Vision-Based Autonomous Mapping & Obstacle Avoidance for a Micro-Aerial Vehicle (MAV) Navigating Canal

A Master Thesis
Presented by

Syed Izzat Ullah

In Fullfilment
of the Requirements for the Degree of
Master in Electrical Engineering

Supervisor: Abubakr Muhammad (LUMS)
Co-supervisor: Murtaza Taj(LUMS)

Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences
October 2019

Acknowledgments

In the name of Allah, the most gracious and the most merciful, I would like to extend my heartfelt gratitude to my advisor Dr.Abubakr Muhammad for his continuous support in completion of my MS thesis: for his indefatigable patience, motivation, enthusiasm, and immense knowledge which helped me in accomplishment of this thesis.

Moreover, I also owe gratitude to the National Center for Robotics & Automation (NCRA), LUMS, Pakistan for enabling me to learn, enhanced my knowledge and make use of practical knowledge. I also thank the DAAD grant DyMaSH: "Dynamic Mapping and Sampling for High Resolution Hydrology" for providing me an opportunity for learning and international exposure.

I thank my fellow labmates in Agritech and Control & Robotics lab: Abbas, Zahoor, Ansir, Waseem, Musab, Zafar, Shabbir, Hashim, Affan, Farooq, and Joudat for their productive feedbacks and stimulating discussions which continuously contributed towards improvement of my knowledge about the research and help me in completing my research successfully. I find it hard to pay enough regards to my friend Mateen for all the support and help during the difficult times.

Last but not least, my deepest gratitude goes to my beloved parents, sisters and brothers for their endless love, prayers and encouragement. Also, not forgetting my wife for her love and care. Thank you for supporting me spiritually throughout my life.

I am very thankful to Allah Almighty, who bestowed his blessings over me and helped me through all the ups and downs during this period. May He keep His blessings over us to provide our services for the goodness of humanity.

Abstract

For optimal agriculture yields, the land designated for agriculture requires proper irrigation. The major part of Pakistan's irrigation system is served through canals that are outflows of dams, barrages, and rivers. The total length of the irrigating canal network in the Indus Basin is about 57,000 kilometers[47]. The manual labor for inspecting and analyzing the condition of water canals and then repairing during the closure period each year cannot be possible by laborious manual inspection. Hence, there is a need for automation in inspection tasks related to the water channels.

The capability of moving robots in accomplishing desired tasks with the help of a couple of sensors mounted on them, not only decreases computation time but guarantees task completion as well. To be able to use robots for canal-like environments, one requirement is obstacle avoidance. We have developed a near to real canal environment in the Unreal engine, with real textures of trees, brushes, and bridges. Various conceivable obstacles that an aerial vehicle can confront in a canal-like environment is added in the simulation.

Robot Operating System (ROS)[10] plays an important role in the simulation as it offers methods and groups of useful libraries (libraries for frame transformations, point cloud processing, visualization, and data monitoring to name a few). The Airsim plugin[55] in the Unreal engine[11] is used to simulate the MAV in the canal. We used *Stereo_image_Proc*[9] for disparity and point cloud construction, *Octomap_server*[61] for Octomap Mapping, Informed RRT *[31] for path planning, and *Airsim_Simple FlightController*[55] for vehicle control.

The vehicle is capable of profiling the canal channels quickly and effectively that will assist the human operator in surveying the canal during an annual canal closure. The developed system autonomously flies over the canal, not only build a three-dimensional map of the canal environment but detects obstacles in the path of the vehicle and eventually avoid these obstacles.

This Master thesis has been examined by a Committee of the Department of Electrical Engineering as follows:

Dr Abubakr Muhammad.....
Thesis Supervisor
Associate Professor of Electrical Engineering

Dr Murtaza Taj .....
Thesis Co-Supervisor
Assistant Professor of Computer Science

Dr Ahmed Kamal Nasir.....
Member, Thesis Committee
Assistant Professor of Electrical Engineering

Contents

List of Figures

Chapter 1

Introduction

Applications and uses of UAVs (Unmanned Aerial Vehicles), also colloquially known as drones, are drawing a lot of interest in the recent years. UAVs have potential to bring revolution in various fields like logistics, agriculture and defense, to name a few. However, a number of research works are still needed to realize robust, smart, and truly autonomous UAVs. Some of these challenges are concerned with obstacle avoidance and autonomous navigation.

Given the recent innovation and research outcomes in navigation and path planning, this thesis explores opportunities to enable and improve functionalities in UAVs using state-of-the-art techniques.

1.1 Motivation

Agriculture is Pakistan's largest sector of the economy and seventy percent of Pakistan's population is mainly dependent on agriculture. Agricultural land needs to be efficiently and effectively irrigated for optimal agriculture yield. The total length of the irrigating canal network is about $57,000$ kilometers which is approximately equivalent to the one and a half of the equatorial circumference of Earth [47]. Due to improper water management, the water table rose, resulting in waterlogging and salinity and about 25% of the irrigating area of Pakistan is being affected by it. The manual labor for inspecting and analyzing the condition of water canals and then

repairing during the closure period each year cannot be possible by laborious manual inspection.

Different groups have tried to use different robotics platforms to solve this problem. One such robot could be a small scale Micro-aerial Vehicle (MAV), that has the capability of flying in the 3D environment and navigate across overhanging trees and other possible obstacles in the canal like environment. A vehicle equipped with a 3D perception system capable of profiling the canal channels quickly and effectively, to assist the human operator in surveying the canal, during an annual canal closure period, avoiding collision with the tree branches, bridges, and other possible obstacles.

Another avenue where this system could be of use in the canal-like environment is that it can recover a floating sensor. The sensor is developed in our lab at the center for water informatics (WIT) that spatially monitors water quality [15]. After the sensor is released in a canal, it is difficult for a human to recover. Since this aerial platform is able to avoid any obstacle in the canal environment so it is possible to recover the float via this system using a robotic arm mounted on the vehicle.

Figure 1-1: Mobile water quality monitoring sensor developed at Center for Water Informatics (WIT) [15]

1.2 Problem Statement

This thesis mainly focuses on the problem of avoiding obstacles in a 3D, cluttered, unknown environment with a Micro-aerial vehicle MAV. The idea is to develop a flying autonomous aerial vehicle equipped with high-resolution cameras and other sensors

that not only build a 3D map of the canal environment but also detect obstacles in the path of drone and eventually avoid these obstacles.

1.3 Related Work

The use of MAVs in an outdoor cluttered environment such as water channels is an active research topic with rising popularity. Multiple research groups use Micro-aerial vehicles MAVs to obtain an overview of the water channel geometry. Others have tried small boats, lightweight inflatable craft [30], but their systems are unable to map the area below the canopy and above the canal banks, if the mapping sensors are mounted at a certain altitude, the probability of collisions with the lower hanging tree branches increases, and the vehicle will not be able to navigate the canal's cluttered area. On the other side, fixed-wing UAVs at higher altitudes [48] are unable to map the area under the canal canopy, since they are mapping from higher altitudes. In simple riverine environments, this platform could be more practical but our goal is to build a system capable of performing in the most problematic situations such as flowing water, blocked rivers or dense canopies of the canals. Preliminary work in canal mapping on small multirotor using passive vision and ultrasonic sensors (for altitude estimation) has been described over short distances [63]. Our work is in the same way motivated but we explicitly deliberate substantially longer missions in which it is important to not only map the magnitude of the river but also to map the vegetation along the bank of the canal and avoid obstacles that might appear in the middle of the canal. We focused on autonomous exploration, where onboard sensors perceive the environment and the vehicle plans obstacle-free routes that navigate the vehicle along the canal and perform the mapping. Similar work for autonomous river mapping and exploration through an aerial vehicle is presented by [52], where a separate frontal looking camera is used for river bank detection and a 2D LiDAR with a rotating mechanism and stereo camera is used to compute the 3D structure of the river. The system is tested on a 2km long river. However, because of having two separate sensors for mapping and exploration requires more computations, memory,

and power.

We achieve this with simulating a stereo camera and a 2D laser scanner being the perception sensors for local obstacle avoidance to navigate over the canal. However, any prior information is not being used, except local goal points from the GPS, and we rely on our local sensors for path planning and collision avoidance. The stereo camera in our setup serves as a primary observation sensor for 3D obstacle avoidance and mapping of the canal, LiDAR on the other hand, add some belief in the occupancy of stereo detection in the 2D LiDAR scan around the MAV.

The focus of this thesis is Obstacle avoidance, path planning, and Mapping for a flying platform. We will discuss the related work in each of these areas one-by-one in detail.

1.3.1 Related Work on Obstacle Avoidance

Obstacle avoidance is a crucial capability for a Micro-aerial Vehicle (MAV) to maneuver close to the trees present at low altitude. The implementation of a robust collision avoidance system, also known as an anti-collision system, is a big challenge to take for greater autonomy at the moment. It allows for new applications and reduces the pilots' skill requirements. That is why collision avoidance is a research and development area of interest.

Scherer [53] and Achtelika [14] uses laser scanners to detect obstacles. Gronzka et al. ([33], [34]), who uses multilevel SLAM with motion and altitude estimation for 3D mapping, positioning, and navigation, has extended this approach. Their implementation is based on the Mikrokopter and is capable of independent flight [6]. The Mikrokopter project began as an open-source platform but is now commercially available. Nevertheless, owing to its technical strain, data processing is not done on-board, but on an external laptop. Another SLAM algorithm was developed by Blösch [19] which also requires external hardware for data processing. Shen [56] combined a laser scanner with a camera for position estimation using the closest iterative point (ICP) algorithm and an extended Kalman filter (EKF) for data fusion. Only with on-board equipment, his system is capable of autonomous flight and needs a 1.6 GHz Atom for

data processing. Weiss [60] is using the same system as an AscTec Pelican quadrotor [1]. He has developed a SLAM algorithm, which includes only the on-board camera for positioning and flight autonomy. Engel et al. ([27], [26]) introduced an EKF-based algorithm that exploits the identification of features. His approach empowers the parrot quadrocopter [8] to fly through 3D figures autonomously. Though, it also requires an external laptop for processing image data. Celik [23] used a monocular camera and US sensors to present a SLAM-based system. Gaurav et al. [38] did experiments with optical flow for obstacle avoidance, however, a stereo vision camera was also used to avoid collisions from straight onward zero-flow regions. Andert et al. [16] used evidential grid-based filtering with stereo image processing, to create a map based on stereo imagery that reactively avoided obstacles in simulation. Viquerat et al. [59] offered a reactive method of avoiding obstacles based on Doppler radar. Grzonka et al. [32] presented a multirotor that has the capability of simultaneous localization and mapping (SLAM).

Becker and Bouabdallah [18] and Bouabdallah et al. ([20], [21]) used four Ultrasonic sensors to detect obstacles and a camera system for positioning based on overground optical flow calculations. By controlling its position, the system is able to avoid collisions; however, it can not cover 360 ° or control the distance and it is also not applicable in avoiding obstacles in a canal-like environment. In comparison, when moving in the opposite direction, Roberts [50] uses four IR sensors and avoids collisions.

In contrast to these solutions, our system uses a stereo camera to capture the 3D-environment, generate a point cloud from the disparity images, and determine occupied and unoccupied space through Octomap mapping. The system can successfully detect an obstacle in the range of 25 meters and has the capability of both obstacle avoidance and mapping simultaneously, without the constraints of a heavy processor for computation.

1.3.2 Related Work on Path Planning

Unmanned aerial vehicle (UAV) 3D path planning aims to find an optimal and collision-free path in a cluttered 3D environment, taking into account geometric, physical and temporal constraints. It is the crucial element of the whole system when defining a mission. In general, path planning attempts to produce a global path to the target in real-time, avoid collisions and minimize a given cost function under kinodynamic constraints [65]. There is great potential for path planning in 3D environments, but the difficulties increase exponentially with dynamic and kinematic constraints becoming much more complex as compared to 2D path planning [65].

A few approaches are being developed over the past decades to address these problems. Algorithms applied in 3D environments include Visibility Graph [54] developed from computer science; random search algorithms such as Rapidly Exploring Random Tree [64] and Probabilistic Roadmap [62]; optimal search algorithms such as Dijkstra's [17], A* [25], and D* [22]; bio-inspired planning algorithms and etc.

The problem of motion-planning is often overcome either by first discretizing the continuous state space with a graph-based search grid or by sampling stochastic incremental searches. Graph-based searches, such as A* [36], are usually optimal resolution and full resolution. They are guaranteed to find the optimum solution, if a solution exists, and otherwise return failure (up to the discretization resolution). Such graph-based algorithms do not scale well with problem size (e.g., problem scope state dimension).

Stochastic searches, such as Rapidly Exploring Random Trees RRTs [43], Probabilistic Roadmap PRMs [41], and Expansive-Spaces Tree ESTs [39], use sampling-based approaches to ignore the need for state-space flexibility. It helps them to easily scale with the size of the problem and to consider kinodynamic constraints explicitly, but the result is a less rigorous guarantee of completeness. Probabilistically RRT's are complete, offering the likelihood of finding an effective solution if one is in existence, as iterations reach infinity, approach unity [31].

To date, these sampling-based algorithms have not made any predictions about

the solution's optimality. Simmons [58] found that using a heuristic sampling to bias improved RRT solutions, but did not measure the results formally. Ferguson [29] understood that the length of a solution limits potential improvements from above and showed an iterative anytime RRT approach to solve a variety of progressively smaller planning problems. Karaman [40] later showed that RRTs return to a sub-optimal path with a single possibility showing that any RRT-based path can almost definitely be suboptimal and present a new class of optimal planners. Both optimal forms have been identified separately from RRTs and PRMs, RRT* and PRM*. Such algorithms are shown to be asymptotically optimal, with the chance to find the optimum resolution reaching unity as infinity approaches the variety of iterations.

RRTs are not asymptotically optimal because future expansion is biased by the existing state graph. By introducing incremental rewiring of the graph, RRT* overcomes this [31]. Not only are new states added to a tree, but they are also considered to substitute parents for existing nearby tree states. This results in an algorithm with uniform global sampling that finds the optimal solution to the planning problem asymptotically by finding the optimal paths from the initial state to each state in the problem domain asymptotically. This becomes costly in high dimensions and is also inconsistent with their single-query nature.

A group of researcher at CMU [31] looked at the asymptotic behavior of RRT* and presented another version of the RRT-based algorithm, called Informed RRT*. Informed RRT* functions as RRT* until a first solution is found, after which it can only test from the sub-state set specified by an admissible heuristic to boost the solution. It is an improved version of RRT* that shows a clear improvement. When the configuration becomes more complex, it demonstrates huge improvements. The algorithm is less reliant on the dimension and domain of the planning problem as well as the ability to find improved topologically distinct paths faster as a result of its focused search. It is also able to find solutions with comparable computation within tighter tolerances of the optimum than RRT*, and in the absence of obstacles, the optimum solution can be found within system zero in the end time. It could also be used to further reduce the search space in conjunction with other algorithms, such as

path smoothing.

In this thesis, we have implemented Informed RRT* because it is the most appropriate planner according to our requirements. Mostly there are no obstacles in the path of the canal so we need a planner to give us a straight path towards the next goal point and Informed RRT* successfully does this. The algorithm returns a smooth path from start to goal point avoiding all obstacles, in almost every situation if one exists, as discussed later in Chapter 5.

1.3.3 Related Work on Mapping

In order to observe the surrounding environment, different types of map generation methods can be used. Using a grid of cubic volumes of equal size (voxels) to discrete the modeled area is a popular approach to 3D modeling environments. Using such a model, Roth-Tabak and Jain [51] and Moravec [44] introduced early works. A large memory requirement is a major drawback of rigid grids. The grid map needs to be configured in such a way that it is at least as large as the mapped area bounding box, regardless of the actual map cell distribution in the region. Memory consumption becomes prohibitive in large outdoor scenarios or when there is a need for fine resolutions.

Using point clouds can avoid discretization of the environment. The endpoints returned by vision sensors such as laser range finders or stereo cameras are used in these maps to model the space occupied in the region. Several 3D SLAM models such as [24], [45] used point clouds. The disadvantages of this approach are that there is no modeling of free space and unknown areas and that sensor noise or dynamic objects can not be dealt with directly. Point clouds are therefore only suitable for sensors with high precision. In addition, this representation's memory consumption increases with the number of measurements. This is troublesome because the upper bound is not there [61].

If it is possible to make some assumptions about the mapped area, 2.5D maps will be enough to construct the environment. A 2D grid is usually used to store the height measured for each cell. It results in an elevation map in its most basic form where

precisely one value per cell is retained by the map [37]. However, elevation maps are limited to one layer and are not capable of modeling bridges, underpasses, tunnels or structures on multiple levels. This strict assumption can be relaxed by allowing multiple surfaces per cell [57] or by using cell classes that match different structure types [35].

In several previous methods, tree-based representations such as octrees were used. By delaying the initialization of map volumes until measurements need to be integrated they avoid one of the main drawbacks in grid systems. The mapped area does not need to be defined in advance in this manner.

Fairfield [28] have suggested an octree-based 3D map representation. Their map structure called Deferred Reference Counting Octree is designed to enable efficient map updates and copying, especially in the SLAM particle filter context. However, his method does not address multiresolution queries. Yguel [66] provided a 3D map based on the data structure of the Haar wavelet. It's also a multi-resolution and probabilistic representation. However, methods of 3D modeling were not analyzed in-depth by the authors. Unknown regions are not modeled in their analysis, and only one virtual 3D dataset is used. It is difficult to evaluate whether this map structure is as memory-efficient as octrees.

Finally, to the best of our knowledge, no implementation of a 3D mapping system that resolves all the previous issues is available except the method described in [61] and we choose this method to implement on our system.

Chapter 2

Sensors

Unlike cars that can carry heavy payloads and sensors, aerial vehicle have limitations on the size and weight of the sensors with which they can fly. So, the first step was to select the sensors for obstacles avoidance and mapping. There are many models and types of obstacle avoidance sensors. For the purpose of this thesis, following sensors were selected:

- ZED Stereo camera [13]
- Hokuyo UTM-30LX LiDAR [5]

Although in this thesis, we did not exclusively use hardware, however, the sensors in simulation have exactly the same properties and behavior as the sensors mentioned in this chapter.

Hokuyo LiDAR and ZED stereo camera are compact and lightweight and is ideal sensors for an aerial vehicle. The stereo camera provides the depth information of the 3D-world through which we can map and percept the actual environment and avoid obstacles. Since Stereo camera has lightening and other certain restrictions, LiDAR, on the other hand, solves this problem and adds a certain probability in the detection of stereo camera. The area around the geometric description of the vehicle up to a certain range is covered by both Stereo camera and LiDAR, which greatly reduces the probability of collision.

This work assumes that both lenses of stereo camera are perfectly aligned with zero radial and tangential distortion.

2.1 Stereo Camera

With two lenses about the same distance apart, the stereo camera takes two images at the same time. This actually simulates the way humans see and therefore creates the 3D effect.

2.1.1 Stereo Camera Model

The stereo camera takes two images by the left and right camera at the same time to achieve depth information. The cameras are placed on the same plane, and their images overlap. The distance to any object can be determined by calculating the difference between the location or coordinates of an object as it appears in the two images. Knowing the focal length of the cameras and the baseline between the two lenses, this calculation can be achieved by using similar triangles. The closer an object is to the camera's location, the larger the deviation in its location in the two images. If an object point lies in both left and right images, two three-dimensional

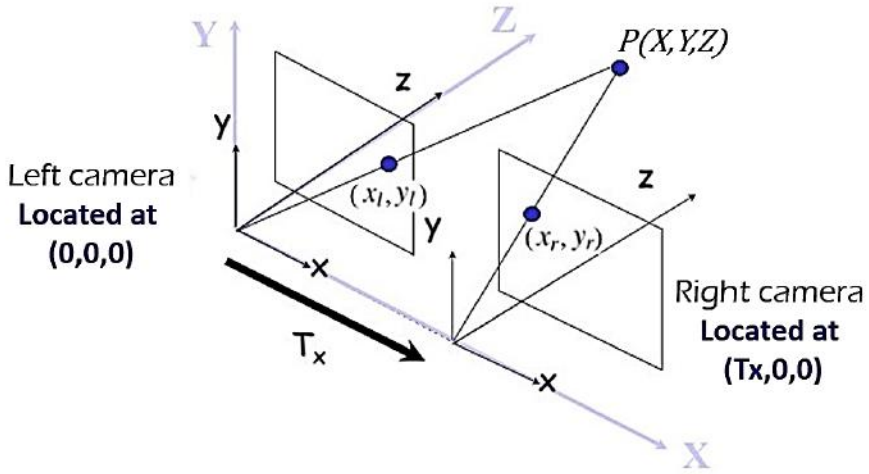


Figure 2-1: Stereo vision camera geometry [15]

rays can be projected back, joining the center of the camera, present in the three-dimensional world frame and the respective identical pixel points in the left and right image. The actual three-dimensional point in front of the camera is determined via the intersection of both of the rays at a common pixel point P. To obtain the three-dimensional position of an object, knowing camera parameters (i-e process for camera calibration) and the positions of matching pixels points (i-e matching process of the stereo camera) in both of the left and the right image.

2.1.2 Stereo Disparity Computation

The translation along the x-axis between the corresponding pixels point of the left and right image of a stereo camera is termed as disparity. If this process is performed pixel-wise for the entire image, then it is called the disparity image or disparity map. Inspired by the human binocular vision system, computer vision algorithms are used on a stereo vision camera to find depth. It relies on the two calibrated parallel images of stereo pair and calculates depth by estimating disparities between the two images.

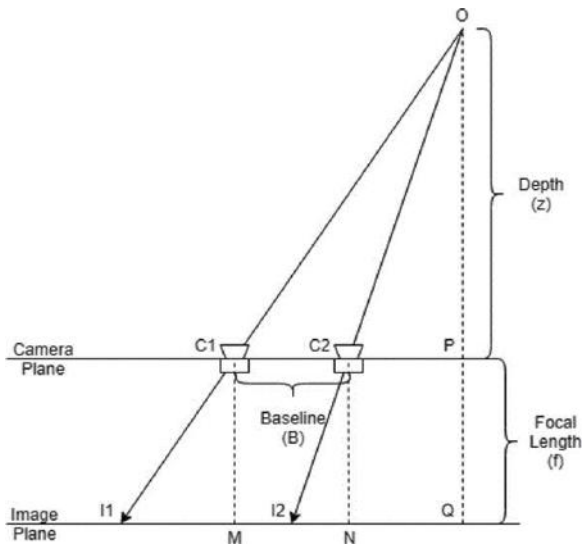


Figure 2-2: Stereo camera disparity image computation [46]

From $Fig.2.2$, we can see that

$$\triangle PQC_L = \triangle C_L M I_L, \tag{2.1}$$

and

$$\triangle PQC_R = \triangle C_R N I_R. \tag{2.2}$$

From Eq.(2.1), we know that

$$\frac{Z}{f} = \frac{C_L Q}{I_L M}, \tag{2.3}$$

and from Eq.(2.2)

$$\frac{Z}{f} = \frac{C_R Q}{I_R M}, \tag{2.4}$$

From Fig.2.2, it is clear that

$$B = C_L Q - C_R Q. \tag{2.5}$$

From Eq.(2.3) and Eq.(2.4), rewriting Eq.(2.5)

$$B = \frac{Z}{f}(I_L M - I_R M). \tag{2.6}$$

From the definition of disparity

$$d = (I_L M - I_R M). \tag{2.7}$$

Hence, Eq. (2.6) becomes,

$$B = \frac{Z}{f} d. \tag{2.8}$$

$$Z = \frac{fB}{d}. \tag{2.9}$$

Equation (2.8) shows, if we know the focal length, baseline between the two cameras of

the stereo pair, and disparity values, the depth information can easily be found. The above discussion is applicable for pixel-wise depth computation. Different algorithms are developed to find the disparity of a block contains many pixels. One such method is explained below.

2.1.3 Stereo Block Matching

A customary block-matching stereo set-up produces depth estimates by finding pixel-block matches between two images. Given a pixel-block within the left image, for instance, the system can search through the epipolar line to seek out the most effective match. The position of the match relative to its coordinate on the left picture, or the inequality, permits the user to work out the 3D position of the object in this pixel-block. One will consider a customary block-matching stereo vision system as an exploration through depth. As we tend to search on the epipolar line for a component cluster that matches our candidate block, we tend to area unit exploring the area of distance aloof from the cameras. For instance, given a pixel-block within the left image, we would begin exploring through the dextral image with an outsized inequality, correlated with an object on the point of the cameras. As we tend to decrease inequality (changing wherever within the right image are searching), we tend to examine pixel-blocks that correspond to things more and more away, till reaching zero inequality, wherever the stereo base distance is unimportant in contrast to space away and that we can't confirm the obstacle's location.

The concept behind block matching is to distribute the target image into blocks of mounted size and search the correlated block that is the best match within the right image. The problem with mounting size blocks is that their boundaries do not coincide with the object boundaries leading to higher prediction errors. By reducing the size of blocks, estimations errors are often reduced however; the ensuing matching is not good. On the opposite hand, by increasing the block size, hardiness is raised against noise in inequality calculation however, the magnitude of estimation error becomes high. As a result of these issues, some adaptational window block matching algorithms are suggested.

2.1.4 ZED Stereo Camera

The ZED [13] stereo camera is among state-of-the-art, developed by Stereolab. It requires NVIDIA graphics card with computation capability greater than 3.0 and CUDA (6.5 and above) to do millions of parallel computations. Unlike many other existing three-dimensional sensors, which provide a maximum range of up to Four meters, this camera has the capability of capturing objects up to 20 meters apart at a resolution of 1080 pixels. It is a lightweight and low-price perception sensor, that is mostly used in autonomous drones for 3D obstacle avoidance.

Technical Specifications

ZED is a passive Stereo camera that computes inputs in the manner of human vision. It is a compliant Universal Video Class UVC, presuming the ZED-SDK is not required to capture the camera's left and right video streams. However, CUDA is required in the GPU of the host computer to compute depth maps in actual-time and to use ZED-SDK to build applications. The technical requirements are described below.

Video

The two images of the stereo camera are synchronized, compressed, and sent as a single side-by-side video structure. The resolution of the ZED stereo camera can be easily changed using API and ZED explorer. The ZED video available outputs at different FPS and resolution is shown in *Table*.2.1.

Mode of the video	FPS	Output
2.2K	15	4416x1242
1080p	30	3840x1080
720p	60	2560x720
WVGA	100	1344x376

Table 2.1: Video Output specifications provided by ZED stereo camera [13]

Depth

The ZED camera computes the depth the way our binocular vision works. Horizontally separated by approx. 65mm on average [3], the left and the right human eye

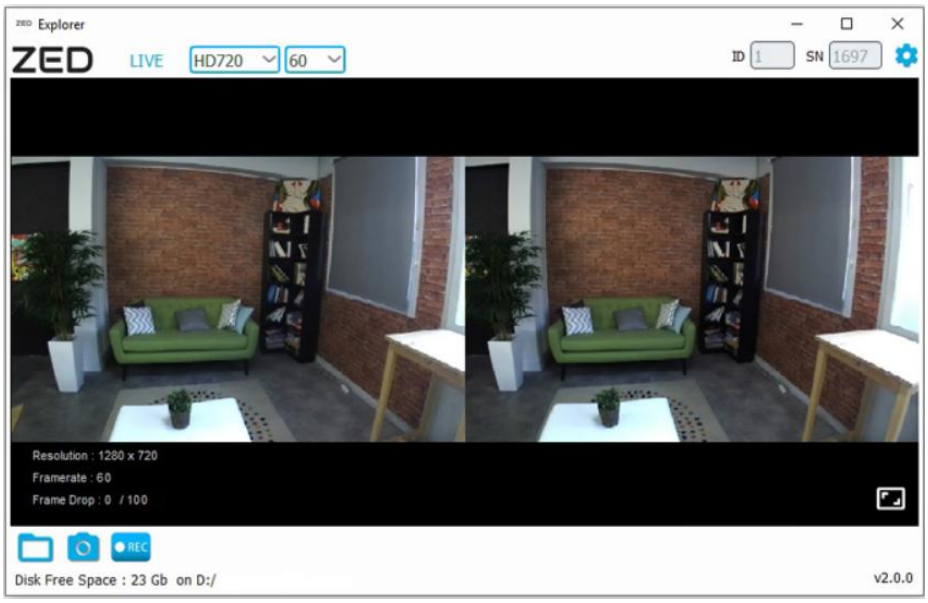


Figure 2-3: Image output of ZED stereo camera [13]

have a slightly different perception of the environment around them. By comparing these two different perceptions, the human brain can infer the depth as well as the 3D motion. Similarly, the ZED has two eyes in the form of lenses, separated by 20cm, captures HD 3D video of the environment and then estimates the depth and motion by comparing pixels difference in the left and right images.

For each pixel, the ZED stores a depth value (Z). The depth is measured in meters and calculated from the left lens of the camera to the object in the environment. The depth map is encoded in 32 bits, due to which it cannot be displayed directly.(Fig.2.4)

Size and Weight

The camera dimension is 175x30x33 (mm) and it weighs 159g.

Lens

The ZED stereo camera has a wide-angle, dual-lens with reduced distortions and f/2.0 aperture. The Field of View (FOV) of the camera is 110-degree max.

Coordinate System

To specify positions and orientations, the ZED camera uses a 3D Cartesian coordinate system (X, Y, Z). The coordinate system can be either left-handed or right-handed. By default, the ZED uses an image coordinate system that is right-handed with the

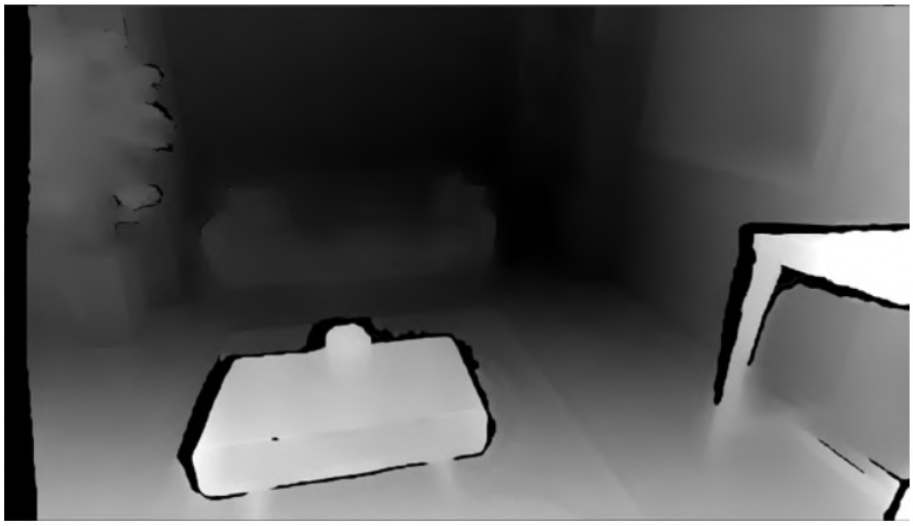


Figure 2-4: ZED Stereo Camera depth visualization [13]

X-axis pointing right, the positive Y-axis pointing down and the Z-axis pointing outward direction from the camera

Sensors

With large 2-micron pixels, the ZED sensor resolution is 4 Mega Pixels. It also has electronic synchronized Rolling Shutter.

Connectivity

For the best performance, ZED supports USB 3.0. Also, it is powered via USB (8V/380 mA).

2.2 Light Detection and Ranging (LiDAR)

Light Detection and Ranging (LiDAR) measures the distance to an object by calculating the time of flight of a pulse needed to travel to an object and reflecting back to the sensor. They usually work in the near-infrared spectrum and some can work outdoors at ranges varying from a few meters to several hundred meters.

A LiDAR can measure a single point (i-e., 1D LiDAR) as shown in $Fig.2.6(a)$, a plane of points (i-e., 2D LiDAR) shown in $Fig.2.6(b)$, or measure multiple planes to scan the complete environment area around it (i-e., 3D LiDAR) shown in $Fig.2.6(c)$.

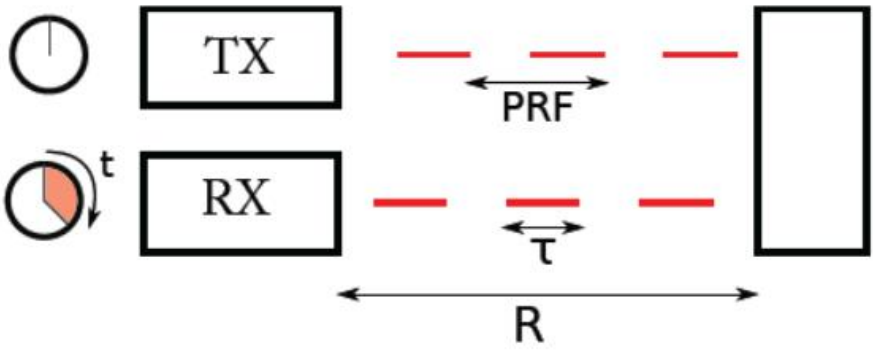


Figure 2-5: Hardware set-up for a pulsed LIDAR flight [15]

The one-dimensional can be useful in detecting a frontal plane but can not be used for navigation purposes, while a two-dimensional LiDAR, because it covers the entire plane can be used for the navigation of ground robots with certain limitations as it can not infer any information above and below the LiDAR scan. The three-dimensional LiDAR, has the capability of analyzing the environment both vertically and horizontally at the same time. Since aerial vehicle moves in the 3D world so the ideal sensor for navigation in a 3D environment would be 3D LiDAR but because of its heavyweight and high cost one can not use it for aerial vehicle navigation. The LiDAR works by continuously rotating a laser beam around the scan area and then reading the returned signal reflected from the obstacles. The output is generated in the form of a polar coordinate that contains range values and angle at which those ranges returned. Range values are the distance between the sensor and the object or obstacle at which the laser beam is reflected.

(a) Hokuyo Lidar (b) RPLIDAR (c) Lidar Lite

Figure 2-6: 1D, 2D and 3D LIDARs [67]

2.2.1 Hokuyo UTM-30LX LiDAR

Hokuyo UTM-30LX [5] is used in this research. It has a 240^o FOV and about 0.36^o angular resolution. The range of Hokuyo UTM-30LX ranges from zero to 30 meters. With increasing range, the width of the cone grows. It operates at 10 HZ and collects data in sweeps, each with N laser measurements. This LiDAR can not provide any information about the intensity of the pulses. A range measurement r_i at angular measurement ϕ_i in i_{th} laser measurement can be found as

$$L_i = \begin{bmatrix} r_i \\ \phi_i \end{bmatrix}, i = 1, ..., N.$$

For mapping applications, the whole FOV is not required as measurements provide information when hitting the object. Hence, the FOV can be taken limited to [min, max]. The range and bearing data from the LiDAR can be transformed into Cartesian coordinate by the following relation,

$$\begin{bmatrix} X_i \\ Y_i \end{bmatrix} = \begin{bmatrix} r_i \cos(\phi_i) \\ r_i \sin(\phi_i) \end{bmatrix}, i = 1, ..., N.$$

2.3 Stereo vs LiDAR Comparison

Stereo vision camera and LiDAR are both used in autonomous vehicles, obstacle detection and collision avoidance. The main differences between both the sensors are summed up in *Table*.4. From the table, we can see that, the pros of one sensor are actually the cons of the other and vice versa. Like the stereo camera cannot directly observe obstacles while the LiDAR does. Similarly, LiDAR operates at a low frequency while the stereo camera can operates at high frequency. Hence, to have an enrich and robust obstacle avoidance system, both of the sensors should be incorporated.

	Stereo-Vision Camera	**LiDAR**
Pros:	High Resolution	Directly Observes Obstacles
	High Frequency	Large Horizontal FOV
	Inexpensive	Easier to Simulate
	Covers 3D environment	Do not Require external Lighting
Cons:	Cannot Directly Observe Obstacles	Low Resolution
	Generally Small Horizontal FOV	Low Frequency
	Difficult to Simulate	Expensive
	Requires external Lighting	Obstacles above or below the scanning plane are invisible

Table 2.2: Comparison between Stereo camera and LiDAR

Chapter 3

Simulation Environment

This chapter [49] describes the simulation environment and tools for performing experiments with the algorithm described in Chapter 4. To implement the mapping and obstacle avoidance, a robotic platform was needed. A setup to simulate a 3D environment and a drone with a stereo camera, LiDAR and other sensors had to be built. A system that can be easily configured to operate with a real sensor and a real drone rather than a simulated one, if needed.

It was clear from the outset that ROS (Robot Operating System) [10] would play a role in this simulation environment, mainly because it provides a decentralized communication middleware and provides an infrastructure that allows components to be easily connected and facilitates the modular design. It also offers methods and groups of useful libraries (libraries for frame transformations, point cloud processing, visualization, and data monitoring to name a few). Besides that, ROS is well established in the robotic community and we have had previous experience with it.

A simulator is the core part of the simulation process. The main requirement was the simulator's ability to simulate a 3D environment. Two simulator candidates Unreal Engine [11] and V-REP [12], are considered and compared. The next section briefly characterizes both. The intended objectives of this thesis are achieved with both of the simulation platforms. The 3D environment in the Unreal engine is more realistic than in V-REP, hence the Unreal engine is chosen.

3.1 V-REP Simulation Platform

V-REP is a simulator of robotic systems designed around a versatile architecture that is adaptive. V-REP has various relatively independent functions, features or more elaborate APIs, which can be activated or deactivated as desired. A distributed control architecture is obtained by enabling an integrated development environment: each object/model can be managed individually via an embedded script, a plugin, a ROS node, a remote API server, or a custom solution. In C / C++, Python, Java, Lua, Matlab, Octave or Urbi, controllers can be written.

V-REP encapsulates several calculation modules that can work directly on one or more items in the scene. Such modules for calculation include,

- the module for collision detection,
- the module for minimum distance calculation,
- the module for forward and inverse kinematics calculation,
- the module for physics or dynamics,
- the module for path or motion planning.

V-REP is easy to use. It has a rich graphical user interface. The GUI and the built-in scripting feature have two ways to modify the simulation's different aspects. V-REP includes many features and computing modules, most of which are programmatically accessible. Nonetheless, the API is inspired by V-REP's GUI design and the user needs to manipulate objects in the GUI in a very similar manner.

3.1.1 Simulation Environment Build in V-REP

Initially, we start working on building our canal simulation environment in the V-REP physics engine. As shown in *Fig.*(3.2), (3.3), and (3.4), the total length of the canal was about 1 km, with a single type of tree modal, imported in V-REP. With this length of the canal the processor and memory consumption reaches its peak because of the ray casting process and fewer graphics support.

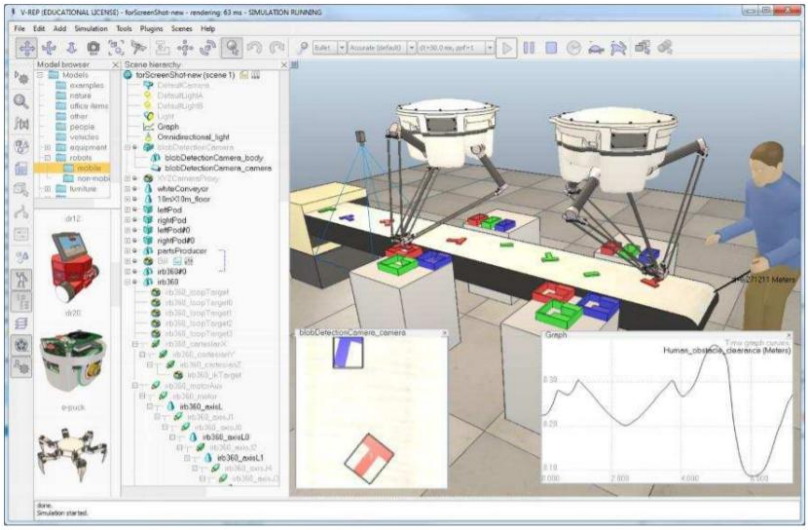


Figure 3-1: V-REP scene with multiple robots is shown [42].

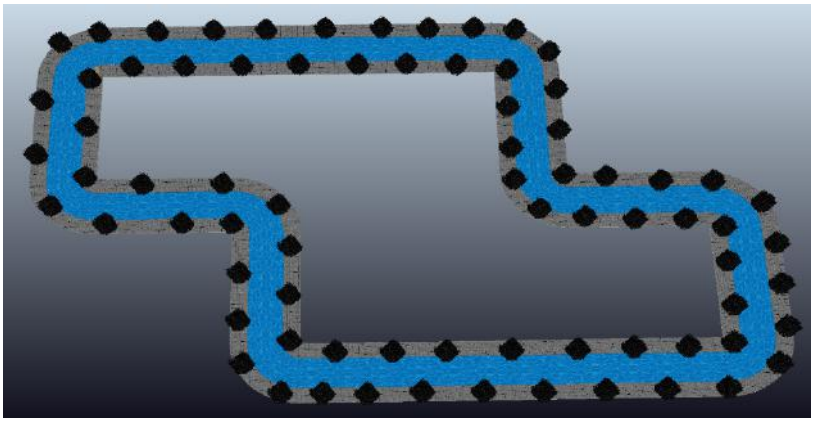


Figure 3-2: Top view of the V-REP simulated canal complete structure

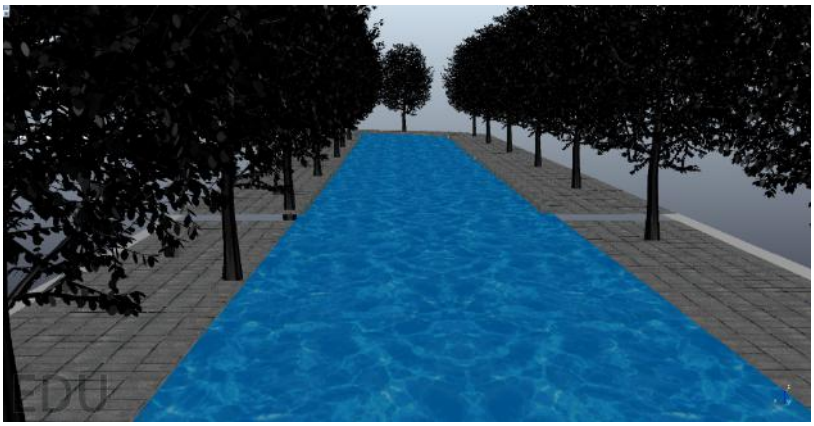


Figure 3-3: Close view of the V-REP canal structure

Figure 3-4: Tree modal of the V-REP simulated canal

3.2 Unreal Engine

Unreal engine is an open-source gaming engine, developed by Epic Games [11] and is released in March 2014. It supports Windows, MacOS, and Linux. Because of its huge community, it has a high availability of free plugin and extensions (e-g, ROS). It has full access to the C++ source code and can be modified to fit one's own specific needs. Having access to the Unreal engine marketplace, it has a huge variety of available assets where many are free and almost everything can be found which saves time and money. Assets from most of the modeling software can be easily imported. It also uses the blueprint system, which is a node-based interface in which functions and variables can be connected via drag and drop. The blueprint system allows to create simple or complex behaviors without having to write C++ code, many functions are already implemented. This can speed up implementation and allows for quickly executed experiments. Unreal engine huge environments can be created with features such as fog, rain, water, vegetation, etc. while maintaining good performance the realism of these can be tuned in regards to need, performance, and available time and budget. Unreal engine is a very complex software, but easy to get started with and has a huge depth. It also has a sequencer, which allows creating professional or cinematics photo-real rendering in real-time. However, the only constraint it offers is a powerful PC to run, especially the graphics card.

3.2.1 Simulation Environment Build in Unreal engine

The total simulation environment built in this thesis consists of a total length of $2,378$ meters that contains eight different types of tree structures, brushes, branches, and bridges ($Fig.3.6-3.9$ depicts the same). The 3D canal mesh is shown in $Fig.3.5$ that is build in Autodesk [2] and then imported to the Unreal engine. Using the spline tool in the Unreal engine, the canal mesh is expended to a closed-loop. With its 3.5m height, 6 to 9m width, and the cemented textures make the 3D canal looks like an actual canal structure in the simulation environment.

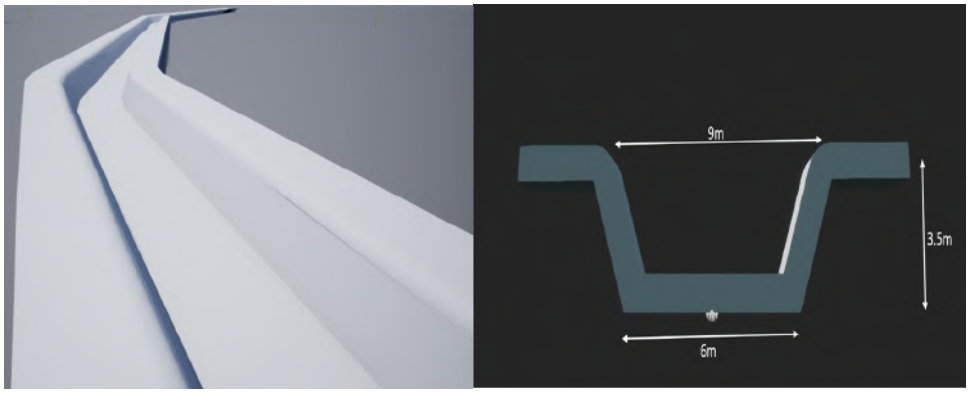


Figure 3-5: Canal 3D mesh, build in Auto Desk and then imported into Unreal engine.

Figure 3-6: Top view of the complete canal in Unreal engine.

Figure 3-7: Close look of the canal in Unreal engine.

Figure 3-8: Close look of the canal bending in Unreal engine.

Figure 3-9: Close look of the bridge over the canal in Unreal engine.

Obstacles in the Simulated Canal environment

Since the objective of the thesis is obstacle avoidance and navigation of an aerial vehicle in the canal environment. In the canal, obstacles can be brushes, branches, trunks, and bridges. Hence, for our path-planning problem, we build a bridge and tilted tree at the same spot (*Fig.*3.10), so that it can serve as a tough obstacle for our drone to navigate through.

Figure 3-10: Bridge and tilted tree at the same spot, serves as an hard obstacle.

3.3 Microsoft Airsim Plugin

AirSim [55] is a simulator based on Unreal Engine for aerial vehicles, cars and more. It is open-source, cross-platform, and supports hardware-in-loop simulations with famous flight controllers such as PX4. It is built as an unreal plugin that can be dropped into any unreal environment. Airsim was developed as a platform for AI research to experiment with autonomous vehicles with deep learning, computer vision, and improving learning algorithms. For this reason, AirSim also exposes APIs for independent platform recovery of data and control vehicles.

3.3.1 Airsim Multirotor

Fig.3.11 shows an airsim AR drone modal that is equipped with many common robotics sensors. The vehicle model includes parameters such as mass, inertia, coefficients for linear and angular drag, coefficients of friction and restitution, which is used by the physics engine to compute rigid body dynamics and the real world drone exhibits these parameters. The dimension of the drone is 1x1, that is its length and width are 1m each. The drone is equipped with several perception sensors, such as a vertical and a horizontal scanning LiDARs, five cameras that are mounted around the drone; three at the front, one at back, and one camera at the bottom center. The drone also has localization sensors such as GPS, IMU, barometer, Gyrometer, magnetometer, and accelerometer.

Figure 3-11: Airsim multirotor, an AR drone, equipped with sensors.

By adding the following lines in the *setting.json* file, the vehicle type for the simulation is automatically set to multirotor.

`"SettingsVersion"`: 1.2,

“`SimMode`”: “`Multirotor`”,

Airsim Multirotor Flight Control

AirSim has an integrated flight controller called simple flight, which is used by default. To use or customize it, you do not have to do anything. AirSim also supports PX4 for advanced users as another flight controller. The vehicle control through simple flight controller can be achieved by inputting in the desired velocity, angle, or position information. One of these modes can be used to define each control axis. Internally simple flight uses a PID controller cascade to essentially produce actuator signals. This implies PID location drives the PID velocity, which drives the PID angle level, which finally drives the PID angle frequency.

Airsim flight control configuration

To use AirSim simple flight, we can define it in *settings.json* as shown below. This is by default enabled, so we do not necessarily have to do it.

```
‘‘Vehicles": {
   “SimpleFlight”: {
      “VehicleType”: “SimpleFlight”,
   }
}
```

Airsim Available Sensors for Multirotor

The following sensors are currently supported by AirSim: Camera, IMU, Magnetometer, GPS, Distance, Barometer, and LiDAR.

Configuration of Sensors

The sensors for Airsim multirotor can be enabled by adding the following lines into the `settings.json file`

```
‘‘DefaultSensors": {
   “Barometer”: {
      “SensorType”: 1,
```

```
        "Enabled" : true
    },
    "Gps": {
        "SensorType" : 1,
        "Enabled" : true
    },
    "Lidar1": {
        "SensorType" : 6,
        "Enabled" : true
        "NumberOfChannels" : 16
        "PointsPerSecond" : 10000
    },
    "Lidar2" : {
        "SensorType" : 6,
        "Enabled" : false
        "NumberOfChannels" : 4
        "PointsPerSecond" : 10000
    },
}
```


3.4 Comparison between Unreal engine and V-REP

A brief comparison between the two simulation platforms is described in the form of a table in *Table*.3.1. Both V-REP & Unreal engine supports sensors such as stereo camera, LiDAR, GPS, and IMU. V-REP can be used in less computing machine but Unreal engine requires powerful machine and graphics card to simulate an environment. V-REP is not meant for a large scale environments while Unreal engine does supports.

Unreal engine	**V-REP**
Developed by Microsoft	Developed by Coppelia Robotics
It has weather effects such as Rain, Wind, Pollen, Dust, Fog, and temperature	It has only Temperature sensing and fog effects
It has robotics sensors such as LiDAR, IMU, Barometer, GPS, Magnetometer, and US	Mostly common sensors available
Simulator for drones, cars and more, built on Unreal Engine	Simulator specifically developed for robotics applications
open-source, cross-platform, and supports hardware-in-loop	open-source, cross-platform, and does not support HIT
Can be integrated with ROS	ROS Integration available

Table 3.1: Comparison between Unreal engine and V-REP.

Chapter 4

Methodology

4.1 System Architecture

The overall system architecture is shown in *Fig*.4.1. The system is divided into three main blocks; Observation sensors, Mapping, and Decision and Control.

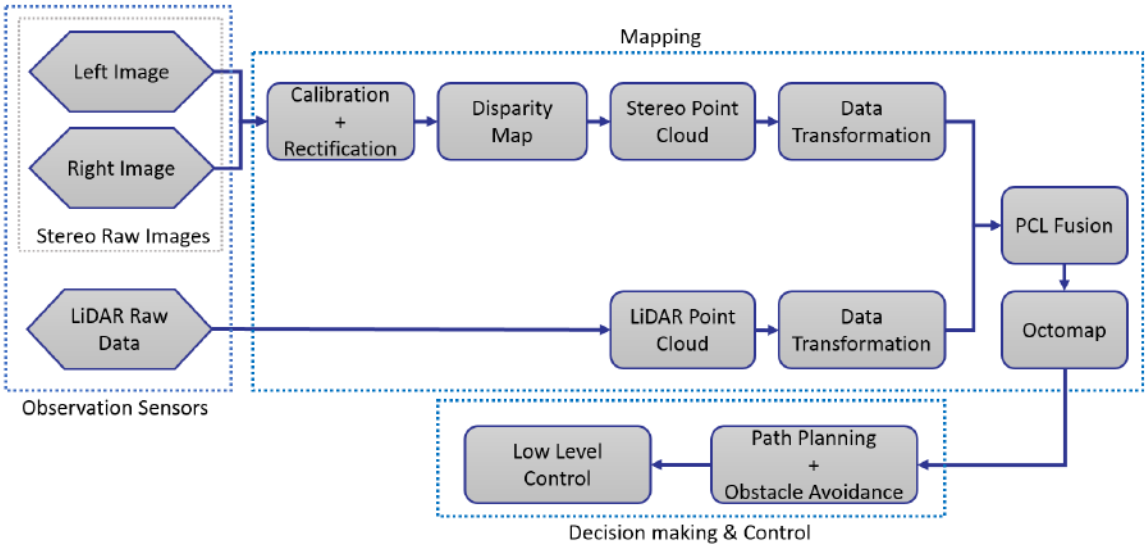


Figure 4-1: Overall system architecture of the obstacle avoidance & mapping system.

1. *Observation Sensors*: A 2D LiDAR and a stereo camera are used as observation sensors to percept the actual environment. The raw images and camera information of both cameras are transported to the second block for further processing.

2. *Mapping*: This block of the system provides a three-dimensional environment map. The images from both the camera sensors are first calibrated and rectified. Next, both rectified images are processed to construct a disparity image and point cloud data of the objects in the camera field of view in the world. At this stage, we have data in the stereo coordinate system, we need to collect the point cloud data and transform it to a common frame of reference, which in our case is the world coordinate system. Similarly, the LiDAR scan is collected, converts them into point clouds and then again transforms it into the same common frame of reference, the world coordinate system. Both point cloud data at this stage is at the same frame of reference, what we have to do now is to combine both point clouds, in order to incorporate both sensors detection, have a robust, and enrich obstacle detection representation. Now our data is ready to be used for mapping. We used Octomap mapping [61] for 3D mapping of the canal environment.

3. *Path planning & control*: The third block of the system is primarily responsible for path planning and low-level control after the map is being generated.

We will explain each of them one by one in detail in the succeeding sections.

4.1.1 Getting Left and Right Image from Stereo camera

Like human binocular vision, the stereo camera has two cameras with two lenses that can capture images simultaneously. In the same way, we use the front left and front right cameras of the Airsim multirotor as a stereo camera and the baseline between them is kept 20mm as shown in *Fig.* 4.2.

Getting images from airsim camera sensor to ROS includes two types of configurations:

The first step to getting an image from the airsim camera to the Unreal engine is done by enabling camera sensors, defining the type of image the camera has to generate, and setting their properties in `settings.json` file. This is done by simply adding the following lines in the `settings.json` file:

```
''Cameras'':
```

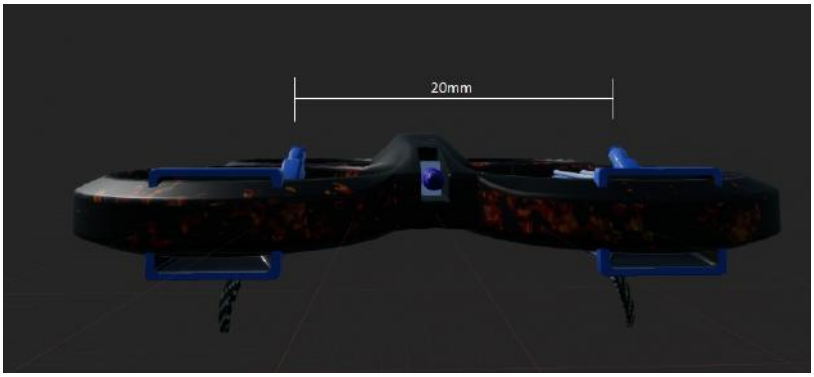


Figure 4-2: Stereo camera on the Airsim multirotor. Both cameras have same pose and focal length.

‘‘CameraName’’: “1”, “ImageType”: “0”, “PixelAsFloat”: “true”, “Compress”: “true”}

“CameraName”: “2”, “ImageType”: “0”, “PixelAsFloat”: “true”, “Compress”: “true”}

This will enable camera sensors and set the type of image the camera has to capture. For image properties, we have to add the following lines

‘‘CameraDefaults’’: {

‘‘CaptureSettings’’: “1”

{

“ImageType”: “0”

“Width”: “640”

“Height”: “480”

“FOVDegrees”: “120”

“AutoExposureSpeed”: “100”

“AutoExposureBias”: “0”

“AutoExposureMaxBrightness”: “0.64”

“AutoExposureMinBrightness”: “0.03”

“MotionBlurAmount”: “0”

“TargetGAmma”: “1.0”

“ProjectionMode”: “”

"OrthoWidth": "5.12"

}

After setting these configurations, we are able to receive left and right rectified images in the Unreal engine.

The next step is to publish the images to ROS using airsim APIs.

```
responses = client.simGetImages([airsim.ImageRequest(0, airsim.ImageType.Scene)])
```

Where the first argument "0" represents the png format of the image.

4.1.2 Airsim LiDAR data to ROS

Airsim multirotor has two built-in LiDARs. Like in the case of a stereo camera, to get LiDAR scans we have to set `settings.json` file and add the following lines.

```
''frontLidarSensor'': {
''SensorType'': 6,

                    ''Enabled'': false,
                    "NumberOfchannels": 1,
                    "RotationsPerSecond": 25,
                    "Range": 30,
                    "X": 0, "Y": 1, "Z": −1,
                    "Roll": 0, "Pitch": 0,
                    "VerticalFOVUpper": −1,
                    "VerticalFOVLower": 1,
                    "HorizontalFOVStart": −90,
                    "HorizontalFOVEnd": 90,
                    "DrawDebugPoints": false,
                    "DataFrame":
```

The data from the Unreal engine is published on a ROS topic using standard ROS publisher method.

4.1.3 Stereo Disparity and Point Cloud construction

For disparity image computation and point cloud generation, we use `Stereo_Image_Proc` [9], the node that creates a stereo image pipeline. `Stereo_Image_Proc` rectifies raw image pairs of the stereo camera. It may also conduct stereo processing to produce disparity images and point clouds. We need in the `.launch file` , remapping of the following topic names:

```
<launch>
<node pkg=‘‘stereo_image_proc’’ type=‘‘stereo_image_proc’’ name=‘‘’’
respawn=‘‘true’’>
<remap from=‘‘left/image_raw’’ to=‘‘/raw_stereo/left/image_raw’’ />
<remap from=‘‘left/camera_info’’ to=‘‘/raw_stereo/left/camera_info’’ />
<remap from=‘‘right/image_raw’’ to=‘‘/raw_stereo/right/image_raw’’ />
<remap from=‘‘right/camera_info’’ to=‘‘/raw_stereo/right/camera_info’’ />
</node>
</launch>
```

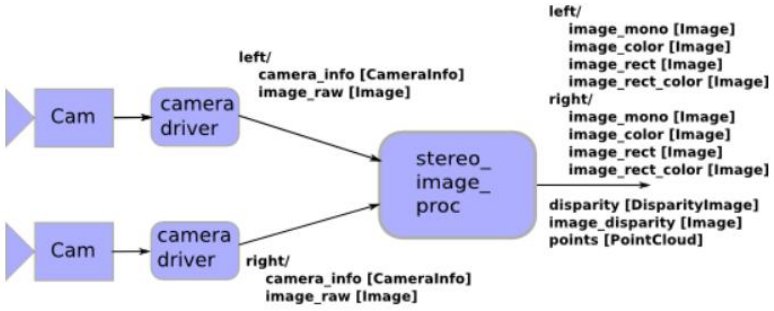


Figure 4-3: Basic block diagram that shows the working of `Stereo_Image_Proc` node[9].

For this setup, a simple basic block diagram is given in *Fig.* 4.3. It subscribes to left, right images of the stereo pair and left, right camera information topics.

After processing, the node publishes left, right rectified images, mono_color images, disparity image, and point cloud as output, as can be seen from *Fig.* 4.3. *Fig.* 4.4 shows a complete `rqt_graph` showing the full flow of the `Stereo_Image_Proc` setup in the ROS system.

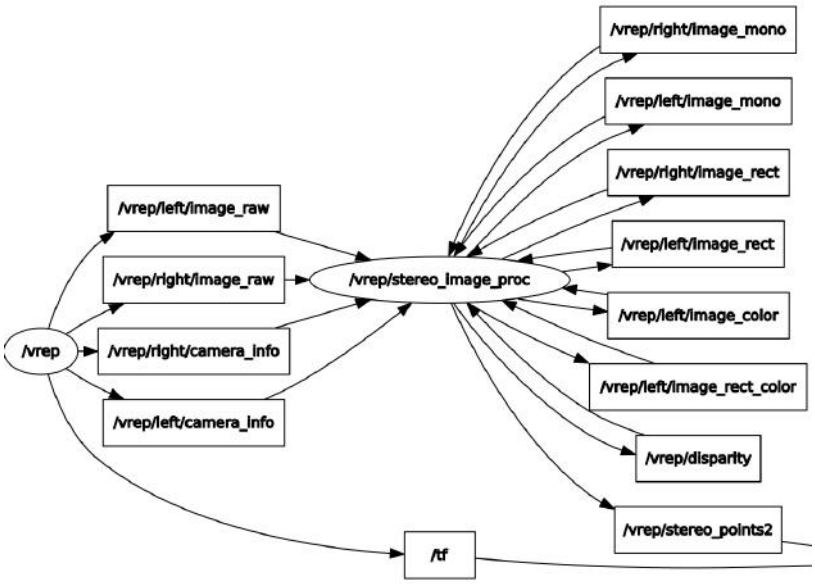


Figure 4-4: `rqt_graph` showing the flow of topics through `Stereo_Image_Proc` node.

The left imager optical frame (X Up, Y Down, Z out) produces point clouds.

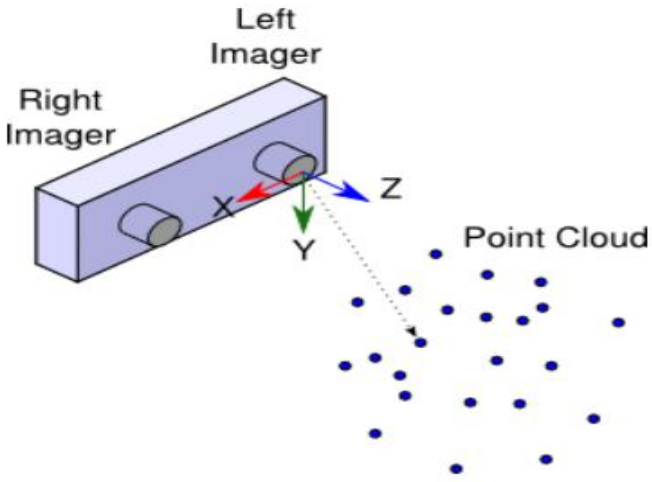
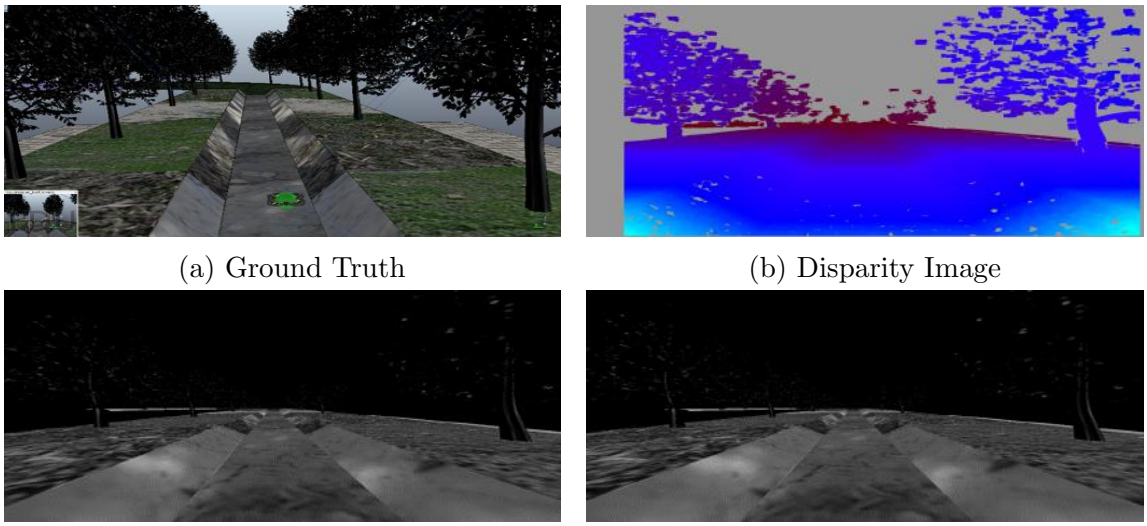


Figure 4-5: Point cloud construction through stereo camera [9].

Disparity Image Output Using V-REP Simulation

The disparity image output using V-REP physics engine is shown in *Fig.* 4.6.



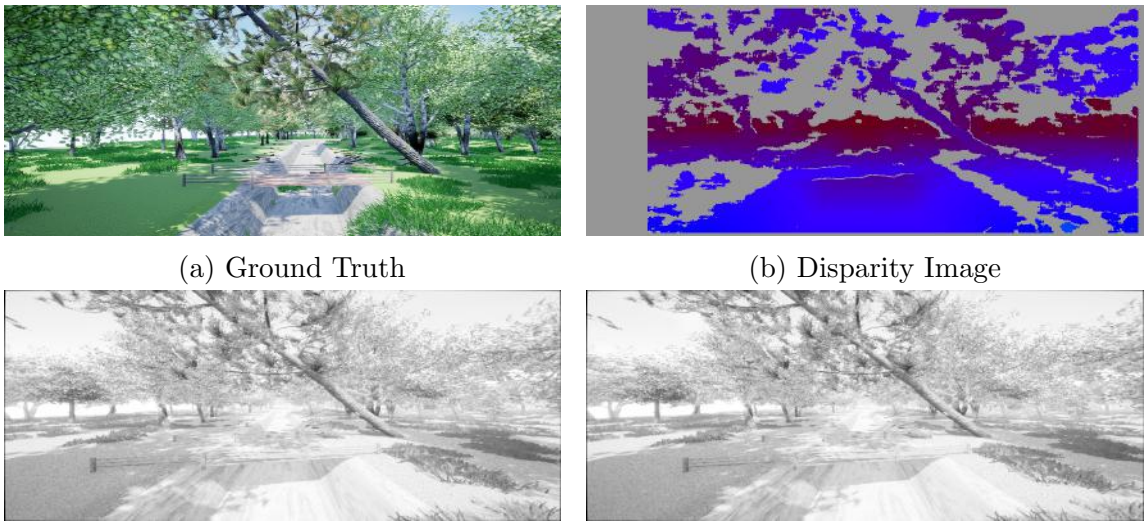
(a) Ground Truth (b) Disparity Image

(c) Left Image (d) Right Image

Figure 4-6: Disparity Image output using V-REP simulation engine.

Disparity Image Output Using Unreal Engine

The disparity image output using Unreal engine is shown in *Fig.* 4.7.



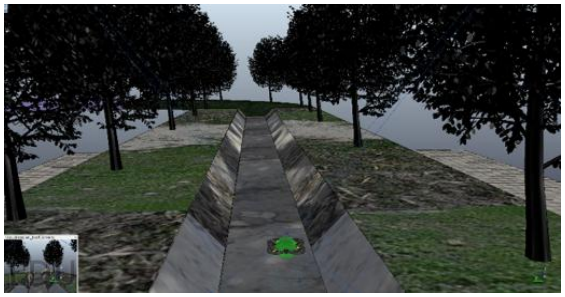
(a) Ground Truth (b) Disparity Image

(c) Left Image (d) Right Image

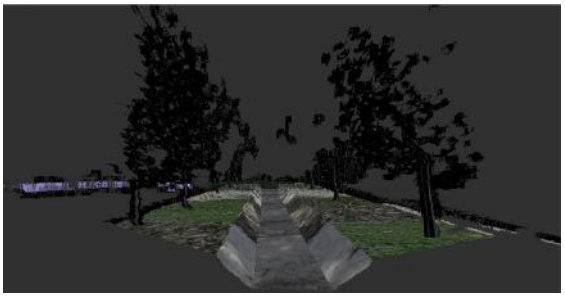
Figure 4-7: Disparity Image output using Unreal engine.

Point Cloud Output Using V-REP Simulation

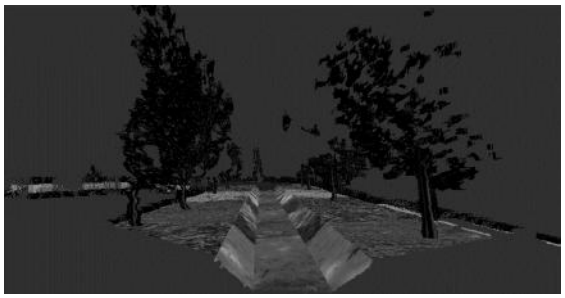
Stereo point clouds with different representations along with actual ground truth are shown in *Fig.* 4.8. The `Stereo_Image_Proc` provides stereo point clouds computed from disparity images.



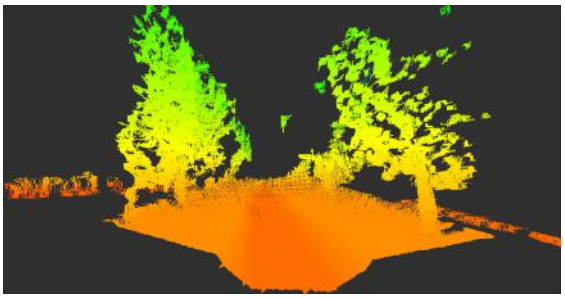
(a) Ground Truth



(b) PCL RGB Color Representation



(c) PCL Mono Color Representation



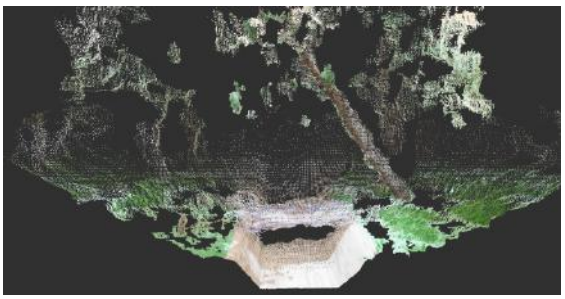
(d) PCL Intensity Representation

Figure 4-8: Point Clouds output using V-REP simulation engine.

Point Clouds Output Using Unreal Engine

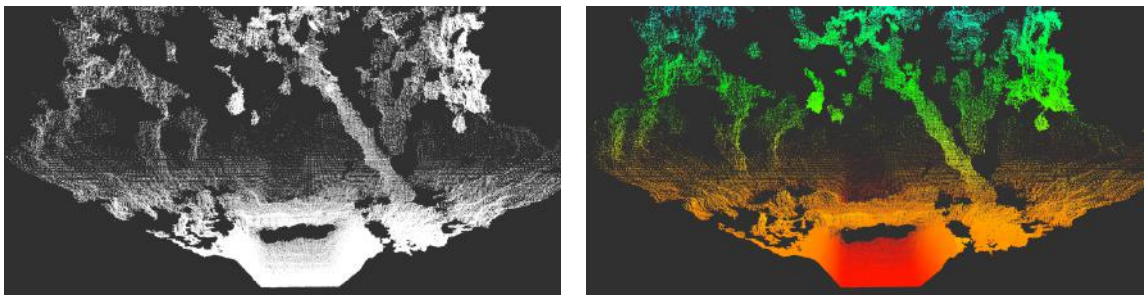
The point clouds output using Unreal engine is shown in *Fig.* 4.9 & 4.10.

(a) Ground Truth



(b) PCL RGB Color Representation

Figure 4-9: Point Clouds output using Unreal engine



(a) PCL Mono Color Representation (b) PCL Intensity Representation

Figure 4-10: Point Clouds output using Unreal engine.

Stereo Camera point cloud Transformation

The point cloud collected from the stereo camera after computing disparity image is in the stereo frame of reference. However, between our stereo camera and the vehicle or its base frame, we need to publish the frame transformation. Point cloud from left and right camera frame is transformed to the stereo frame which is the center of both the cameras. The stereo frame is then transformed into the multirotor frame which in our case is base frame, and it is the center of the multirotor as can be seen from *Fig*.4.11 & *Fig*.4.12. The best way to achieve this transformation is by using the `ROS tf package` .

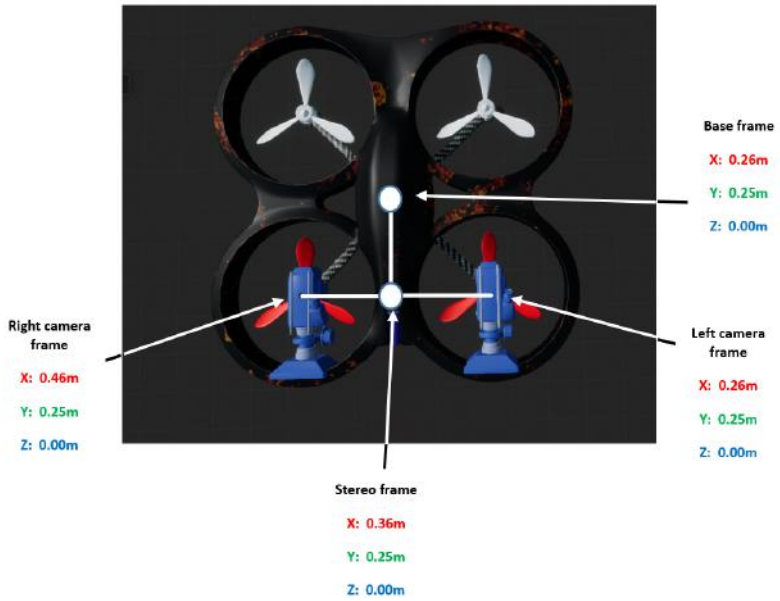


Figure 4-11: Stereo camera transformation w.r.t drone base.

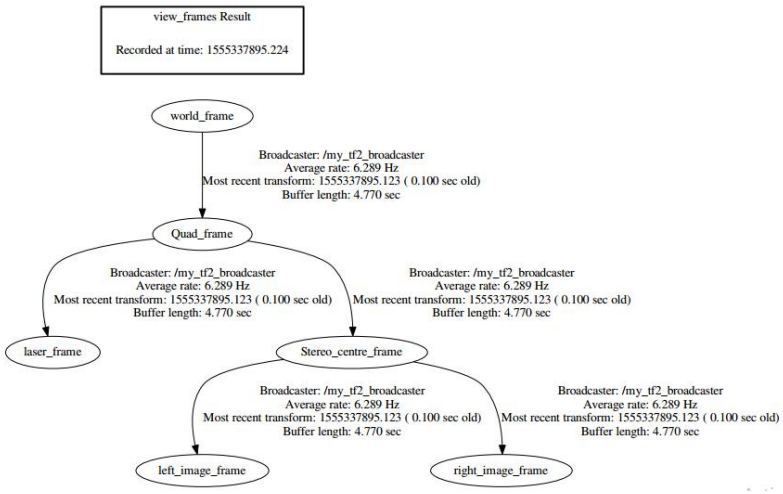


Figure 4-12: tf tree transformation between Stereo camera & LiDAR to drone base frame

LiDAR Point cloud Transformation

Since LiDAR is mounted on the top of the multirotor, the point cloud generated by LiDAR also needs to be transformed to the base frame. LiDAR in Airsim multirotor is mounted at the center of the vehicle with a shift along the z-axis. Again, using the same `ROS tf package` , we transformed data from LiDAR frame to base frame.

4.2 Sensor data fusion via Concatenation

Uptill now, we have data from both the sensors in the same frame of reference. Now we need to combine them to have a reliable collision detection system to incorporate both sensors detection. We concatenate the data by implementing the below method. The ground truth and its result are shown in *Fig.* 4.13 & 4.14

```
concatPcl=stereoPcl;
concatPcl+=LidarPcl;
```

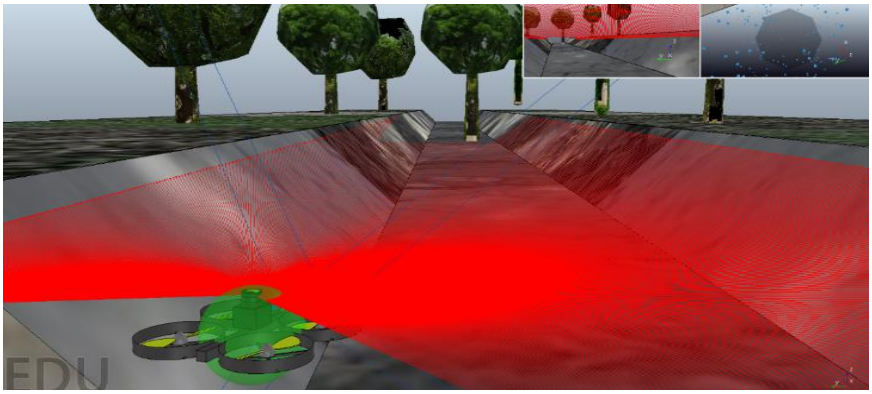


Figure 4-13: Actual Ground Truth.

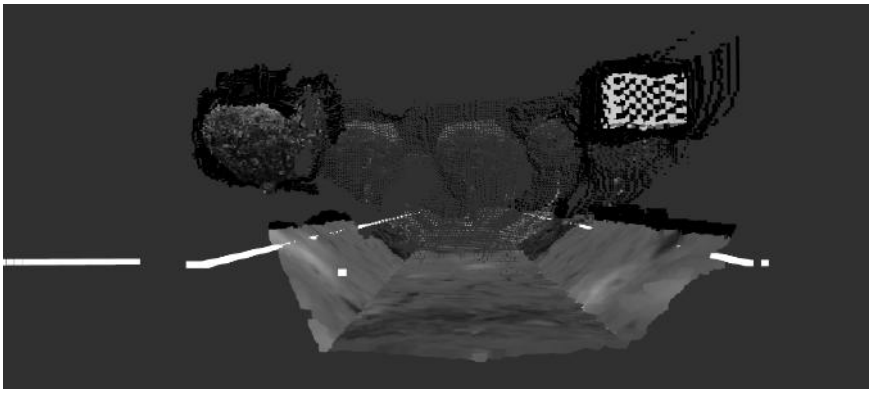


Figure 4-14: Visualization of the concatenated sensor data.

4.3 OctoMap mapping framework

Octomap mapping uses a tree-based representation to provide maximum flexibility in the mapped region and resolution. To ensure updatability and cope with sensor noise, it conducts a probabilistic occupancy calculation. In addition, compression methods ensure the compactness of the resulting models.

Octree

An octree is a hierarchical 3D spatial classification data structure. The space found in a cubic volume, usually called a voxel, is defined by each node in an octree. This volume is subdivided recursively into eight sub-volumes until, as shown in Fig.4.15, a given minimum voxel size is achieved. The minimum size of the voxel determines the octree's resolution. Since an octree is a hierarchical data structure, if the inner nodes are preserved correctly, the tree can be split at any point to obtain a coarser

subdivision.

Octrees can be used in its most basic form to model a Boolean property. This is

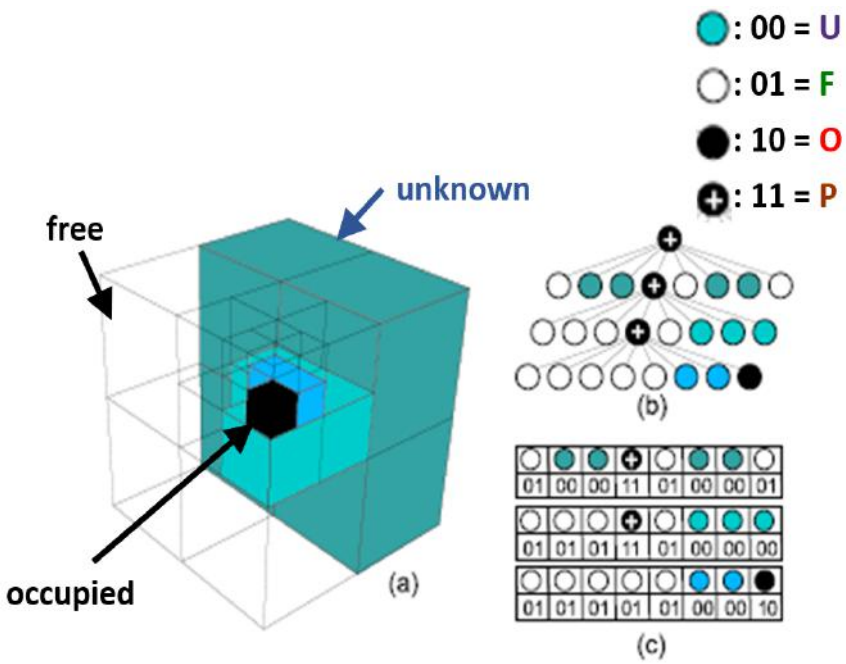


Figure 4-15: Description of a free (shaded white) and occupied (black) cell stored in an octree (a), the corresponding representation of the tree (b), and the corresponding compact bitstream in a directory (c) [61].

generally a volume occupancy in the sense of robotic mapping. The corresponding node in the octree is initialized if a certain volume is counted as occupied. Any uninitialized node in this Boolean setting may be free or unknown. The developers[61] of the Octomap has specifically represented free volumes in the tree in order to resolve this confusion. These are created along a ray determined with raycasting in the area between the sensor and the measured endpoint. Implicitly modeling regions of undefined space is not initialized. An octree example of free and occupied nodes from stereo camera data can be seen in *Fig*.4.16. Having Boolean occupancy states or discrete labels allows the octree to be compactly represented: if all children of a node have the same state (occupied or free) they can be pruned. This results in a major reduction in the number of nodes to be held in the list.

Octomap mapping provides ways to combine the compactness of octrees using discrete labels with probabilistic modeling updates and versatility.

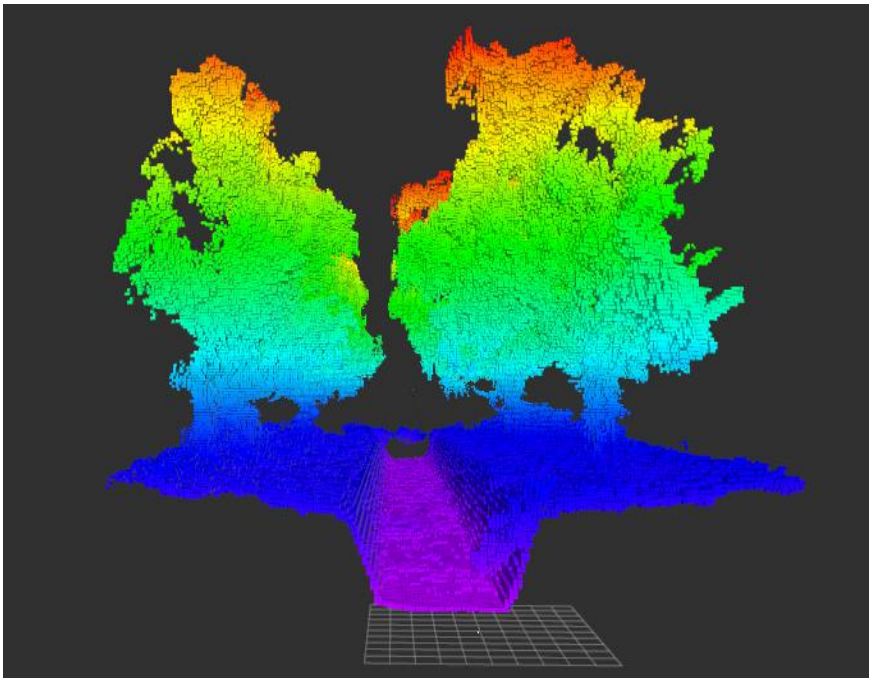


Figure 4-16: An octree example of free and occupied nodes from stereo camera data, in a canal like environment.

Because of the tree structure, octrees require overhead in terms of data access complexity relative to a fixed-size 3D grid. The complexity of $O(d) = O(log_n)$ can be performed with a single random query on a tree data structure containing n nodes with a tree depth of d. Traversing the entire tree in a depth-first way requires $O(n)$ complexity. An octree is limited to a defined total d_{max} depth in practice. This results in $O(d_{max})$ complexity of a random node search with d_{max} constant. Thus, the overhead relative to a similar 3D grid is constant for a fixed depth d_{max}

4.3.1 Probabilistic sensor fusion

Sensor readings are combined with occupancy grid mapping in Octomap mapping. The probability of occupying the $P(n)$ of a leaf node n provided the measurements of the sensor $z_{1:t}$ is calculated by

$$P\left(n|z_{1:t}\right) = \left[1 + \frac{1 - P(n|z_t)}{P(n|z_t)} \frac{1 - P(n|z_{1:t-1})}{P(n|z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right]^{-1}. \tag{4.1}$$

This update equation depends on z_t which is the current measurement, $P(n)$ which is a prior probability, and $P(n|z_{1:t-1})$ which is the previous estimate of a node leaf n. $P(n)$ is the term that refers to the likelihood of occupying voxel n given the z_t measurement. This value is unique to the sensor that has provided z_t.

$$L\left(n|z_{1:t}\right) = L(n|z_{1:t-1}) + L(n|z_t), \tag{4.2}$$

where,

$$L\left(n\right) = \log\left[\frac{P(n)}{1-P(n)}\right]. \tag{4.3}$$

This update rule formulation allows quicker updates as multiplications are replaced by additions. This update rule formulation allows quicker updates as multiplications are replaced by additions. The logarithms do not need to be determined during the update process in the case of pre-computed sensor models. Remember that log-odds values can be translated to probabilities and therefore store this value for each voxel instead of the likelihood of occupancy vice versa and Octomap. It is worth noting that this probability change has the same impact as counting hits and misses for certain sensor template configurations that are symmetrical, i.e. nodes being updated as hits have the same weight as those updated as misses.

A limit on the occupancy probability $P(n|z_{1:t-1})$ is often applied when a 3D map is used for navigation. When the threshold is reached, a voxel is considered to be occupied and is otherwise presumed to be free, thus creating two discrete states. From eq. (4.4) It is clear that, in order to change the voxel state, we need to incorporate as many observations as have been incorporated in order to define its currency. In other words, if a voxel has been observed free for k times, it should be observed occupied at least k times before it is considered to be occupied by the threshold (assuming free and occupied measurements are equally likely in the sensor model).

4.4 Octomap Robot Operating System (ROS) Implementation

The `Octomap_server` [61] ROS package is used in this work, which calculates and publishes OctoMaps as ROS messages `octomap_msgs/Octomap` .

Octomap is incorporated into ROS so that in a ROS environment, 3D octree occupancy grid maps can be created from 3D point cloud data. Octomap developers provide a map server for the ROS system which subscribes to point cloud published on ROS and creates a map using the `tf` library localization. The `rqt_graph` of Octomap_mapping is shown in the *Fig.*4.17. A stereo camera is used to publish the 3D point cloud data to the Octomap ROS node in the `/stereo/points` topic. On `/tf` topic, the transformations between sensor and map frames are published that actually represent the relative transformation between stereo and map frame. The node also provides services for visualizing and saving the map to disk. Flow chart of the ROS Octomap package, `Octomap_server` is shown in *Fig.*4.18.

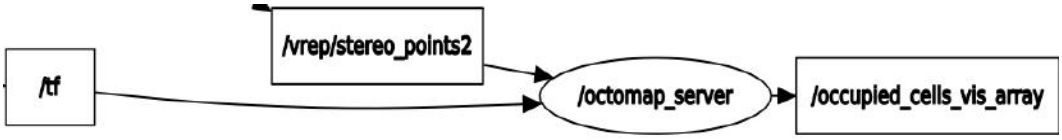


Figure 4-17: Topics connection between nodes of our canal system using Octomap.

The following topics are being published by octomap server:
`octomap_binary` : A compact binary version of the OctoMap that has `octomap_msgs/Octomap` message type and stores only `Free` and `occupied` voxel states.
`octomap_full` : Octomap_full is `octomap_msgs/Octomap` type of topic that publishes the `full state` of the OctoMap.
`occupied_cells_vis_array` : `Occupied_cells_vis_array` is a `visualization_msgs/MarkerArray` type of topic that is used to visualize occupied voxels in RVIZ.
`free_cells_vis_array` : `free_cells_vis_array` is a `visualization_msgs/MarkerArray` type of topic that is not published but can be enabled with the `publish_free_space`

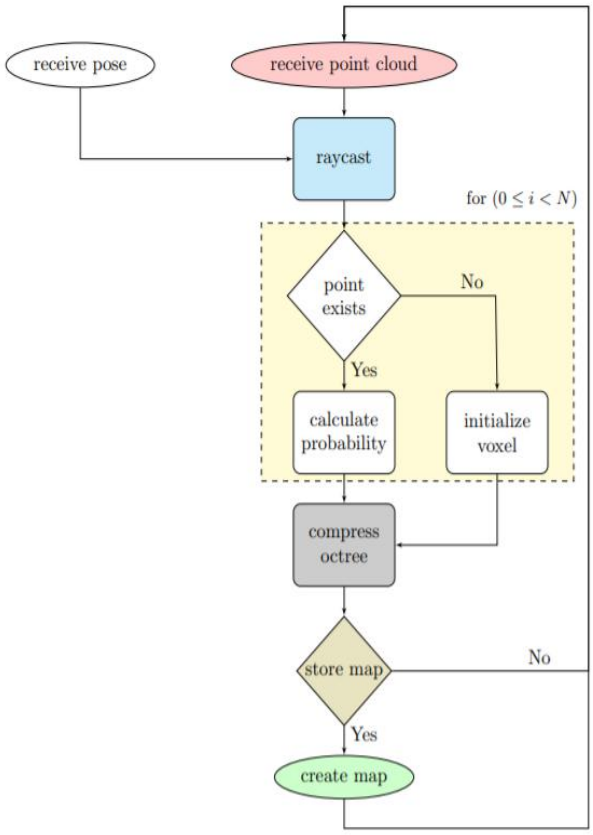


Figure 4-18: Flow chart of `Octomap_Server_Package`

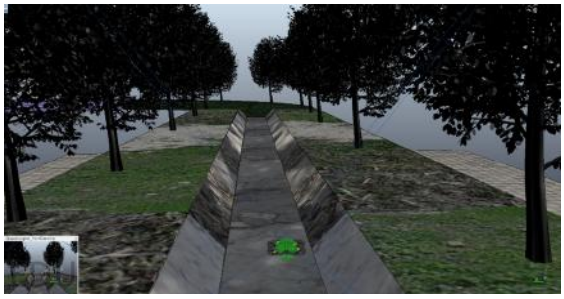
parameter.

`octomap_point_cloud_centers` : The middle points as a point cloud of all the filled voxels. Since there is no volume in a point cloud (as opposed to the box markers used in occupied cells vis array), there will be differences between voxel center points in the visualization. With the different voxel size resolutions, the gaps vary in size.

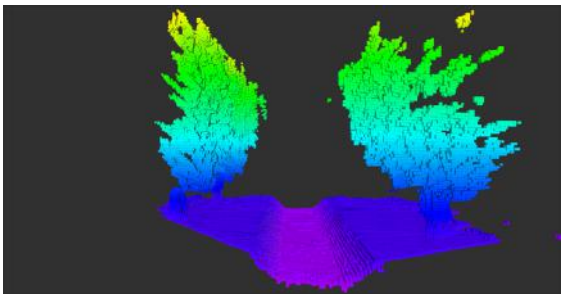
`projected_map` : A type of `nav_msgs/OccupancyGrid` that generates a 2D down projected occupancy map of the 3D OctoMap.

4.4.1 Octomap Outputs Using V-REP Physics engine

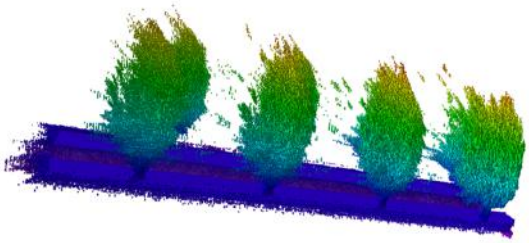
Local and global map of the canal environment using V-REP is shown in *Fig*.4.19.



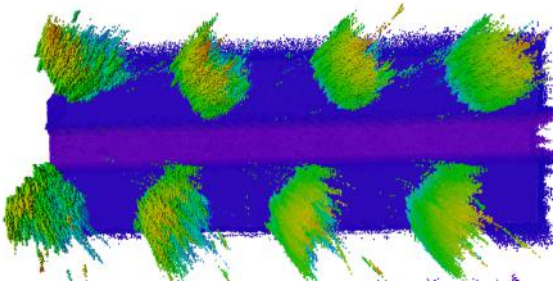
(a) Ground Truth



(b) Local Octomap of the canal



(c) Global Octomap(Side view)



(d) Global Octomap(Top view)

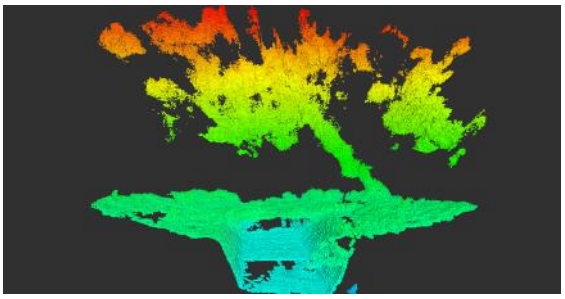
Figure 4-19: Octomap mapping Results using V-REP.

4.4.2 Octomap Outputs Using Unreal engine

Local Map

During the experiment an environment scene that contains a bridge and a fallen tree at the same location to analyze the 3D local map of both hard obstacles such as bridge and cluttered obstacles like trees. Both types of objects are fully and efficiently mapped, which can be seen from $Fig.4.20(a)(b)$.

(a) Local Ground Truth

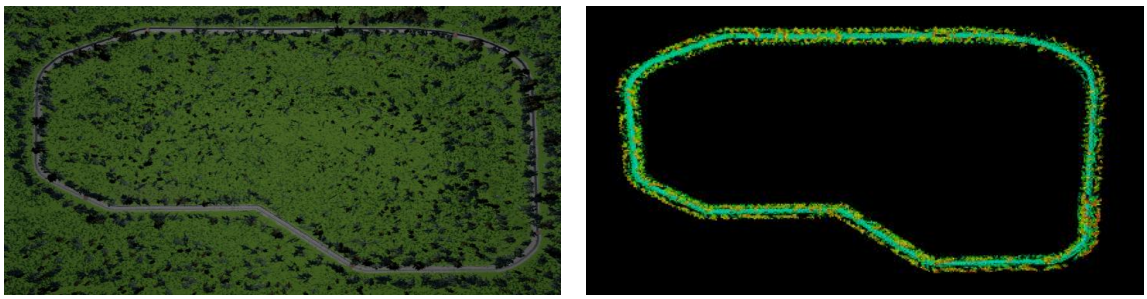


(b) Local Octomap of the canal

Figure 4-20: Local Octomap Results using Unreal engine.

Global Map

Fig.4.21(a) & (b) shows the actual ground truth and the global map of the simulation environment. While mapping, the velocity of the drone was $3.6km/h$, it took 39.63 minutes to complete the canal mapping and consumed $184.9MB$ of memory.



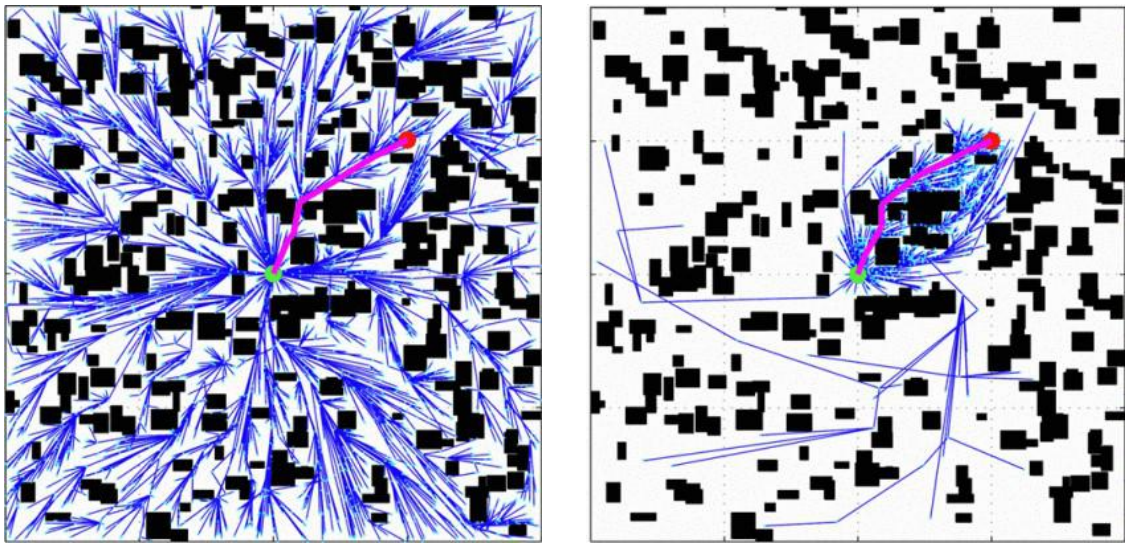
(a) Actual complete canal environment (b) Global map of canal environment

Figure 4-21: Global Octomap Results using Unreal engine.

4.5 Path Planning

The problem of motion-planning is often overcome either by first discretizing the continuous state space with a graph-based search grid or by sampling stochastic incremental searches. Graph-based searches, such as A* [36], are usually optimal resolution and full resolution. They are guaranteed to find the optimum solution, if a solution exists, and otherwise return failure (up to the discretization resolution). Such graph-based algorithms do not scale well with problem size (e.g., problem scope state dimension).

Stochastic searches, such as Rapidly Exploring Random Trees RRTs [43], Probabilistic Roadmap PRMs [41], and Expansive-Spaces Tree ESTs [39], use sampling-based approaches to ignore the need for state-space flexibility. It helps them to easily scale with the size of the problem and to consider kinodynamic constraints explicitly, but the result is a less rigorous guarantee of completeness. Probabilistically RRT's are complete, offering the likelihood of finding an effective solution if one is in existence, as iterations reach infinity, approach unity [31].



(a) RRT* (b) Informed RRT*

Figure 4-22: RRT^* and $InformedRRT^*$ solutions, same cost, and on a random world. Once an initial solution has been found, $InformedRRT^*$ focuses the search to optimize the solution in the ellipsoidal subset. That's why $InformedRRT^*$ is finding a better solution than RRT^* [31].

To date, these sampling-based algorithms have not made any predictions about the solution's optimality. Urmson and Simmons [58] found that using a heuristic sampling to bias improved RRT solutions, but did not measure the results formally. Ferguson and Stentz [29] understood that the length of a solution limits potential improvements from above and showed an iterative anytime RRT approach to solve a variety of progressively smaller planning problems. Karaman and Frazzoli [40] later showed that RRTs return to a suboptimal path with a single possibility showing that any RRT-based path can almost definitely be suboptimal and present a new class of optimal planners. Both optimal forms have been identified separately from RRTs and PRMs, RRT^* and PRM^*. Such algorithms are shown to be asymptotically optimal, with the chance to find the optimum resolution reaching unity as infinity approaches the variety of iterations.

RRTs are not asymptotically optimal because future expansion is biased by the existing state graph. By introducing incremental rewiring of the graph, RRT^* overcomes this [31]. Not only are new states added to a tree, but they are also considered to substitute parents for existing nearby tree states. This results in an algorithm with uniform global sampling that finds the optimal solution to the planning problem

asymptotically by finding the optimal paths from the initial state to each state in the problem domain asymptotically. This becomes costly in high dimensions and is also inconsistent with their single-query nature.

In this thesis, we concentrated on the issue of optimal route planning as it concerns reducing the length of the path in R^n. For such problems, the inclusion of states from an ellipsoidal subset of the planning domain is a necessary condition of improving the solution at any iteration. The probability of introducing these states by uniform sampling is indefinitely limited as the size of the planning problem decreases or the solution exceeds the hypothetical limit and provides an exact method for specifically sampling of the ellipsoidal subset. It is also shown that this direct sampling results in linear convergence to the optimal solution with strict assumptions (i.e., no obstacles).

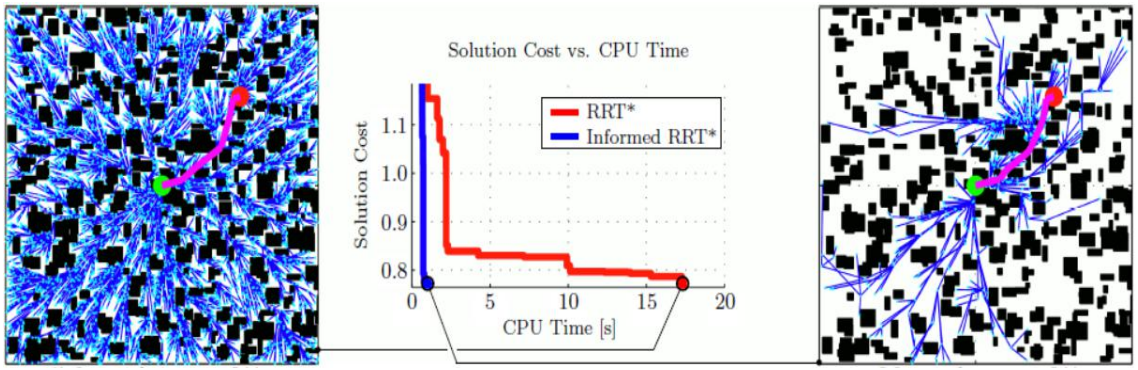


Figure 4-23: In a random world problem, the solution costs for RRT^* and $InformedRRT^*$ versus computational time [31].

This method of direct sampling enables informed-sampling planners to be developed. Such a planner, Informed RRT^*, is basically introduced to show the benefits of informed incremental search ($Fig.4.22$). $InformedRRT^*$ functions as RRT^* until a first solution is found, after which it can only test from the sub-state set specified by an admissible heuristic to boost the solution. This set implicitly balances exploitation versus exploration and does not require additional standardization (i.e. there are no additional parameters) or assumptions (i.e. all relevant homotopic categories are searched). Although heuristics may not always boost the search its importance in real-world planning shows its practicality. In situations where no additional in-

formation is provided (e.g. where the informed subset includes the entire planning problem), Informed RRT^* is RRT^* equivalent. $InformedRRT^*$ is an improved version of RRT^* that shows a clear improvement. When the configuration becomes more complex, it demonstrates huge improvements in order as shown in $Fig.4.23$. The algorithm is less reliant on the dimension and domain of the planning problem as well as the ability to find improved topologically distinct paths faster as a result of its focused search. It is also able to find solutions with comparable computation within tighter tolerances of the optimum than RRT^*, and in the absence of obstacles the optimum solution can be found within system zero in the end time ($Fig.4.24$).

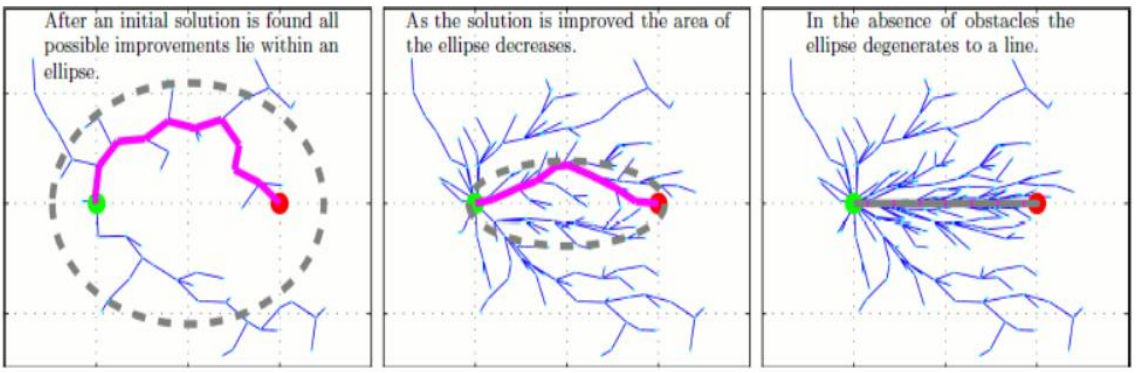


Figure 4-24: In the absence of obstacles, the path planned by $InformedRRT^*$ from start state to goal state is a straight line [31].

4.5.1 Informed RRT*

A pseudo algorithm of the informed RRT* is shown in $Algs.1$ and 2. It is the same as RRT* with the addition of lines 3, 6, 7, 30, and 31. Informed RRT*, like RRT*, searches for optimum path in a planning problem, by incrementally expanding a tree in state space, $T = (V, E)$, which consists of a set of vertices, and edges, towards randomly selected states, new vertices are being added by growing the graph in the free space. The graph is reconnected with each other such that the cost of the neighboring vertices is minimized. Informed RRT* differs from RRT* in a way that, when the solution is found, it focuses the search on the part that can minimize and improve the solution. It does it by directly sampling the ellipsoidal heuristic. As can

be seen from line 30, once a solution is found, informed RRT* adds it to the list of the possible solutions. The algorithm uses, the minimum of this list (line 6 describes this) to directly sample and estimate X_f.

It is convenient to describe the subfunctions introduced in the algorithms.

`Sample`: Given two states, $x_{start}, x_{goal} \in x_{free}$ and a maximum heuristic value, $C_{max} \in R$, the independent and identically distributed (i.i.d) from the state space, $x_{new} \in X$, is being returned from the function $Sample(x_{start}, x_{goal}, C_{max})$, such that the path between x_{start} and x_{goal} that has to go through x_{new} is less than C_{max} as described in $Alg.2$. In most planning problems this is computed only once at the start of the problem.

`InGoalRegion`: Given a state, $x \in X_{free}$, the $inGoalRegion$ function returns $True$ or $False$ if the state is in the goal region or not, respectively. Normally, a ball of radius r_{goal} is defined around r_{goal}, hence if the state is in the ball the function $InGoalRegion()$ returns $True$ otherwise $False$.

`RotationToWorldFrame`: Given two states, the focal points of hyperellipsoid, $x_{start}, x_{goal} \in X$, the rotation matrix $C \in SO(n)$ is returned by $RotationToWorldFrame()$ function, from the hyperellipsoid to the world frame as per line 6. This rotation is also calculated once at the start of the problem.

`Nearest Neighbor`: Given a graph tree $T = (V, E)$, a point $x \in X$, the function $Nearest : (T, x) \leftarrow v \in V$ which is closest to x in terms of distance, that is
$Nearest(T = (V, E), x) := argminv \in V \|x–v\|$

`Near Vertices`: Given a tree $T = (V, E)$, a point $p \in X$, and a number q such that $q \in R > 0$, the function $Near : (T, p, q)$ returns a vertex in V that lies inside a ball with radius q centered at p, i-e., $Near(T, p, q) := \{v \in V, v \in p, q\}$

`Steering`: The function $Steer(x, y)$ returns a point z in such a way so that the distance between y and z is closer than the distance between x and z.

`Collision Test`: $CollisionFree()$ is a Boolean function that returns $True$ if the state or the line between $[p, q] \in X_{free}$ is $free$ and $False$ otherwise.

Algorithm 1 Informed RRT*(X_{start}, X_{goal})

Input:
Output:

$V \leftarrow x_{start}$
$E \leftarrow \phi$;
$X_{soln} \leftarrow \phi$;
$T = (V, E)$;
for $iteration = 1...N$ **do**
    $c_{best} \leftarrow min_{xsoln} \in X_{soln}\{Cost(x_{soln})\}$;
    $x_{rand} \leftarrow Sample(x_{start}, x_{goal}, c_{best})$;
    $x_{nearest} \leftarrow Nearest(T, x_{rand})$;
    $x_{new} \leftarrow Steer(x_{nearest}, x_{rand})$;
    if $CollisionFree(x_{nearest}, x_{new})$ **then**
        $V \leftarrow x_{new}$;
        $X_{near} \leftarrow Near(T, x_{new}, r_{RRT^*})$;
        $x_{min} \leftarrow x_{nearest}$;
        $c_{min} \leftarrow Cost(x_{min}) + c.Line(x_{nearest}, x_{new})$;
        for $\forall x_{near} \in X_{near}$ **do**
            $c_{new} \leftarrow Cost(x_{near}) + c.Line(x_{near}, x_{new})$;
            if $c_{new} < c_{min}$ **then**
                if $CollisionFree(x_{near}, x_{new})$ **then**
                $X_{min} \leftarrow X_{near}$
                $c_{min} \leftarrow c_{new}$
                end if
            end if
        end for
        $E \leftarrow E \cup \{(X_{min}, X_{new})\}$
        for $\forall x_{near} \in X_{near}$ **do**
            $c_{near} < Cost(x_{near})$
            $c_{new} \leftarrow Cost(x_{new}) + c.Line(x_{near}, x_{new})$;
            if $c_{new} < c_{near}$ **then**
                if $CollisionFree(x_{near}, x_{new})$ **then**
                $x_{parent} < Parent(x_{near})$
                $E \leftarrow E \setminus \{(x_{parent}, x_{near})\}$
                $E \leftarrow E \cup \{(x_{new}, x_{near})\}$;
                end if
            end if
        end for
        if $InGoalRegion(x_{new})$ **then**
            $X_{soln} \leftarrow X_{soln} \cup x_{new}$;
        end if
    end if
end for
return T;

Algorithm 2 Sample (X_{start}, X_{goal})

Input:
Output:
if $c_{max} < \infty$ **then**
    $c_{min} \leftarrow \|x_{goal} - x_{start}\|_2$;
    $x_{center} \leftarrow (x_{start} + x_{goal})/2$;
    $C \leftarrow RotationToWorldFrame(x_{start} + x_{goal})$;
    $r_1 \leftarrow c_{max}/2$;
    $\{r_i\}_{i=2,....,n} \leftarrow (\sqrt{{c_{max}}^2 - {c_{min}}^2})/2$;
    $L \leftarrow diag\{r_1, r_2, ..., r_n\}$;
    $x_{ball} \leftarrow SampleUnitBall$;
    $x_{rand} \leftarrow (CLx_{ball} + x_{center}) \cap X$;

else
    $x_{rand} \sim U(X)$;
    return x_{rand};

4.6 Informed RRT* Robot Operating System (ROS) Implementation

For a long time, autonomous waypoint navigation was an integral part of drone applications. This approach works well when the drone flies at high altitudes without obstructions. However, in the case of low altitude flights, it becomes difficult for drones to navigate independently and require sensors to prevent them from colliding with the obstacles around them. Situations such as these could be avoided if a planning algorithm would take advantage of previous observations in the form of a 3D map and use it to guide the Micro-aerial vehicle (MAV) in the collision-free path to preserve the global navigation waypoint plan. This can be done by creating a mapping system that would use depth information from stereo cameras or lidars to create a map of occupancy. In order to navigate autonomously, the planner would use this map and global plan as input and build control commands for the MAV.

The idea is to continue to generate the environment's 3D map on the fly and attempt to reach the goal point by reactively calculating intermediate waypoints to the final goal, avoiding the obstacles on the map.

The 3D mapping of the environment is explained in the Octomap section of this

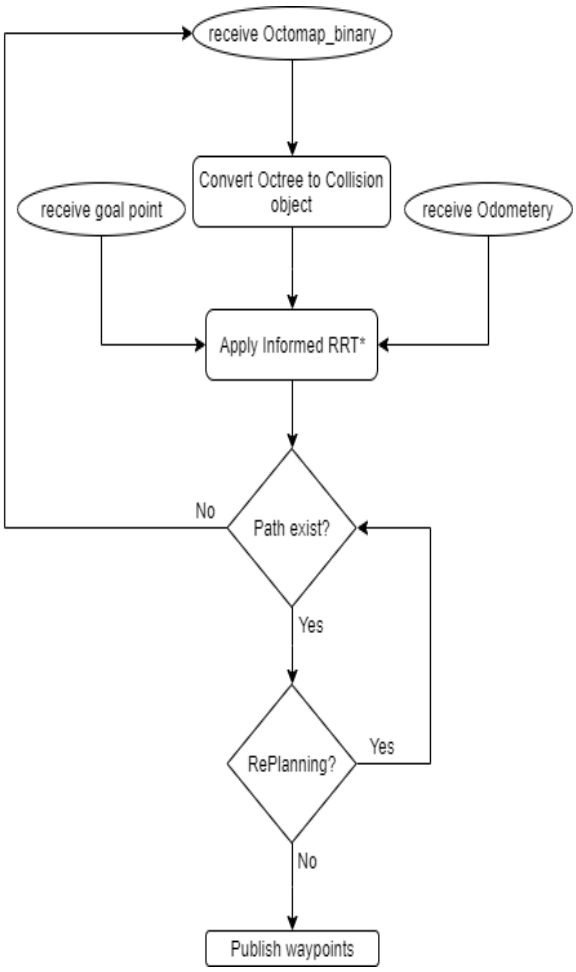


Figure 4-25: Flow chart of the $InformedRRT^*$ and ROS implementation.

chapter, where Octomap, an octree based data structure was implied. An Octree encodes the data on the 3D grid in the memory efficiently and allows operations such as traversing very fast. So the input to this tree is a point cloud and we get a binary occupancy map representation consisting of information about all the *occupied* and *unoccupied* cells after thresholding the probabilities.

In terms of path planning, we actually take advantage of the Flexible Collision Check (FCL) [4] and Open Motion Planning Library (OMPL) [7] libraries in implementing $InformedRRT^*$. Initially, `Octomap_binary, odometry, & goal_point` topics are subscribed by the planner, searches for a path from `start` to `goal` using OMPL. Meanwhile checks if each node is collision-free through free collision library (FCL). The optimal path returned by the planner is published on another ROS

topic of `trajectory_msgs::MultiDOFJointTrajectory` type, which in our case is `waypoints` . *Fig.* 4.25 shows the flow chart for the implementation of $InformedRRT^*$.

4.7 Autonomous canal exploration

The ultimate objective of this thesis is to autonomously navigate the drone over the canal while knowing the predefined GPS points at 25 meters apart from each other. Since our observation sensor can percept the environment for 25 meters so we have set this range as a local goal. *Fig.*4.26, shows the flow chart to autonomously explore the canal, assuming we have fixed GPS points at certain known locations.

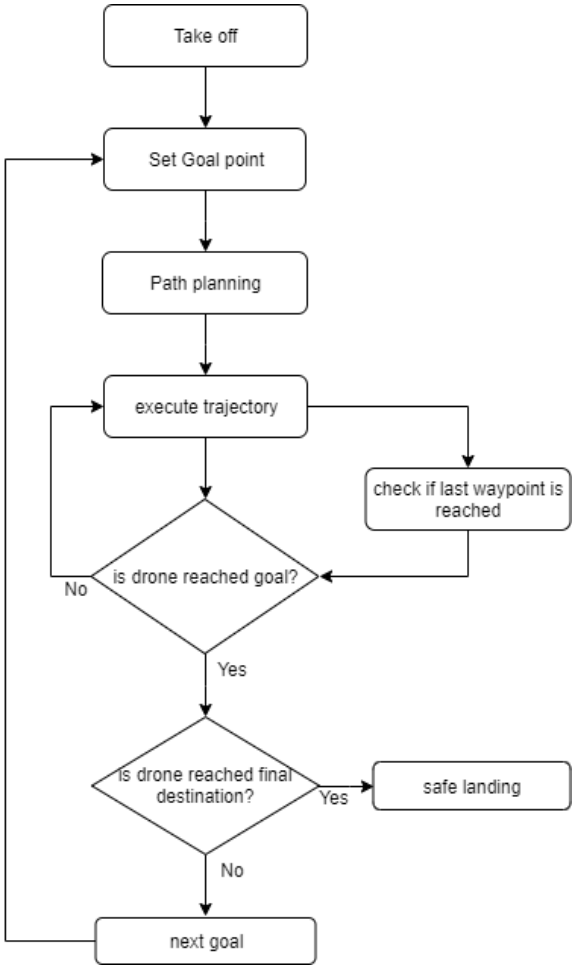


Figure 4-26: Flow chart of the autonomous canal exploration implementation.

`LocalGoals` : A 3 x $n - dimensional$ array, that contains the location of fixed points, 25 meters apart from each other, is supplied to the goal stack. This stack provides each goal iteratively when the drone reaches a circle centered at the previous goal with a certain radius r_{goal} .

`PathPlanning` : when the current position of the drone from odometry and local is supplied to the path planning part, it plans a path between the current position of the drone and the goal point.

`ExecuteTrajectory` : The planned path is then provided to the drone controller as a waypoint. The drone has to follow the path until reaches to the goal point.

`IsDroneReachedGoal` : After planning a path, the function $IsDroneReachedGoal$ returns $True$, if the drone has reached the circle centered at last point of the waypoint with a radius r_{goal} otherwise it returns $False$.

`IsDroneReachedFinalDestination`: The function $IsDroneReachedFinalDestination$ returns $True$ if the drone traverses all the goals in the goal stack, and $False$ if still, goals are to be traversed.

`NextGoal` : The function $NextGoal$ increment the pointer in the goal stack to be traversed. If all the goal points are traversed by drone, this function simply returns $None$.

`TakeOff` : The function $TakeOff$ sets the drone altitude to 4 meters and stay hovering until an obstacle-free path is planned to follow.

`SafeLanding` : If the drone traverses all the goal points and has reached the final destination point, the function $SafeLanding$ enables him to land safely.

Chapter 5

Experimental Results

5.1 Path planning evaluation

To assess the proposed method, we have evaluated the system in certain situations where it can either succeed or fail. Here, we introduce the word *situations*, where the environment, the starting position, and the target are given, the platform should be able to find a path and allow the drone to travel to the goal position without colliding with any obstacle.

5.1.1 Situation 1: No obstacles

There are no obstacles to this situation. Going in any direction is not going to lead to a collision. The purpose of this situation is to see if the path-planning algorithm is planning a straight path to the goal position. During the experiment, the start and goal position of the drone was (0, 1, 6) and (23, 1, 6), as *Fig.*5.1 clearly infers that the drone follows the straight path as there were no obstacles in the heading of the drone.

5.1.2 Situation 2: Hanging branches of the Tree

In this situation, the drone has to avoid tree branches hanging in the heading of the drone. In this case, the drone has two options, to fly over the top of the tree or

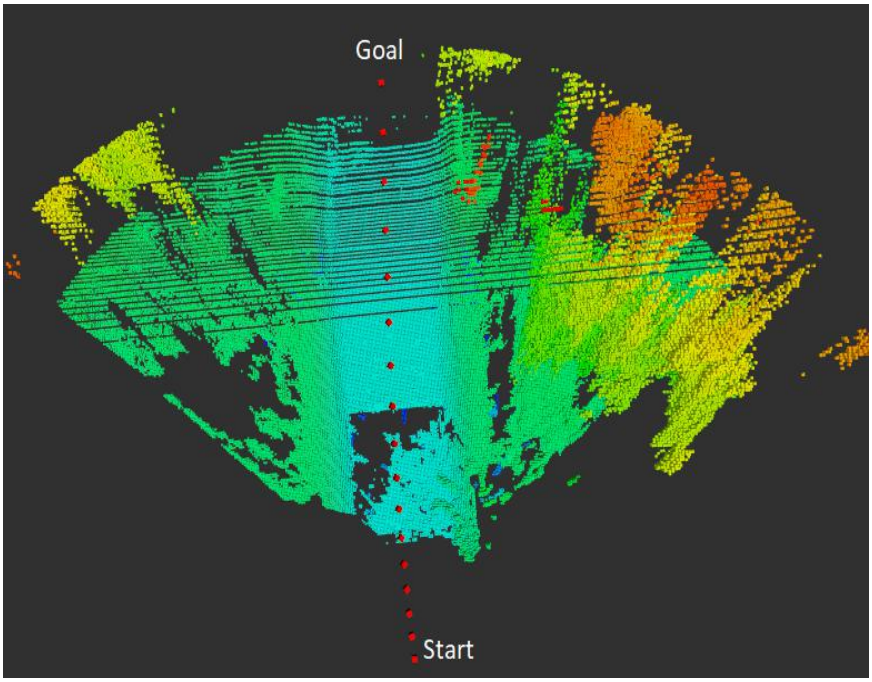


Figure 5-1: Situation 1, where there is no obstacle In the drone path, and the path-planning algorithm has to plan a straight path which is the shortest path.

make an attempt finding an obstacle-free window through the branches if any. In this experiment, a safe window for the drone exists where the drone can safely navigate through the window and reach the goal position as shown in the *Figures.*(5.2), (5.3) & (5.4).

Figure 5-2: Ground truth of situation 2, when there is hanging tree branches.

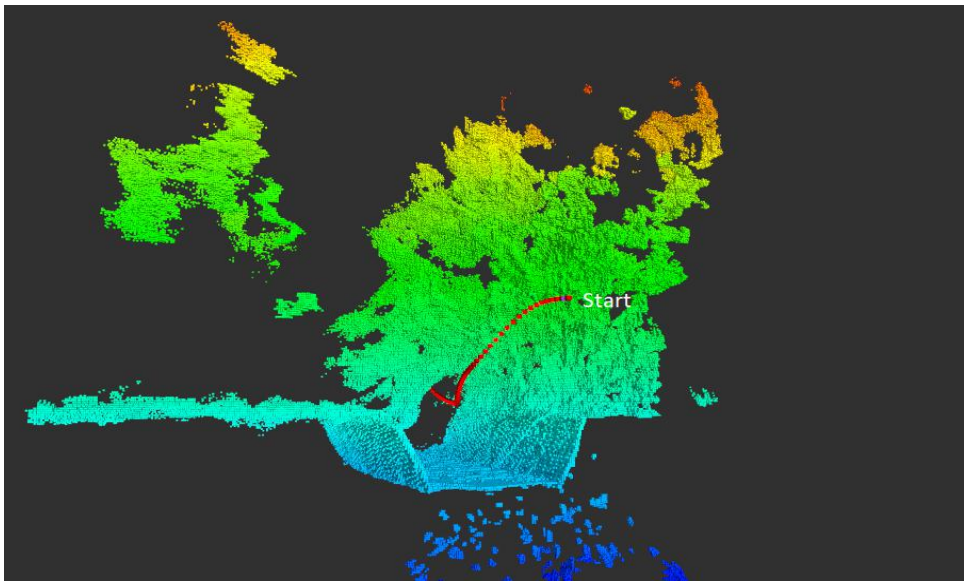


Figure 5-3: Planned path (front view).

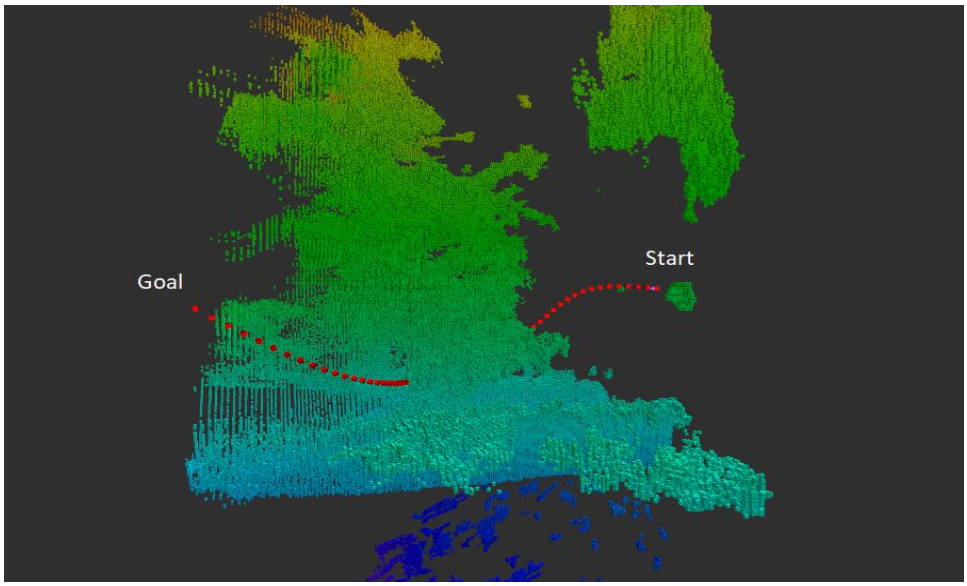


Figure 5-4: Planned path (side view).

Another similar experiment, where the canal is more denser with tree branches. The vehicle follows a similar trajectory as in the case of the above one. *Fig*.5.5 & 5.6 shows the planned path and actual ground truth.

Figure 5-5: Ground truth of situation 2, when there is a more cluttered tree branches in the path of the drone.

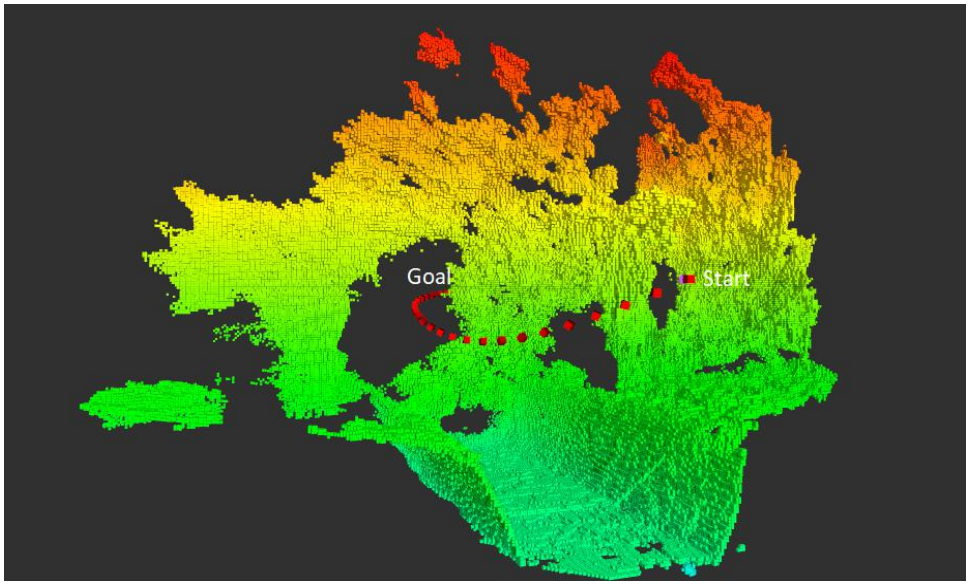


Figure 5-6: Drone path of situation 2.

5.1.3 Situation 3: Bridge Avoidance

In the canal like environment, one among the possible obstacles could be bridges over the canal. In this situation, an attempt was made to check the behavior of the vehicle while avoiding collision with a bridge that comes in its heading over the canal. Here, the vehicle has two possibilities, to cross over the bridge or cross under the bridge.

While experimenting, the algorithm shows both types of behaviors and crosses the bridge from either of the aspects. *Fig.*5.7 shows the ground truth and *Fig.*(5.8) & (5.9) shows the planned path for this situation.

(a) Bridge (Front view). (b) Bridge (Top view)

Figure 5-7: Ground truth of Situation 3.

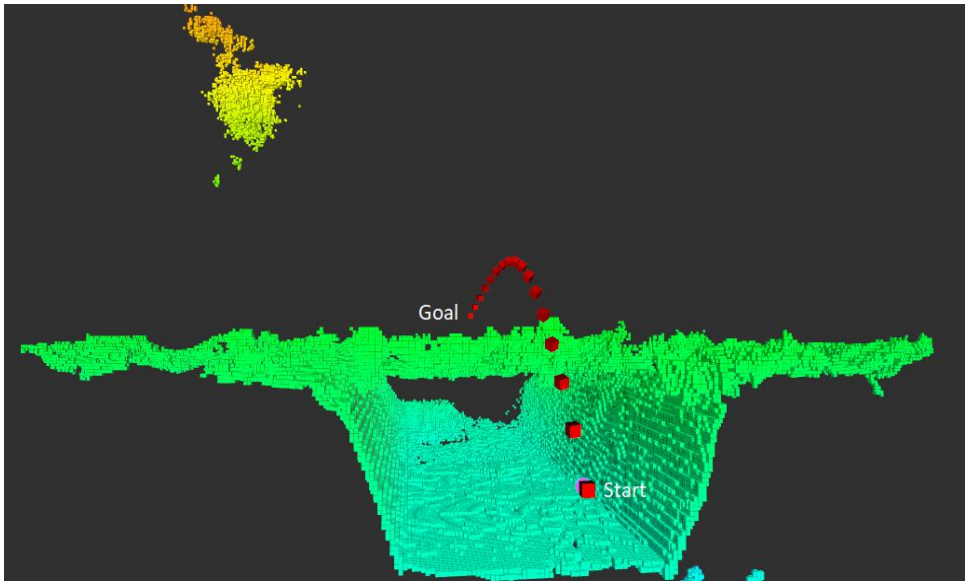


Figure 5-8: Drones trajectory, passing over the bridge (front view).

5.1.4 Situation 4: Avoidance of a tree trunk that is right above the canal

Another possible obstacle in a canal-like environment could be a tree trunk that is tilted towards the canal, as shown in *Fig.*5.10. The result of our method in this

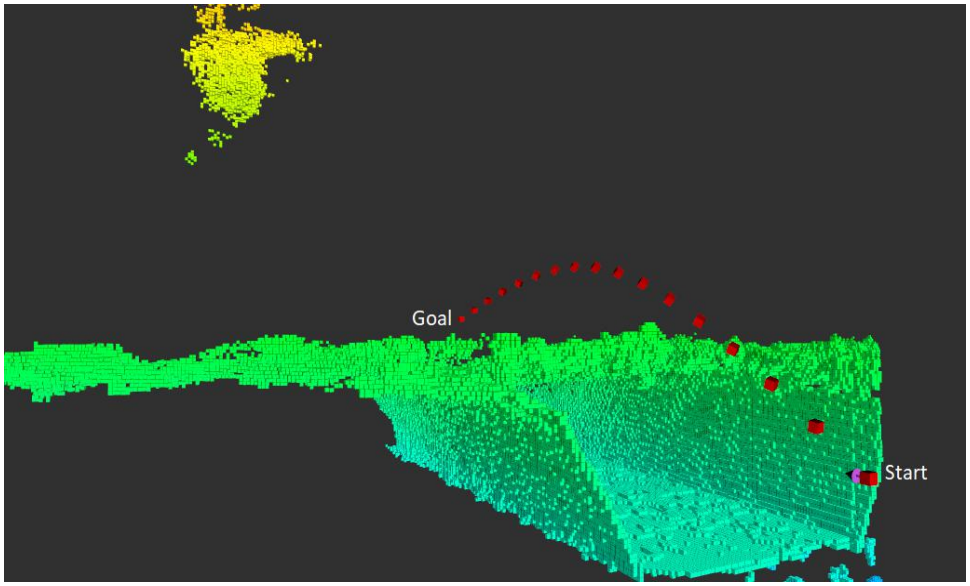


Figure 5-9: Drones trajectory, passing over the bridge (side view).

situation is amazing and the vehicle avoided it as can be seen from *Fig.*(5.11) & (2.12). These are probably not as much a hard obstacle for the drone as in the case of situation 2.

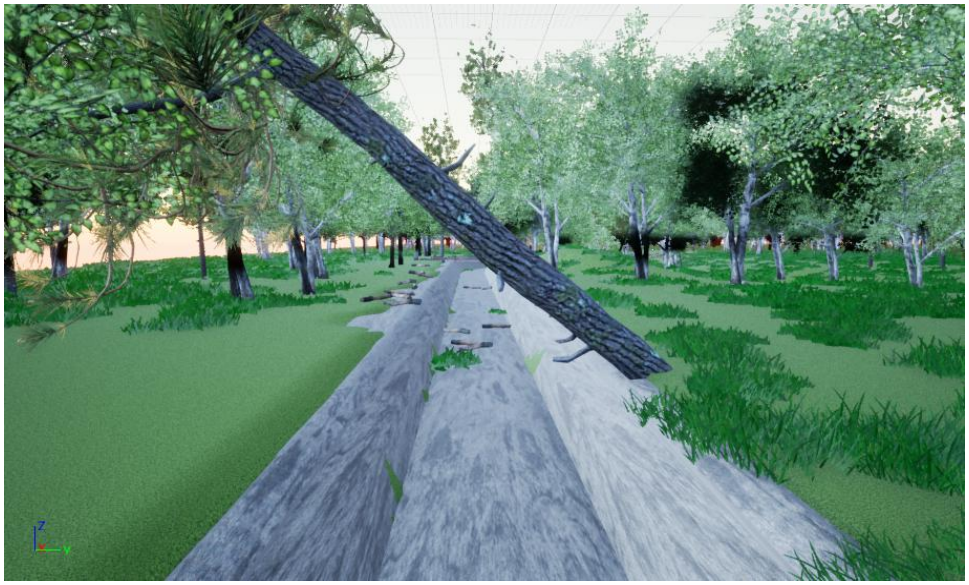


Figure 5-10: Ground truth of situation 4, where the drone has to avoid tree trunk that comes in the path of the drone.

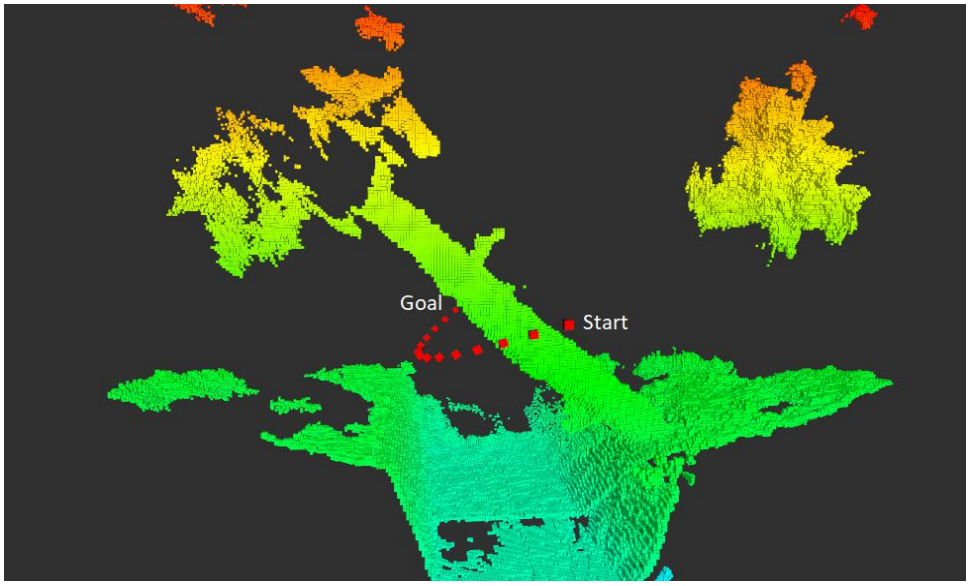


Figure 5-11: Drone trajectory for situation 4 (Front view).

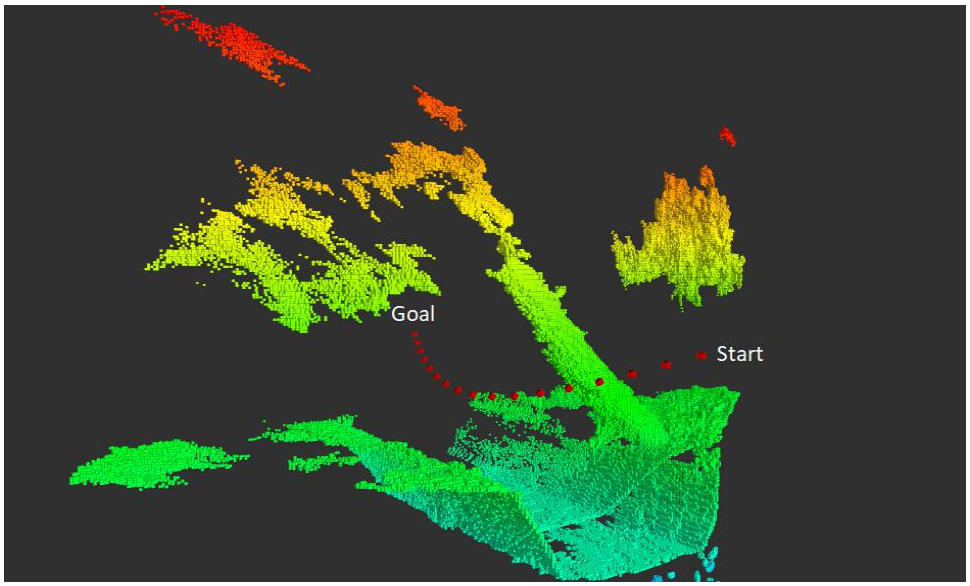


Figure 5-12: Drone trajectory for situation 4 (side view).

5.1.5 Situation 5: Canal completely stuck with obstacles

Unlike situation 1, here the path-planning algorithm has no choice to plan a path for the drone as the path ahead is completely blocked, however, since we have a complete map of the past, the drone can navigate back to any position if required.

Chapter 6

Conclusions & Future Work

The main objective of this thesis was to develop a collision-avoidance system for a Micro-aerial vehicle capable of operating autonomously in a canal-like environment in simulation. The vehicle is equipped with a stereo camera and a 2D LiDAR for sensing the environment in front of the vehicle in their respective ranges.

We used the Octomap method [61] to map the 3D canal structure. From the sensors data, Octomap makes a probabilistic map of the canal environment which has the ability to deal with sensor noise and a dynamic environment. The drawback with the use of OctoMap is that, in order to avoid obstacles, only obstacles within a limited range could be considered as looping through the entire map created is not computationally feasible. Also, during the execution, Octomap was observed to consume more processing and memory for long-run drone canal mapping. This problem can be resolved if voxel downsampling, resolution reduction, and reduction of the range of the sensor are incorporated.

The obstacle avoidance system is currently unable to detect small objects such as wires, leaves and thin branches of the tree. In order to overcome these deficiencies, the disparity image and Octomap could be further improved through suitable parameter selection.

Informed RRT* [31] shows promising results in almost every test scenario if a path exists. The algorithm is tested in five different test scenarios, and the planner effectively provided an optimum path to avoid all obstacles, keeping drones safety

distance from obstacles. After having the results of Informed RRT*, we felt that it is indeed the best planning algorithm for canal mapping application through an aerial vehicle, it will consume less power and time to complete a mission.

We used the airsim [55] plugin in the Unreal engine [11] to simulate the canal environment. One can build a near-real simulation environment in the Unreal engine. In our simulation environment, we include almost every possible obstacle that can happen in a real-world canal system, and an effort is being made to make it as much real as it could. To the best of our knowledge, this is the first attempt, a canal environment is simulated in a realistic way in the Unreal engine.

In this thesis, the localization was assumed to be perfect and we use the ground truth provided by the simulation engine. To have somehow realistic localization, noise could be added to the ground truth, and then apply state estimation.

We are finally able, to deliver a Micro-aerial vehicle, that has the capabilities of mapping the 3D structure of the canal-like environment, avoiding all obstacles, and planning an obstacle-free path from start to goal position of the canal. We tested our vehicle for a 2,378m length of canal mapping, as can be seen in the Octomap result part of this thesis.

Bibliography

[1] Asctec., available: http://www.asctec.de/, October 19 2019.

[2] Autodesk, available: https://www.autodesk.com/, October 19 2019.

[3] Binocular disparity, available: https://en.wikipedia.org/wiki/binocular_disparity, October 19 2019.

[4] flexible collision library, available: https://github.com/flexible-collision-library/fcl, October 19 2019.

[5] Hokuyo utm-30lx, available: https://www.hokuyo-aut.jp/search/single.php?serial=169, October 19 2019.

[6] Mikrocopter, available: http://www.mikrokopter.de/, October 19 2019.

[7] Open motion planning library, available: https://ompl.kavrakilab.org/, October 19 2019.

[8] Parrot, available: https://www.parrot.com/us/, October 19 2019.

[9] Ros stereo image proc, available: http://wiki.ros.org/stereo_image_proc, October 19 2019.

[10] Ros wiki, available: http://wiki.ros.org/, October 19 2019.

[11] Unreal engine 4, available: https://www.unrealengine.com, October 19 2019.

[12] V-rep physics engine, available: http://www.coppeliarobotics.com/, October 19 2019.

[13] Zed explorer, available: https://www.stereolabs.com/docs/getting-started/installation/, October 19 2019.

[14] Markus Achtelik, Abraham Bachrach, Ruijie He, Samuel Prentice, and Nicholas Roy. Autonomous navigation and exploration of a quadrotor helicopter in gps-denied indoor environments, 2009.

[15] Zahoor Ahmad, Rubab Khalid, and Abubakr Muhammad. Spatially distributed water quality monitoring using floating sensors. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society, Washington, DC, USA, October 21-23, 2018*, pages 2833–2838, 2018.

[16] Franz Andert, Florian-M. Adolf, Lukas Goormann, and Jörg S. Dittrich. Autonomous vision-based helicopter flights through obstacle gates. *Journal of Intelligent and Robotic Systems*, 57(1):259, Aug 2009.

[17] Umit Atila, İsmail Karaş, and Alias Rahman. A 3d-gis implementation for realizing 3d network analysis and routing simulation for evacuation purpose. *Lecture Notes in Geoinformation and Cartography*, pages 249–260, 10 2013.

[18] Marcelo Becker, Rafael Sampaio, Samir Bouabdallah, Vincent Perrot, and Roland Siegwart. In-flight collision avoidance controller based only on os4 embedded sensors. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 34:294–307, 09 2012.

[19] M. Blösch, S. Weiss, D. Scaramuzza, and R. Siegwart. Vision based mav navigation in unknown and unstructured environments. In *2010 IEEE International Conference on Robotics and Automation*, pages 21–28, May 2010.

[20] Samir Bouabdallah. Design and control of quadrotors with application to autonomous flying. 01 2007.

[21] Samir Bouabdallah, Marcelo Becker, Vincent de Perrot, and Roland Siegwart. Toward obstacle avoidance on quadrotors. 2007.

[22] J. Carsten, D. Ferguson, and A. Stentz. 3d field d: Improved path planning and replanning in three dimensions. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3381–3386, Oct 2006.

[23] K. Celik, S. Chung, M. Clausman, and A. K. Somani. Monocular vision slam for indoor aerial vehicles. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1566–1573, Oct 2009.

[24] D. M. Cole and P. M. Newman. Using laser range data for 3d slam in outdoor environments. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1556–1563, May 2006.

[25] Luca De Filippis, Giorgio Guglieri, and Fulvia Quagliotti. Path planning strategies for uavs in 3d environments. *Journal of Intelligent & Robotic Systems*, 65(1):247–264, Jan 2012.

[26] Jakob Engel, Jurgen Sturm, and Daniel Cremers. Camera-based navigation of a low-cost quadrocopter. pages 2815–2821, 10 2012.

[27] Jakob Engel, Jürgen Sturm, and Daniel Cremers. Accurate figure flying with a quadrocopter using onboard visual and inertial sensing. *IMU*, 320, 01 2012.

[28] Nathaniel Fairfield, George Kantor, and David Wettergreen. Real-time slam with octree evidence grids for exploration in underwater tunnels. *J. Field Robotics*, 24:03–21, 02 2007.

[29] David Ferguson and Anthony (Tony) Stentz. Anytime rrts. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '06)*, pages 5369 – 5375, October 2006.

[30] A. S. Gadre, S. Du, and D. J. Stilwell. A topological map based approach to long range operation of an unmanned surface vehicle. In *2012 American Control Conference (ACC)*, pages 5401–5407, June 2012.

[31] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2997–3004, Sep. 2014.

[32] S. Grzonka, G. Grisetti, and W. Burgard. Towards a navigation system for autonomous indoor flying. In *2009 IEEE International Conference on Robotics and Automation*, pages 2878–2883, May 2009.

[33] Slawomir Grzonka. *Mapping, state estimation, and navigation for quadrotors and human-worn sensor systems*. PhD thesis, 01 2011.

[34] Slawomir Grzonka, Giorgio Grisetti, and Wolfram Burgard. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics*, 28:90–100, 02 2012.

[35] Jens-Steffen Gutmann, Masaki Fukuchi, and Masahiro Fujita. 3d perception and environment map generation for humanoid robot navigation. *The International Journal of Robotics Research*, 27(10):1117–1134, 2008.

[36] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1972.

[37] M. Herbert, C. Caillas, Eric Krotkov, I.S. Kweon, and Takeo Kanade. Terrain mapping for a roving planetary explorer. pages 997 – 1002 vol.2, 06 1989.

[38] Stefan Hrabar and Gaurav Sukhatme. Vision-based navigation through urban canyons. *J. Field Robotics*, 26:431–452, 05 2009.

[39] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[40] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[41] L. E. Kavraki, P. Svestka, J. . Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, Aug 1996.

[42] OMAR KHALFAOUI. Development of an industrial robotic cell simulation environment for safe human robot interaction purposes. 10 2014.

[43] Steven M. LaValle and Jr. James J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[44] Hans Moravec. Robot spatial perception by stereoscopic vision and 3d evidence grids," robotics institute. 04 2011.

[45] Andreas Nuchter, Kai Lingemann, Joachim Hertzberg, and Hartmut Surmann. 6d slam - 3d mapping outdoor environments. *Fraunhofer IAIS*, 24, 11 2006.

[46] Satyarth Praveen. *Efficient Depth Estimation Using Sparse Stereo-Vision with Other Perception Techniques*. 05 2019.

[47] Asad Qureshi. Managing salinity in the indus basin of pakistan. *International Journal of River Basin Management*, Vol. 7, No. 2 (2009), pp. 111–117, 06 2009.

[48] S. Rathinam, P. Almeida, Z. Kim, S. Jackson, A. Tinka, W. Grossman, and R. Sengupta. Autonomous searching and tracking of a river using an uav. In *2007 American Control Conference*, pages 359–364, July 2007.

[49] Joern Rehder, Kamal Gupta, Stephen T. Nuske, and Sanjiv Singh. Global pose estimation with limited gps and long range visual odometry. In *Proceedings of IEEE Conference on Robotics and Automation*, May 2012.

[50] James Roberts, Timothy Stirling, Jean-Christophe Zufferey, and Dario Floreano. Quadrotor using minimal sensing for autonomous indoor flight. 01 2007.

[51] Y. Roth-Tabak and R. Jain. Building an environment model using depth information. *Computer*, 22(6):85–90, June 1989.

[52] Sebastian Scherer, Joern Rehder, Supreeth Achar, Hugh Cover, Andrew D. Chambers, Stephen T. Nuske, and Sanjiv Singh. River mapping from a flying robot: state estimation, river detection, and obstacle mapping. *Autonomous Robots*, 32(5):189 – 214, May 2012.

[53] Sebastian Scherer, Sanjiv Singh, Lyle Chamberlain, and Mike Elgersma. Flying fast and low among obstacles: Methodology and experiments. *The International Journal of Robotics Research*, 27(5):549–574, 2008.

[54] Flemming Scholer, Anders la Cour-Harbo, and Morten Bisgaard. Configuration space and visibility graph generation from geometric workspaces for uavs. 08 2011.

[55] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[56] S. Shen, N. Michael, and V. Kumar. Autonomous multi-floor indoor navigation with a computationally constrained mav. In *2011 IEEE International Conference on Robotics and Automation*, pages 20–25, May 2011.

[57] R. Triebel, P. Pfaff, and W. Burgard. Multi-level surface maps for outdoor terrain mapping and loop closing. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2276–2282, Oct 2006.

[58] C. Urmson and R. Simmons. Approaches for heuristically biasing rrt growth. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 2, pages 1178–1183 vol.2, Oct 2003.

[59] Andrew Viquerat, Lachlan Blackhall, Alistair Reid, Salah Sukkarieh, and Graham Brooker. *Reactive Collision Avoidance for Unmanned Aerial Vehicles Using Doppler Radar*, pages 245–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[60] Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Monocular-slam-based navigation for autonomous micro helicopters in gps-denied environments. *J. Field Robotics*, 28:854–874, 11 2011.

[61] Kai M. Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap : A probabilistic , flexible , and compact 3 d map representation for robotic systems. 2010.

[62] Fei Yan, Yi-Sha Liu, and Ji-Zhong Xiao. Path planning in complex 3d environments using a probabilistic roadmap method. *International Journal of Automation and Computing*, 10(6):525–533, Dec 2013.

[63] Junho Yang, Dushyant Rao, Soon-Jo Chung, and Seth Hutchinson. Monocular vision based navigation in gps-denied riverine environments. *AIAA Infotech at Aerospace Conference and Exhibit 2011*, 03 2011.

[64] K. Yang and S. Sukkarieh. Real-time continuous curvature path planning of uavs in cluttered environments. In *2008 5th International Symposium on Mechatronics and Its Applications*, pages 1–6, May 2008.

[65] Liang Yang, Juntong Qi, Jizhong Xiao, and Xia Yong. A literature review of uav 3d path planning. *Proceedings of the World Congress on Intelligent Control and Automation (WCICA)*, 2015:2376–2381, 03 2015.

[66] Manuel Yguel, Christopher Keat, Christophe Braillon, Christian Laugier, and Olivier Aycard. Dense mapping for range sensors: Efficient algorithms and sparse representations. 06 2007.

[67] Yuan Zhang. Localization and 2d mapping using low-cost lidar. 2018.

Appendix A

Airsim Codes

A.1 Acquiring Left, Right images and Vehicle Pose from Unreal-Airsim to ROS

```
#!/usr/bin/env python

import setup_path
import airsim
import numpy as np
import rospy
from sensor_msgs.msg import Image,CameraInfo
from tf2_msgs.msg import TFMessage
from geometry_msgs.msg import TransformStamped
from geometry_msgs.msg import PoseStamped
from math import pi
from cv_bridge import CvBridge
import cv2

CAMERA_FX = 224.066931372
CAMERA_FY = 224.066931372
CAMERA_CX = 320
CAMERA_CY = 240
```

```
19
20 Tx = -0.50 * CAMERA_FX
21
22 CAMERA_K1 = -0.000591
23 CAMERA_K2 = 0.000519
24 CAMERA_P1 = 0.000001
25 CAMERA_P2 = -0.000030
26 CAMERA_P3 = 0.0
27
28 IMAGE_WIDTH = 640  # resolution should match values in settings.json
29 IMAGE_HEIGHT = 480
30
31 class stereoPublisher:
32     def __init__(self):
33         self.bridge_rgb = CvBridge()
34         self.msg_rgb_right = Image()
35         self.msg_rgb_left = Image()
36         self.msg_info_right = CameraInfo()
37         self.msg_info_left = CameraInfo()
38         self.msg_tf = TFMessage()
39         self.sim_pose_msg = PoseStamped()
40         self.odom_msg = Odometry()
41
42     def getRGBImageRight(self,response_rgb):
43         img1d = np.fromstring(response_rgb.image_data_uint8, ...
               dtype=np.uint8)
44         img_rgb_right = img1d.reshape(response_rgb.height, ...
               response_rgb.width, 3)
45         img_rgb_right = img_rgb_right[..., :3][..., ::-1]
46         return img_rgb_right
47
48     def getRGBImageLeft(self,response_rgb):
49         img2d = np.fromstring(response_rgb.image_data_uint8, ...
               dtype=np.uint8)
50         img_rgb_left = img2d.reshape(response_rgb.height, ...
               response_rgb.width, 3)
```

```
51          img_rgb_left = img_rgb_left[..., :3][..., ::-1]
52          return img_rgb_left
53
54      def GetCurrentTime(self):
55          self.ros_time = rospy.Time.now()
56
57      def CreateRGBMessageRight(self,img_rgb_right):
58          self.msg_rgb_right.header.stamp = self.ros_time
59          self.msg_rgb_right.header.frame_id = "/rightCam_link"
60          self.msg_rgb_right.encoding = "bgr8"
61          self.msg_rgb_right.height = IMAGE_HEIGHT
62          self.msg_rgb_right.width = IMAGE_WIDTH
63          self.msg_rgb_right.data = ...
                self.bridge_rgb.cv2_to_imgmsg(img_rgb_right, "bgr8").data
64          self.msg_rgb_right.is_bigendian = 0
65          self.msg_rgb_right.step = self.msg_rgb_right.width * 3
66          return self.msg_rgb_right
67
68      def CreateRGBMessageLeft(self,img_rgb_left):
69          self.msg_rgb_left.header.stamp = self.ros_time
70          self.msg_rgb_left.header.frame_id = "/leftCam_link"
71          self.msg_rgb_left.encoding = "bgr8"
72          self.msg_rgb_left.height = IMAGE_HEIGHT
73          self.msg_rgb_left.width = IMAGE_WIDTH
74          self.msg_rgb_left.data = ...
                self.bridge_rgb.cv2_to_imgmsg(img_rgb_left, "bgr8").data
75          self.msg_rgb_left.is_bigendian = 0
76          self.msg_rgb_left.step = self.msg_rgb_left.width * 3
77          return self.msg_rgb_left
78
79      def CreateInfoMessageRight(self): #Right camera camera_info
80          self.msg_info_right.header.frame_id = "/rightCam_link"
81          self.msg_info_right.height = self.msg_rgb_right.height
82          self.msg_info_right.width = self.msg_rgb_right.width
83          self.msg_info_right.distortion_model = "plumb_bob"
84
```

```
85          self.msg_info_right.D.append(CAMERA_K1)
86          self.msg_info_right.D.append(CAMERA_K2)
87          self.msg_info_right.D.append(CAMERA_P1)
88          self.msg_info_right.D.append(CAMERA_P2)
89          self.msg_info_right.D.append(CAMERA_P3)
90
91          self.msg_info_right.K[0] = CAMERA_FX
92          self.msg_info_right.K[1] = 0
93          self.msg_info_right.K[2] = CAMERA_CX
94          self.msg_info_right.K[3] = 0
95          self.msg_info_right.K[4] = CAMERA_FY
96          self.msg_info_right.K[5] = CAMERA_CY
97          self.msg_info_right.K[6] = 0
98          self.msg_info_right.K[7] = 0
99          self.msg_info_right.K[8] = 1
100
101         self.msg_info_right.R[0] = -1
102         self.msg_info_right.R[1] = 0
103         self.msg_info_right.R[2] = 0
104         self.msg_info_right.R[3] = 0
105         self.msg_info_right.R[4] = -1
106         self.msg_info_right.R[5] = 0
107         self.msg_info_right.R[6] = 0
108         self.msg_info_right.R[7] = 0
109         self.msg_info_right.R[8] = -1
110
111         self.msg_info_right.P[0] = CAMERA_FX
112         self.msg_info_right.P[1] = 0
113         self.msg_info_right.P[2] = CAMERA_CX
114         self.msg_info_right.P[3] = Tx
115         self.msg_info_right.P[4] = 0
116         self.msg_info_right.P[5] = CAMERA_FY
117         self.msg_info_right.P[6] = CAMERA_CY
118         self.msg_info_right.P[7] = 0
119         self.msg_info_right.P[8] = 0
120         self.msg_info_right.P[9] = 0
```

```
121         self.msg_info_right.P[10] = 1
122         self.msg_info_right.P[11] = 0
123 
124         self.msg_info_right.binning_x = ...
               self.msg_info_right.binning_y = 0
125         self.msg_info_right.roi.x_offset = ...
               self.msg_info_right.roi.y_offset = ...
               self.msg_info_right.roi.height = ...
               self.msg_info_right.roi.width = 0
126         self.msg_info_right.roi.do_rectify = False
127         self.msg_info_right.header.stamp = ...
               self.msg_rgb_right.header.stamp
128         return self.msg_info_right
129 
130     def CreateInfoMessageLeft(self): #left camera camera_info
131         self.msg_info_left.header.frame_id = "/leftCam_link"
132         self.msg_info_left.height = self.msg_rgb_left.height
133         self.msg_info_left.width = self.msg_rgb_left.width
134         self.msg_info_left.distortion_model = "plumb_bob"
135 
136         self.msg_info_left.D.append(CAMERA_K1)
137         self.msg_info_left.D.append(CAMERA_K2)
138         self.msg_info_left.D.append(CAMERA_P1)
139         self.msg_info_left.D.append(CAMERA_P2)
140         self.msg_info_left.D.append(CAMERA_P3)
141 
142         self.msg_info_left.K[0] = CAMERA_FX
143         self.msg_info_left.K[1] = 0
144         self.msg_info_left.K[2] = CAMERA_CX
145         self.msg_info_left.K[3] = 0
146         self.msg_info_left.K[4] = CAMERA_FY
147         self.msg_info_left.K[5] = CAMERA_CY
148         self.msg_info_left.K[6] = 0
149         self.msg_info_left.K[7] = 0
150         self.msg_info_left.K[8] = 1
151 
```

```
152         self.msg_info_left.R[0] = 1
153         self.msg_info_left.R[1] = 0
154         self.msg_info_left.R[2] = 0
155         self.msg_info_left.R[3] = 0
156         self.msg_info_left.R[4] = 1
157         self.msg_info_left.R[5] = 0
158         self.msg_info_left.R[6] = 0
159         self.msg_info_left.R[7] = 0
160         self.msg_info_left.R[8] = 1
161 
162         self.msg_info_left.P[0] = CAMERA_FX
163         self.msg_info_left.P[1] = 0
164         self.msg_info_left.P[2] = CAMERA_CX
165         self.msg_info_left.P[3] = 0
166         self.msg_info_left.P[4] = 0
167         self.msg_info_left.P[5] = CAMERA_FY
168         self.msg_info_left.P[6] = CAMERA_CY
169         self.msg_info_left.P[7] = 0
170         self.msg_info_left.P[8] = 0
171         self.msg_info_left.P[9] = 0
172         self.msg_info_left.P[10] = 1
173         self.msg_info_left.P[11] = 0
174 
175         self.msg_info_left.binning_x = self.msg_info_left.binning_y ...
               = 0
176         self.msg_info_left.roi.x_offset = ...
               self.msg_info_left.roi.y_offset = ...
               self.msg_info_left.roi.height = ...
               self.msg_info_left.roi.width = 0
177         self.msg_info_left.roi.do_rectify = False
178         self.msg_info_left.header.stamp = self.msg_rgb_left.header.stamp
179         return self.msg_info_left
180 
181     def CreateTFMessage(self):
182         self.msg_tf.transforms.append(TransformStamped())
183         self.msg_tf.transforms[0].header.stamp = self.ros_time
```

```
184         self.msg_tf.transforms[0].header.frame_id = "/world"
185         self.msg_tf.transforms[0].child_frame_id = "/base_link"
186         self.msg_tf.transforms[0].transform.translation.x = ...
               sim_pose_msg.pose.position.x
187         self.msg_tf.transforms[0].transform.translation.y = ...
               sim_pose_msg.pose.position.y
188         self.msg_tf.transforms[0].transform.translation.z = ...
               sim_pose_msg.pose.position.z
189         self.msg_tf.transforms[0].transform.rotation.x = ...
               sim_pose_msg.pose.orientation.x
190         self.msg_tf.transforms[0].transform.rotation.y = ...
               sim_pose_msg.pose.orientation.y
191         self.msg_tf.transforms[0].transform.rotation.z = ...
               sim_pose_msg.pose.orientation.z
192         self.msg_tf.transforms[0].transform.rotation.w = ...
               sim_pose_msg.pose.orientation.w
193
194         self.msg_tf.transforms.append(TransformStamped())
195         self.msg_tf.transforms[1].header.stamp = self.ros_time
196         self.msg_tf.transforms[1].header.frame_id = "/base_link"
197         self.msg_tf.transforms[1].child_frame_id = "/stereo_link"
198         self.msg_tf.transforms[1].transform.translation.x = 0.46
199         self.msg_tf.transforms[1].transform.translation.y = 0.0
200         self.msg_tf.transforms[1].transform.translation.z = 0.0
201         qs = airsim.to_quaternion(0,0,0)
202         self.msg_tf.transforms[1].transform.rotation.x = qs.x_val
203         self.msg_tf.transforms[1].transform.rotation.y = qs.y_val
204         self.msg_tf.transforms[1].transform.rotation.z = qs.z_val
205         self.msg_tf.transforms[1].transform.rotation.w = qs.w_val
206
207         q = airsim.to_quaternion(0,pi/2,0)
208         self.msg_tf.transforms.append(TransformStamped())
209         self.msg_tf.transforms[2].header.stamp = self.ros_time
210         self.msg_tf.transforms[2].header.frame_id = "/stereo_link"
211         self.msg_tf.transforms[2].child_frame_id = ...
               "/leftCam_optical_link"
```

```
212         self.msg_tf.transforms[2].transform.translation.x = 0.0
213         self.msg_tf.transforms[2].transform.translation.y = -0.25
214         self.msg_tf.transforms[2].transform.translation.z = 0.0
215         self.msg_tf.transforms[2].transform.rotation.x = q.x_val
216         self.msg_tf.transforms[2].transform.rotation.y = q.y_val
217         self.msg_tf.transforms[2].transform.rotation.z = q.z_val
218         self.msg_tf.transforms[2].transform.rotation.w = q.w_val
219
220         self.msg_tf.transforms.append(TransformStamped())
221         self.msg_tf.transforms[3].header.stamp = self.ros_time
222         self.msg_tf.transforms[3].header.frame_id = "/stereo_link"
223         self.msg_tf.transforms[3].child_frame_id = ...
              "/rightCam_optical_link"
224         self.msg_tf.transforms[3].transform.translation.x = 0.0
225         self.msg_tf.transforms[3].transform.translation.y = 0.25
226         self.msg_tf.transforms[3].transform.translation.z = 0.0
227         self.msg_tf.transforms[3].transform.rotation.x = q.x_val
228         self.msg_tf.transforms[3].transform.rotation.y = q.y_val
229         self.msg_tf.transforms[3].transform.rotation.z = q.z_val
230         self.msg_tf.transforms[3].transform.rotation.w = q.w_val
231
232         qo = airsim.to_quaternion(pi/2,0,0)
233         self.msg_tf.transforms.append(TransformStamped())
234         self.msg_tf.transforms[4].header.stamp = self.ros_time
235         self.msg_tf.transforms[4].header.frame_id = ...
              "/leftCam_optical_link"
236         self.msg_tf.transforms[4].child_frame_id = "/leftCam_link"
237         self.msg_tf.transforms[4].transform.translation.x = 0.0
238         self.msg_tf.transforms[4].transform.translation.y = 0.0
239         self.msg_tf.transforms[4].transform.translation.z = 0.0
240         self.msg_tf.transforms[4].transform.rotation.x = qo.x_val
241         self.msg_tf.transforms[4].transform.rotation.y = qo.y_val
242         self.msg_tf.transforms[4].transform.rotation.z = qo.z_val
243         self.msg_tf.transforms[4].transform.rotation.w = qo.w_val
244
245         self.msg_tf.transforms.append(TransformStamped())
```

```
246         self.msg_tf.transforms[5].header.stamp = self.ros_time
247         self.msg_tf.transforms[5].header.frame_id = ...
               "/rightCam_optical_link"
248         self.msg_tf.transforms[5].child_frame_id = "/rightCam_link"
249         self.msg_tf.transforms[5].transform.translation.x = 0.0
250         self.msg_tf.transforms[5].transform.translation.y = 0.0
251         self.msg_tf.transforms[5].transform.translation.z = 0.0
252         self.msg_tf.transforms[5].transform.rotation.x = qo.x_val
253         self.msg_tf.transforms[5].transform.rotation.y = qo.y_val
254         self.msg_tf.transforms[5].transform.rotation.z = qo.z_val
255         self.msg_tf.transforms[5].transform.rotation.w = qo.w_val
256
257         return self.msg_tf
258
259     def get_sim_pose(self):
260         # get state of the multirotor
261         drone_state = client.simGetGroundTruthKinematics()
262         pos_ned = drone_state.position
263         orientation_ned = drone_state.orientation
264         pos_enu = airsim.Vector3r(pos_ned.x_val,
265                   -pos_ned.y_val,
266                  - pos_ned.z_val+9)
267         orientation_enu = airsim.Quaternionr(orientation_ned.w_val,
268                         - orientation_ned.z_val,
269                         - orientation_ned.x_val,
270                           orientation_ned.y_val)
271         # populate PoseStamped ros message
272         sim_pose_msg = PoseStamped()
273         sim_pose_msg.pose.position.x = pos_enu.x_val
274         sim_pose_msg.pose.position.y = pos_enu.y_val
275         sim_pose_msg.pose.position.z = pos_enu.z_val
276         sim_pose_msg.pose.orientation.w = orientation_enu.w_val
277         sim_pose_msg.pose.orientation.x = orientation_enu.x_val
278         sim_pose_msg.pose.orientation.y = orientation_enu.y_val
279         sim_pose_msg.pose.orientation.z = orientation_enu.z_val
280         sim_pose_msg.header.seq = 1
```

```
281         sim_pose_msg.header.frame_id = "world"
282         return sim_pose_msg
283
284 if __name__ == "__main__":
285     client = airsim.MultirotorClient()
286     client.confirmConnection()
287     client.enableApiControl(True)
288     client.armDisarm(True)
289
290     rospy.init_node('airsim_publisher', anonymous=True)
291     publisher_rgb_right = ...
            rospy.Publisher('/raw_stereo/right/image_raw', Image, ...
            queue_size=10)
292     publisher_rgb_left = ...
            rospy.Publisher('/raw_stereo/left/image_raw', Image, ...
            queue_size=10)
293     publisher_info_right = ...
            rospy.Publisher('/raw_stereo/right/camera_info', CameraInfo, ...
            queue_size=10)
294     publisher_info_left = ...
            rospy.Publisher('/raw_stereo/left/camera_info', CameraInfo, ...
            queue_size=10)
295     publisher_tf = rospy.Publisher('/tf', TFMessage, queue_size=10)
296     pose_pub = rospy.Publisher("airsim/pose", PoseStamped, queue_size=1)
297     rate = rospy.Rate(30)  # 30hz
298     pub = stereoPublisher()
299
300     while not rospy.is_shutdown():
301         sim_pose_msg = pub.get_sim_pose()
302         responses = client.simGetImages([airsim.ImageRequest("1", ...
              airsim.ImageType.Scene, False, False),
303                                          airsim.ImageRequest("2", ...
                                             airsim.ImageType.Scene, ...
                                             False, False)])
304         img_rgb_right = pub.getRGBImageRight(responses[0])
305         img_rgb_left = pub.getRGBImageLeft(responses[1])
```

```
306
307         pub.GetCurrentTime()
308         msg_rgb_right = pub.CreateRGBMessageRight(img_rgb_right)
309         msg_rgb_left = pub.CreateRGBMessageLeft(img_rgb_left)
310
311         msg_info_right = pub.CreateInfoMessageRight()
312         msg_info_left = pub.CreateInfoMessageLeft()
313
314         msg_tf = pub.CreateTFMessage()
315
316         publisher_rgb_right.publish(msg_rgb_right)
317         publisher_rgb_left.publish(msg_rgb_left)
318         publisher_info_right.publish(msg_info_right)
319         publisher_info_left.publish(msg_info_left)
320
321         publisher_tf.publish(msg_tf)
322         pose_pub.publish(sim_pose_msg)
323
324         del pub.msg_info_right.D[:]
325         del pub.msg_info_left.D[:]
326         del pub.msg_tf.transforms[:]
327
328         rate.sleep()
```

A.2 Acquiring LiDAR data from Airsim to ROS

```
1 #!/usr/bin/env python
2
3 import setup_path
4 import airsim
5 import pprint
6 import rospy
7 import tf
8 from std_msgs.msg import String
```

```
9  from sensor_msgs.msg import PointCloud
10 from sensor_msgs.msg import PointCloud2, PointField
11 from geometry_msgs.msg import Vector3, Point32
12 import numpy
13 import time
14
15 def lidarpcpub():
16     pub_front = rospy.Publisher("/ust_scan", PointCloud, queue_size=10)
17     rospy.init_node('lidarpcpub', anonymous=True)
18     rate = rospy.Rate(200)
19
20     # connect to the AirSim simulator
21     client = airsim.MultirotorClient()
22     client.confirmConnection()
23     simLidar = PointCloud()
24     simLidar2 = PointCloud()
25     vec = Vector3()
26
27     while not rospy.is_shutdown():
28         lidarData2 = ...
              client.getLidarData(lidar_name="FrontLidarSensor", ...
              vehicle_name= "Drone1")
29         if (len(lidarData2.point_cloud)< 3):
30             print("\tNo points received from Front Lidar Data")
31         else:
32             points2 = numpy.array(lidarData2.point_cloud, ...
                  dtype=numpy.dtype('f4'))
33             points2 = numpy.reshape(points2, ...
                  (int(points2.shape[0]/3), 3))
34
35                 #print("\tReading %d: time_stamp: %d ...
                       number_of_points: %d" % (i, ...
                       lidarData.time_stamp, len(points)))
36                 #rospy.loginfo(lidarData2.point_cloud)
37
38             simLidar2.header.stamp = rospy.Time.now()
```

```
39              simLidar2.header.frame_id = "leftCam_link"
40              simLidar2.points = []
41
42              for m in range(points2.shape[0]):
43                  simLidar2.points.append(Point32(points2[m][0],points2[m][1],points2[m][2
44
45          pub_front.publish(simLidar2)
46          rate.sleep()
47
48  # main
49  if __name__ == '__main__':
50      try:
51          lidarpcpub()
52      except rospy.ROSInterruptException:
53          pass
```