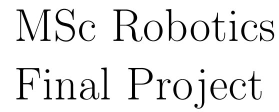


MSc Robotics
Final Project

Multi-UAV Real-Time Collaborative Aerial Mapping for 3D Reconstruction

Pragna Shastry

Supervisor: Prof. Dr. Ing. Francesco Nex
External Supervisor: Dr. Chiara Gabellieri

November, 2024

Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Research Objectives . . . 3
- 1.3 Outline . . . 3

2 Background **4**
- 2.1 Fundamentals . . . 4
  - 2.1.1 Transformation Matrix . . . 4
  - 2.1.2 Euler Angles and Quaternions . . . 6
  - 2.1.3 Camera Model and Matrices . . . 7
  - 2.1.4 Epipolar Geometry . . . 8
  - 2.1.5 Coordinate Systems Used in Aviation . . . 8
- 2.2 Visual SLAM Framework . . . 9
- 2.3 Evaluation Metrics for SLAM . . . 11

3 State Of The Art **13**
- 3.1 Overview of SLAM Methods . . . 13
- 3.2 Overview of Collaborative SLAM Methods . . . 15
- 3.3 CCM-SLAM Architecture . . . 17
- 3.4 3D Modeling of the Simulation Environment . . . 18

4 Methodology **22**
- 4.1 Building 3D Model of the Environment . . . 23
- 4.2 Building the Gazebo World . . . 25
- 4.3 Setting up the Drone Simulator . . . 27
- 4.4 Creating Datasets . . . 28
- 4.5 Comparing CCM-SLAM and COVINS . . . 32
- 4.6 Configuring CCM-SLAM for Simulated Data . . . 34
- 4.7 Workflow . . . 36
- 4.8 Evaluation . . . 37
- 4.9 Georeferencing using GNSS data . . . 38

5 Results and Discussion **40**
- 5.1 Varying Configuration Parameters . . . 40
- 5.2 Results of the UTwente Datasets . . . 41
- 5.3 Results of the Vatican Datasets . . . 44
- 5.4 3D Reconstruction Results from Point Clouds . . . 46

6 Conclusion and Future Work **52**

List of Figures

2.1 Transforming point **p** using transformation matrix **T**. [1] 4
2.2 Comparing the properties of transformation matrices [1] 5
2.3 Rotation matrices about the z,y,z axes [2] 6
2.4 Quaternion rotation matrix [2] 6
2.5 Pinhole camera model [1] 7
2.6 Epipolar geometry between two camera images [1] 8
2.7 Coordinate systems in aviation 9
2.8 The classic visual SLAM framework [1] 10
2.9 Evaluation metrics for SLAM algorithms [3] 12

3.1 ORB-SLAM system overview [4] 15
3.2 CCM-SLAM system architecture [5] 17
3.3 Examples of Gazebo world environments 20
3.4 Examples of cityscape models in Sketchfab 20
3.5 Two approaches for the simulation environment 21

4.1 Overview of the methodology: 1. The data acquisition process 2. The SLAM algorithm process 22
4.2 Installing the Blosm add-on 23
4.3 Blosm tool panel 23
4.4 Selecting rectangular areas in OpenStreetMap 24
4.5 Imported models in Blender 24
4.6 Steps to export model from Blender 25
4.7 Simple folder structure for gazebo simulation 26
4.8 Imported worlds in Gazebo 26
4.9 Parameters of the downward facing camera 28
4.10 1. The flying drone with a downward-facing camera in the UTwente Gazebo world. 2. Planning the mission path in QGC and viewing its progress. 3. Visualizing the `/down_cam1/image_raw` topic as recorded by the drone. . . . 29
4.11 Visualization of the ROS nodes and topics during data acquisition 30
4.12 Data acquisition for the `ut_t8` dataset 32
4.13 The ground truth pose of the Ut_t7 and Ut_t8 datasets plotted together . 32
4.14 Trajectory plots and Point Clouds from running CCM-SLAM and COVINS-G on Euroc MAV MH-01, MH-02, MH-03 data 34
4.15 Point cloud densities and distribution of the CCM-SLAM (left) and COVINS (right) outputs 35
4.16 RVIZ visualization of collaborative SLAM on Vat-t7 and Vat-t8 datasets . . 38

5.1 Trajectory results from SLAM run on individual agents of UT dataset . . . 42
5.2 Trajectory results from collaborative SLAM run on UT dataset 42

5.3 Georeferenced trajectories from Ut-t7 and Ut-t8 running non-collaborative SLAM . . . 43
5.4 Georeferenced trajectories of Ut-t7 and Ut-t8 run on collaborative SLAM . 43
5.5 Ground truth data of Vat-t6, Vat-t7 and Vat-t8 . . . 44
5.6 Non-collaborative SLAM run on Vat-t6 dataset . . . 45
5.7 Non-collaborative SLAM run on Vat-t6, Vat-t7, Vat-t8 datasets . . . 45
5.8 Collaborative SLAM run on Vat-t6, Vat-t7, Vat-t8 datasets . . . 46
5.9 Ground truth point clouds from Blender . . . 47
5.10 Results from non-collaborative SLAM on Ut-t7 and Ut-t8 before and after georeferencing . . . 47
5.11 Results from collaborative SLAM on Ut-t7 and Ut-t8 before and after georeferencing . . . 48
5.12 Comparing maps from non-collaborative and collaborative SLAM against ground truth . . . 48
5.13 Comparing cloud to cloud distances of the maps in non-collaborative and collaborative scenarios with max. distance of 20 . . . 49
5.14 Maps obtained from non-collaborative (left) and collaborative (right) on Vat datasets . . . 49
5.15 Maps obtained from non-collaborative (left) and collaborative (right) aligned with their ground truths after georeferencing. . . . 50
5.16 Cloud to cloud distance for the non-collaborative (left) and collaborative (right) cases, with max. distance of 2. . . . 50
5.17 The merged maps from collaborative SLAM on UTwente and Vatican datasets aligned with their respective 3D models. . . . 51

List of Tables

4.1 Details of the datasets from the UTwente and Vatican models 31
4.2 Comparing performance of CCM-SLAM and COVINS-G with MH datasets 33

5.1 Comparing mapping parameters against RMSE 40
5.2 Comparing the RMSE values of Ut-t7 and Ut-t8 run with and without collaborative SLAM 44
5.3 Comparing the RMSE values of Ut-t7 and Ut-t8 run with and without collaborative SLAM 46

Abstract

UAVs are being increasingly used to perform Simultaneous Localization and Mapping (SLAM) tasks due to being compact, low cost, lightweight, and their ability to be used in complex environments. Multiple UAVs can be used to map the area rapidly and in real-time, especially in disaster areas. They can work together and share information to make the SLAM process more robust, accurate and scalable. However, developing and testing SLAM systems in the real world conditions can be challenging due to logical or financial constraints. A simulation environment provides a controlled platform for creating datasets and testing SLAM algorithms.

In this project, a 3D environment model is created using Blender using 3D map tiles. A simulation system consisting of this model and a drone is built in Gazebo, and Ardupilot is used as the flight controller. Using QGroundControl(QGC) and mavros, missions are planned on the drone, which flies around the simulated environment and captures data from the camera and GNSS sensors. The model from Blender and plugins from Gazebo provide the ground truth for this data. Two sets of data are used for this project, the UTwente dataset containing data from two drones, and the Vatican dataset containing data from three drones. A survey and preliminary test of useful collaborative SLAM frameworks is conducted, where CCM-SLAM is chosen as the viable option for this project. This algorithm is then configured for the simulation data. SLAM is performed on the simulated data from both the datasets, comparing the individual agent scenario with the multi-agent scenario. Some parameters of the algorithm are tuned based on the Root Mean Square Error (RMSE) response. The output from the SLAM algorithm has arbitrary scale and orientation, and to improve that, the data from the GNSS sensor is used to georeference the trajectory data. With this, the trajectories are aligned in the same way and then compared with the ground truth. RMSE values are calculated for individual and collaborative agent scenarios. It is seen that the percent RMSE with respect to the trajectory length reduces in the case of collaborative SLAM. The 3D reconstruction outputs are also examined by analyzing the point clouds. Collaborative SLAM merges the maps of individual agents and runs optimization on them. Hence, the maps from collaborative SLAM are denser and more accurately aligned with the ground truth.

Keywords: UAV, Visual-SLAM, Collaborative SLAM, Simulation, Gazebo, Point cloud

Chapter 1

Introduction

Simultaneous Localization and Mapping (SLAM) is the process of obtaining the 3D structure of the environment while also tracking the motion of the sensor in the environment. SLAM systems use inputs from various sensors such as cameras, LiDARs (Light Detection and Ranging) or IMUs (inertial measurement units) to create a dynamic understanding of their surroundings. This is useful in autonomous navigation, especially in GPS denied environments, or if pre-existing maps are unavailable or unreliable like in disaster areas. Autonomous vehicles need SLAM to plan paths and avoid obstacles, or to perform tasks without human intervention. Especially with UAVs, they can be deployed in inaccessible environments to navigate complex paths and get views from different perspectives. This can be used to create real-time maps of disaster zones, construction sites, and exploration areas. 3D-reconstruction is used to obtain a 3D model of the scene or object from different angles. It is then possible to detect anomalies by measuring angles and distances, identifying patterns, and visualizing the environment.

Visual SLAM (V-SLAM) uses visual information from cameras to perform SLAM tasks. The sensor configuration is simple, cameras are low cost and lightweight compared to other sensors like LiDAR. It uses computer vision principles to extract features from images, track their motion across frames, and reconstruct the surroundings in a 3D map. Photogrammetry techniques like triangulation, bundle adjustment, and pose estimation are used for extracting 3D information from 2D images by analyzing overlapping images. Although the processing overhead is more, many low computational cost algorithms are being developed [3]. Collaborative SLAM (C-SLAM) extends this concept to multiple agents that are working together to map an environment. This is used to make SLAM scalable, more robust, and improve accuracy.

UAVs, especially rotary ones, are compact, low cost, lightweight, and user-friendly solutions that can be autonomous. While solutions involving satellites or other aerial platforms are limited by image resolution and platform flexibility, UAVs have an upper hand in some ways [6]. They can be used in real-time applications and for frequent applications. They can be configured to adjust their trajectories and views according to the structure and geometry of the subject. Additionally, they can capture high resolution images that are useful for fine scale applications. Hence, UAV-based SLAM finds applications in inspection of structures like transmission lines or bridges, identification of vegetation species, precision management of crops, cultural heritage documentation, and disaster risk assessment. In real-time, it can be used to show the current situation and reveal changes.

In recent years, there have been efforts to perform tasks through collaboration of multiple UAV systems [7]. Drones in a collaborative system can work together to cover large areas efficiently, especially in cases where extensive coverage is needed rapidly. Dense and comprehensive mapping is possible with different perspectives. There is a redundancy in data collection, which improves reliability in case one drone fails, or when a point of interest already inspected by one drone needs to be revisited by another. Sharing information and tasks between each other also helps in resource allocation and task distribution optimization, since single drones may be limited by battery or line of sight restrictions.

In research, while developing a SLAM algorithm, its performance needs to be evaluated to check if the output of the algorithm matches the vehicle's actual pose. Hence, testing SLAM algorithms requires accurate datasets with well-defined ground truth data to objectively validate its performance, and quantitatively compute errors. A controlled environment that can mimic complex scenarios and generate high-quality datasets can be beneficial for SLAM testing [8].

1.1 Motivation

The development of real-time, collaborative, visual-SLAM is motivated by the growing demand for autonomous systems capable of navigating and mapping complex environments. UAVs, due to their ability to access difficult areas, are being increasingly used to create 2D and 3D maps to support disaster response, environmental monitoring and search and rescue missions. These tasks often span areas that are too large or intricate for a single UAV to map effectively. The spatial range that a single UAV can cover may be limited due to battery life or regulations prohibiting a drone from flying above sensitive structures or leaving the pilot's line of sight. Therefore, multiple drones are used to cover a larger area by working together and sharing information to build a unified, accurate map in real-time. This makes the tasks more efficient and scalable, and enhances robustness against individual sensor failures or occlusions. However, reconstructing a 2D or 3D map from multiple sources on a centralized system is non-trivial [9]. Challenges include loop closing, network connectivity and communication between UAVs.

Developing and testing SLAM systems in real-world conditions can be challenging due to logical constraints, safety risks or difficulty in gathering high quality ground truth data. There are a limited number of benchmark datasets that have multi-drone data, with sufficient overlap between the drones as suited for collaborative SLAM [7]. Simulation environments provide a controlled and cost-effective platform for creating datasets and testing SLAM algorithms. It can be used to mimic real-world conditions but with controlled parameters for things like noise, lighting, friction, wind, communication networks, dynamic objects, etc. By changing these parameters, different scenarios can be created to evaluate the performance of the algorithm in those conditions.

Based on this, the aim of this project is to:

- Explore efficient real-time monocular visual-SLAM frameworks
- Build a simulation environment to collect data from UAVs
- Implement an efficient collaborative mapping system that includes multiple (at least two) UAVs and a central server.

1.2 Research Objectives

The main objective of this project is to implement a collaborative SLAM algorithm to run on the data collected from drones within a simulated environment. In order to achieve this, the following research objectives need to be addressed:

1. Collect data from drones in a simulation environment
    - How to create a 3D model that can serve as an environment for the drone to fly in?
    - How to collect data from a simulated drone flying in that environment?
    - How to obtain the ground truth information so that it can be used to evaluate the performance of the SLAM algorithm?
    - What is the configuration of the drone, environment, camera, etc. that is suitable for visual-SLAM algorithms?
2. Implement an efficient and accurate real-time collaborative localization and mapping system for multi-UAV systems
    - What is the current state-of-the-art for SLAM algorithms?
    - What are the existing approaches for real-time collaborative visual-SLAM and which one is the most relevant for our application?
    - How does the existing SLAM algorithm perform with the simulated data when compared to the ground truth?
    - How does the collaborative approach compare to the non-collaborative one?
    - How can the SLAM algorithm be improved to make the results more accurate to the ground truth?

1.3 Outline

This report starts with the background of the fundamentals required for Visual-SLAM and an overview of its framework in chapter 2. In chapter 3, the state-of-the-art of the current SLAM landscape and the collaborative visual-SLAM architecture is presented. An exploration of techniques to create the simulation environment is also carried out. The methodology in chapter 4 describes the procedure followed in this project to create the simulation environment, collect data from the drones, run collaborative visual-SLAM on that data, and then improve the results to match the ground truth. The mapping and 3D reconstruction results obtained from the SLAM algorithm are presented in chapter 5. The report ends with a conclusion and recommendations for future work in chapter 6.

Chapter 2

Background

This chapter provides an overview of the background knowledge required as prerequisites for this project. First, some fundamental concepts like transformation matrix, euler angles and quaternions, camera model, epipolar geometry, and the coordinate systems used in aviation are explained. Then, the general framework of a visual-SLAM algorithm is described, along with the evaluation metrics used. The information for this chapter is derived from textbooks and lectures, such as *14 Lectures on Visual SLAM: From Theory to Practice* by Xiang Gao, et. al. [1], *Global Positioning Systems, Inertial Navigation, and Integration* by Mohinder Grewal, et. al. [10], and other related literature.

2.1 Fundamentals

In this section some of the some fundamental concepts behind SLAM algorithms and the techniques used in this project are explained.

2.1.1 Transformation Matrix

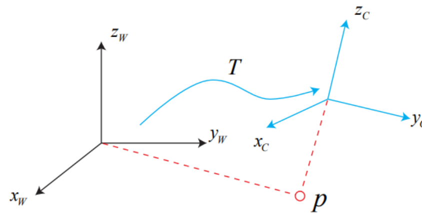


Figure 2.1: Transforming point $\mathbf{p}$ using transformation matrix $\mathbf{T}$. [1]

In a 3D linear space, if the base is described by (e_1, e_2, e_3), then a vector $\mathbf{a}$ has a coordinate $(a_1, a_2, a_3)^T$ under this base as:

$$\mathbf{a} = [e_1, e_2, e_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a_1e_1 + a_2e_2 + a_3e_3. \tag{2.1}$$

Suppose the world coordinate system is defined by x_w, y_w, z_w and the robot's coordinate system is defined by x_c, y_c, z_c, as in figure 2.7, a point $\mathbf{p}$ will have coordinates $\mathbf{p_w}$ and $\mathbf{p_c}$ in the two systems respectively. The coordinate values can be converted from one to another using a transformation matrix $\mathbf{T}$.

The Euclidean transformation involves rotation and translation during rigid body motion. If there is a unit-length orthogonal base (e_1, e_2, e_3) and after rotation it becomes (e'_1, e'_2, e'_3), then the coordinates of the vector $\mathbf{a}$ in the two systems are $[a_1, a_2, a_3]^T$ and $[a'_1, a'_2, a'_3]^T$ respectively. According to the definition of coordinates, the vector does not change, so:

$$[e_1, e_2, e_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = [e'_1, e'_2, e'_3] \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} \tag{2.2}$$

Multiplying both sides by $\begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix}$, the equation becomes:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} e_1^T e'_1 & e_1^T e'_2 & e_1^T e'_3 \\ e_2^T e'_1 & e_2^T e'_2 & e_2^T e'_3 \\ e_3^T e'_1 & e_3^T e'_2 & e_3^T e'_3 \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = \mathbf{R}\mathbf{a}' \tag{2.3}$$

Here $\mathbf{R}$ is the rotation matrix. When there is a translation $\mathbf{t}$ in addition to rotation $\mathbf{R}$ while transforming vector $\mathbf{a}$ to $\mathbf{a}'$, the equation becomes:

$$\mathbf{a}' = \mathbf{R}\mathbf{a} + \mathbf{t} \tag{2.4}$$

To make this transformation linear, homogeneous coordinates are used by adding a 1 at the end of the 3D vector, and the equation can be rewritten as:

$$\begin{bmatrix} a' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0^T} & 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix} = \mathbf{T} \begin{bmatrix} a \\ 1 \end{bmatrix} \tag{2.5}$$

$\mathbf{T}$ is the transformation matrix, where the upper left is the rotation matrix $\mathbf{R}$, right side is the translation vector $\mathbf{t}$, lower left is the $\mathbf{0}$ vector and lower right is 1. This is called as the special Euclidean group or $\mathbf{SE(3)}$.

Transform Name	Matrix Form	Degrees of Freedom	Invariance
Euclidean	$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	6	Length, angle, volume
Similarity	$\begin{bmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	7	volume ratio
Affine	$\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$	12	Parallelism, volume ratio
Perspective	$\begin{bmatrix} \mathbf{A} & \mathbf{t} \\ \mathbf{a}^T & v \end{bmatrix}$	15	Plane intersection and tangency

Figure 2.2: Comparing the properties of transformation matrices [1]

In addition to the Euclidean transformation, there are other transformations in 3D space, such as the similarity, affine, and perspective transformations. Euclidean transformations retain the shape, whereas the others will change the shape of the vector. These transformations are compared in figure 2.2. The similarity transform is also called as a $\mathbf{Sim(3)}$ transformation, which is used when there is scaling required.

2.1.2 Euler Angles and Quaternions

Euler angle provides an intuitive way to describe rotation by decomposing a rotation into three rotations around the three axes. Depending on the order in which the rotation occurs around the axes, it can be XYZ or ZYX and so on. One of the most commonly used Euler angles is the yaw-pitch-roll angles, which is equivalent to the rotation of the ZYX axis. The body is rotated along the Z-axis to get the *yaw* angle, then around Y-axis to get the *pitch* angle, and then around the X-axis to get the *roll* angle. The intrinsic 3D rotation matrices for yaw (α), pitch (β) and roll (γ) angles are given in figure 2.3. It is to be noted that if the reference frame is itself being rotated, the signs of the *sin* terms will be reversed.

$$R = R_z(\alpha)\, R_y(\beta)\, R_x(\gamma) = \overset{\text{yaw}}{\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}} \overset{\text{pitch}}{\begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}} \overset{\text{roll}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}}$$

Figure 2.3: Rotation matrices about the z,y,z axes [2]

A drawback of Euler angles is that it encounters the issue of Gimbal lock. If three real numbers are to be used to express three-dimensional rotation, there will be a singularity problem. Instead, Quaternions can be used, which are compact and not singular. Quaternions are less intuitive, and can be represented by a real part and three imaginary parts, as:

$$\mathbf{q} = w + xi + yj + zk \tag{2.6}$$

A 3D point $\mathbf{p} = [x, y, z]^T$ is rotated using a quaternion $\mathbf{q}$ to become $\mathbf{p}'$, the 3D point can be extended to a quaternion as:

$$\mathbf{p} = [0, x, y, z]^T = [0, \mathbf{v}]^T \tag{2.7}$$

The three coordinates are put into the imaginary part while the real part is 0. The rotated point $\mathbf{p}'$ can be expressed as the product:

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \tag{2.8}$$

To apply quaternion rotation to cartesian coordinates, the coordinates can be multiplied by the rotation matrix Q given in figure 2.4.

$$Q = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2zw & 2xz + 2yw \\ 2xy + 2zw & 1 - 2x^2 - 2z^2 & 2yz - 2xw \\ 2xz - 2yw & 2yz + 2xw & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

Figure 2.4: Quaternion rotation matrix [2]

2.1.3 Camera Model and Matrices

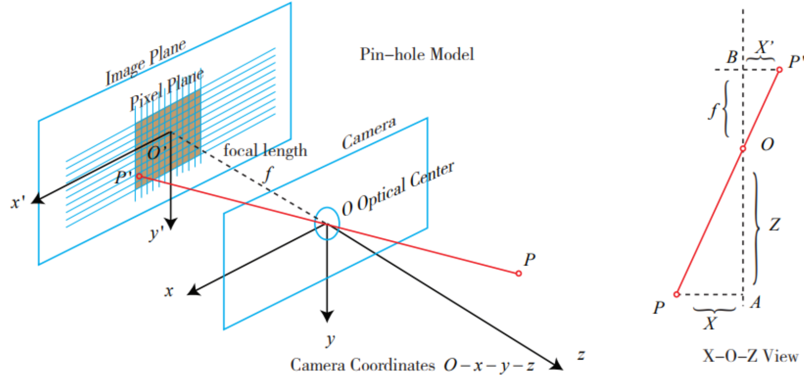


Figure 2.5: Pinhole camera model [1]

The pinhole camera model is shown in figure 2.5. In this, $O - x - y - z$ is the camera coordinate system, $O' - x' - y'$ is the imaging plane. A 3D point P in $[X, Y, Z]^T$ produces the image point P' in $[X', Y', Z']$. Then from the similarity of triangles in the right image:

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'} \tag{2.9}$$

In the equation, the negative sign is removed because modern cameras flip the image on their own and do not display them upside down. Given the pixel plane $o - u - v$, the pixel coordinates of P' is $[u, v]^T$. The relation between the coordinates of P' and the pixel coordinate is:

$$\begin{aligned} u &= f\frac{X}{Z} + c_x \\ v &= f\frac{Y}{Z} + c_y \end{aligned} \tag{2.10}$$

Here, f is the focal length and $[c_x, c_y]^T$ is the principal point. Using homogeneous coordinates:

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{KP} \tag{2.11}$$

The matrix $\mathbf{K}$ is the inner parameter matrix or intrinsics of the camera. In the equation, $\mathbf{P}$ is actually in world coordinate system $\mathbf{P_w}$ and should be converted to camera coordinate system by using rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$. Hence:

$$Z\mathbf{P_{uv}} = Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}(\mathbf{RP_w} + \mathbf{t}) = \mathbf{KTP_w} \tag{2.12}$$

The transformation matrix **T** is the extrinsic parameter of the camera. The intrinsic and extrinsic together is called the projection matrix or camera matrix.

2.1.4 Epipolar Geometry

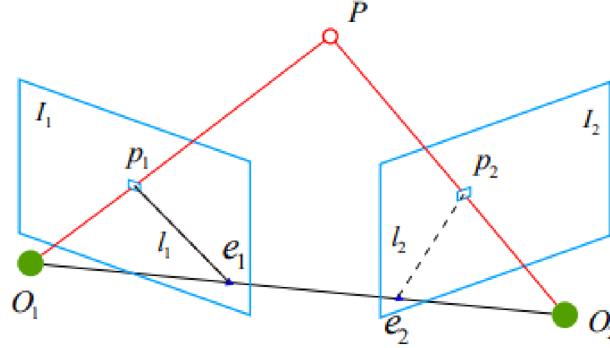


Figure 2.6: Epipolar geometry between two camera images [1]

When there are pairs of matched points between two images, the camera motion between the frames can be captured by epipolar geometry techniques. In figure 2.6, a feature point p_1 in frame I_1 corresponds to the feature point p_2 in frame $I2$. If O_1 and $O2$ are the centers of the two camera positions, the lines O_1p_1 and O_2p_2 will intersect at the point P in the 3D space. O_1, O_2, P form the plane called the epipolar plane. The line O_1O_2 is called the baseline, and the points where this line meets the frames, $e1, e2$ are the epipoles. The lines l_1, l_2 intersecting the epipolar plane and the image planes are called epipolar lines. For each feature point in one image, the same point must be observed in the other image on the same epipolar line. The line e_2p_2 in the second image is the projection of point P and the line O_1p_1. Hence, if the feature points in different images are known, the 3D location of the point P can be determined. Given the camera intrinsic matrix **K** and the transformation between the frames is **R,t** the epipolar constraint is given by:

$$\mathbf{p_2^T K^{-T} t R K^{-1} p_1} = 0 \tag{2.13}$$

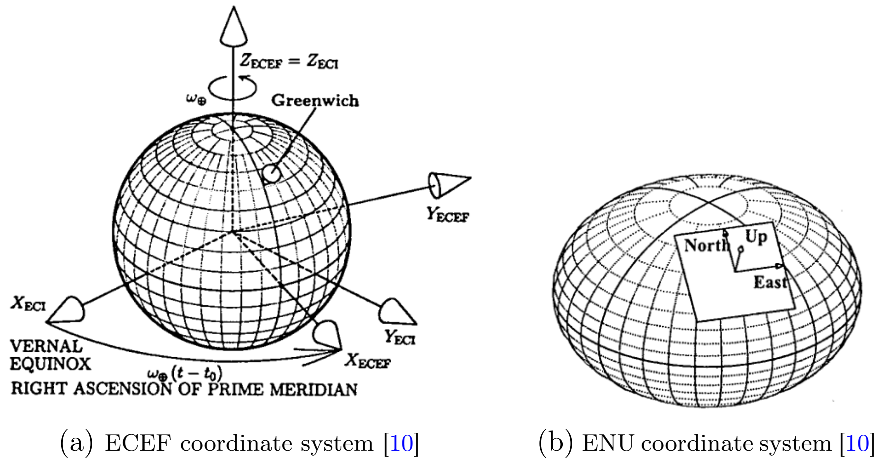
Here the Fundamental matrix F and Essential matrix E are given by:

$$\mathbf{E} = \mathbf{tR}, \quad \mathbf{F} = \mathbf{K^{-T} E K^{-1}}, \quad \mathbf{p_2^T F p_1} = 0 \tag{2.14}$$

2.1.5 Coordinate Systems Used in Aviation

There are many ways of representing a location on earth by a set of coordinates, such as the polar coordinates, earth-centered inertial (ECI) coordinates, earth-centered earth-fixed (ECEF) coordinates, etc. In this report, we do not go into the details of the various spatial

reference systems and their exact working, but the simplified topics required for this project are described. The ECEF coordinate system (figure 2.7a) has the center of the earth as its origin, the x-axis through the equator-prime meridian intersection, z-axis through the north pole, and the y-axis orthogonal to x and z axes. It rotates with the earth. GNSS uses ECEF as its primary coordinate system, and we have the latitude, longitude and altitude values. For example, the World Geodetic System (WGS84) is the ECEF system used in GPS.



(a) ECEF coordinate system [10]　　(b) ENU coordinate system [10]

Figure 2.7: Coordinate systems in aviation

The East-North-Up (ENU) in figure 2.7b and North-East-Down (NED) are two local tangent plane coordinates, which are defined by the local vertical direction and earth's axis of rotation. NED coordinates are preferred for aviation because the coordinate axes coincide with the vehicle's roll-pitch-yaw (RPY) coordinates when the vehicle is headed north. The transformation between ENU and NED frames is done by a 90-degree rotation in the XY plane by swapping x and y values, and then flipping of the z value.

2.2 Visual SLAM Framework

SLAM is a joint estimation of a robot's state and a model of its surrounding environment [7]. The robot's pose (position and orientation) and other sensor calibration parameters make up its state, whereas the environmental model consists of the representation of landmarks in a map.

A classic Visual SLAM framework involves the following steps: sensor data acquisition, visual odometry, filters, optimization, loop closure, and reconstruction [1], as shown in figure 2.8. The main sensor data for visual SLAM is collected from cameras, but other sensors like IMU, GNSS, encoders, etc. can also be integrated and synchronized. For the descriptions in this section, the book in [1] is referred.

Visual Odometry

Visual Odometry (VO) is the process of estimating the camera movement between adjacent frames and generating a local map. This happens in the frontend. There are direct,

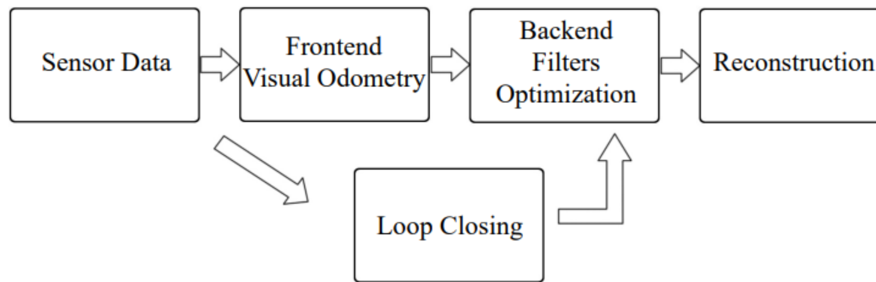


Figure 2.8: The classic visual SLAM framework [1]

indirect and demi-direct approaches to VO. Direct methods analyze the pixel intensities from the whole image and result in dense mapping, whereas indirect (feature-based) methods extract features from the images to estimate motion and result in sparse mapping. Semi-direct methods use a combination of the two methods. This project is focused on feature-based SLAM, hence these methods are described below.

Feature Matching:

Feature-based VO tracks features in the image to estimate motion. In that, some representative features from the image are selected so that they can be tracked between consecutive frames. For example, features like corners, edges, or blocks are more identifiable than individual pixels and can be distinguished between different images. Researchers have developed more stable local image features such as SIFT, SURF, ORB, etc. that are more distinctive, localized and efficient for tracking motion. ORB is one the most frequently used methods, and is explained below.

ORB (Oriented FAST and Rotated BRIEF) has shown to take significantly less time to extract features when compared to SIFT and SURF, hence it is suitable for SLAM applications. The key point is "oriented FAST", an algorithm that extracts corner points based on the principle that if a pixel is too bright or too dark compared to its neighbors, it is a corner point. BRIEF (Binary Robust Independent Elementary Feature) is a binary descriptor. The description vector encodes the relationship between random pixels near the key point by comparing the intensity levels and recording it as 1 if greater, 0 if lesser. This randomly selected binary representation makes it faster. The direction information from the FAST stage is used to rotate the BRIEF feature to achieve rotation invariance. Feature matching is then done by calculating distance between feature descriptors and selecting the closest one.

Triangulation:

Estimating the depth of the map points by using epipolar geometry is called triangulation. By observing the same landmark points at different locations, the distance to that landmark point is determined. It is important to note that triangulation is only possible when there is enough translation between the frames, and it cannot be done for pure rotation. Monocular SLAM uses the first few frames to extract features to initialize and extract depth information.

Filtering and Optimization

This block applies filtering or optimization techniques to the different camera poses from VO and the loop closing information to generate an optimized trajectory and map. This happens in the backend. Monocular SLAM initially used filtering techniques like Extended Kalman Filter (EKF) where every frame is processed to jointly estimate map feature locations and camera use. They have drawbacks of wasting computation power and accumulation of linear errors. Keyframe-based approaches on the other hand estimate only using selected keyframes allowing to perform costly but accurate bundle adjustment operations [4].

Bundle Adjustment and Graph Optimization are non-linear techniques used to adjust the camera poses and 3D points to ensure that the projected 2D features (bundles) match the detected results. Efficient sparse matrix solvers, such as Ceres [11] or g2o [12], are used to handle the sparse nature of the optimization problem. Local Bundle Adjustment is performed on a subset of keyframes (e.g., recent keyframes in a sliding window), while the Global Bundle Adjustment refines the entire map, typically after detecting loop closures.

Pose graph optimization represents SLAM as a graph, where the nodes correspond to camera poses and the edges represent relative transformations between poses, derived from visual feature matching or loop closure constraints. It is useful for drift correction. Efficient graph optimizers like g2o can be used.

Loop Closure

Loop closing is the process of determining whether the agent has visited the position before, and if so, the trajectory and map will be readjusted to reduce the accumulated drift. Place recognition algorithms are used to match the current camera frame against past keyframes using visual features. Bag-of-Words (BoW) is one of the frequently used methods for loop detection [13]. It is used to describe an image in terms of the features present by comparing them to pre-defined dictionary. When a loop is detected, the relative pose between the current camera position and the matched previous position is estimated using methods like PnP (Perspective-n-Point). The detected loop is then sent to the pose graph as a new constraint for optimization.

Reconstruction

The 3D reconstruction step involves constructing a 3D representation of the environment by using the depth information obtained during the triangulation step. The 3D map is represented as a sparse point cloud of 3D landmarks corresponding to features in the keyframes. Optimization techniques like bundle adjustment are used to minimize the reprojection error. Monocular SLAM has scale ambiguity due to having no information about the actual distances.

2.3 Evaluation Metrics for SLAM

The most commonly used metrics to evaluate SLAM accuracy are Relative Pose Error (RPE) and Absolute Trajectory Error (ATE). RPE is used to estimate the system drift by measuring the local accuracy of the estimated trajectory. ATE is used to compute the

$$E_i = (Q_i^{-1} Q_{i+\Delta})^{-1} (P_i^{-1} P_{i+\Delta})$$

$$RMSE(E_{1:n}, \Delta) = (\frac{1}{m} \sum_{i=1}^{m} ||trans(E_i)||^2)^{1/2}$$

(a) RMSE of Relative Pose Error [3]

$$F_i = Q_i^{-1} S P_i$$

$$RMSE(F_{1:n}) = (\frac{1}{n} \sum_{i=1}^{n} ||trans(F_i)||^2)^{1/2}$$

(b) RMSE of Absolute Trajectory Error

Figure 2.9: Evaluation metrics for SLAM algorithms [3]

difference between the true pose in the ground truth and estimated value from the SLAM algorithm. The root mean square error (RMSE) is used to calculate the overall error for the whole trajectory. The equations for these metrics are shown in figure 2.9a and 2.9b respectively. Here, for a timestep i, E_i is the RPE for the estimated pose P_i and the ground truth Q_i over the fixed time interval of Δ. F_i is the ATE for estimated trajectory P_i, ground truth trajectory Q_i and the rigid body transformation S corresponding to the least-squares solution.

Chapter 3

State Of The Art

This section first explores the state-of-the-art developments in Visual-SLAM, and some open-source SLAM implementations. Then the collaborative SLAM methods are explored, and some collaborative SLAM implementations are mentioned. Among these algorithms, the framework architectures that are interesting for this project are then discussed in detail. Lastly, since a part of this project is focused on building a simulation environment, the methods used to create it are also explored.

3.1 Overview of SLAM Methods

In [3], the authors present a survey of existing Visual-SLAM techniques based on the type of sensors used, the method of information extraction from images, and modern developments in SLAM methods, and they are as follows. The different types of sensors used in V-SLAM can be Monocular, Stereo, RGB-D, and Event cameras, or cameras in combination with other sensors like IMU and LiDAR [8]. V-SLAM can also be classified into Direct, Feature-based and Semantic SLAM based on the methods used to extract information from images. Some modern SLAM methods employ dynamic-aware or dynamic-inclusive methods to handle dynamically changing environments. Others use deep-learning techniques to enhance accuracy.

Monocular camera based SLAM such as MonoSLAM [14] is simple and cheap, and relies on tracking the motion from images captured by a single camera. But these cameras cannot accurately estimate depth, especially when static. Stereo and RGB-D cameras are good for depth estimation but they can have complex calibration and narrow measurement range respectively [3]. Multimodal SLAM combines two or more sensors to increase robustness. The fusion of IMU with cameras in Visual-Inertial SLAM, or the fusion of other sensors like GNSS and LiDAR with cameras, is useful for improving tracking and accuracy especially in texture-less or uneven lighting environments [15].

Direct SLAM methods use the entire original images as input for tracking and mapping [16]. This results in a dense reconstruction and detailed representation of a scene, but requires high computational resources. Examples of some direct SLAM methods are DTAM [17], LSD-SLAM [18] and DSO [19]. Feature-based SLAM, such as ORB-SLAM [4], extracts features from the images and uses computer vision techniques to estimate motion. This requires lower computational resources and is robust in dynamic scenes, but constructs a sparse reconstruction. Semi-direct methods like SVO [20] use both methods, such as performing feature-based tracking but mapping using direct methods.

Semantic SLAM performs semantic segmentation and extracts semantic features to produce a more informative map [21], but it is also computationally intensive.

Dynamic-aware methods, such as in [22], [23], detect and remove dynamic objects from the visual data as a pre-processing step, but lacks scalability. Dynamic-inclusive methods, such as [24] utilize motion models to estimate movement of objects and include it in the SLAM process, but this is computationally demanding.

In recent years, deep learning methods have been used to extract rich, high-level features from images, and can be fused with traditional SLAM methods [25].

From the data compiled by the authors in [3], it is seen that Monocular SLAM and Feature-based SLAM are the most commonly researched and published areas in the past twenty years. Monocular SLAM using a single camera that is low-cost, compact and easily available everywhere makes it a ubiquitous solution for various applications. Feature-based methods use well-established techniques that perform well in a wide range of environments. Moreover, it is computationally less intensive and allows for real-time processing on standard hardware. These characteristics are the most important for a collaborative real-time SLAM system, hence it was decided that this project will use monocular indirect SLAM methods. When used with other sensors like IMU or GNSS, the arbitrary scaling problem in monocular SLAM can be resolved. Other methods involving deep learning are beyond the scope of this project, but they can also be used to improve the results.

Open-Source SLAM Approaches

A comparison of open-source Visual-SLAM approaches has been presented in [26]. For the purposes of this project only the real-time, feature-based, monocular methods among them are considered.

ORB-SLAM [4], and its successors, ORB-SLAM2 [27] and ORB-SLAM3 [28] are very popular and have proved to be efficient in various scenarios. The **ORB-SLAM** architecture as shown in figure 3.1 uses ORB features, Bags of Binary Words and g2o optimizer for performing tracking, mapping, localization and loop closure. ORB-SLAM has three parallel processes, one to construct local trajectory by matching key points, one to build the local map with bundle adjustment and one for loop closure. A global optimization is then carried out on the entire trajectory. The second version, **ORB-SLAM2** is similar to the architecture in figure 3.1, but allows for the use of stereo and depth cameras. **ORB-SLAM3** improves upon this system, and is capable of performing visual, visual-inertial and multi-map SLAM with monocular, stereo, RGB-D cameras of both pinhole and fisheye lens models. It also has an updated place recognition module with improved recall. **Open-VSLAM** [29] is similar to the ORB-SLAM2 framework, but with additional features from other frameworks like ProSLAM and UcoSLAM. It offers practical extensions and support for different camera models like perspective, fisheye and equirectangular lens.

In some frameworks, an IMU sensor is used in combination with the camera in order to drift and arbitrary scale in purely monocular systems. The **VINS** algorithm [30] uses the Kalman filter to merge image and IMU data to estimate the trajectory. OpenVINS [31], VINS-Mono [32], VINS-Fusion [33] are examples of algorithms that implement this. VINS-Fusion is an extension of VINS-Mono and supports loop closure with multiple visual-inertial sensor types along with GPS. **Kimera** [34] is a C++ library that uses a fast VIO pipeline to build a semantically annotated 3D mesh.

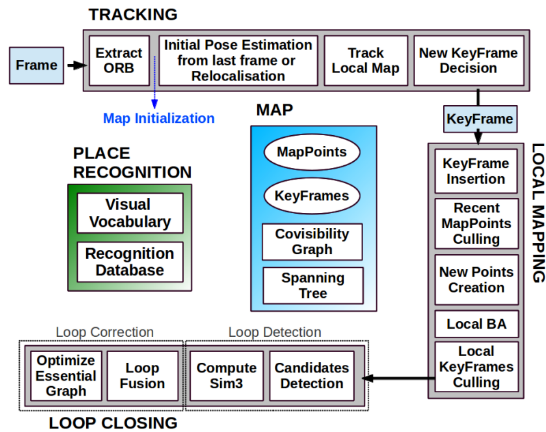


Figure 3.1: ORB-SLAM system overview [4]

After performing various tests on these algorithms, the authors in [26] have concluded that ORB-SLAM2 and ORB-SLAM3 provide reliable trajectory estimation in different environments, sensor setups and presence of dynamic objects. The VINS algorithms benefit from using IMU data and show competitive results for versatile datasets. But ORB-SLAM2 and OpenVSLAM also get some of the best results among different datasets even without the use of IMU when compared to the other VINS algorithms. Kimera was found to be of low usability when tested on datasets other than EuRoC. However, it is to be noted that no solution can be one-for-all, and that the SLAM algorithm chosen would work best when it is task and environment specific.

The authors in [26] also cite the practical problems that arise when trying to use the open-source solutions. These algorithms require specific versions of Ubuntu and other packages to function properly. If they have a ROS interface, it makes it convenient for launching. Or, the most convenient way is to have a ready-made Docker image that can be run, which takes care of installing required packages and launching files. The authors of the SLAM systems should also provide adequate documentation and have an active support forum to address issues. Hence, ease and convenience of operation becomes important while choosing a SLAM algorithm.

3.2 Overview of Collaborative SLAM Methods

In single-robot SLAM, the pose and map estimates are expressed in a local reference frame. But in Collaborative-SLAM (C-SLAM), the pose and map of each robot needs to be expressed in a shared global reference frame. This allows the robots to collectively perceive the environment and benefit from each other's observations. Such a collaborative system can be built in a centralized, decentralized, or distributed manner [7].

In the case of a **centralized** system, there is a central estimator has a global view of all the agents and performs the SLAM estimation. In some works, the individual agents send their sensor data to the central node which then processes the data [7]. In some others, such as in [5], [35], the system may be distributed, which means that the individual agents are also capable of performing some computation parts while the central node merges the

results and performs computationally intensive tasks. Some systems may also have replicated central servers so that the agents do not all have to rely on a single computer.

As the number of agents increases, it becomes difficult for centralized systems to operate due to communication constraints. The system can be **decentralized** to improve scalability, where each robot has its own view and only a partial view of its neighboring agents. The system is made to ensure that the estimates on each robot is consistent with its neighbors, so that it reaches global consistency iteratively and over time. These systems, such as in [36], [37], are therefore usually distributed, where each robot performs its own estimation using its local information and the partial information from the neighbors. Decentralized systems have their own issues, such as synchronization, avoiding double counting and bookkeeping.

Open-Source Collaborative SLAM Approaches

Among the several methods for SLAM, centralized monocular approaches were of interest. This is because they offer simplicity, cost-effectiveness, require less computational resources on board, so they can be deployed on small devices like UAVs. There are a few open-source packages that satisfy the conditions of being real-time, collaborative, centralized, and monocular, and they are described briefly below. Other approaches including decentralized, non-monocular and non-real-time methods are not considered here, but there are quite a few options for them as well, as discussed above.

In [9], a centralized architecture is proposed, where each client agent independently explores the environment running a limited-memory SLAM onboard, and sends its data to a central server with increased computational resources. The server manages maps from all agents and triggers loop closure, map fusion, optimization, and sends updated map information back to the agents. **CCM-SLAM** [5] builds on this architecture with an efficient communication strategy accounting for bandwidth constraints and is also demonstrated on real experiments. It is able to detect and remove redundancy from the server map without compromising robustness, and handle network problems while flying three UAVs simultaneously. **CVI-SLAM** [38] presents a visual-inertial centralized collaborative framework. The architecture is similar to that of [9], but the agents run visual-inertial odometry on IMU and camera data. Adding an IMU helps to improve accuracy and manage the arbitrary scale to an extent. **ORB-SLAM3** [28] presents the first system to perform visual, visual-inertial and multi-map SLAM with monocular, stereo and depth cameras of different lens models. It is also the first system to allow map reuse with tracking recovery. It is shown to operate robustly in real-time, in indoor and outdoor environments. **COVINS** [39] extends the architecture of CVI-SLAM towards a more flexible, scalable and efficient setup. Some of the improvements include- a generic communication interface allows for interfacing different VIO systems with the server, more efficient map management, optimization schemes, and redundancy detection. This allows for the use of up to twelve agents simultaneously. **COVINS-G** [40] builds upon the COVINS framework and presents a generalized server back-end that is compatible with any VIO front-end. This allows for support of diverse hardware and shifts from a tightly coupled implementation to a more generic and reusable architecture. It has been tested on different cameras including the Realsense depth and tracking cameras, and different frontends like VINS-Fusion, ORB-SLAM3 and SVO.

3.3 CCM-SLAM Architecture

Based on the above investigation, it was found that CCM-SLAM and COVINS could be viable options for this project. The architecture of CCM-SLAM is discussed below in detail. The architecture of COVINS is not discussed here in detail, because it is similar to that of CCM-SLAM. The difference is that it uses a VIO module instead of a VO module on the agents, and has additional redundancy detection and optimization methods in the central server module.

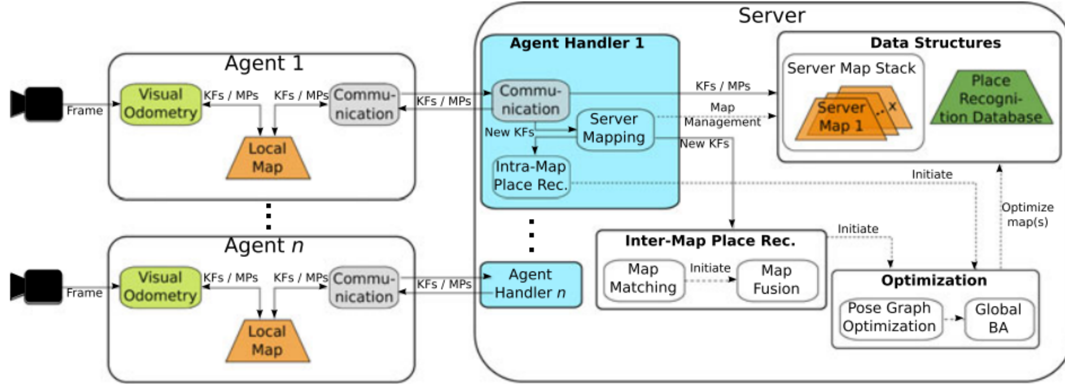


Figure 3.2: CCM-SLAM system architecture [5]

Figure 3.2 shows the architecture of the CCM-SLAM system [5]. Each agent runs its own real-time VO to estimate pose and map. The local map is limited to N closest keyframes in the vicinity of the agent to conserve resources. The server acts as a book-keeper and stores the map information from all agents in the server map stack. It also runs place recognition, global optimization (bundle adjustment) and redundancy detection. All maps use a local odometry frame and inter-map information is exchanged in relative coordinates from one frame to another, and a fixed global reference frame is not used. The server and clients run a bidirectional communication module to establish exchange of map information. All communication between agents happens through the server, so if the connection to the server is lost, the agent cannot receive the map information from the server. However, the agent can still run its own VO with limited map window so that its autonomy is not lost.

Keyframe-based VO: The VO of CCM-SLAM is implemented using front-end of ORB-SLAM as seen in the earlier section. The motion of the camera is estimated by tracking landmarks in the local map, which are stored as 3D map-points. Only the selected keyframes that represent significant changes in the environment are stored. The keyframes and map-points in the map are connected by edges as in a graph-based map. The tracking and mapping threads are run in separate parallel threads.

Local map: The local map is a SLAM graph connecting consecutive keyframes and map-points in the agent's vicinity. A pose graph is deduced from this map for pose-graph optimization. A trimming algorithm is used to limit the number of keyframes stored to N, with a buffer of another N to avoid data loss, and this number is dependent on the computational power of the system.

Server map stack: All the maps from the agents are stored in the server map stack, one for each of them. It does not have a limitation of N keyframes, so all the previous

keyframes are also stored. When two server maps are merged, a new single map is created and the previous two maps are deleted.

Agent handler: Each agent gets its own agent handler in the server to initiate communication and create a new server map. A map manager and intra-map place recognition module is created for the server map and they all run in parallel. Additionally, when two server maps are merged, a Sim(3)-transformation is carried out to convert the frame of one agent to another.

Communication modules: ROS is used for communication between the agent and the server. Technically, ROS is not real-time, but since neither the agent nor server need data at a fixed rate and can keep track of changes, it is not a necessity. On the agent side, the communication module keeps track of changes in the map, and is responsible for packing the converted messages in way to not overload the system. On the server side, it sends the keyframes in the server map with the strongest covisibility edges to those in the agent's local map. These exchanges use a relative coordinate scheme instead of an absolute one. check if anything else needs to be added here

Server mapping: In the server mapping module, the newly arrived keyframes are forwarded to the keyframe database, intra-map place recognition and map-matching module. The connections between new information and existing pose graph is established. Redundancy detection in case of previously visited areas, or information from multiple agents, is also another task performed.

Place recognition and keyframe database: This detects overlaps between locations in the server map stack using the keyframe database, even if different camera parameters exist. DBOW2 is used for efficiently looking-up features. A successful intra-map place recognition triggers a bundle adjustment.

Map fusion: If there is an overlap between two server maps, a map fusion is triggered. Keyframes from the two maps are matched, and a Sim(3)-transformation is calculated. A new map is created to fuse all map information by retaining one coordinate frame and transforming the other. Global BA is then performed on the resulting map, and transferring the information to the agent handlers.

Global BA: Global optimization is activated when there is a match between two server maps, or when there is a loop detected within a server map. It minimizes the reprojection error to improve accuracy and reduce scale drift. In CCM-SLAM, the g2o algorithm is used for BA.

3.4 3D Modeling of the Simulation Environment

Tests for SLAM algorithms are usually performed on benchmark datasets such as the EuRoC MAV, TUM-VI, KITTI, etc. Surveys and comparison of these datasets have been done in some papers such as in [26]. But as the authors in [7] explain, not many datasets are available for collaborative-SLAM. The current approach is to take one of the benchmark datasets and divide it into multiple parts while ensuring sufficient overlap between the parts for loop closing. Some papers show researchers running their own experiments with multiple UAVs to record datasets to feed into the SLAM algorithm. This process can be time-consuming and may not be feasible in some cases. A simulation environment that can record data can be very beneficial to test SLAM algorithms.

- The configuration of sensors, environment, factors like noise, lighting, occlusions, dynamic objects, etc. can be controlled.
- Experiments are reproducible and can be repeated with identical conditions.
- Tests can be performed by varying parameters, and the results compared.
- The ground truth for the trajectories and map become directly available.
- The environment can be tailored to the application, such as indoor, outdoor, urban, driving, UAV, etc. and a variety of sensors can be implemented.
- Certain real-world scenarios like issues that arise due to connectivity, environmental factors and noise cannot be fully realized in a simulation, but models to approximate those cases can be used.

To build a good simulation environment for Visual-SLAM, the camera data should have certain characteristics to ensure accurate feature extraction, and consequently, for motion estimation and map creation. The images should have a good resolution, consistent frame rate, stable lighting and sufficient overlap with steady motion, while the area captured should be textured with distinct visual features [1]. With a simulation environment, it is possible to control these criteria and modify them as required. To supply sensor data to the SLAM algorithm, a simulated drone should fly around in a simulated environment and collect data from its sensors such as cameras, IMU, GNSS, etc.

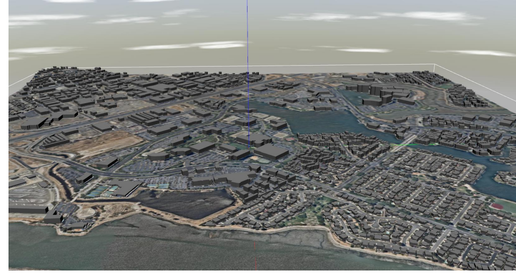
Methods to Build a 3D Model

The first step to build such an environment is to construct the model of an area to be scanned. One way to do it is to manually make a 3D model from scratch, but that could be tedious. Instead, assets and models can be downloaded and modified from websites like CGTrader [41], Sketchfab (Fab) [42], Open3DModel [43], TurboSquid [44] or collections such as in [45] or [46]. 3D asset websites also have high quality models that can be purchased, apart from the free and open-source options. While the websites can offer more realistic looking models, the world collections for gazebo are less detailed with limited textures. An example of a small city model and the KSQL airport model is shown in figure 3.3. Some well known 3D modeling software are Blender [47], Unity [48], Unreal Engine [49], and they are all free for personal use. The most commonly used simulation software in robotics is Gazebo [50], which easily integrates with ROS [51], has a physics-based customizable environment with plugins for real-time sensor applications.

Another way to build realistic models of an outdoor urban area is to simulate a cityscape. Some famous landmarks and cities are available on the websites mentioned above, and can be directly downloaded. Examples of such models from Sketchfab are shown in figure 3.4. But it is also possible to easily import photorealistic 3D map tiles provided by mapping platforms.

OpenStreetMap (OSM) [52] is an open-source collaborative platform that allows free access to map images and data to be used by other applications such as QGIS, Mapbox, Blender and Cesium [53]. Cesium [54] is specialized in 3D geospatial data and provides high resolution map tiles and terrain models. It can integrate with OSM for building tiles, and with Unreal Engine to create interactive 3D environments. Esri ArcGIS [55] is a leading geospatial information system (GIS) platform that provides 3D map tiles, including

(a) A Gazebo world of a small city



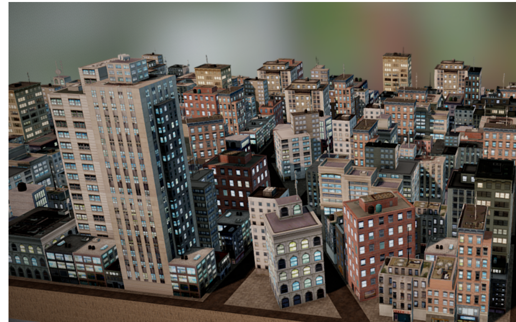
(b) KSQL airport in the PX4 Gazebo world collection

Figure 3.3: Examples of Gazebo world environments

detailed terrain and building data. There are tools and APIs hat can be used with custom data layers to access 3D tiles.

Google Maps is a well known mapping service known for its extensive satellite imagery and street data. Since 2023, Google has made Photorealistic 3D Tiles [56] publicly accessible through APIs. However, the coverage for 3D tiles is only around 55 countries while for the 2D tiles it is over 250 countries, and more details about that can be found in [57]. The availability of highly detailed map tiles is also limited to areas with dense population or famous landmarks, while the other areas are less detailed. Additionally, some residential areas may be blurred due to privacy concerns. Despite this, the Google 3D Map tiles were found to be quite convenient to use and a good resource for obtaining a 3D model for our purposes, since we just need a well detailed 3D model of an urban area.

(a) Model of Château de Wideville in France



(b) Model of New York City

Figure 3.4: Examples of cityscape models in Sketchfab

Drone Simulator

The second part of the simulation is to use a drone simulator to fly around the 3D environment and collect data. There are several drone autopilots available, but PX4 and ArduPilot are most widely used and have good compatibility with ROS. For research purposes, either one of them can be used without issues, and it depends on the user's level of comfort. For this project, ArduPilot was deemed suitable due to its ease of operation with ROS and Gazebo. Together, ROS [51], Gazebo [50], and ArduPilot [58] provide a robust simulation environment for testing drone applications. Gazebo acts as the physics-based simulation platform, offering sensor modeling and physics interactions for drones. ROS provides an interface to interact with the model using nodes and topics. ArduPilot can be

integrated with ROS and Gazebo using MAVROS to simulate the drone's flight controller behavior and receive data from the onboard sensors.

AirSim [59], an open-source simulator for UAVs is an alternative. It provides a realistic environment for navigating drones and collecting sensor data. It is based on Unreal Engine, and also has certain APIs and wrappers to connect to ROS nodes.

Simulation Workflows

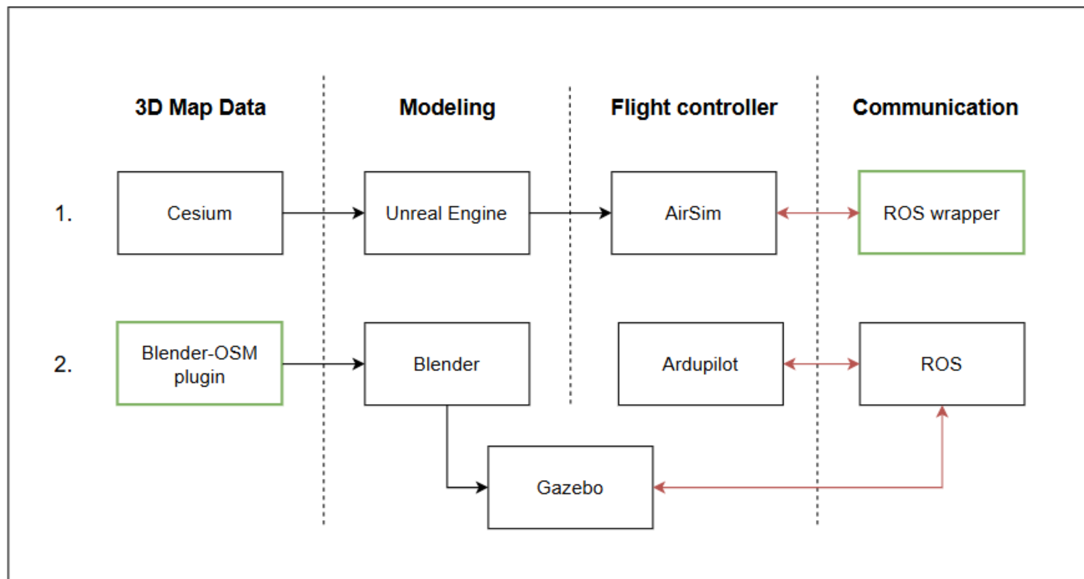


Figure 3.5: Two approaches for the simulation environment

Based on the above options, there are two workflows that were identified, as shown in figure 3.5. The red arrows mark communication pathways, the green boxes represent plugins/wrappers. The first method is to use Unreal Engine with Cesium plugin to import a 3D model of a city, and then use the AirSim drone simulator with sensor plugins and ROS wrapper to collect data. The second is to use Blender with OSM plugin to import the 3D model and use Gazebo-ROS to collect data. Cesium and Unreal Engine provide high-resolution 3D map data that are rendered into photorealistic environments. But because of this, they require high computational requirements. Gazebo environments are less realistic, but they have lower computational overhead and are easy to integrate into ROS-based pipelines. Moreover, it is widely used in robotics research and has a large support community. Due to this, the second workflow is more desirable for this project.

Chapter 4

Methodology

This chapter describes the procedure followed during the course of this project to meet the objectives. An overview of this is depicted in figure 4.1. There are two sections to this project. One is the data acquisition part, and the other is the SLAM algorithm part. The steps involved in the first part are: making a 3D model of the environment, building the gazebo world, setting up the drone simulator, and sensor data acquisition from drones flying in the simulation environment. The steps involved in the second part are: comparing CCM-SLAM and COVINS with a benchmark dataset, configuring the chosen algorithm to run on the simulated data and changing parameters, and georeferencing the outputs using GNSS data.

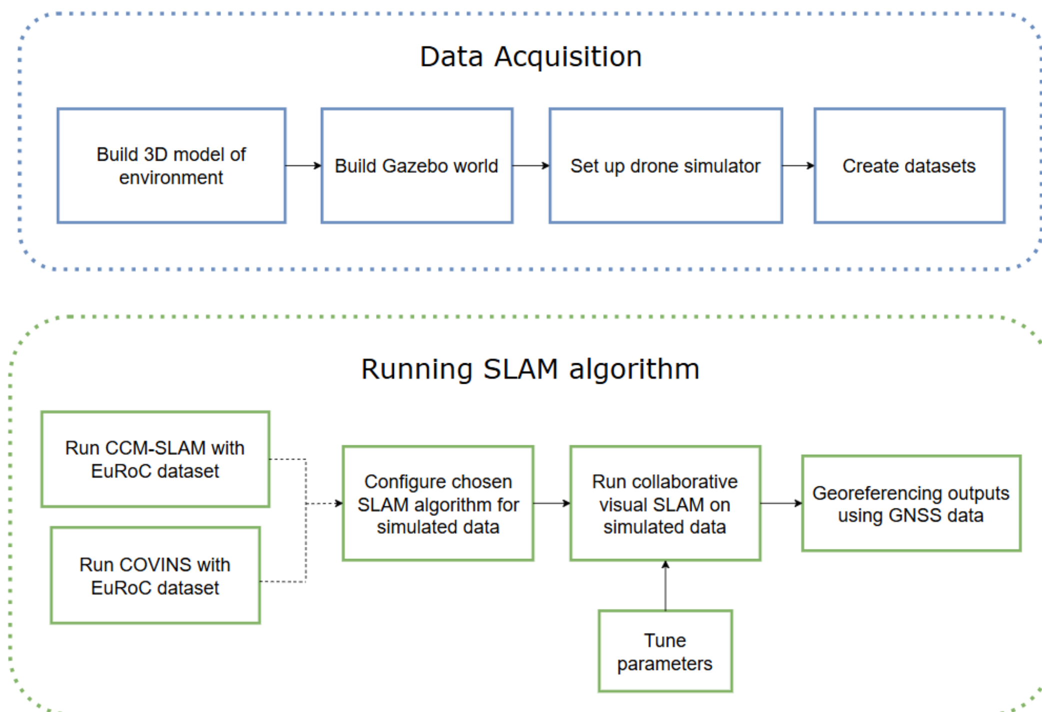


Figure 4.1: Overview of the methodology: 1. The data acquisition process 2. The SLAM algorithm process

Section 1: Data Acquisition Process

4.1 Building 3D Model of the Environment

BLOSM (Blender-OSM) [60] is an add-on for Blender to easily download and import OpenStreetMap, Google 3D tiles and terrain data. There are two versions - a base version which is free, and a premium version which is paid. While the premium version offers more detailed texture options, the base version is also a decent choice to create urban models with textures. For this project, the free addon for Blender is used.

Blosm vesion 2.7.8 was installed on Blender 4.1, as shown in figure 4.2. As seen in the picture, the tool requires an access token/key to access satellite imagery from either ArcGIS, Mapbox, or Google 3D Tiles. Since the Google map tiles is being used, a Google API key [61] needs to be created, for which a billing account is required. However, Google offers $200 worth of free credit per month, which is sufficient to download a few maps. After creating the key, the Map Tiles API needs to be activated, and then the key can be copied to the add-on tray. After the blosm tool is activated in Blender add-ons, it appears on the side panel as shown in figure 4.3.

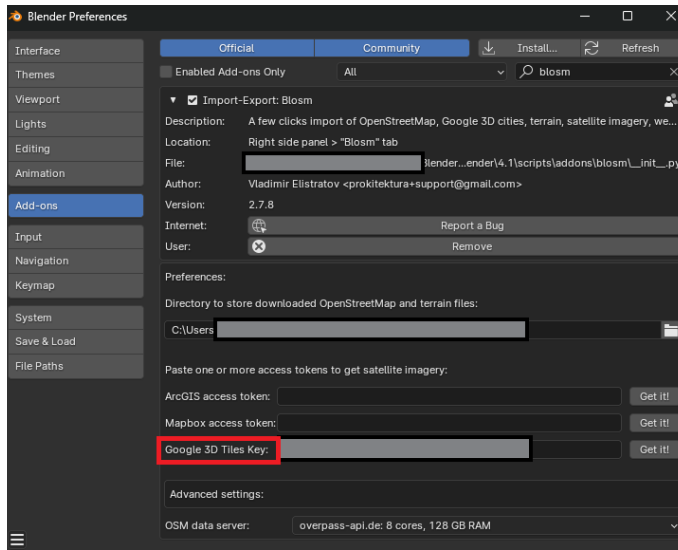


Figure 4.2: Installing the Blosm add-on

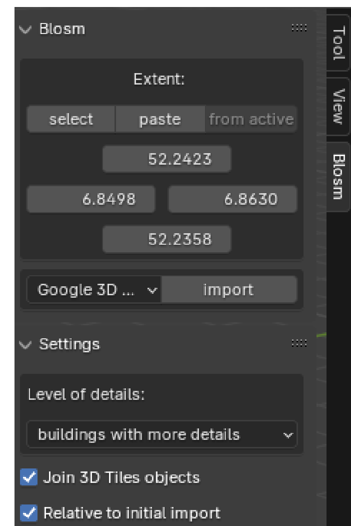
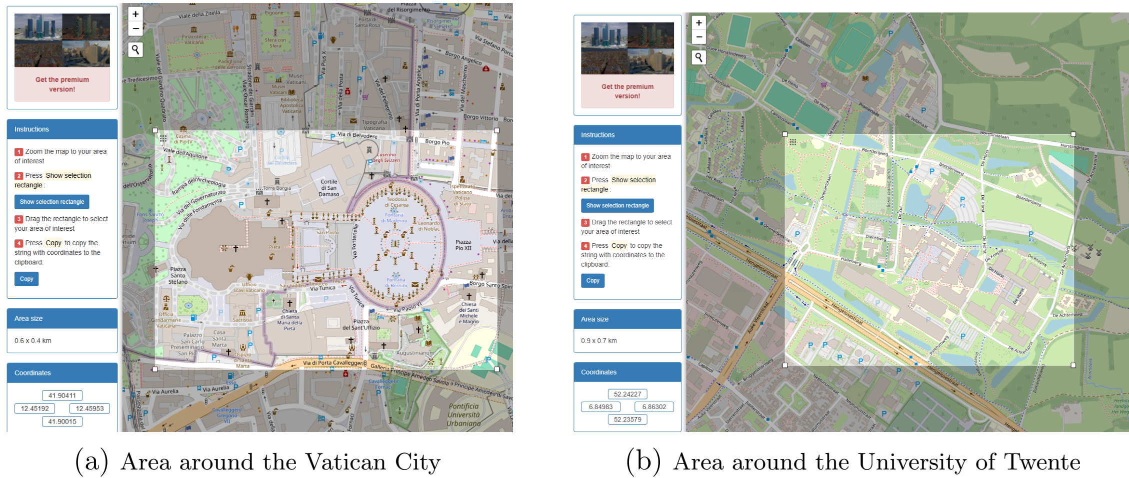


Figure 4.3: Blosm tool panel

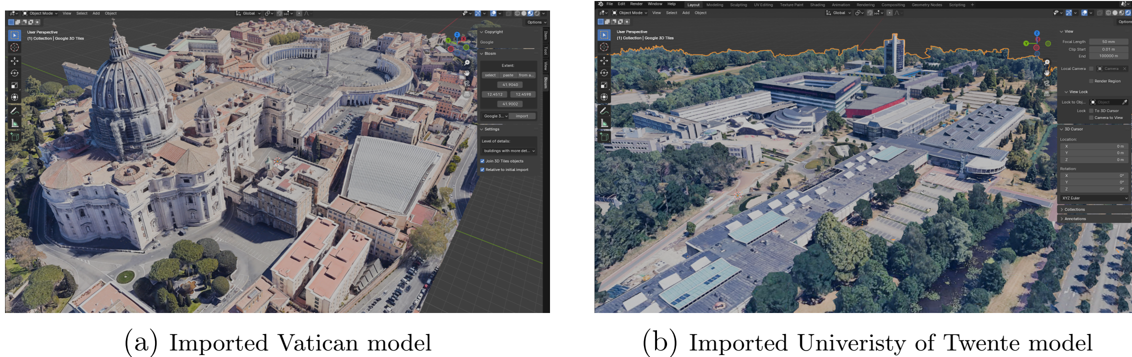
Here, an area selection can be done to download the Google 3D map tiles for that location. To select its corresponding latitudes and longitudes, the OpenStreetMap tool can be used, as shown in figure 4.4. Examples of selecting the area for the model of the Vatican City and the University of Twente are shown in figures 4.4a and 4.4b. The locations selected for making these models, like the Vatican City or University of Twente, are public places, so that makes it easier to obtain the detailed 3D map tiles. Once the area is selected, its geographical coordinates can be copied to the blosm tool window. In the tool window, the options for "Google 3D map tiles" for "buildings with more details" are selected from the drop-down menus. It is to be noted that downloading very large detailed

areas takes a long time to import, and uses a lot of tiles, which might exceed the free limit. It was found that selecting an area upto 1km x 1km works well in Blender, but requesting larger areas will take more time and processing power. An area of around 0.5km x 0.5km is suitable for creating the world file in Gazebo, since a larger model would require a powerful machine to run the multi-drone setup. The imported 3D models for the selected areas are shown in figure 4.5. To import an area of 0.5km x 0.4km it took around 35s, and for an area of 0.9km x 0.7km it took around 110s on a Lenovo Thinkpad P15v Gen 2i with Intel i7-11800H processor (2.30GHz, 8 cores), 32GB RAM. The imported model is slightly offset from the origin, so it is aligned by bringing the center of the model to the origin.



(a) Area around the Vatican City (b) Area around the University of Twente

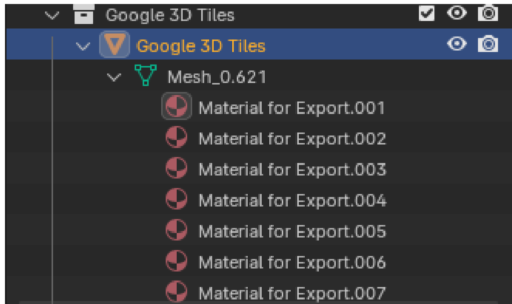
Figure 4.4: Selecting rectangular areas in OpenStreetMap



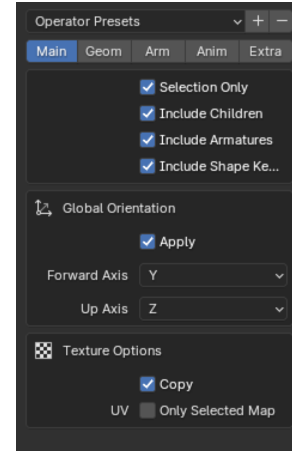
(a) Imported Vatican model (b) Imported Univeristy of Twente model

Figure 4.5: Imported models in Blender

Before exporting the model for Gazebo, the materials are prepared by replacing them with export-ready ones. When the materials are ready for export, they look like the figure 4.6a. Then the texture resources are unpacked and saved into separate image files. Then the model can be exported as a DAE (COLLADA) file with the settings shown in figure 4.6b. If the model has been correctly exported along with the texture files, the DAE file looks like the figure 4.6c. Here, each of the material from 4.6a corresponds to a JPEG image and will be used to provide texture to the mesh.



(a) Materials ready for export



(b) Export options for the .dae file

```
<library_images>
  <image id="Image_0" name="Image_0">
    <init_from>Image_0.jpg</init_from>
  </image>
  <image id="Image_0_001" name="Image_0_001">
    <init_from>Image_0.001.jpg</init_from>
  </image>
  <image id="Image_0_002" name="Image_0_002">
    <init_from>Image_0.002.jpg</init_from>
  </image>
  <image id="Image_0_003" name="Image_0_003">
    <init_from>Image_0.003.jpg</init_from>
  </image>
  <image id="Image_0_004" name="Image_0_004">
    <init_from>Image_0.004.jpg</init_from>
  </image>
  <image id="Image_0_005" name="Image_0_005">
    <init_from>Image_0.005.jpg</init_from>
  </image>
```

(c) Textures included in .dae file as .jpg images

Figure 4.6: Steps to export model from Blender

4.2 Building the Gazebo World

The Gazebo simulation was carried out on a Dell Precision 7920 Tower running Ubuntu 20.04.6 LTS with an Intel Xeon Gold 6144 CPU (32 cores @ 4.20 GHz), NVIDIA TITAN Xp GPU, and 256 GB of RAM.

The model exported from Blender is imported in Gazebo by creating a gazebo model and world file. The structure of the gazebo project folder is shown in figure 4.7. The texture JPEG files and the DAE file from the Blender model are copied to the meshes folder. The `model.config` file is a configuration file that provides metadata about the model. The `model.sdf` file is used to describe the model and environments. This is used to reference the DAE file and create a *link* with certain parameters like pose, visual and collision elements.

The world file is used to define the environment (like lighting, textures, etc.), physical properties (like gravity, friction, etc.), models (environment, robot, etc.) and their configurations. A directional light source is used with shadows enabled. Parameters like gravity, friction, magnetic field, atmospheric effects are defined. The simulation is set to run at real-time with an update rate of 1000 times per second with a step size of 0.001s.

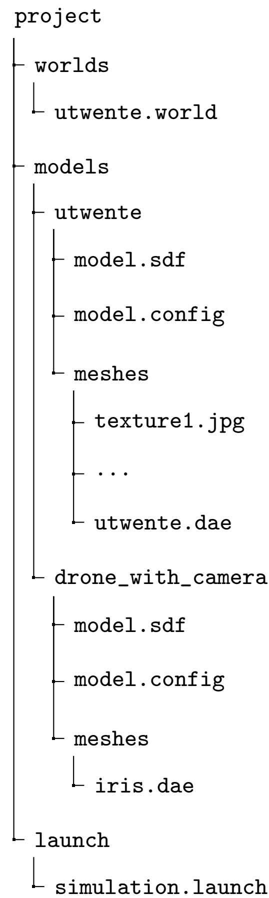
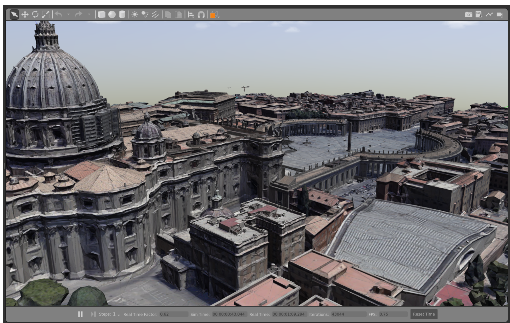
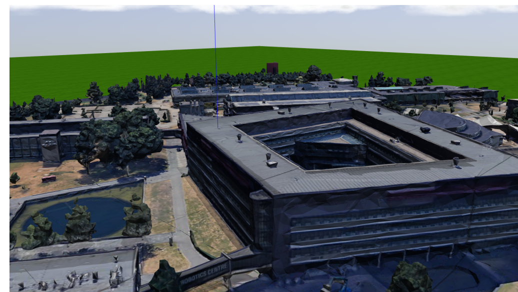


Figure 4.7: Simple folder structure for gazebo simulation



(a) Imported Vatican world in Gazebo



(b) Imported UTwente model in Gazebo

Figure 4.8: Imported worlds in Gazebo

Depending on the computational capabilities of the system, these values can be changed as required. In the `utwente.world` file, a ground plane is defined, upon which the `utwente` model is placed. In aviation, there is a difference between the coordinate system of a Flight Controller Unit (FCU), which uses NED (North-East-Down), compared to Gazebo, which uses ENU (East-North-Up). The simplest way to resolve this is to rotate the model in gazebo by 90 degrees in the counter-clockwise direction so that it can match with the heading of the FCU enabled drone. So the pose of the `utwente` model looks something like `( 0 0 0 0 0 -1.571)`.

The launch file (`simulation.launch`) is used to reference and launch the world file in Gazebo using ROS. Successfully imported models in Gazebo are shown in figure 4.8.

4.3 Setting up the Drone Simulator

Setting up the drone simulator with the help of Gazebo, ROS, ArduPilot and MAVROS provides a comprehensive environment to acquire data from the environment. Either PX4 or ArduPilot can be chosen as the FCU and they would work well, but ArduPilot is chosen for this project . A drone model is introduced to the above gazebo world, and can interact with its surroundings and capture data. Different sensors are attached to it, and ROS plugins are used to simulate sensor behavior. ArduPilot acts as its flight controller and simulates its behavior like a real drone's. MAVROS acts as a bridge between ROS and ArduPilot and enables access to MAVLink topics for drone telemetry and control. RVIZ is used for visualization and QGC is used for controlling the drone and planning missions.

For this project, the Intelligent Quads simulation package [62] is used as a base. This is a very useful resource to get the setup started and running without hassle. The prerequisite process to get all the required software involves the following steps:

- Installing ArduPilot, which also installs required packages like MAVProxy [63]
- Installing the ArduPilot-Gazebo plugin, after which a simple SITL drone can be run in Gazebo
- Installing MAVROS and MAVLink, to send commands to the drone and receive messages from it
- Installing QGC to make it easier to plan missions and send commands to the drone

The IQ_SIM package provides preconfigured drone models with sensors like camera, and integrates with ROS. It also has concise launch files to execute the mavros modules. For this project, the scripts from that package are modified to suit our needs.

The structure of the `drone_with_camera` model folder is similar to that of the `utwente` model in figure 4.7. The default gazebo iris model is used as the base, parameters like its rotor configurations, collisions, joints, inertia, mass, and other physical quantities are left as it is in the IQ_SIM package. To this drone, a downward-facing camera is added by rotating it along the y-axis by 90 degrees. The gazebo ros plugin for camera is used to receive the images as a `/down_cam1/image_raw` topic. The intrinsic camera parameters are left as default. They can be accessed by subscribing to the `/down_cam1/camera_info` topic. Some of the important parameters of this camera are shown in figure 4.9. Other sensors like the IMU can also be added similarly, but it is not used in this project.

Additionally, when using mavros with a gazebo-ardupilot drone, various topics become available, which provide information about the drone's state, sensor outputs and system control. The frequency of these mavros topics is fixed at 4Hz. These topics publish useful information from the drone, such as its status, telemetry, communication channels, positioning data, IMU data, etc. Some of the useful topics are `/mavros/global_position/global`, `/mavros/global_position/local`, and `/mavros/local_position/pose`. The drone's global GPS position in the WGS84 coordinate system is given by the `/mavros/global_position/global`

`Update rate:`	20Hz
`Image width:`	640 pixels
`Image height:`	480 pixels
`Intrinsic matrix:`	$\begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 467.7427 & 0 & 320.5 \\ 0 & 467.7427 & 240.5 \\ 0 & 0 & 1 \end{bmatrix}$

Figure 4.9: Parameters of the downward facing camera

topic and has latitude, longitude and altitude values. It is sourced from the Gazebo GPS plugin. The `/mavros/global_position/local` topic provides the drone's pose (position and orientation) in a local ENU frame converted from the GPS coordinates. Technically, the ArduPilot information is in the NED frame, but mavros converts it to ENU according to the ROS standards. Therefore, this topic is used as the source for GPS sensor data in the later sections. The `/mavros/local_position/pose` topic is obtained from the drone to signify its position in its local ENU frame. Its values are obtained from the flight controller unit (ArduPilot) after it fuses information from its simulated sensors like IMU and GPS.

Another useful topic for this project is the `/gazebo/model_states`. It contains the pose and twist values for all the models in the gazebo world. This is important because it serves as the ground truth data for the drones. The pose information for the drone can be used as its ground truth values to compare with the trajectory produced by the SLAM algorithm.

This model of the drone is then saved to the models folder as in the figure 4.7. The model is then added to the world file so that when the world file is launched, the drone is incorporated in the simulation.

To specify the location of the drone in the environment, the convention followed in this project is as follows. The position of the environment model (like the utwente model) is fixed at `origin = (0 0 0)`, which is the centroid of the model. When ArduPilot is launched, it is launched at a custom location corresponding to `ref = (lat0, lon0, alt0, heading0)`, which corresponds to this origin. Here, the latitude and longitude values are in the WGS84 frame, and the altitude and heading are taken as 0 for simplicity. If the drone is to be launched at a location `pos = (lat1, lon1, alt1)`, that location needs to be converted to the local NED frame so that it matches the coordinate frame of ArduPilot. This conversion can be done using the `pymap3d` library, where the geodetic coordinates of `pos` in the WGS84 frame is converted to the NED frame as `drone_pos = (N_pos, E_pos, D_pos)` with `ref` as the reference point. But since we are using mavros with ROS, the `E_pos` needs to be flipped to match the ROS body frame convention. So essentially, while specifying the drone's position in the world file, the coordinates are specified as `(N_pos, -E_pos, D_pos)`.

After having the gazebo world file configured, it can be launched along with ArduPilot, MAVROS and QGC. When the drone is setup in Gazebo and is publishing the topics, it is ready for the data collection step.

4.4 Creating Datasets

The procedure to collect data from a single drone from the UTwente model is as follows:

1. Launch the ArduCopter simulator using ArduPilot at a custom location of `(52.238067, 6.857023, 0, 0)` which corresponds to GPS coordinates of the centroid of the Gazebo model. For other models, their corresponding GPS coordinates can be used.

2. Launch the `simulation.launch` using ROS. With this, the `utwente.world` file will open, with drone and the `utwente` model setup. This drone connects to ArduPilot using MavProxy.

3. Launch the MAVROS node to access the ROS topics from sensors.

4. Run RVIZ to visualize the camera topic and RQT_GRAPH to visualize the ROS nodes and topics.

5. Run QGC and plan the drone's mission. When QGC connects to the drone's autopilot correctly, the drone shows up on the map in the expected GPS location with a notification that it is "Ready To Fly". A mission can be planned by adding the mission items such as "Mission Start", "Takeoff", "Waypoints" and "Return to Launch". Then upload this mission to the drone and start it. The drone follows the path of the defined waypoints.

6. When the drone is flying over the region of interest, record the ROS topics of camera, GNSS and gazebo model states to a rosbag. This rosbag file will serve as the input for the SLAM algorithm.

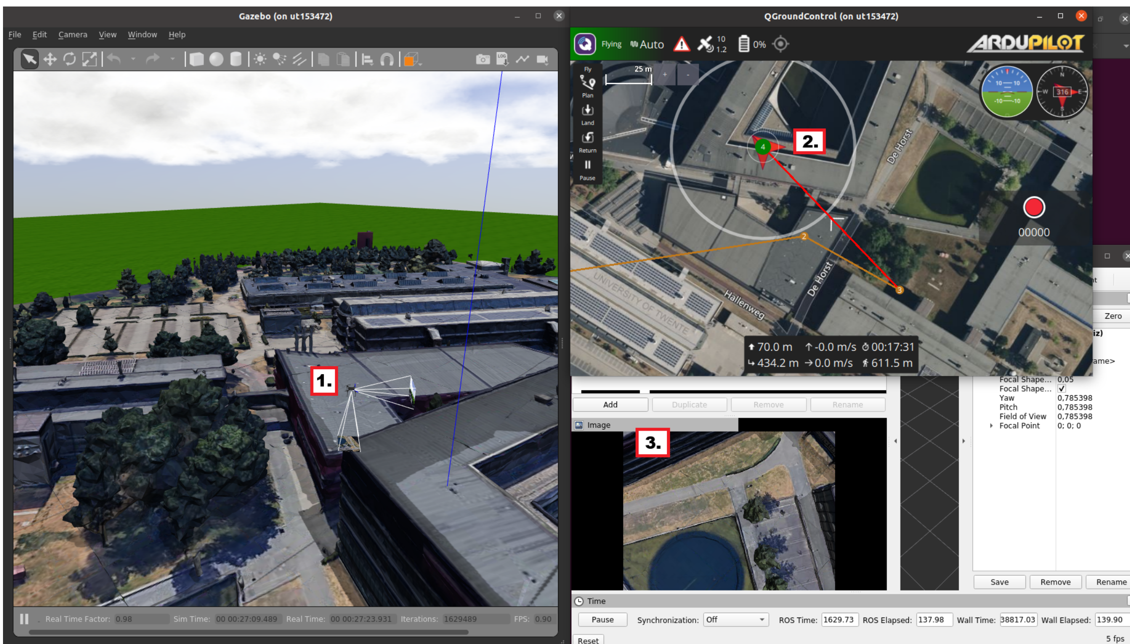


Figure 4.10: 1. The flying drone with a downward-facing camera in the UTwente Gazebo world. 2. Planning the mission path in QGC and viewing its progress. 3. Visualizing the `/down_cam1/image_raw` topic as recorded by the drone.

Figure 4.10 shows this setup running. QGC is used to plan the mission and send commands to the drone. The drone flies around the environment and captures image data from its downward-facing camera. These images can be viewed in RVIZ by subscribing to the ros topic. This figure is used to demonstrate the setup of Gazebo, QGC, and RVIZ while

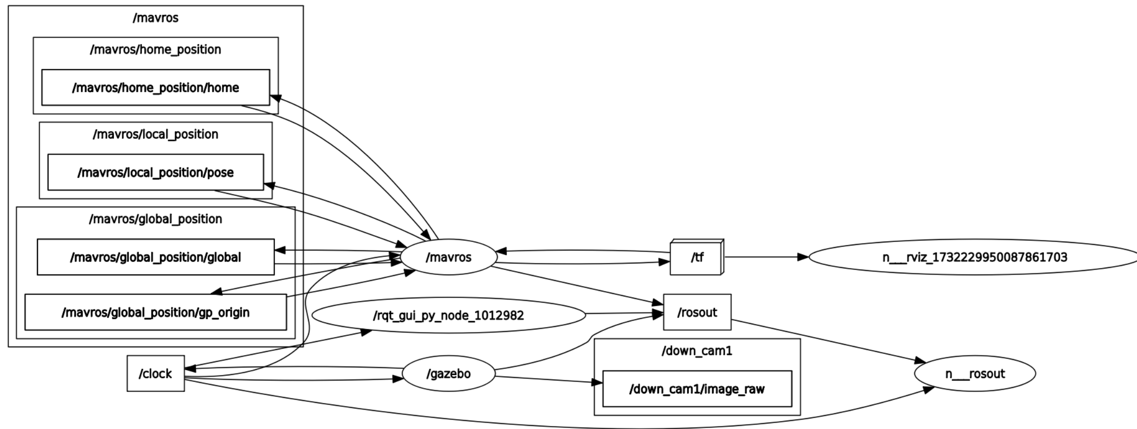


Figure 4.11: Visualization of the ROS nodes and topics during data acquisition

collecting data. More information about the flight path is provided in the following section. The ROS nodes and ROS topics visualized in a graph using RQT_GRAPH is shown in figure 4.11. Only a select few topics that are useful for this project are represented here. The nodes are represented in ovals, while the topics are reporesented in rectangles. The gazebo node supplies the camera images via the ROS topic `/down_cam1/image_raw`. The mavros node interacts with the global and local pose topics. The /tf topic is used to signify the coordinate frame transforms that are taking place. The ROS topics are also being accessed by the RVIZ node for visualization.

Running multiple drones

To run multiple drones in the same environment, multiple done models should be added to the world file. But before this, copies of the drone model are to be made, named as `drone_with_camera_2` and so on. ArduPilot communicates with Gazebo via UDP connections to access the sensor data. These channels need to be unique to each drone. Hence, the `fdm_port_in` and `fdm_port_out` parameters need to be changed in the SDF files for each copy of the drone model. For example, the in and out port values can be 9002, 9003 for drone1, 9012,9013 for drone2, and so on. The ROS topic names for sensor data must also be made unique to each drone, such as `/down_cam2/image_raw`, and so on. To differentiate between the other topics coming from the drone, the drones must be assigned their own ID in the parameter files of the ArduPilot simulator. Similarly, a unique MAVROS instance must also be created for each drone using the MavLink TCP ports. After configuring these parameters, the models can be added to the world file by positioning them at different locations in the environment. A separate instance of ArduPilot must be run for each drone, with a unique TCP address for the output port. These ports need to be added to the TCP link settings in QGC, which will allow it to connect to each drone separately. In QGC, each drone can be selected separately and assigned a mission. The IQ_SIM package is used as a reference for this procedure as well.

This method works sufficiently well to fly multiple drones in a gazebo environment. However, we also need to collect sensor data from the drones at a fixed rate. Running

graphical software such as Gazebo, QGC, RVIZ requires computational power and puts a load on the CPU and GPU, which may affect the real-time behavior of the system if the resources are limited. Though we expect the camera to publish at 20Hz consistently, it was observed that when multiple drones are used the system can get overloaded and the camera topic rate sometimes drops to 3-4Hz. This rate for images is not ideal for a SLAM algorithm on a drone flying at a speed of 8-10m/s. It is also inconvenient to operate the drones and collect data from them if the system is overloaded.

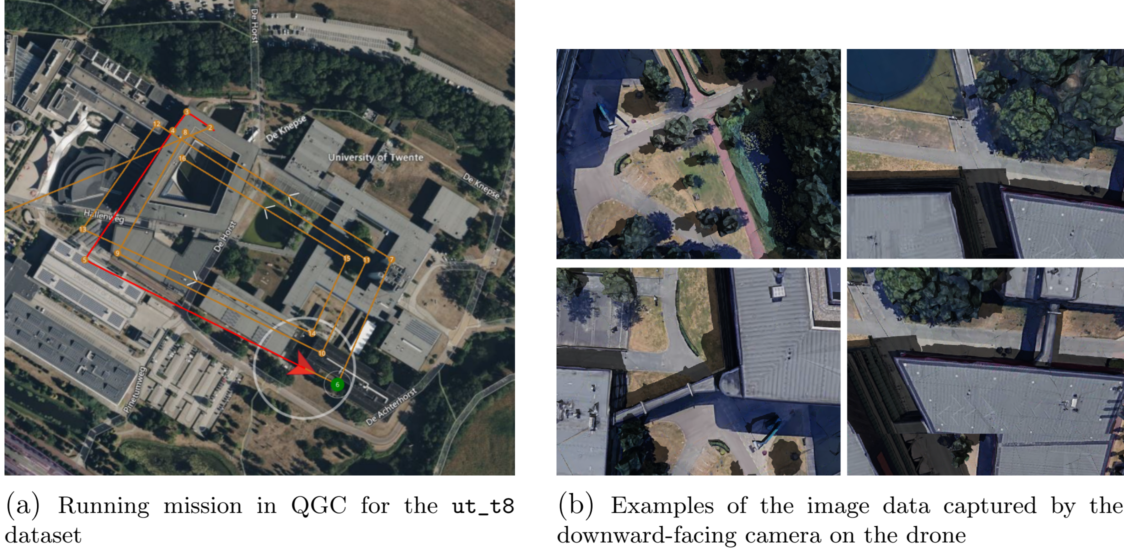
As a workaround for this issue, for this project, a single drone is used to collect data multiple times in the same environment to simulate the presence of multiple drones. Multiple trials were conducted, with varying start location, start times, and path of the drone. Since it is a simulated environment, by starting the data recording at different times, the timestamps can be made to either overlap or not overlap between different trials. This can give the sense of drones running simultaneously, or one after the other. Since QGC is being used to plan the missions, it is easy to ensure that there is sufficient overlap in location between the different trials, and ensure that there are loop closures. This procedure ensures that the simulation runs smoothly in real time and does not affect the update rate of the sensors.

Several trials were conducted to acquire data during the project, by modifying the path of the drone. Some of these trials had less overlap in the drone's paths, or did not have sufficient loops, or did not have ground truth data saved, and these trials are not included in this report. The ones that are used for testing the SLAM algorithm are shown in table 4.1. The Ut_t7 and Ut_t8 trials were conducted on the UTwente model. These dataset each have a drone flying at constant but different altitude levels, so the trajectories do not intersect. Each drone makes three rectangular loops with some area of overlap between them. The Vat_t6, Vat_t7 and Vat_t8 datasets were collected on the Vatican model. The Vat_t7 and Vat_t8 datasets each have a drone flying three loops with some overlap between them. The Vat_t6 data has a drone making a criss-cross loop spanning the whole area covered by the other two drones. In the Vat datasets, the drones do not fly at a constant altitude, and the trajectories of all drones intersect with each other.

Dataset	Duration (s)	Length (m)	Avg. Velocity (m/s)	Size (GB)
`ut_t7.bag`	217.38	1760.578	8.09	3.8
`ut_t8.bag`	234.7	1934.178	8.24	4.1
`vat_t6.bag`	163.65	362.373	2.21	2.8
`vat_t7.bag`	81.55	365.463	4.48	1.4
`vat_t8.bag`	101.04	434.105	4.29	1.8

Table 4.1: Details of the datasets from the UTwente and Vatican models

An example of acquisition of the `ut_t8` dataset is shown in figure 4.12. Figure 4.12a shows the drone flying through the waypoints along its mission path in QGC, and figure 4.12b shows examples of the images captured by this drone. Figure 4.13 shows an example of the ground truth data of the Ut_t7 and Ut_t8 drones, collected from the gazebo model states topic.



(a) Running mission in QGC for the `ut_t8` dataset

(b) Examples of the image data captured by the downward-facing camera on the drone

Figure 4.12: Data acquisition for the `ut_t8` dataset

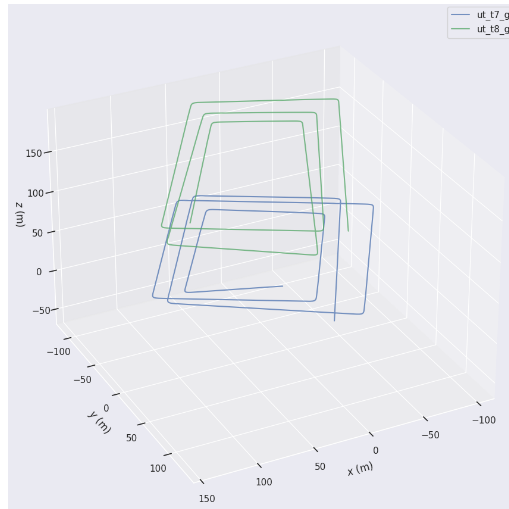


Figure 4.13: The ground truth pose of the Ut_t7 and Ut_t8 datasets plotted together

Section 2: Running the SLAM Algorithm

4.5 Comparing CCM-SLAM and COVINS

Based on the literature survey, it was found that CCM-SLAM and COVINS are suitable for performing collaborative visual-SLAM. Though COVINS is newer, since it does not offer a purely monocular method, it was decided to test both these algorithms with a benchmark dataset, to see if CCM-SLAM can be a viable option for this project.

CCM-SLAM and COVINS-G (with ORB-SLAM3 front-end) are tested on the data from three UAVs in the EuRoC MAV dataset [64]. For CCM-SLAM, only the monocular camera input is given, and for COVINS, an IMU input is also given along with the camera

data. The trajectory plots obtained from the keyframe pose information and the merged 3D point cloud results are shown in figure 4.14. The trajectory for each drone, namely MH-01, MH-02 and MH-03, are represented using different colors labeled as euroc_012_0_traj, euroc_012_1_traj and euroc_012_2_traj respectively. To evaluate the performance, the concatenated trajectory from all three drones are compared with the MH ground truth data using the evo ape (absolute pose error) tool [65]. The ATE RMSE values and RMSE percentage with respect to trajectory length is shown in table 4.2. The value of RMSE increases with increase in length of trajectory in SLAM due to accumulated drifts. Hence the percentage RMSE is calculated with respect to the total trajectory length, in order to be consistent between different trajectories.

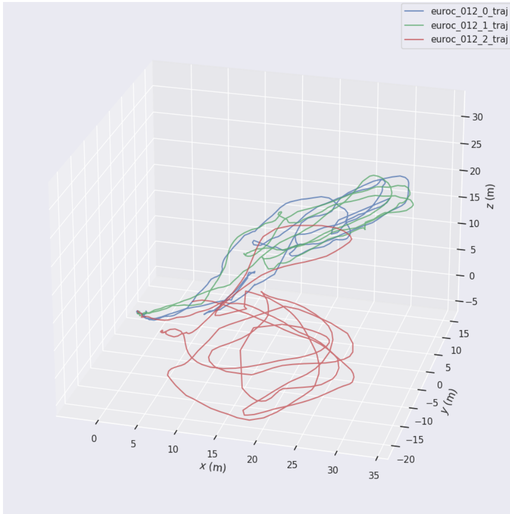
Metric	**CCM-SLAM**	**COVINS-G**
RMSE	0.0388	0.0693
Percentage RMSE	0.015%	0.027%
Points in pointcloud	38,748	26,409
Outliers in pointcloud	More	Less

Table 4.2: Comparing performance of CCM-SLAM and COVINS-G with MH datasets

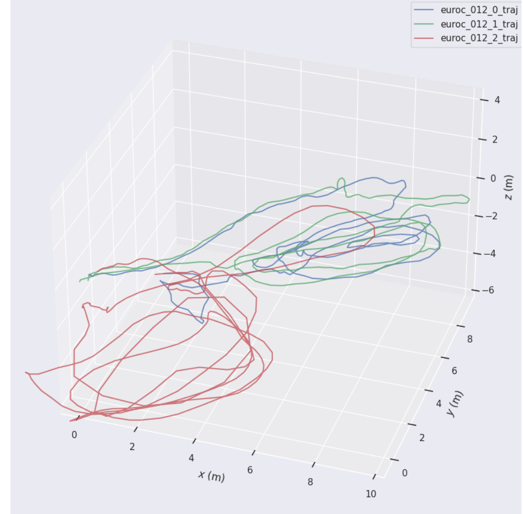
There are a couple of things to be noted here. First, the RMSE values differ from those in the source papers. This is because SLAM performance varies between different machines, and also between different runs on the same machine. Factors like computational resources available, system configuration, processing power, threading behavior, and CPU/GPU utilization can affect the performance of the SLAM algorithm, and result in slightly different results between machines. The SLAM output can vary between different runs on the same machine because the keyframes, features, loop detections, etc. extracted by the algorithm can vary between different runs. Hence, RMSE values need to be computed over multiple runs to get an average value. Second, we expect that the performance of COVINS-G to better than that of CCM-SLAM, but the table shows otherwise. While COVINS-G generally does show a better performance than CCM-SLAM, the table depicts one such instance where the values from CCM-SLAM are slightly better. This is to show that CCM-SLAM can indeed be used for purely monocular cases as an alternative for COVINS.

With respect to the point cloud, it is seen from the table that CCM-SLAM has more number of points in the point cloud. But there are also more outliers, and the bounding box is quite large compared to that of COVINS. The point cloud densities and their distribution is shown in figure 4.15. To get this, the point clouds are first aligned and scaled by point-picking registration in CloudCompare [66]. Then their volume densities are calculated with a radius of 30, and the density values are presented in points per volume. It can be seen that CCM-SLAM has some denser clusters while COVINS is sparser in comparison. However, the point cloud from COVINS is more refined with edges clearly visible. This is as expected, because from literature we know that COVINS has additional steps for redundancy removal and optimization. Because CCM-SLAM does not have this, there will be more outliers and unfiltered regions.

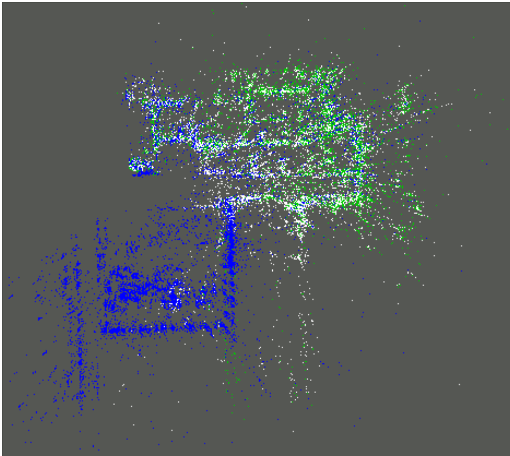
Comparing the two algorithms based on these observations, it was concluded that CCM-SLAM can be a suitable option for our application, since it can produce comparable results. But it may need some modifications to overcome some drawbacks like drift, arbitrary scaling, and outliers in the point cloud.



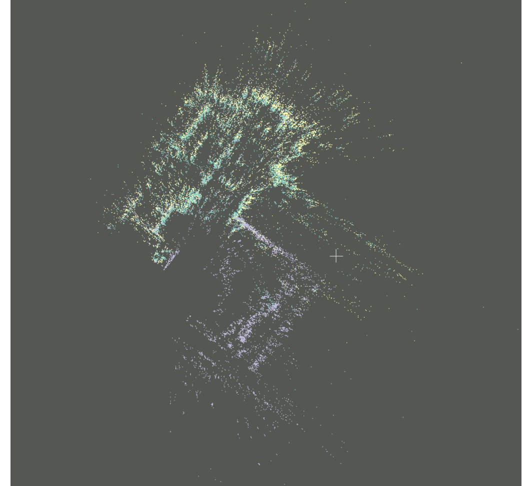
(a) Trajectory plots after running CCM-SLAM



(b) Trajectory plots after running COVINS-G



(c) Merged point cloud after running CCM-SLAM



(d) Merged point cloud after running COVINS-G

Figure 4.14: Trajectory plots and Point Clouds from running CCM-SLAM and COVINS-G on Euroc MAV MH-01, MH-02, MH-03 data

4.6 Configuring CCM-SLAM for Simulated Data

To allow CCM-SLAM to operate on the simulation data, some configuration and launch files need to be written. These include:

- A `simulation.yaml` file consisting of the camera parameters
- The `Server.launch` file to start the server node that connects to the clients, assigns each of them a client ID, and establishes the odometry frames for the server maps.
- A `Client_n.launch` for each of the n clients to start the client nodes that subscribe to the camera topics and establishes a local odometry frame.

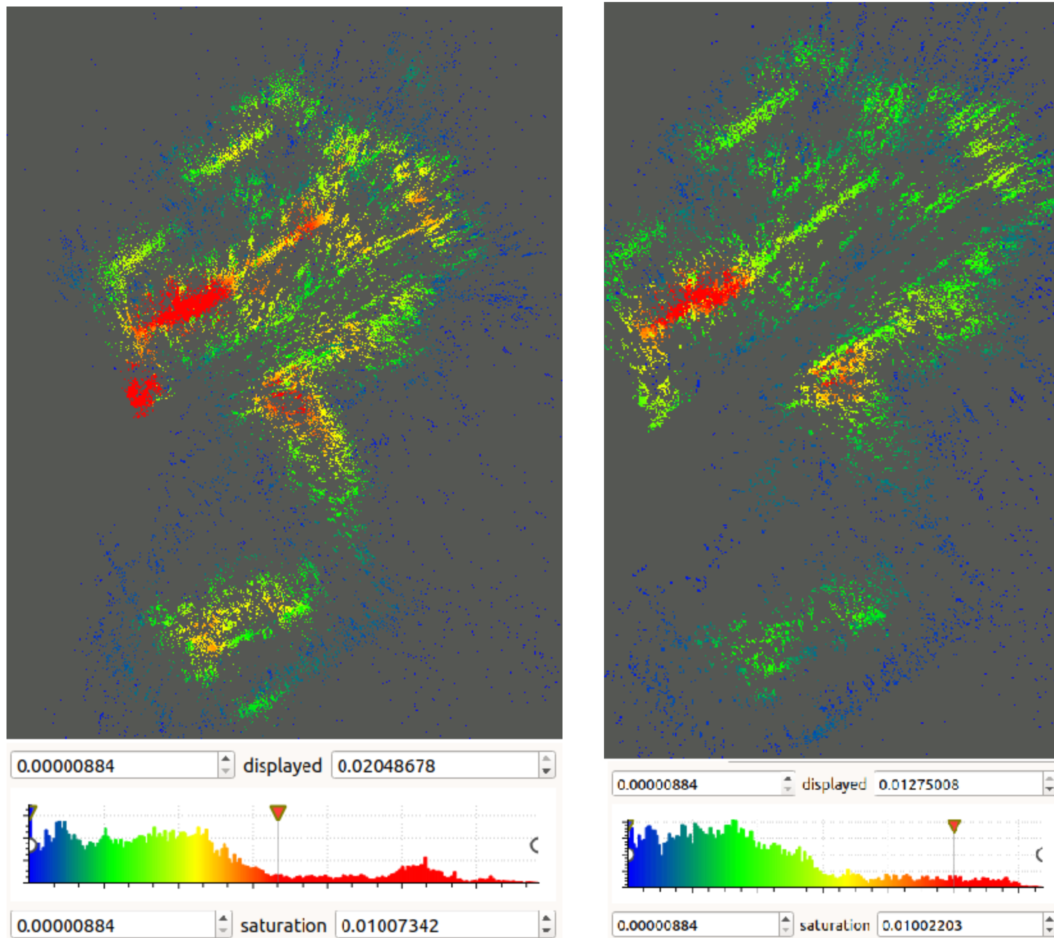


Figure 4.15: Point cloud densities and distribution of the CCM-SLAM (left) and COVINS (right) outputs

- A `config.yaml` file that is used to specify the paramters and numbers used during different stages of the SLAM algorithm.
- A `ccmslam.rviz` file that contains the settings parameters for viewing the maps in RVIZ

Camera parameters

The camera parameters need to be known by the algorithm to accurately extract features from the images. The intrinsic camera parameters are obtained from the table 4.9. The camera distortion parameters are all 0, since no distortion is added to the camera model in the simulation. The camera fps (frames per second) is 20. These values can also be verified by subscribing to the topic `/down_cam1/camera_info`. Is there is any difference between the body frame and the camera frame, that transformation matrix is also to be added here. However, in the simulation environment, it is assumed that the camera frame aligns with the body frame.

Odometry frames

These odometry frames are used to initialize the local or server map frames, which are most useful during visualization. To establish these, `static_transform_publisher` nodes are used, whose pose is of the format `x y z yaw pitch roll` In this project, the client and server maps are initialized at the pose `(0 0 0 0 0 -3.142)` for simplicity. The roll value is made -3.142 since a downward facing camera is used. The SLAM algorithm will adjust the frames based on loop closure. The position values for the clients can be changed if required, to make visualization in RVIZ easier.

CCM-SLAM parameters

The `config.yaml` file is used to describe parameters required for timing, ORB feature extraction, tracking, mapping, communication, place recognition and optimization modules of the SLAM algorithm. It is also used to specify the viewer parameters. The authors of CCM-SLAM have adjusted these parameters based on the tests they conducted on their benchmark test data. But some of these parameters were not optimal for the datasets in this project. To investigate which values should be used for the datasets from the simulation environment, some tests were performed, which are discussed in Chapter 5.

Some of the parameters of interest for this project are:

- `Mapping.LocalMapSize`: The local map of the agent has N keyframes. Default value being used is 50.
- `Mapping.LocalMapBuffer`: This is a B number of buffer keyframes that are stored on the agent before they are removed from memory. This is needed in case there are communication losses. Default value is 20.
- `Mapping.RecentKFWindow`: A window of the most recent W keyframes are not included in the keyframe rejection process while optimizing for redundancy. Default value is 20.
- `Mapping.RedThres`: This is the threshold for keyframe redundancy. A value of 1 means no keyframes are rejected. Default value is 0.98.
- `Viewer parameters`: Viewer parameters like scale factor, display keyframes, etc. are used to improve visualization.

Visualization parameters

The rviz config file is used to configure the RVIZ environment to display the outputs from the SLAM algorithm. It subscribes to the topics of the server and client maps, marker points, and keyframe poses and displays them according to the coordinate frame described earlier. Other viewing parameters like color, scale, etc. are also assigned.

4.7 Workflow

CCM-SLAM is installed in a docker container, in order to make compilation easier and satisfy all the library versions. Then the workflow followed in this project is detailed below. The documentation in the CCM-SLAM github page is used as a reference.

1. Source the environment and start `roscore`.
2. Start the server node by launching the `Server.launch` file. It is then ready to connect to clients.
3. Start each of the clients in separate terminals by launching the `Client_n.launch` files. The agents will initialize the camera and SLAM parameters and connect to the server with a unique Client ID.
4. Play the rosbag files containing camera data by specifying the topic name according to the configuration in their respective client launch files.
5. Since an analysis will be later performed on the point clouds outputs, the server map topics are recorded into a rosbag file.
6. The RVIZ environment is loaded using its configuration file to view the SLAM process in real-time.
7. After bundle adjustment, the keyframe poses are written to CSV files. These are also saved to analyze later. The pose file corresponding to each agent is stored in separate files, and should be concatenated in case of multi-agent systems. An example of single-agent trajectory saved is `ut_t7_traj.csv`, and for multi-agent is `ut_t7t8_traj_cat.csv`.
8. The point cloud topic saved in the rosbag file can be converted to a PCD (Point Cloud Data) format using the `pcl_ros` library. This PCD file can be used for point cloud analysis in CloudCompare.
9. Perform this procedure for single-agent and multi-agent scenarios and compare their performance.
10. Improve the results by using the GNSS data by the process of georeferencing.

In this project, a comparison is made between single-agent SLAM and multi-agent SLAM. For the single-agent scenario, there is one trajectory and one point cloud file saved for each agent. For the multi-agent scenario, there is a trajectory file for each agent that can be concatenated together, but only one point cloud file that contains the map from all the agents.

Figure 4.16 shows the RVIZ visualization of two agents, `Vat-t7` and `Vat-t8` running collaborative SLAM. The keyframe poses from the two agents are represented by different colors, but there is just a single 3D map point cloud which contains the merged map from the two drones. In this case, the algorithm has found keyframe matches between the two drones, as represented by the red lines, and this is where new loop closures are found and the map is merged.

4.8 Evaluation

The evaluation metric used in this project for the trajectories of keyframe poses is the RMSE of Absolute Trajectory Error (ATE) as described in Chapter 2. The tool evo_ape [65] is used for calculation. It is to be noted that RMSE values can increase with an increase in trajectory length due to accumulation of errors. To account for this, the RMSE

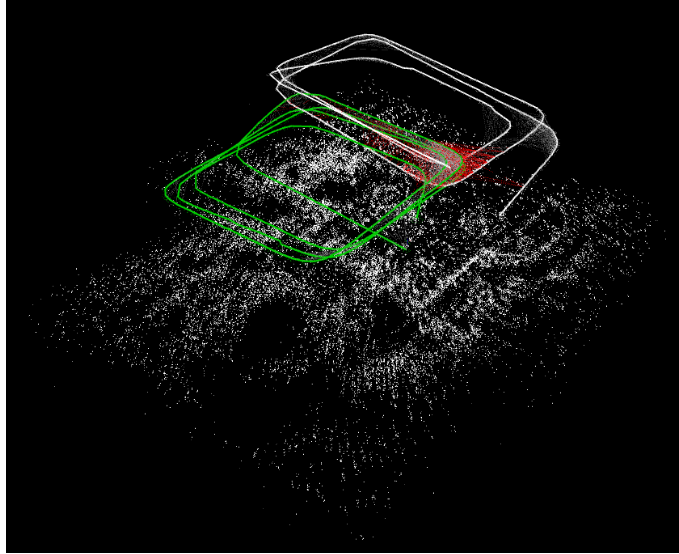


Figure 4.16: RVIZ visualization of collaborative SLAM on Vat-t7 and Vat-t8 datasets

is expressed as a percentage of the trajectory length. The results of this evaluation is discussed in Chapter 5.

For single-agent case: Each agent's trajectory (e.g. `ut_t7_traj.csv`) is compared to its ground truth (e.g. `ut_t7_gt.csv`) to get individual RMSE values.

For multi-agent case: The overall trajectory (e.g. `ut_t7t8_traj_cat.csv`) is compared to ground truth corresponding to all agents together (e.g. `ut_t7t8_gt_cat.csv`) to get an RMSE values for the collaborative SLAM trajectory.

These values are then compared by calculating the percentage RMSE with respect to the total trajectory length.

Qualitative evaluation of the trajectories is done by plotting them in a 3D plot. The keyframe poses output by the SLAM is in the TUM format `(timestamp x y z q_x q_y q_z q_w)`, which are the position and orientation (quaternions) values. The quaternions are converted to a rotation matrix (as described in Chapter 2) and this is used to orient the position values to give a trajectory. In this project, the evo_traj tool is used to plot the trajectories.

Point clouds: The 3D reconstruction maps from the SLAM algorithm are represented in the form of a point cloud. They can be visualized and evaluated in CloudCompare. For the ground truth, a pointcloud of the 3D model can be exported as a PCD file from Blender. The output point clouds of the SLAM algorithm are compared against the ground truth qualitatively. Quantitative comparison is not as straight forward as for the trajectories, since monocular SLAM scales its outputs arbitrarily. However, point cloud registration using point-picking, or the ICP algorithm can be used to align the cloud with its ground truth, and that results in an RMSE value for the residuals.

4.9 Georeferencing using GNSS data

The outputs from monocular SLAM suffer from scale ambiguity and do not have a common coordinate frame, since each agent uses its own local frame. To improve this, we can use

the data from the GNSS sensor on the drone. Using the latitude, longitude and altitude values from the GNSS, the real-world scale can be estimated, which then also improves the orientation issue. This process of GNSS integration can be fused into the SLAM algorithm, or be done as a post-processing step. In this project, the latter is carried out.

The topic `/mavros/global_position/global` provides the GNSS values in the global frame. But since we are using pose values for the trajectory, we also need the pose values for the GNSS data in the local frame. For this, we can use the topic `/mavros/global_position/local`, where the conversion from the world frame to the local frame has already been done by mavros. A **Sim(3)** transformation is done to the trajectory pose using a transformation matrix **T** which has a scale **s**, rotation **R** and translation **t**. **T** is obtained by aligning the trajectory to the pose of the GNSS data, and taking the transformation values that minimize the RMSE. The tool `evo_ape` can be used for obtaining the transformation matrix. The newly obtained trajectory, `traj_new` can then be compared against the ground truth, whose results are discussed in the next chapter. The process of georeferencing is done for the entire trajectory at the end of the run for each trial, using an automated script.

If $GNSS_{global}$ is the GNSS values in the WGS84 frame and **G** is the transformation to the local ENU frame, $GNSS_{local}$ is obtained by:

$$GNSS_{local} = \mathbf{G} \cdot GNSS_{global} \tag{4.1}$$

The quaternions in the pose values are kept the same, since they give the required orientation values. The position values need to be transformed using the transformation matrix. If **T** is the Sim(3) transformation between $GNSS_{local}$ and $Traj_pos$ obtained using the alignment method to minimize RMSE, the new trajectory $Traj_{new}_pos$ is obtained by:

$$Traj_{new}_pos = \mathbf{T} \cdot Traj = \begin{bmatrix} s\mathbf{R} & t \\ 0 & 1 \end{bmatrix} [Traj_pos] \tag{4.2}$$

Point clouds alignment

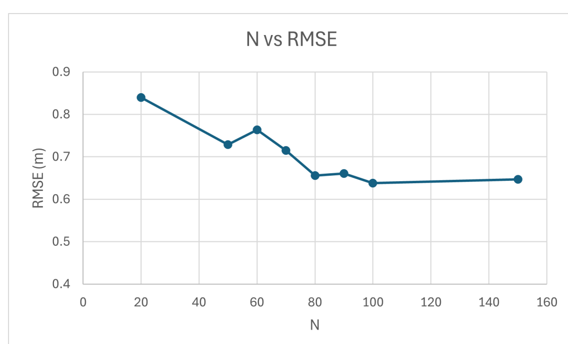
The point cloud outputs from SLAM can also be georeferenced using the transformation matrix. However, the same transformation matrix as above cannot be used. The rotation matrix remains the same as above, but an additional 90 degree rotation has to be applied to account for the difference between mavros and gazebo frames. For the scale factor, the scale of the RVIZ viewer needs to be taken into account and adjusted. The translation should also be adjusted depending on the positioning of the 3D model in the ENU frame of the gazebo world.

Chapter 5

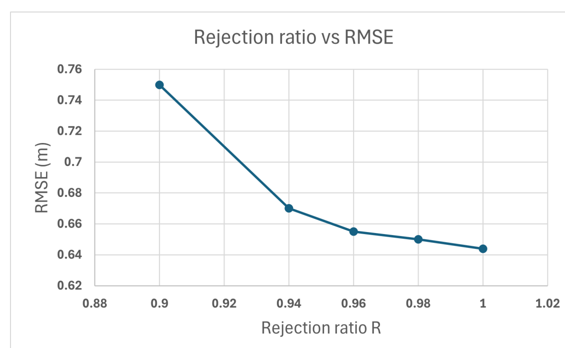
Results and Discussion

This chapter discusses the results obtained during the course of the experiments conducted during the project. First, the RMSE results of varying SLAM configuration parameters are discussed. The trajectory results from the UTwente and Vatican datasets are discussed both qualitatively and quantitatively. Lastly, the 3D reconstruction results from the SLAM algorithm are discussed.

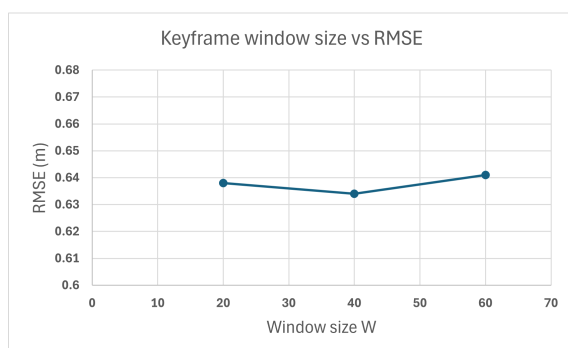
5.1 Varying Configuration Parameters



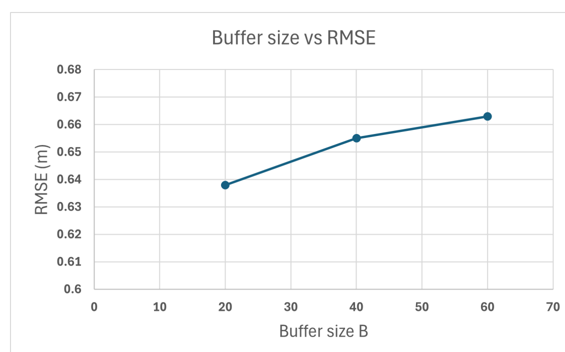
Local map size N vs RMSE (m)



Keyframe rejection ratio R vs RMSE (m)



Keyframe window size W vs RMSE (m)



Buffer size B vs RMSE (m)

Table 5.1: Comparing mapping parameters against RMSE

CCM-SLAM requires some parameters to be tuned for the algorithm to adapt to the

environment and give accurate results. Some tests were conducted to evaluate the performance of the algorithm, by varying the parameter values and calculating the RMSE of the acquired trajectory against the ground truth. For these tests, CCM-SLAM is run on the single agent in the `ut_t7.bag` dataset. The results from these tests are presented in table 5.1.

The local map on the agent is limited to N keyframes to save computational resources, and the rest are passed on to the server. Increasing N and having a larger map increases accuracy of the map and helps to reduce drift But a very large map may not be beneficial in case of limited resources and result in reduction in real-time performance. Hence an optimal value must be chosen. As seen in the N vs RMSE graph, the RMSE indeed reduces as N increases, but after a certain number, it does not change much. Based on this, a value of 100 is chosen for N.

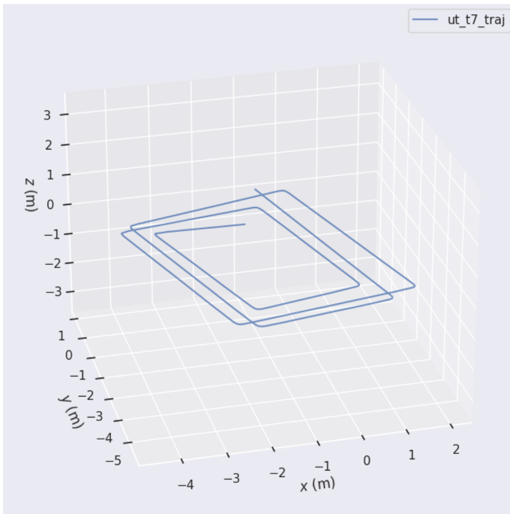
Keyframe rejection is an important step in removing redundancy in case there are multiple keyframes in the same location. Increasing the rejection ratio means lesser number of keyframes of good quality to process, which can reduce the time required for bundle adjustment. However, as seen in the graph of R vs RMSE, reducing the ratio below 0.95 starts to reduce the quality of the map. A value of 0.98 is chosen for R.

The keyframe window consists of the most recent keyframes that are excluded from rejection during optimization process. With the local map size as 100, this window size is varied, and it is seen that the RMSE does not vary much. This is because the SLAM algorithm is optimized so that even by considering a small window, the algorithm should still maintain its accuracy.

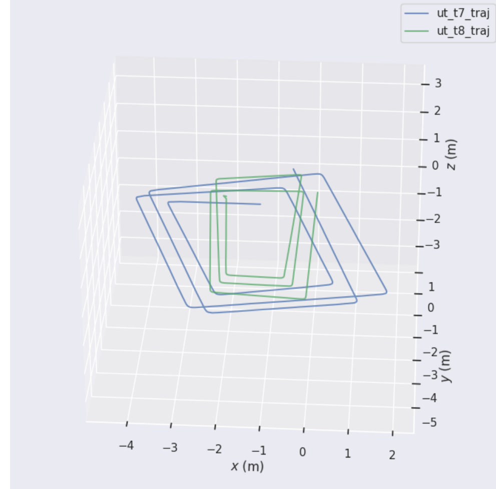
The buffer size B refers to the extra keyframes that are stored on the local agent, to ensure no communication losses between the client and the server. There is a slight increase in the RMSE observed as the buffer size increases. This is because the older frames may be used for optimization and affect the accuracy of estimating the newer data. But this parameter is more important when multi-agent system is in use, since lesser values may lead to inaccurate tracking. Hence an optimum value like 40, which balances the RMSE and map quality is chosen.

5.2 Results of the UTwente Datasets

First, CCM-SLAM is run on the `ut_t7` and `ut_t8` datasets individually as single-agent SLAM, and their trajectories are plotted. For example, the trajectory of `ut_t7` is shown in figure 5.1a as `ut_t7_traj`. However, since no map-merging takes place between these maps, the trajectories of the two agents are in different frames and scales, as shown in figure 5.1b. Then, the collaborative SLAM is run with two agents, where the maps are merged, new loop closures are found and bundle adjustment is performed on the whole map. With that, the maps from both the agents are adjusted such that they are aligned with one another, and appears like the ground truth. This is shown in figure 5.2, where the output of collaborative SLAM for the two clients (IDs 0 and 1), `ut_t7t8_0_traj` and `ut_t7t8_1_traj`, is compared to the ground truth of `ut_t7_gt`. But even then, the scales and overall orientation of the output does not match that of the ground truth.

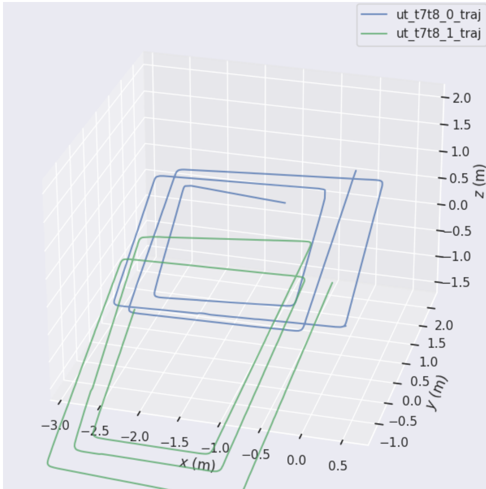


(a) Trajectory of Ut-t7

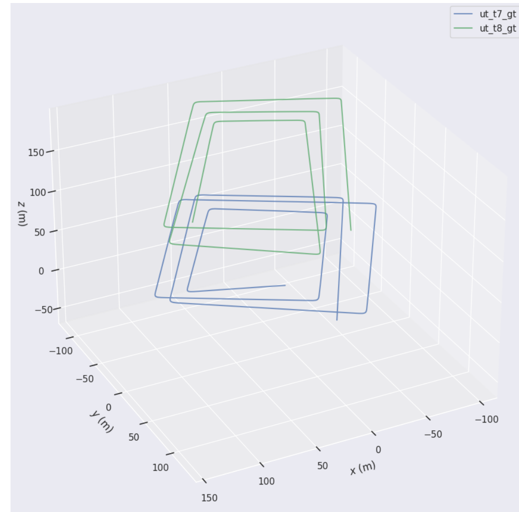


(b) Trajectories of Ut-t7 and Ut-t8 run without collaboration

Figure 5.1: Trajectory results from SLAM run on individual agents of UT dataset



(a) Trajectories of Ut-t7 and Ut-t8 run with collaboration

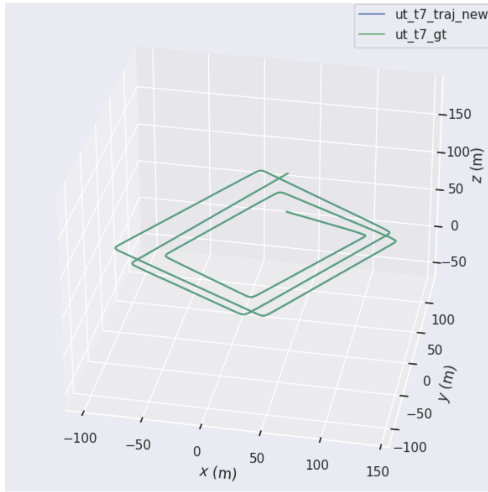


(b) Ground truth of Ut-t7 and Ut-t8

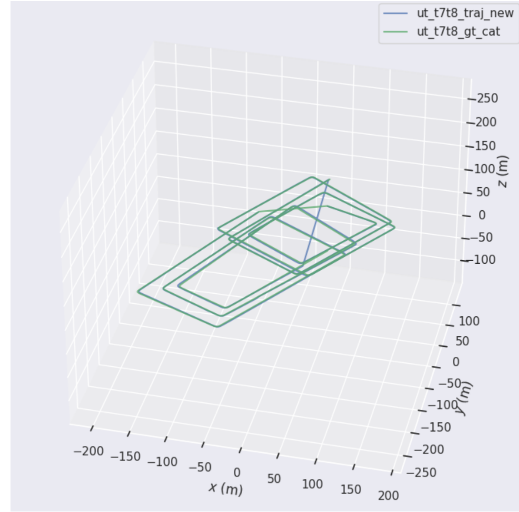
Figure 5.2: Trajectory results from collaborative SLAM run on UT dataset

To improve the results and adjust for the scale and orientation mismatch, georeferencing is done to the trajectories, as described in the previous chapter. For the single-agent SLAM, the trajectories from each agent are georeferenced separately with their GNSS values. A georeferenced Ut-t7 trajectory looks like the figure 5.3a, and it is compared to the ground truth in the same plot. It can be seen that the trajectory is scaled and oriented in the right way now. The same thing is done for Ut-t8, and the results of both drones are superimposed and compared to the ground truth, as shown in figure 5.3b. Compared to figure 5.1b, the difference is clear, and the trajectories are oriented well. Then trajectory from collaborative SLAM is georeferenced as well. Here, the whole trajectory of both drones is georeferenced with the whole ground truth at once. The obtained result is shown in figure 5.4. This figure also shows good alignment with the ground truth. To simply look at the graphs qualitatively, the results from georeferenced collaborative and

non-collaborative SLAM seem similar, but this is not the case, there are some variations. This can be evaluated using RMSE, and then by examining the 3D maps.



(a) Georeferenced Ut-t7 trajectory compared with its ground truth



(b) Georeferenced Ut-t7 and Ut-t8 compared to their ground truth

Figure 5.3: Georeferenced trajectories from Ut-t7 and Ut-t8 running non-collaborative SLAM

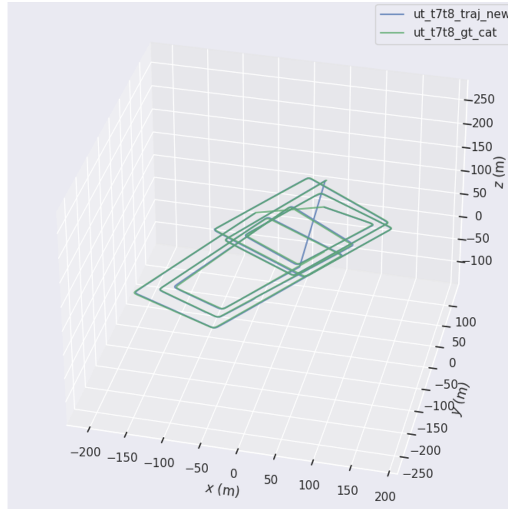


Figure 5.4: Georeferenced trajectories of Ut-t7 and Ut-t8 run on collaborative SLAM

RMSE calculations are made for the trajectories from single-agent SLAM individually, and then for the multi-agent SLAM trajectory. The results are shown in table 5.2. The RMSE value of the collaborative SLAM should be ideally comparable, or even less than the single-agent SLAM, but this is not the case here. This is because errors accumulate over time and distance in SLAM. These datasets contain trajectories going over large distances (>1500m). To account for this, the percentage RMSE is calculated, and the value from collaborative SLAM is indeed lower than that of both the individual trajectories. This behavior is further examined in the next section, where the Vatican datasets are used and their trajectory lengths are much smaller.

Dataset	Trajectory length (m)	RMSE(m)	Percentage RMSE
Ut-t7 (single-agent)	1759.382	0.662	0.037%
Ut-t8 (single-agent)	1932.4	0.751	0.038%
Ut-t7t8 (multi-agent)	3880.908	1.33	**0.034%**

Table 5.2: Comparing the RMSE values of Ut-t7 and Ut-t8 run with and without collaborative SLAM

5.3 Results of the Vatican Datasets

Similar tests as described in the previous section are performed on the Vat-t6, Vat-t7 and Vat-t8 datasets, and their results are shown below. The difference here is that there are three agents instead of two, and the trajectories, duration of flight are smaller. The ground truth data from all three drones are shown in figure 5.5. It is to be noted that this dataset has all drone flying in a non-planar path, and the trajectories of all drones intersect with each other.

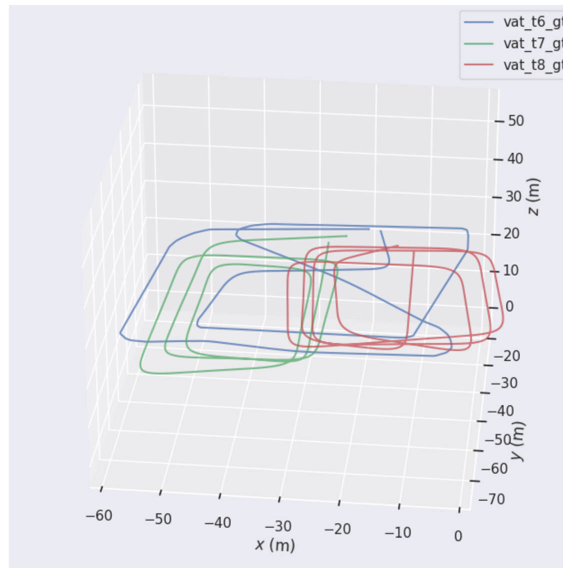


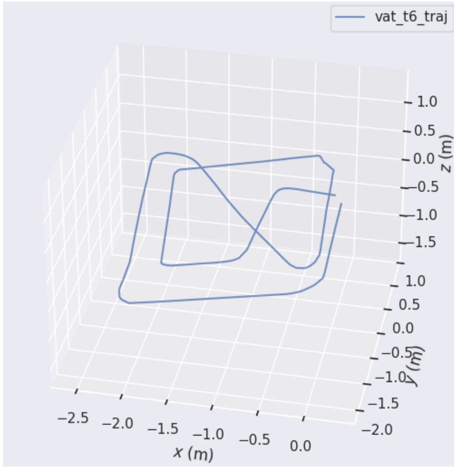
Figure 5.5: Ground truth data of Vat-t6, Vat-t7 and Vat-t8

CCM-SLAM is run on each of the datasets individually without collaborative SLAM. The trajectory of one drone, Vat-t6, is shown in figure 5.6a, and the corresponding georeferenced trajectory is shown in figure 5.6b. The superimposed trajectories of all three datasets are shown in figure 5.7a. They are not oriented and aligned in the same manner. When georeferenced, they look like figure 5.7b.

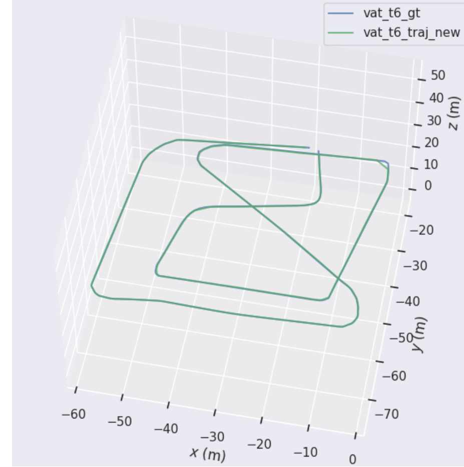
Then collaborative-SLAM is run on all three agents, and the trajectory results are shown in figure 5.8a. As expected, these trajectories are well aligned with each other. When georeferenced, it can be seen that the trajectories align reasonably well with the ground truth, as in figure 5.8b.

The performance is as expected, and similar to that of the UT datasets. However, some differences can be seen between the SLAM outputs and the ground truth when the collaborative SLAM is run. CCM-SLAM is able to perform well when the drones are flying

at different altitudes and there is no intersection of trajectories. But when there is are many drone paths passing through the same area, some keyframes from some agents may go missing due to redundancy checks, and affect their trajectories. For this, some of the configuration parameters had to be tuned to get a satisfactory result, for example, the buffer size had to be increased to accommodate for increasing number of agents.

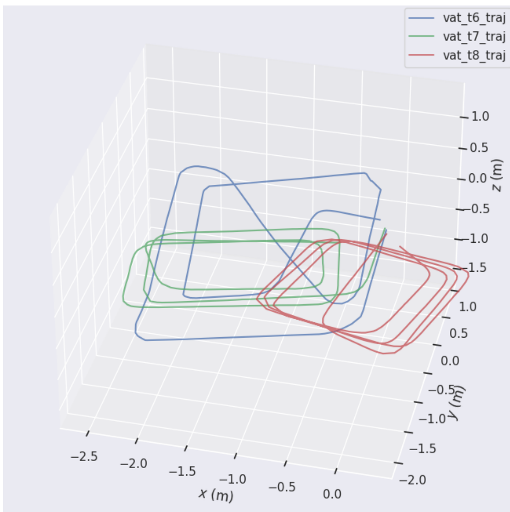


(a) Trajectory of Vat-t6 run on non-collaborative SLAM

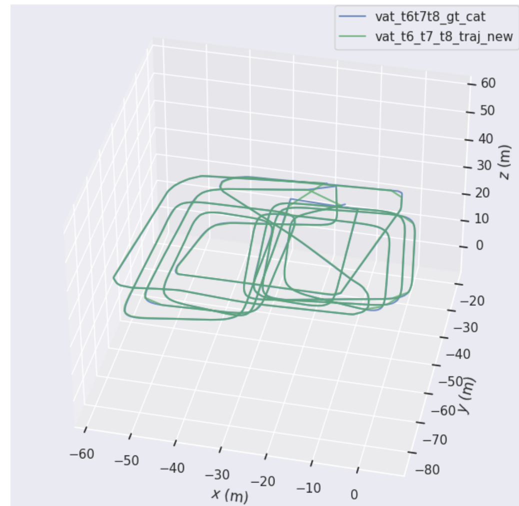


(b) Georeferenced trajectory of Vat-t6 comapred to ground truth

Figure 5.6: Non-collaborative SLAM run on Vat-t6 dataset



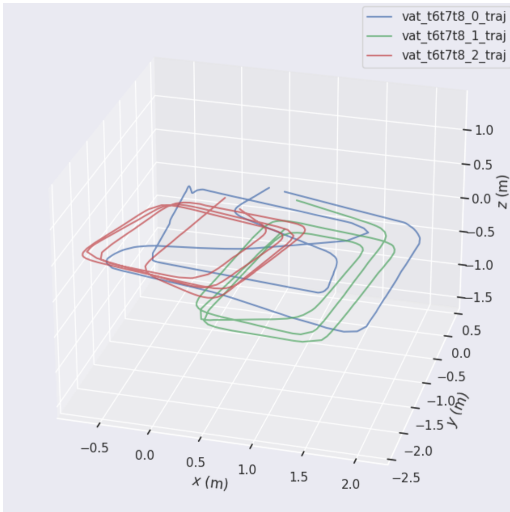
(a) Superimposed trajectories of Vat-t6, Vat-t7, Vat-t8 run on non-collaborative SLAM



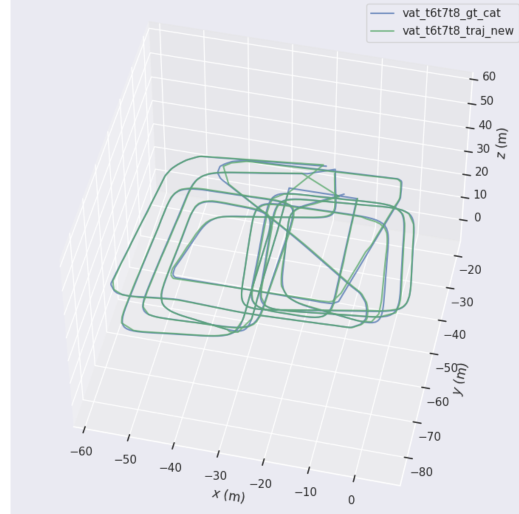
(b) Georeferenced trajectory of Vat-t6, Vat-t7, Vat-t7 run on non-collaborative SLAM, compared to ground truth

Figure 5.7: Non-collaborative SLAM run on Vat-t6, Vat-t7, Vat-t8 datasets

The RMSE values are calculated for quantitative evaluation, and the results are shown in table 5.3. The RMSE value for the collaborative SLAM is not less than the individual agents, but the percent RMSE is the lowest, as expected. These RMSE values are also lesser than the values obtained with the UT dataset, and this is understandable since the trajectory lengths are higher.



(a) Superimposed trajectories of Vat-t6, Vat-t7, Vat-t8 run on collaborative SLAM



(b) Georeferenced trajectory of Vat-t6, Vat-t7, Vat-t7 run on collaborative SLAM, compared to ground truth

Figure 5.8: Collaborative SLAM run on Vat-t6, Vat-t7, Vat-t8 datasets

Dataset	Trajectory length (m)	RMSE(m)	Percentage RMSE
Vat-t6 (single-agent)	362.363	0.265	0.07%
Vat-t7 (single-agent)	365.789	0.412	0.11%
Vat-t8 (single-agent)	434.349	0.355	0.08%
Vat-t6t7t8 (multi-agent)	1184.286	0.513	**0.04%**

Table 5.3: Comparing the RMSE values of Ut-t7 and Ut-t8 run with and without collaborative SLAM

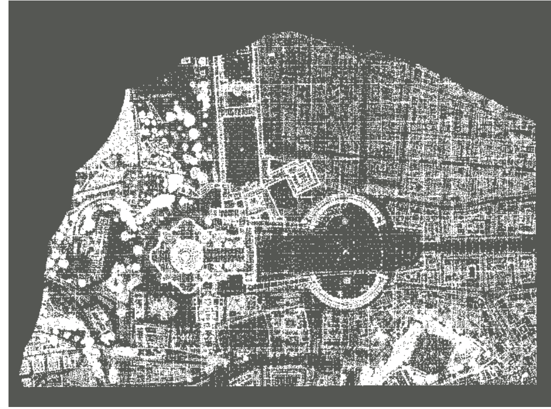
5.4 3D Reconstruction Results from Point Clouds

The ground truth point clouds are obtained directly from the 3D Blender model. The ground truth for the UTwente and Vatican datasets are shown in figure 5.9. In the following pages, all the point cloud results are not presented, only the ones that need to be discussed.

UTwente Dataset

First, considering the case where the SLAM algorithm was run on the UT dataset. When the SLAM algorithm is run on the agents individually without collaborative SLAM, their maps are not aligned, as seen in figure 5.10a. When georeferenced using GNSS data, they will be aligned, as shown in figure 5.10b. In the case of collaborative SLAM, the output will contain the merged map from both the drones, as shown in figure 5.11a. This point cloud is also georeferenced to account for the scale difference in figure **??**. It is to be noted here

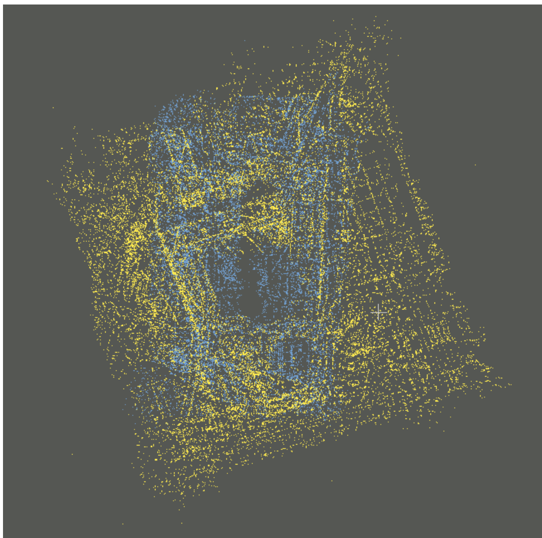
(a) Ground truth point cloud of UTwente model



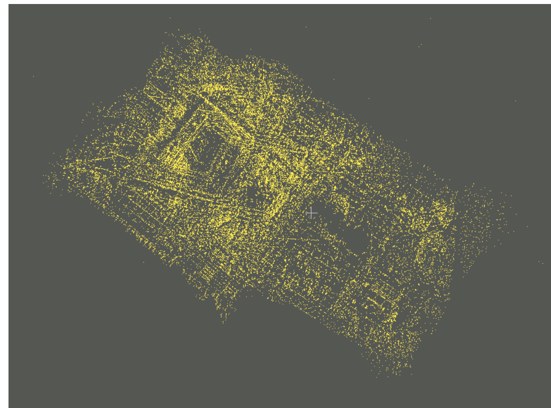
(b) Ground truth point cloud of Vatican model

Figure 5.9: Ground truth point clouds from Blender

that the SLAM algorithm assigns a color for the map from each agent in the first figure, but we do not use the RGB value for further processing. From preliminary examination, it is seen that the map result from the collaborative SLAM is properly merged with clear features visible. The one without collaborative SLAM is not fully merged, only aligned, so it looks noisier.



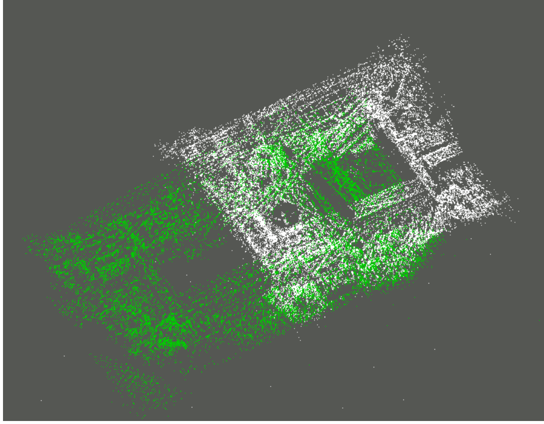
(a) Point clouds obtained from the individual agents without collaborative SLAM



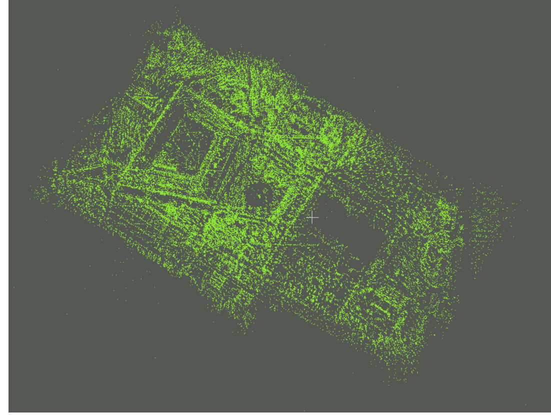
(b) Georeferenced maps of both individual agents after non-collaborative SLAM

Figure 5.10: Results from non-collaborative SLAM on Ut-t7 and Ut-t8 before and after georeferencing

Following georeferencing, the two maps are compared against the ground truth. Since the transformation matrix used is correct, the point clouds easily align with the ground truths, as shown in figure 5.12. Even in this graph, we can see that the first map is not aligned as well as the second map with the ground truth. In the second map, clear lines are seen at the building edges.

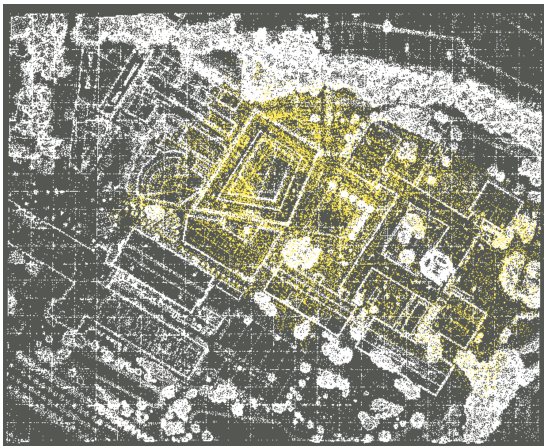


(a) Merged point cloud obtained from the collaborative SLAM

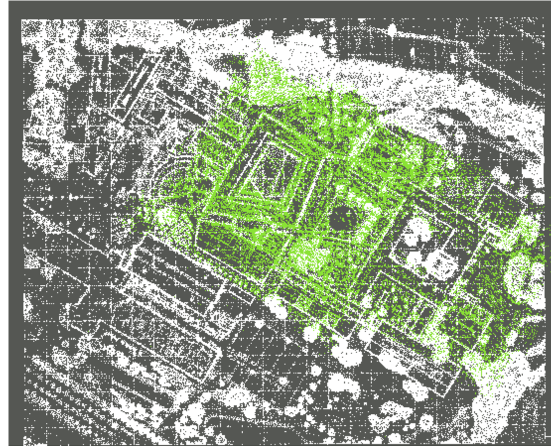


(b) Georeferenced map after collaborative SLAM

Figure 5.11: Results from collaborative SLAM on Ut-t7 and Ut-t8 before and after georeferencing



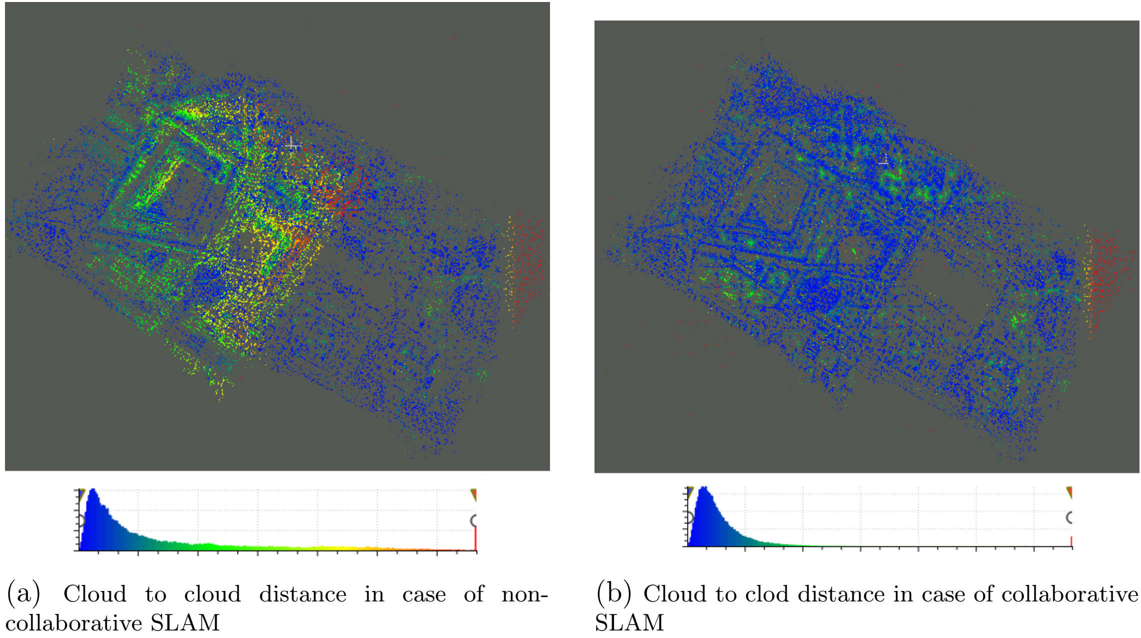
(a) Map from non-collaborative SLAM against the ground truth



(b) Merged map from collaborative SLAM against ground truth

Figure 5.12: Comparing maps from non-collaborative and collaborative SLAM against ground truth

To evaluate the performance of the algorithm, the cloud to cloud distance analysis is performed, as shown in figure 5.13. With a maximum distance of 20, the cloud to cloud distance for each map against the ground truth is carried out, which then gives a mean distance value, where lower it is, the better. In figure 5.13a, we can see that there are areas where the distance from the ground truth is higher, and there are more number of points like that. These are areas where map merging should take place to reduce the misalignment. In figure 5.13b, almost all the points are closer to the ground truth, and there is less discrepancy in areas where map merging has occurred. The mean distance for the first case is **4.415**, while in the second case it is **1.814**. Hence there is a reduction in errors in the case of collaborative SLAM.



(a) Cloud to cloud distance in case of non-collaborative SLAM

(b) Cloud to clod distance in case of collaborative SLAM

Figure 5.13: Comparing cloud to cloud distances of the maps in non-collaborative and collaborative scenarios with max. distance of 20

Vatican Dataset

Now, considering the Vatican datasets, where 3 drones are used. The behavior is similar to that of the UTwente datasets. The individual maps from non-collaborative SLAM and the merged map from collaborative SLAM performed on the Vat-t6, Vat-t7 and Vat-t8 datasets are shown in figure 5.14. The map from collaborative SLAM is decently merged.

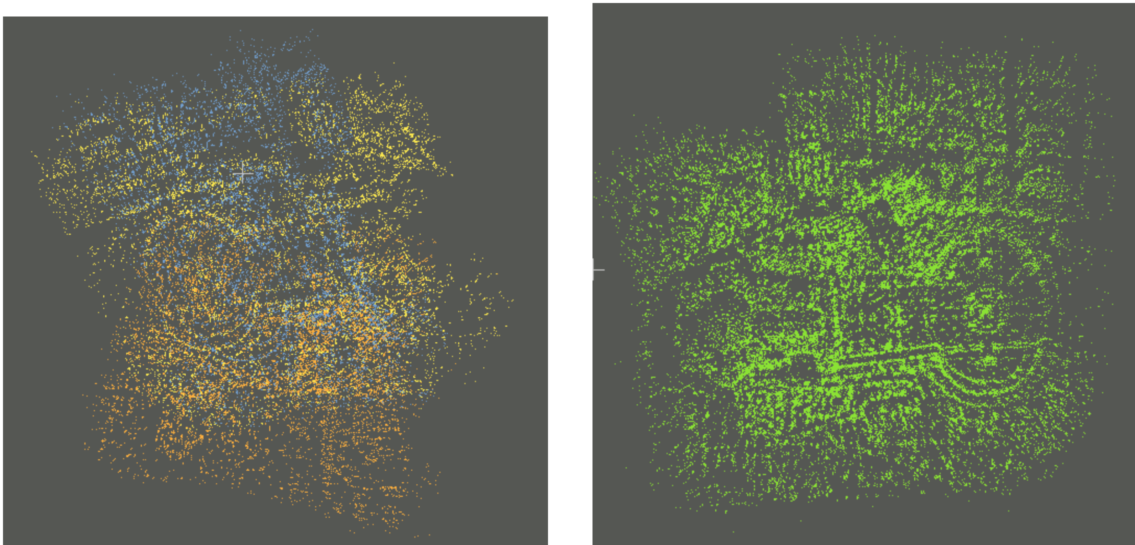


Figure 5.14: Maps obtained from non-collaborative (left) and collaborative (right) on Vat datasets

Once georeferencing is performed, the point clouds are compared with their ground truths, as shown in figure 5.15. Once again, clearer features and edges are seen in the merged map from collaborative SLAM as compared to the noisier one on the left.

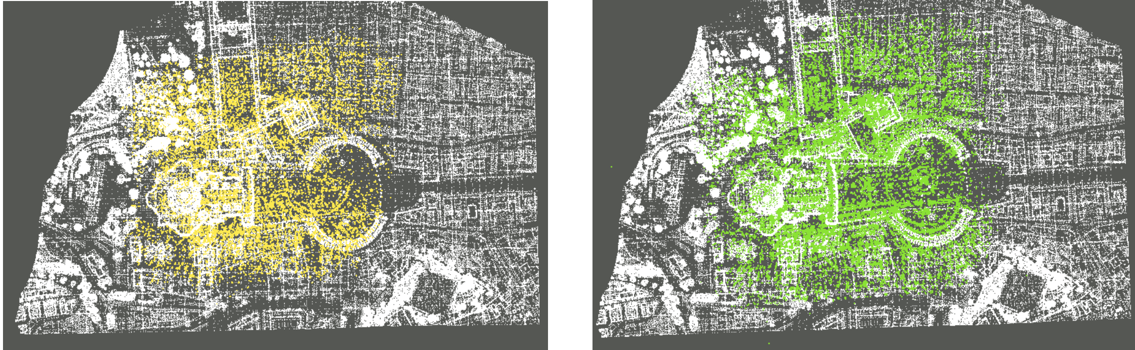


Figure 5.15: Maps obtained from non-collaborative (left) and collaborative (right) aligned with their ground truths after georeferencing.

The cloud to cloud distance analysis is also performed on these results, as shown in figure 5.16. Since the Vatican Blender model was smaller, a maximum distance of 2 was used as the maximum cloud distance. In the non-collaborative case, there are more points having a greater cloud to cloud distance than in the case of collaborative SLAM. The mean distance in the first case was **0.289** and in the second case it was **0.114**, once again demonstrating that the collaborative SLAM performs better.

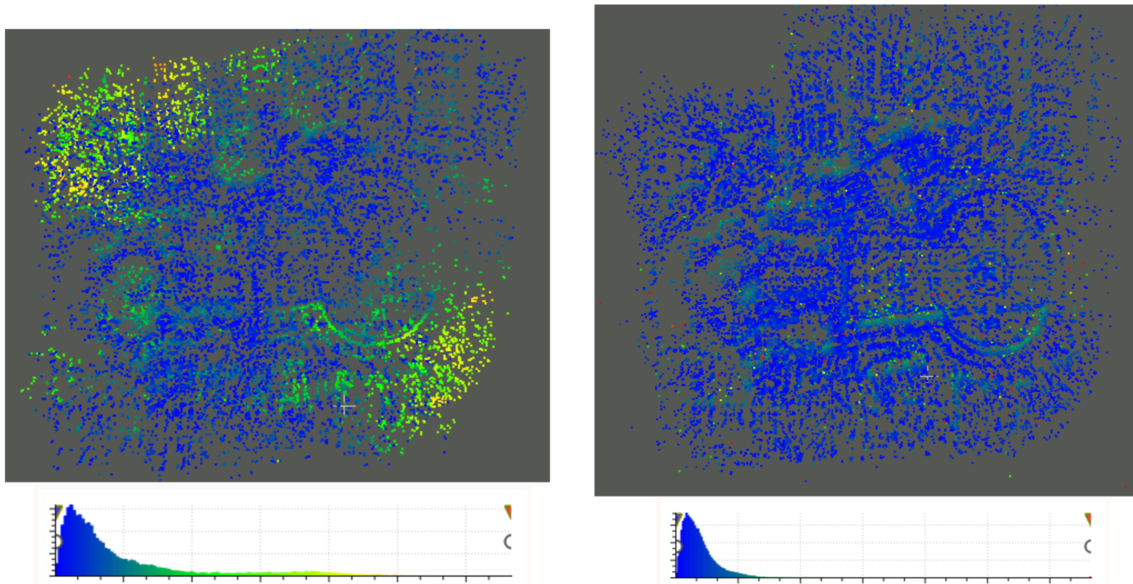
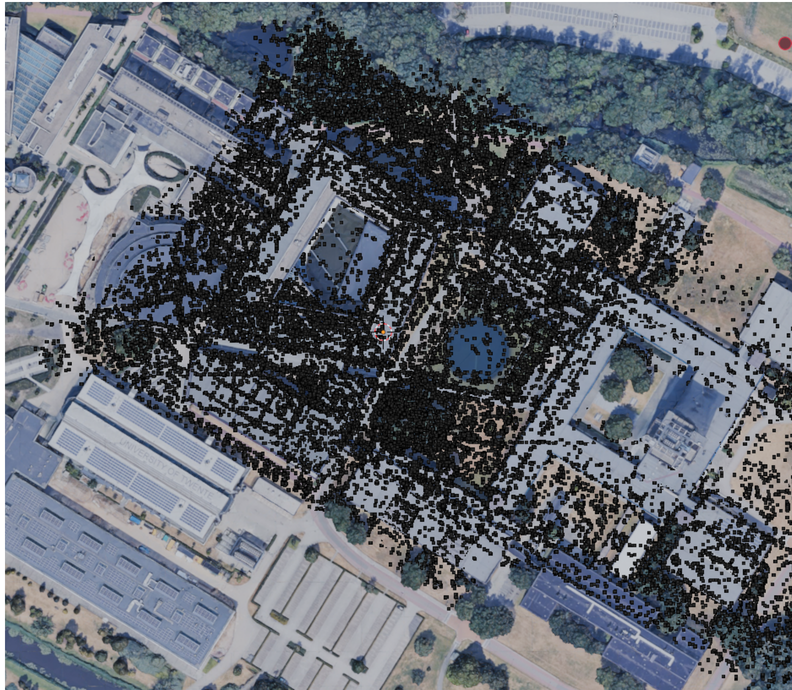
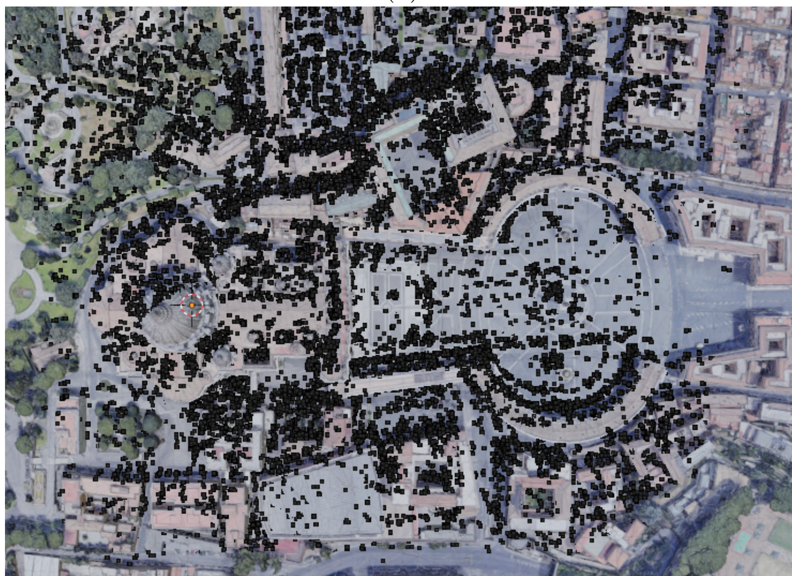


Figure 5.16: Cloud to cloud distance for the non-collaborative (left) and collaborative (right) cases, with max. distance of 2.

Lastly, the merged point clouds from the collaborative SLAM cases of both UTwente and Vatican models are imported to Blender, to compare with the actual 3D model. This is shown in figure 5.17. The sparse point clouds are decently aligned with their models.



(a)



(b)

Figure 5.17: The merged maps from collaborative SLAM on UTwente and Vatican datasets aligned with their respective 3D models.

Chapter 6

Conclusion and Future Work

The goal of this project was to implement a real-time, collaborative, visual-SLAM algorithm on simulated data. There were two parts to this: one, to create a simulation environment and collect data from UAVs, and two, to run a collaborative SLAM algorithm on this data and evaluate its performance.

The data acquisition steps involved building a 3D model of the environment, building a Gazebo world, setting up a drone simulator, and creating datasets. The 3D model of the environment was created in Blender using the OSM-plugin and Google 3D map tiles. A gazebo world was built, incorporating this environment and a model of a drone equipped with sensors like a downward-facing camera and GNSS. The framework of Gazebo-ROS-MAVROS-ArduPilot was chosen for the flight simulator. Using QGC, missions were planned to fly the drone in the 3D simulation environment, capture data, and store them in rosbag files. Among the various ROS topics available, the ones important to this project were the camera image, GNSS global and local positions, and the gazebo model states information to serve as the ground truth. The UT-dataset contains two drones, while the Vat-dataset contains 3 drones.

Before running a SLAM algorithm on this dataset, a literature survey into the state-of-the-art of visual-SLAM and collaborative-SLAM algorithms was carried out. From the survey, two collaborative SLAM frameworks, CCM-SLAM and COVINS, were found to be interesting for the goals of this project. Conducting some preliminary tests on the two algorithms using benchmark datasets, CCM-SLAM was chosen for this project. New configuration files were written to accommodate the simulation data. Some tests were carried out to tune the SLAM algorithm parameters, by comparing the RMSE values of the resulting trajectories. It was found that for a balance between performance and accuracy, a local map size of around 100, a keyframe rejection ratio of 0.98 and a window size of 40 can be used for mapping. The buffer size of around 40 is good, but can be varied depending on the number of agents used. CCM-SLAM is then run on each of the datasets individually as a non-collaborative agents, and then together as collaborative agents. The trajectories thus obtained are plotted to check for qualitative results. It is found that for in the non-collaborative case, the pose and scale of each agent is different since it is assigned arbitrarily in monocular SLAM. In the collaborative case, the SLAM algorithm is able to merge the maps from different agents and align their trajectories accordingly. To improve the orientation and scale of results, georeferenced using the data from the GNSS sensor as a post-processing step. By doing this, the trajectories are aligned with the ground truth values, and can be observed in the plots. For quantitative evaluation, the trajectories are

compared with their ground truths and the RMSE is calculated. Though the absolute value of RMSE is slightly higher than the individual RMSE, the percent RMSE with respect to trajectory length is lower than all the individual RMSE values. A 0.034% RMSE was obtained for the collaborative-SLAM of the UT datasets, and 0.04% RMSE was obtained for the Vatican dataset. Lastly, the 3D reconstruction results were discussed by comparing the point clouds obtained from individual agents with that obtained from collaborative SLAM. A point cloud of the 3D model acts as the ground truth. Just like the trajectories, the point clouds are not aligned in case of non-collaborative SLAM. They are also not in the same orientation as the ground truth, which is improved by georeferencing. It is seen that in the collaborative scenario, the maps from multiple agents are merged together and that results in a denser and more accurate point cloud. Based on these results, it can be concluded that the trajectory estimation and 3D mapping process is improved with collaborative SLAM.

Future Work

The advantage of the simulation environment used in this project is that it could be used as a test bed for SLAM algorithms. By varying different parameters like lighting, adding occlusions or dynamic objects, new datasets can be collected. Additional sensors like IMU can be added to the drone to facilitate monocular-inertial fusion based algorithms. For this project, fairly simple trajectories were used, but this can be changed to include more complex drone movements like sharp turns, variation in altitude, intersecting paths, etc. As technology improves, it will be possible to create even more detailed environments using less computational resources. Then more drones can be smoothly integrated into the environment.

To improve the SLAM algorithm, tests can be performed on more varied types of data. In this project, datasets were recorded in rosbags to be supplied later to the algorithm. But real-time tests can be performed by running the SLAM algorithm while the drone is running in the simulation environment. This would probably require a good share of computational resources, and the tasks may need to be delegated between different systems. GNSS data was used to georeference the SLAM output as a post-processing step, but this can be instead fused with the camera data within the algorithm using filters. Other improvements to the algorithm could be to explore deep-learning techniques that work alongside this architecture to improve accuracy. Another possibility could be make it possible to use a swarm of agents efficiently.

Bibliography

[1] Xiang Gao, Tao Zhang, Yi Liu, and Qinrui Yan. *14 Lectures on Visual SLAM: From Theory to Practice*. Publishing House of Electronics Industry, 2017.

[2] Rotation matrix - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Rotation_matrix.

[3] Ali Rida Sahili, Saifeldin Hassan, Saber Muawiyah Sakhrieh, Jinane Mounsef, Noel Maalouf, Bilal Arain, and Tarek Taha. A survey of visual slam methods. *IEEE Access*, 11:139643–139677, 2023.

[4] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. Orb-slam: a versatile and accurate monocular slam system. 2015.

[5] Patrik Schmuck and Margarita Chli. Ccm-slam: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams. *Journal of Field Robotics*, 36(4):763–781, December 2018.

[6] San Jiang, Wanshou Jiang, and Lizhe Wang. Unmanned aerial vehicle-based photogrammetric 3d mapping: A survey of techniques, applications, and challenges. *IEEE Geoscience and Remote Sensing Magazine*, 10(2):135–171, June 2022.

[7] Pierre-Yves Lajoie, Benjamin Ramtoula, Fang Wu, and Giovanni Beltrame. Towards collaborative simultaneous localization and mapping: a survey of the current research landscape. *Field Robotics*, 2(1):971–1000, March 2022.

[8] Weifeng Chen, Guangtao Shang, Aihong Ji, Chengjun Zhou, Xiyang Wang, Chonghui Xu, Zhenxiong Li, and Kai Hu. An overview on visual slam: From tradition to semantic. *Remote Sensing*, 14(13):3010, June 2022.

[9] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, page 3863–3870. IEEE, May 2017.

[10] Lawrence R. Weill Angus P. Andrews and Mohinder S. Grewal. *Global Positioning Systems, Inertial Navigation, and Integration*. Wiley, June 2006.

[11] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 10 2023.

[12] Rainer Kummerle, Giorgio Grisetti, Hauke Strasdat, Kurt Konolige, and Wolfram Burgard. Glt;supgt;2lt;/supgt;o: A general framework for graph optimization. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, May 2011.

[13] D. Galvez-López and J. D. Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, October 2012.

[14] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, June 2007.

[15] Mohammed Chghaf, Sergio Rodriguez, and Abdelhafid El Ouardi. Camera, lidar and multi-modal slam systems for autonomous ground vehicles: a survey. *Journal of Intelligent amp; Robotic Systems*, 105(1), April 2022.

[16] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. Visual slam algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, 9(1), June 2017.

[17] Richard A. Newcombe, Steven J. Lovegrove, and Andrew J. Davison. Dtam: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*. IEEE, November 2011.

[18] Jakob Engel, Thomas Schöps, and Daniel Cremers. *LSD-SLAM: Large-Scale Direct Monocular SLAM*, page 834–849. Springer International Publishing, 2014.

[19] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry, 2016.

[20] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, page 15–22. IEEE, May 2014.

[21] Kaiqi Chen, Jianhua Zhang, Jialing Liu, Qiyi Tong, Ruyu Liu, and Shengyong Chen. Semantic visual simultaneous localization and mapping: A survey, 2022.

[22] Yuxiang Sun, Ming Liu, and Max Q.-H. Meng. Improving rgb-d slam in dynamic environments: A motion removal approach. *Robotics and Autonomous Systems*, 89:110–122, March 2017.

[23] Berta Bescos, Jose M. Facil, Javier Civera, and Jose Neira. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes. *IEEE Robotics and Automation Letters*, 3(4):4076–4083, October 2018.

[24] Shichao Yang and Sebastian Scherer. Cubeslam: Monocular 3-d object slam. *IEEE Transactions on Robotics*, 35(4):925–938, August 2019.

[25] Saad Mokssit, Daniel Bonilla Licea, Bassma Guermah, and Mounir Ghogho. Deep learning techniques for visual slam: A survey. *IEEE Access*, 11:20026–20050, 2023.

[26] Dinar Sharafutdinov, Mark Griguletskii, Pavel Kopanev, Mikhail Kurenkov, Gonzalo Ferrer, Aleksey Burkov, Aleksei Gonnochenko, and Dzmitry Tsetserukou. Comparison of modern open-source visual slam approaches. *Journal of Intelligent amp; Robotic Systems*, 107(3), March 2023.

[27] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, October 2017.

[28] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. Orb-slam3: An accurate open-source library for visual, visual-inertial and multi-map slam. 2020.

[29] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. Openvslam: A versatile visual slam framework. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19. ACM, October 2019.

[30] Tong Qin and Shaojie Shen. Robust initialization of monocular visual-inertial estimation on aerial robots. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page 4225–4232. IEEE, September 2017.

[31] Patrick Geneva, Kevin Eckenhoff, Woosik Lee, Yulin Yang, and Guoquan Huang. Openvins: A research platform for visual-inertial estimation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2020.

[32] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, August 2018.

[33] Tong Qin and Shaojie Shen. Online temporal calibration for monocular visual-inertial systems. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page 3662–3669. IEEE, October 2018.

[34] Antoni Rosinol, Marcus Abate, Yun Chang, and Luca Carlone. Kimera: an open-source library for real-time metric-semantic localization and mapping. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2020.

[35] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative monocular slam with multiple micro aerial vehicles. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, November 2013.

[36] Titus Cieslewski, Siddharth Choudhary, and Davide Scaramuzza. Data-efficient decentralized visual slam. 2017.

[37] Yulun Tian, Yun Chang, Fernando Herrera Arias, Carlos Nieto-Granda, Jonathan P. How, and Luca Carlone. Kimera-multi: Robust, distributed, dense metric-semantic slam for multi-robot systems, 2021.

[38] Marco Karrer, Patrik Schmuck, and Margarita Chli. Cvi-slam—collaborative visual-inertial slam. *IEEE Robotics and Automation Letters*, 3(4):2762–2769, October 2018.

[39] Patrik Schmuck, Thomas Ziegler, Marco Karrer, Jonathan Perraudin, and Margarita Chli. Covins: Visual-inertial slam for centralized collaboration, 2021.

[40] Manthan Patel, Marco Karrer, Philipp Bänninger, and Margarita Chli. Covins-g: A generic back-end for collaborative visual-inertial slam, 2023.

[41] Free Environment 3D Models | CGTrader — cgtrader.com. https://www.cgtrader.com/free-3d-models/environment.

[42] Sketchfab - The best 3D viewer on the web — sketchfab.com. https://sketchfab.com/.

[43] Architecture 3D Models for Free Download - Open3dModel — open3dmodel.com. https://open3dmodel.com/3d-models/architecture.

[44] 3D Models for 3D Pros. https://www.turbosquid.com/.

[45] GitHub - leonhartyao/gazebo_models_worlds_collection: collection of gazebo models and worlds — github.com. https://github.com/leonhartyao/gazebo_models_worlds_collection/tree/master, 2020.

[46] Gazebo Worlds | PX4 User Guide (v1.12) — docs.px4.io. https://docs.px4.io/v1.12/en/simulation/gazebo_worlds.html.

[47] Blender Foundation. blender.org - Home of the Blender project - Free and Open 3D Creation Software — blender.org. https://www.blender.org/.

[48] Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine — unity.com. https://unity.com/.

[49] Unreal Engine: The most powerful real-time 3D creation tool. https://www.unrealengine.com.

[50] Getting Started with Gazebo? &x2014; Gazebo ionic documentation — gazebosim.org. https://gazebosim.org/docs/latest/getstarted/.

[51] ROS: Home — ros.org. https://www.ros.org/.

[52] OpenStreetMap — openstreetmap.org. https://www.openstreetmap.org/.

[53] 3D development - OpenStreetMap Wiki — wiki.openstreetmap.org. https://wiki.openstreetmap.org/wiki/3D_development.

[54] Cesium: The Platform for 3D Geospatial — cesium.com. https://cesium.com/.

[55] ArcGIS | Geospatial Platform - GIS Software for Business & Government — esri.com. https://www.esri.com/en-us/arcgis/geospatial-platform/overview.

[56] Photorealistic 3D Tiles | Google Maps Tile API | Google for Developers — developers.google.com. https://developers.google.com/maps/documentation/tile/3d-tiles.

[57] Google Maps Platform Coverage Details | Google for Developers — developers.google.com. https://developers.google.com/maps/coverage#countryregion-coverage.

[58] ArduPilot. ArduPilot — ardupilot.org. https://ardupilot.org/.

[59] Home - AirSim — microsoft.github.io. https://microsoft.github.io/AirSim/.

[60] Blosm - Blender — blender-addons.org. https://blender-addons.org/blosm/.

[61] Use API Keys with Map Tiles API | Google Maps Tile API | Google for Developers — developers.google.com. https://developers.google.com/maps/documentation/tile/get-api-key.

[62] GitHub - Intelligent-Quads/iq_sim: example gazebo ardupilot simulation package — github.com. https://github.com/Intelligent-Quads/iq_sim.

[63] GitHub - ArduPilot/ardupilot: ArduPlane, ArduCopter, ArduRover, ArduSub source — github.com. https://github.com/ArduPilot/ardupilot.

[64] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[65] Michael Grupp. evo: Python package for the evaluation of odometry and slam. https://github.com/MichaelGrupp/evo, 2017.

[66] Daniel Girardeau-Montaut. CloudCompare - Open Source project — danielgm.net. https://www.danielgm.net/cc/.