Master Thesis

Robotics Motion Planning and Control in the Wild through Camera Neural Perception

Theara SENG

Supervised by:

Thibault NEVEU

Master in Mobile, Autonomous and Robotic Systems,
Grenoble-INP ENSE3, UGA

August 19, 2022

Contents

ABSTRACT

The purpose of this project is the motion planning of a robot using a stereo camera. With the stereo camera, the disparity is obtained by matching every pixel in the left image with its corresponding pixel in the right image. Then the depth image is converted from the disparity based on the baseline and focal length of the camera. In this project, there are also several tasks which are done as explained below .

First is the trajectory tracking of a quadcopter. In this work, a mathematical model of a quadcopter's dynamics is derived, using Newton's and Euler's laws. Then a linearized version of the model is obtained. Model Predictive Control (MPC) method is an optimization that is used to track the trajectory path of the quadcopter. The goal of this optimization is to find the sequence of the inputs that minimize the previous cost function taking into account the constrained by the system dynamic.

Second, 3D mapping of an environment using the Airsim simulator. Airsim provides information about the RGB camera as well as the depth image. With this information, we can convert them into point cloud data using depth image proc which is the method is ROS. Octomap server is a package that subscribes to the point cloud data and converts it into a completed maximum-likelihood occupancy map as a compact Octomap binary stream also the down-projected 2D occupancy map from the 3D map.

Third, a 2D grip map of an environment using a Realsense D435 camera. Realsense D435 provides two grayscale images, an RGB image as well as the depth image. However, with the two grayscale images, the disparity is obtained by matching every pixel in the left image with its corresponding pixel in the right image. Then the distance is computed for each pair of matching pixels. A disparity map is obtained by representing such distance value as an intensity image. The depth image is inversely proportional to the disparity. When the disparity is near zero, small disparity differences produce large depth differences. However, when disparity is large, small disparity differences do not change the depth significantly.

Finally is the navigation of mobile robots using the Dijkstra algorithm with the help of the Dynamic Window Approach (DWA). A key component of Dijkstra's algorithm is that it keeps tracking a short distance from the start node to each individual cell. The DWA package is used to provide a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. The planner creates a kinematic trajectory for the robot to get from a start to a goal location.

NOMENCLATURE

DWA Dynamic Window Approach

ROS Robot Operating System

rviz ROS Visualization

tf Transform Frame

LIST OF FIQURES

1. Introduction

1.1. Study Background

Nowadays, an autonomous robot is widely used for several purposes such as industrial transport, logistics, food serving, and mobile operation. A critical challenge in the autonomous robotic is navigation which is the ability to answer the question "Where am I?" and "Where am I going?" .

Figure 1.1: Mobile Robot

The solution to these questions is known as localization which uses sensors to estimate the position of the robot. However, the robot also needs a map that represents the physical environment instead of using the preliminary map stored in its memory. The main reason is that the complete and geometrical representation of the environment increases the data amount and the computational complexity for the database searching during the robot localization and path planning process. Another reason is related to the possibility of robots working in unstructured and unknown environments. Such places may be dangerous or impossible for humans to construct a map. Then, the robots will build their map before being able to execute other tasks.

1.2. Objective of Project

The main functionality of the robot is navigation in which the robot will have the ability to navigate from the start position to the end position without collision with its surroundings.
To test with navigation algorithm, Gazebo is a software simulation that can be used to test the robot with the design environment.

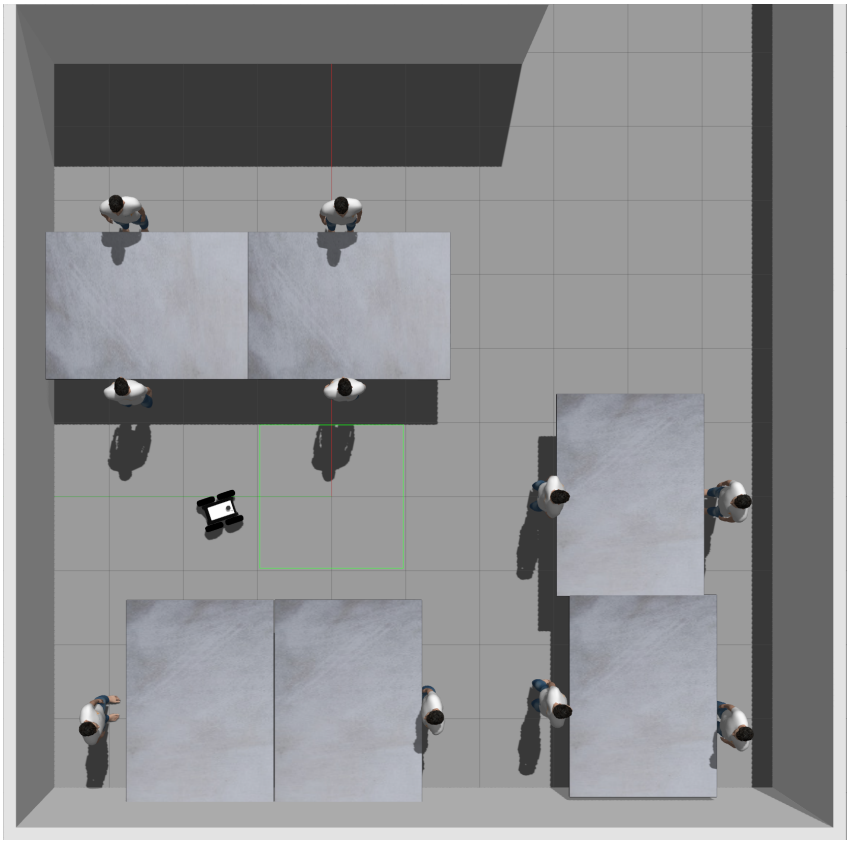


Figure 1.2: Gazebo Simulator

In this project, Airsim is also introduced to test the trajectory tracking of quadcopters using the Model Predictive Control.

Figure 1.3: Airsim Environment

1.3. Scope of Work

The scope of the project:

- Trajectory Tracking of the Quadcopters using Model Predictive Control with Airsim Simulator
- Depth Estimation using Airsim Simulator.
- 3D mapping of the Environment with Airsim
- Depth Estimation using Intel Realsense Camera.
- Simulation with Gazebo simulator to construct the 2D map
- Navigation of the mobile robot from a start position to the final position avoid the obstacle using Dijkstra algorithm with the help from DWA planner ROS .

1.4. Outline Thesis

There are seven main chapters which contains in this thesis:

- **Introduction:** it includes the study background, objective of project, scope of work and thesis study.
- **Mathematics Model:** It introduces dynamic model of the quadcopters as well as the discrete optimization of the Model Predictive Control (MPC)
- **Camera Depth Estimation:** It describes the relevant method to get the depth image from the stereo camera.
- **Trajectory Tracking of a Quadcopter:** It shows the graph result of the trajectory of quadcopters using MPC as well as the ROS nodes.
- **Mapping of an Environment:** It introduces the method of getting a map from the environment using Airsim Simulator and intel Realsense D435.
- **Navigation of a Mobile Robot:** Navigate from a start position to the goal position using Dijkstra algorithm with the help from Dynamic Window Approach.
- **Conclusion and Future Work:** It concludes the achieved progress with recommendation for future work development.

2. Mathematical Model

2.1. Quadcopters Dynamic

2.1.1. Euler Angles

Euler angles provide a way to represent the 3D orientation of an object using a combination of three rotations about different axes. The rotation matrix of Euler angles consist of Yaw, Pitch, and Roll as shown in **Figure 2.1**.

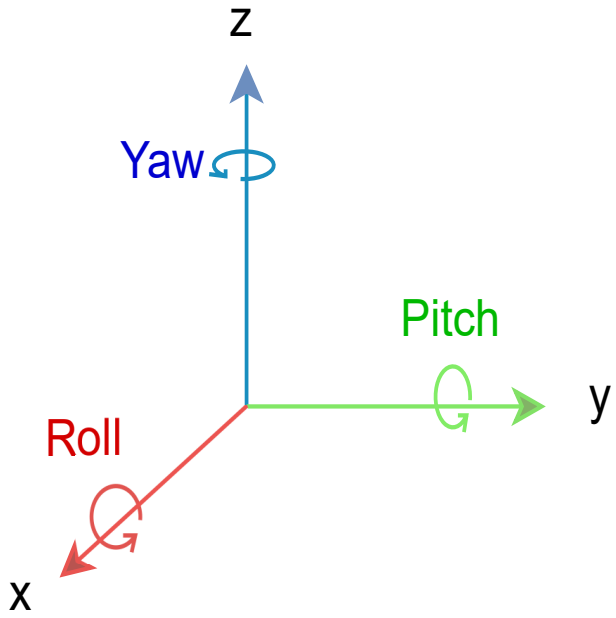


Figure 2.1: Euler Angles

Euler angles represent a sequence of three elemental rotations and the combination used is described by the following rotation matrices [15]

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

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

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

So the body reference coordinated are related by the rotation matrix $R_{zyx}(\phi, \theta, \psi)$

$$R_{zyx}(\phi, \theta, \psi) = R_z(\psi) R_y(\theta) R_x(\phi)$$

$$R_{zyx}(\phi, \theta, \psi) = \begin{bmatrix} c(\theta)c(\psi) & s(\phi)s(\theta)c(\psi) - c(\phi)s(\psi) & c(\phi)s(\theta)c(\psi) + s(\phi)s(\psi) \\ c(\theta)s(\psi) & s(\phi)s(\theta)s(\psi) + c(\phi)c(\psi) & c(\phi)s(\theta)s(\psi) - s(\phi)c(\psi) \\ -s(\theta) & s(\phi)c(\theta) & c(\phi)c(\theta) \end{bmatrix}$$

2.1.2. Quadcopter model

$[x \quad y \quad z \quad \phi \quad \theta \quad \psi]$ containing the linear and angular position of the quadcopter in earth frame.
$[u \quad v \quad w \quad p \quad q \quad r]$ containing the linear and angular velocity of the quadcopter in body frame [15]

$$\begin{aligned} v &= Rv_b \\ \omega &= T\omega_b \end{aligned}$$

$$\text{where} \quad v = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}, \quad v_b = \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \quad \omega = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad \text{and} \quad \omega_b = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

T is a matrix for angular transformation where

$$T = \begin{bmatrix} 1 & s(\phi)t(\theta) & c(\phi)t(\theta) \\ 0 & cos(\phi) & -s(\phi) \\ 0 & \frac{s(\phi)}{c(\theta)} & \frac{c(\phi)}{c(\theta)} \end{bmatrix}$$

Then:

$$\begin{aligned} \dot{x} &= w[s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta)] - v[c(\phi)s(\psi) - c(\psi)s(\phi)s(\theta)] + u[c(\psi)c(\theta)] \\ \dot{y} &= -w[c(\psi)s(\phi) - c(\phi)s(\psi)s(\theta)] + v[c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta)] + u[c(\theta)s(\psi)] \\ \dot{z} &= w[c(\phi)c(\theta)] + v[c(\theta)s(\phi)] - us(\theta) \\ \dot{\phi} &= p + r[c(\phi)t(\theta)] + q[s(\phi)t(\theta)] \\ \dot{\theta} &= qc(\phi) - rs(\phi) \\ \dot{\psi} &= r\frac{c(\phi)}{c(\theta)} + q\frac{s(\phi)}{c(\theta)} \end{aligned}$$

2.1.3. Newton's law

Total force acting on the quadcopter [13]

$$m(\omega_b \wedge v_b + \dot{v_b}) = f_b, \quad \text{where} \quad f_b = [f_x \quad f_y \quad f_z]^T$$

Total torque applied to the quadcopter [11]

$$I\dot{\omega_b} + \omega_b \wedge (I\omega_b) = m_b, \quad \text{where} \quad m_b = [m_x \quad m_y \quad m_z]^T$$

By

$$I = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

So the dynamic model of the quadcopter in the body frame is:

$$\begin{aligned} f_x &= m(\dot{u} + qw - rv) \\ f_y &= m(\dot{v} - pw + ru) \\ f_z &= m(\dot{w} + pv - qu) \\ m_x &= \dot{p}I_x - qr(I_y - Iz) \\ m_y &= \dot{q}I_y + pr(I_x - I_z) \\ m_z &= \dot{r}I_z - pq(I_x - I_y) \end{aligned}$$

- The external force in the body frame is given by [15]

$$f_B = mgR^T\hat{e}_z - f_t\hat{e}_3 + f_w$$

- The external moments in the body frame m_B is given by

$$m_B = \tau_B - g_a + \tau_w$$

Then:

$$\begin{aligned} -mgs(\theta) + f_{wx} &= m(\dot{u} + qw + -rv) \\ mgc(\theta)s(\phi) + f_{wx} &= m(\dot{v} - pw + ru) \\ mgc(\theta)c(\phi) + f_{wz} - f_t &= m(\dot{w} + pv - qu) \\ \tau_x + \tau_{wx} &= \dot{p}I_x - qr(I_y - Iz) \\ \tau_y + \tau_{wy} &= \dot{q}I_y + pr(I_x - I_z) \\ \tau_z + \tau_{wz} &= \dot{r}I_z - pq(I_x - I_y) \end{aligned}$$

The input that can be applied to the system in order to control the behavior of the quadcopter

$$\begin{aligned} f_t &= b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \\ \tau_x &= bl(\Omega_3^2 + \Omega_4^2 - \Omega_1^2 - \Omega_2^2) \\ \tau_y &= bl(\Omega_3^2 + \Omega_1^2 - \Omega_2^2 - \Omega_4^2) \\ \tau_z &= d(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \end{aligned}$$

- l is the distance between any rotor and the center of the drone
- b is the thrust factor
- d is the drag factor

2.1.4. State Space model

State space

$$\begin{cases} \dot{X} = AX + BU \\ Y = CX + DU \end{cases}$$

$$X = [\phi \quad \theta \quad \psi \quad p \quad q \quad r \quad u \quad v \quad w \quad x \quad y \quad z]$$
$$\dot{X} = [\dot{\phi} \quad \dot{\theta} \quad \dot{\psi} \quad \dot{p} \quad \dot{q} \quad \dot{r} \quad \dot{u} \quad \dot{v} \quad \dot{w} \quad \dot{x} \quad \dot{y} \quad \dot{z}]$$

From the dynamic and kinematic model of the quadcopter :

$$\begin{aligned}
\dot{\phi} &= p + r[c(\phi)t(\theta)] + q[s(\phi)t(\theta)] \\
\dot{\theta} &= q[c(\phi)] - r[s(\phi)] \\
\dot{\psi} &= r\frac{c(\phi)}{c(\theta)} + q\frac{s(\phi)}{c(\theta)} \\
\dot{p} &= \frac{I_y - I_z}{I_x}rq + \frac{\tau_x - \tau_{wx}}{I_x} \\
\dot{q} &= \frac{I_z - I_x}{I_y}pr + \frac{\tau_y + \tau_{wy}}{I_y} \\
\dot{r} &= \frac{I_x - I_y}{I_z}pq + \frac{\tau_z + \tau_{wz}}{I_z} \\
\dot{u} &= rv - qw - g[s(\theta)] + \frac{f_{wx}}{m} \\
\dot{v} &= pw - ru + g[s(\phi)c(\theta)] + \frac{f_{wy}}{m} \\
\dot{w} &= qu - pv + g[c(\theta)c(\phi)] + \frac{f_{wz} - f_t}{m} \\
\dot{x} &= w[s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta)] - v[c(\phi)s(\psi) - c(\psi)s(\phi)s(\theta)] + u[c(\psi)c(\theta)] \\
\dot{y} &= -w[c(\psi)s(\phi) - c(\phi)s(\psi)s(\theta)] + v[c(\phi)c(\psi) + s(\phi)s(\psi)s(\theta)] + u[c(\theta)s(\psi)] \\
\dot{z} &= w[c(\phi)c(\theta)] + v[c(\theta)s(\phi)] - u[s(\theta)]
\end{aligned}$$

[14]If angle is small $cos(angle) = 1$ and $sin(angle) = angle$
then :

$$\begin{aligned}
\dot{\phi} &= p + r\theta + q\phi\theta \\
\dot{\theta} &= q - r\phi \\
\dot{\psi} &= r + q\phi \\
\dot{p} &= \frac{I_y - I_z}{I_x} rq + \frac{\tau_x - \tau_{wx}}{I_x} \\
\dot{q} &= \frac{I_z - I_x}{I_y} pr + \frac{\tau_y + \tau_{wy}}{I_y} \\
\dot{r} &= \frac{I_x - I_y}{I_z} pq + \frac{\tau_z + \tau_{wz}}{I_z} \\
\dot{u} &= rv - qw - g\theta + \frac{f_{wx}}{m} \\
\dot{v} &= pw - ru + g\phi + \frac{f_{wy}}{m} \\
\dot{w} &= qu - pv + g + \frac{f_{wz} - f_t}{m} \\
\dot{x} &= w(\phi\psi + \theta) - v(\psi - \phi\theta) + u \\
\dot{y} &= v(1 + \phi\theta\psi) - w(\phi - \psi\theta) + u\psi \\
\dot{z} &= w + v\phi - u\theta
\end{aligned}$$

2.1.5. Linearization

In order to perform the linearization, an equilibrium point is needed. Such an equilibrium point can be

$$\begin{aligned}
\bar{x} &= [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad \bar{x} \quad \bar{y} \quad \bar{z}]^T \\
\bar{u} &= [mg \quad 0 \quad 0 \quad 0]^T
\end{aligned}$$

Then:

$$A = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix} \quad and \quad B = \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & \frac{1}{I_x} & 0 & 0 \\
0 & 0 & \frac{1}{I_y} & 0 \\
0 & 0 & 0 & \frac{1}{I_z} \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\frac{1}{m} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad and \quad D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

From newton's law:

$$m\dot{v} = Rf_B = mg\hat{e_z} - f_t R\hat{e_3}$$

$$\begin{cases} \ddot{x} = -\frac{f_t}{m}[s(\phi)s(\psi) + c(\phi)c(\psi)s(\theta)] \\ \ddot{y} = -\frac{f_t}{m}[c(\phi)s(\psi)s(\theta) - c(\psi)s(\phi)] \\ \ddot{z} = g - \frac{f_t}{m}[c(\phi)c(\theta)] \end{cases}$$

Now a simplified is made by setting $[\dot{\phi} \quad \dot{\theta} \quad \dot{\psi}] = [p \quad q \quad r]$. This assumption hold true for small angle of movement [17].

$$\begin{cases} \ddot{\phi} = \frac{I_y - I_z}{I_x}\dot{\theta}\dot{\psi} + \frac{\tau_x - \tau_{wx}}{I_x} \\ \ddot{\theta} = \frac{I_z - I_x}{I_y}\dot{\phi}\dot{\psi} + \frac{\tau_y + \tau_{wy}}{I_y} \\ \ddot{\psi} = \frac{I_x - I_y}{I_z}\dot{\phi}\dot{\theta} + \frac{\tau_z + \tau_{wz}}{I_z} \end{cases}$$

Linearize the Newtow second law:

$$\begin{cases} \ddot{\phi} = \frac{\tau_x}{I_x} \\ \ddot{\theta} = \frac{\tau_y}{I_y} \\ \ddot{\psi} = \frac{\tau_z}{I_z} \\ \ddot{x} = -g\theta \\ \ddot{y} = g\phi \\ \ddot{z} = -(g - \frac{f_t}{m}) \end{cases}$$

2.2. Model Predictive Control

2.2.1. Optimization problem

The cost function can be written:

$$J = J_e + J_u$$

- J_e represents precision cost

$$J_e = \int_0^t e_\tau^T Q e(\tau) d\tau, \quad \text{where} \quad e(t) = \overrightarrow{ref} - Cx(t)$$

- J_u represents energetic cost

$$J_u = \int_0^t \vec{U}^T \tau R \vec{U}(\tau) d\tau$$

The cost function can be expressed as the weight module of the error and the command respectively

- Q and R are the weight matrix
- C is the state matrix that is related to output of the system

The goal of the optimization is to find the sequence of the inputs that minimize the previous cost function taking into account the constraint imposed by the system dynamic and physical limitation of the controller[10].

$$\begin{aligned} \min_U(J) &= \min_U(J_e + J_u) \\ \min_U(J) &= \min_{U(t)} \frac{1}{2} \int_0^t [e^T \tau Q \vec{e}(\tau) + \vec{U}^T(\tau) R \vec{U}(\tau)] d\tau \end{aligned}$$

$$\text{Constrains:} \quad \begin{cases} \dot{x} = f(x(t), U(t)) \\ x(0) = x_0 \\ U_L \leq U \leq U_R \end{cases}$$

- U_L lower boundary.
- U_U upper boundary.
- x_0 is the initial state

2.2.2. Discrete Optimization

$$\begin{aligned} J &= J_e + J_u \\ J_e &= \sum_{k=1}^{N} \vec{e}^T(k) Q \vec{e}(k), \vec{e} = \overrightarrow{ref} - C\vec{x}(k) \\ J_u &= \sum_{k=0}^{N-1} \vec{U}^T(k) R \vec{U}(k) \\ \min_{U(k)} J &= \min_{U(k)} \frac{1}{2} \sum_{k=0}^{N-1} [\vec{e}^T(k) Q \vec{e}(k) + \vec{U}^T(k) R \vec{U}(k)] \end{aligned}$$

$$\text{Constrain:} \begin{cases} \vec{x}(k+1) = A\vec{x}(k) + B\vec{U}(k) \\ x(0) = \vec{x_0} \\ U_L \leq U \leq U_U \end{cases}$$

Q and R has a more immediate effect on the behavior of the quadcopter. The value of the Q will be kept between 1 and 0, depending on if we want that state to affect the result of the optimal problem. The only states that take into account are the 3 position states and the way angle. The value of Q will remain constant while the value of R will be be used to tune the MPC .

- The first value affects the throttle command.
- The second value affects the roll.
- The third value affects the pitch.
- The fourth value affects the yaw.

The lower values would mean a faster system because the controller will use higher command, but it will result in bigger overshoot or even an unstable close loop system. The value for throttle and yaw can be set higher but not more than one.

3. Camera Depth Estimation

3.1. Pinhole Camera

Cameras are the sensors used to capture images. They take the points in the world and project them onto a 2D plane which we see as images. This transformation is usually divided into two parts which are Extrinsic and Intrinsic. The extrinsic parameters of a camera depend on its location and orientation and have nothing to do with its internal parameters. However, the intrinsic parameters of a camera depend on how it capture the images and connect to the parameters of the focal length, aperture, field of view and resolution. The commonly used coordinated systems in Computer Vision are:

- World coordinate system (3D)
- Camera coordinate system (3D)
- Image coordinate system (2D)
- Pixel coordinate system (2D)

3.1.1. World to Camera Coordinate

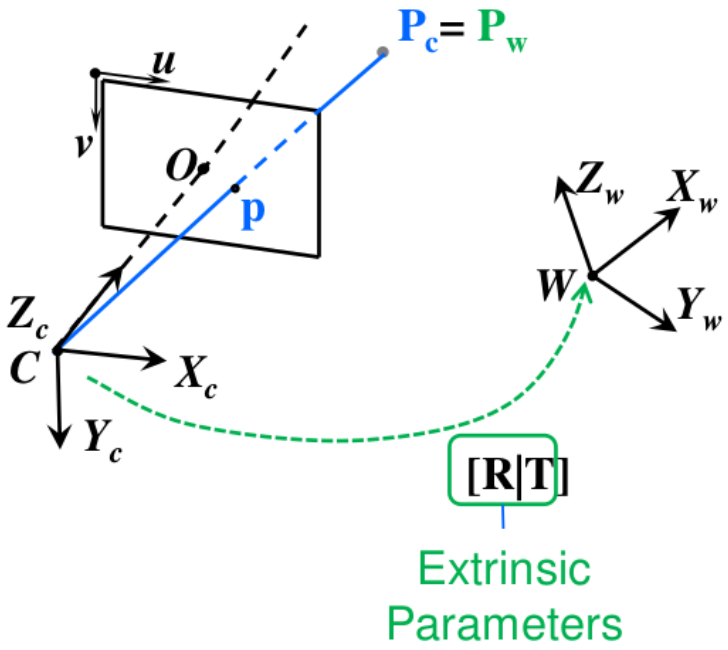


Figure 3.1: World to Camera Coordinate

$[X_C \quad Y_C \quad Z_C]$ is the camera coordinate system that measure relative to camera's orientation. The 4×4 transformation matrix that converts points from the world coordinate system to the camera coordinate system is known as the camera extrinsic

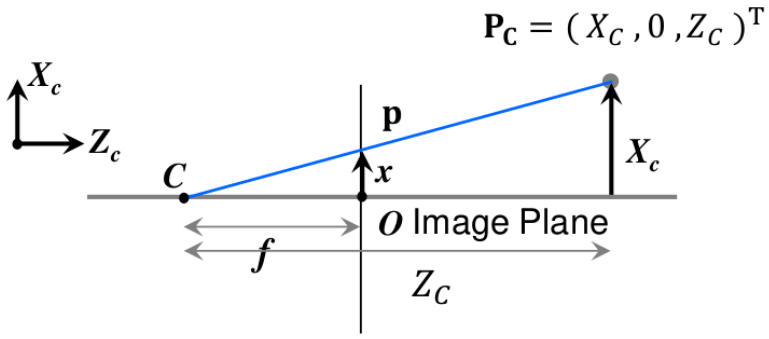


Figure 3.2: Camera to Image Coordinate

matrix [18]. The camera extrinsic matrix change if the physical location and orientation of the camera is changed.

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ 0_{1\times3} & 1_{1\times1} \end{bmatrix}_{4\times4} \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix} = [R|T] \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

- where $[R|T]$ is the camera extrinsic matrix with the rotation R and transition T operation.

3.1.2. Camera to Image Coordinate

$[x \quad y]^T$ is a 2D coordinate system that has the 3D points in the camera coordinate system projected to a 2D plane. The camera point $P_C = (X_C \quad 0 \quad Z_C)$ projects to point $p = (x \quad y)$ onto the image plan.
From similar triangle:

$$\frac{x}{f} = \frac{X_C}{Z_C} => x = \frac{fX_C}{Z_C}$$
$$\frac{y}{f} = \frac{Y_C}{Z_C} => y = \frac{fY_C}{Z_C}$$

Hence the following transformation matrix from the camera coordinate system to image coordinate system is:

$$\begin{bmatrix} x \\ y \\ Z_C \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix}$$

3.1.3. Image to Pixel Coordinate

$[u \quad v]$ represents the integer values by discretizing the points in the image coordinate system. Pixel coordinate of an image are discrete values withing the range that can be achieved by diving the image coordinate by pixel width and height parameter of the camera.

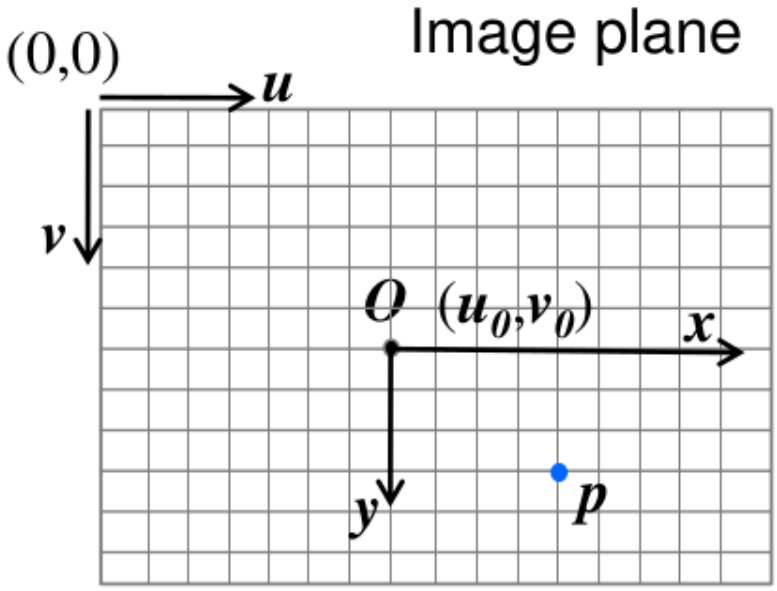


Figure 3.3: Image to Pixel Coordinate

To convert p, from the local image plane coordinates (x, y) to the pixel coordinate (u, v), we need to account for the pixel coordinate of the camera optical center $O = (u_0, v_0)$ and the scale factor k

$$\begin{aligned} u &= u_0 + kx = u_0 + k\frac{fX_C}{Z_C} \\ v &= v_0 + ky = v_0 + k\frac{fY_C}{Z_C} \end{aligned}$$

Expressed in matrix form and homogeneous coordinate

$$\begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix} = \begin{bmatrix} kf & 0 & u_0 \\ 0 & kf & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = K \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix}$$

So the perspective Projection matrix for the pinhole camera is

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[R|T] \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix}$$

Where

- λ is the depth parameter
- f is the focal length
- u_0 and v_0 are the pixel coordinates of the camera optical center
- K is the intrinsic parameter matrix
- $[R|T]$ is the extrinsic parameter matrix

3.2. Stereo Vision

Stereo Vision is the process of obtaining depth information from a pair of images coming from two cameras that look at the same scene from a different but known position. From a single camera, the ray can only be deducted on which each image point lies while with a stereo camera, can be solved for the intersection of the rays and recover the 3D structure [9].

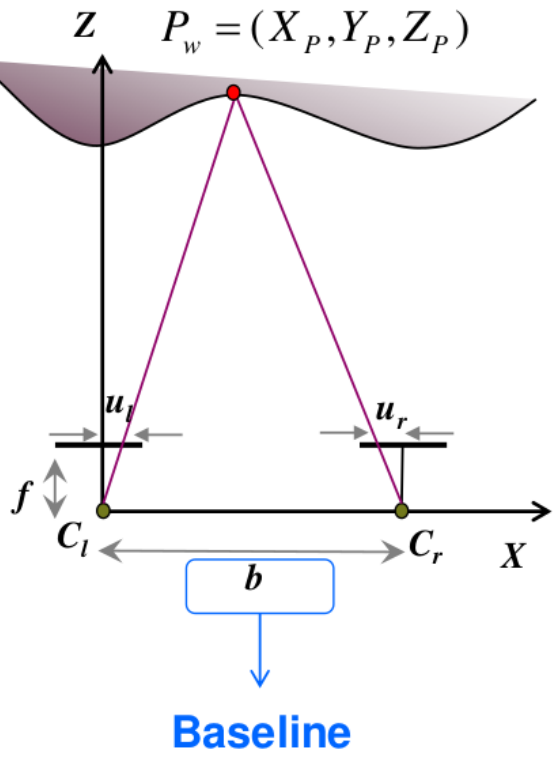


Figure 3.4: Geometry Stereo Vision

Assume that both cameras are identical and aligned with the x-axis.

- Left Camera Triangulation

$$\begin{aligned} u_l &= f_x \frac{X_P}{Z_P} + o_x \\ v_l &= f_y \frac{Y_P}{Z_P} + o_y \end{aligned}$$

- Right Camera Triangulation

$$\begin{aligned} u_r &= f_x \frac{X_P - b}{Z_P} + o_x \\ v_r &= f_y \frac{Y_P}{Z_P} + o_y \end{aligned}$$

From perspective projection:

$$\begin{aligned} (u_l, v_l) &= (f_x \frac{X_P}{Z_P} + o_x, f_y \frac{Y_P}{Z_P} + o_y) \\ (u_r, v_r) &= (f_x \frac{X_P - b}{Z_P} + o_x, f_y \frac{Y_P}{Z_P} + o_y) \end{aligned}$$

Solving for X_P, Y_P and Z_P:

$$\begin{aligned} X_P &= \frac{b(u_l - o_x)}{u_l - u_r} \\ Y_P &= \frac{b f_x (u_l - o_y)}{f(u_l - u_r)} \\ Z_P &= \frac{b f_x}{u_l - u_r} \end{aligned}$$

where

- b is the baseline between two cameras
- f_x and f_y are the focal length along x and y axis
- o_x and o_y are the camera optical center in x and y axis
- $u_l - u_r$ is called the disparity

The disparity is the apparent motion of objects between a pair of stereo images. Given a pair of stereo images, to compute the disparity map, we first match every pixel in the left image with its corresponding pixel in the right image. Then with this, distance can be computed for each pair of matching pixels. Finally, the disparity map is obtained by representing such distance value as an intensity image. As previously observed, the depth is inversely proportional to the disparity. If the geometric arrangement of the cameras is known, then the disparity map can be converted to the depth with the equation below

$$Z_P = \frac{b f_x}{u_l - u_r}$$

When the disparity is near zero, small disparity differences produce large depth differences. However, when disparity is large, small disparity differences do not change the depth significantly. Hence, stereo vision systems have high depth resolution only for objects relatively near the camera.

4. Trajectory Tracking of Quadcopters

Giving the 3D way-points in the space with the following parameter waypoints= $[0,0,2],[-3,3,2],[-3,-3,2],[3,3,2],[3,-3,2],[0,0,2]$.

4.1. Interpolate Trajectory without Cubic Spline

By interpolate the point between the way-points i and $i+1$, the following trajectory of the quadcopter is shown in Figure 4.1 .

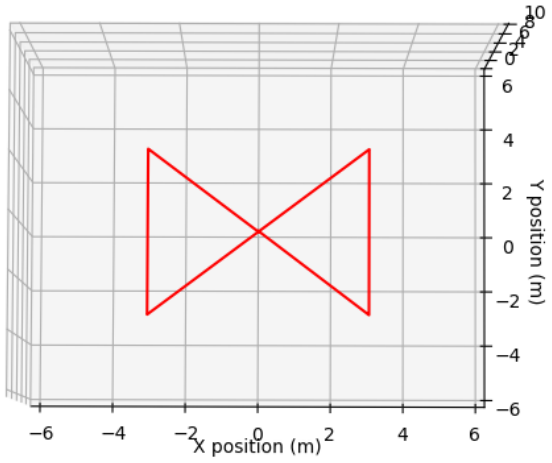


Figure 4.1: Interpolate without Cubic Spline

4.2. Cubic Spline Trajectory

[6] The Cubic Polynomial is written as

$$p(u) = au^3 + bu^2 + cu + d$$

where a, b, c and d are the vectors which are needed to find, and u is a scalar Three Cubic Polynomial can be written as:

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned}$$

In matrix notation can be written as:

$$[x(u) \quad y(u) \quad z(u)] = [u^3 \quad u^2 \quad u \quad 1] \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

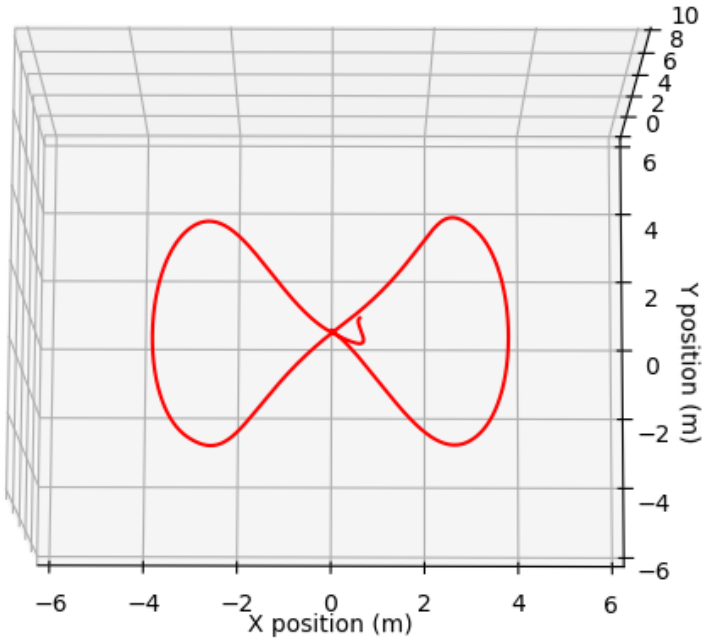


Figure 4.2: Cubic Spline Trajectory

Figure 4.2 shows the trajectory of quadcopter using cubic spline.

4.3. ROS Graph of Trajectory Tracking

Figure 4.3 shows the ROS Nodes and Topics of the trajectory tracking of the quadcopters using MPC simulates with Airsim Simulator.

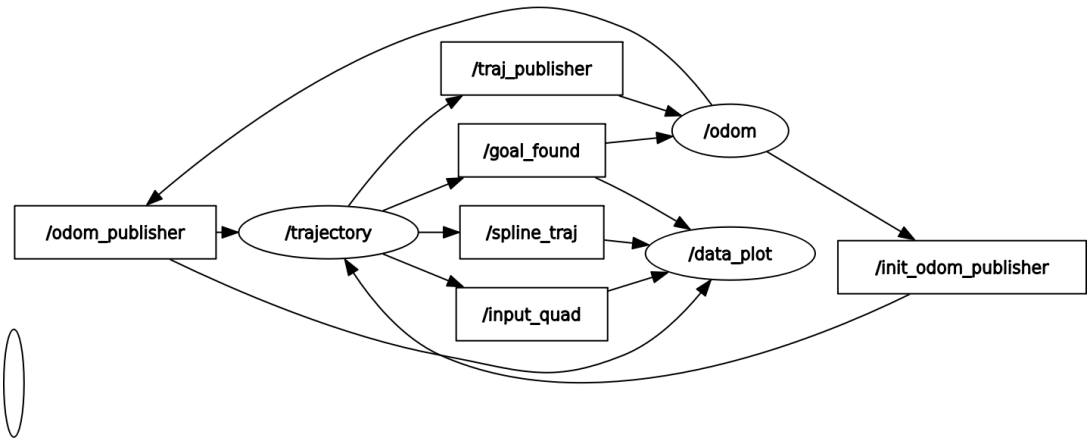


Figure 4.3: ROS graph of Trajectory Tracking

The Odom node publishes the odometry_publisher topic which is the position and orientation of the quadcopter and then the trajectory node subscribes to this topic. With the Trajectory input and convert to the spline trajectory then this spline trajectory

will use as the reference for tracking. Then the trajectory node is the control part which is using the MPC to track the spline trajectory. This node will publish the goal_found topic to check if it reaches the target or not and also publish the traj_publisher topic for the quadcopters in the Airsim simulator. It also publishes the input_quad and spline_traj to the data_plot node to view the graph of the input and trajectory of the quadcopters as shown in Figure 4.4

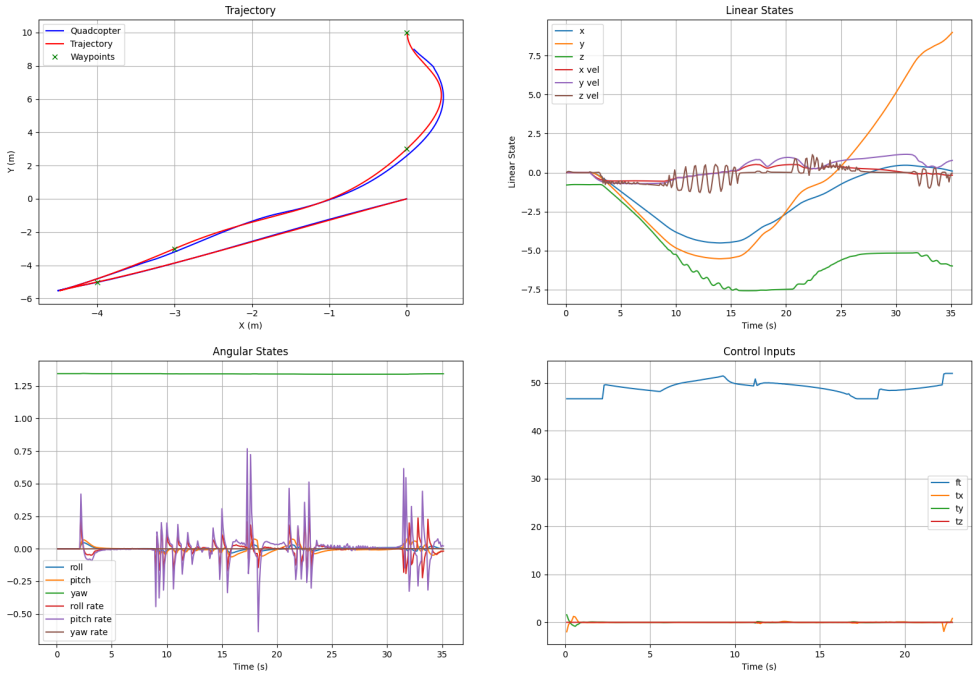


Figure 4.4: Quadcopter Graph

4.4. ROS Visualization

Rviz is the ROS visualization tool which is used to view the behavior of the Quadcopter where the position, orientation, TF and also the trajectory of the robot can be seen. The trajectory of the quadcopters is shown in Figure 4.5.

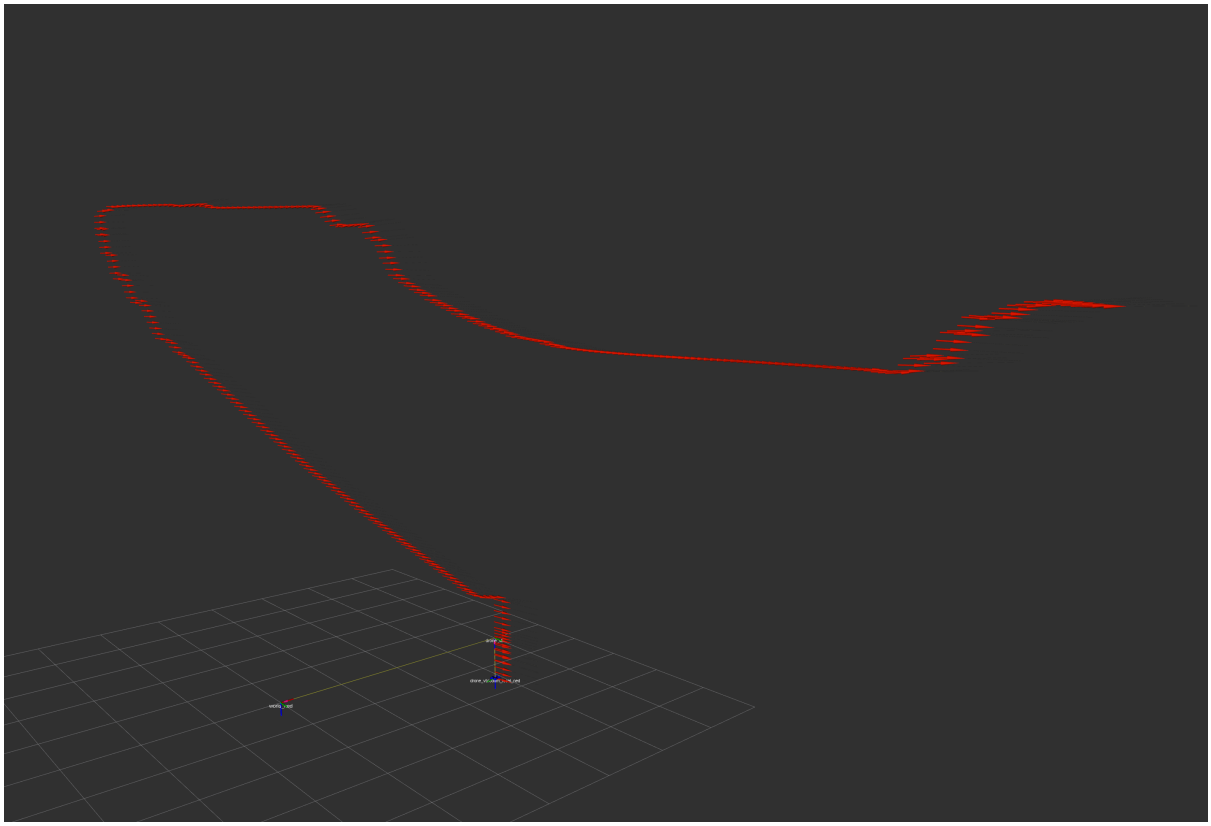


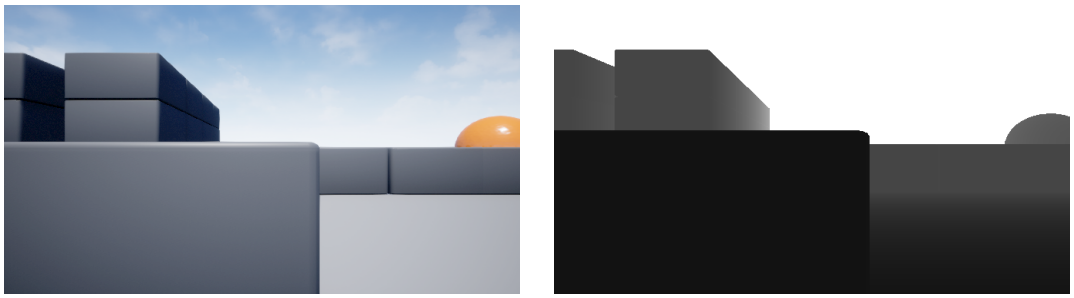
Figure 4.5: ROS Visualization of Trajectory Tracking of Quadcopter

5. Mapping of an Environment

5.1. Airsim Data

5.1.1. Airsim Depth Ground Truth

Airsim ROS provide the depth map which is shown below compared to the scene of the rgb image:



(a) RGB Image **(b)** Depth Map

Figure 5.1: RGB image and Depth Map

5.1.2. Point Cloud Data

With the depth map, it can convert this depth into point cloud data. The transformation of depth map into point cloud is actually the transformation of coordinate system. The formula is shown below:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = D \begin{bmatrix} \frac{1}{f_x} & 0 & 0 \\ 0 & \frac{1}{f_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

The depth_image_proc/point_cloud_xyzrgb nodelet in ROS is used to convert the depth map into point cloud data. This nodelet needs the information ROS topic of the RGB image, depth registered and the camera information to publish the information of the point cloud data. With the RGB image and depth map, The image which is shown in Figure 5.1 is used.

For the camera information, the data which are needed to publish are shown in the table below:

- K is the intrinsic camera matrix for the raw image which

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

- R is the Rotation matrix aligning the camera coordinate system to the ideal stereo image

- P is the projection camera matrix

$$P = \begin{bmatrix} f_x & 0 & c_x & T_x \\ 0 & f_y & c_y & T_y \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

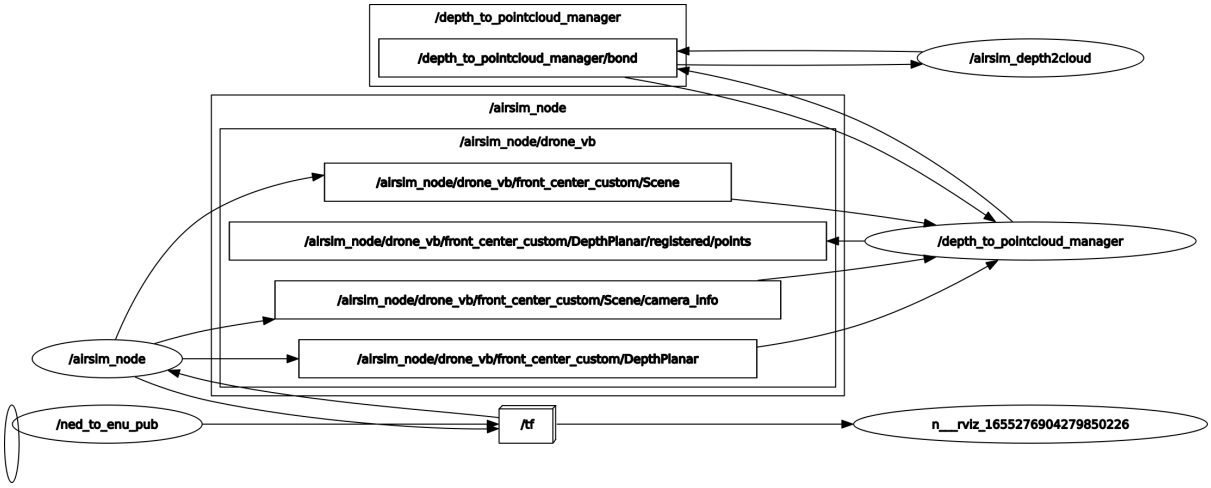


Figure 5.2: Rosgraph of Convert Depth to Point Cloud Data

Figure 5.2 shows the graph of converting the depth into point cloud data. The airsim_node publishes the topics of the scene, camera information, and the depth image. Then the depth_to_pointcloud_manager subscribes to these topics and publishes the topic of the point cloud data with the help of the depth_image_proc. The result is shown in ROS visualization as shown in Figure 5.3 .

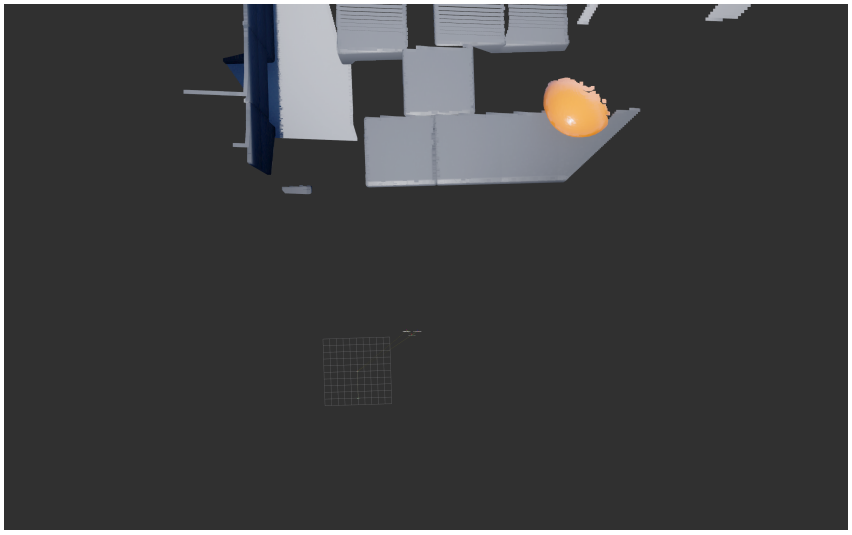


Figure 5.3: Ground Truth Point Cloud Data

5.1.3. Octomap

Octomap server builds and distributes volumetric 3D occupancy maps as OctoMap binary stream and in various ROS-compatible formats for obstacle avoidance and

visualization. The map can be a static Octomap .bt file or can be incrementally built from incoming range data. This Octomap needs the information of 3D point cloud data and also provides a tf transform between the sensor data and the static map frame. Then it will publish the information of:

- **octomap_full**: is the completed maximum-likelihood occupancy map as compact Octomap binary stream, encoding free and occupied space.
- **occupied_cells_vis_array**: is a marker for visualization in Ros visualization.
- **octomap_point_cloud_centers**: is the centers of all occupied vovels as point cloud, useful for visualization.
- **map**: downprojected 2D occupancy map from 3D map.

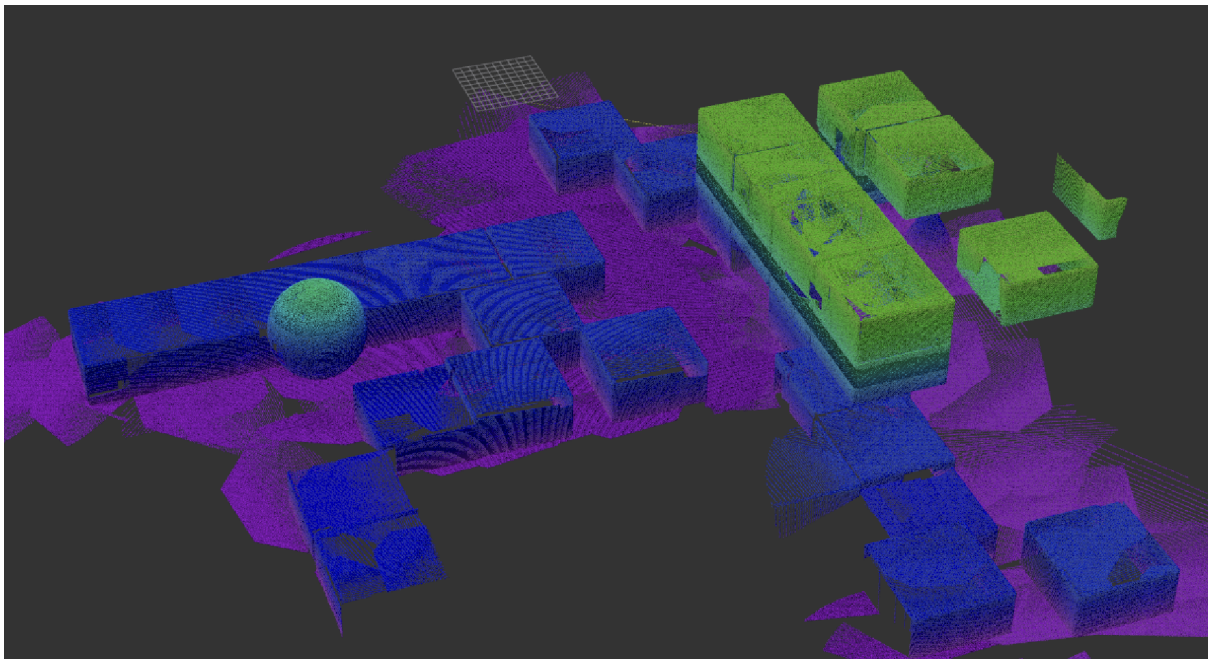


Figure 5.4: 3D mapping

5.2. Intel Real Sense

Figure 5.5: Intel Realsense D435

Intel Realsense D435 is a stereo solution, offering quality depth for a variety of applications. It is wide field of view is perfect for application such as robotics with a range up to 10m. Intel Realsense provides with the two gray scale cameras, a RGB Camera, and also an IR projector. In this case, two grayscale cameras are used to get the depth image.

5.2.1. Gray Scale Image

(a) left image **(b)** right image

Figure 5.6: Gray Scale Images

The two grayscale images and convert to RGB images by duplicating the channels. Then two images which are shown in Figure 5.6 are received. Using the model which is trained in Pytorch, the disparity is produced.

Then this disparity image is converted to depth with the formula below:

$$depth = \frac{baseline \times focal\_lenght}{disparity}$$

By

- baseline is distance between the left and right camera equation to 50mm

- focal length is number of pixel depend on the resolution of the image. Since the 640 resolution is used then the focal length of the image is 338px

so:

$$depth = -\frac{0.05 \times 338}{disparity}$$

5.2.2. ROS Node

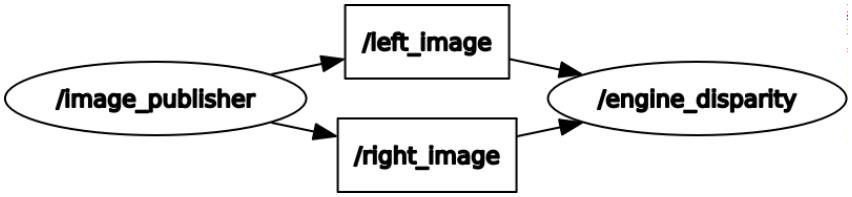
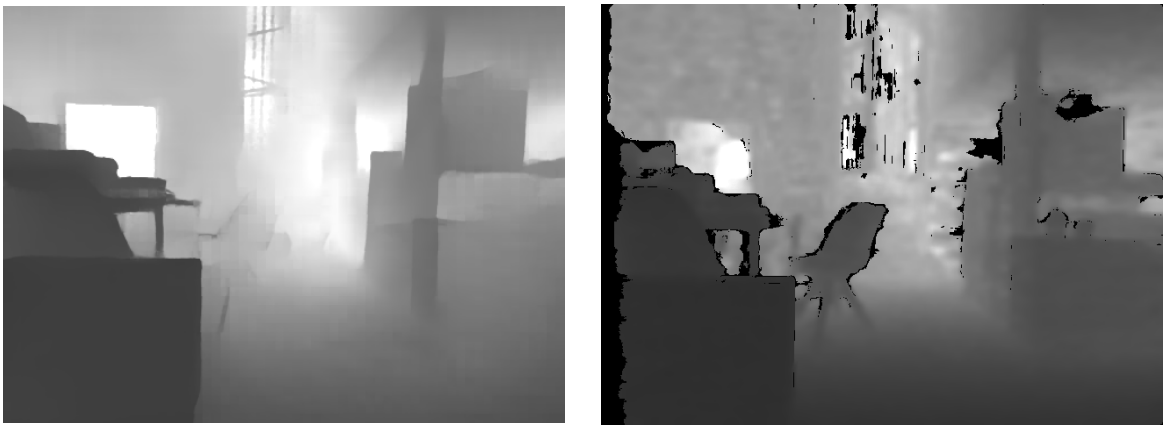


Figure 5.7: Disparity Image ROS Graph

Figure 5.7 shows the image_publisher node that publishes the two grayscale image topics which receive from the Realsense camera then the engine Disparity will subscribe these two topics then load in the engine with return the output of disparity then the disparity can be converted to depth as shown in Figure 5.8



(a) Depth from the Engine **(b)** Depth from the Realsense D435

Figure 5.8: Depth Image

5.2.3. Navigation Area

The navigation area engine is the model that separates the obstacle and the navigation area. If the output is bigger than 1 then it is the navigation area and if is lower than 1, it is the obstacle. With this, the depth image can be combined with the navigation area to get an accurate map.

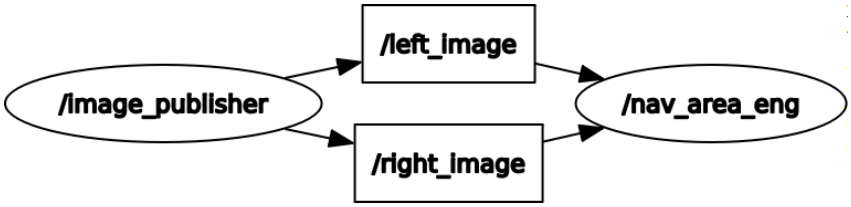


Figure 5.9: Navigation Area ROS Graph

Figure 5.9 shows another node call engine_disparity that subscribes one of the image topic from the image_publisher node and convert it to rgb image. Then this image is loaded in the engine and produce the output as shown in the Figure

Figure 5.10: Navigation Area Image

5.2.4. Depth and Navigation Area

In this scenario, the depth and navigation area are combined to get the depth image that separate between the obstacle and the navigation area.

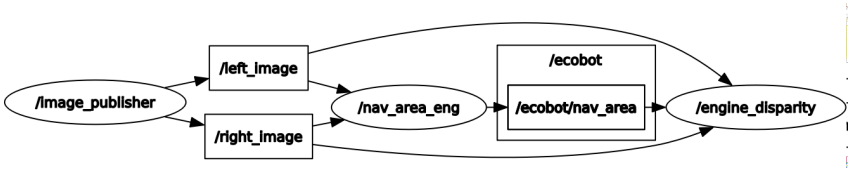


Figure 5.11: Depth and Navigation Area ROS Graph

The ROS graph in Figure 5.11 is shown the depth and navigation area node in which the image publisher publish the two grayscale image topic both left and right and the nav_area_eng node subscribe to one of these topics and then it will publish the topic of the navigation area call /ecobot/nav_area. Then the engine_disparity node will subscribe to them and output the depth image which is shown in Figure 5.12

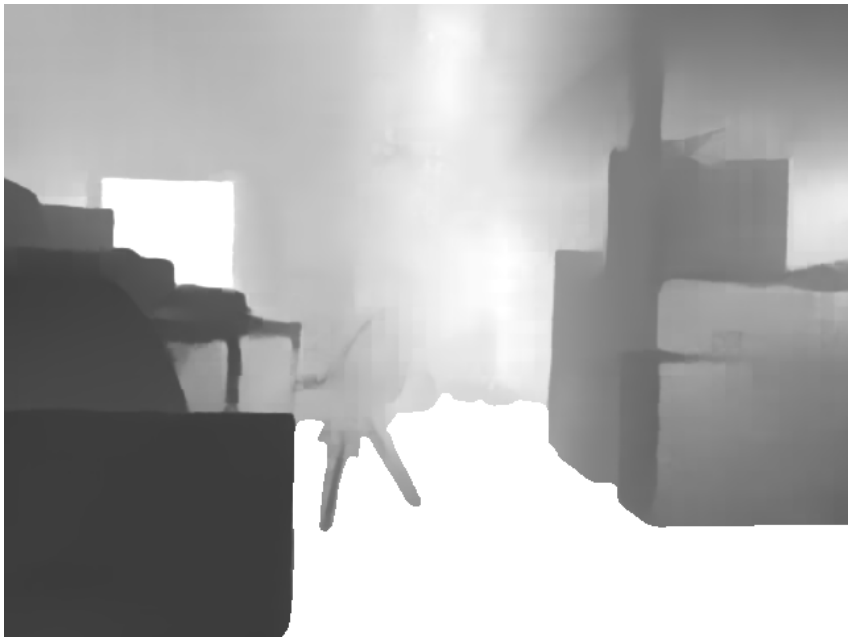


Figure 5.12: Depth and Navigation Area image

5.2.5. Octomap with Realsense D435

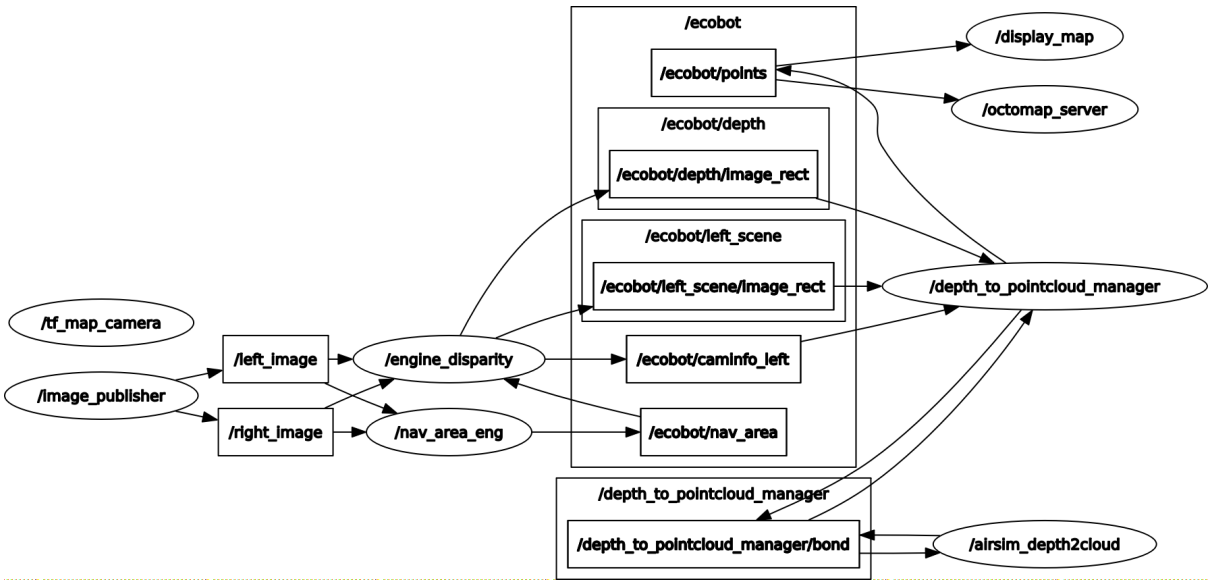


Figure 5.13: ROS Graph of 3D mapping using Realsense D435

Figure 5.13 shows the graph of converting the image from the realsense D435 into the depth image and the navigation area. Then this depth image is converted into point cloud data. After that Octomap is used to convert the point cloud into a 3D map and also the 2D occupancy grid map of the environment as shown in Figure 5.14 [5].

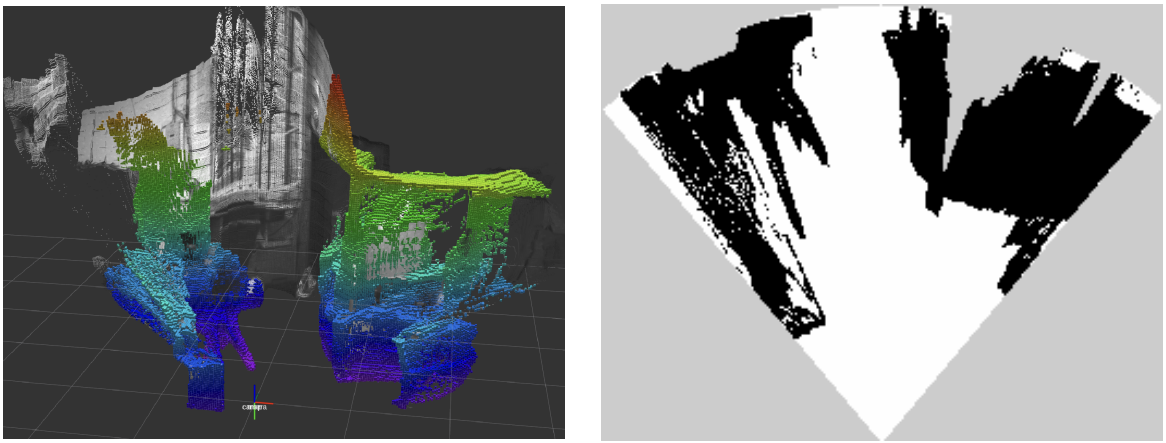


Figure 5.14: 3D and 2D grid map

6. Navigation of a Mobile Robot

6.1. Occupancy Grid Map

To self-localize with no error, the robot needs an occupancy grid map. An occupancy grid map stores data on free and occupied areas of the robot's environment. With the depth image from the camera and convert it to a point cloud. Then using an Octomap server, the occupancy grid map is received as shown in Figure 6.1 .

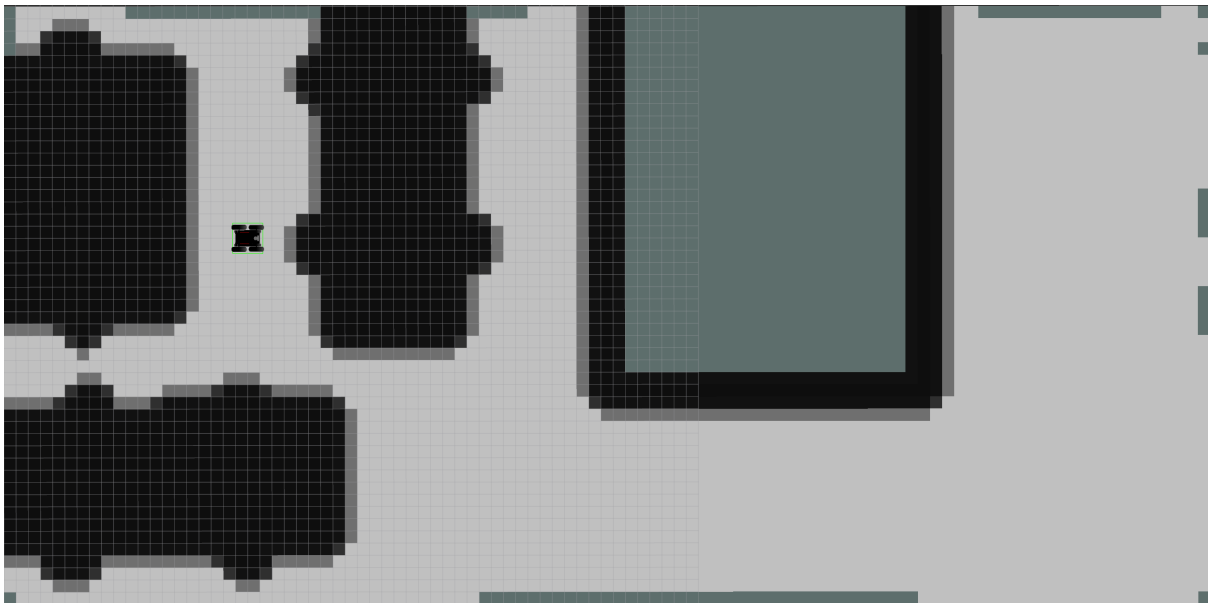


Figure 6.1: Occupancy Grid Map

The black area on the map is detected as the obstacle while the free space is the white area. Additionally, the robot's start position where the robot is situated is known as the goal location where the robot has to go is also given. From any given grid cell, the robot can only move to an adjacent free grid cell and cannot go outside of the map boundary.

6.2. Dijkstra Algorithm

Dijkstra is that it always keeps track of the shortest distance from the start node to each cell[16]. It refers to value as the cost of a node and displays it on each grid cell corner. Before the search process starts, the starting point gets a cost of 0 as its distance to itself is 0. Later, as the algorithm progresses, these values will be updated to the actual shortest distance from the start node [12].

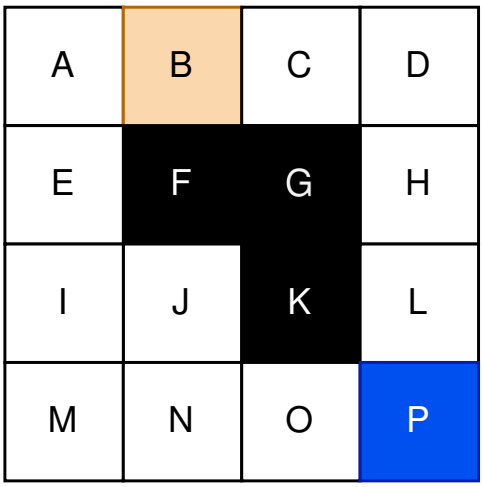


Figure 6.2: 4×4 Grid Map

Figure 6.2 shows the 4×4 grid map. To find the path from the starting position to the goal position, node B which is marked with an orange outline is denoted as the starting position while node P with the blue mark is the goal position.

The iteration step of Dijkstra algorithm is defined by:

- Pick a current node: The iterative search process starts by picking the node inside the open list with the lowest cost. This node is called the current node. At this moment open list only contains node B.
- Neighbors of the current node: The neighbors of the current node can be identified by arrows pointing from the current node toward the neighbors grid cells that are not an obstacle.

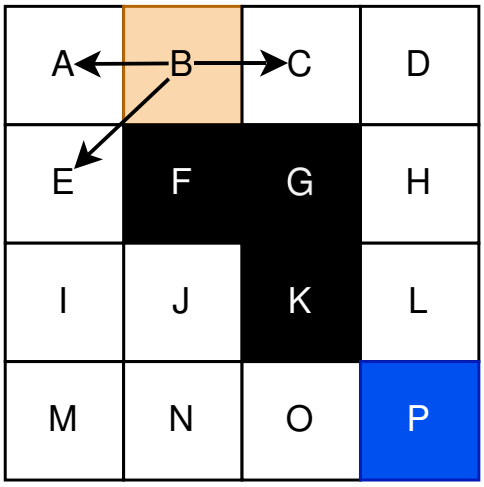


Figure 6.3: Neighbors of the Current Node

Figure 6.3 shows the neighbors of the current node B which are the A, C, and E nodes. F and G nodes are the obstacle, so node B is not pointing to these nodes. In this case, the orthogonal cost from node B to node A and C is 0.2, while the diagonal cost from node B to node E is 0.28 .

- Update travel distance values, store parent node Once all neighbors nodes are identified in the free space and update their cost. Then, the parent node is set of each neighbor. The parents node is the node from the update of the cost. After done considering all neighbors of B, the node B is marked as visited. Visited node will be kept in a separate list called closed list.

After this iteration, the new cycle is repeated until reaching the target. Once the target is reached, the shorted route is extracted by every single node's parent node until reaching the start node. Once the backtracking is completed, the newly created list will contain the shortest path as a sequence of grid cells to visit, but in the wrong order.

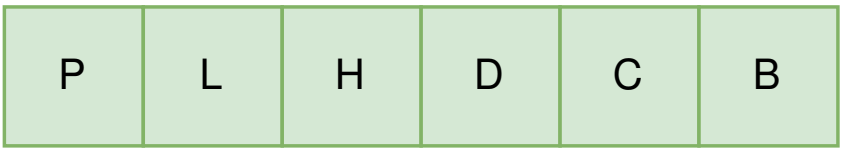


Figure 6.4: Wrong order path cell

To have the path from the start node to the end node, the reverse is needed

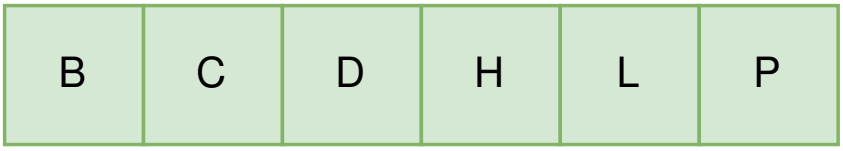


Figure 6.5: Order path cell

So the short path of the robot using Dijkstra algorithm is shown in Figure 6.6

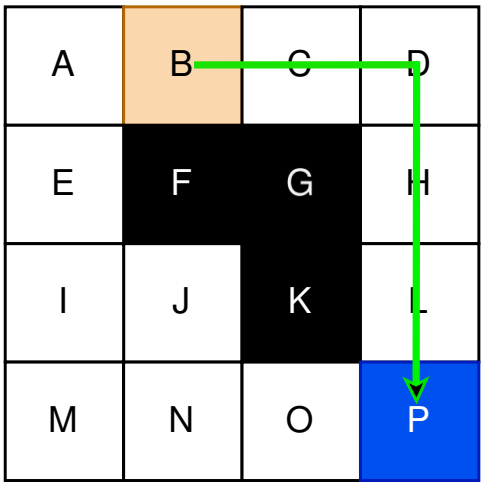


Figure 6.6: Path of the Robot using Dijkstra Algorithm

6.3. ROS with Dijkstra Algorithm

To simulate the robot in a gazebo environment using the Dijkstra algorithm, the planner is needed to connect the path to the robot. In this simulation. the dwa_local_planner package is used to provide a controller that drives a mobile base

in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine velocities to send to the robot.

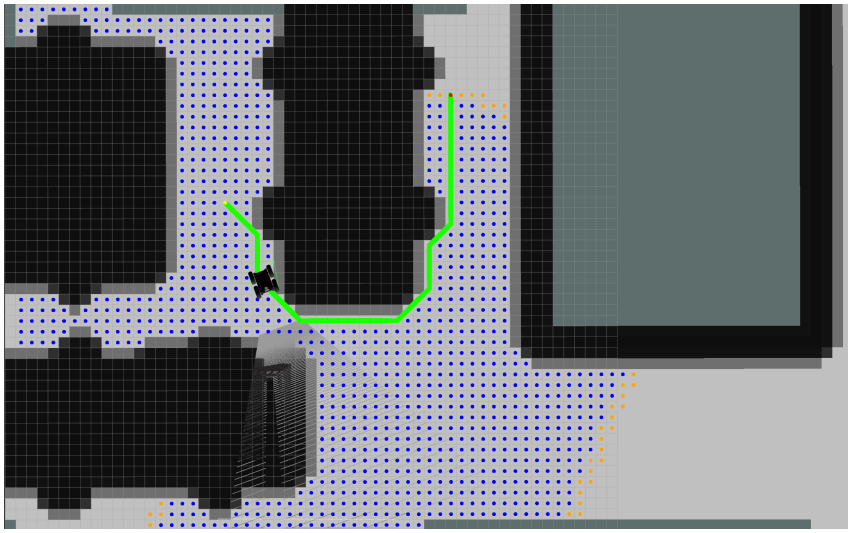


Figure 6.7: Navigation of a Mobile Robot

Figure 6.7 shows the navigation of the robot using the Dijkstra algorithm with the help of the DWA planner ROS. The yellow dot is the starting position of the robot and the red dot is the target position of the robot while the blue dot node uses the Dijkstra algorithm to find the short path from the starting position to the target position while avoiding the obstacle.

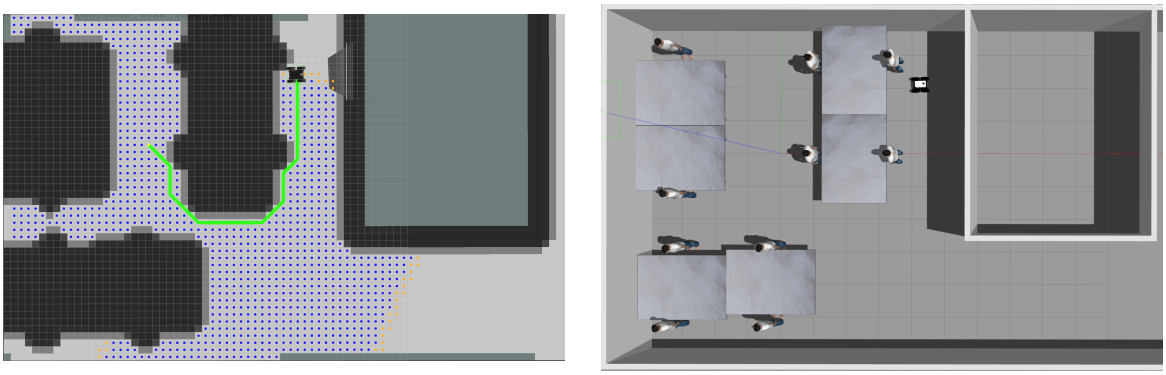


Figure 6.8: Target Position

Figure 6.8 shows the robot reaching the target position using the Dijkstra algorithm with the help of the DWA planner ROS. The left image is the ROS visualization while the right is the gazebo environment showing the robot reaching the target.

7. Conclusion and Future Work

In this project, a mathematical model of quadcopter dynamics is derived, using Newton's and Euler's laws. Then a linearized version of the model is obtained and Linear Model Predictive Control is used to track the trajectory of the quadcopter. With the depth image of an Airsim Environment, the 2D and 3D map can be achieved by converting the depth image into point cloud data, and then Octomap will subscribe this information to construct the map. Realsense D435 provides two grayscale images. Then the disparity is obtained by matching every pixel in the left image with its corresponding pixel in the right image. The depth image is inversely proportional to the disparity. Finally, the Dijkstra algorithm is used with the help of the Dynamic Window Approach to navigate using Gazebo Simulator from the start position to the goal position while avoiding the obstacle. It keeps tracking a short distance from the start node to each individual cell. A future project will be applying the Model Predictive Control with the quadcopter dynamic to the real-life quadcopter, also for the mobile robot, the Dijkstra algorithm with the Dynamic Window Approach which is the planner in ROS will be implemented on a Rover Mobile Robot to test the navigation with the obstacle avoidance.

References

[1] Younes Al Younes and Martin Barczyk. Nonlinear model predictive horizon for optimal trajectory generation. *Robotics*, 10(3):90, 2021.

[2] Gayan Brahmanage and Henry Leung. Outdoor rgb-d mapping using intel-realsense. In *2019 IEEE SENSORS*, pages 1–4. IEEE, 2019.

[3] Trevor W Caplinger. *Path Planning and Control of an Autonomous Quadrotor Testbed in a Cluttered Environment*. West Virginia University, 2015.

[4] Long Cheng, Cheng-dong Wu, and Yun-zhou Zhang. Indoor robot localization based on wireless sensor networks. *IEEE Transactions on Consumer Electronics - IEEE TRANS CONSUM ELECTRON*, 57:1099–1104, 08 2011.

[5] Pavel Chmelar, Ladislav Beran, and Lubos Rejfek. The depth map construction from a 3d point cloud. In *MATEC Web of Conferences*, volume 75, page 03005. EDP Sciences, 2016.

[6] Magnus Egerstedt and Clyde F Martin. Optimal trajectory planning and smoothing splines. *Automatica*, 37(7):1057–1064, 2001.

[7] Eduardo Gamaliel Hernández-Martínez, Guillermo Fernandez-Anaya, Enrique D Ferreira, José-Job Flores-Godoy, and Alexandro Lopez-Gonzalez. Trajectory tracking of a quadcopter uav with optimal translational control. *IFAC-PapersOnLine*, 48(19):226–231, 2015.

[8] Yousof Koohmaskan and Behzad Samadi. Convex optimization in model predictive control. *Department of Electrical Engineering*, 2010.

[9] A Lipnickas and A Knyš. A stereovision system for 3-d perception. *Elektronika ir Elektrotechnika*, 91(3):99–102, 2009.

[10] Marta Marques, Bruno J Guerreiro, Rita Cunha, and Carlos Silvestre. Trajectory planning and control for drone replacement for multidrone cinematography. *IFAC-PapersOnLine*, 52(12):334–339, 2019.

[11] Fang Nan, Sihao Sun, Philipp Foehn, and Davide Scaramuzza. Nonlinear mpc for quadrotor fault-tolerant control. *IEEE Robotics and Automation Letters*, 7(2):5047–5054, 2022.

[12] Masato Noto and Hiroaki Sato. A method for the shortest path search by extended dijkstra algorithm. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, volume 3, pages 2316–2320. IEEE, 2000.

[13] Mohamed Owis, Seif El-Bouhy, and Ayman El-Badawy. Quadrotor trajectory tracking control using non-linear model predictive control with ros implementation. In *2019 7th International Conference on Control, Mechatronics and Automation (ICCMA)*, pages 243–247. IEEE, 2019.

[14] Gabriele Perozzi. A toolbox for quadrotors: from aerodynamic science to control theory. 2018.

[15] Francesco Sabatino. Quadrotor control: modeling, nonlinearcontrol design, and simulation, 2015.

[16] Huijuan Wang, Yuan Yu, and Quanbo Yuan. Application of dijkstra algorithm in robot path-planning. In *2011 second international conference on mechanic automation and control engineering*, pages 1067–1069. IEEE, 2011.

[17] Pengcheng Wang, Zhihong Man, Zhenwei Cao, Jinchuan Zheng, and Yong Zhao. Dynamics modelling and linear control of quadcopter. In *2016 International Conference on Advanced Mechatronic Systems (ICAMechS)*, pages 498–503. IEEE, 2016.

[18] Matt Young. The pinhole camera: Imaging without lenses or mirrors. *The Physics Teacher*, 27(9):648–655, 1989.

[19] Kaixiang Zhang, Jian Chen, Yang Li, and Xinfang Zhang. Visual tracking and depth estimation of mobile robots without desired velocity information. *IEEE Transactions on Cybernetics*, 50(1):361–373, 2018.

[20] Zhenghong Zhang, Mingkang Xiong, and Huilin Xiong. Monocular depth estimation for uav obstacle avoidance. In *2019 4th International Conference on Cloud Computing and Internet of Things (CCIOT)*, pages 43–47. IEEE, 2019.

Appendix A Quadcopter Model

```
class Quadcopter:
  def __init__(self, **init_kwargs):
    self.Ix = 1.
    self.Iy = 1.
    self.Iz = 1.5
    self.g = 9.8
    self.m = 5.
    self.N = 20
    self.DT = 0.1
    self.u0 = 10.
    self.reach_goal_thresh = 1.5
    self.nsim = 400  # Number of simulation time steps
    self.idx_incr = 18 # Amount of 'lookahead' for trajectory

    #  States
    self.x_dot = init_kwargs['x_dot'] if 'x_dot' in
       init_kwargs.keys() else 0.
    self.y_dot = init_kwargs['y_dot'] if 'y_dot' in
       init_kwargs.keys() else 0.
    self.z_dot = init_kwargs['z_dot'] if 'z_dot' in
       init_kwargs.keys() else 0.
    self.x = init_kwargs['x'] if 'x' in init_kwargs.keys()
       else 0.
    self.y = init_kwargs['y'] if 'y' in init_kwargs.keys()
       else 0.
    self.z = init_kwargs['z'] if 'z' in init_kwargs.keys()
       else 0.

    self.roll_dot = init_kwargs['roll_dot'] if 'roll_dot' in
       init_kwargs.keys() else 0.
    self.pitch_dot = init_kwargs['pitch_dot'] if 'pitch_dot'
       in init_kwargs.keys() else 0.
    self.yaw_dot = init_kwargs['yaw_dot'] if 'yaw_dot' in
       init_kwargs.keys() else 0.
    self.roll = init_kwargs['roll'] if 'roll' in init_kwargs.
       keys() else 0.
    self.pitch = init_kwargs['pitch'] if 'pitch' in
       init_kwargs.keys() else 0.
    self.yaw = init_kwargs['yaw'] if 'yaw' in init_kwargs.keys
       () else 0.

    self.A_zoh = np.eye(12)
    self.B_zoh = np.zeros((12, 4))
```

```
    self.states = np.array([self.roll, self.pitch, self.yaw,
       self.roll_dot, self.pitch_dot, self.yaw_dot,
    self.x_dot, self.y_dot, self.z_dot, self.x, self.y, self.z
       ]).T

  @property
  def A(self):
    # Linear state transition matrix
    A = np.zeros((12, 12))
    A[0, 3] = 1.
    A[1, 4] = 1.
    A[2, 5] = 1.
    A[6, 1] = -self.g
    A[7, 0] = self.g
    A[9, 6] = 1.
    A[10, 7] = 1.
    A[11, 8] = 1.
    return A

  @property
  def B(self):
    # Control matrix
    B = np.zeros((12, 4))
    B[3, 1] = 1/self.Ix
    B[4, 2] = 1/self.Iy
    B[5, 3] = 1/self.Iz
    B[8, 0] = 1/self.m
    return B

  @property
  def C(self):
    C = np.eye(12)
    return C

  @property
  def D(self):
    D = np.zeros((12, 4))
    return D

  @property
    def Q(self):
    # State cost
    Q = np.eye(12)
    Q[8, 8] = 5.  # z vel
    Q[9, 9] = 10.  # x pos
    Q[10, 10] = 10.  # y pos
    Q[11, 11] = 100.  # z pos
    return Q
```

```
@property
  def R(self):
  # Actuator cost
  R = np.eye(4)*.001
  return R

def zoh(self):
  # Convert continuous time dynamics into discrete time
  sys = control.StateSpace(self.A, self.B, self.C, self.D)
  sys_discrete = control.c2d(sys, self.DT, method='zoh')

  self.A_zoh = np.array(sys_discrete.A)
  self.B_zoh = np.array(sys_discrete.B)

  def set_horizontal_length(self,N):
  self.N=N
  return self.N

def run_mpc(self, rx,x,u,x_init):
  cost = 0.
  umin =np.array([7.7 , 7.7 , 7.7 , 7.7])-self.u0
  umax =np.array([13. , 13. , 13. , 13.])-self.u0
  INF = np.inf
  xmin = np.array([-0.2, -0.2, -2*np.pi, -.25, -.25, -.25,
     -INF,  -INF,  -INF, -INF, -INF, -INF])
  xmax = np.array([0.2,  0.2,   2*np.pi,  .25, .25,  .25,
     INF,   INF,   INF,   INF,  INF, INF])

  constr = [x[:, 0] == x_init]
  for t in range(self.N):
  cost += cp.quad_form(rx - x[:, t], self.Q) + cp.quad_form(
     u[:, t], self.R)  # Linear Quadratic cost
  constr += [xmin <= x[:, t], x[:, t] <= xmax]  # State
     constraints
  constr += [umin <= u[:, t], u[:, t] <= umax]
  constr += [x[:, t + 1] == self.A_zoh * x[:, t] + self.
     B_zoh * u[:, t]]

  cost += cp.quad_form(x[:, self.N] - rx, self.Q)  # End of
     trajectory error cost
  problem = cp.Problem(cp.Minimize(cost), constr)
  return problem



def update_states(self, ft, tx, ty, tz):
  roll_ddot =   tx / self.Ix
```

```
    pitch_ddot =  ty / self.Iy
    yaw_ddot =    tz / self.Iz
    x_ddot = -self.g*self.pitch
    y_ddot = self.g*self.roll
    z_ddot = -1*(self.g - (ft/self.m))

    self.roll_dot += roll_ddot*self.DT
    self.roll += self.roll_dot*self.DT
    self.pitch_dot += pitch_ddot * self.DT
    self.pitch += self.pitch_dot * self.DT
    self.yaw_dot += yaw_ddot * self.DT
    self.yaw += self.yaw_dot * self.DT

    self.x_dot += x_ddot * self.DT
    self.x += self.x_dot * self.DT
    self.y_dot += y_ddot * self.DT
    self.y += self.y_dot * self.DT
    self.z_dot += z_ddot * self.DT
    self.z += self.z_dot * self.DT

    self.states = np.array([self.roll, self.pitch, self.yaw,
       self.roll_dot, self.pitch_dot, self.yaw_dot,
    self.x_dot, self.y_dot, self.z_dot, self.x, self.y, self.z
       ]).T



  def done(self,curr_pos, final_waypt):
    if np.linalg.norm(curr_pos - final_waypt) <= self.
       reach_goal_thresh:
      return True
    return False

  def __call__(self, ft=0., tx=0., ty=0., tz=0.):
    hover = self.m*9.8
    self.update_states(hover+ft, tx, ty, tz)
```

Appendix B MPC Controller

```
class mpc_controllers(Quadcopter):
  def __init__(self,waypoints):
    super(mpc_controllers,self).__init__()
    rospy.init_node("trajectory_publisher")

    self.odom_data_sub = rospy.Subscriber("odom_publisher",
       odometry , self.odom_data_callback)
    self.odom_ned_sub = rospy.Subscriber("/airsim_node/
       drone_vb/odom_local_ned", Odometry , self.
       odometry_callback)

    self.traj_pub = rospy.Publisher("traj_publisher" ,
       quad_traj, queue_size=10)
    self.goal_pub = rospy.Publisher("goal_found" , goal_found,
        queue_size=5)
    self.input_pub = rospy.Publisher("input_quad" , input_msg,
        queue_size=10)
    self.spline_data_pub = rospy.Publisher("spline_traj" ,
       spline_traj, queue_size=10)

    self.odometry = Odometry()
    self.splines = spline_traj()
    self.goal_found = goal_found()
    self.goal_found.goal_found = False
    self.goal_reach = False
    self.waypoints = waypoints

  def odometry_callback(self,odom):
    self.odometry = odom.pose.pose.position

  def odom_data_callback(self,init_odom):

    init_pos=init_odom.pose
    init_vel =init_odom.vel
    init_ang =init_odom.ang
    init_angRate=init_odom.angRate

    init_dict = {'roll':init_ang[0], 'pitch':init_ang[1], 'yaw
       ':init_ang[2],
      'roll_dot': init_angRate[0], 'pitch_dot': init_angRate
         [1], 'yaw_dot': init_angRate[2],
      'x_dot': init_vel[0],'y_dot': init_vel[1], 'z_dot':
         init_vel[2],
      'x': init_pos[0], 'y': init_pos[1], 'z': init_pos[2]}
```

```
init_pose = np.array([init_pos[0],init_pos[1],init_pos
   [2]])

quad = Quadcopter(**init_dict)
quad.zoh()

spline_gen = SplineGenerator()
spline_data = spline_gen.create_splines(self.waypoints,
   init_pose)

spline_x_data = spline_data[:, 0]
spline_y_data = spline_data[:, 1]
spline_z_data = spline_data[:, 2]

way_points=[]

for row in self.waypoints:
  for ele in row:
    way_points.append(ele)

self.splines.way_points = way_points
self.splines.x_data = spline_x_data
self.splines.y_data = spline_y_data
self.splines.z_data = spline_z_data
self.spline_data_pub.publish(self.splines)

  # Initial solver states (copy of quadcopter states)
x0 = np.array([init_dict['roll'], init_dict['pitch'],
   init_dict['yaw'], init_dict['roll_dot'],
init_dict['pitch_dot'], init_dict['yaw_dot'], init_dict['
   x_dot'], init_dict['y_dot'],
init_dict['x_dot'], init_dict['x'], init_dict['y'],
   init_dict['z']])

# Desired states to track
des_states = {'roll': 0., 'pitch': 0., 'yaw':init_ang[2],
   'roll_dot': 0. 'pitch_dot': 0., 'yaw_dot': 0., 'x_dot':
   0., 'y_dot': 0., 'z_dot': 0., 'x': self.waypoints[0][0],
    'y':self.waypoints[0][1], 'z':self.waypoints[0][2]}


idx = self.idx_incr  # Index for current spline point

[nx, nu] = quad.B.shape

# Convex optimization solver variables
x = cp.Variable((nx, self.N+1))
u = cp.Variable((nu, self.N))
```

```
x_init = cp.Parameter(nx)

for i in range(1,self.nsim+1):

  ref_x = spline_x_data[idx]
  ref_y = spline_y_data[idx]
  ref_z = spline_z_data[idx]

  idx += self.idx_incr

  # If spline index tries to go past last waypoint
  if idx >= len(spline_x_data):
    idx = len(spline_x_data) - 1

  # Update reference states
  xr = np.array([des_states['roll'], des_states['pitch'],
     des_states['yaw'], des_states['roll_dot'],
  des_states['pitch_dot'], des_states['yaw_dot'],
     des_states['x_dot'], des_states['y_dot'],des_states['
     x_dot'], ref_x, ref_y, ref_z])

  # Run optimization for N horizons
  prob = quad.run_mpc(xr,x,u,x_init)

  # Solve convex optimization problem
  x_init.value=x0
  prob.solve(solver=cp.OSQP, warm_start=True)
  x0 = quad.A_zoh.dot(x0) + quad.B_zoh.dot(u[:, 0].value)

  # Send only first calculated command to quadcopter, then
      run optimization again
  quad(u[0, 0].value, u[1, 0].value, u[2, 0].value, u[3,
     0].value)

  #input
  in_msg = input_msg()
  in_msg.thrust = u[0,0].value+(self.m*self.g)
  in_msg.x_torque = u[1,0].value
  in_msg.y_torque = u[2,0].value
  in_msg.z_torque = u[3,0].value
  self.input_pub.publish(in_msg)

  #output
  q = quad_traj()
  q.quad_x = quad.x
  q.quad_y = quad.y
  q.quad_z = quad.z
  self.traj_pub.publish(q)
```

```
      if quad.done(np.array([self.odometry.x, self.odometry.y,
         self.odometry.z]),np.array([self.waypoints[-1][0],self
         .waypoints[-1][1],self.waypoints[-1][2]])):
        print('GOAL REACHED')
        self.goal_reach = True
        self.goal_found.goal_found = self.goal_reach
        self.goal_pub.publish(self.goal_reach)
        break

    if not self.goal_reach:
    print("GOAL FAILED")
    self.goal_reach = False
    self.goal_found.goal_found = self.goal_reach
    self.goal_pub.publish(self.goal_reach)


if __name__=="__main__":

  if rospy.has_param("/waypoints"):
    data = rospy.get_param("/waypoints")
    waypoints = [float(item) for item in data]
    dimension = 3
    waypoints = np.asarray(waypoints).reshape((len(waypoints)
       //dimension),dimension)
  else:
    waypoints = np.array
       ([[-4,-5,-6],[-3,-3,-8],[0,3,-5],[0,10,-7]])

  mpc_controllers=mpc_controllers(waypoints)

  rate=rospy.Rate(10)
  while not rospy.is_shutdown():
    rate.sleep()
  try:
    rospy.spin()
  except KeyboardInterrupt:
    print("interrup mpc")
```

Appendix C Quadcopter ROS Launch File

```
<launch>
  <param name="robot_description" command="$(find␣xacro)/xacro
     ␣␣$(find␣quadcopter_ros)/urdf/drone_vb.urdf"/>

  <node name="joint_state_publisher_gui" pkg="
     joint_state_publisher_gui" type="joint_state_publisher_gui
     " />
  <node name="robot_state_publisher" pkg="
     robot_state_publisher" type="robot_state_publisher" />

  <!-- <include file="$(find␣airsim_ros_pkgs)/launch/
     airsim_node.launch"></include> -->
  <node name="odom" pkg="quadcopter_ros" type="
     quadcopter_odometry.py" output="screen"></node>
  <node name="trajectory" pkg="quadcopter_ros" type="
     quadcopter_mpc.py" output="screen"></node>
  <node name="data_plot" pkg="quadcopter_ros" type="
     quadcopter_dataPlot.py" output="screen"></node>

  <node type="rviz" name="quadcopter_rviz" pkg="rviz" args="-d
     ␣$(find␣quadcopter_ros)/rviz/default.rviz" />
</launch>
```