College of Engineering, Science and Environment

School of Engineering

Mechanical Engineering Project B

Semester 2 - 2024 MECH4841 A

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

FINAL YEAR PROJECT

PROJECT TITLE

Simulation of autonomous exploration

NAME & STUDENT NUMBER

Hamish Bottin C3380478

SUPERVISOR

Dr Alejandro Donaire

0.1 Dot Point Summary

I developed an autonomous quadcopter simulation with the capability to explore unknown environments.

Key research areas:

- Quadcopter kinematics and dynamics.
- Nonlinear MPC control.
- OctoMaps for 3D environment representation.
- Frontier detection and selection for exploration.
- Rapidly-exploring Random Trees Star (RRT*) for path planning.

The simulation combines MATLAB's inbuilt functions with custom algorithms implemented in the MATLAB Simulink environment. The physical environment is simulated in Unreal Engine.

Declaration

I declare that this thesis is my own work unless otherwise acknowledged and is in accordance with the University's academic integrity policy available from the Policy Library on the web at http://www.newcastle.edu.au/policylibrary/000608.html

I certify that this assessment item has not been submitted previously for academic credit in this or any other course.

I acknowledge that the School of Engineering may, for the purpose of assessing this thesis:

- Reproduce this thesis and provide a copy to another member of the Faculty; and/or
- Communicate a copy of this assessment item to a plagiarism checking service (which may then retain a copy of the item on its database for the purpose of future plagiarism checking).
- Submit the assessment item to other forms of plagiarism checking.
- Where appropriate, provide a copy of this thesis to other students where that student is working in/on a related project.

I further acknowledge that the School of Engineering may, for the purpose of sector wide Quality Assurance:

- Reproduce this thesis to provide a copy to a member of another engineering faculty.

Furthermore, I have checked that the electronic version is complete and intact on the submitted location.

Student name: Hamish Bottin

Signature [illegible]

0.2 Abstract

This project aims to develop algorithms enabling quadcopters to autonomously explore unknown environments. This is achieved through the creation of three-dimensional occupancy maps, known as OctoMaps, for spatial representation; wavefront frontier detection (WFD) and Frontier-Tracing Frontier Detection (FTFD) to identify search areas; and RRT* for path planning. A Nonlinear Model Predictive Control (NLMPC) framework manages the quadcopter's inherent nonlinear dynamics to control its movement. This proof-of-concept implementation demonstrates the feasibility of combining these systems for exploration tasks and has the potential to significantly enhance search and rescue operations in complex environments.

0.3 Acknowledgements

I would like to thank Dr. Alejandro for his support and recommendations in achieving the goals of this project. also the use of LLM such as GPT and Claude for there help in debuging code and enhancing readability of this report by fixing grammatical errors.

Contents

List of Figures

List of Tables

I have not left a mess in the lab declaration page.

I   Hamish Bottin having completed the project Simulation od autonomous exploration , and using laboratory spaces (N.A.), hereby submit that the area has been cleaned to level equal to or better than when I commenced my work. All items I borrowed have been returned, and the area is now suitable for inspection and sing off by the appropriate laboratory manager.

Signed [illegible]

Date 27/10/24

Appropriate Manager   Signed   Date

0.4 Introduction

In hazardous environments, quickly and accurately locating people or assessing conditions is crucial, yet often difficult and dangerous. This challenge is especially pronounced in emergency scenarios, such as search and rescue missions after natural disasters, where obstacles like debris, downed power lines, and blocked roads can delay rescue efforts and put responders at risk. Advances in autonomous quadcopter technology offer a promising solution. These drones can access dangerous areas faster and more safely than human teams. Equipped with mapping and navigation capabilities, they can explore hard-to-reach places and generate 3D maps of their surroundings in real-time. This project demonstrates a proof of concept for an autonomous exploration and mapping drone designed for complex environments. Although it does not include object or person detection, the drone successfully generates 3D maps of its surroundings. These results highlight the potential of such drones for real-world applications, especially in situations where human presence may be unsafe or impractical.

0.5 Problem Scope

The objective of this project is to develop a fully autonomous system capable of mapping a previously unknown environment using a quadcopter. To achieve this, both a comprehensive model of the system and a suitable control structure must be designed and implemented. Additionally, the system must be able to reliably map the environment, determine the optimal location to move next, and navigate safely.

However, this project does not include state estimation or the physical implementation of the system. Instead, it focuses on the conceptual and computational aspects necessary for autonomous exploration and mapping.

Chapter 1

UAV Modeling

A quadcopter, also known as a quadrotor or quadrotor helicopter, is an unmanned aerial vehicle (UAV) propelled by four rotors typically arranged in a cross or X configuration. The quadcopter is represented in Figure 1.1,it is assumed to be a black box where the only control inputs are the voltages applied to the four motors. All other inputs are treated as disturbances. Navigation (state estimation) is assumed to be an internal process within the black box, which outputs the six velocity and six acceleration states. Developing a model of this black box is crucial for understanding the dynamics of the system and predicting its behavior under different conditions. By accurately modeling how the black box processes the motor voltages and disturbances to produce velocity and acceleration outputs, This model enables the design of more effective control strategies.

1.1 Quadcopter Dynamics

For this project, the quadcopter is assumed to be in a cross '+' configuration. Each rotor produces both thrust and torque about the center of mass. Typically, two rotors spin clockwise, and two spin counterclockwise, with adjacent rotors rotating in opposite directions. This configuration has two benefits: first, it can balance the torque from the rotors, and second, it allows independent rotation about the z-axis from the x and y axes. Additionally, the typical coordinate system of a quadcopter is shown in Figure 1.1. The figure also illustrates the direction of force each rotor applies and their direction of rotation. The body-fixed coordinate system is also illustrated. [1]

In a quadcopter, thrust, roll, pitch, and yaw are controlled by varying the speeds of the individual rotors. Thrust, which lifts the quadcopter off the ground, is adjusted by increasing or decreasing the speed of all four rotors equally. Roll, the rotation around the longitudinal axis, is achieved by increasing the speed of the rotors on one side while decreasing the speed of the rotors on the opposite side. Pitch, the rotation around the lateral axis, is controlled by increasing the speed of the rear rotors and decreasing the speed of the front rotors, or vice versa. Yaw, the rotation around the vertical axis, is managed by creating a differential in torque between the clockwise-rotating rotors and the counterclockwise-rotating rotors. By finely adjusting rotor speeds, the flight controller can stabilize the quadcopter and control its movements for responsive flight.

The equations governing these rotations are given by equation 1.2. [6]

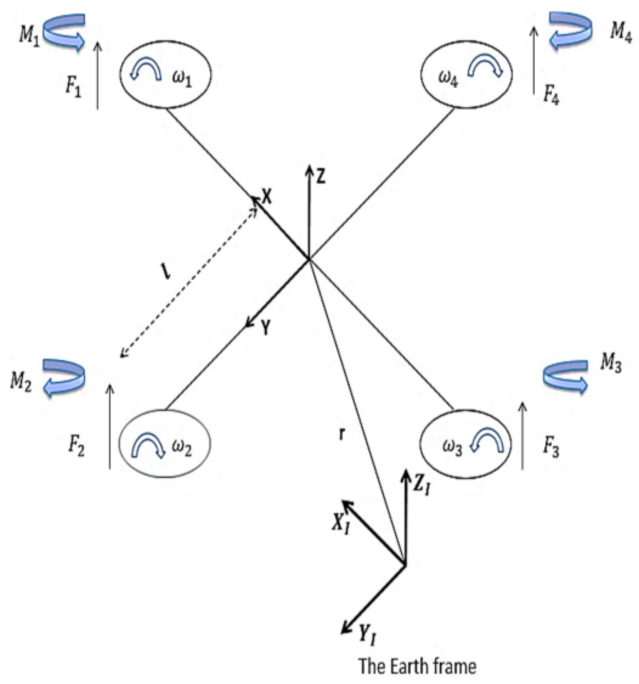

Figure 1.1: Quadcopter system model. [1]

$$T = k(T_1 + T_2 + T_3 + T_4) \tag{1.1a}$$
$$\tau_\phi = lk(-T_2 + T_4) \tag{1.1b}$$
$$\tau_\theta = lk(-T_1 + T_3) \tag{1.1c}$$
$$\tau_\psi = b(-T_1 + T_2 - T_3 + T_4) \tag{1.1d}$$

$$\boldsymbol{\tau}_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{1.2}$$

where T_i represents the thrust produced by the i-th rotor, l is the distance from the center of the quadcopter to each rotor, k is the lift constant related to the aerodynamic properties of the propellers, m is the quadrotor mass, and b is the drag constant.

In these equations, ϕ, θ, and ψ represent the roll, pitch, and yaw angles, respectively. The torques τ_ϕ, τ_θ, and τ_ψ are the moments around the body-fixed axes of the quadcopter corresponding to these angles.

1.2 Kinimatics

To describe the pose (position and orientation) of the quadcopter, two reference coordinate systems are required. The first is fixed, known as the inertial coordinate system, where Newton's first law is considered valid. The second coordinate system is attached to the quadcopter's center of mass. In this project, an X, Y, Z coordinate system is used. Both coordinate systems are depicted in Figure 1.1. [7]

To describe the orientation of the quadcopter's body frame with respect to the inertial frame, Euler angles are commonly used. In the ZYX convention, the Euler angles represent successive rotations about the Z, Y, and X axes of the body frame. The rotation matrix R_{ZYX} for the ZYX convention can be expressed as:

$$R_{\text{ZYX}} = R_z(\psi) \cdot R_y(\theta) \cdot R_x(\phi) \tag{1.3}$$

where:

- ψ (psi) is the yaw angle, representing the rotation about the Z-axis.
- θ (theta) is the pitch angle, representing the rotation about the Y-axis.
- ϕ (phi) is the roll angle, representing the rotation about the X-axis.

Each individual rotation matrix $R_i(\alpha)$ for the Z, Y, and X axes can be defined as:

$$R_z(\psi) = \begin{bmatrix} c(\psi) & -s(\psi) & 0 \\ s(\psi) & c(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4a}$$

$$R_y(\theta) = \begin{bmatrix} c(\theta) & 0 & s(\theta) \\ 0 & 1 & 0 \\ -s(\theta) & 0 & c(\theta) \end{bmatrix} \tag{1.4b}$$

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c(\phi) & -s(\phi) \\ 0 & s(\phi) & c(\phi) \end{bmatrix} \tag{1.4c}$$

The full rotation matrix R_{ZYX} combines these individual rotations to represent the orientation of the body frame relative to the inertial frame.

$$R_{\text{ZYX}} = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\phi)s(\theta) - c(\phi)s(\psi) & s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta) \\ c(\theta)s(\psi) & c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta) & c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\phi)c(\theta) \end{bmatrix} \tag{1.5}$$

Body velocities in the inertial frame can be described using the rotation matrix _ textZYX and the body velocities in the body frame with the following formula:

$$\mathbf{v_i} = R_{\text{ZYX}} \cdot \mathbf{v_b} \tag{1.6}$$

where $\mathbf{v_i} = [u_0 \; v_0 \; w_0]^T$ is the linear velocity of the mass center expressed in the inertial frame, and $\mathbf{v_b} = [u_L \; v_L \; w_L]^T$ is the linear velocity of the mass center expressed in the body-fixed frame.

the dynamics of the quadcopter are defined by the state space equation 1.7 where $\dot{X} = f(X, U)$, where X is the state vector and U is the input vector. The state vector is defined as $X = [x \; y \; z \; \phi \; \theta \; \psi \; \dot{x} \; \dot{y} \; \dot{z} \; \dot{\phi} \; \dot{\theta} \; \dot{\psi}]^T$ equivalently $X = [x_1 \; x_2 \; x_3 \; x_4 \; x_5 \; x_6 \; x_7 \; x_8 \; x_9 \; x_{10} \; x_{11} \; x_{12}]^T$ [8]

$$f(X, U) = \begin{bmatrix} x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ \frac{u}{m}\left(c(x_4)s(x_5)c(x_6) + s(x_4)s(x_6)\right) \\ \frac{u}{m}\left(c(x_4)s(x_5)s(x_6) - s(x_4)c(x_6)\right) \\ -g + \frac{u}{m}\left(c(x_4)c(x_5)\right) \\ \ddot{\eta}(1) \\ \ddot{\eta}(2) \\ \ddot{\eta}(3) \end{bmatrix} \tag{1.7}$$

where u is the sum of elements of U and $\ddot{\eta}$ is definde in equation 1.15

$\ddot{\eta}$ is derived from the lagrange-Euler equations 1.8

$$M(\eta)\ddot{\eta} + C(\eta, \dot{\eta})\dot{\eta} = \boldsymbol{\tau}_B \tag{1.8}$$

where $M(\eta)$ is the mass matrix defined by 1.9 and $\eta = [\phi\ \theta\ \psi]^T$

$$M(\eta) = W'_{\eta} J W_{\eta} \tag{1.9}$$

$$W_{\eta} = \begin{bmatrix} 1 & 0 & -s(\theta) \\ 0 & c(\phi) & c(\theta)s(\phi) \\ 0 & s(\phi) & c(\phi)s(\theta) \end{bmatrix} \tag{1.10}$$

$$J = \begin{bmatrix} Ixx & 0 & 0 \\ 0 & Iyy & 0 \\ 0 & 0 & Izz \end{bmatrix} \tag{1.11}$$

$$M(\eta) = \begin{bmatrix} I_{xx} & 0 & -I_{xx}s(\theta) \\ 0 & I_{yy}c^2(\phi) + I_{zz}s^2(\phi) & (I_{yy} - I_{zz})c(\phi)s(\phi)c(\theta) \\ -I_{xx}s(\theta) & (I_{yy} - I_{zz})c(\phi)s(\phi)c(\theta) & I_{xx}s^2(\theta) + I_{yy}s^2(\phi)c^2(\theta) + I_{zz}c^2(\phi)c^2(\theta) \end{bmatrix} \tag{1.12}$$

$C(\eta, \dot{\eta})$ is the Coriolis matrix defined by 1.13

$$C(\eta, \dot{\eta}) = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \tag{1.13}$$

$$\begin{aligned}
&C_{11} = 0, \\
&C_{12} = (I_y - I_z)(\dot{\theta}c(\phi)s(\phi) + \dot{\psi}s^2(\phi)c(\theta)) + (I_z - I_y)(\dot{\psi}c^2(\phi)c(\theta)) - I_x\dot{\psi}tc(\theta), \\
&C13 = (I_z - I_y)\dot{\psi}c(\phi)s(\phi)c^2(\theta), \\
&C21 = (I_z - I_y)(\dot{\theta}c(\phi)s(\phi) + \dot{\psi}s^2(\phi)c(\theta)) + (I_y - I_z)(\dot{\psi}c^2(\phi)c(\theta)) + I_x\dot{\psi}c(\theta_t), \\
&C22 = (I_z - I_y)\dot{\phi}c(\phi)s(\phi), \\
&C23 = -I_x\dot{\psi}s(\theta)c(\theta) + I_y\dot{\psi}s^2(\phi)s(\theta)c(\theta) + I_z\dot{\psi}c^2(\phi)s(\theta)c(\theta), \\
&C31 = (I_y - I_z)\dot{\psi}c^2(\theta)s(\phi)c(\phi) - I_x\dot{\theta}c(\theta), \\
&C32 = (I_z - I_y)(\dot{\theta}c(\phi_t)s(\phi_t)s(\theta_t) + \dot{\phi}s^2(\phi)c(\theta)) + (I_y - I_z)(\dot{\phi}c^2(\phi)c(\theta)) \\
&\qquad + I_x\dot{\psi}s(\theta)c(\theta) - I_y\dot{\psi}_t s^2(\phi)s(\theta)c(\theta) - I_z\dot{\psi}c^2(\phi)s(\theta)c(\theta), \\
&C33 = (I_y - I_z)\dot{\phi}c(\phi)s(\phi)c^2(\theta) - I_y\dot{\theta}s^2(\phi)c(\theta)s(\theta) - I_z\dot{\theta}c^2(\phi)c(\theta)s(\theta) + I_x\dot{\theta}c(\theta)s(\theta).
\end{aligned} \tag{1.14}$$

therefor $\ddot{\eta}$ obtained by 1.15 compleating $F(X, U)$

$$\ddot{\eta} = M(\eta)^{-1}(\boldsymbol{\tau}_B - C(\eta, \dot{\eta})\dot{\eta}) \tag{1.15}$$

Chapter 2

System components

2.1 Control

Model Predictive Control (MPC) is a dynamic control methodology that computes control inputs by solving an optimization problem at each time step while considering future system behavior over a finite horizon. This method excels in handling multi-variable systems with input and state constraints. However, the assumption of linearity in traditional MPC models often falls short when dealing with the inherently nonlinear dynamics of complex systems such as quadcopters, which require more sophisticated modeling for precise control.

Nonlinear Model Predictive Control (NLMPC) addresses this limitation by incorporating nonlinear system models into the optimization framework. The objective of NLMPC is to minimize the deviation of the quadcopter from its desired trajectory while keeping the control efforts within reasonable bounds. This is typically achieved through a quadratic cost function defined as follows:

$$J = \sum_{k=0}^{N} \left[(\mathbf{x}_k - \mathbf{x}_{ref})^T \mathbf{Q} (\mathbf{x}_k - \mathbf{x}_{ref}) + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k \right]$$

where $\mathbf{x}_{ref}$ represents the desired state trajectory, $\mathbf{Q}$ is the weighting matrix for state errors, $\mathbf{R}$ is the weighting matrix for control efforts, and N is the prediction horizon. The NLMPC controller predicts future states over this horizon and optimizes control inputs to achieve desired performance.

Additionally, NLMPC incorporates constraints on both states and control inputs to ensure the quadcopter operates within safe and feasible limits:

$$\mathbf{x}_{min} \leq \mathbf{x}_k \leq \mathbf{x}_{max}$$
$$\mathbf{u}_{min} \leq \mathbf{u}_k \leq \mathbf{u}_{max}$$

These constraints reflect physical limitations, such as maximum angular velocities and thrust limits. The prediction horizon N allows for accurate forecasting of the quadcopter's trajectory, while the control horizon M specifies how many future time steps the controller plans control inputs. Typically, the control horizon is shorter than the prediction horizon to balance computational demands and ensure effective trajectory adjustments.

2.2 PointCloud Generation Methods

There are several ways to generate a point cloud, Lidar scanners, Structure from Motion, Stereo Photogrammetry or Structured Light Scanner. The quadcopter that was intended to be used was equipped with a Intel RealSense camera [9], this camera has the ability to produce RGB-D images, where D is for depth values. The depth values are obtained from stereo photogrammetry technology assisted by infrared projections (Structured Light Scanner); Stereo photogrammetry is used to extract 3D information from 2D images by capturing the same scene from two or more different perspectives using cameras. This process is based on the principle of triangulation, where the spatial position of points in the scene is determined from the relative displacement (disparity) between the images of those points as seen by the cameras.

The cameras are separated by a fixed distance known as the baseline. When an image is captured, the same point in the scene will appear at different positions in the two images, depending on the distance of that point from the cameras. By identifying corresponding points in the images (also called tie points), the depth of the points can be calculated using triangulation.

The mathematical foundation of stereo photogrammetry is the collinearity equations. These equations describe the relationship between a 3D point in space and its 2D projection on the camera's image plane. Let the 3D point have coordinates (X, Y, Z), and let the corresponding image coordinates be (x, y). The collinearity equations are:

$$x = x_0 - f\frac{(X - X_0)R_{11} + (Y - Y_0)R_{12} + (Z - Z_0)R_{13}}{(X - X_0)R_{31} + (Y - Y_0)R_{32} + (Z - Z_0)R_{33}}$$
$$y = y_0 - f\frac{(X - X_0)R_{21} + (Y - Y_0)R_{22} + (Z - Z_0)R_{23}}{(X - X_0)R_{31} + (Y - Y_0)R_{32} + (Z - Z_0)R_{33}}$$

Where:

- f is the focal length of the camera.
- x_0, y_0 are the image coordinates of the principal point (the point where the optical axis intersects the image plane).
- R is the rotation matrix that describes the orientation of the camera in space.
- X_0, Y_0, Z_0 are the coordinates of the camera in the world coordinate system.

These equations express how a 3D point is projected onto the 2D image plane, considering the camera's position, orientation, and internal parameters [10].

The Intel RealSense camera uses infrared (IR) projection to enhance its ability to produce depth estimation. The camera projects a structured IR pattern onto the scene, which adds artificial texture to areas that might otherwise lack distinctive features. This pattern helps in identifying corresponding points between images even in low-texture environments, improving depth accuracy. The deformation of the IR pattern is analyzed by the stereo cameras to assist in triangulating the 3D coordinates of points in the scene. [11]

After corresponding points are matched and their 3D coordinates are computed using triangulation, a point cloud is generated.

Another prominent method for generating point clouds is through the use of LiDAR (Light Detection and Ranging) technology. LiDAR is an active remote sensing technology that uses laser light to measure distances and create detailed 3D representations of the environment. Unlike passive stereo photogrammetry, which relies on ambient light and image matching, LiDAR actively emits laser pulses and measures the time it takes for the light to return after hitting an object.

The basic principle of LiDAR is based on the time-of-flight measurement. The distance to an object is calculated using the speed of light and the time taken for the emitted laser pulse to return to the sensor:

$$d = \frac{c \times t}{2}$$

Where d is the distance to the object, c is the speed of light, and t is the round-trip time of the laser pulse. [12]

Obtaining XYZ Values in Local Coordinates

To obtain the XYZ values of a point in the local coordinate system of the LiDAR sensor, the distance measurement is considered alongside the angular position of the laser beam. In a typical LiDAR system with a rotating mirror that deflects the laser beam, the local coordinate system is defined with the origin at the center of the LiDAR sensor, the X-axis pointing forward, the Y-axis pointing to the left, and the Z-axis pointing upward.

The equations to calculate the XYZ coordinates of a point in this local system are:

$$\begin{aligned} x &= d\cos(\theta)\cos(\phi) \\ y &= d\sin(\theta)\cos(\phi) \\ z &= d\sin(\phi) \end{aligned}$$

Where d is the measured distance, θ is the horizontal angle (azimuth) of the laser beam, and ϕ is the vertical angle (elevation) of the laser beam.

Point Cloud Construction

The point cloud is built by aggregating the XYZ coordinates of multiple points measured by the LiDAR system. Each point P_i in the cloud can be represented as:

$$P_i = (x_i, y_i, z_i, I_i, t_i)$$

Where (x_i, y_i, z_i) are the 3D coordinates of the point in the local system, I_i is the intensity of the returned signal, and t_i is the timestamp of the measurement.

Transformation from Local to World Coordinates

To represent the point cloud in a global reference frame (world coordinates), a transformation is applied that accounts for the position and orientation of the LiDAR sensor. This transformation includes both rotation and translation:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = R \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} + T$$

Where R is the 3x3 rotation from equation 1.5 matrix and T is the 3x1 translation vector.

The translation vector T represents the position of the LiDAR sensor in the world coordinate system:

$$T = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$$

Where (X_0, Y_0, Z_0) are the coordinates of the LiDAR sensor's origin in the world frame.

By applying these transformations, the LiDAR system can generate a dense and accurate 3D point cloud representation of the environment, complementing other point cloud generation methods such as stereo photogrammetry and structured light scanning. [8]

2.3 Mapping

A three-dimensional model of volumetric space is crucial for quadcopters in obstacle avoidance and path planning. OctoMaps provide an efficient and probabilistic way to represent this 3D space. In OctoMaps, space is divided into small, cubic units called voxels. The value of each voxel represents the probability that the voxel is occupied. A value of zero indicates empty space, one indicates occupied space (an obstacle), and intermediate values signify uncertainty or unknown space due to sensor noise or lack of data.

OctoMaps are based on octrees, a hierarchical data structure that recursively divides 3D space into eight smaller regions called octants. Each node in an octree represents a cubic region of space, and as you move deeper into the tree, these regions become smaller, providing a finer level of detail. This recursive subdivision allows OctoMaps to efficiently store and represent large-scale environments by dynamically adjusting the resolution based on the level of detail needed in each region. Figure 2.1 illustrates this concept, showing both the volumetric model and the corresponding tree representation of an octree. In the volumetric model (left), It represents a 3D space divided into octants, with white areas representing free space and black areas indicating occupied space. The tree representation (right) demonstrates how this 3D structure is hierarchically organized, with each node potentially subdividing into eight child nodes. This structure allows for efficient storage and querying of spatial information at various resolutions, making it ideal for representing the complex environments navigated by quadcopters.

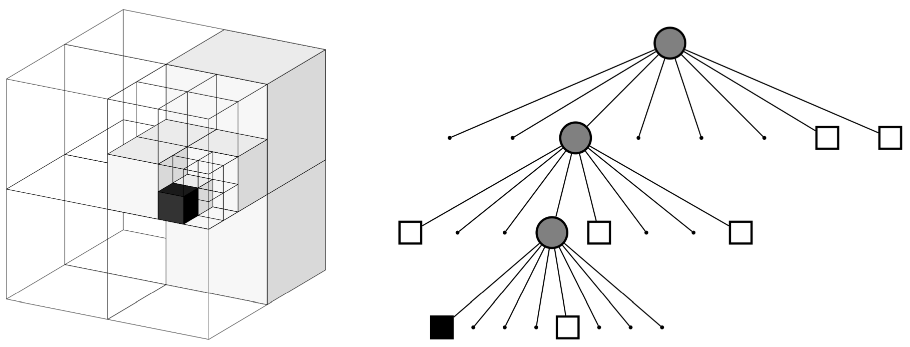

Figure 2.1: Example of an octree storing free (shaded white) and occupied (black) cells. The volumetric model is shown on the left and the corresponding tree representation on the right. [2]

For quadcopters, which need real-time updates on their surroundings to safely navigate, the octree structure of OctoMaps is particularly advantageous. Large regions of space with no obstacles can be represented at a coarse resolution, saving memory and processing power. On the other hand, areas with potential obstacles, such as trees, buildings, or other objects, are subdivided further to create a finer resolution, allowing for precise obstacle detection. Figure 2.2 illustrates this multi-resolution capability of OctoMaps. The figure shows the same environment represented at three different resolutions: 0.08 m, 0.64 m, and 1.28 m. This flexibility in resolution is achieved by limiting the depth of the octree query. Moving from left to right in the figure the level of detail changes:

1. At 0.08 m resolution (left), a highly detailed representation where small obstacles and features are clearly visible.
2. The 0.64 m resolution (center) provides a balance between detail and efficiency, where major obstacles are still discernible but smaller features are amalgamated.
3. At 1.28 m resolution (right), a coarse representation that efficiently captures large-scale structures while abstracting away smaller details.

This multi-resolution approach allows quadcopters to adapt their environmental representation based on their current needs. For instance, a quadcopter might use a coarse resolution for initial path planning over large areas, then switch to a finer resolution when navigating close to obstacles or performing precise maneuvers. This dynamic adjustment of detail helps balance the trade-off between computational efficiency and navigation accuracy, which is crucial for real-time operation in complex environments.

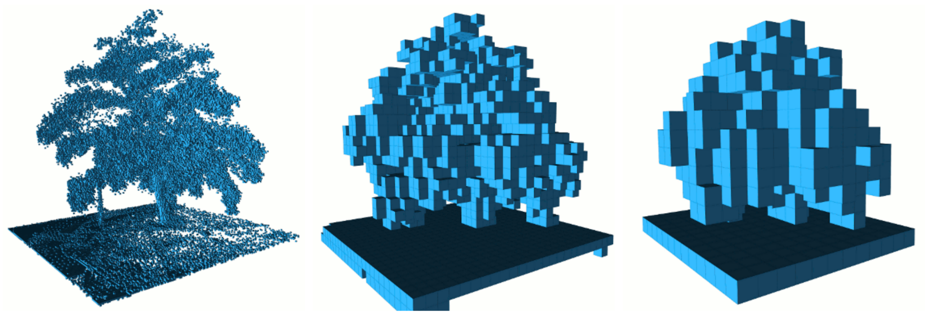

Figure 2.2: By limiting the depth of a query, multiple resolutions of the same map can be obtained at any time. Occupied voxels are displayed in resolutions 0.08 m, 0.64 , and 1.28 m. [2]

The probabilistic nature of OctoMaps makes them well-suited for environments where sensor data may be noisy or incomplete. As the quadcopter moves through space and collects data from sensors like LiDAR, cameras, or depth sensors, the OctoMap updates the occupancy probabilities for each voxel. This incremental update ensures that the map becomes more accurate over time.

To update the OctoMap, ray-casting is commonly used. When a sensor detects an obstacle, rays are cast through space, and each voxel the ray passes through is updated. Voxels that are crossed but contain no obstacle are marked as free (closer to zero), while the voxel where the ray terminates is updated as occupied (closer to one) this is visualized in Figure 2.3. Over time, this process of accumulating sensor data helps resolve the uncertainties in the map, allowing for more reliable path planning and obstacle avoidance. [2]

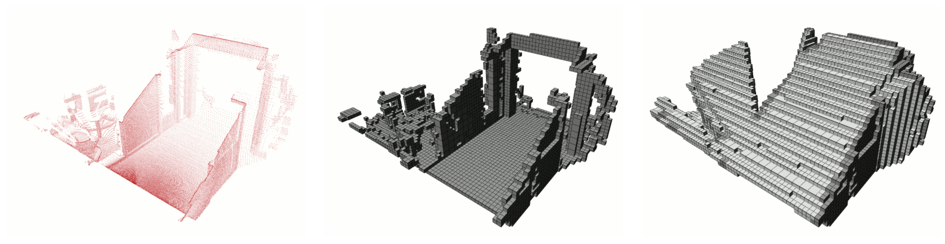

Figure 2.3: An octree map generated from example data. Left: Point cloud recorded in a corridor with a tilting laser range finder. Center: Octree generated from the data, showing occupied voxels only. Right: Visualization of the octree showing occupied voxels (dark) and free voxels (white). The free areas are obtained by clearing the space on a ray from the sensor origin to each end point. Lossless pruning results in leaf nodes of different sizes, mostly visible in the free areas on the right. [2]

2.4 Frontires

Frontier-based exploration is a common approach in robotics and SLAM (Simultaneous Localization and Mapping). In this context, a frontier represents the boundary between explored (free) space and unexplored (unknown) space. The robot or drone identifies these frontiers to decide where to move next, aiming to efficiently cover the entire environment. By continuously detecting and moving toward frontiers, the goal is to reduce unknown areas and ultimately create a complete map of the environment. This section examines two approaches to frontier detection: wavefront frontier detection (WFD) and Frontier-Tracing Frontier Detection (FTFD).

2.4.1 Detection

WFD Wavefront Frontier Detection, introduced by Keidar and Kaminka, is a method for identifying frontier cells in a robot's map. The algorithm operates on the principle of expanding a wavefront from the robot's current position through known free space until it encounters frontier cells. The algorithm begins at the robot's current location in the map. From this starting point, a Breadth-First Search (BFS) is initiated, expanding through known free space cells. As the BFS progresses, it checks each cell to determine if it's a frontier. A cell is considered a frontier if it's free and has at least one unknown neighbor. The BFS continues until all reachable free space has been explored or all frontiers have been found. WFD is more efficient than naive approaches that check every cell in the map, as it only evaluates known free space. It also guarantees finding all reachable frontiers from the robot's current position. However, in large open spaces, WFD may process more cells than necessary since it check all free cells every time the function is called, reducing efficiency as the amount of free space grows.

FTFD Frontier-Tracing Frontier Detection, proposed by Quin et al., is designed to be highly efficient, especially in open environments and for incremental mapping scenarios. It leverages information from previous frontier detections and focuses on the perimeter of newly observed areas. FTFD begins with the set of frontiers from the previous timestep. It then determines the active area based on the latest sensor scan and identifies frontiers from the previous timestep that lie within this active area. The algorithm initiates a BFS with these filtered previous frontiers and the endpoints of the current sensor scan. During the BFS, FTFD evaluates cells along the perimeter of the new scan, adding new frontier cells to the frontier set and removing cells that are no longer frontiers. The resulting new frontier set is stored for the next iteration. FTFD is particularly fast in open environments, as it only processes cells along the boundary of newly observed areas. It's well-suited for processing after each new sensor scan, making it efficient for real-time mapping. By leveraging previous frontier information, it minimizes redundant calculations. However, FTFD may potentially miss isolated pockets of unexplored space in complex environments and relies on the accuracy of previous frontier detections. [13] The FTFD algorithm can be represented in pseudocode in algorithm 1. This algorithm efficiently updates the frontier set by focusing on the active area and leveraging previous frontier information. It uses a

Algorithm 1 Frontier-Tracing Frontier Detection (FTFD) [13]

```
**Input:** Frontier set $F$, ray endpoints $rayEndpoints$, active area $A_t$
**Output:** Updated frontier set $F$, labelling $labelling$
Initialize queue with $queue \leftarrow \text{rayEndpoints} \cup (F \cap A_t)$
Initialize visited set $visited \leftarrow \emptyset$
while $queue \neq \emptyset$ do
    Extract current cell $c \leftarrow \text{POP}(queue)$
    if IS_FREESPACE($c$) then
        if IS_FRONTIER($c$) then
            Update frontier set $F \leftarrow F \cup c$
        end if
        if IS_FRONTIER($c$) or IS_OBSTACLE($c$) then
            Get adjacent cells $C_a \leftarrow$ GET_ADJACENT_CELLS($c$)
            for all adjacent cells $a$ in $C_a$ do
                if $a \notin visited$ and $a \in A_t$ and IS_FREESPACE($a$) then
                    Add cell to visited set $visited \leftarrow visited \cup a$
                    Add cell to queue PUSH($queue, a$)
                end if
            end for
        end if
    end if
end while
for all $f \in F \cap A_t$ do
    if not IS_FRONTIER($f$) then
        Remove cell from frontier set $F \leftarrow F \setminus f$
    end if
end for
Perform labelling using Kosaraju's algorithm $labelling \leftarrow$ KOSARAJU_DFS($F$)
```

breadth-first search approach to explore the perimeter of newly observed areas, updating the frontier set accordingly.

2.4.2 Frontier Point Clustering

A method to cluster and simplify frontier points is outlined in "A Multi-Resolution Frontier-Based Planner for Autonomous 3D Exploration" by Batinovic et al. [14] This approach addresses the challenge of efficiently handling thousands of frontier points in 3D exploration scenarios, balancing computational efficiency with comprehensive coverage. The clustering process consists of three main stages:

2.4.3 OctoMap-based Hierarchical Clustering

The authors leverage the multi-resolution structure of the OctoMap for initial clustering. Frontier points are initially detected at the highest resolution level ($d_{max} = 16$), typically resulting in thousands of points. These are then grouped according to their parent nodes at a lower resolution level (d_{exp}), chosen based on the environment's characteristics. This process is formalized as:

$$F^j_{exp} = v^j_{exp} : v^j_{exp} = parent(v^j_f) \text{ at } d_{exp}, \forall v^j_f \in F^j_g \tag{2.1}$$

Where:

- F^j_{exp} is the set of parent frontier voxels at the exploration level in iteration j
- F^j_g is the global frontier set in iteration j

- v_f^j represents individual frontier voxels
- d_{exp} is the desired exploration depth (e.g., $d_{exp} = 14$ in their experiments)

2.4.4 Mean-Shift Clustering

The parent frontier voxels (F_{exp}) undergo further consolidation using the mean-shift algorithm. This density-based method iteratively shifts points towards areas of higher point density. The algorithm is defined as:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)x_i}{\sum_{x_i \in N(x)} K(x_i - x)} \tag{2.2}$$

Where:

- x is the center of the kernel
- $N(x)$ is the neighborhood of x
- K is the kernel function (Gaussian in this case)

2.4.5 Centroid Calculation

For each cluster resulting from the mean-shift algorithm, a representative point (typically the centroid) is computed. These points form the set of frontier candidates (F_c) for further evaluation.

2.4.6 Best Frontier Selection

The best frontier voxel is selected using an information gain metric combined with distance:

$$G(v_c) = \frac{I(v_c)}{e^{\lambda L(p_i, p_{v_c})}} \tag{2.3}$$

Where $I(v_c)$ is the information gain (estimated using a cube-based approximation), $L(p_i, p_{v_c})$ is the Euclidean distance to the frontier, and λ is a weighting factor. This multi-stage approach enables the algorithm to efficiently process large numbers of frontier points, crucial for real-time operation in complex 3D environments. By reducing thousands of points to a manageable set of candidates, it allows for more computationally intensive evaluation methods to be applied in the final frontier selection stage, improving the overall exploration strategy while maintaining computational feasibility.

2.5 Rapidly-exploring Random Trees Star (RRT*)

The quadcopter must navigate from point A to point B while avoiding obstacles and ideally following the shortest path. To achieve this, the RRT* (Rapidly-exploring Random Tree Star) algorithm has been developed for path planning, which produces an optimized path that meets these requirements.

RRT* works by randomly generating points within the environment and connecting them to the closest node in an existing tree structure. Figure 2.4 illustrates this expansion process. In this image, there are several key elements:

- x_init (red dot): The starting point of the path.
- x_goal (green dot): The target or goal point.

- x_near (blue dot): The nearest existing node in the tree to a newly sampled random point.
- x_new (blue dot): The new node to be added to the tree.
- x_rand (blue dot): A randomly sampled point in the configuration space.
- λ (lambda): The step size or maximum allowed distance between x_near and x_new.

The solid blue lines represent existing connections in the tree, while the dashed line indicates a potential new connection.

The tree starts at point A (x_init) and incrementally grows toward point B (x_goal). At each step, RRT* checks if the newly generated point can be connected to the tree without colliding with obstacles. If the connection is valid, the point is added to the tree; otherwise, it is discarded, and the process repeats until either the target point is reached or a set maximum number of iterations is exceeded.

Algorithm 2 outlines the basic steps of the RRT algorithm, which forms the foundation of RRT*. The algorithm takes as input the starting point (x_init), the goal point (x_goal), and a map of the environment (Map). It outputs a feasible path (η) from x_init to x_goal.

The algorithm begins by initializing an empty tree. It then enters a main loop that runs for N iterations. In each iteration:

1. It randomly samples a point (x_rand) in the configuration space.
2. It finds the nearest existing node (x_near) in the tree to x_rand.
3. It creates a new point (x_new) by moving from x_near towards x_rand by a distance of at most λ.
4. It checks if the path between x_near and x_new is obstacle-free.
5. If collision-free, x_new is added to the tree and connected to x_near.
6. If x_new is in the goal region, the algorithm returns the path.

If the goal isn't reached after N iterations, the algorithm terminates without finding a path.

Unlike the original RRT algorithm, RRT* includes an optimization step. When a new point is added, the algorithm rewires the tree to ensure the path is shortened if a better route is found. It first checks whether there is a closer node to the new point than the one that initially connected it. If a closer node exists, the algorithm connects the new point to it instead. Additionally, RRT* evaluates the neighboring nodes to determine if their overall path length would be reduced by connecting through the new point. If a shorter path is found, the algorithm rewires the tree by replacing the old connections with the new, shorter ones.

Given infinite iterations, RRT* will converge to the shortest path between points A and B. However, the added path optimization in RRT* increases computational time compared to the basic RRT algorithm. While RRT* produces shorter and more efficient paths, this comes at the cost of longer computation per generated path.

This algorithm is particularly effective in high-dimensional spaces and environments with obstacles, making it well-suited for robot path planning tasks like quadcopter navigation. [15]

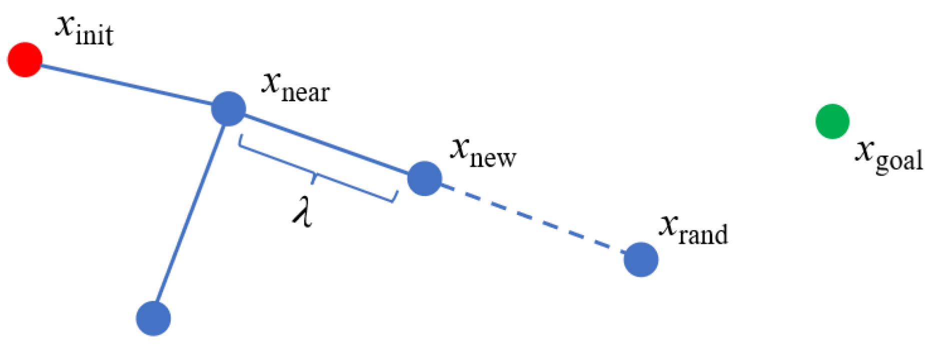

Figure 2.4: The expansion process of the RRT algorithm. The new state x_new is inserted into the search tree. [3]

Algorithm 2 RRT (Rapidly-exploring Random Tree) [3]

```
**Input:** $x_{\text{init}}$, $x_{\text{goal}}$, $Map$
**Output:** A feasible path $\eta$ from $x_{\text{init}}$ to $x_{\text{goal}}$
$Tree$.init()
**for** $i = 1$ to $N$ **do**
    $x_{\text{rand}} \leftarrow \text{UniformSample}(Map)$
    $x_{\text{near}} \leftarrow \text{Near}(x_{\text{rand}}, Tree)$
    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{rand}}, x_{\text{near}}, \lambda)$
    **if** CollisionFree($x_{\text{near}}, x_{\text{new}}, Map$) **then**
        $Tree$.AddNode($x_{\text{new}}$)
        $Tree$.AddEdge($x_{\text{near}}, x_{\text{new}}$)
        **if** $x_{\text{new}} \in X_{\text{goal}}$ **then**
            **return** $\eta$
        **end if**
    **end if**
**end for**
```

Chapter 3

Simulation

To simulate a autonomous quadcopter exploration of an unknown enviroment a combination of Matlab, Simulink and Unreal Engine 5 was used. Matlab was used to intilization the system and utilized for functions that could not be implamented easly or at all nativaly within Simulink. Unreal engine was used to simulate different three-dimensional enviroments.

3.1 Simulation Intilization

Figure 3.1 shows the overall loop required for the exploration of the unknown enviroment, the following sections step into each of the components and how each of there objectives are reached going from inputs all the way to there outputs.

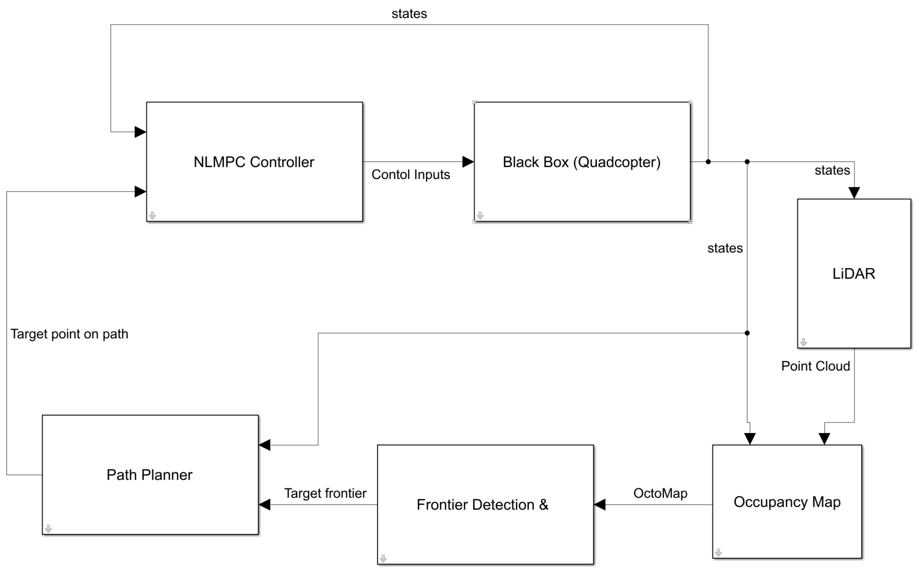

Figure 3.1: Layout of simulation architecture.

3.2 Quadcopter

The exploration loop was intended to be installed onto a Quansar Qdrone. This drone comes equipped with an Intel Aero compute board and RealSense (R200) RGBD camera. Its frame is made out of lightweight carbon fiber for advanced applications. The drone also features open software and seamless communication through Quanser's Quarc to Simulink workspace. The integration of communication with the drone being handled in Simulink drove the decision of the choice of programming workspace. Other options were a C++ implementation with simulation physics handled by Gazebo using Robot Operating System to handle the communication between code and Gazebo. Simulink was ultimately chosen due to Quarc's existing integration and its ability to simulate environments in Unreal Engine 5. To ensure safety and reliability, a simulation environment was developed before running untested code on the physical Qdrone. This simulation models the dynamics of a quadcopter using a simplified version of the true dynamics. The governing equations, shown in (1.7), were implemented directly in a MATLAB-interpreted function. While these equations omit environmental factors such as ground effect, wind, and air resistance, they provide a sufficient approximation for the project's primary focus: demonstrating the quadcopter's exploration capabilities. This simplified model allows for effective testing and development of exploration algorithms without the need for an exhaustive dynamic representation.

Another key assumption is that the quadcopter can output its states perfectly, without the need for measurement devices. This assumption raises significant concerns, as several other blocks in the system use the quadcopter's states as inputs, and they were not explicitly designed to handle the uncertainties present in real-world state estimation. As a result, this limits the solution to an idealized simulation scenario.

3.2.1 Physical attributes

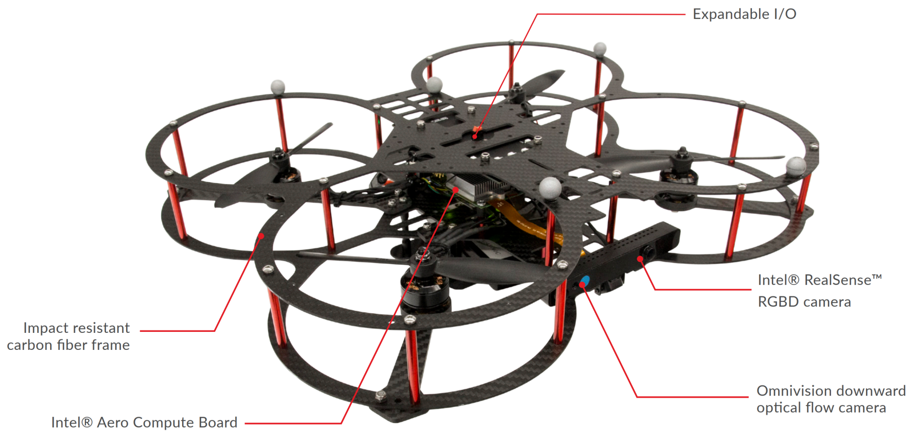
Expandable I/O
Intel® RealSense™ RGBD camera
Impact resistant carbon fiber frame
Omnivision downward optical flow camera
Intel® Aero Compute Board

Figure 3.2: Qdrone with components labeled [4]

Figure 3.2 shows the QDrone with its main components labeled. The QDrone's physical properties are crucial for understanding its performance characteristics and for accurate modeling. Table 3.1 presents the key physical attributes of the QDrone.

The QDrone's capabilities are further enhanced by its onboard equipment. Table 3.2 summarizes the major capabilities of the QDrone.

The Intel Aero Compute board and the custom QDrone expansion board are key components of the QDrone's capabilities. Tables 3.3 and 3.4 provide detailed specifications for these components.

The QDrone's propulsion system consists of four motors. Table 3.5 provides detailed specifications

Table 3.1: QDrone physical properties [5]

Specification	Value
Dimensions (L $\times$ W $\times$ H)	400 $\times$ 400 $\times$ 150 mm
Mass (M)	1.121 kg
Roll moment of inertia I_{xx}	0.01 kg m^2
Pitch moment of inertia I_{yy}	0.0082 kg m^2
Yaw moment of inertia I_{zz}	0.0148 kg m^2
Roll motor-to-motor distance L_ϕ	0.2136 m
Pitch motor-to-motor distance L_θ	0.2136 m
Vertical C.O.G location h_{cg}	0.051 m

Table 3.2: QDrone capabilities [4]

Specification	Value
Camera	Intel RealSense (R200)
Camera	Omnivision OV7251
Compute board	Intel Aero Compute board
Expansion board	Custom QDrone expansion board

Table 3.3: Specifications for the Intel Aero Compute board [5]

Item	Description
Processor	Intel Atom x7-Z8750 quad-core 64-bit 2.56 GHz
Memory	4-GB LPDDR3-1600 RAM
Storage	32-GB eMMC
FPGA	Altera Max 10 (Preconfigured for expandable IO)
Sensors	BMI160 IMU 6-DOF 16-bit triaxial accelerometer and gyroscope; BMM150 Magnetometer 3-axis geomagnetic sensor; MS5611 Barometer 24-bit pressure and temperature sensor
Wifi	IEEE 802.11 b,g,n,ac Intel Dual Band Wireless - AC 8620 2x2 MIMO
LEDs	1 tricolor and 1 orange user-programmable LED indicator

Table 3.4: Specifications for the custom QDrone expansion board [5]

Item	Description
Expandable IO	PWM (8x), UART (2x), SPI (3x SS pins), I2C, ADC (5x with 1 reserved for battery voltage reading), Encoder Inputs (3x), CPU GPIO (5x) - All 3.3V except Encoder Inputs which can be 3.3V or 5.0V
Battery voltage	Built-in voltage divider circuit to provide battery voltage measurement (via reserved ADC channel)
Regulated power supply	5V and 3.3V with 5A maximum current limit (each)
ESC Disable Switch	Disables PWM commands from the FPGA to the ESCs
LEDs	5x user programmable tri-color LED indicators

for these motors.

3.3 Nonlinear Model Predictive Control Design

Building upon the theoretical foundation established in the Control section, this section delves into the practical implementation of Nonlinear Model Predictive Control (NLMPC) for quadcopter trajectory

Table 3.5: QDrone motor information [5]

Specification	**Value**
Motors	4 x 2206 Cobra motors (2100 Kv)
Hover motor current A_{hover}	5.82 A
Motor torque constant k_t	4.5×10^{-3} Nm/A
Motor speed constant (specified) K_v	2100 RPM/V
Effective motor speed constant $K_{v,eff}$	1295.4 RPM/V
Voltage to Angular velocity offset ω_c	2132.6 RPM
Angular velocity to force offset ω_f	1004.5 RPM
Motor force constant C_t	2.0784×10^{-8} N/RPM^2
Motor force offset F_b	-0.2046 N
Motor thrust-torque constant k_τ	81.0363 N/Nm

tracking. This design leverages the Simulink NLMPC controller from the Model Predictive Control Toolbox to address the inherent nonlinearities in quadcopter dynamics.

3.3.1 Controller Configuration

The NLMPC controller is configured with the following parameters:

- Number of states (n_x): 12
- Number of outputs (n_y): 12
- Number of inputs (n_u): 4
- Sample time (T_s): 0.1 seconds
- Prediction horizon (p): 15 steps
- Control horizon (m): 2 steps

3.3.2 Cost Function and Weights

For this project a quadratic cost function was implamented for the NLMPC controller:

$$J = \sum_{k=0}^{p} \left[(\mathbf{x}_k - \mathbf{x}_{\text{ref}})^T \mathbf{Q} (\mathbf{x}_k - \mathbf{x}_{\text{ref}}) + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k \right]$$

Where:

- $\mathbf{Q} = \text{diag}([1,1,1,1,1,1,0,0,0,0,0,0])$ (state error weights)
- $\mathbf{R} = 0.01 \cdot \mathbf{I}_4$ (control effort weights)

Additionally, weights for manipulated variable rates are set to $[0.1, 0.1, 0.1, 0.1]$ to penalize overly aggressive control actions.

3.3.3 Constraints

NLMPC allows for a lot of diffenrent constraints, for this project the inputs and states are constraind.

Control Input Constraints

$$\mathbf{u}_{\min} \leq \mathbf{u}_k \leq \mathbf{u}_{\max}$$
$$\dot{\mathbf{u}}_{\min} \leq \dot{\mathbf{u}}_k \leq \dot{\mathbf{u}}_{\max}$$

Where:

- $\mathbf{u}_{\min} = [0, 0, 0, 0]^T$ and $\mathbf{u}_{\max} = [10, 10, 10, 10]^T$ (voltage limits)
- $\dot{\mathbf{u}}_{\min} = [-2, -2, -2, -2]^T$ and $\dot{\mathbf{u}}_{\max} = [2, 2, 2, 2]^T$ (change in voltage limits)

State Constraints

State constraints are applied to attitude, linear velocity, and angular velocity, while position states are left unconstrained:

- Position (x, y, z): Unconstrained
- Attitude (ϕ, θ): $[-\pi/2, \pi/2]$ radians and ψ: Unconstrained
- Linear velocity (v_x, v_y, v_z): $[-5, 5]$ m/s
- Angular velocity $(\omega_x, \omega_y, \omega_z)$: $[-2\pi, 2\pi]$ rad/s

These constraints can be expressed mathematically as:

$$\mathbf{x}_{\min} \leq \mathbf{x}_k \leq \mathbf{x}_{\max}$$

Where:

$$\mathbf{x}_{\min} = [-\infty, -\infty, -\infty, -\pi/2, -\pi/2, -\infty, -5, -5, -5, -2\pi, -2\pi, -2\pi]^T$$
$$\mathbf{x}_{\max} = [\infty, \infty, \infty, \pi/2, \pi/2, \infty, 5, 5, 5, 2\pi, 2\pi, 2\pi]^T$$

3.3.4 Prediction Model

The nonlinear prediction model is defined using:

- State function: `QuadrotorStateFcn` this function implaments equation 1.7
- Analytical Jacobian: `QuadrotorStateJacobianFcn`

These functions are derived from MATLAB's "Derive Quadrotor Dynamics for Nonlinear Model Predictive Control" example, allowing for customization of quadcopter parameters.

3.3.5 Design Rationale

1. **Prediction and Control Horizons**: The prediction horizon $p = 15$ allows for accurate trajectory forecasting, while the shorter control horizon $m = 2$ balances computational load with effective control.

2. **Output Weighting**: The first six states (position and orientation) are prioritized for reference tracking, acknowledging the limited control authority of the four motors over all twelve states.

3. **Input Weighting**: Low weights on manipulated variables (0.01) and moderate weights on their rates of change (0.1) promote smooth control actions while allowing necessary aggressiveness for trajectory tracking.

4. **Constraints**: The voltage limits [0, 10] and rate limits [-2, 2] ensure the controller respects physical actuator limitations. State constraints on attitude, linear velocity, and angular velocity reflect realistic operational limits, while leaving position unconstrained allows for unlimited range in the x, y, and z directions.

This NLMPC design strikes a balance between performance and robustness, enabling the quadcopter to effectively track desired trajectories while maintaining stable flight within physical constraints. The incorporation of the nonlinear model directly addresses the limitations of linear MPC approaches discussed in the Control section, making it particularly suitable for the complex, nonlinear dynamics of quadcopter systems.

To evaluate the effectiveness of the controller, a known reference trajectory was provided and plotted alongside the position of the quadcopter. The reference trajectory is defined by the following equations:

$$x_{\text{desired}}(t) = 6 \sin\left(\frac{t}{3}\right)$$

$$y_{\text{desired}}(t) = -6 \sin\left(\frac{t}{3}\right) \cos\left(\frac{t}{3}\right)$$

$$z_{\text{desired}}(t) = 6 \cos\left(\frac{t}{3}\right)$$

Where t is the current time in seconds.

The results in Figure 3.3 show that the controller approximated the general shape of the target function, though not perfectly. In Figure 3.4, there is a consistent phase shift of 1.1 seconds in the position states between the expected and actual paths. The cause of this phase shift was not determined, but its effect remains within an acceptable range, as expected environments do not require millimeter-level precision.

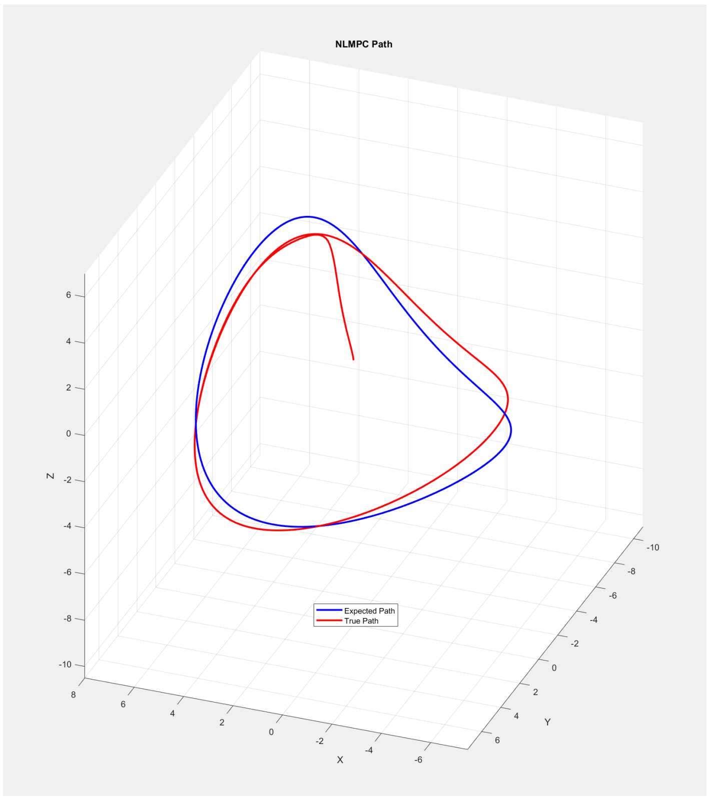

Figure 3.3: 3D Result of controller following a reference path

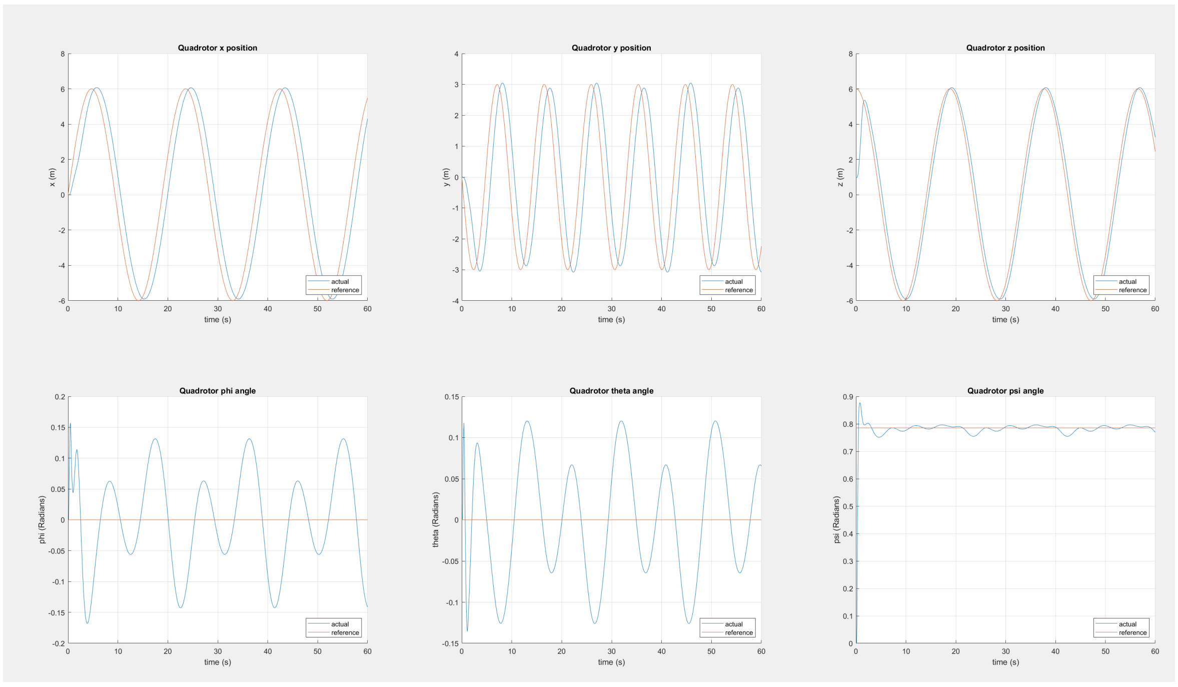

Figure 3.4: Pose result of controller following a reference path

3.4 Unreal engine and data colection

The Unreal Engine from Epic Games was used to simulate the environment and physics for the project. The maps were generated using the Maze Dungeon Generator Plugin by Zachary Kolansky to test various capabilities.

To link Simulink and Unreal Engine, the Simulation 3D Scene Configuration block establishes and maintains communication between the two applications. For the drone, the Quadcopter block from the "Simulate a Quadcopter" example was used to insert an STL model of a drone into the environment, complete with spinning propellers for visual representation. This example also provided recommendations on how to configure the related blocks. However, since the model was only intended to represent the mathematical model, the control inputs were not passed to the visual model.

To simulate the output of a RealSense camera, researchers employed a LiDAR block within the Unreal Engine environment. This block emits rays and records the points where they intersect with solid objects, generating a point cloud. However, this approach presents a challenge: it returns NaN (Not a Number) values for out-of-range measurements, which fail to update the free space along those rays. Figure 3.5 illustrates this effect. In the image, one can observe where the LiDAR beam passes through free space and exits through a gap in the right-hand wall. When the beam exceeds the maximum range, it results in NaN values being returned. Thus not being able to update the free space.

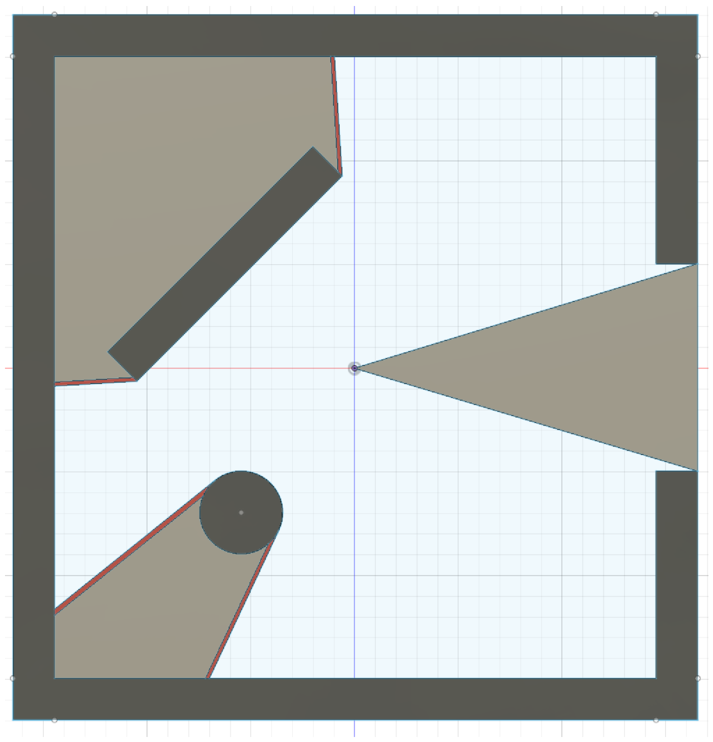

Figure 3.5: Result of a LiDAR scan. Dark gray areas represent walls, light grey indicates unknown space, red marks expected frontiers, and clear areas show known free space.

To address the issue of NaN values in the point cloud, the cloud was reshaped by removing rows containing NaNs. However, this is not the ideal solution. A better approach would be to replace NaN values with their corresponding maximum range XYZ locations. These points should be tracked, and after inserting the PointCloud, the max range endpoints should be manually updated in the OctoMap to mark them as free.

Figure 3.6 illustrates the blocks used to interface Simulink with Unreal Engine, along with their respective inputs and outputs. The input labeled 'output' represents the twelve state variables defined in equation (1.7); however, only the position and orientation variables are utilized in this configuration.

The block labeled with a question mark links to the ‘Simulate a Quadcopter’ documentation. The block managing the interface, labeled Simulation 3D Configuration, requires the file path to the Unreal Engine project and the specification of an actor, which is defined in the Quadcopter block. This configuration enables custom 3D bodies to be rendered in Unreal Engine, directly controlled by Simulink signals to represent the quadcopter accurately.

To gather LiDAR readings from Unreal Engine, the Simulation 3D LiDAR block is used. This sensor can be attached to the quadcopter body to simulate as-built locations. Similarly, the Simulation 3D Camera Get block can be attached to the quadcopter to capture image frames from its point of view.

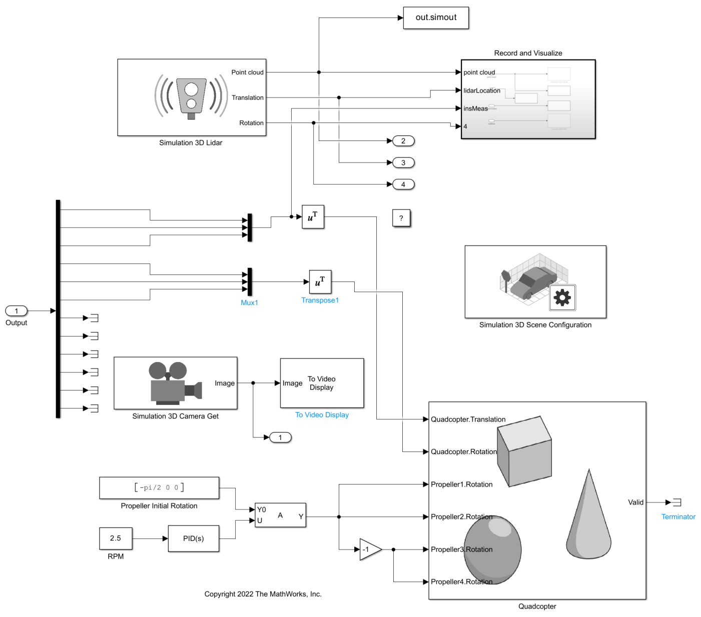

Figure 3.6: Layout of simulink interface with Unreal Engine

3.5 Mapping

The core objective is to iteratively integrate point cloud data into the map. To achieve this, an occupancy map is initialized with a minimal voxel size of $125mm^3$, balancing resolution and computation time.

When updating the map, the point cloud is inserted, but the data is initially in the local coordinates of the sensor, not the world frame. This is easily corrected by applying a rotation matrix to transform the points from the sensor frame to the world frame.

Once the point cloud is correctly positioned, ray tracing is performed from the sensor to the endpoints, updating the free space along the rays. The system operates on a probabilistic model, where the likelihood of a voxel being marked as occupied increases with the number of scan endpoints that land in that voxel. This approach allows for mapping of dynamic environment.

However, the pre-defined maximum resolution can be problematic. If objects are thinner than the voxel size, false negatives can occur, with voxels being updated as free space because more rays passed through than terminated in that voxel. This can be especially dangerous if the system relies on passive avoidance, where path planning does not actively detect obstacles but follows the map’s output. As a result, the system may plan a path through undetected objects, risking collisions.

In figure 3.7, it is evident that the thin sections of some components were incorrectly marked as

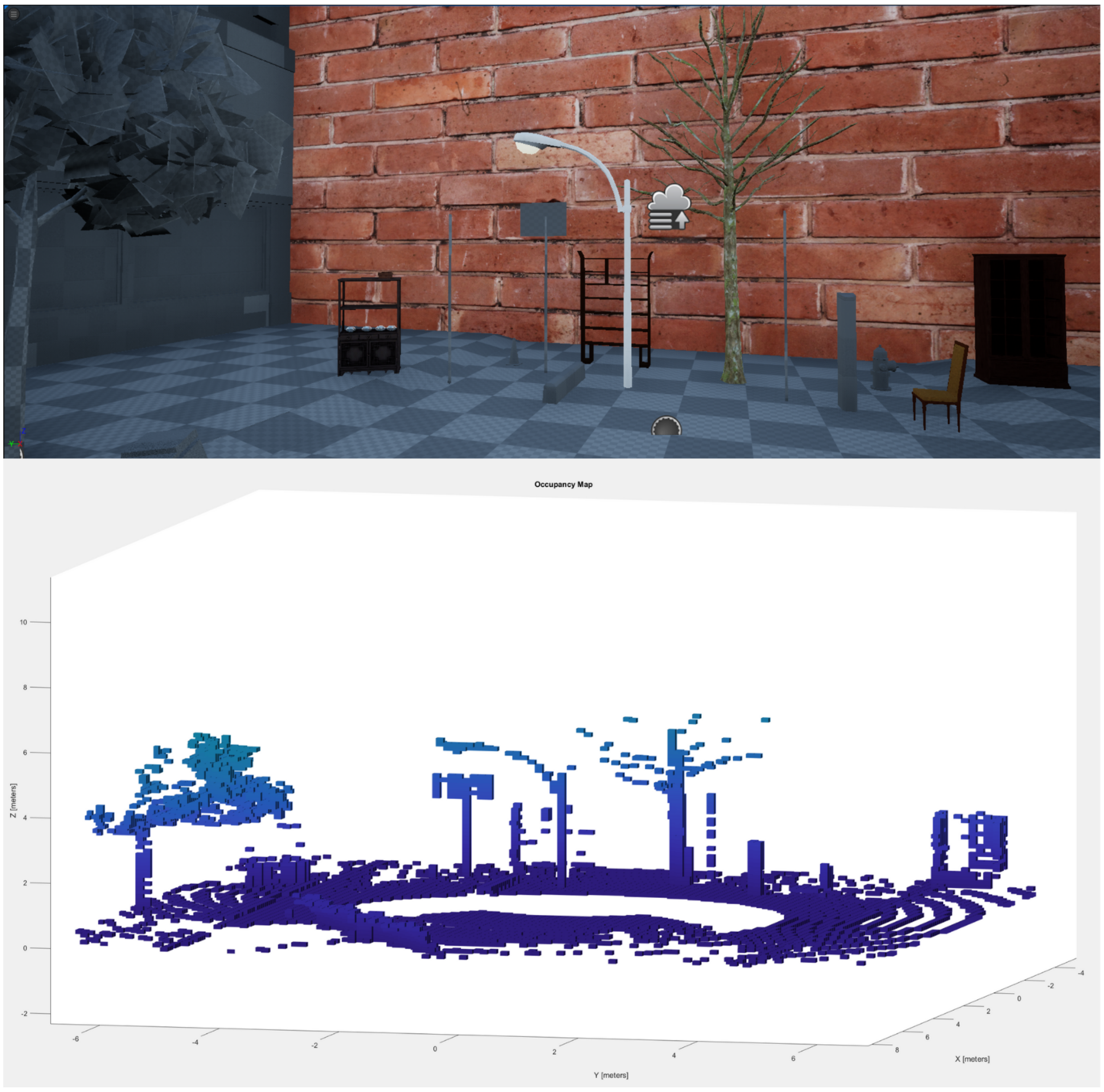

Figure 3.7: Comparison between Unreal Engine scene (top) and 3D occupancy map (bottom), with the back wall visible in the occupancy map (Appendix A).

free space. This emphasizes the importance of selecting an appropriate resolution for the intended environment. The resolution limits the system's ability to capture fine details, particularly thin objects, which can result in inaccuracies in the occupancy map.

This project assumes a relatively simple environment, where the resolution is sufficient for most objects. However, in more complex or cluttered environments, this trade-off could become a more significant issue, especially when relying on passive obstacle avoidance.

To implement the occupancy mapping, the MATLAB class occupancyMap3D was used. This class is a MATLAB implementation of the OctoMap C++ library.

Ideally, the output of this process would be an occupancy map. However, Simulink only allows signal values as outputs, not class objects. As a result, both frontier detection and path planning were consolidated into a single function call. For the purpose of this explanation, for simplicity it is assumed possible to output the map separately in the flow of information.

3.6 Frontires

The occupancy map retains a history of the locations the quadcopter has explored. Frontiers play a crucial role in identifying the boundaries between explored and unknown areas. To achieve this, two different frontier detection methods were tested, with the clustering and optimization processes being consistent across all methods.

Depending on the detection method, the inputs for frontier detection include the occupancy map, the quadcopter's pose, and the most recent point cloud. The expected output is the frontier with the highest information value that is closest to the desired distance from the quadcopter.

To minimize computational load, the detection-clustering-optimization loop is only triggered when the quadcopter reaches the final point of the previous path. Since the best frontier is used only during path planning, recalculation is only required when the path planner needs a new target location.

3.6.1 Detection

The first approach taken for frontier detection was WFD as outlined in section 2.4.1

Algorithm 3 Frontier Detection using WFD (Wavefront Distance)

```
Input: $occupancyMap$, $robotPosition$, $explorationBounds$
Output: A list of frontier cells $frontierCells$
Initialize $frontierCells$ as an empty list
Initialize a queue with the $robotPosition$
Mark all cells in the occupancy map as unvisited
while queue is not empty do
    Dequeue the first cell $currentCell$ from the queue
    if $currentCell$ is within map bounds AND not visited then
        Mark $currentCell$ as visited
        if $currentCell$ is a frontier then
            Add $currentCell$ to $frontierCells$
        else
            Get neighboring cells of $currentCell$
            for each neighboring cell do
                if neighboring cell is free AND within exploration bounds AND not visited then
                    Enqueue the neighboring cell
                end if
            end for
        end if
    end if
end while
    return $frontierCells$
```

The only input not mentioned in algorithm 3 elsewhere is the explorationBounds, which is a constraint set during the initialization of the system. It defines a bounding box for the area to be explored.

Algorithm 3 employs a breadth-first search approach, processing cells in the queue to determine whether each one is a frontier. If a cell has already been visited, it is skipped.

To identify if a cell is a frontier, the algorithm evaluates the occupancy status of the cell and its neighbors, which include the 26 interconnected voxels. A cell is classified as a frontier if it is free and adjacent to at least one unknown cell.

For cells that are not classified as frontiers, the algorithm retrieves their neighboring cells. If a neighboring cell is free, within the defined exploration bounds, and unvisited, it is added to the queue for further exploration.

Once all reachable cells have been processed, the algorithm returns the list of identified frontier cells, which can be used for planning the robot's next movements.

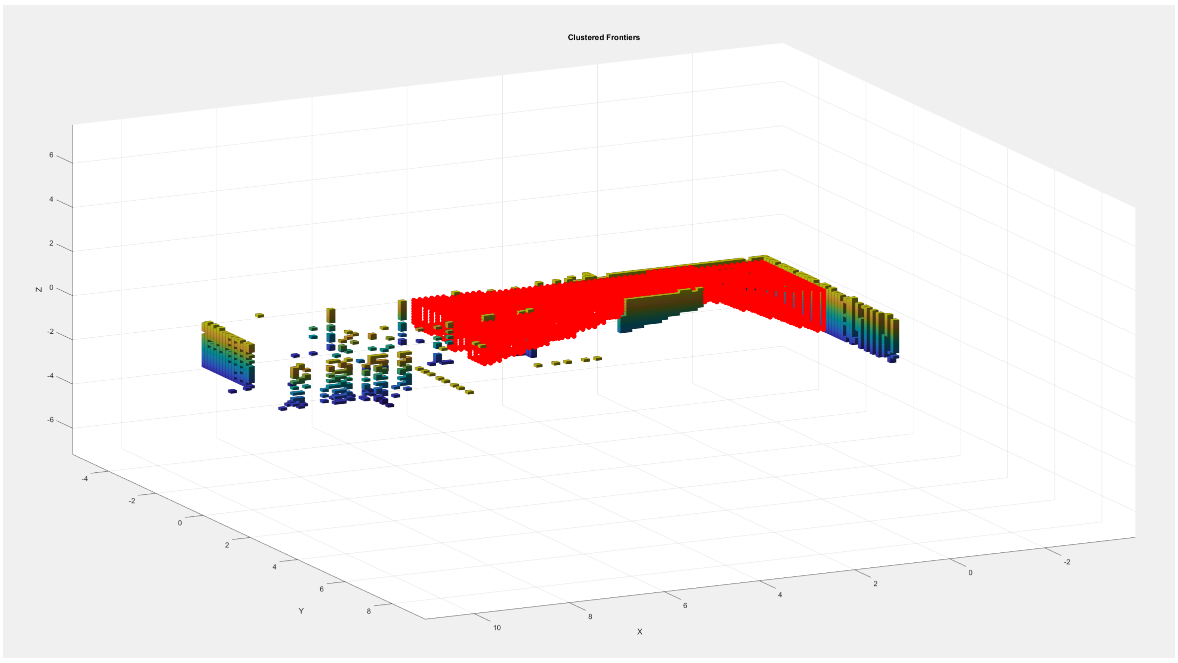

Figure 3.8: WFD found frontiers

In figure 3.8, it is evident that WFD successfully identified all frontier cells within the first few scans of the environment. However, it took 13.13 seconds for WFD to detect these frontiers, and this time increased as the size of the map grew.

One potential way to speed up detection is to limit the search to only the 6 face neighbors, rather than all 26 neighboring cells. The reason some parts of the map lack frontiers is that there are still unknown cells between the quadcopter and those regions.

An analogy for WFD is like inflating a balloon—the search expands outward from the quadcopter, stopping in directions where it encounters either unknown space or occupied cells. The portions of the balloon that touch only unknown space are returned as frontiers. This means that if there is less than $(3 \times \text{resolution})^3$ of free space in any area, the balloon will stop expanding there.

This soultion would be affective for small complex enviroments where a there is high importance of the entire area being mapped.

The second method implamented was FTFD this was implamented in the same way as algorithm 1 but with the addition of a max number of frontiers found for that loop, and a test for the amount of unknown cells surroundings the frontier to remove frontiers in explored areas. Algorithm 4 uses a breadth-first search approach to detect frontiers, processing cells from the queue to determine whether they are frontiers. It initializes with the previous frontiers and the current scan endpoints, ensuring efficient frontier detection over time.

To classify a cell as a frontier, the algorithm checks the occupancy status of the cell and its neighbors, considering the 26 neighboring voxels. A cell is marked as a frontier if it is free and adjacent to a sufficient proportion of unknown cells.

If a cell is not classified as a frontier, the algorithm retrieves its neighboring cells. If a neighboring cell is free, within the exploration bounds, and unvisited, it is added to the queue for further exploration.

Algorithm 4 Frontier Detection using FTFD (Frontier-Tracing Frontier Detection)

```
Input: $occupancyMap$, $robotPose$, $scanEndpoints$, $maxFrontiers$
Output: A list of frontier cells $frontierCells$
Initialize $frontierCells$ as an empty list
Initialize a queue with $prevFrontiers$ and $scanEndpoints$
Mark all cells in the occupancy map as unvisited
Initialize $numFrontiers \leftarrow 0$
while queue is not empty AND $numFrontiers < maxFrontiers$ do
    Dequeue the first cell $currentCell$ from the queue
    if $currentCell$ is within map bounds AND not visited then
        Mark $currentCell$ as visited
        if $currentCell$ is a frontier AND the proportion of unknown neighbors is high then
            Add $currentCell$ to $frontierCells$
            Increment $numFrontiers$
        else
            Get neighboring cells of $currentCell$
            for each neighboring cell do
                if neighboring cell is free AND within exploration bounds AND not visited then
                    Enqueue the neighboring cell
                end if
            end for
        end if
    end if
end while
    return $frontierCells$
```

Additionally, the algorithm ensures that the total number of detected frontiers does not exceed a defined maximum, maxFrontiers, to limit the processing time.

Once the queue is empty or the maximum number of frontiers has been found, the algorithm returns the list of detected frontier cells. These cells can be used for planning the robot's next actions. the location of the frontiers found using FTFD are showen in figure 3.9 There are significantly less frontiers found using this method but it only took 0.38s to find them, there location appears to be in a sensaible location However there location becomes less sensaible or explable as showen in figure 3.10 where it seams there is a repeating patterns of frontiers that are not where they wold be expected. even though there behavior was not compleatly explaneable the faster exacution time of the approach made it more favorable and preformed well enough.

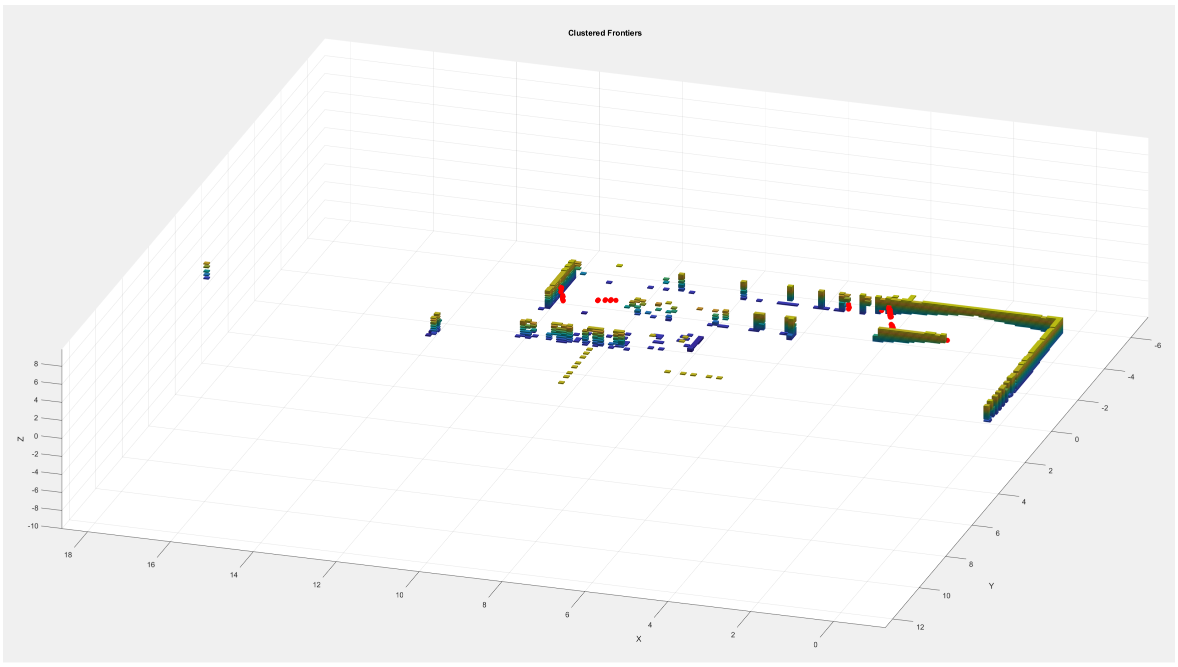

Figure 3.9: FTFD found frontiers

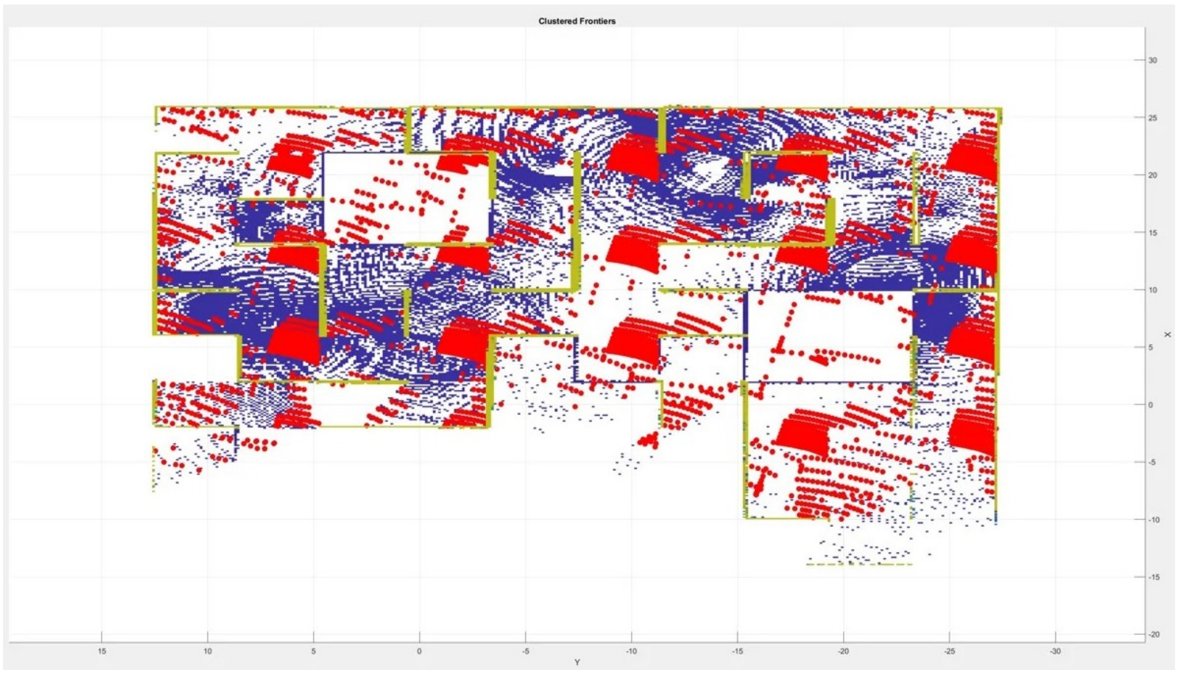

Figure 3.10: FTFD caiotic behavior

3.6.2 Clustering

The detected frontiers where then clustered to reduce the number of frontiers and also to return the center of the frontiers to maximize the potential to explore unknown space. To cluster the frontiers a MeanShift Clustering function writen by Bryan Feldman was used, it implaments the process from section 2.4.4 it takes the points and a bandwith as inputs, the bandwith is the neighborhood size to be considered, and returns the frontiers as clusters with there centers.

3.6.3 Optimization

Once the frontiers have been detected and clustered, the next step is to optimize the selection of the best frontier for exploration. The optimization process focuses on selecting the frontier that maximizes the potential for information gain while minimizing the travel distance required for the quadcopter to reach it.

The selection of the best frontier is performed using a weighted scoring mechanism based on two key factors:

- **Information Gain (IG)**: This measures the amount of unknown space the quadcopter would reveal by moving to a specific frontier. Higher information gain implies that more unknown areas are expected to be mapped, which makes a frontier more valuable for exploration.
- **Distance Factor**: This accounts for the distance between the quadcopter's current position and the candidate frontiers. Moving towards a closer frontier reduces travel time and energy consumption, but the distance is balanced with the need to reveal as much new information as possible.

The optimization is based on the following steps:

1. **Calculate Information Gain**: For each cluster, the **information gain** is computed by estimating the proportion of unknown cells in a box centered at the cluster's centroid. This is done using the `calcMIBox` function, which counts the number of unknown cells in a bounding box around the cluster. The box size is determined by a fixed parameter (`boxInfGainSize`), and the ratio of unknown cells to total cells in the box provides a measure of the information gain.
2. **Calculate Distance Factor**: The **distance factor** is calculated as a Gaussian function of the distance between the current position of the quadcopter and the centroid of each cluster. The distance factor is designed to prioritize frontiers that are closer to the quadcopter, with an optimal exploration distance defined by a parameter (`optimalDist`) a optimal distance of three meters was chosen to ballence exploration and distance the path planner had to attempt to traverse. The Gaussian function ensures that frontiers at or near the optimal distance receive a higher weight, while those much closer or farther away are penalized.

   The equation used for the distance factor is:

   $$\text{distance factor} = \exp\left(-\frac{(d - \text{optimalDist})^2}{2\sigma^2}\right)$$

   where:

   - d is the Euclidean distance between the quadcopter and the frontier,
   - σ of one was chosen to have a high weighting on desired distance.
3. **Score Calculation**: For each cluster, a weighted score is calculated based on both the information gain and the distance factor. The score is given by the equation:

   $$\text{score} = k_{\text{Gain}} \times \text{Information Gain} + k_{\text{Dist}} \times \text{Distance Factor}$$

where k_{Gain} and k_{Dist} are scaling constants that control the relative importance of information gain and distance. By adjusting these constants, the system can prioritize either gathering more information or minimizing travel distance based on the mission's requirements.A one-to-one relationship was found to be effective but could be further refined if needed.

4. **Select the Best Frontier**: Once all clusters have been scored, the frontier with the highest score is selected as the best frontier for the next exploration step. This approach balances the exploration of unknown space with the need to minimize the quadcopter's travel time.

The effectiveness of this optimization can be seen in the trade-off it strikes between exploring large amounts of unknown space and ensuring efficient navigation. If the quadcopter consistently prioritizes distant frontiers, it could end up wasting time and energy. Conversely, if it always chooses the closest frontier, it may fail to explore more informative areas. This method ensures a balanced exploration strategy by weighting both information gain and distance.

3.7 Path Planning

In the quadcopter navigation system, frontier detection identifies a target location, while mapping generates an occupancy map of the environment. With these components in place, the path planner can effectively plan a route from the current location to the target. This is achieved using MATLAB's built-in planner class, which includes various path planning algorithms, including the RRT* (Rapidly-exploring Random Tree Star) algorithm.

The RRT* planner requires specific inputs, including a state space and a state validator, to effectively plan the quadcopter's path while considering its dynamics. The state space of SE3 (Special Euclidean group in 3D) is appropriate because it allows the quadcopter to move independently in all directions, enabling unrestricted positional navigation without imposing constraints on its pose. SE3 encompasses both the position (x, y, z) and orientation (ϕ, θ, ψ) of the quadcopter, facilitating the planning of complex maneuvers.

However, while the SE3 framework allows for flexibility in movement, it does not inherently account for the quadcopter's dynamics, particularly during sharp turns or rapid direction changes. Such maneuvers can introduce significant errors in path following, potentially causing the quadcopter to drift into obstacles.

The state validator is constructed using the occupancy map, which defines the environment's obstacles. It incorporates a minimum validation distance of 0.1 m from occupied or unknown cells to reduce the likelihood of collisions. Additionally, the occupancy map is inflated by 0.125 m to create a safety buffer around obstacles, further ensuring the quadcopter can navigate safely. Together, these inputs enable the RRT* planner to generate feasible paths that account for both the quadcopter's movement capabilities and environmental constraints.

By integrating the SE3 state space with a robust state validator, the RRT* planner can effectively navigate the quadcopter through complex environments while minimizing collision risks and accounting for its dynamic behavior.

The path planner implements a variant of the RRT* algorithm as outlined in Section 2.5, with the inclusion of state space checks. The behavior of RRT* can be influenced by several parameters, such as the maximum number of iterations, goal bias, and connection distance, which were discussed in the previous section. These parameters control how aggressively the tree grows toward the target, how many random samples are considered, and the maximum distance between nodes.

The chosen parameters for the planner were set to balance computational efficiency and path accuracy. For example, a goal bias of 0.1 was selected to ensure that the planner focuses on reaching the target, while still maintaining enough randomness to explore alternative paths when necessary. The maximum connection distance was set to 0.1 meters, In an attempt to navigate through narrow free space.

Additionally, the maximum number of iterations was set to 20,00 to ballence having enough iterations to reach the loaction but also to fail quickly if target location has an infeasible path.

The planner outputs a set of nodes from the current position to the end location, and the resulting path is interpolated to make it denser, allowing the controller to follow a smoother trajectory. This reduces the risk of abrupt directional changes that could cause instability in the quadcopter's flight.

However, there are instances where the planner does not find a path to the target. The exact reasons for this are uncertain, but potential causes could include:

- The target location being too far from the starting point for the given number of iterations.
- Narrow gaps in free space that fail the state validator's collision checks.

In such cases, the control loop is slowed down as new frontiers are generated, and the planner reattempts to find a viable path. This loop may eventually produce a reachable target or, conversely, repeatedly attempt to reach the same unreachable point. One mitigating factor is the quadcopter's continuous rotation, which updates the occupancy map and generates new potential frontiers for the planner to explore.

An alternative strategy could involve selecting the closest node in the tree to the target location and reconstructing a partial path from there. However, this method does not guarantee improved exploration efficiency, as it may result in the quadcopter spending unnecessary time traversing previously explored areas rather than progressing toward new regions.

In figure 3.11, a series of paths are shown. The start of each path is represented by a green circle, and the target location is indicated in red. The red lines illustrate the found paths, while the green lines represent the other branches of the trees leading to the found paths. The yellow lines correspond to the branches of a failed path planning attempt. The environment is three-dimensional but is shown as two-dimensional for simplicity, allowing for a clearer visualization of the paths and tree branches.

From observation of the figure, when a path is found, very few alternative branches are created, indicating a more direct approach to the goal. However, when the path fails, the branches of the tree explore the environment more extensively. The RRT* algorithm is capable of finding a path across long distances, but at times it fails. To better understand these failures, it would have been beneficial to visualize both free and unknown space, which could reveal more about the behavior of the path planner.

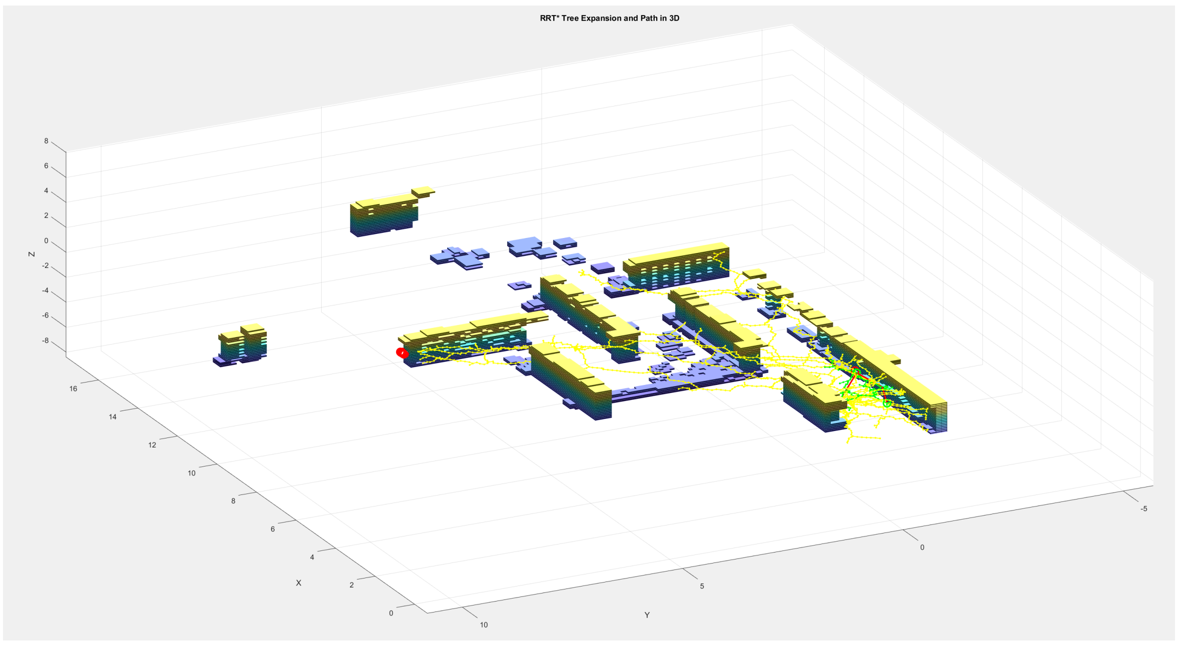

Figure 3.11: RRT* series of paths

Chapter 4

Results

With path planning implemented, the control loop is complete, allowing for analysis of the solution's effectiveness. The first test examined how well the system worked in large complex environments that could be compressed into two-dimensional maps using FTFD. The control loop ran for ten minutes and produced a map and paths shown in Figure 4.1. This can be compared to the true environment shown in Figure 4.2.

From visual inspection, the majority of the 20 m^2 environment was successfully mapped, though the quadcopter exhibited some notable behavioral patterns. Starting from the origin, it showed a tendency to get stuck exploring the same areas repeatedly. This behavior can be attributed to the frontier detection mechanism's design, which relies solely on the current LiDAR scan for initialization. When no new frontiers are detected in the current scan, the system defaults to examining previously identified frontiers. While this fallback mechanism effectively prevents the quadcopter from becoming trapped in dead ends by enabling backtracking, it also increases the likelihood of revisiting well-explored areas instead of prioritizing unexplored regions.

The path planning component may also contribute to this behavior, as it sometimes fails to plan paths through narrow free spaces to reach optimal frontiers. This is evidenced by the red circles without connecting path lines, which indicate target locations where path planning failed.

Some sections of the map were mapped incorrectly. This is particularly evident at coordinates (8, 20), where the quadcopter never achieved a position allowing the LiDAR to detect the wall segment. While this could indicate data collection issues from Unreal Engine, this remains speculative.

The test environment was deliberately constructed with partial flooring and no roof to focus frontier detection on wall mapping rather than floor mapping and to improve result visualization.

A map was made to test its capabilities in a small but complex three-dimensional environment. The simulation was designed to run for ten minutes but was terminated at five minutes due to no movement for a considerable time. The results are shown in Figure 4.3. This figure shows that it was able to conduct limited three-dimensional exploration; however, it got stuck in a location where the frontiers chosen were not achievable for the path planner. This is illustrated by the high density of red circles with no lines connecting them. The location being in a well-explored area indicates that the FTFD was not performing as expected and may need to validate its performance to determine how to improve its reliability.

The performance of FTFD in a small three-dimensional environment was compared to WFD. However, there is a difference in the path planning process, so the results are not directly comparable. The difference is that WFD will find all frontiers around the drone, meaning that the best frontier will be the same if the path planning fails and the frontier detection is conducted again, which could lead to getting stuck trying to find a path to the same point. To rectify this problem, the closest node of

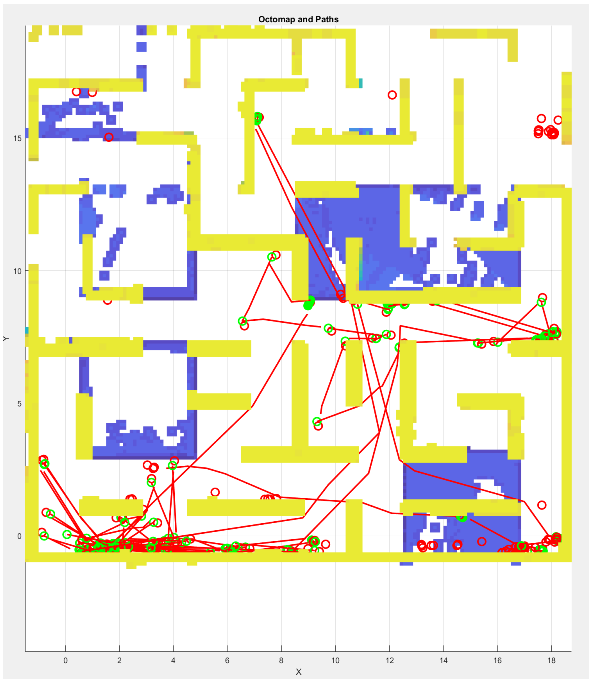

Figure 4.1: FTFD exploration results of large environment

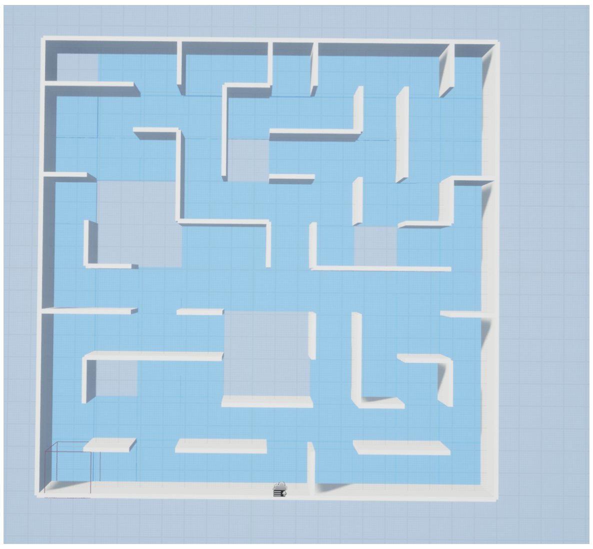

Figure 4.2: Large environment true map

the path tree to the target location was taken, and a partial path was reconstructed. However, this process was not validated to ensure it worked as required.

The results are shown in Figure 4.4. WFD was more successful at exploring the three-dimensional environment; however, there are some sections of concern. There are a few paths where they were planned through walls and floors, likely due to an incorrect reconstruction of a partial path. This test was meant to run for ten minutes; however, due to the time it can take to find frontiers, the synchronization between Simulink and Unreal Engine was broken, causing the simulation to terminate. Even though the path planning was not performing as expected, WFD was still able to provide frontiers that explored more of the environment compared to FTFD.

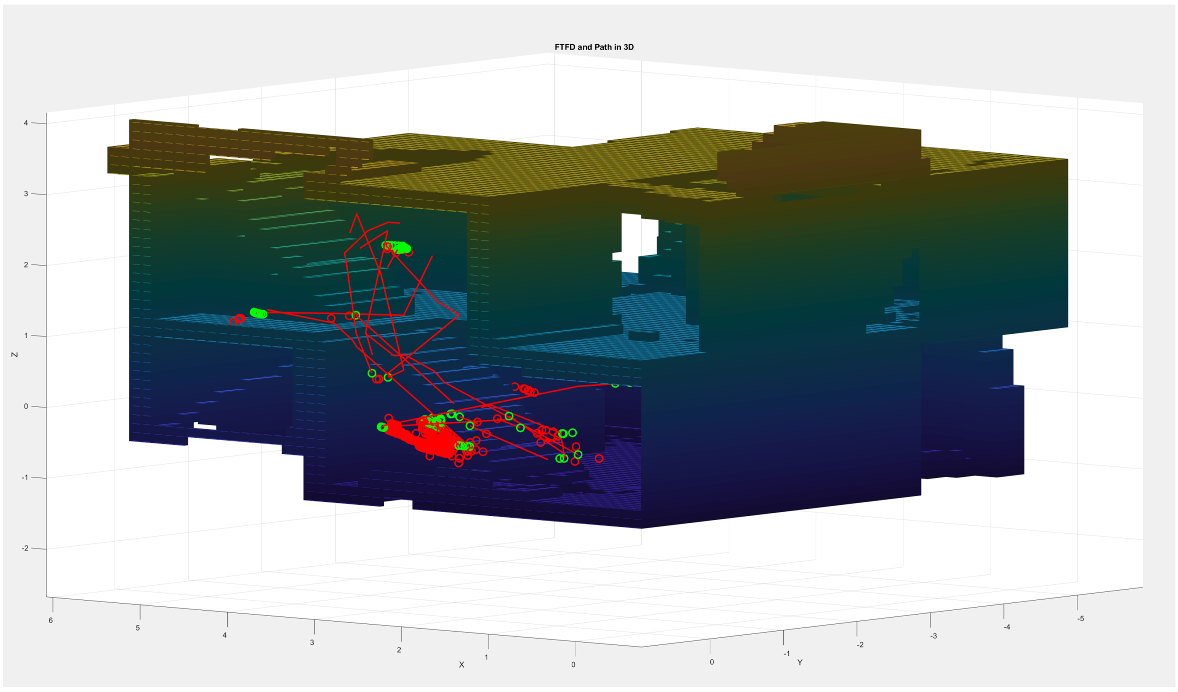

Figure 4.3: FTFD 3D environment results

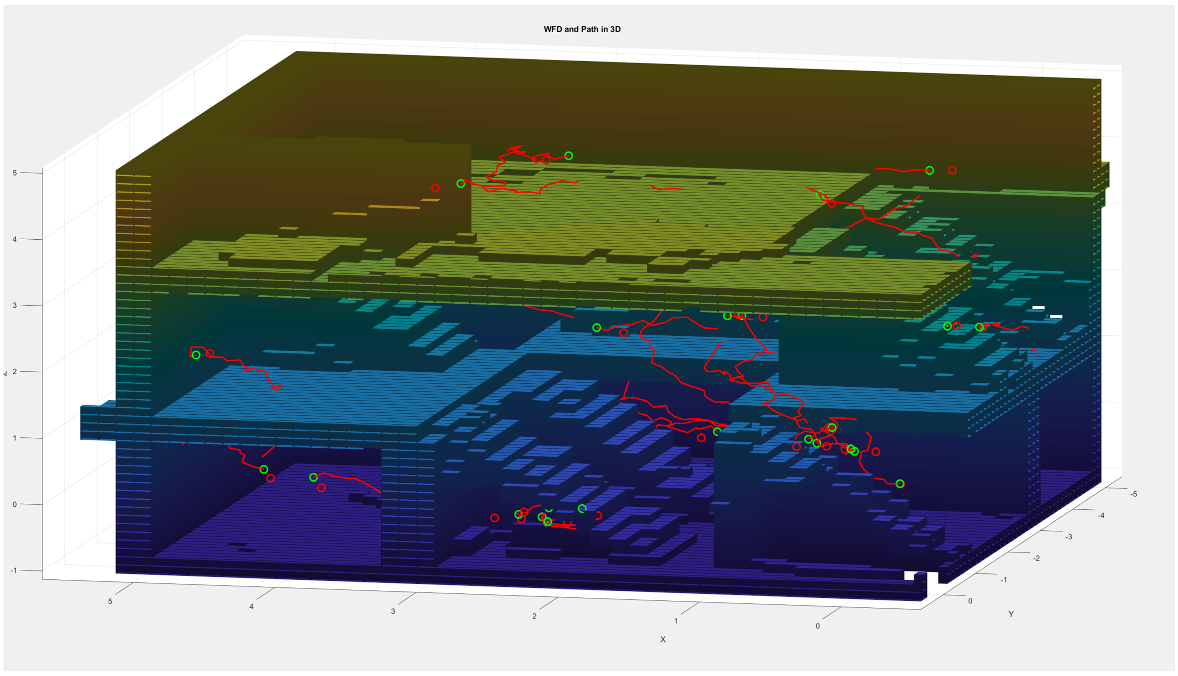

Figure 4.4: WFD 3D environment results

4.1 Future Work

Before any additions can be made to this project, several current areas need to be addressed. The frontier detection methods require further validation, and a more robust path planning procedure is needed. Once these areas are improved, some of the assumptions made in this report could be reconsidered. For instance, assuming access to the true states of the quadcopter is unrealistic, so a state estimator could be added.

Another enhancement would be to implement active object avoidance, which would aid in path planning by allowing paths to be planned through unknown space in the occupancy map. To make the quadcopter more versatile, object classification could be added, enabling it to function as both a mapping and search drone.

Additionally, to increase the project's transferability and enable real-time operation, the codebase could be migrated to an embedded systems language, such as C++.

4.2 Conclusion

This project aimed to develop an exploration quadcopter to explore unknown environments. It successfully mapped these environments using OctoMaps, identified frontiers through WFD, and attempted FTFD, although further testing is required to evaluate its implementation. The quadcopter was able to plan paths to the best frontiers using RRT* and control its movements with a Nonlinear Model Predictive Control (NLMPC) controller. Despite some limitations, this project serves as a proof of concept that these principles can effectively facilitate the exploration of three-dimensional environments, making it valuable for applications in search and rescue operations.

Bibliography

[1] M. S. Esmail, M. H. Merzban, A. A. M. Khalaf, H. F. A. Hamed, and A. I. Hussein, "Attitude and altitude nonlinear control regulation of a quadcopter using quaternion representation," *IEEE Access*, vol. 10, pp. 5884–5894, 2022.

[2] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, software available at https://octomap.github.io. [Online]. Available: https://octomap.github.io

[3] X. Li and Y. Tong, "Path planning of a mobile robot based on the improved rrt algorithm," *Applied Sciences*, vol. 14, no. 1, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/14/1/25

[4] Quanser, "Qdrone product data sheet v1.2," 2018, accessed: 2024-10-27. [Online]. Available: https://www.quanser.com/wp-content/uploads/2018/02/Qdrone-Product-Data-Sheet-v1.2.pdf

[5] ——, "Qdrone product page," 2024, accessed: 2024-10-27. [Online]. Available: https://www.quanser.com/products/qdrone/

[6] M. B. Mamo, "Control of quadcopter by designing nonlinear PID controller," 2024, includes performance analysis of Nonlinear PID control for quadcopter altitude, pitch, roll, and yaw control.

[7] F. Sabatino, "Quadrotor control: modeling, nonlinearcontrol design, and simulation," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:61413561

[8] G. V. Raffo, M. G. Ortega, and F. R. Rubio, "An integral predictive/nonlinear $h\infty$ control structure for a quadrotor helicopter," *Automatica*, vol. 46, no. 1, pp. 29–39, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0005109809004798

[9] I. Corporation, "Intel realsense camera r200: Embedded infrared assisted stereovision 3d imaging system with color camera," https://www.mouser.com/pdfdocs/intel_realsense_camera_r200.pdf, Intel Corporation, Product Datasheet 334616-001, Jun. 2016, part Number: MM#939143, Revision 001.

[10] N. Zomrawi, "The solution of collinearity condition equations with 6-terms via 10-terms," 02 2008.

[11] J.-c. Jeong, H. Shin, J. Chang, E.-G. Lim, S. M. Choi, K.-J. Yoon, and J.-i. Cho, "High-quality stereo depth map generation using infrared pattern projection," *ETRI Journal*, vol. 35, no. 6, pp. 1011–1020, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.13.2013.0052

[12] Synopsys, "What is lidar?" 2024, accessed: 2024-10-27. [Online]. Available: https://www.synopsys.com/glossary/what-is-lidar.html

[13] P. Quin, D. D. K. Nguyen, T. L. Vu, A. Alempijevic, and G. Paul, "Approaches for efficiently detecting frontier cells in robotics exploration," *Frontiers in Robotics and AI*, vol. 8, 2021. [Online]. Available: https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2021.616470

[14] A. Batinovic, T. Petrovic, A. Ivanovic, F. Petric, and S. Bogdan, “A multi-resolution frontier-based planner for autonomous 3d exploration,” *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4528–4535, 2021.

[15] T. Chinenov, “Robotic path planning: Rrt and rrt* - exploring the optimized version of an orthodox path planning algorithm,” Medium, February 14, 2019, 2019, accessed: 2024-10-27. [Online]. Available: https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

Appendix A

Extra results

To demonstrate that figure 3.7 the undetected objects are not a resualt of the LiDAR block returning nan values and thus staying as unknown space, this is everdent of the wall being detected behind the items that where not detected.

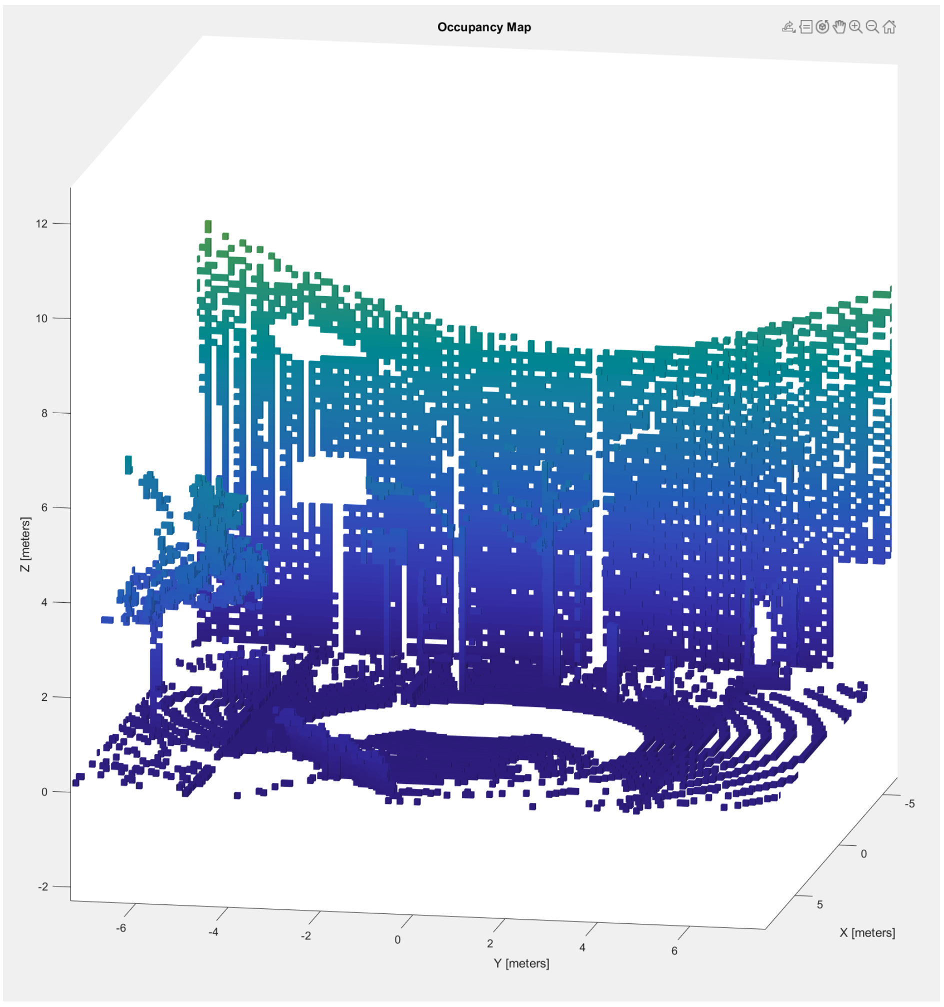

Figure A.1: Occupancy map with back wall to show that the reason for missing detail is not from nan values in the LiDAR data.