case was made to show that the robot can avoid a trap situation and that it can avoid obstacles by flying over them.

Figure 4.5: Shows the set up of test case 4. The robot needs to avoid multiple obstacles to get through the corridor.

4.4.5 Test case 4: Obstacle corridor

This test case is the most complicated one. Here a corridor with a few different obstacles is placed between the robot and the goal point. The robot needs to fly through the window, over a wall of height 1.5m and then under a wall that goes as low as 1.6 m height, and then zig-zag between the remaining walls that block the left and right sides of the corridor. See figure 4.5. This test case was designed to show the ability of the robot to avoid multiple sequential obstacles from different directions.

Chapter 5

Results

This chapter describes the results of running the system on each of the test cases. In all cases the robot starts at position (x, y, z) = (0, 0, 0) and ends at a position 1 m above the ground (z = 1) where either the x or the y coordinate is zero. There is an exception with test case 2 where the robot start and goal points are at a height of 2m so that it is always father away from the ground than the velocity safety distance. In the following results, the closest obstacle distance is taken from the spherical matrix, which is made from the OctoMap, and is therefore not perfectly accurate. The robot managed to complete all of the test cases successfully. Additional runs can be seen in appendix C.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
1.930	2.096	0.391	0.266

Table 5.1: Obstacle distance and translational velocity for test case 0.

5.1 Test case 0: No obstacles

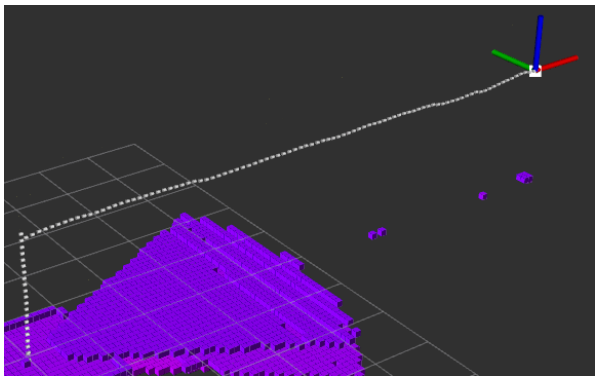


Figure 5.1: The trajectory and OctoMap generated by the robot while solving test case 0. Screen shot from Rviz.

The robot flew straight to the goal point after flying to the starting position 2m above the ground as expected, see figure 5.1. The velocity command was at around 0.4 m/s throughout the run which the actual velocity of the robot approached. See figure 5.2b. The nearest obstacle distance was always the ground since there were no other obstacles present. The height of the robot over the ground can be seen in 5.2c. The reason for the nearest obstacle distance in figure 5.2a having peaks that are larger than 2m is because of the ground directly beneath the robot not being registered in OctoMap at all times, see figure 5.1. Plots and numbers conserning the test can be seen in figure 5.2 and table 5.1.

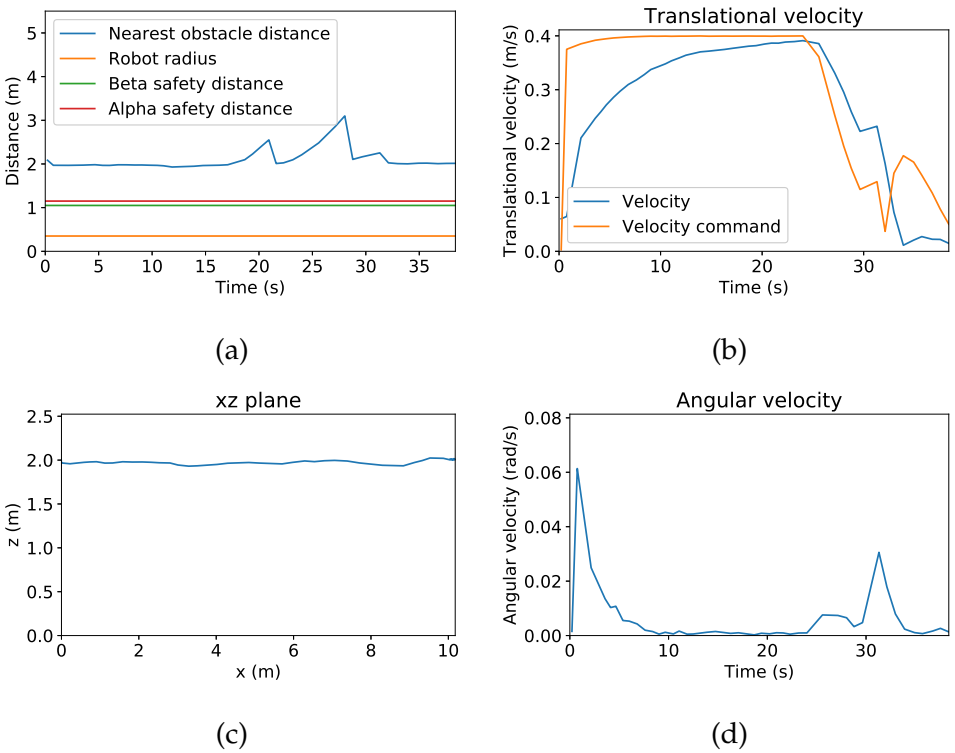


Figure 5.2: Test case 0. (a) Nearest obstacle distance at each time. (b) Translational velocity command and the actual translational velocity of the robot. (c) The trajectory of the robot in the xz plane. (d) Angular velocity of the robot.

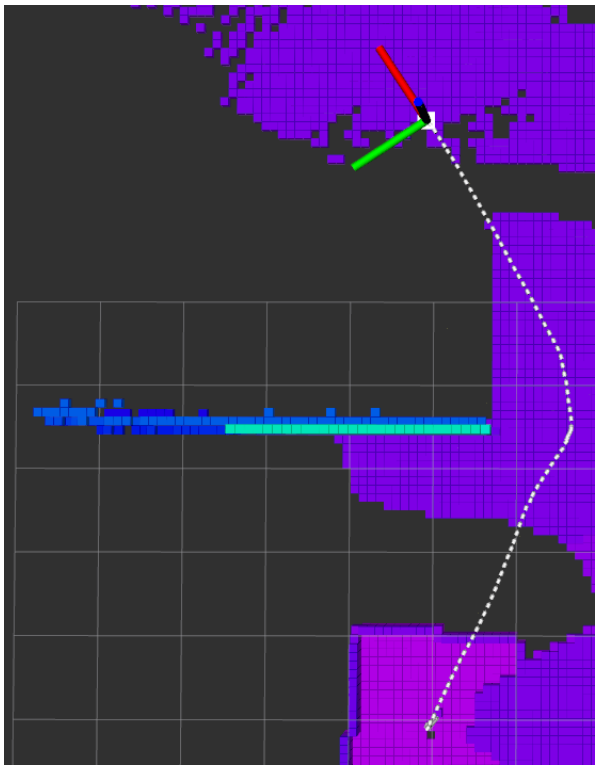


Figure 5.3: The trajectory and OctoMap generated by the robot while solving test case 1. Screen shot from Rviz.

5.2 Test case 1: Simple wall

The trajectory generated by the robot during this test can be seen in figure 5.3. The robot flew to the right of the obstacle, stopped and turned towards the goal point and then flew there. The reason it had to stop and turn is because it had to align with a new chosen direction when it had flown past the wall, which was more than 10° from the heading of the drone. The minimum shortest distance to an obstacle through the run was 0.818m from the robot center, 0.468m from the robot edge. Graphs and numbers concerning the test run can be seen in figure 5.4 and table 5.2. Notice how the translational velocity is low when the angular velocity is high by comparing figures 5.4b and 5.4d.

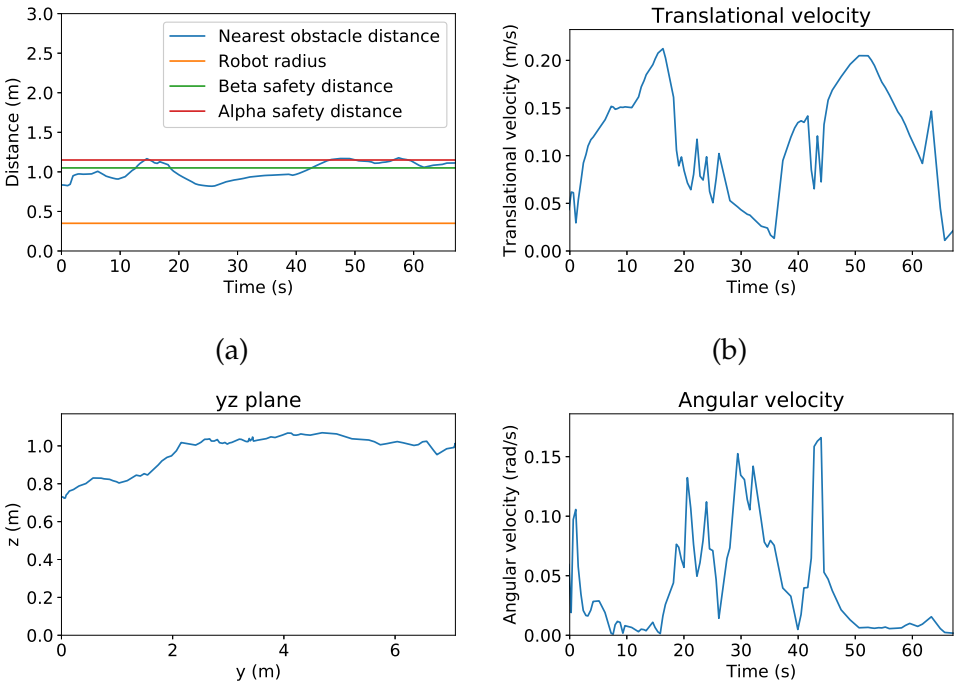


Figure 5.4: Test case 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.818	1.000	0.212	0.113

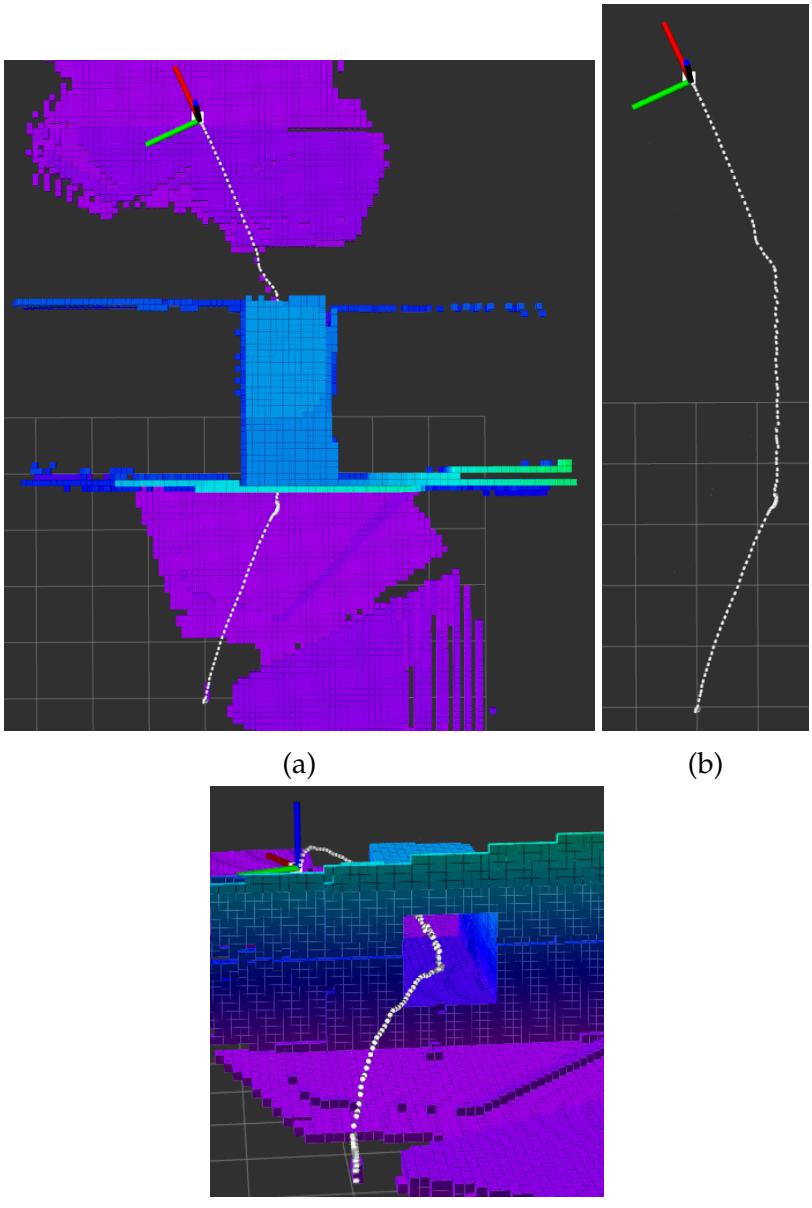
Table 5.2: Obstacle distance and translational velocity for test case 1.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.404	0.495	0.210	0.024

Table 5.3: Obstacle distance and translational velocity for test case 2.

5.3 Test case 2: Narrow passage

The trajectory of the robot for this test case can be seen in figure 5.5. The robot managed to turn and fly through the tunnel without colliding with the wall but the closest obstacle distance was 0.404 cm to the robot center, or 0.054 m to its edge. From figures 5.6a and 5.6b we can see that the robot was close to obstacles and its translational velocity was the lowest when it was inside the tunnel. The robot flew very slowly inside the tunnel since it was close to the wall and the velocity is close to zero when the distance to an obstacle is close to the robot radius, as can be seen in equation 2.21. Further details can be seen in figure 5.6 and table 5.3.



(a) (b)

(c)

Figure 5.5: The trajectory of the robot when completing test case 2. (a) Top view. (b) Top view without the generated OctoMap. (c) Side view.

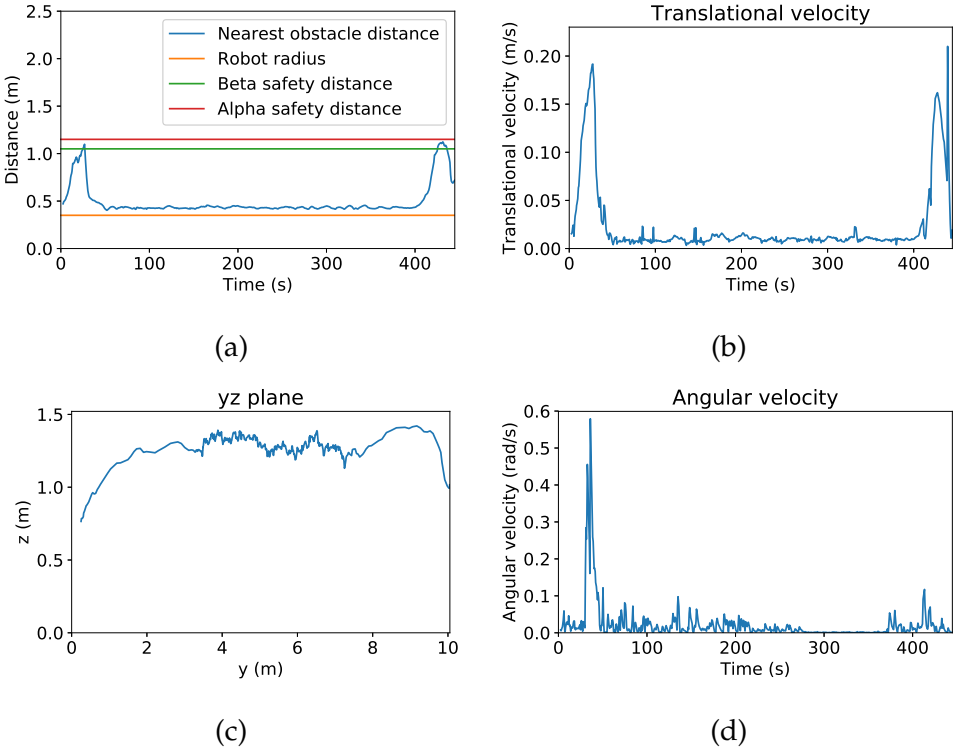


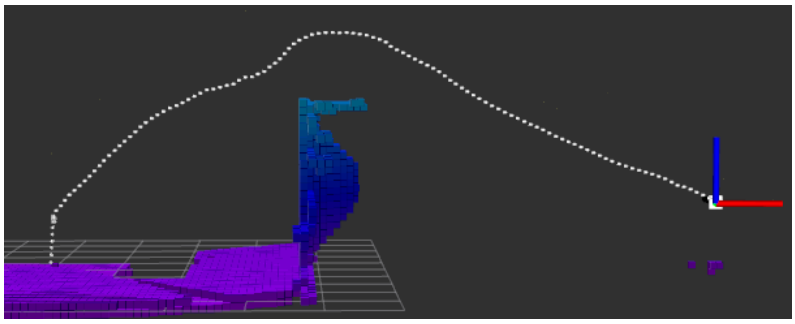
Figure 5.6: Test case 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.788	1.444	0.372	0.222

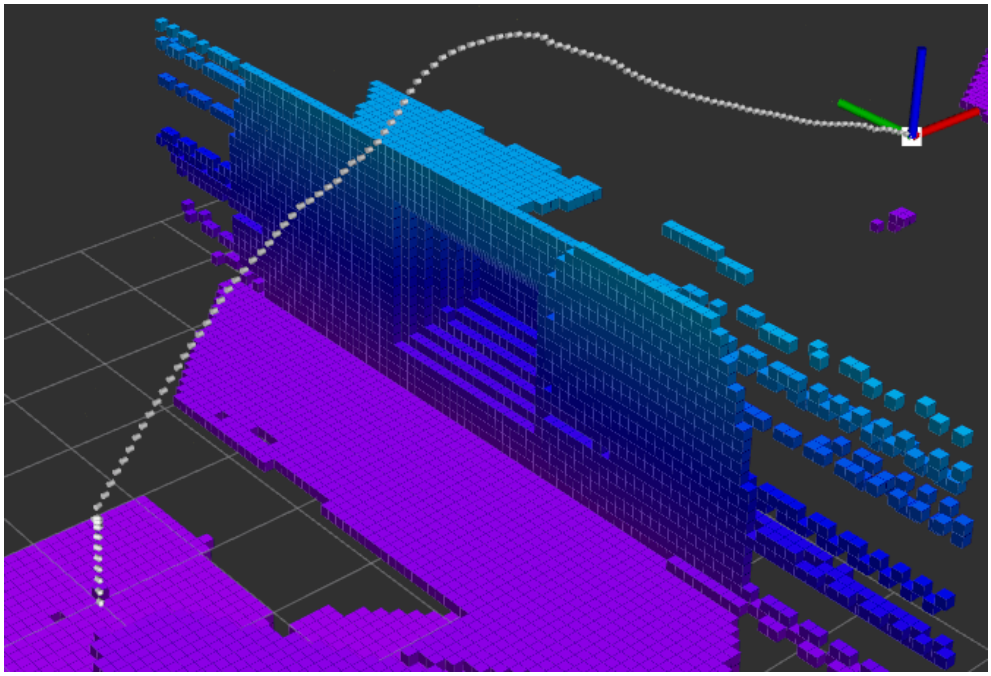
Table 5.4: Obstacle distance and translational velocity for test case 3.

5.4 Test case 3: Trap

The trajectory of the robot can be seen in figure 5.7. The nearest obstacle distance is 0.788 m from the robot center, 0.438 m from the robot edge. From figures 5.6a and 5.6b we can see how the translational velocity profile roughly follows the nearest obstacle distance, i.e. when the robot is closer to an obstacle it moves slower. Further details can be seen in figure 5.8 and table 5.4.



(a)



(b)

Figure 5.7: The trajectory of the robot after completing test case 3 seen from two different views.

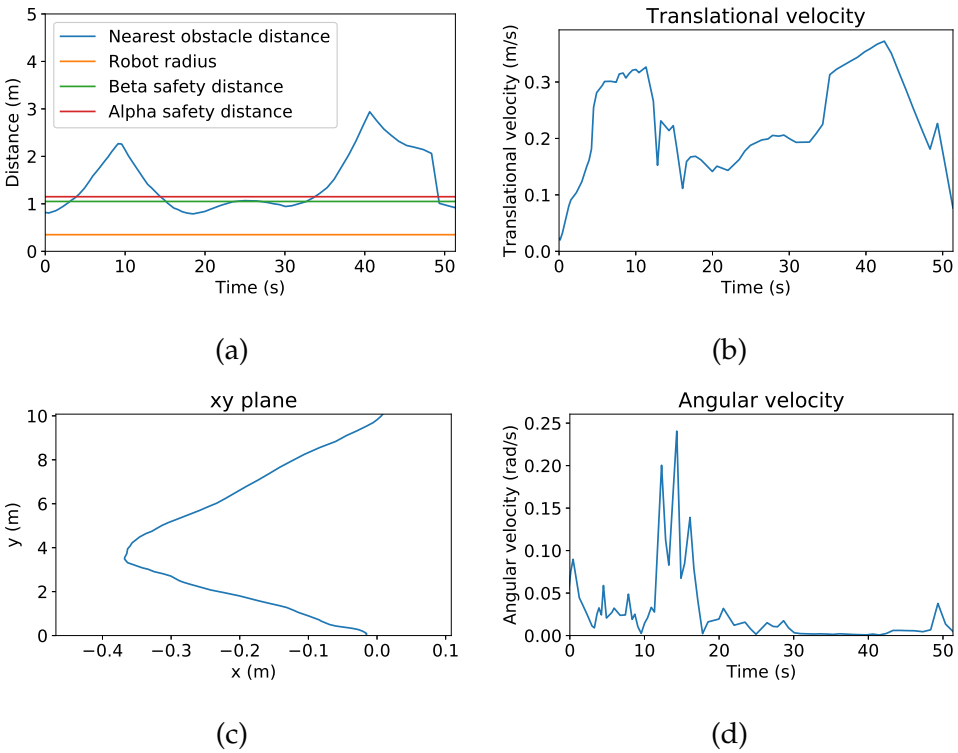


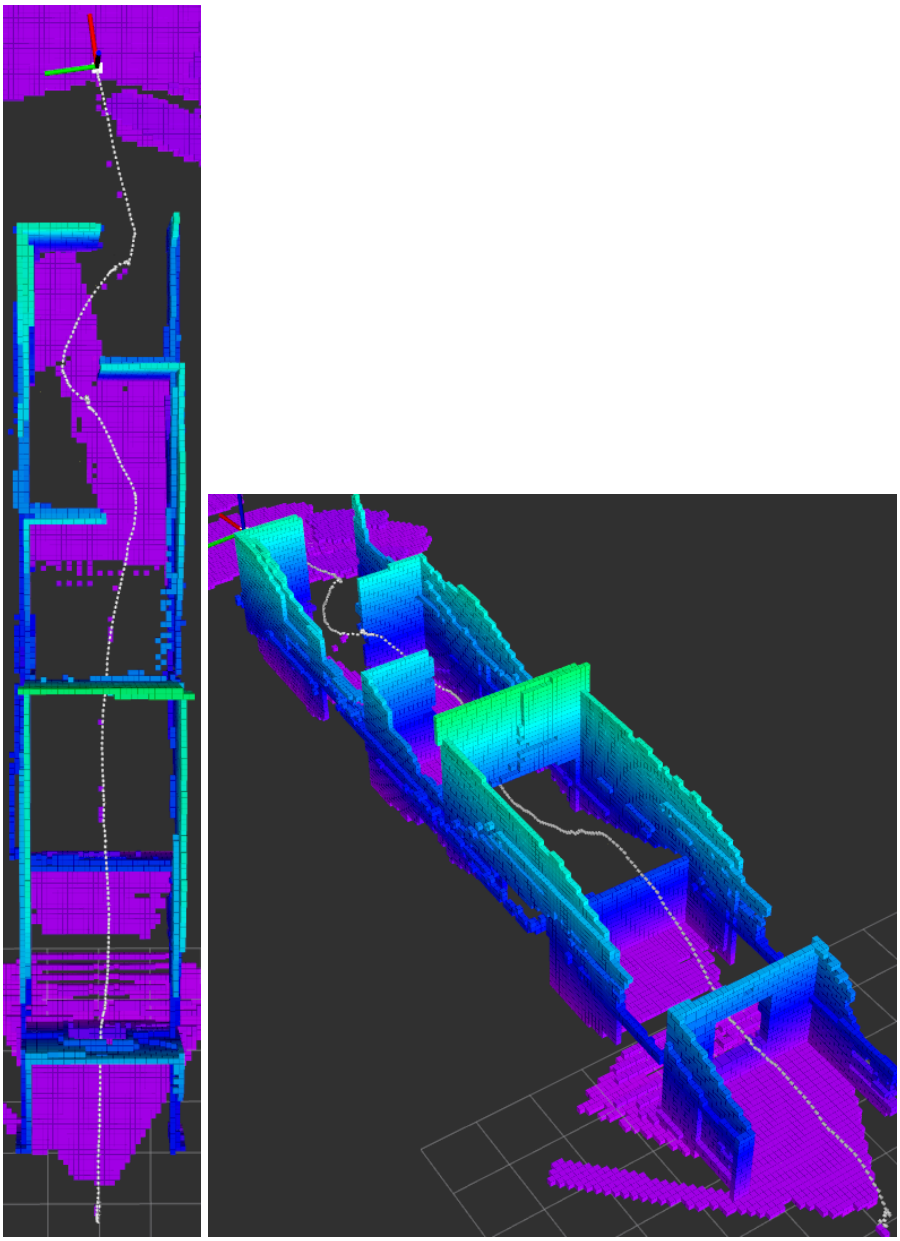
Figure 5.8: Test case 3. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane. The trajectory in the yz plane can be seen in figure 5.7b.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.467	0.738	0.303	0.085

Table 5.5: Obstacle distance and translational velocity for test case 4.

5.5 Test case 4: Obstacle corridor

The trajectory of the robot for this test case can be seen in figure 5.9 and 5.10c. The minimum distance to an obstacle through the run was 0.117m from the robot edge when it flew through the window in the beginning of the corridor. Further details can be seen in figure 5.10 and table 5.5.



(a) (b)

Figure 5.9: The trajectory of the robot after completing test case 4 seen from two different views.

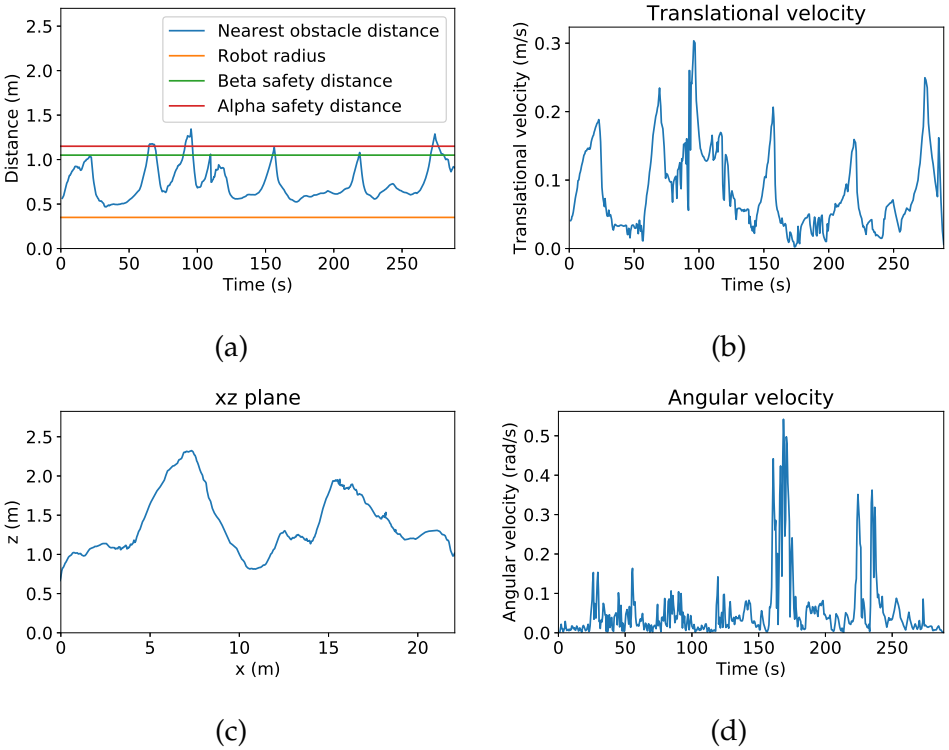


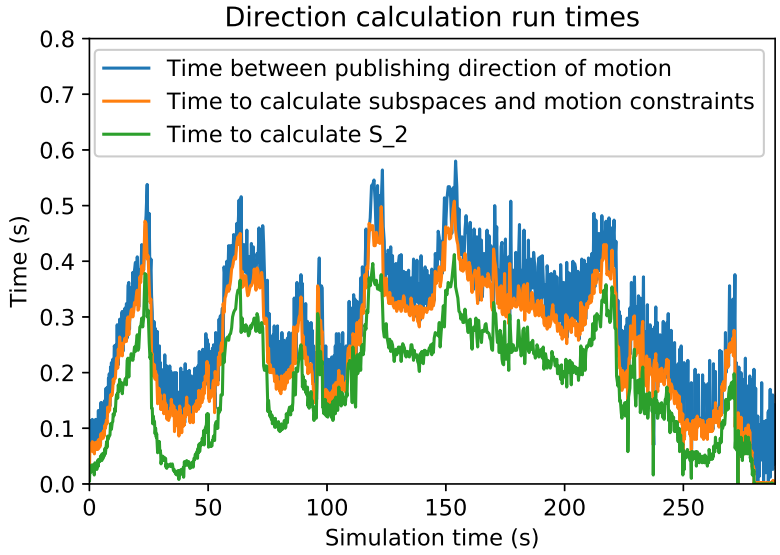
Figure 5.10: Test case 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xz plane.

Average run times (s)				
Updating the direction of motion	Calculation of subspaces and motion constraints	Calculation of S_2	Updating OctoMap	Calculation of the spherical matrix
0.212	0.180	0.124	0.624	0.109

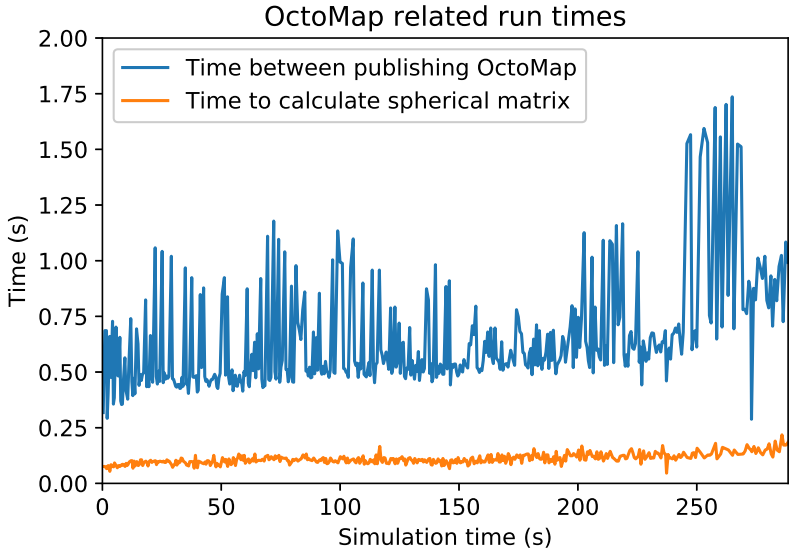
Table 5.6: Average run times for test case 4.

5.6 Run time measurements

Run times of different components of the system during a flight through the obstacle corridor, test case 4, can be seen in figure 5.11 and table 5.6. The maximum time between updating the direction of motion was 0.508s which we try to do at the update rate of the goal point which is at 20 Hz. The hardware specifications can be seen in table 4.1. As we can see in figure 5.11a, the time between updating the chosen direction (0.212) is almost entirely based on how long it takes to calculate the subspaces and motion constraints (0.180). Of those, the time to calculate S_2 is the biggest factor (0.124). From 5.11 we can furthermore conclude that the time between publishing OctoMap is the main source of delay in the system, and thereafter comes the calculation of S_2.
Since the time it takes to update OctoMap is generally longer than the time it takes to update the direction of motion, the selected direction of motion is often calculated more than once for each OctoMap. This is unnecessary since the spherical matrix is only calculated once for every OctoMap. A better implementation would calculate a new direction once for every spherical matrix. The spherical matrix could then be calculated more than once for every OctoMap since it changes depending on the location of the robot.



(a)



(b)

Figure 5.11: Run times from a run through test case 4, the obstacle corridor. (a) Shows the total time between updating the chosen direction of motion and the time it takes to calculate the subspaces and motion constraints. The time it takes to calculate S_2 is also shown. (b) Shows the time between publishing of OctoMap and the time it takes to calculate the spherical matrix from OctoMap. Those run times are not dependent on each other.

Chapter 6

Discussion

In this section the results of this thesis will be discussed.

6.1 Test cases

In this section the results of each test case will be discussed.

6.1.1 Test case 0: No obstacles

In this test case the robot flew straight to the goal point as expected. The robot was father than the velocity safety distance away from the closest obstacle through the run so the robot was given a velocity command equal to the chosen maximum velocity.

6.1.2 Test case 1: Simple wall

The robot avoided the obstacle here as expected. It flew to the right of the wall, stopped to turn towards the goal and then flew there.
When approaching the wall, the robot got within the set safety distance of the wall. The reason could be that our target direction belonged to the motion restriction of more than one subspace, and when we take the average of the dominant directions of the subspaces, it is possible that the resulting direction falls within some motion constraints, see figure 2.9.
Another reason could be because of the discretization of the spherical matrix. When we choose what points are on the boundary of S_2, it is

possible that we choose a direction that is up to half the angular resolution of the spherical matrix away from the boundary. This means our chosen direction might be up to 3° within the boundary in our case and in that way bring us closer to the obstacle. This could be fixed by increasing the angular resolution or choosing $D_{s,alpha} > D_{s,beta}$ with enough difference between the two safety distances so that the robot doesn't get within $D_{s,beta}$.

6.1.3 Test case 2: Narrow passage

In this test case, the robot moved through a narrow tunnel without a collision. During the test, the minimum distance from the robot edge to an obstacle was only 0.053m. In the real world, where the robot can be less stable and have a worse positioning system this could possibly result in a collision. In a perfect run through the 1.2m x 1.2m tunnel, the nearest obstacle distance would always be 0.25m.

Since the obstacles surrounded the robot while inside the tunnel the special case of when the motion constraint of a quadrant covers all possible directions of motion influenced the motion of the robot. In that case the negative of the average of all directions where the obstacle distance is within a safety distance is chosen as the dominant direction of that quadrant. See equation 6.1 below and section 3.5.

Moreover, since the robot flew close to the tunnel wall the robot got inside the critical safety distance which also influenced the motion. Then the robot moves slowly in the opposite direction of the closest obstacle distance, as described in section 3.6.

To improve the result in this test case those special cases would need to be looked further into. One problem is that the final direction of motion is computed as the average of four directions that each represent a different portion of the space, which are not necessarily equal in size, see section 3.5. This essentially means that when we have a relatively long safety distance in a narrow tunnel, and there are no free directions of motion in any of the four quadrants, the direction of motion is chosen as

$$-\sum_{G} \frac{\sum_i u^G_{close,i}}{n_{G,close}} \tag{6.1}$$

where $G \in \{TL, DL, TR, DR\}$, $u^G_{close,i} \in G$ is a direction in which the distance is shorter than $D_{s,\beta}$ and $n_{G,close}$ is the number of such directions in quadrant G.

In this case the direction of motion should perhaps rather be chosen either as the opposite of the direction of the closest obstacle or the opposite of the average of all the directions within the β-safety distance:

$$-\frac{\sum_i u_{close,i}}{n_{close}} \tag{6.2}$$

where $u_{close,i}$ is a direction in which the distance is lower than $D_{s,\beta}$ and n_{close} the total number of such directions.

6.1.4 Test case 3: Trap

The robot managed to fly over the wall in this case. This shows the robot's ability to avoid a trap and fly over an obstacle. Similarly to test case 1, the robot got within the safety distance of the wall. If the trap would be deeper or farther away, the robot would fly towards the inside of the trap until it would see its bottom.

6.1.5 Test case 4: Obstacle corridor

Here the robot managed to move between close obstacles through the corridor. It shows that the robot can avoid multiple subsequent obstacles with limited space to maneuver.

6.2 Advantages and limitations of the method

The advantages of this method compared to other methods are as described in the background section 2; the ability to avoid trap situations as seen in test case 3, relatively free of oscillations in narrow spaces and the ability of passing through narrow passages as seen in test case 2.
Another advantage of the method according to [21] is that it performs better in open spaces than the nearness diagram described in section 2.1.5. In this project only obstacle points within 5 meter distance are considered to reduce computing cost since we loop through all the OctoMap voxels inside a bounding box, and also because the 3D camera has a limited range. This means that the motion can be sub optimal when the robot needs to avoid obstacles that are farther away than 5 meters, i.e. the motion in open spaces is sub optimal. This could however be acceptable if we have a global planner that sends way points

with maximum 5 m in between.
Another limitation of this method is that it counts unknown space as free. This is partially tackled by the robot always turning in the direction that it wants to go, but it could still fly into obstacles that are above or below the FOV of the camera and are marked as unknown space. An experiment was made where unknown space was set as occupied space, which mostly worked, but resulted sometimes in strange motion for the robot, when it tried to avoid close unknown space that was actually free.
The computational cost is another downside of the method, described further in section 5.6. However, computational efficiency was not the aim of the project.
The robot travels quite slowly when it is close to obstacles. A physical drone could have some limitations on how slowly it can move reliably, as well as on battery life. In cases similar to the narrow tunnel this could be problematic. It is possible to increase the speed of the drone but that would mean that it would fly closer to obstacles in some cases because of its inertia and the time it takes to calculate a new direction of motion.

Chapter 7

Conclusion

In this project an obstacle avoidance system was developed based on 3D ORM by [36] for a drone equipped with a 3D camera, a lidar and distance sensors facing up and downwards. OctoMap was used to represent the environment to deal with the limited FOV of the sensors. OctoMap makes a probabilistic map of the environment from the sensor data which can deal with noise from sensors and a dynamic environment [17]. A limitation with using OctoMap is that only obstacles in a limited range could be considered for the obstacle avoidance since looping through the whole generated map is not feasible computationally. Four test cases were used to show the functionality of the system. The robot managed to complete all the test cases without a collision which shows the ability of the system to fulfill its purpose.

7.1 Future work

Since there is sub optimal motion in open spaces because of limited range of the 3D camera and the way we use OctoMap as described in section 6.2 the method could be improved by using a 2D obstacle avoidance method with the lidar when the goal point and the obstacles are far away. A problem with that is that when the robot moves forward, it tilts a little down in the front, which means the lidar will not align perfectly with the ground which has a larger effect on the result when we use values far away from the drone.
The method presented approximates the robot as a sphere. Since this is usually not true in reality it would be interesting to expand the method to take into account the shape of the robot.

Since with the current method the robot could technically fly towards obstacles that are marked as unknown space, it could be expanded by tackling that problem.
The calculation of the motion constraints and OctoMap seem to be the main sources of delay in the system, see section 5.6. Therefore, optimization of the code with regard to those components would be beneficial to be able to better deal with a dynamic environment.

Bibliography

[1] B. Bellekens, S. Vanneste, and M. Weyn, "3DVFH+: Real-Time Three-Dimensional Obstacle Avoidance Using an Octomap," 2014, Retrieved from: http://ceur-ws.org/Vol-1319/morse14_paper_08.pdf

[2] J. Borenstein and Y. Koren, "High-speed obstacle avoidance for mobile robots," Proceedings IEEE International Symposium on Intelligent Control 1988, Arlington, VA, 1988, pp. 382-384.

[3] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots," in IEEE Transactions on Systems, Man, and Cybernetics, vol. 19, no. 5, pp. 1179-1187, Sept.-Oct. 1989.

[4] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots in cluttered environments," Proceedings., IEEE International Conference on Robotics and Automation, Cincinnati, OH, 1990, pp. 572-577 vol.1.

[5] O. Brock and O. Khatib, "High-speed navigation using the global dynamic window approach," Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), Detroit, MI, 1999, pp. 341-346 vol.1.

[6] H. Chao, Y. Gu and M. Napolitano, "A survey of optical flow techniques for uav navigation applications," International Conference on Unmanned Aircraft Systems, Grand Hyatt Atlanta, Atlanta, GA, 2013, pp. 710–716

[7] D. Claes, D. Hennes, K. Tuyls and W. Meeussen, "Collision avoidance under bounded localization uncertainty," 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vilamoura, 2012, pp. 1192-1198.

[8] G. C. S. Cruz and P. M. M. Encarnação, "Obstacle Avoidance for Unmanned Aerial Vehicles, " Journal of Intelligent & Robotic Systems, 2012, pp 203–217, vol. 65.

[9] Devantech SRF08 UltraSonic Ranger, Accessed 11. June 2018, http://coecsl.ece.illinois.edu/ge423/devantechsrf08ultrasonicranger.pdf

[10] D. Duberg, "Safe Navigation of a Tele-operated Unmanned Aerial Vehicle", Master of Science Thesis, KTH, 2017.

[11] J. W. Durham and F. Bullo, "Smooth Nearness-Diagram Navigation," 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, Nice, 2008, pp. 690-695.

[12] I. Fantoni and G. Sanahuja, "Optic Flow-Based Control and Navigation of Mini Aerial Vehicles," AerospaceLab, 2014, p. 1-9.

[13] P. Fiorini and Z. Shiller. "Motion Planning in Dynamic Environments Using Velocity Obstacles," The International Journal of Robotics Research, Vol 17, Issue 7, pp. 760 - 772, July 1998.

[14] D. Fox, W. Burgard and S. Thrun, "The dynamic window approach to collision avoidance," in IEEE Robotics & Automation Magazine, vol. 4, no. 1, pp. 23-33, Mar. 1997.

[15] M. S. Geyer and E. N. Johnson, " 3D Obstacle Avoidance in Adversarial Environments for Unmanned Aerial Vehicles," AIAA Guidance, Navigation, and Control Conference and Exhibit 21 - 24 August 2006, Keystone, Colorado, 2006

[16] Hokuyo Automatic, UST-20LX Specification, Retrieved 11. June 2018 from http://www.technical-avenue.com/Portals/0/datasheet/Hokuyo/UST-20LX_Specifications.pdf

[17] A. Hornung,. K.M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees" in Autonomous Robots, 2013; DOI: 10.1007/s10514-012-9321-0. Software available at http://octomap.github.com.

[18] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," Proceedings. 1985 IEEE International Conference on Robotics and Automation, 1985, pp. 500-505.

[19] J. O. Kim and P. K. Khosla, "Real-time obstacle avoidance using harmonic potential functions," in IEEE Transactions on Robotics and Automation, vol. 8, no. 3, pp. 338-349, Jun 1992.

[20] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," Proceedings. 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, 1991, pp. 1398-1404 vol.2.

[21] J. Minguez, "The obstacle-restriction method for robot obstacle avoidance in difficult environments," 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005, pp. 2284-2290.

[22] J. Minguez, F. Lamiraux and JP. Laumond, "Motion Planning and Obstacle Avoidance," In: Siciliano B., Khatib O. (eds) Springer Handbook of Robotics. Springer, Cham, 2016

[23] J. Minguez and L. Montano, "Nearness diagram navigation (ND): a new real time collision avoidance approach," Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113), Takamatsu, 2000, pp. 2094-2100 vol.3.

[24] J. Minguez and L. Montano, "Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios," in IEEE Transactions on Robotics and Automation, vol. 20, no. 1, pp. 45-59, Feb. 2004.

[25] J. Minguez, J. Osuna and L. Montano, "A "divide and conquer" strategy based on situations to achieve reactive collision avoidance in troublesome scenarios," Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on, 2004, pp. 3855-3862 Vol.4.

[26] National Science Foundation, "Small, Unmanned Aircraft Search for Survivors in Katrina Wreckage," News Release 05-160, September 14, 2005, Retrieved 21. June 2018 from https://www.nsf.gov/news/news_summ.jsp?cntn_id=104453

[27] PrimeSense, PrimeSense 3D Sensors, Retrieved 11. June 2018 from http://www.i3du.gr/pdf/primesense.pdf

[28] P. Saranrittichai, N. Niparnan and A. Sudsang, "Robust local obstacle avoidance for mobile robot based on Dynamic Window approach," 2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, Krabi, 2013, pp. 1-4.

[29] R. A. Sasongko and S. S. Rawikara, "3D obstacle avoidance system using ellipsoid geometry," 2016 International Conference on Unmanned Aircraft Systems (ICUAS), Arlington, VA, 2016, pp. 562-571.

[30] J. Serres and F. Ruffier, "Optic flow-based collision-free strategies: From insects to robots," Arthropod Structure & Development, vol. 46, Issue 5, sep. 2017, pp. 703-717

[31] S. Scherer, S. Singh, L. Chamberlain and M. Elgersma, "Flying Fast and Low Among Obstacles: Methodology and Experiments," The International Journal of Robotics Research, 2008, pp. 549 - 574, vol 27, no 5.

[32] Terabee, TeraRanger One Specification Sheet, Retrieved 11. June 2018 from https://www.terabee.com/wp-content/uploads/2017/09/TeraRanger-One-Specification-Sheet1-1.pdf

[33] I. Ulrich and J. Borenstein, "VFH+: reliable obstacle avoidance for fast mobile robots," Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146), Leuven, 1998, pp. 1572-1577 vol.2.

[34] I. Ulrich and J. Borenstein, "VFH*: local obstacle avoidance with look-ahead verification," Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), San Francisco, CA, 2000, pp. 2505-2511 vol.3.

[35] J. van den Berg, D. Wilkie, S. J. Guy, M. Niethammer and D. Manocha, "LQG-obstacles: Feedback control with collision avoidance for mobile robots with motion and sensing uncertainty," 2012 IEEE International Conference on Robotics and Automation, Saint Paul, MN, 2012, pp. 346-353.

[36] D. Vikenmark, "The Obstacle-Restriction Method (ORM) for Reactive Obstacle Avoidance in Difficult Scenarios in Three-Dimensional Workspaces," Master of Science Thesis, KTH, 2006.

[37] Y. Wang, H. Liu and Q. Tao, "A realistic method for real-time obstacle avoidance without the Calculation of Cspace Obstacles," 2012, retrieved from: http://www.cs.cmu.edu/ hanliu/papers/Robotica.pdf

[38] Wikipedia, "History of unmanned aerial vehicles", Retrieved 20. June from https://en.wikipedia.org/wiki/History_of_unmanned_aerial_vehicles

[39] D.W. Yoo, D.Y. Won and M.J. Tahk, "Optical Flow Based Collision Avoidance of Multi-Rotor UAVs in Urban Environments," International Journal of Aeronautical and Space Sciences, 2011, pp.252-259, vol. 12, no.3.

[40] L. Zhao, "3D Obstacle Avoidance for Unmanned Autonomous System (UAS)," UNLV Theses, Dissertations, Professional Papers, and Capstones, 2507, 2015.

[41] S. Zingg, D. Scaramuzza, S. Weiss and R. Siegwart, "MAV navigation through indoor corridors using optical flow," 2010 IEEE International Conference on Robotics and Automation, Anchorage, AK, 2010, pp. 3361-3368.

Appendix A

Sustainability, Ethics and Social Impact

With further development of drones they become more popular which can be bad for sustainability since consumer UAVs are often mostly made from plastic which takes a very long time to break down in nature. However, good obstacle avoidance can also have the effect that drones have fewer collisions which increases the lifetime of the product and is good for sustainability.
In many cases drones can be used instead of helicopters for surveillance and photography, for example, which is good for sustainability since they are often powered by electricity and use less energy than fossil fueled helicopters. Then again, most consumer drones are used by people who would never spend money on a helicopter for taking videos.
Drones have been used in search and rescue missions where they can look for people in dangerous environments. They were for example used to search for survivors after hurricane Katrina [26]. They can also be used to deliver goods to unaccessible places after natural disasters. Drones have however also been widely used in warfare, which fueled most of their original development [38]. Autonomous obstacle avoidance contributes to increased autonomy of drones and the results of this thesis therefore interesting to the developers of such drones. An ethical issue arises when autonomous drones crash and cause damage since it is hard to hold a computer program accountable for those damages.
When drones begin to be able to do some work autonomously, such

as delivering products or surveillance it could make some jobs unnecessary. However the first jobs that are taken over by technological advancements are usually the simplest jobs, which means that the human workforce can focus on something more complicated. Technology also creates new kinds of jobs. Autonomous drones are also perhaps not the most entertaining colleagues since they can not socialize.

Appendix B

Algorithm to check if a tunnel is blocked

When we check if a point is locally reachable from the robot location, we construct a tunnel between the robot and the point and check if it is completely blocked in C-space. In this appendix we show the algorithm used to check if a tunnel is blocked.

Algorithm 1 Algorithm to check if tunnel is blocked

Input: C-space matrix, $Cspace$, of dimensions $cspace\_length * cspace\_width * cspace\_width$. Occupied points and points that are outside the tunnel but in the matrix because it is rectangular have already been marked as occupied.
Output: *false* if the tunnel is completely blocked, *true* otherwise.

Initialize each element in $edge\_points$ of size $cspace\_length * cspace\_width * cspace\_width$ to 0.
\\ $edge\_points$ mark the edge of the explored space in the C-space.
$front\_edge \leftarrow 0$
$cspace\_half\_width \leftarrow floor(cspace\_width)$
$edge\_points[0][cspace\_half\_width][cspace\_half\_width] \leftarrow 1$
\\ 0 in $Cspace$ means it is free, 1 means it is occupied, 2 means it is reachable.
loop
    $new\_front\_edge \leftarrow -1$
    for $y \leftarrow 0$ to $cspace\_width$ **do**
        for $z \leftarrow 0$ to $cspace\_width$ **do**
            if $edge\_points[front\_edge + 1][y][z] = 1$ **then**
                if $Cspace[front\_edge + 1][y][z] = 0$ **then**
                    \\ This means we can move forward
                    \\ in the tunnel.
                    $Cspace[front\_edge + 1][y][z] \leftarrow 2$
                    $edge\_points[front\_edge + 1][y][z] \leftarrow 1$
                    $new\_front\_edge \leftarrow front\_edge + 1$
                    go to next
                else
                    \\ Otherwise, go through surrounding points.
                    if $y < cspace\_width - 1$ **then**
                        if $Cspace[front\_edge][y + 1][z] = 0$ **then**
                            $Cspace[front\_edge][y + 1][z] \leftarrow 2$
                            $edge\_points[front\_edge][y + 1][z] \leftarrow 1$
                            $new\_front\_edge \leftarrow front\_edge$

```
26:                     if z < cspace_width − 1 then
27:                         if Cspace[front_edge][y][z + 1] = 0 then
28:                             Cspace[front_edge][y][z + 1] ← 2
29:                             edge_points[front_edge][y][z + 1] ← 1
30:                             new_front_edge ← front_edge
31:                     if z > 0 then
32:                         if Cspace[front_edge][y][z − 1] = 0 then
33:                             Cspace[front_edge][y][z − 1] ← 2
34:                             edge_points[front_edge][y][z − 1] ← 1
35:                             new_front_edge ← front_edge
36:                     if y > 0 then
37:                         if Cspace[front_edge][y − 1][z] = 0 then
38:                             Cspace[front_edge][y − 1][z] ← 2
39:                             edge_points[front_edge][y − 1][z] ← 1
40:                             new_front_edge ← front_edge
41:     next:
42:     if new_front_edge = -1 then
43:         \\ Now we need to move backwards
44:         if front_edge ≤ 0 then
45:             \\ Now we are back where we started
46:             return false
47:         \\ Count if all points in cross section are either reachable
48:         \\ or occupied. If they are, we cannot go forward
49:         \\ and the tunnel is blocked.
50:         count ← 0
51:         for y ← 0 to cspace_width do
52:             for z ← 0 to cspace_width do
53:                 if Cspace[front_edge][y][z] > 0 then
54:                     countx ← countx + 1
55:         if count = cspace_width ∗ cspace_width then
56:             return false
57:         front_edge ← front_edge − 1
58:     elsefront_edge ← new_front_edge
59:     if front_edge = cspace_length − 1 then
60:         \\ Now we have reached our goal.
61:         return true
```

Appendix C

Extra runs

In this appendix, results from four extra runs of each test case are shown.

C.1 Test case 1: No obstacles

C.1.1 Run 1

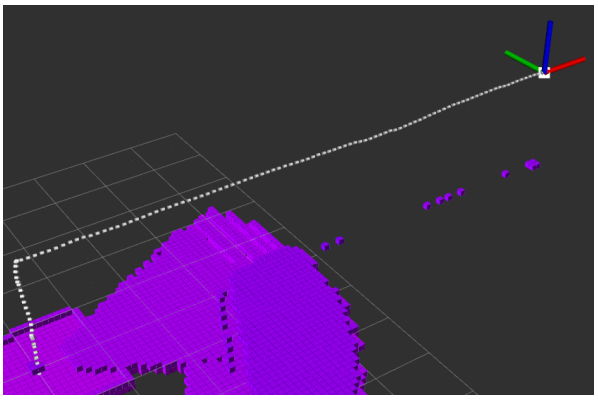


Figure C.1: The trajectory and OctoMap generated by the robot while solving test case 0. Extra run 1. Screen shot from Rviz.

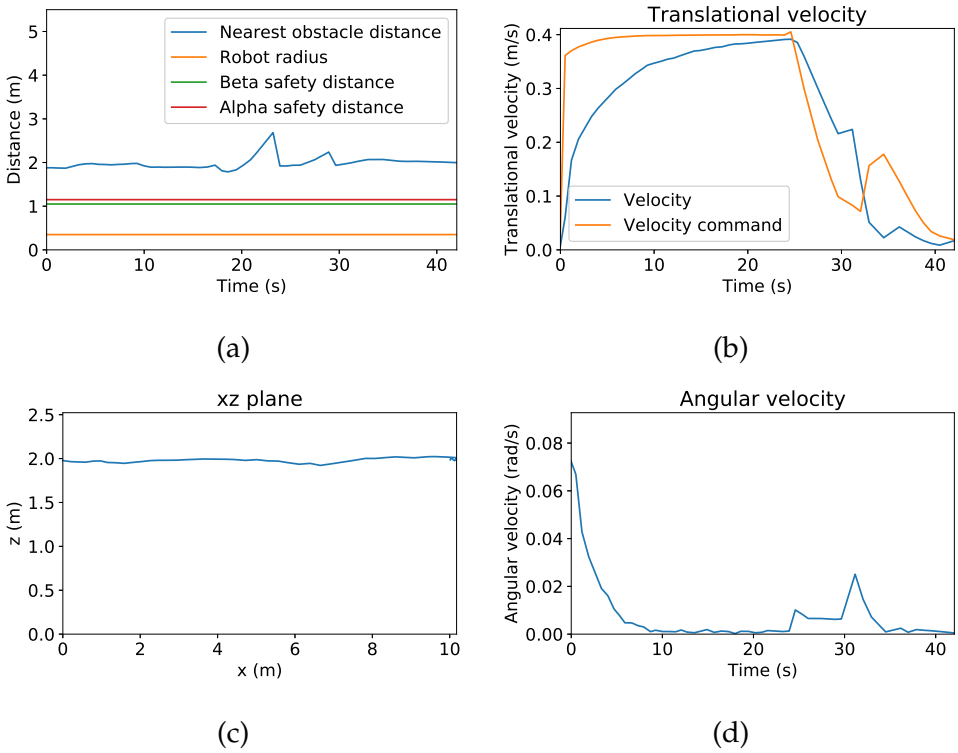


Figure C.2: Test case 0. Extra run 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
1.786	1.973	0.392	0.260

Table C.1: Obstacle distance and translational velocity for test case 0. Extra run 1.

C.1.2 Run 2

Here, the obstacle distance appears to be longer than it really is in figure C.4a, because we don't have the ground beneath the robot marked in OctoMap the whole time. The distance to the ground is stored in the spherical matrix directly from the sonar readings when this happens but unfortunatly that is not included in figure C.4a.

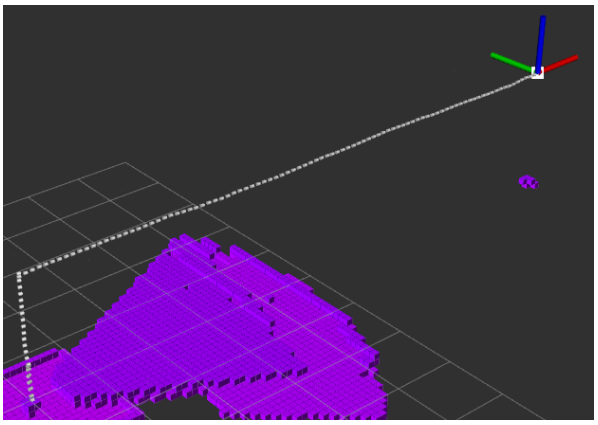


Figure C.3: The trajectory and OctoMap generated by the robot while solving test case 0. Extra run 2. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
1.956	N/A	0.391	0.285

Table C.2: Obstacle distance and translational velocity for test case 0. Extra run 2.

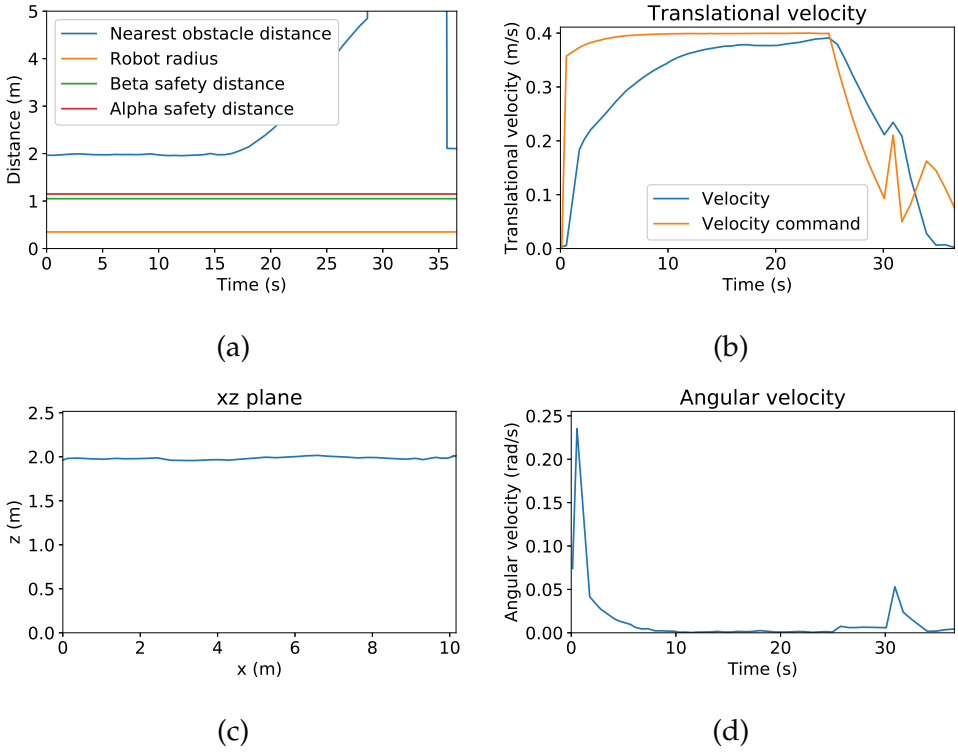


Figure C.4: Test case 0. Extra run 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.1.3 Run 3

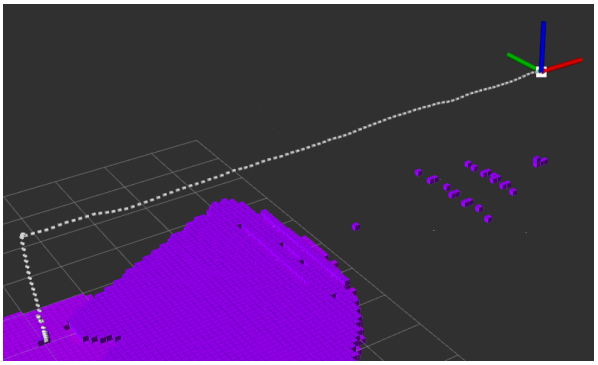


Figure C.5: The trajectory and OctoMap generated by the robot while solving test case 0. Extra run 3. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
1.758	1.916	0.394	0.272

Table C.3: Obstacle distance and translational velocity for test case 0. Extra run 3.

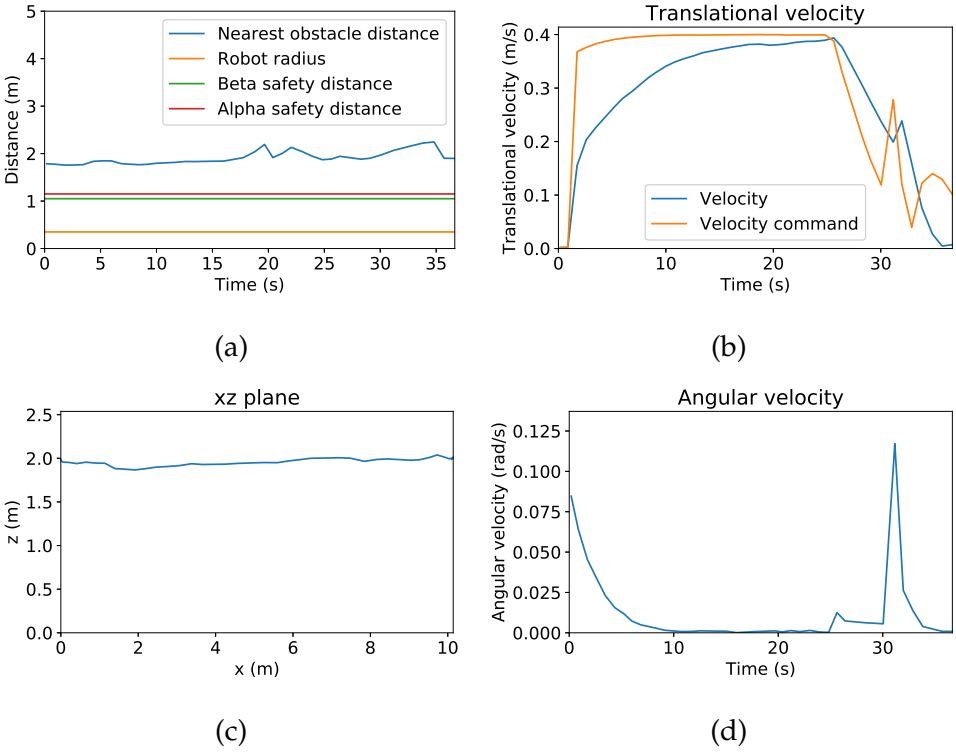


Figure C.6: Test case 0. Extra run 3. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.1.4 Run 4

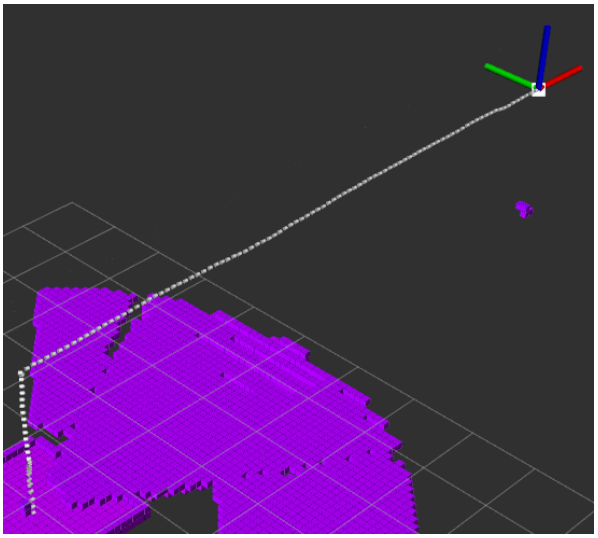


Figure C.7: The trajectory and OctoMap generated by the robot while solving test case 0. Extra run 4. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
2.001	N/A	0.389	0.284

Table C.4: Obstacle distance and translational velocity for test case 0. Extra run 4.

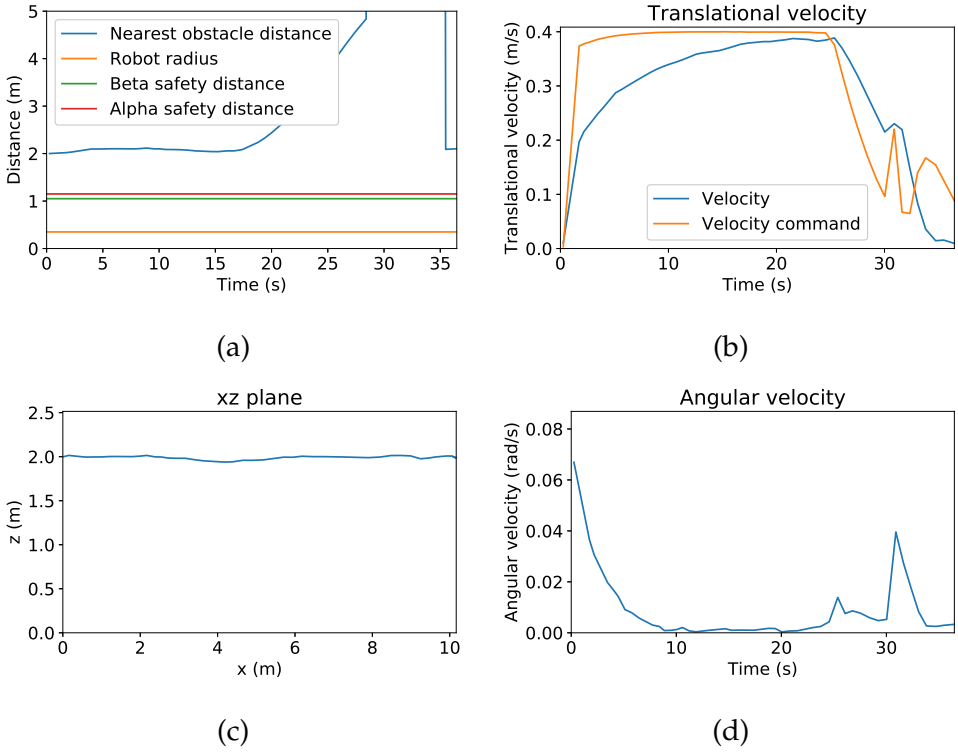


Figure C.8: Test case 0. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.2 Test case 1: Simple wall

C.2.1 Run 1

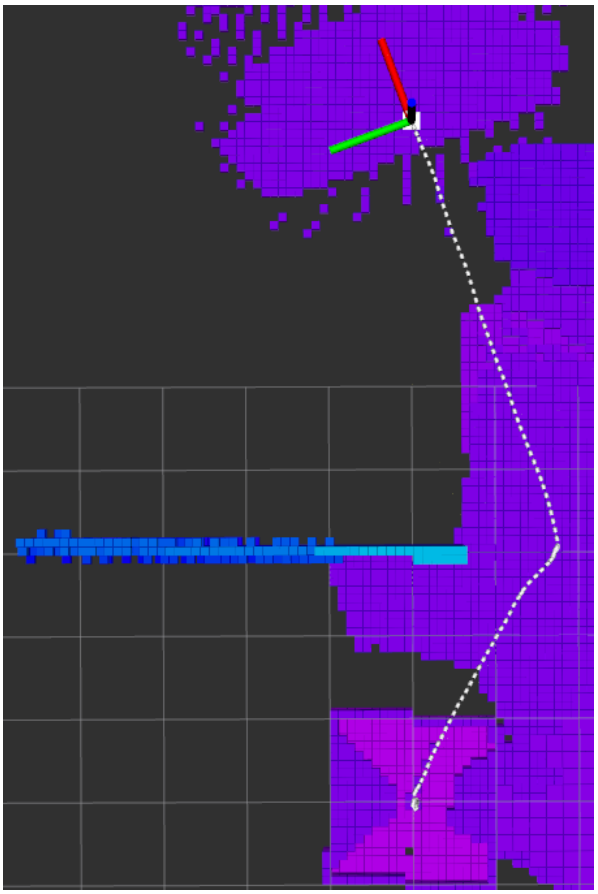


Figure C.9: The trajectory and OctoMap generated by the robot while solving test case 1. Extra run 1. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.737	1.025	0.312	0.141

Table C.5: Obstacle distance and translational velocity for test case 1. Extra run 1.

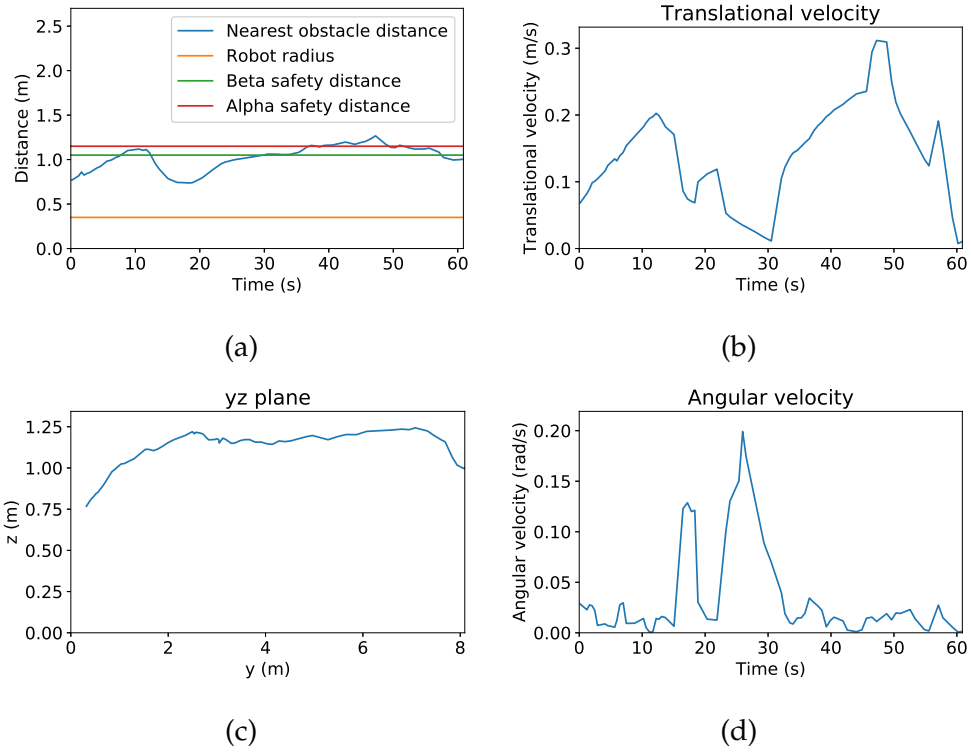


Figure C.10: Test case 1. Extra run 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.2.2 Run 2

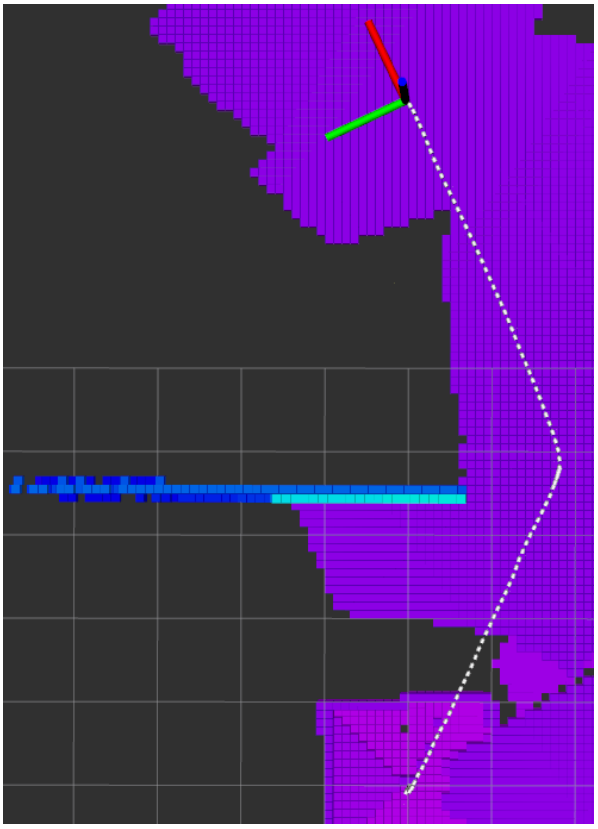


Figure C.11: The trajectory and OctoMap generated by the robot while solving test case 1. Extra run 2. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.830	1.030	0.335	0.152

Table C.6: Obstacle distance and translational velocity for test case 1. Extra run 2.

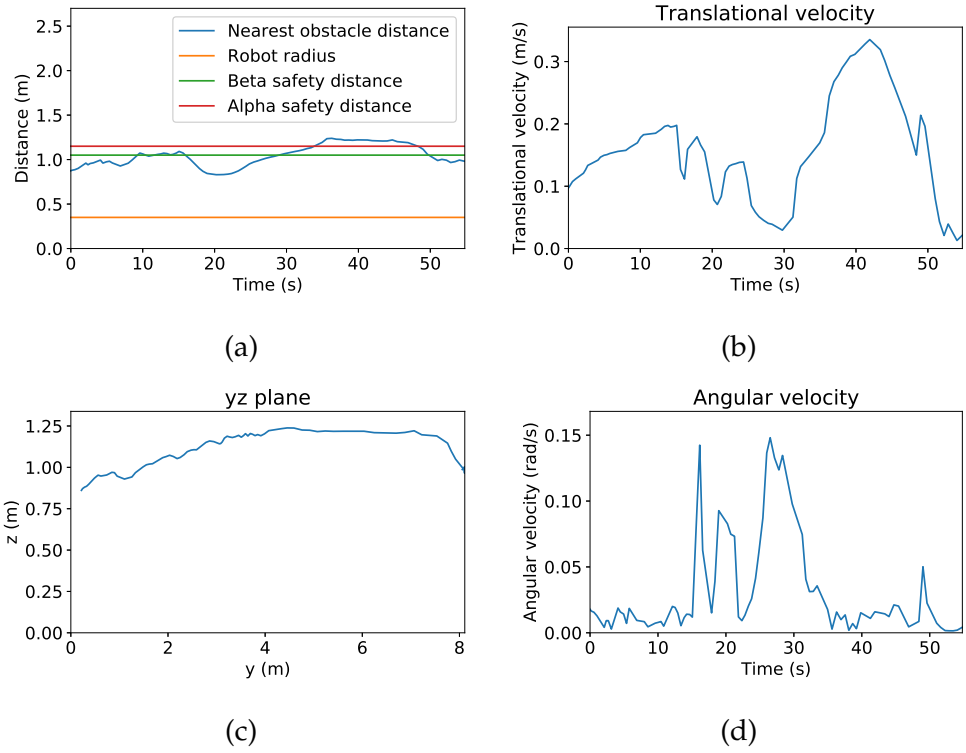


Figure C.12: Test case 1. Extra run 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.2.3 Run 3

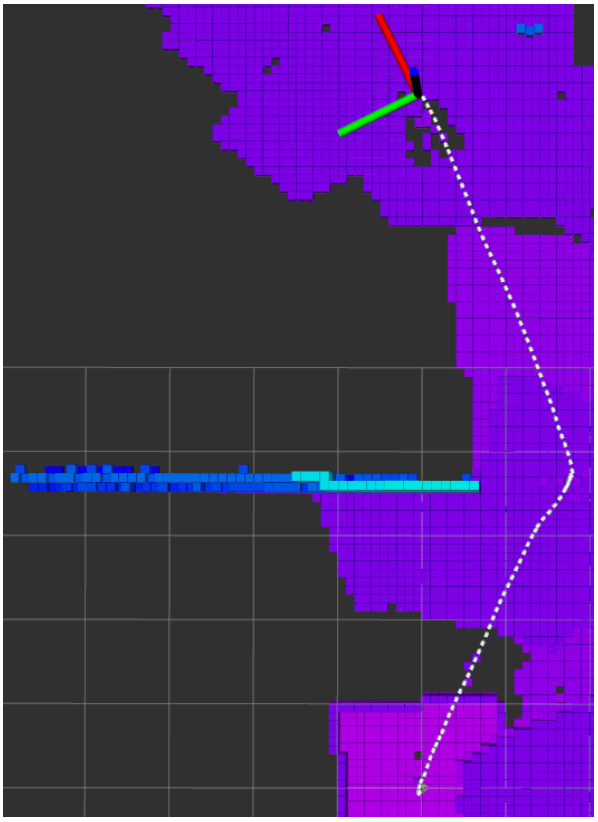
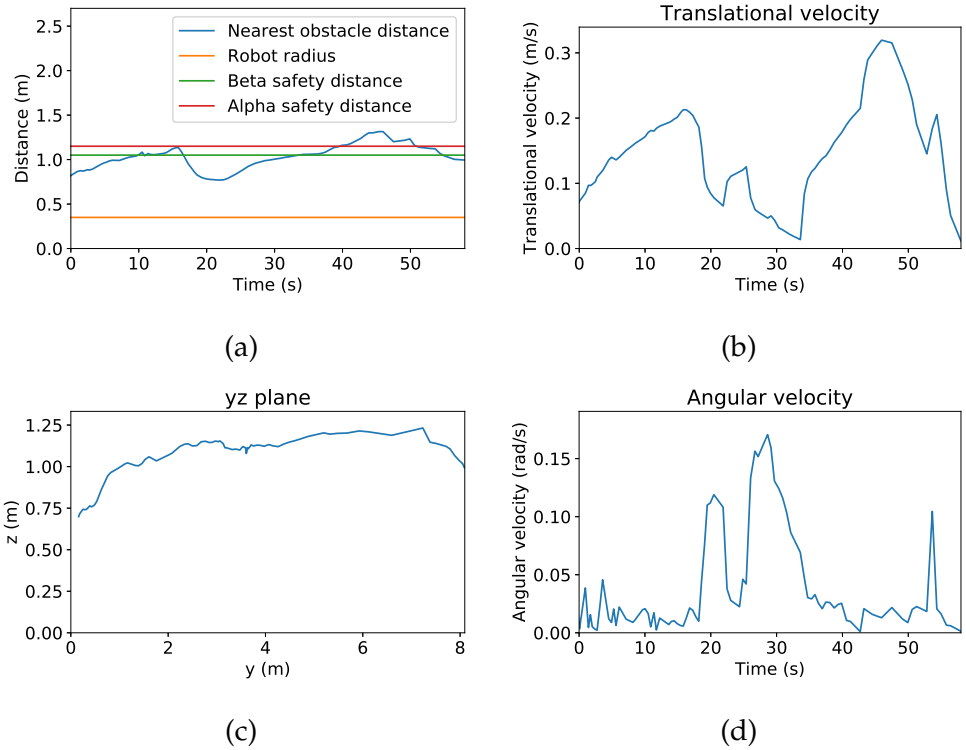


Figure C.13: The trajectory and OctoMap generated by the robot while solving test case 1. Extra run 3. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.769	1.019	0.319	0.145

Table C.7: Obstacle distance and translational velocity for test case 1. Extra run 3.



(a) (b)

(c) (d)

Figure C.14: Test case 1. Extra run 3. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.2.4 Run 4

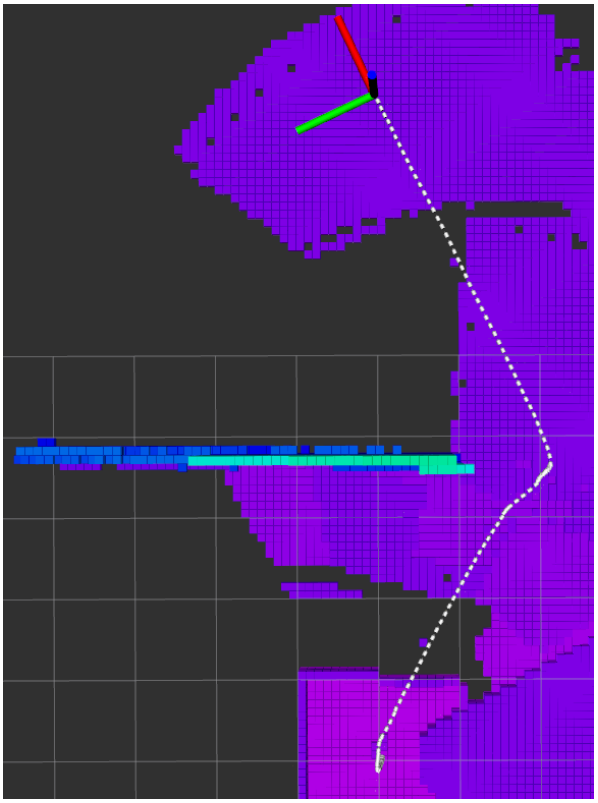


Figure C.15: The trajectory and OctoMap generated by the robot while solving test case 1. Extra run 4. Screen shot from Rviz.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.722	1.028	0.309	0.121

Table C.8: Obstacle distance and translational velocity for test case 1. Extra run 4.

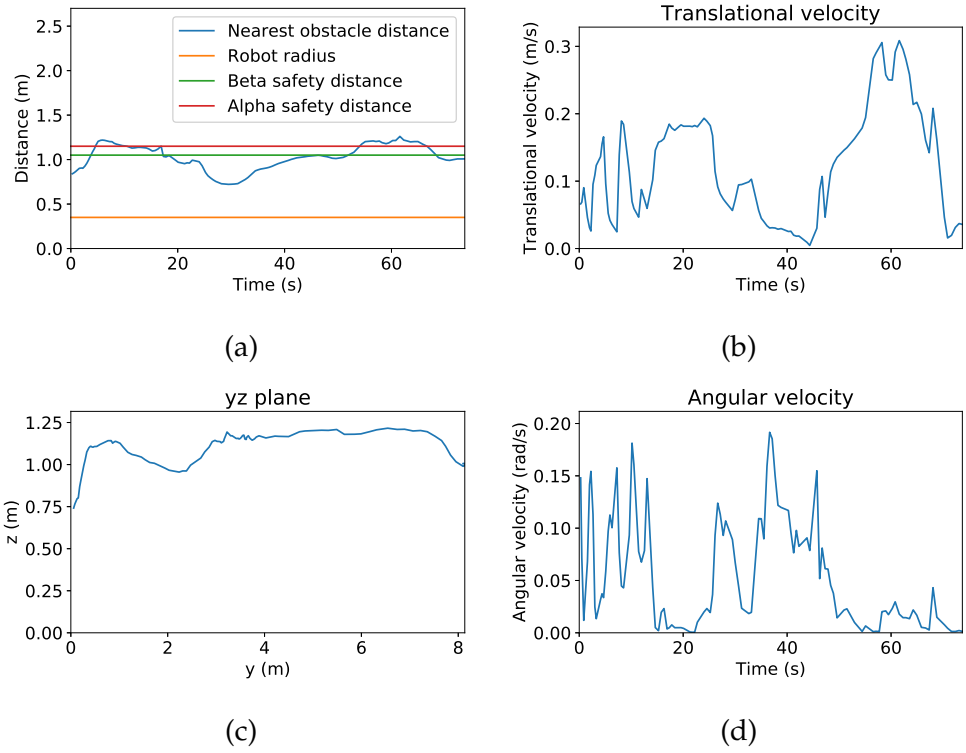
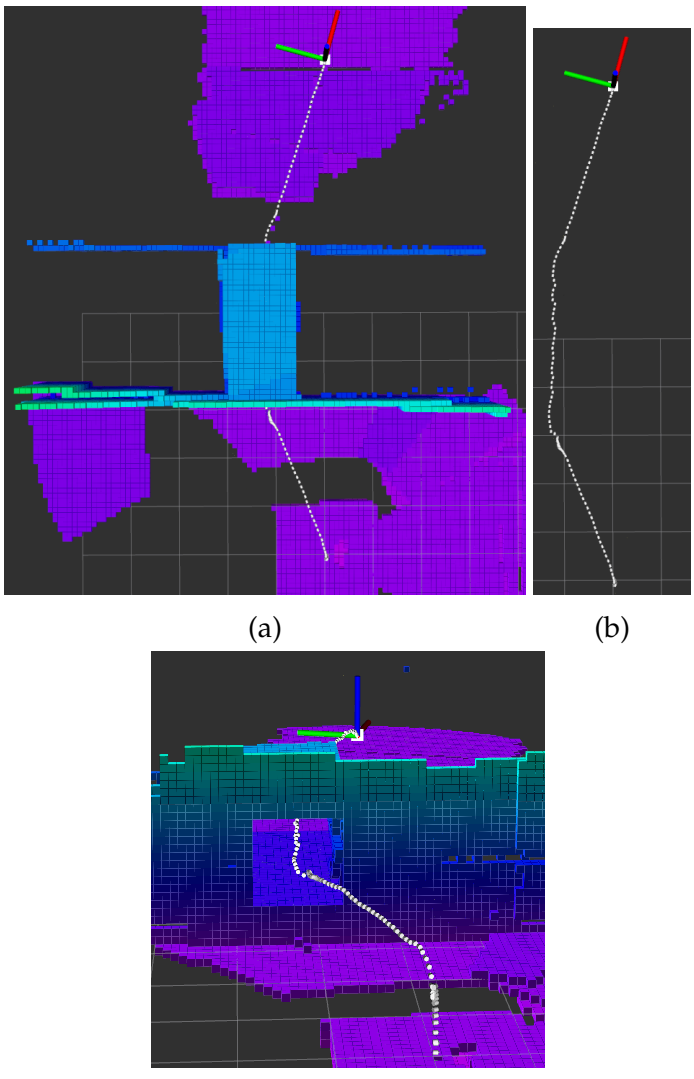


Figure C.16: Test case 1. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and angular velocities. (c) The trajectory of the robot in the yz plane.

C.3 Test case 2: Narrow passage

C.3.1 Run 1



(a) (b)

(c)

Figure C.17: The trajectory of the robot after completing test case 2, extra run 1. (a) Top view. (b) Top view without the generated OctoMap. (c) Another view.

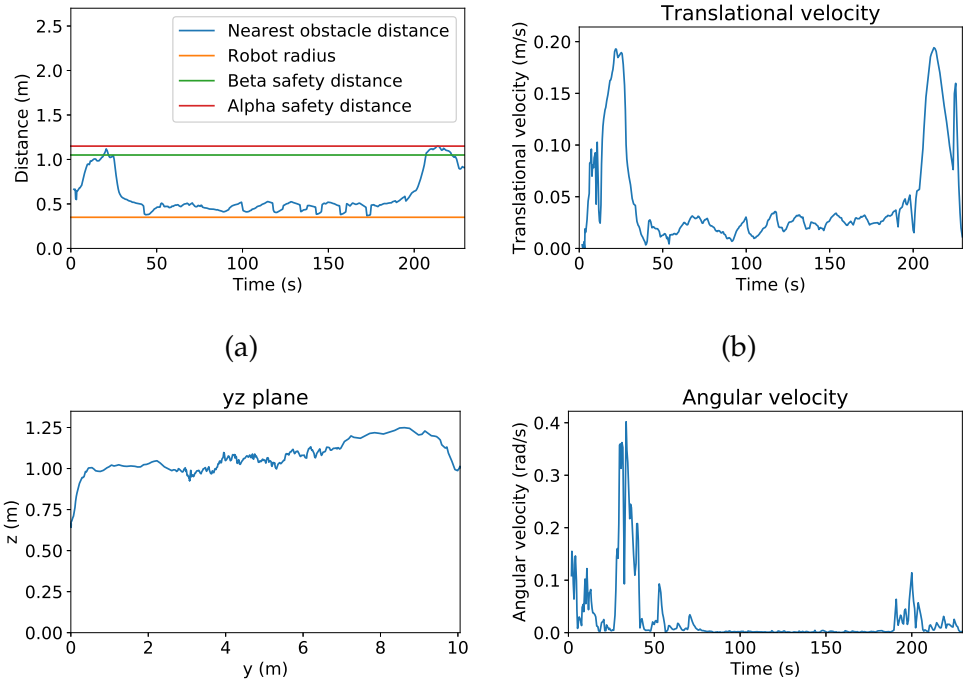
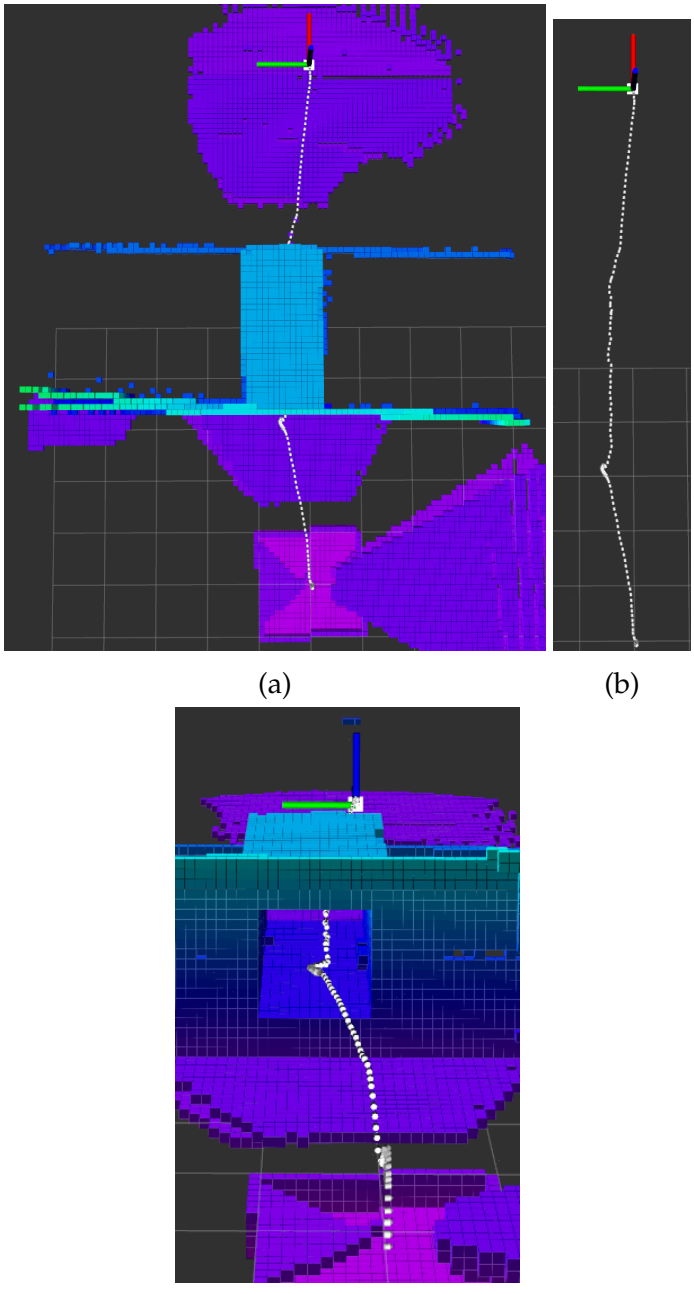


Figure C.18: Test case 2. Extra run 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.366	0.591	0.194	0.046

Table C.9: Obstacle distance and translational velocity for test case 2. Extra run 1.

C.3.2 Run 2



(a) (b)

(c)

Figure C.19: The trajectory of the robot after completing test case 2, extra run 2. (a) Top view. (b) Top view without the generated OctoMap. (c) Another view.

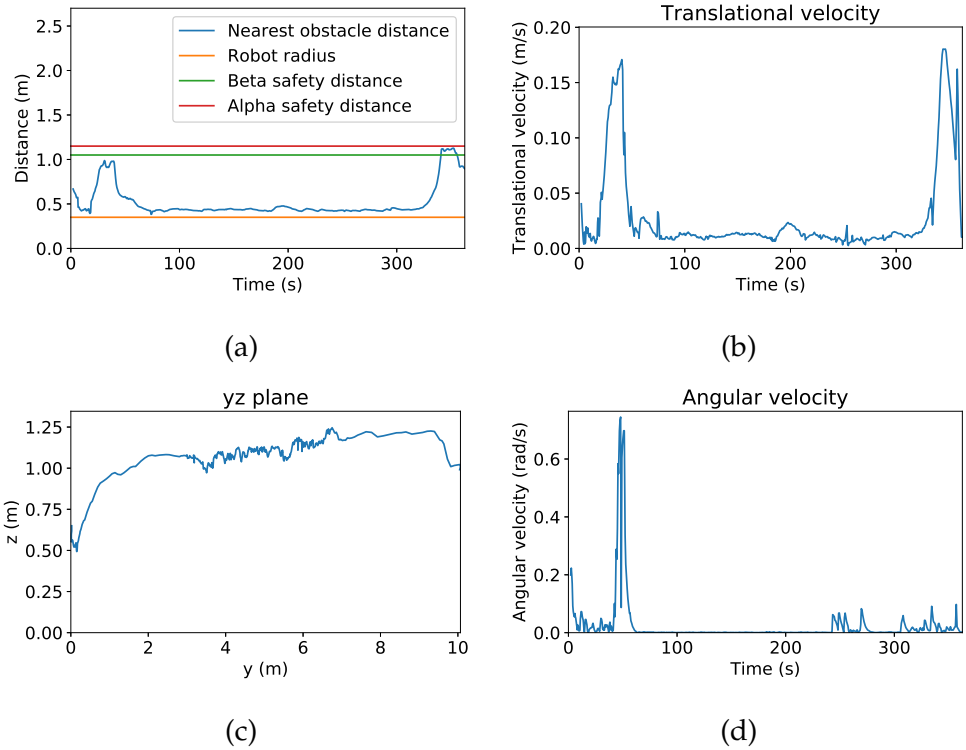
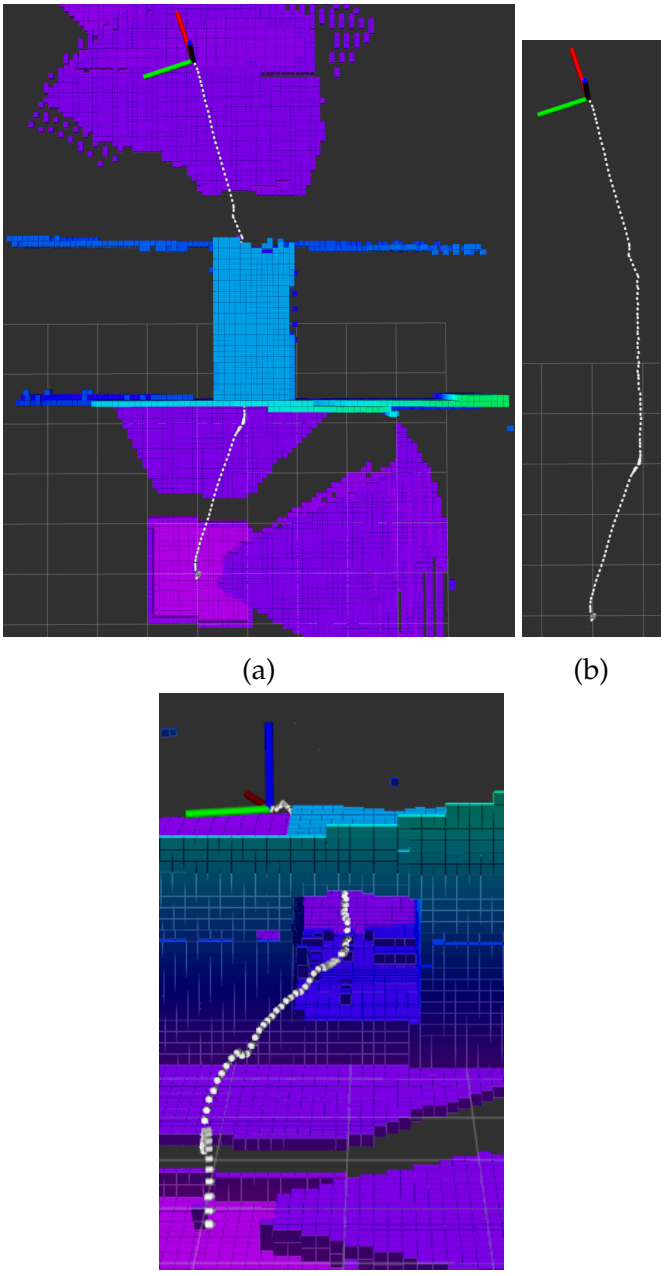


Figure C.20: Test case 2. Extra run 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.383	0.516	0.180	0.028

Table C.10: Obstacle distance and translational velocity for test case 2. Extra run 2.

C.3.3 Run 3



(a) (b)

(c)

Figure C.21: The trajectory of the robot after completing test case 2, extra run 3. (a) Top view. (b) Top view without the generated OctoMap. (c) Another view.

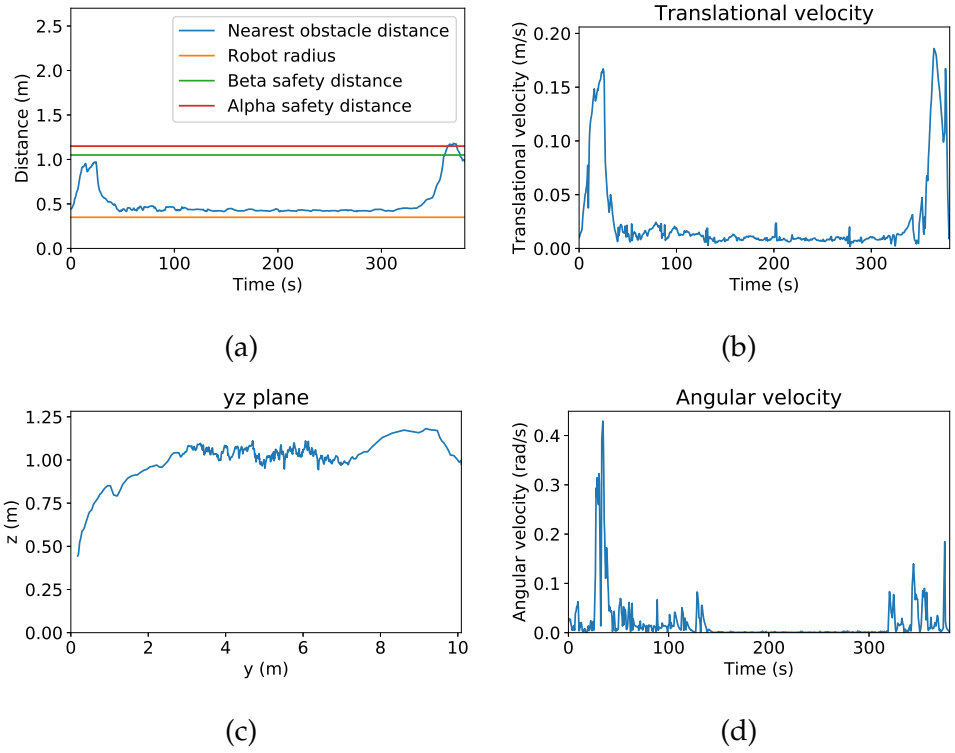
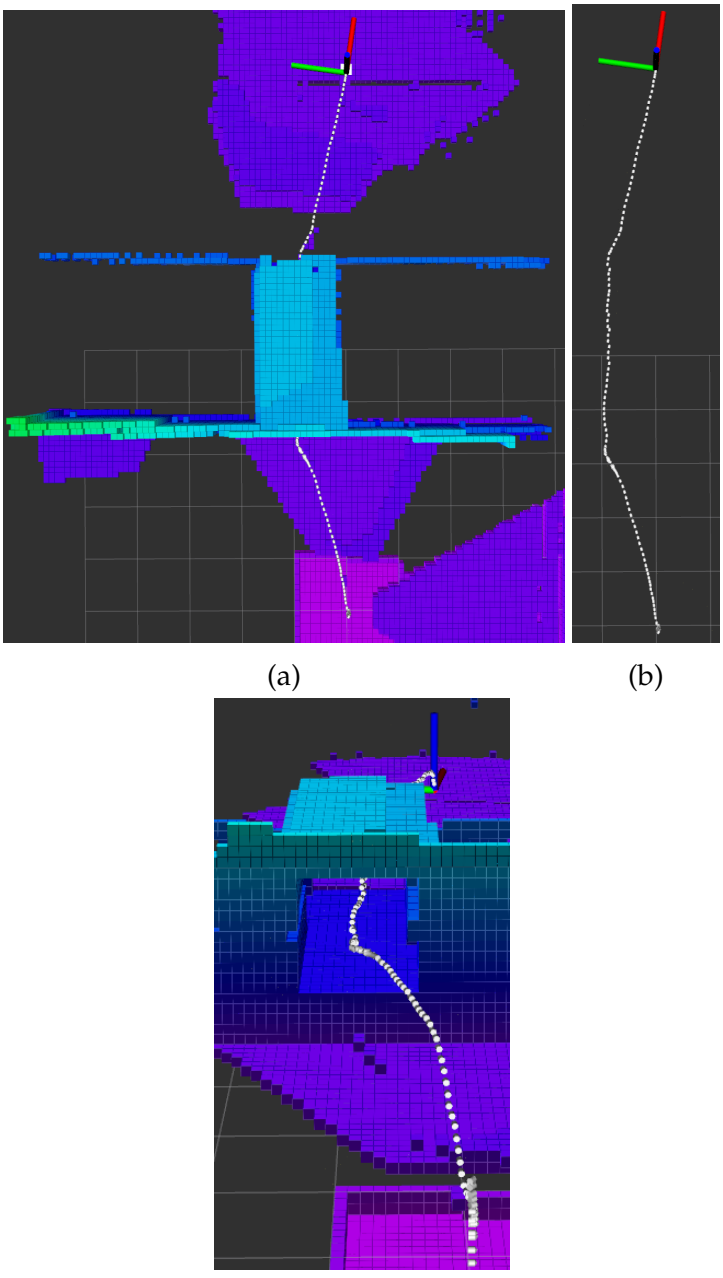


Figure C.22: Test case 2. Extra run 3. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.413	0.509	0.186	0.026

Table C.11: Obstacle distance and translational velocity for test case 2. Extra run 3.

C.3.4 Run 4



(a) (b)

(c)

Figure C.23: The trajectory of the robot after completing test case 2, extra run 4. (a) Top view. (b) Top view without the generated OctoMap. (c) Another view.

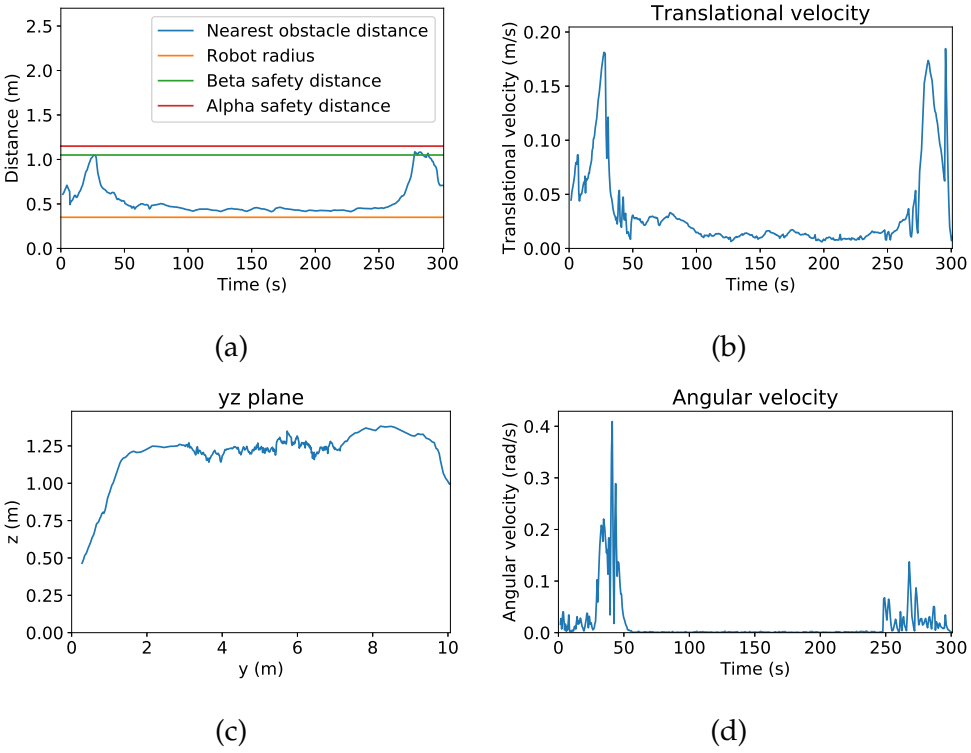


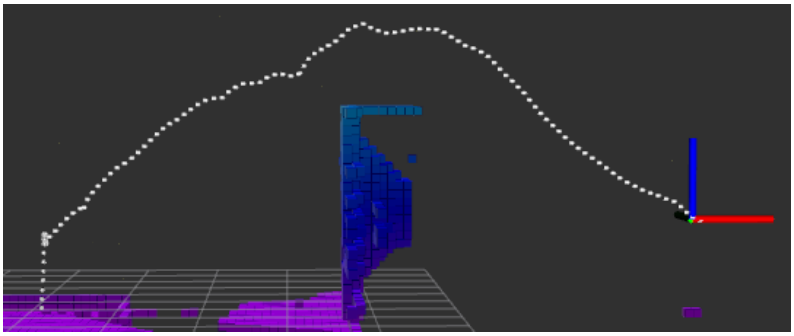
Figure C.24: Test case 2. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the yz plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.413	0.538	0.185	0.034

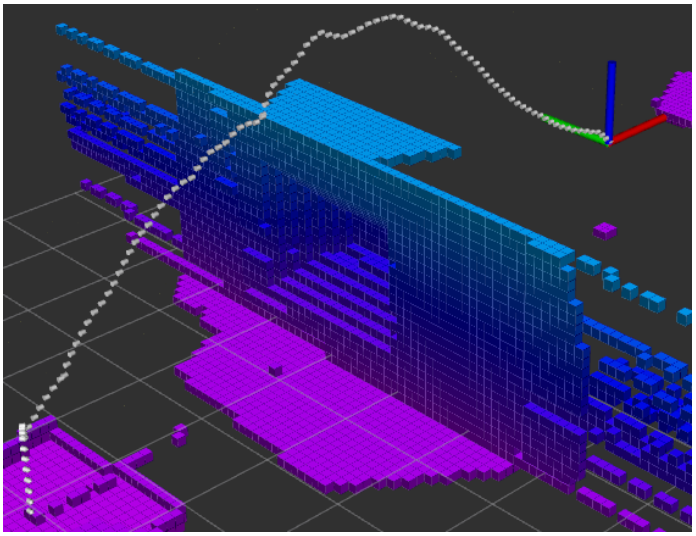
Table C.12: Obstacle distance and translational velocity for test case 2. Extra run 4.

C.4 Test case 3: Trap

C.4.1 Run 1



(a)



(b)

Figure C.25: The trajectory of the robot after completing test case 3, extra run 1, seen from two different views

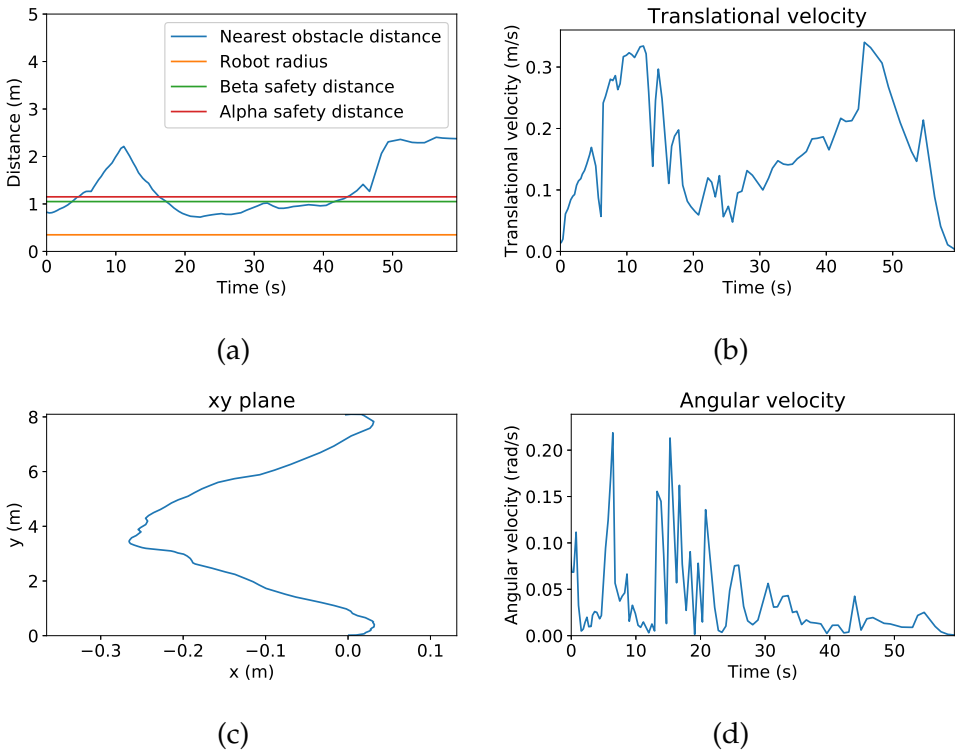
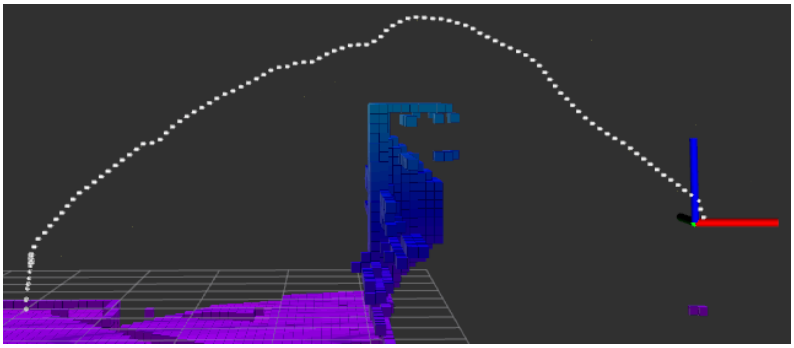


Figure C.26: Test case 3. Extra run 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

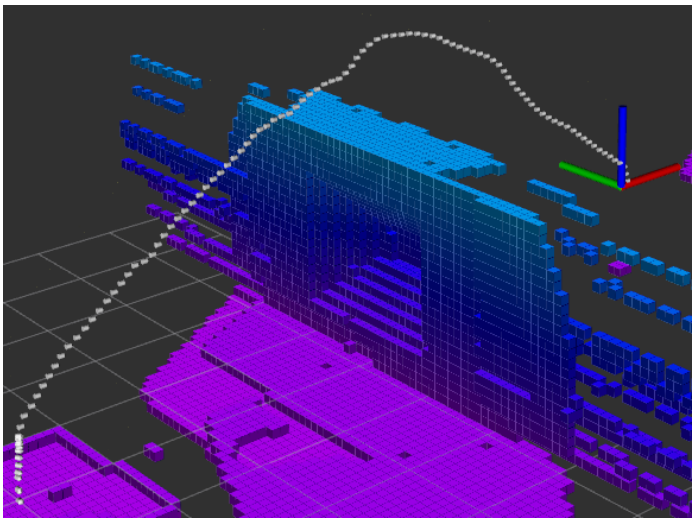
Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.723	1.278	0.340	0.165

Table C.13: Obstacle distance and translational velocity for test case 3. Extra run 1.

C.4.2 Run 2



(a)

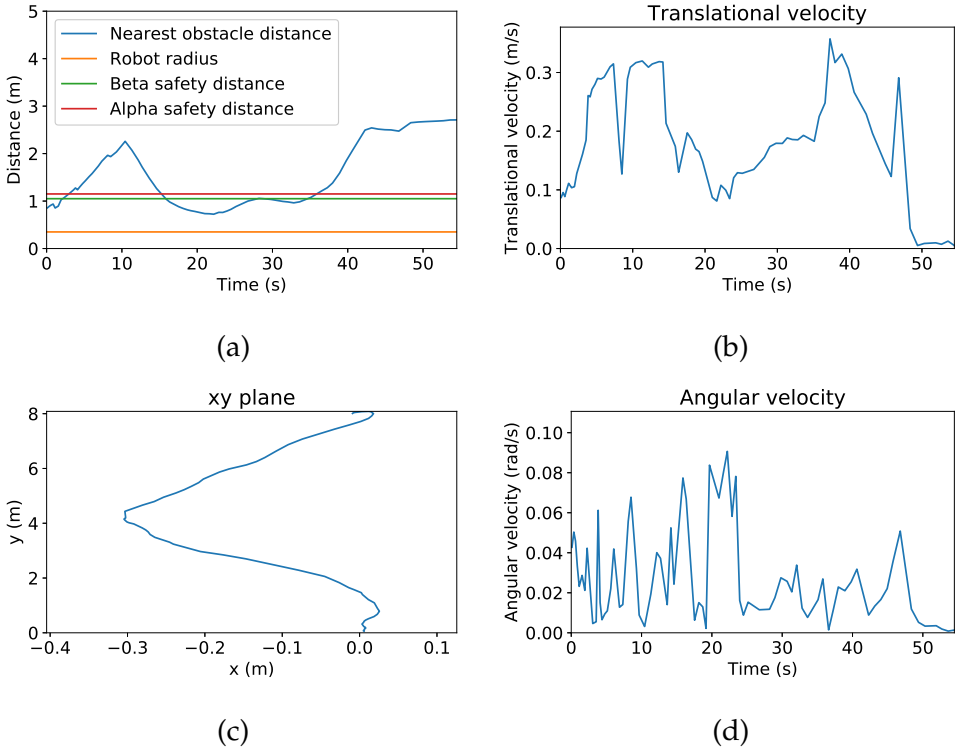


(b)

Figure C.27: The trajectory of the robot after completing test case 3, extra run 2, seen from two different views

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.723	1.467	0.357	0.184

Table C.14: Obstacle distance and translational velocity for test case 3. Extra run 2.

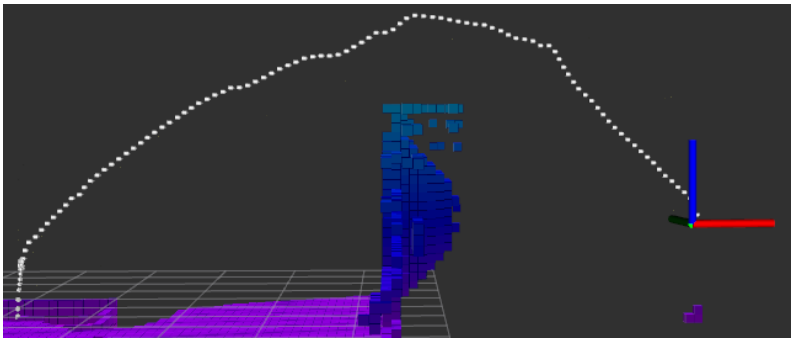


(a) (b)

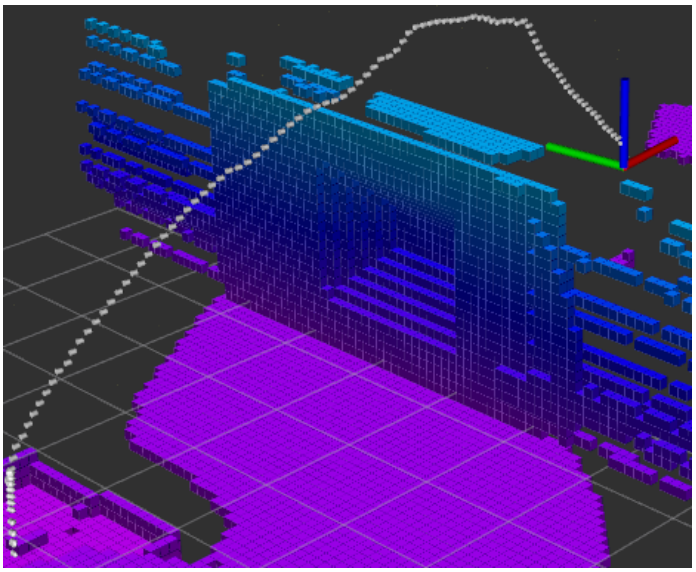
(c) (d)

Figure C.28: Test case 3. Extra run 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

C.4.3 Run 3



(a)



(b)

Figure C.29: The trajectory of the robot after completing test case 3, extra run 3, seen from two different views

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.833	1.453	0.369	0.205

Table C.15: Obstacle distance and translational velocity for test case 3. Extra run 3.

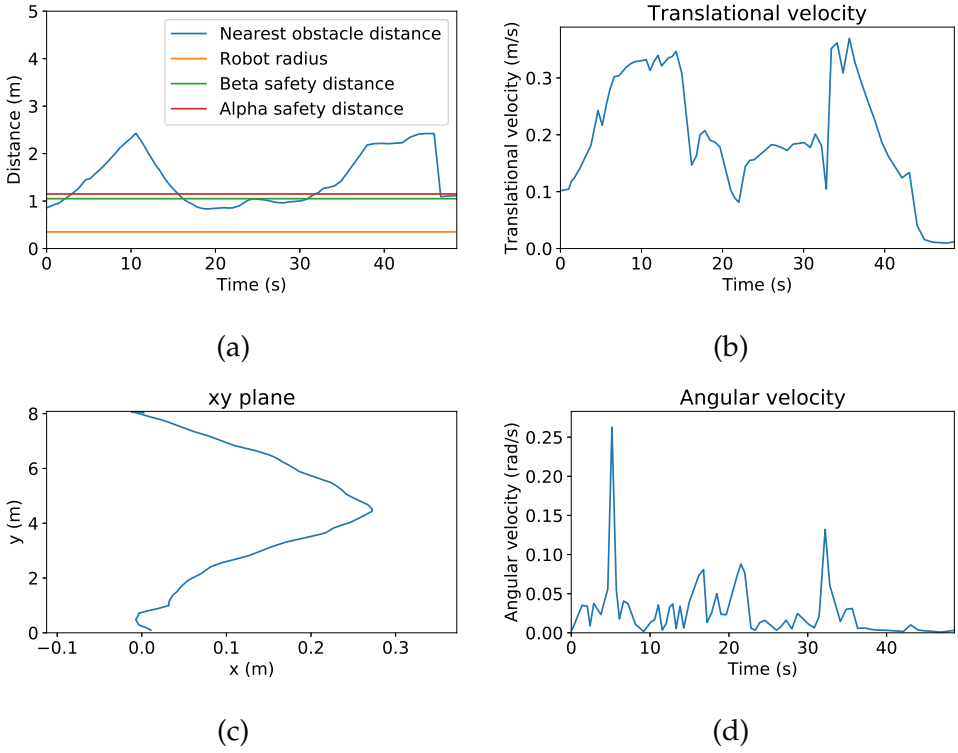
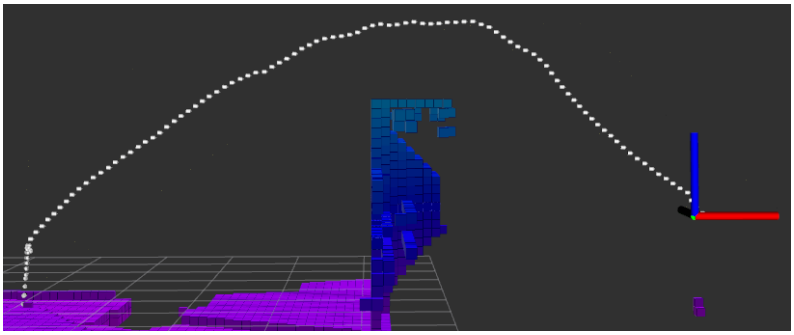
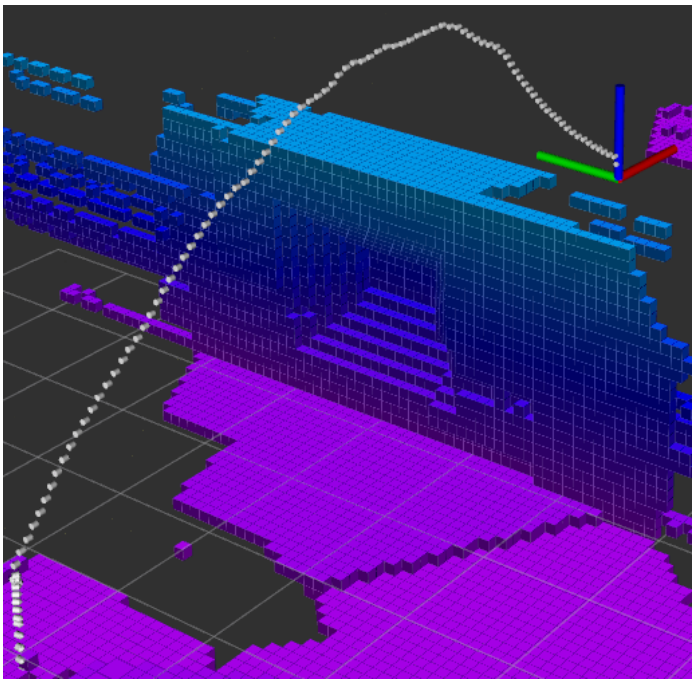


Figure C.30: Test case 3. Extra run 3. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

C.4.4 Run 4



(a)



(b)

Figure C.31: The trajectory of the robot after completing test case 3, extra run 4, seen from two different views

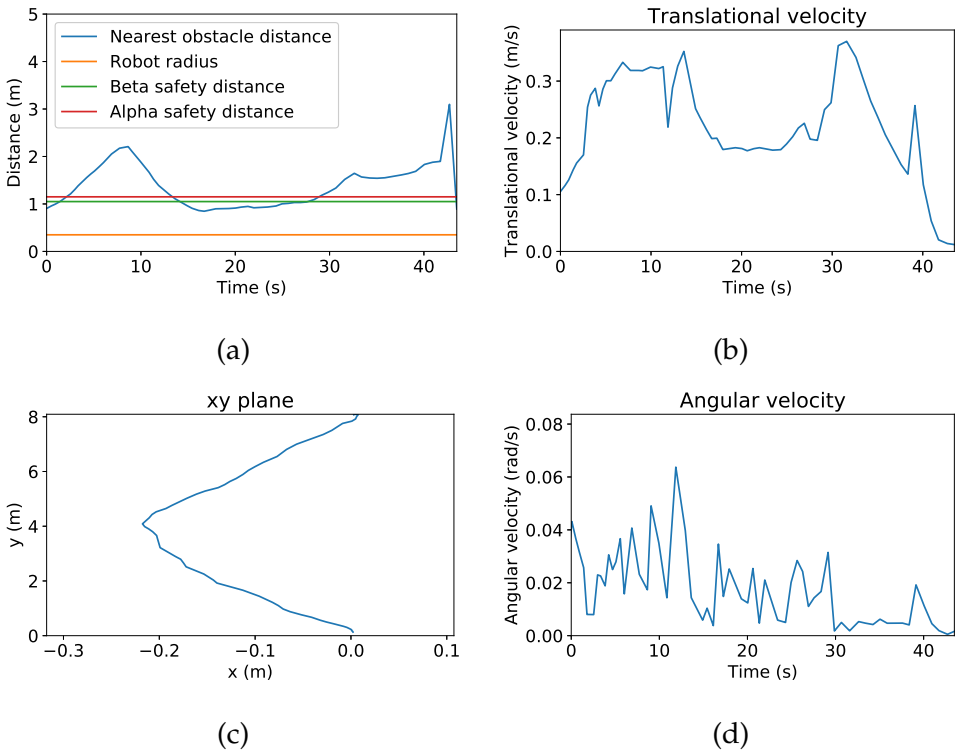


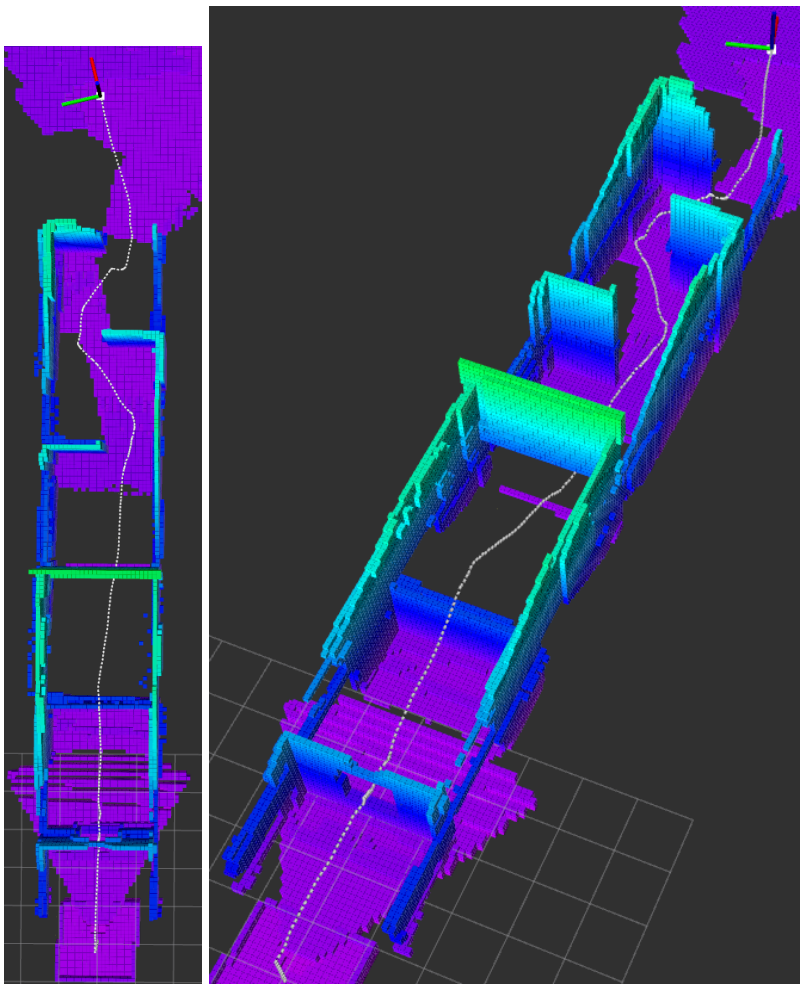
Figure C.32: Test case 3. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

Nearest obstacle distance (m)		Translational velocity (m/s)	
Minimum	Average	Maximum	Average
0.847	1.350	0.370	0.220

Table C.16: Obstacle distance and translational velocity for test case 3. Extra run 4.

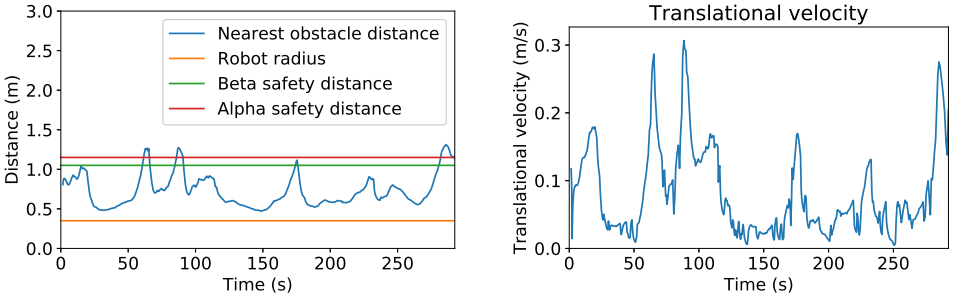
C.5 Test case 4: Obstacle corridor

C.5.1 Run 1



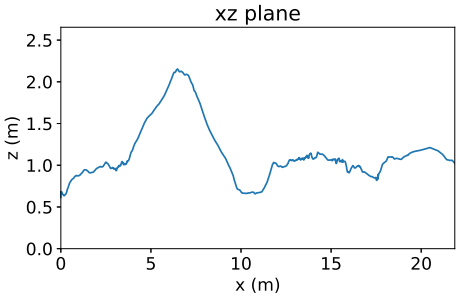
(a) (b)

Figure C.33: The trajectory of the robot after completing test case 4, extra run 1, seen from two different views.

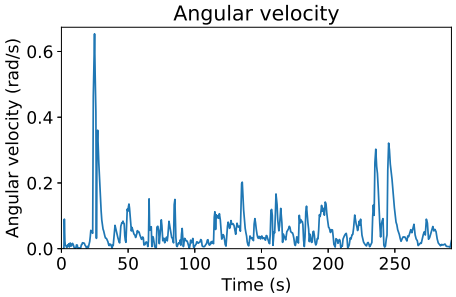


(a)

(b)



(c)



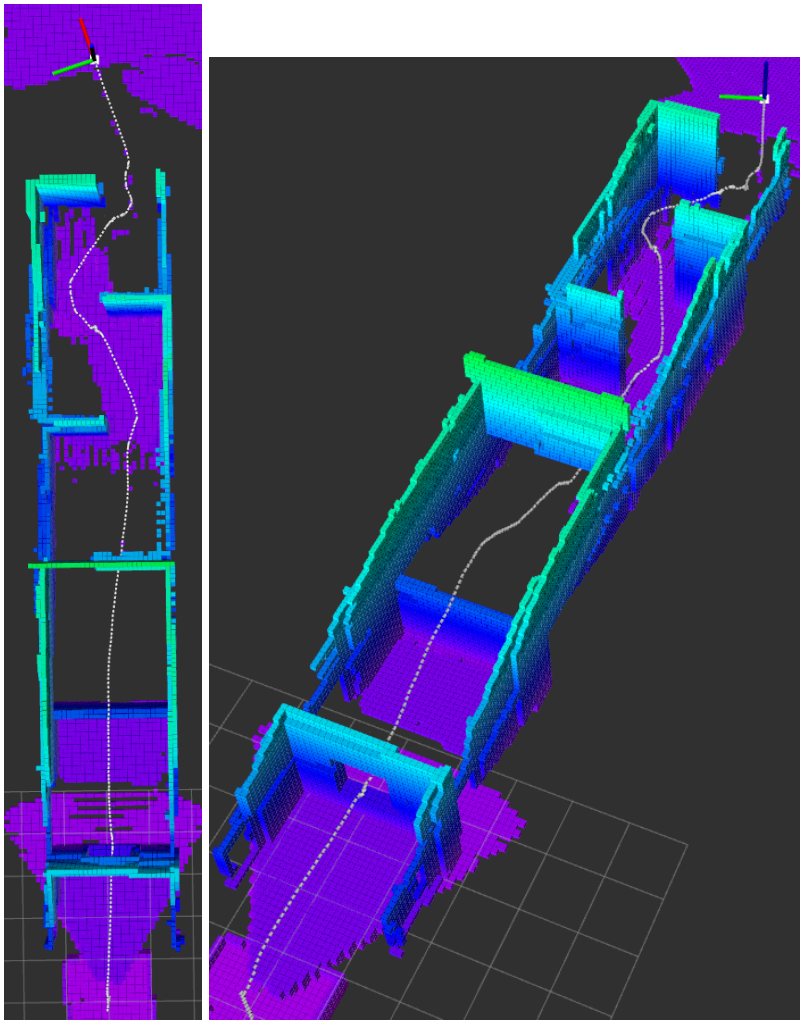
(d)

Figure C.34: Test case 4. Extra run 1. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

Nearest obstacle distance		Translational velocity	
Minimum	Average	Maximum	Average
0.472	0.721	0.307	0.080

Table C.17: Obstacle distance and translational velocity for test case 4. Extra run 1.

C.5.2 Run 2



(a) (b)

Figure C.35: The trajectory of the robot after completing test case 4, extra run 2, seen from two different views.

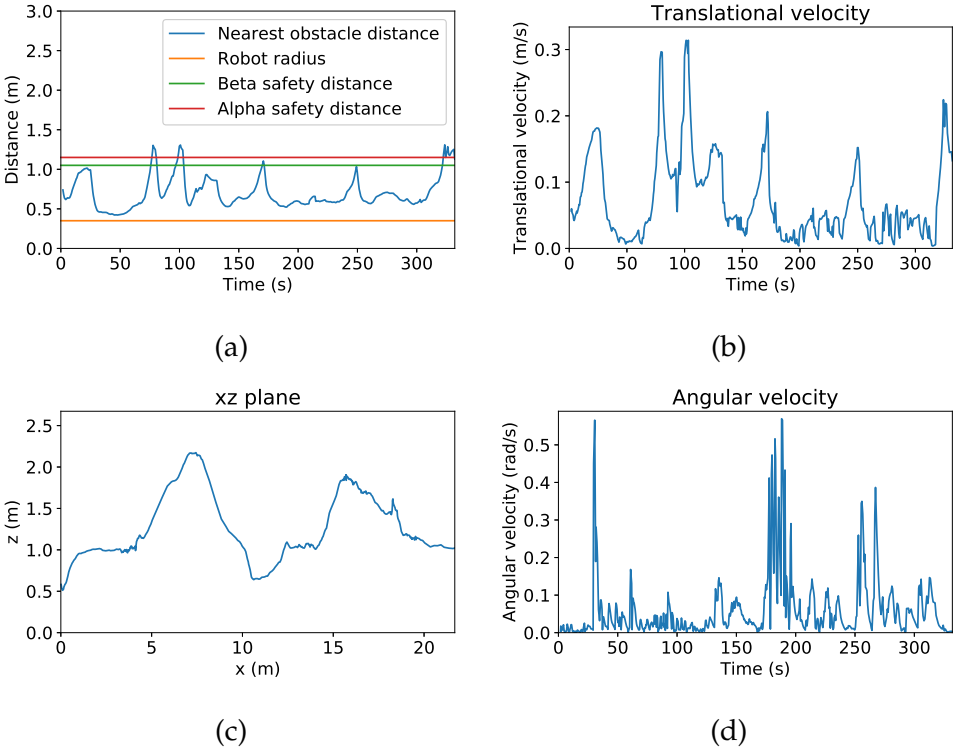
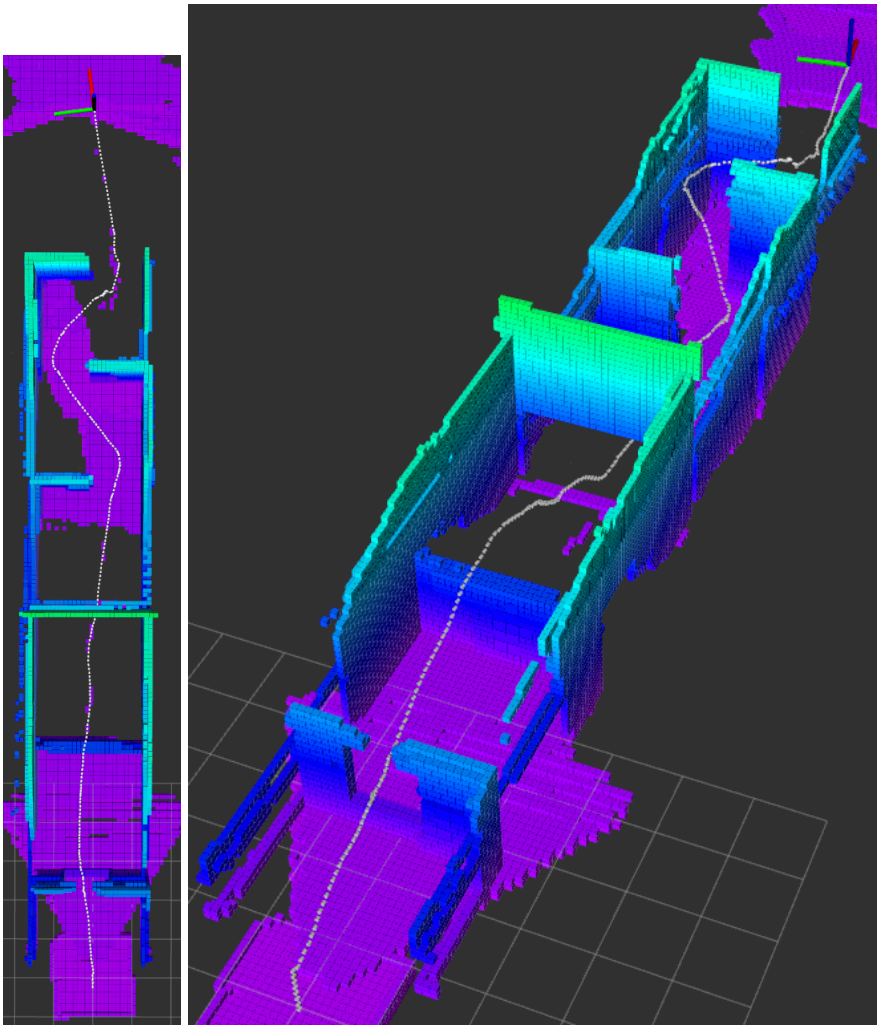


Figure C.36: Test case 4. Extra run 2. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

Nearest obstacle distance		Translational velocity	
Minimum	Average	Maximum	Average
0.422	0.693	0.314	0.072

Table C.18: Obstacle distance and translational velocity for test case 4. Extra run 2.

C.5.3 Run 3



(a) (b)

Figure C.37: The trajectory of the robot after completing test case 4, extra run 3, seen from two different views.

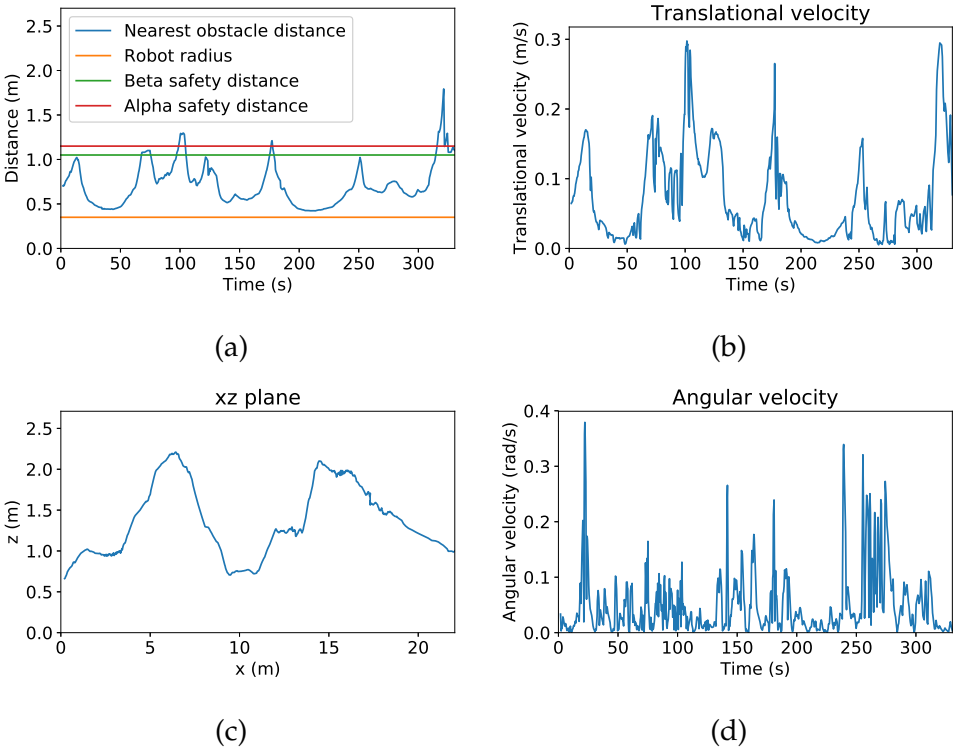
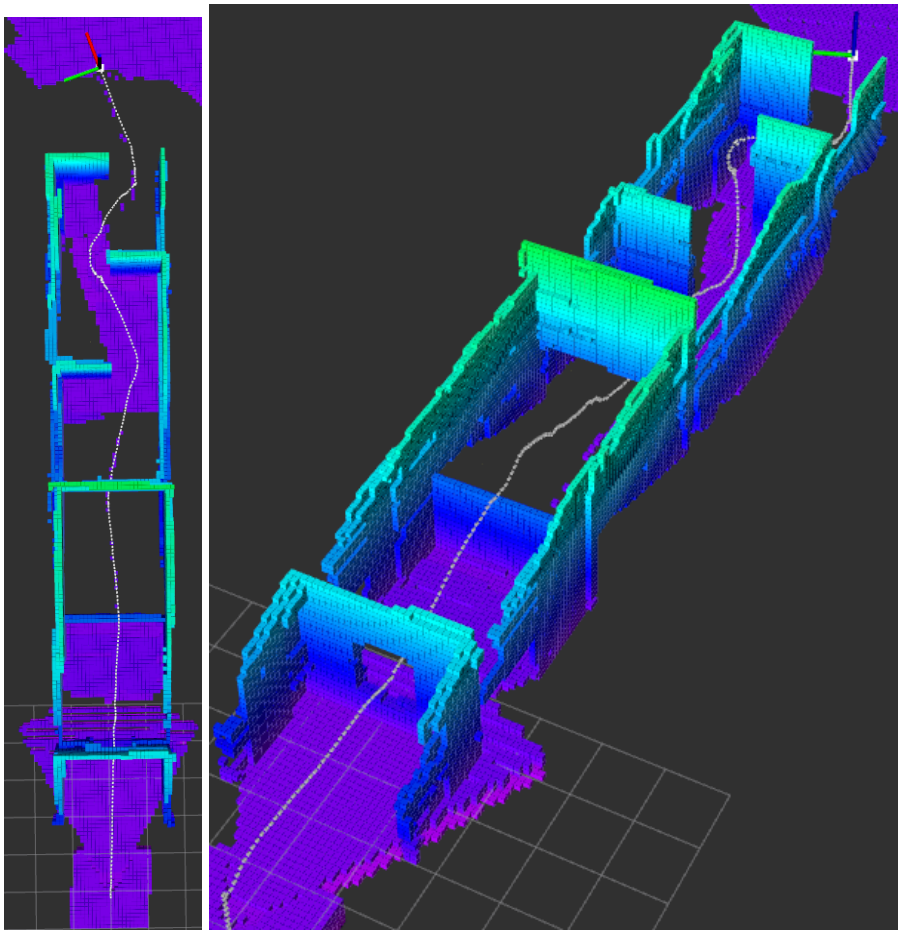


Figure C.38: Test case 4. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

Nearest obstacle distance		Translational velocity	
Minimum	Average	Maximum	Average
0.422	0.700	0.298	0.072

Table C.19: Obstacle distance and translational velocity for test case 4. Extra run 3.

C.5.4 Run 4



(a) (b)

Figure C.39: The trajectory of the robot after completing test case 4, extra run 4, seen from two different views.

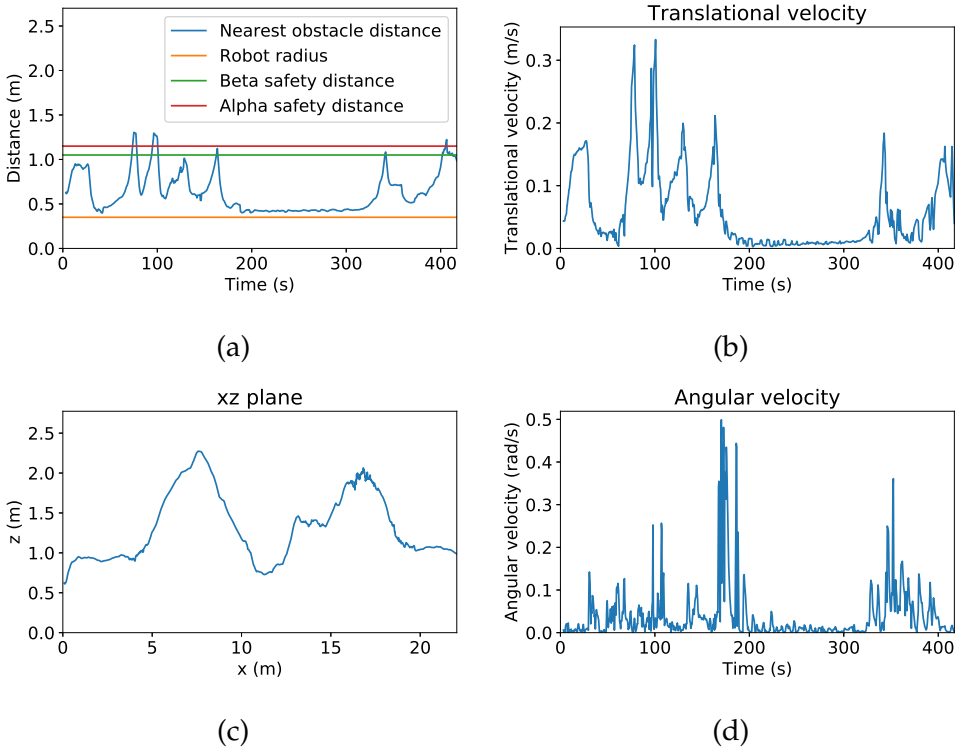


Figure C.40: Test case 4. Extra run 4. (a) Nearest obstacle distance at each time. (b), (d) Translational and rotational velocities. (c) The trajectory of the robot in the xy plane.

Nearest obstacle distance		Translational velocity	
Minimum	Average	Maximum	Average
0.397	0.630	0.333	0.062

Table C.20: Obstacle distance and translational velocity for test case 4. Extra run 4.