Autonomous Navigation in Simulation by Leveraging Equirectangular Capable VSLAM

Conor Brooks Carmichael

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

February 11, 2022

Abstract

In this paper an approach to VSLAM is proposed that makes use of OpenVSLAM's highly effective open source development. A multi-camera rig is established in a simulation environment, to determine the efficacy of a stitched-equirectangular view in place of an expensive single equirectangular camera. The experiment, done in simulation proposes that multi-camera rigging is effective enough to leverage OpenVSLAM's equirectangular capabilities, without the need for acquiring an expensive camera, nor risking its safety when mounted on a robot. The method reaps the benefits of omnidirectional sensing, modern, highly capable and open source VSLAM software, and avoids the entry cost and risk. The approach set forth in this paper comes within 5% of a optimal system on a medium difficulty environment in navigation tasks, and displays promising results in map generation.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Conor Carmichael _____ Signature: _____

Acknowledgements

I would like to thank Dr. Jan Paul Siebert for all the guidance and support over the past few months. This project has exposed me to an area I am now very passionate about, and I am grateful to have been able to pursue this project under his guidance. Additionally, I would like to thank Dr. Helen Purchase for providing very valuable guidance to me at the start of the academic year.

Next, I would like to thank my family, I would not have been able to pursue this degree without their never ending support for my academic goals. Additionally, I would like to thank my girlfriend, who supported me several ways, not the least of which being emotionally during the ups and downs of such a strenuous academic program. I do not take for granted the sacrifices made by those around me, as I chose to pursue this degree.

Lastly, I would like to extend my gratitude to the ROS community. The support for newcomers via tutorials, open source packages, and extensive help forums was instrumental in every step of the way.

Contents

1 Introduction **5**

1.1 Idea 5

1.2 Motivation 5

1.3 Background 6

1.3.1 Robotics Operating System 6

1.3.2 SLAM 6

1.3.3 Visual SLAM 7

1.3.4 Gap in the State of the Art 7

1.4 Approach 8

1.5 Results 8

1.6 Document Overview 9

2 Implementation **10**

2.1 Implementation Overview 10

2.1.1 ROS Packages 11

2.2 Data Pipeline 13

2.2.1 Creating a Video of the Environment 13

2.2.2 Running OpenVSLAM 14

2.2.3 Converting to Point Cloud Data Format 14

2.2.4 Post Processing the Point Cloud File 14

2.2.5 Point Cloud to .yaml 15

2.3 Navigation Stack 15

3 Evaluation **17**

3.1 Evaluation Overview 17

3.2 Qualitative Evaluation 17

3.2.1 Point Clouds 17

3.2.2 Occupancy Grids 18

3.3 Quantitative Evaluation 19

4 Conclusion **21**

4.1 Summary of Results 21

4.2 Reflection on Limitations and Challenges 21

4.3 Further Work 22

A First appendix **23**

A.1 Gazebo Environment Captures 23

Chapter 1

Introduction

1.1 Idea

This dissertation documents the means of establishing an all around look multi-camera rig and how it can be effectively used in equirectangular (or panoramic) VSLAM to map an environment, thus enabling autonomous navigation. The approach sees application where cost-effectiveness is paramount, or where movement of a robot is limited and the benefits of omnidirectional awareness are most useful. This expands and combines the ideas from research on utilizing camera stitching for VSLAM [3], VSLAM in simulation [4], and equirectangular VSLAM approaches [10].

1.2 Motivation

There are many motivating factors for pursuing this project. In the simulation software Gazebo, there are many plugins to enable physics, sensors, and other key pieces of robotics to be simulated accurately. While they have plugins for camera sensors, there is no direct plugin for a 360°, or equirectangular fields of view. Therefore, this project may serve as an example for those looking to explore the topic of 360° camera sensing in simulation.

One form omnidirectional camera sensing is done by equirectangular projection, which takes a spherical image and projects it flat into a rectangular space. Equirectangular cameras record in all directions from the device, and combine the data into a single rectangular frame. This benefit is very significant in robotics [10], as it allows for much more information to be used for tracking, mapping, and localization than perspective cameras, which are limited to a narrow front facing view. The motivation here is to take advantage of the significant benefit of equirectangular imaging, however, doing so comes at a cost normally. An equirectangular camera, such as the Ricoh Theta used by Sumikura, et al. [10], bares a much more significant cost than pinhole cameras. The field of view benefit provided by equirectangular cameras is significant, but the added view overhead view may not contribute significantly to the VSLAM equation for a robot with three degrees of freedom. Therefore, the cost for the hardware may exceed the benefit. If it is possible to see the benefits offered by OpenVSLAM's equirectangular VSLAM algorithm, without incurring the cost of the intended hardware, this could be very beneficial to the future of omnidirectional sensing in robotics.

1.3 Background

1.3.1 Robotics Operating System

The Robotics Operating System (ROS) is a robotics development middleware intended to ease the difficulty of developing research within the robotics field [9]. It allows for great code reuse through packages, where it had previously been non-trivial. The individual packages are run on nodes, which run executable files defined by the user, and can share information with other nodes by sending messages. The messages are of a data type or data structure, and can be shared on a topic (communication channel) or via a service. Services share a similar design to web services, and topics are established on the principle of a publisher and subscriber method. The publisher node broadcasts information on a topic, independent of if what is subscribing. The subscriber nodes subscribe to the topic, and receive updates at a set rate.

The robotics operating system makes setting up sensors, collecting readings, and outputting actions from the data simple. With the Image message type, camera sensor information can be published on a topic and then subscribed to by some processing node. The robotics operating system has made enabling robots with SLAM (see section 1.3.2) capabilities simpler with this, and thus it is easy to collect and process data remotely. ROS has great support for the many different sensors used for SLAM.

1.3.2 SLAM

SLAM (Simultaneous Localization and Mapping) is a set of algorithms whose goal is to take in sensor data to build a map of the environment, while computing the current location in that environment [2]. In moving around the environment, the sensors estimate poses of landmarks, and attempt to estimate the agent's relative pose. Probabilistic SLAM aims to compute the probability distribution (taken directly from [2])

$$P(x_k, m | Z_{0:k}, U_{0:k}, x_0) \tag{1.1}$$

at time k where

- x_k is the agents location
- m is the set of all landmarks $\{m_1, m_2, ..., m_n\}$
- $Z_{0:k}$ is the set of all landmark observations
- $U_{0:k}$ is the history of control inputs
- x_0 is the agents location at time 0

This distribution defines the joint posterior density at time k of what landmarks are observed and the vehicles location, given knowledge through time k of what inputs the agent has taken, the set of prior landmarks observe up, and the initial state of the agent. The two most notable early solutions to solving the SLAM problem were with the Extended Kalman filter (EKF) and FastSLAM with

applying a Rao-Blackwellisation (R-B) filter to the sample space. More modern approaches have been created, but these solutions were large steps in the field.

The idea of SLAM being a probabilistic problem is important to note. Sensor readings are imperfect and noisy, therefore it is important to account for that in the SLAM algorithm [6]. The algorithm can only estimate the robots location and the pose of the landmarks in the environment. The robots location can be in part estimated by using the odometry information, but the location of the landmarks is only determined by the way the sensor readings are handled. The presence of landmarks is determined through feature extraction.

An important aspect of SLAM is the idea of ”loop closure”. When an agent has moved about its environment for some time, loop closure is the challenge of determining if the agent has been to this location before. As demonstrated by Newman, Ho [7], camera sensors can supplement traditional laser scanning to establish robust loop closure detection. It is also noted that comparatively, ”cameras are cheap, ubiquitous, passive and information rich”, therefore they were demonstrated useful in determining loop closure. At the time (2005), this rich source of information was not yet taken advantage of for SLAM, but soon was demonstrated capable to be used for real-time SLAM, extending the SLAM field.

1.3.3 Visual SLAM

Visual SLAM, is the branch of SLAM algorithm using camera sensors as the information source for Simultaneous Localization and Mapping. Visual SLAM research was greatly progressed by Davison et al. with MonoSLAM [1], accomplishing real-time monocular VSLAM with a single camera. This was the first instance of VSLAM on a mobile agent using no other sensor data. The VSLAM problem is challenging mostly due to the lack of direct depth measurement. By the inherent property of laser scans, depth can be accurately assessed - camera imaging does not cleanly relay this information. In MonoSLAM, EKF was used for estimating the pose of features and the camera’s motion.

Multi-camera rigging (a set of multiple cameras on a robot) was proven a viable solution for VSLAM, which leverages more information for superior localization [3]. This system leveraged eight cameras, pointed in different directions so as to not overlap, and customized VSLAM algorithm which relies partially on odometry information. However, multi-camera rigging was shown to be a useful way of implementing an omnidirectional camera sensor, and made a good observation on how equirectangular projection may be excess information.

OpenVSLAM released the first open source VSLAM framework which accepted equirectangular camera data [10]. This system is highly modular, and is a key facet of the design. A great deal of VSLAM related research excels in testing, but lacks the foresight for reproducibility and extensability. Another significant feature is the ability to store and reuse the maps created during the VSLAM run. New images can be localized based on these pre-existing map files

1.3.4 Gap in the State of the Art

The features OpenVSLAM offers makes it a very impressive contribution to the robotics’ open source community. Its capabilities leave it in a great position to influence new research and com-

mercial development, especially with respect to its equirectangular capability. However, equirectangular cameras are rather expensive - at the time of writing this, the camera in OpenVSLAM equirectangular examples costs over 300 USD. Therefore, a middle ground between utilizing the equirectangular option, and avoiding the potentially innovation-stifling cost of the requisite camera, could be quite valuable.

The aspect of using SLAM algorithms of all kinds in navigation and further tasks is frequently neglected in research [5]. OpenVSLAM's output stores landmark information on the disk, which is a useful aspect of the research. These landmarks can be loaded in for localization methods and further mapping. One reason that it is important to finish the pipeline of using these methods in navigation is that the output of most VSLAM algorithms is a sparse point cloud. This results in enough detail for presence of objects, but not enough for scene recreation. Error in sparse point clouds has the potential to throw off navigation more as there is less information to use in path planning and determining which locations are safe.

Lastly, VSLAM research is largely done in real world environments, and it is interesting to see how these algorithms perform in simulated environments. There is far more detail in real world situations. The lighting, the textures, and general scene detail adds a lot of features for extraction in camera processing. Simulation environments are essential for robotics development, as they allow for prototyping and testing without risking safety and hardware degradation.

1.4 Approach

To actualize the idea set forth in section 1.1, many different components needed to come together. A mobile robot with 3 degrees of freedom was established and equipped with a multi-camera rig. The rig setup produced a smooth, 360° camera stream with enough information such that OpenVSLAM was capable of extracting landmarks, and was able to accept it as a spherical camera (the same as it does with the Ricoh Theta camera). An important aspect is making sure that this camera feed does not mislead the VSLAM algorithm with respect to landmark relation and distances. This includes aligning the cameras properly, and processing the feeds together in a way that does not feel disjointed. In order for OpenVSLAM to work in simulation, a highly detailed and well lit environment was designed.

If OpenVSLAM is able to properly analyze the camera feed established, the next step was to make use of the information output. The landmark information is then used to create a sparse point cloud, which in turn was used for navigation. The output needed to be structured and clean enough to be used in path planning, such that the ROS navigation stack could be effective.

1.5 Results

In evaluation of this system, the following results were accomplished:

- A successful prototype for 360° camera development in Gazebo simulation.
- A pipeline to use OpenVSLAM's equirectangular VSLAM in navigation.

- Valuable point cloud noise reduction.
- Accurate mapping of three simulated Gazebo environments.
- Highly successful navigation in two of three environments.
- Navigation comparable to a system with perfect knowledge of the environment.

1.6 Document Overview

Moving forward, the document will follow the following format: Chapter 2 outlines the implementation, discussing the ROS components, Gazebo simulation environments, and the data processing pipelines, with a section dedicated to each package written for this project; Chapter 3 discusses the methods of evaluation, the qualitative map and point cloud evaluation, and quantitative navigation results; and lastly, Chapter 4 concludes the dissertation with a discussion of the results, limitations and challenges, and how the project can be improved in the future.

Chapter 2

Implementation

2.1 Implementation Overview

The implementation of this system considers the goals and restrictions of the requisite software. The design needed to include the following:

- A robot capable of taking user input to drive around the environment, and also accept commands for autonomous navigation.
- Camera sensors to capture a full 360° view around the robots base.
- A process for taking the output of OpenVSLAM, and making it useful with respect to navigation.
- Sensible simulation environments for testing, evaluating, and proving the concept of the system. The environments needed to be rich in detail and texture.

To accomplish this, ROS packages for the robot's description, the camera stitching, Gazebo environments, and autonomous navigation were necessary. There were many iterations of design in each of the final packages. The robot initially was two-wheeled and controlled by the differential drive plugin. Environments were made from scratch in Blender, and discarded of due to texture and detail concerns in favor of using Gazebo's model database to construct environments via the user interface.

Due to reasons outlined in section 1.3.4, OpenVSLAM is the VSLAM software of choice for implementation. OpenVSLAM is tested on Ubuntu 16.04, which drives other software decisions. ROS 1 was chosen for the robotics middleware due to its great community resources, and most importantly its compatibility with Python and OpenVSLAM. OpenVSLAM has support for enabling VSLAM through a subscriber node in ROS, and ROS supports writing publisher and subscriber nodes in Python 2.7. ROS Kinetic was used, as it is the recommended release to be used with Ubuntu 16.04. For a simulation environment, the Gazebo software allows for great environment creation through its set of open source models, and it also has great support for simulating ROS controlled robots.

2.1.1 ROS Packages

Robot Description

The `bot_description` package contains the Universal Robot Description Files (URDF) which details the specifications for the robots form and the camera system. The robot has a rectangular base, connected to four "wheels" (thin cylinders) via continuous joints. The robot's footprint extends 0.5 meters laterally, and 0.8 meters front to back. The robot is controlled by Gazebo's *skid_steer_drive_controller* which allows for accurate turning in tight spaces.

The multi-camera rig consists of four 90° (1.5708 radians) horizontal field of view (HFOV) cameras, each centered on the +/- relative x and y axes, shown in Figure 2.1. This allows each direction to be simultaneously captured on a separate ROS topic. The cameras are represented in the URDF as sensors of type *wideanglecamera*. This allows for a custom lens parameter, which is used to add a slight distortion to the imaging which smooths the transition between each camera sensors frame in the stitched camera feed. The cameras each record at 30 frames per second, and a resolution of 960 pixels by 540 pixels.

Camera Processing

The `camera_processing` package contains the code for stitching the camera streams together. It subscribes to the four camera Image message topics and synchronizes the feeds. Then, using OpenCV it creates a new image (3840 pixels by 540 pixels), inserting the images in the following order:

1. Right half of the negative x-axis camera
2. Negative y-axis camera
3. Positive x-axis camera
4. Positive y-axis camera
5. Left half of the negative x-axis camera

Using a feed synchronizer furthers the smoothness of the camera by ensuring the frames used in execution are from the same time stamp. A smooth camera feed is essential to allowing OpenVSLAM to accept and properly analyze the camera data. When the subscribers are synchronized, they call a common callback function on each tick. The callback function deals with stitching the camera feeds together. The camera stitching order (as denoted above) is done so that the front-facing camera stays centered in the new 360° view. This allows for accurate localization and for landmarks to be properly oriented when running VSLAM.

In the callback, the images are originally encoded as ROS Image messages. This does not allow for much flexibility, thus each of the four incoming images are converted to OpenCV images. OpenCV allows for images to be treated as NumPy arrays, which in turn allows for efficient and fast concatenation the (now) five images. Once the new image is created, it is converted to an Image message with proper encodings, and published to the *'/camera/image_raw'* topic. An example of the cameras output is shown in Figure A.4.

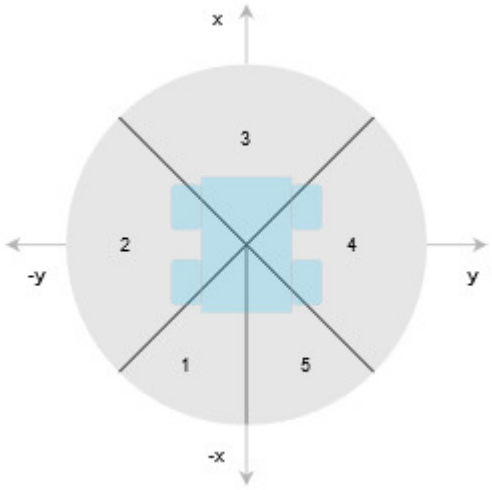

Figure 2.1: Representation of the multi-camera rig. Blue represents the footprint of the robot, and the grey area shows the coverage of each camera, according to the list.

Navigation

The `navigation` package contains:

- Scripts for controlling and tracking navigation goals.
- Goal sets for navigation for each world.
- A metrics folding storing the results of navigation evaluation.
- A config folder storing files to set parameters for the global and local costmaps, and the navigation planner.

Generally, the ActionLib library can be used to send navigation goals to the `move_base` package. However, certain methods were incompatible with the development stack in use. Therefore, it was necessary to implement features to send navigation goals and check the status of those goals. The navigation_controller.py file reads in a list of goal poses (which form a loop around the environment), sends the goals one at a time, checking for success of the previous goal before sending. The goals are sent to the *'/move_base_/goal'* topic as MoveBaseActionGoal messages.

The navigation_tracker.py file tracks the status of the goals being sent. Based off the MoveBaseActionGoal identifier key, it tracks groups of goals as a loop, and measures the time taken to complete the whole loop, and the distance covered by the robot during the loop. At each tick, the distance between pose of the robot and the previous pose is used to calculate the distance it covers, accounting for any reversing, extra movements when imprecisely reaching a goal. When the loop has ended, the results are written to a .csv file for later analysis.

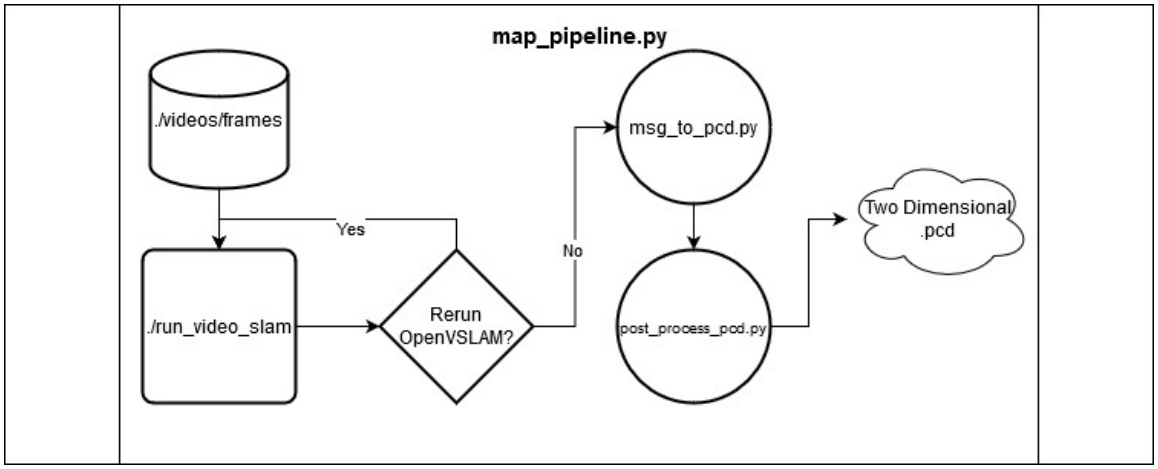

Figure 2.2: Image representation of the process from image collection to a .pcd file ready to be used as an occupancy grid

Gazebo Environment

The `gazebo_env` package stores all information pertinent to the environments the robot is tested in. It stores-

- Three world files of increasing levels of difficulty for navigation. The environments are shown in Figures A.1, A.2, and A.3
  - Easy: Tests the ability of the system to perform loop closure, and a provides a good proof of concept scenario.
  - Medium: Further tests the system to handle objects that need navigating about.
  - Hard: Provides a significantly more difficult environment with more complex obstacles, and much more to avoid in path planning.
- The respective map files, each world file has a ground truth, OpenVSLAM generated map.
- Launch files for mapping and navigation.

2.2 Data Pipeline

A notable piece of this system is the pipeline from the camera still image feed, to a two-dimensional point cloud representation of the map. The pipeline, shown in 2.2 is controlled by one script-map_pipeline.py, all components are optional, but proceeds as follows [1].

2.2.1 Creating a Video of the Environment

Before a video can be created, using an open source teleop package, the robot is driven about the environment. During this, the *camera_processing* node clears a directory, and when stitching

[1]It is important to not that this system can be reconfigured to run OpenVSLAM from a ROS topic Image stream.

together the published camera feed, it also saves each frame. Then, ffpmeg is used to transform the directory of frames, into a video. It is important to note, that this pipeline is setup to save time when consistently adjusting pieces. The camera stream is published on its own topic, and therefore can be used for VSLAM live. However, it makes most sense here to have one video, that can be reused for VSLAM when adjusting camera configuration values, and tweaking OpenVSLAM parameters.

2.2.2 Running OpenVSLAM

OpenVSLAM is very adaptable to each use case. It can be used by getting input from a ROS topic, a set of images, or a video. While I do start this pipeline with folder of images, it worked best being able to view the .mp4 before passing it to the VSLAM. The main script passes in the newly created video, and performs OpenVSLAM's VSLAM algorithm. By using a video, the pipeline can rerun *./run_video_slam* until a satisfactory run is complete.

The VSLAM program takes in a configuration .yaml file, which is based on one of the example configuration files for the Ricoh Theta camera model. The output of *./run_video _slam* is a MessagePack file, which encodes the landmark pose data. This file is written to the maps folder in the OpenVSLAM build directory, which is then passed to the next piece of the pipeline.

2.2.3 Converting to Point Cloud Data Format

Next, a script is run to convert the default MessagePack output file to a Point Cloud Data file (.pcd). The script reads the MessagePack file, extracts the Landmark data, then the x, y, z values. It writes these points to a a new file, with the appropriate Point Cloud Data file format's heading.

2.2.4 Post Processing the Point Cloud File

A crucial step of this pipeline is the point cloud processing. The point cloud shown in figures 2.2.4 presents a good reading of the environment, but if no processing is done it would lead to suboptimal pathing. The point cloud has far reaching outliers, as well as isolated outliers in the clear areas the robot traversed.

The script puts the point cloud through several operations. The script rotates the point cloud to properly conform in RViz, filters outliers via a minimum neighbor algorithm, and flattens the point cloud to two dimensions. It also applies a scaling factor to the point cloud. Since the *fake_localization* package is used (due to time constraints) it is necessary to adjust the scale to assure it matches up between RViz and Gazebo. If the localization module were included this should not be necessary.

Outlier removal is essential part of working with point cloud data [8], and the script applies a minimum neighbor filter to each point as stated. The algorithm iterates over all points, checks within a radius r for a minimum amount of points k, and if there are not enough points it removes it from the point cloud. The values 0.2 and 5 were determined to be fitting for r and k respectively through test runs.

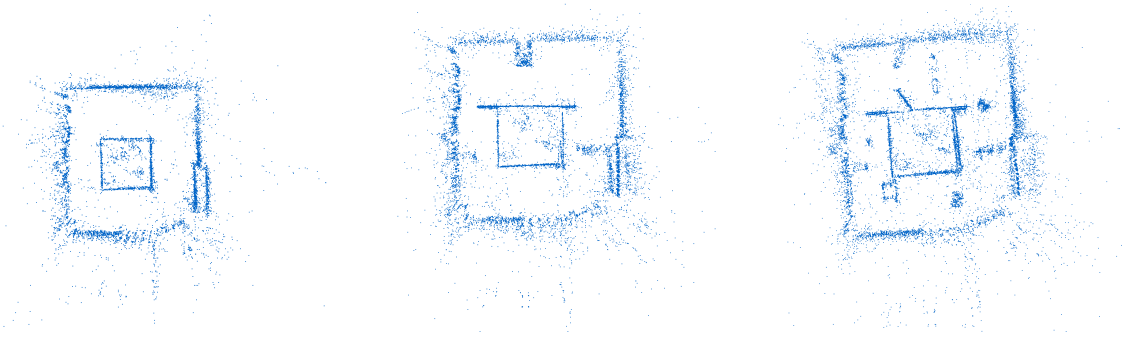

Figure 2.3: Generated 3d point clouds of the (from left to right) Easy, Medium, and Hard worlds.

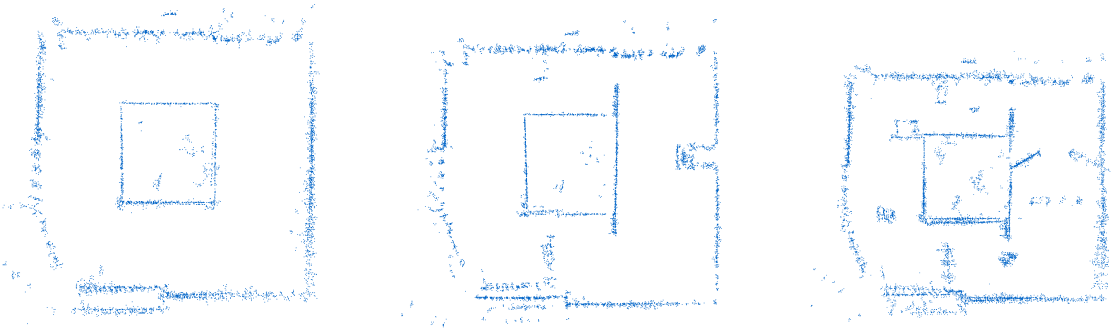

Figure 2.4: Processed output 2d point clouds of the (from left to right) Easy, Medium, and Hard worlds. Note, the processed point clouds are rotated 90° clockwise.

2.2.5 Point Cloud to .yaml

The last operation has to be done outside of the main pipeline script. A launch file is used to create an `octomap_server` that serves the two dimensional point cloud. When this is running, a `map_server` node of type `map_saver` is launched to create a file compatible with the `map_server` package. This step is not necessary for navigation, as the system can run navigation goals using the `octomap_server` package. This is done for consistency in evaluation; when evaluating the system, a ground truth map occupancy grid served by `map_server` is used for comparison. This way in navigation, the same server is used, and the two occupancy grids can be directly compared (visually).

2.3 Navigation Stack

The implementation of the navigation stack was heavily derived from the ROS tutorials, shown in figure 2.5, though some notable changes were made to support the use case of this project. The navigation stack requires a properly configured transform (tf) tree. To do this, *joint_state_publisher* and *robot_state_publisher* nodes were added to publish the links of the robot. A *static_transform_publisher* was used to link the world and map frames. Once a transform tree was connecting the world, map, odom, and base link of the robot, the rest of the stack could be implemented.

The robot was localized in the environment using the *fake_localization* package, which uses the true

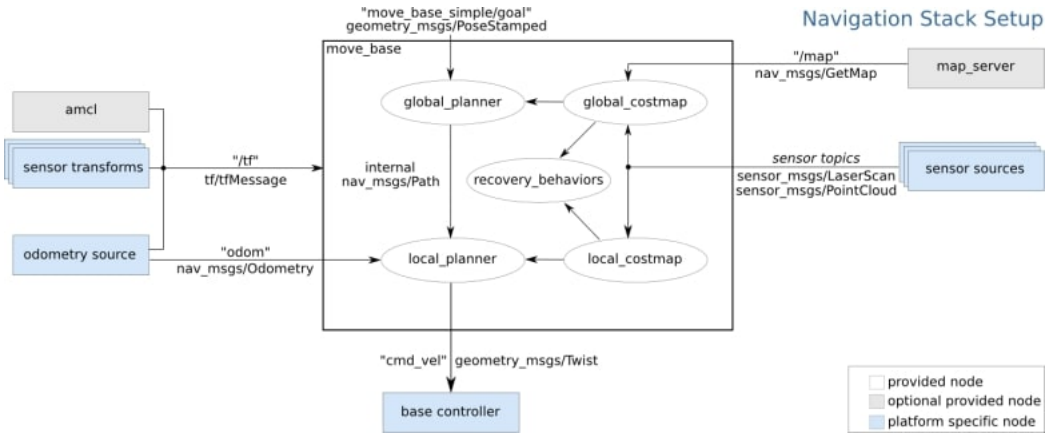

Figure 2.5: ROS Navigation stack: http://wiki.ros.org/navigation/Tutorials/RobotSetup

pose of the robot given by the Gazebo simulator and remaps it to the ”odom” frame. The decision to use this was made due to time constraints mostly, but also to focus on evaluating the efficacy of the initial mapping. There was not enough time to work OpenVSLAM's localization node into the stack. The `map_server` node serves the map generated in the by the launch file described in section 2.2.5.

The *move_base* node is responsible for taking in the information gathered by the aforementioned pieces of the navigation stack, and creating a costmap for path planning, finding the best path, and sending movement commands on the *'/cmd_vel'* topic. This piece needs to be well tuned to the robot in use. There are many parameters to go over, concerning the maximum and minimum movement speeds it can handle, the base footprint of the robot, frequency to publish on *'/cmd_vel'* and more. Performance is highly dependent on how well the parameters represent the robot and environment.

Chapter 3

Evaluation

3.1 Evaluation Overview

Evaluation of the designed system must determine its efficacy for creating maps to use in autonomous navigation. This evaluation is done in two parts - qualitatively and quantitatively. Qualitative evaluation is to ensure that the overall output of the system is accurate to the actual simulation environment. The point clouds and maps generated must not mislead navigation systems, but instead must properly identify the structure of the environment (i.e., walls) and the obstacles impeding paths (i.e., construction cones). Quantitative evaluation is performed to see how well the maps can be used in navigation tasks. The requirement set forth is that, given a ground truth map of the environment and a generated map, navigation tasks are completed in sufficiently similar times and distances. It is expected that a ground truth system would perform better in these categories, thus this acts as the ceiling for performance. Navigation should not be better with the map that is generated, but it should not fall markedly short either.

3.2 Qualitative Evaluation

3.2.1 Point Clouds

It is important to analyze the point cloud representations of the environment. A major factor for success is how well the OpenVSLAM output, a sparse point cloud of the environment, is able to be processed into a clean occupancy grid. The first step in that is to transform the 3-D point cloud to remove noise and properly orient it. Figures 2.2.4 and 2.2.4 demonstrate that much of the noise is filtered out. The edges of walls are cleaner, and the obstacle bounds are more properly defined. In some cases, possibly too much information is removed, but not enough to seem as though the path is clear.

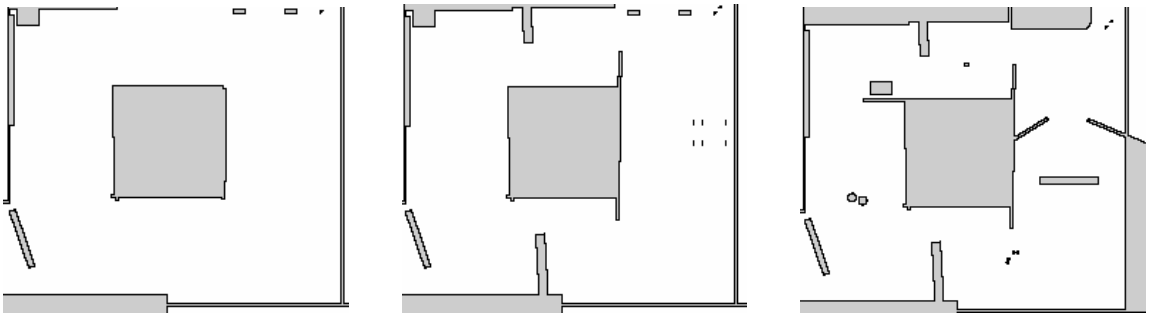

Figure 3.1: Ground truth maps: easy, medium, hard.

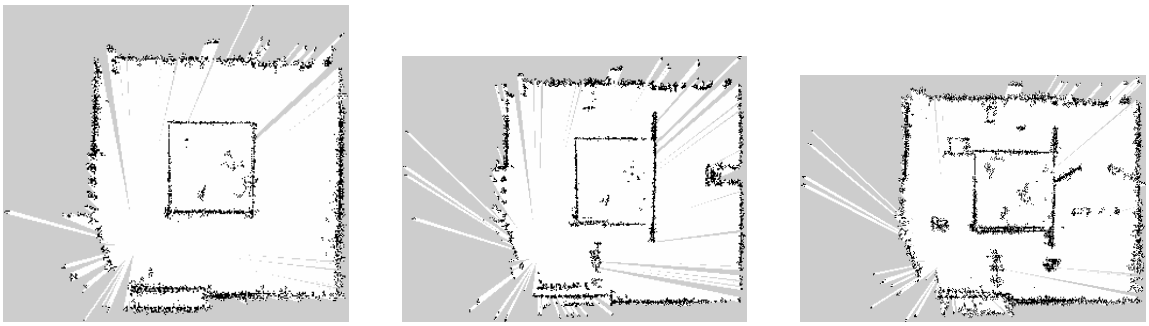

Figure 3.2: OpenVSLAM generated maps: easy, medium, hard.

3.2.2 Occupancy Grids

To qualitatively evaluate how the system was performing, a comparison was drawn between the ground truth map and the pipeline's output. The ground truth images in Figure 3.2 depict what a perfect occupancy grid of the environment would look like. The robot was used to collect image data in two passes of the environment, and locations with multiple splits in the each side were mapped on the first or second pass. The ground truth images come from a plugin [1] which takes a cross section of the environment, and outputs the occupancy grid of that cross section. It is important to note that the cross section is at wheel-height for the vehicle (middle right) in the medium world, so the only the tires show in the occupancy. All occupancy grids were created with a 0.1 resolution.

In the easy world file, the main objects to recognize are well recognized, and through processing, an appropriate occupancy grid is created. The central walls are properly squared off and the outer walls are appropriately reconstructed.

With regards to the medium world file, we see a similar level of success. The first obstruction the robot encounters is a jersey barrier. Some of the right side of it is underrepresented, but the general structure is captured. The extended walls at the right side of the central walls are well represented. The jersey barrier encountered at the top of the image is not well captured, but it is captured well enough that navigation would avoid it, though it is still important to note the lesser representation.

The final test is with the hard world. The point cloud again is quite accurate to the ground truth, and most of this information is well represented in the occupancy map. The outer and inner walls are mostly well defined. The barrier before the angled jersey barriers on the right does not see great

[1] Plugin found at: `https://github.com/marinaKollmitz/gazebo_ros_2Dmap_plugin`

representation, but it is enough to dictate avoidance. The other smaller obstacles get well clustered.

3.3 Quantitative Evaluation

In order to demonstrate the effectiveness of the mapping, the maps are used in navigation tasks and compared against the performance of navigation using the ground truth maps.

Three world files were created to test the system on increasingly difficult environments. The worlds (easy, medium, hard) were the same base environment, and difficulty was increased by adding obstacles, and arranging them in ways to disrupt paths. Two passes of the environment were done for OpenVSLAM to map, and then process. During evaluation, the bot was sent three navigation goals that would loop the environment and come back to the start. The order was rotated half the time to switch between moving clockwise and counterclockwise about the loop. While navigating, the time and distance covered were tracked, as well as the result (successful or not). This process was done several times with the ground truth map, and the OpenVSLAM based map. The easy world was a clear path, proving that the system could handle simpler navigational tasks. The medium map forced the robot to weave left and right to show more robust mapping and navigation capabilities. The hard map was much more difficult, as it had less simply shaped obstacles (i.e., a jersey barrier compared to a walking human model).

The ground truth performance in this evaluation (on successful trials) shows what performance can be maximally attained. In distance and time, the system is able to achieve very close results, displayed in Table 3.3.

In the easy environment, the average time only differs by one one hundredth of a second, and the average distance covered is within half a meter. This indicates that in that environment, the map created by the system was nearly identical in terms of navigation effectiveness. The success rate is not 100% for the ground truth map, but is for the OpenVSLAM based map, which is a consequence of the navigation stack being suboptimally tuned for the robot in use. The ground truth maps have very well defined walls, and the path planning sends the robot very close to these walls. Therefore, when it strays to far from the path, it contacts the wall and is occasionally unable to recover. With the OpenVSLAM generated maps, there is noise in the map, and the artifacts result in the wall at times being represented as a slightly larger (or less uniform) than it is. Therefore, path planning is actually more distant to the wall and there is more room for error. The success rate is effectively an indicator of how many trials yielded useful readings, as six maps were all accurate and clean enough to see successful navigation.

On the medium world, a similar performance was attained, but with slightly lesser results. The distance and timing averages were not too far off, and were within 5% of the ground truth performance. The hard world fell victim to the inaccuracies in path adherence. The robot struggled to stay on path, and mostly made irrecoverable contact with obstacles. Therefore, the statistics lose a lot of significance. More testing could have been done for the hard world, but due to time constraints, it was determined to be a less effective use of time than more thorough evaluation on other aspects of the system.

Map Generator	World	Trials	Success	Distance	Time	Speed
Ground Truth	Easy	11	0.82	45.36	102.15	0.44
	Medium	20	0.88	47.73	110.15	0.43
	Hard	6	0.14	58.625	129.13	0.45
OpenVSLAM	Easy	11	1.00	45.58	102.16	0.44
	Medium	21	1.00	49.74	114.89	0.43
	Hard	7	0.16	61.3	140.05	0.43

Table 3.1: Table of results for quantitative evaluation. *Values are averages. Units: Distance-meters, Time-seconds, Speed-m/s.*

Chapter 4

Conclusion

4.1 Summary of Results

In designing this system, the aim was to create a simulated 360°, multi-camera setup that would be compatible with OpenVSLAM's algorithms. Thus, proving the potential for lowering the barrier of entry to leveraging equirectangular VSLAM capabilities, and demonstrating it's capabilities in simulation and navigation. The goals of the system were met- the camera system was compatible, the resultant maps were accurate to the true form, and navigation in this pipeline was achieved.

The camera system established provides a solid foundation to do 360° camera work in simulation. The synchronization of the feeds, the smooth output, and quick publishing could in other tasks such as 360° deep learning tasks. The idea

The maps generated were very accurate and sufficient for navigation in a better tuned navigation stack. The maps do stand to see improvement through more sophisticated processing techniques, such as the approaches demonstrated by Ling, Shen [5]. The navigation on the harder maps failed in large part due to a suboptimal navigation configuration. Since the robot URDF used was custom, tuning the navigation stack was difficult and suffered when very precise navigation was needed. However, this is beyond the scope of this paper- demonstration of navigation and comparison between performances were more important.

4.2 Reflection on Limitations and Challenges

There were many challenges faced over the 14 weeks of development on this project, some of these challenges limited the scope or results of the project. A large limitation to the project was needing to use Ubuntu 16.04, and a powerful computer. OpenVSLAM recommends usage with Ubuntu 16.04, and after getting a decent ways in and testing with 20.04, it was determined necessary to use 16.04. This wouldn't be too big of an issue, but for a while Ubuntu 16.04 was proving to be difficult to install on my desktop and not having access to the school of computing's resources made this a bigger issue. I tested virtual environments and using a laptop (which had 16.04 installed), but both were not powerful enough to run the requisite software. Eventually however, I was able to

get passed the WiFi driver issue my desktop faced when installing Ubuntu 16.04, and was able to proceed. Being restricted to development at a desktop was it's own limitation.

Several technologies being used were out of date which resulted in some issues in making all the pieces work together. Relying on Ubuntu 16.04, ROS 1, and python 2.7 brought compatibility issues. Notably, ActionLib has unresolved issues that necessitated the re-implementation of some key features for sending and tracking navigation goals. Gazebo was very unstable in this setup; about 50-60% of the time when launching it would crash. This resulted in either retrying the launch file, or if the issue persisted, needing to restart the machine. This made debugging and testing take much longer in some cases, if either required the Gazebo environment to be running.

4.3 Further Work

There are many directions to potentially expand upon this project. One idea to investigate would be other ways of doing setting up the multi-camera rigging. Following similar testing and evaluation strategies used here, setups with 6 or 8 cameras could be used to evaluate what the ideal setup is for this. Also, testing could be done to show the increase in mapping efficiency between these multi-camera setups, and a single front facing camera.

Another idea is to exploit the separate camera streams in order to easily enable parallel processing on each camera stream. This would allow for omnidirectional object recognition, or reinforcement learning to be implemented.

Given more time to improve the results on this system, better outlier detection and smoothing of edges in the point cloud could be implemented. Most research about noise reduction with and without deep learning is intended to be used on dense point clouds, but there are surely ideas to be implemented in a system such as this with sparse point cloud representations of the environment.

Appendix A

First appendix

A.1 Gazebo Environment Captures

Figure A.1: ’Easy’ Gazebo world, top down view.

Figure A.2: ’Medium’ Gazebo world, top down view.

Figure A.3: 'Hard' Gazebo world, top down view.

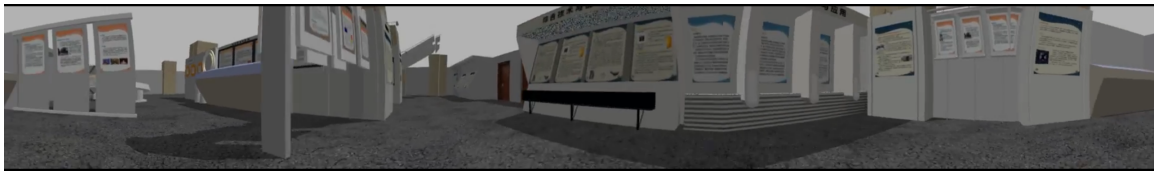

Figure A.4: Example frame of the processed view.

Bibliography

[1] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[2] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–110, 2006.

[3] Michael Kaess and Frank Dellaert. Probabilistic structure matching for visual slam with a multi-camera rig. *Computer Vision and Image Understanding*, 114(2):286 – 296, 2010. Special issue on Omnidirectional Vision, Camera Networks and Non-conventional Cameras.

[4] Adam Kalisz, Florian Particke, Dominik Penk, Markus Hiller, and Jörn Thielecke. B-slam-sim: A novel approach to evaluate the fusion of visual slam and gps by example of direct sparse odometry and blender. pages 816–823, 01 2019.

[5] Y. Ling and S. Shen. Building maps for autonomous navigation using sparse visual slam features. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1374–1381, 2017.

[6] Megan R Naminski. An analysis of simultaneous localization (slam) algorithms. Jul 2013.

[7] P. Newman and Kin Ho. Slam-loop closing with visually salient features. volume 2005, pages 635 – 642, 05 2005.

[8] Xiaojuan Ning, Fan Li, Ge Tian, and Yinghui Wang. An efficient outlier removal method for scattered point cloud data. *PLOS ONE*, 13:1–22, 08 2018.

[9] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[10] 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, pages 2292–2295, New York, NY, USA, 2019. ACM.