MERGING OF OCTREE BASED 3D OCCUPANCY GRID MAPS

CONVERGENCE CARTOGRAPHIQUE PAR ARBE OCTAIRE DES QUADRILLERS D'OCCUPATION 3D

A Thesis Submitted

to the Division of Graduate Studies of the Royal Military College of Canada

by

James Pieter Jessup, B.Eng., RMC
Acting Sub-Lieutenant

In Partial Fulfillment of the Requirements for the Degree of

Master of Applied Science

April, 2013

Abstract

Jessup, James Pieter. M.A.Sc. Royal Military College of Canada, April, 2013. *Merging of Octree Based 3D Occupancy Grid Maps*. Supervised by Dr. Sidney Givigi and Dr. Alain Beaulieu.

A technique for merging 3D octree occupancy grid maps into a single global map of an evironment is proposed and implemented in software. Octrees are a memory efficient way to represent a 3D environment by recursively subdividing space at multiple depths in a tree structure. The use of an octree representation of a 3D environment allows large environments to be mapped while limiting the amount of memory used in comparison to other techniques. When multiple robots are used to map an environment, a more accurate map of a larger space can be produced in less time. Techniques are introduced to address information from multi-depth sources in each map's tree structure as well as techniques to address relative transformations between maps that are not axis aligned. These techniques allow the octree representation of an environment to be extended to multi-robot applications, specifically those situations where relative map reference frame transformations are unknown prior to mapping. Given the flexibility of this work to situations involving no prior knowledge of map transformations, this work also explores the problem of merging maps when the sources of map transformations are uncertain. Therefore registration techniques using commonly mapped portions of the environment to obtain a better estimate of transformations between maps are explored. The application of these techniques is demonstrated by merging maps built by robots in simulated and real-world environments. The results of this work show that the techniques proposed in this work to merge octree based occupancy grids are valid and that an octree based occupancy grid is a suitable map representation for multi-robot problem spaces requiring a 3D

model of the environment.

Keywords: Octrees, Mapping, Simultaneous Localization and Mapping (SLAM), Cooperative Robotics, Navigation, Localization, Computer Vision, Data Fusion

Résumé

Jessup, James Pieter. M.Sc.A. Collège militaire royal du Canada, Avril, 2013. *Convergence cartographique par arbe octaire des quadrillers d'occupation 3D.* Thèse dirigée par M. Sidney Givigi, Ph.D. et M. Alain Beaulieu, Ph.D.

' Une technique pour la fusion de deux mappes représentées en forme de grilles d'occupation octree en 3D vers une seule mappe globale d'un environnement est proposée et implémentée en logiciel. Les Octrees sont efficaces en utilisation de la mémoire pour représenter un environnement 3D en sous-divisant récursivement l'espace dans une structure d'arbre à profondeurs multiples. L'utilisation de la représentation octree d'un environnement 3D nous permet de mapper de vastes environnements tout en limitant la mémoire utilisée en comparaison avec les autres techniques. Lorsque plusieurs robots sont utilités pour mapper un environnement, une mappe plus précise d'un espace plus grand peut être générée en moins de temps. Des techniques sont introduites pour dénouer l'information de différentes sources qui ont une profondeur différente dans chaque structure ainsi que des techniques qui visent les transformations relatives entre les mappes qui ne sont pas alignées sur leurs axes. Ces techniques permettent à la représentation par octree d'un environnement à être portée aux applications multi-robots, spécifiquement dans les situations où les transformations de cadre de référence relatif à chaque mappe sont inconnues avant le mappage. Étant donné la flexibilité de ce travail qui adressent les situations qui impliquent aucune connaissance des transformations des mappes avant le mappage, ce travail explore aussi le problème de la fusion de mappes quand les sources de transformations des mappes sont incertaines. Donc les techniques de régistration qui utilisent des portions communes mappées de l'environnement pour obtenir un meilleur estimé des transformations entre les mappes sont aussi examinées. L'application de ces techniques est démontrée en fusionnant des mappes construites par des robots dans des

environnements simulés et dans le monde réel. Les résultats de ce travail démontrent que les techniques proposées dans ce travail pour faire la fusion des mappes en forme de grilles d'occupation octree sont valides et que les grilles d'occupation octree sont une représentation satisfaisante pour les mappes générées par un espace couvert par plusieurs robots qui ont besoin d'un modèle de l'environnement en 3D.

Mots clés : Arbes octaires, Mappage, Localisation et mappage simultané, Robotique coopérative, Navigation, Localisation, Vision artificielle, Fusion de données

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols

M	A set of m maps to be merged into one global map
M'	The result of the merger of a set of maps, M
$M_2 \cup M_1$	The merger of the map M_2 into M_1
n_i	A leaf node element corresponding to a voxel in the map M_i
T_1^2	A transformation matrix from map 2 to map 1's frame of reference.
$M_2^{T_1^2}$	Map 2 transformed into the reference frame of map 1.
$P(n)$	The occupancy probability of a voxel corresponding to leaf node n
$\boldsymbol{x}$	A state vector which is an element of a feature map
$C(\boldsymbol{x})$	The covariance matrix of the state vector $\boldsymbol{x}$
$L(n)$	The log-odds occupancy of the voxel corresponding to leaf node n
$L(n\|z_t)$	The log-odds occupancy of node n from an observation z at time t
$L(n\|z_{1:t})$	The log-odds occupancy of node n for all observations from $t = 1$ to $t = t$
l_{min}	The lower log-odds clamping threshold for octree map compression
l_{max}	The upper log-odds clamping threshold for octree map compression
P_{Cl}	A point cloud
$\boldsymbol{p}_i$	The ith element of a point cloud
C	A final interpolated value from consecutive interpolations
x_d	The displacement of an interpolated and nearest source point in the x-axis
y_d	The displacement of an interpolated and nearest source point in the y-axis
z_d	The displacement of an interpolated and nearest source point in the z-axis
$T_{R(\alpha)}$	A 2D rotation in the image plane by an angle α.
W	A "data" point cloud to be used for Iterative Closest Point registration
X	A "model" point cloud to be used for Iterative Closest Point registration
Y	The set of points in W corresponding to points in X
$\boldsymbol{q}$	An Iterative Closest Point registration state vector
$\boldsymbol{q}_R$	A unit quaternion rotation vector, the rotation part $\boldsymbol{q}$
$\boldsymbol{q}_T$	A 3D vector, the translation part of $\boldsymbol{q}$
d_{ms}	The mean-square registration error from an Iterative Closest Point iteration
T_c	The transformation of the pose of the current robot to its own map
T_o	The transformation of the pose of the other robot to its own map
T_r	The transformation of the pose of the current robot to the pose of the other robot
ϵ_r	The error metric for the rotation part of the refined transform
ϵ_t	The error metric for the translation part of the refined transform

List of Acronyms

AI Artificial Intelligence

DOF Degrees of Freedom

EKF Extended Kalman Filter

ICP Iterative Closest Point

PCL Point Cloud Library

ROS Robot Operating System

SLAM Simultaneous Localization and Mapping

TCP Transmission Control Protocol

UAV Unmanned Aerial Vehicles

UGV Unmanned Ground Vehicles

UUV Unmanned Underwater Vehicles

Chapter 1

Introduction

In order for robots to be autonomous they need to build maps of their environment. The Simultaneous Localization and Mapping (SLAM) problem arises when a robot is placed in an unknown environment and must estimate a map of its surroundings as well as its position and orientation (pose) relative to the map. SLAM is a highly active research topic in the Artificial Intelligence (AI) and robotics communities. This is not a trivial problem since determining the robot's position typically requires knowledge of its surroundings, and in addition, building a map of the surroundings requires knowledge of the robot's pose. In order to build a map, the robot must traverse an unknown environment and take measurements of its surroundings and new position incrementally. Due to uncertainty and noise in measurements from sensors, errors will accumulate over time and distort the map, which will in turn distort the robot's estimate of its position in the map.

To date many authors have proposed techniques to account for errors in measurements that occur with SLAM. These techniques use a stochastic approach to create a best-estimate for maps and robot position to reduce accumulated errors. Approaches to SLAM have evolved since their introduction in 1990 [3]. These approaches have evolved primarily in two ways. As the number of tasks performed by autonomous robots has grown, as well as the complexity of their environments, the SLAM problem space has grown from two to three dimensions. In addition, the physical size of unknown environments to be explored and mapped has increased. In order to map a large area in a reasonable amount of time, multiple robots are now being used to map an area.

Currently the state of the art in mapping and localization involves the exploration of three dimensional spaces where a robot is free to move with a full six Degrees of Freedom (DOF). This is a critical endeavour since autonomous vehicles such as Unmanned Aerial Vehicles (UAV), Unmanned Underwater Vehicles (UUV),

and Unmanned Ground Vehicles (UGV) in complex environments currently explore a three dimensional problem space. However, generalizing a robot's map and localization techniques to three dimensions presents significant challenges in limiting memory consumption and required processing power. One approach to mitigate these issues involves the use of an octree based map representation [4, 5, 6, 7, 8, 9]. This approach is able to accurately map large spaces with efficient memory usage. Despite this representation's many advantages, it has not been extended to a distributed approach with multiple robots building their own maps and merging them upon rendez-vous. In order to build maps of large spaces quickly and more accurately, this approach can be extended to multiple robots where the results of each robot's individual maps are merged into one global map.

1.1 Motivation

This thesis is motivated by the need to create accurate 3D maps from collaborative and autonomous platforms deployed in an unknown environment. These maps must be built in an efficient and accurate manner that reduces memory usage and processing power.

1.2 Thesis Topic

Our research goal is to investigate the implementation of a map merging algorithm which merges maps with a 3D representation of the environment using octree occupancy grids for a memory efficient representation of the environment. Further, the pursuit of map merging techniques in a distributed manner with a discretized model of the environment implies that independent robots will not build maps with perfectly aligned discretizations due to the fact that they are unaware of each other's reference frames prior to beginning the mapping process. Therefore, the exploration of interpolation techniques to merge maps with different discretization alignments is of particular interest. Additionally, the nature of octree based mapping means that

commonly mapped areas may be mapped at different resolutions due to the individual mapping perspective of each robot. Therefore, the exploration of techniques to address the problem of determining the best estimate of an environment from information of multiple resolutions is also of interest. Finally, since transformations between maps to be merged are subject to error since they are derived from noisy sensor observations, techniques to obtained improved transform estimates by aligning commonly mapped areas of the environment using data from each map is also of interest.

1.3 Contributions

Our contributions through this research are as follows:

1. Proposal of theoretical algorithms and rules for merging octree occupancy grid maps which address:
   (a) The merger of commonly mapped areas which are mapped at different depths in the octree.
   (b) The merger of octree maps with misaligned discretizations.
   (c) The refinement of erroneous transformation estimates using map data from commonly mapped areas of the environment.
2. Implementation of the proposed algorithms in software.
3. Verification and validation of the software in the Gazebo simulation environment.
4. Verification and validation of the software on a physical platform (Turtlebot UGV).

1.4 Thesis Organization

This thesis is organized in the following chapters:

- In Chapter 2, an introduction to the problem of mapping as it pertains to autonomous robotics is presented.
- In Chapter 3, an introduction to the problem of creating a global map from the maps of multiple robots is presented.
- In Chapter 4, the software resources used to support this research are described.
- In Chapter 5, the problem of octree based occupancy grid map merging is formally defined.
- In Chapter 6, we describe our algorithm for octree based occupancy grid map merging and its implementation in software.
- In Chapter 7, we provide a summary of the experimental results, as well as the validation and verification of the merged maps. We also discuss the hardware and supporting software used to obtain these results.
- In Chapter 8, we present conclusions based on the results gathered and summarize contributions and future work.

Chapter 2

Mapping

In this Chapter, we give a brief overview of mapping strategies in the literature, followed by an outline of the basic concepts in the use of octrees as it applies to this thesis.

2.1 The Stochastic Map

Smith et al. are the first authors in the literature to propose the use of a stochastic solution for representing a robot's environment [3]. Prior to their work, accurate navigation relied on highly accurate, but also expensive sensors and controlled environments. The authors in this work argue that an alternative approach would be to combine multiple sensors or measurements including their uncertainty to obtain a better estimate. Therefore, their work presents a way to represent a map which takes into account uncertainty in measurements, as well as procedures to read and build the map. Not only do the authors build the mathematical model for representing uncertainty in the map, they consistently support their model with an example of how one would apply their solution with a robot making observations about its surroundings as it passes through an unknown space.

2.2 The Simultaneous Localization and Mapping (SLAM) Problem

The aforementioned problem of a robot traversing an environment and estimating a map of its surroundings as well as its position in the map is known as the SLAM problem. Smith et al. use the Extended Kalman Filter (EKF) as an estimator to provide the best estimate of robot's map and its location within that map. Subsequent improvements have been made to this by Murphy in [10]. Murphy proposed that the SLAM problem be approached from a Bayesian perspective and that Monte Carlo methods or a particle filter be used to estimate the robot's map and its position within

the map. Murphy's proposal has subsequently been implemented by Montemerlo et al. in [11, 12] as the FastSLAM Algorithm.

2.3 Map Representations

For mapping scenarios which consider 2D space, the most common map representations are the feature map and the occupancy grid map. These representations have grown to become the *de facto* standards for map representations in robotics [13]. while these representations are proven for 2D environments, they bring their own challenges when extending the representation of the environment to 3D.

2.3.1 The Feature Map

The feature map was the first of the two representations commonly used in mapping, originally proposed in the work of Smith et al. [3]. The feature map contains a collection of uniquely identifiable landmarks represented as vectors $\boldsymbol{x}$ containing their position, x and y, and orientation ϕ. Their uncertainty is represented using covariance matrix $C(\boldsymbol{x})$:

$$\boldsymbol{x} = \begin{bmatrix} x \\ y \\ \phi \end{bmatrix} \qquad , \qquad C(\boldsymbol{x}) = \begin{bmatrix} \sigma_{x^2} & \sigma_{xy} & \sigma_{x\phi} \\ \sigma_{xy} & \sigma_{y^2} & \sigma_{y\phi} \\ \sigma_{x\phi} & \sigma_{y\phi} & \sigma_{\phi^2} \end{bmatrix} \tag{2.1}$$

Maps are then built by adding additional elements to the collection of landmarks, or by making additional observations of existing landmarks. These additional measurements are incorporated into the map as constraints to the system typically with *Kalman Filtering* to improve the best estimate of the map.

2.3.2 The Occupancy Grid

In the case of an occupancy grid, the map contains an arbitrary grid of cells containing the likelihood of whether or not that grid is occupied. This representation was first proposed in [14]. In an occupancy grid, the environment is discretized into a set of cells of a given size. Each cell in the grid contains a probability of whether it is filled or empty. In the case of range sensors, cells where a measurement ray passes through

unobstructed will be observed as empty, and cells that contain the end of the ray will be observed as occupied. Occupancy grid representations often take into account unknown territory, where grid cells may include counters to account for areas of the environment that do not correspond to any received sensor observations. An example of an occupancy grid is shown in Figure 2.1. Where, black represents occupied cells, white represents empty cells, and grey represents unexplored territory.

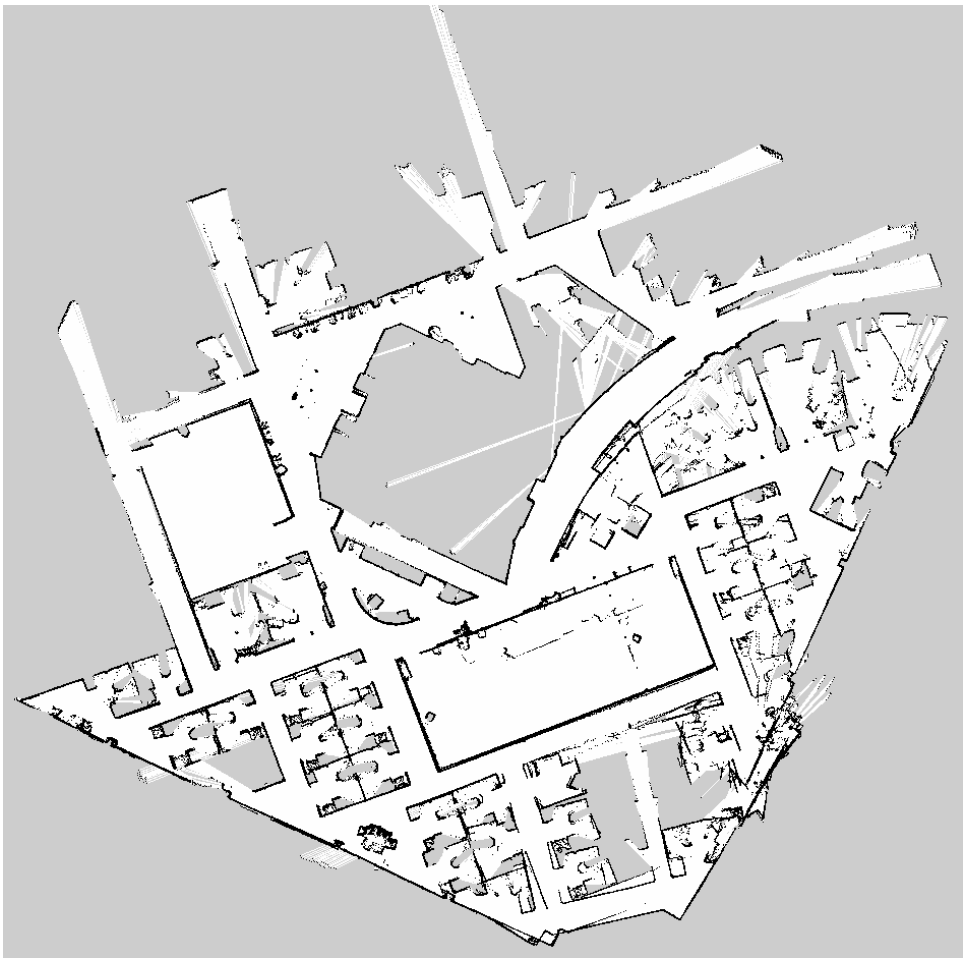


Figure 2.1: An example of an occupancy grid map (used from [1] with permission)

While an occupancy grid map is built, grid cells in the map corresponding to new sensor observations are updated according to a *binary Bayes Filter* [15]. The belief of a particular cell, n, in the grid to be occupied given a set of measurements, $z_{1:t}$, is denoted:

$$bel_t(n) = P(n|z_{1:t}) \tag{2.2}$$

Often a *log-odds* representation of a cell's occupancy is used, where the *odds* of cell, n, to be occupied is defined as:

$$\frac{P(n)}{P(-n)} = \frac{P(n)}{1 - P(n)} \tag{2.3}$$

And the log-odds belief, $L(n)$, is the logarithm of this ratio:

$$L(n) = \log\left[\frac{P(n)}{1 - P(n)}\right] \tag{2.4}$$

Using a binary Bayes Filter as described in [15], the odds of a particular cell is updated according to:

$$\frac{P(n|z_{1:t})}{P(-n|z_{1:t})} = \frac{P(n|z_t)}{1 - P(n|z_t)} \frac{P(n|z_{1:t-1})}{1 - P(n|z_{1:t-1})} \frac{1 - P(n)}{P(n)} \tag{2.5}$$

Where $P(n|z_{1:t})$ is the new occupancy probability of the cell, $P(n|z_{1:t-1})$ is the cell's previous estimate, $P(n|z_t)$ is the probability that the cell is occupied given the current measurement obtained through a sensor model, and $P(n)$ is the initial assumption of the cell's occupancy.

Combining a log-odds belief representation with a common uniform cell prior probability of $P(n) = 0.5$, equation 2.5 can be rewritten:

$$L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t) \tag{2.6}$$

Using a log odds representation for cell occupancy allows for faster updates since additions computed instead of multiplications. In addition sensor models may be precomputed so that the update step does not require the computation of logarithms for each update.

2.3.3 Representing an Environment in 3D

When one considers how to model an environment in 3D space several approaches have been adopted. As described in [8] the choice of map representation must satisfy the criteria of a probabilistic representation, which models free, occupied, and unknown territory, as well as an efficient implementation with respect to runtime and memory usage.

One popular approach is to extend an occupancy grid to 3D and use a grid of cubic volumes of equal size, called voxels to subdivide the volume to be mapped [16]. This approach suffers from three difficiencies: for large spaces it has a high memory requirement; it cannot represent spaces in finer resolutions when required for precise tasks; and finally, there is a need to know the extent of the environment prior to mapping.

Another approach which does not discretize the environment is the use of 3D point clouds [17, 18]. A point cloud, P_{Cl}, is defined as a set of points $\boldsymbol{p}$.

$$P = \{\boldsymbol{p_0}, \boldsymbol{p_1}, \ldots, \boldsymbol{p_{n-1}}, \boldsymbol{p_n}\} \tag{2.7}$$

Where, an element of the point cloud $\boldsymbol{p_i}$ is a 3D point.

$$\boldsymbol{p_i} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.8}$$

Often additional data may be included in each point such as intensity, color, or a unit normal, for processing purposes. When this is the case, the length of the point vector $\boldsymbol{p}$ and this data is added as another element of the vector. This approach however does not model free space or unknown areas, and requires increased memory usage for every sensor observation. Finally an additional approach involves the creation of 2.5D elevation maps. In this case a 2D grid stores the elevation of a particular cell. This approach is memory efficient, however, it does not represent space in a volumetric way. It is therefore not sufficient for representing the actual environment for applications such as localization in 3D spaces.

2.4 Octree Based Maps

One popular topic in literature is the use of an octree based occupancy grid representation to map a robot's environment in 3D space [4, 5, 6, 7, 8, 9]. This approach is volumetric and avoids many of the drawbacks of approaches using a 3D grid of fixed size voxels. An octree is a hierarchical data structure for spacial subdivision first proposed in the computer graphics community for efficient rendering of 3D volumes

[19]. An octree is a collection of nodes which discretizes a space into voxels which then are recursively subdivided into eight sub-volumes or child nodes until a desired resolution is reached. The subdivision of space and the tree structure of an octree is shown in Figure 2.2. In an octree map, each node contains the occupancy probability of the respective volume.

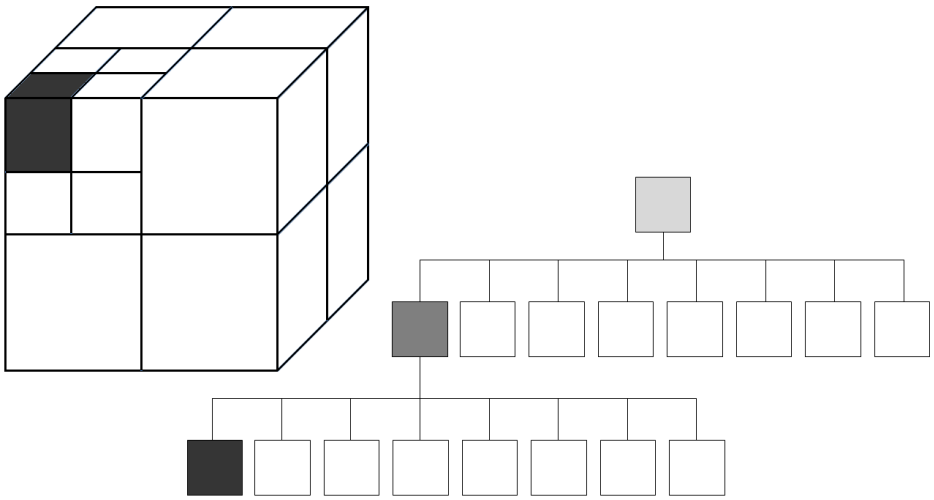


Figure 2.2: The subdivision of space and the tree structure of an octree

Hornrung et al. have implemented such an octree mapping framework as an open source software library and integrated it into Robot Operating System (ROS) [8, 9]. A key contribution to their approach is the implementation of a compression method to reduce the memory requirement of a map. In their approach, a map is built at the lowest subdivision of the tree structure. In [8, 9], sensor observations are incorporated into map such that the log-odds occupancy of a node, n, is clamped to be within an upper and lower bound, l_{min} and l_{max} respectively:

$$l_{min} \leq L(n) \leq l_{max} \tag{2.9}$$

This clamping strategy keeps the confidence in the map bounded while simultaneously allowing the map to be more quickly adaptable to dynamic environments. Using this clamping policy, it is shown in [8] that the map update formula described in Equation

(2.6) becomes:

$$L(n|z_{1:t}) = \max(\min(L(n|z_{1:t-1}) + L(n|z_t), l_{max}), l_{min}) \tag{2.10}$$

This clamping policy allows the compression of the map by pruning, or removing nodes from the octree. When the occupancy of a node in a tree reaches either l_{min} or l_{max} it is considered stable. If all the children of an inner node in the map have reached the same stable state those children are pruned from the map. Pruning is efficient since it reduces the number of nodes to manage in the map. Furthermore, by applying this compression method, the only probability information that is lost is information close to $P(n) = 0$ and $P(n) = 1$.

2.5 Multi-Robot Mapping

In many cases it is necessary to use multiple robots to map a large space within a reasonable time constraint. Therefore, it is necessary for robots to merge the results of their individual maps into one large global map. The first work to consider multiple robots mapping a large space used a common reference frame where relative poses for each robot were known prior to mapping [20]. As it may not be reasonable to assume that a team of robots mapping an area will have the same world reference, it is necessary to determine a transformation from one robot's frame of reference to the other's. This involves the calculation of the relative pose of each robot's reference frame as well as the transformation of the information in one map to the other map's frame of reference. In the two dimensional case any relative pose can be measured by a rotation of some angle θ and a translation by some vector $\boldsymbol{t}$ [21]. In two dimensions the transform for points x and y from one robot's map into points x' and y' in the other's can be performed by one matrix multiplication using homogeneous coordinates:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.11}$$

This concept of using matrix multiplication to transform from one reference frame to another can also be generalized to $\mathbb{R}^3$ to support 3D maps. This however provides

added difficulty in accounting for additional degrees of freedom and for when a discretized representation of the environment is used.

2.6 Conclusion

This chapter presented an introduction to stochastic mapping and the SLAM problem. Different ways to represent the environment were discussed for both two and three dimensional problem spaces, identifying octree occupancy grids as a suitable choice for 3D mapping. Additionally, the problem of mapping with multiple robots is introduced. This problem is discussed further in Chapter 3.

Chapter 3

Map Merging

In this chapter the problem of merging maps from multiple robots is discussed. A review of literature pertaining to the process of merging maps is also presented.

3.1 Multi Robot Mapping Strategies

As previously mentioned the easiest way to map a large area with high accuracy within a reasonable time constraint is to incorporate multiple mapping robots. In this case it is necessary for the mapping robots to merge their maps into one global map of the area. Early strategies solved the merging problem by using a team of robots, with each robot building maps within a common reference frame [20]. This solution is attractive since it is not all that different than single robot mapping. However it is not a flexible solution since it might not be a reasonable assumption that all robots will have the same reference frame.

More flexible solutions have presented themselves in the literature. One solution involves a centralized approach to map merging where maps are merged once the whole environment has been explored [22]. Another solution is a decentralized approach where maps are merged upon robots meeting one another while traversing the environment [21]. Recently, reinforcement learning has been applied with a decentralized approach so that maps will be merged only in situations where a better estimate of the explored environment is obtained [23].

Regardless of the approach to map merging, when robots build maps in independent reference frames, the merger requires the calculation of a transformation from one robot's reference frame to the other. This transformation can be represented by a matrix multiplication of the same form as shown in Equation (2.11). This thesis will focus on the merging process rather than the mapping strategy. Therefore, any mapping strategy that does not use a common world reference frame will be suitable

for this work.

3.2 Map Transformations

Once a suitable transformation matrix is found it is necessary to perform this transformation so that a global map can be obtained. The implementation of this transformation depends largely on the map representation itself.

For feature maps each state vector of the map is transformed using the matrix multiplication described in Equation (2.11). In the case of an occupancy grid, the matrix multiplication cannot be performed directly on the source elements due to the environment discretization. Since its representation is equivalent to that of a pixel image, the same geometric transformation techniques used for image processing can be applied to the occupancy grid.

When transforms are performed by mapping source pixels to their destination with transforms that include rotation, often the transformed image or map is no longer continuous since some destination pixels are not addressed. This phenomenon is known as *aliasing*. One common method to address the aliasing problem is to determine the range of destination pixels that will be present in the transformed image, then, for each destination pixel, determine the location of the source pixel by the inverse transformation. Since this inverse transform will often not directly map to integer pixel locations, interpolation techniques such as nearest neighbour, bilinear, or bicubic interpolation may be used to determine the destination pixel's final value. Bilinear interpolation is often used due to it's superior results over nearest neighbour interpolation, while offering easier implementation and reduced computational complexity over other options such as bicubic or cosine interpolation [24].

Bilinear interpolation is a method to interpolate between pixel values on a 2D grid. In the case of a 2D occupancy grid, the pixel values would represent the occupancy of each cell in the grid. When a destination pixel is transformed to a non-integer source pixel coordinate, the final value is interpolated from the four closest surrounding pixels, using their centre points as coordinates.

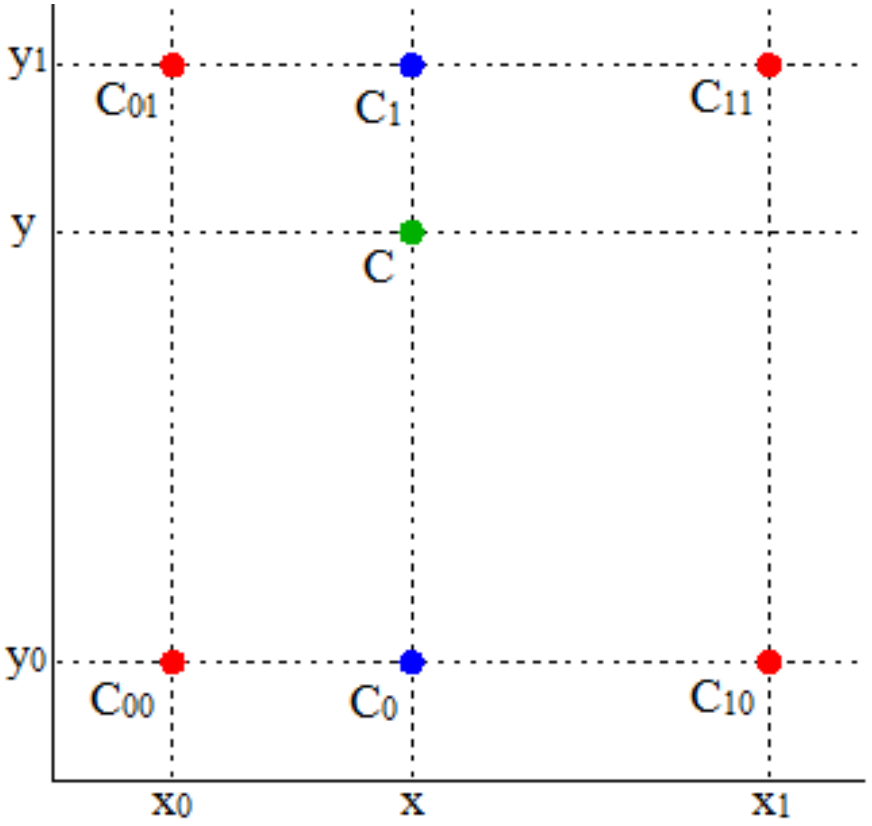


Figure 3.1: An illustration of the process of bilinear interpolation.

One simple way to perform bilinear interpolation is by performing two consecutive linear interpolations as shown in in Figure 3.1. In this example, the final scalar value to be interpolated, C, located at (x, y), falls between four source values, C_{00} to C_{11} located within the square bounded by the coordinates (x_0, y_0), (x_1, y_1). Firstly, linear interpolation is performed in the x-direction, obtaining the values C_0 and C_1:

$$C_0 = (1 - x_d)C_{00} + x_d C_{10} \tag{3.1}$$

$$C_1 = (1 - x_d)C_{01} + x_d C_{11} \tag{3.2}$$

Where,

$$x_d = \frac{x - x_0}{x_1 - x_0} \quad , \quad y_d = \frac{y - y_0}{y_1 - y_0} \tag{3.3}$$

Finally the result of the first linear interpolation can then be interpolated in the y-direction obtaining the destination pixel's final value:

$$C = (1 - y_d)C_0 + y_d C_1 \tag{3.4}$$

$$\tag{3.5}$$

This process can also be done with 3D grids using trilinear interpolation in the same fashion with three consecutive linear interpolations. In this case, the final value of a destination voxel is interpolated between eight neighbour voxels. An illustration is shown in Figure 3.2. When applying this technique to a 3D grid map, the location of the values to be interpolated between , C_{000} to C_{111}, represent the centre-points of nodes in the map, and their values would be equal to the occupancy of each node.

For example, one could perform the first interpolation along the x-axis as:

$$C_{00} = (1 - x_d)C_{000} + x_d C_{100} \tag{3.6}$$

$$C_{10} = (1 - x_d)C_{010} + x_d C_{110} \tag{3.7}$$

$$C_{01} = (1 - x_d)C_{001} + x_d C_{101} \tag{3.8}$$

$$C_{11} = (1 - x_d)C_{011} + x_d C_{111} \tag{3.9}$$

Where,

$$x_d = \frac{x - x_0}{x_1 - x_0} \quad , \quad y_d = \frac{y - y_0}{y_1 - y_0} \quad , \quad z_d = \frac{z - z_0}{z_1 - z_0} \tag{3.10}$$

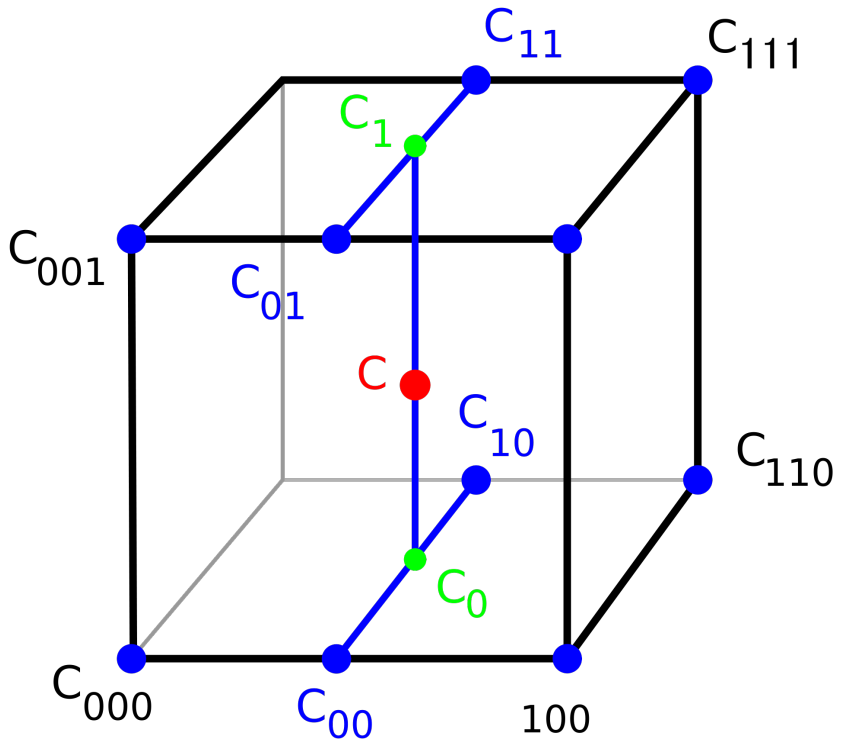


Figure 3.2: An illustration of the process of trilinear interpolation.

The second interpolation along the y-axis:

$$C_0 = (1 - y_d)C_{00} + y_d C_{10} \tag{3.11}$$

$$C_1 = (1 - y_d)C_{01} + y_d C_{11} \tag{3.12}$$

With the final value of the destination voxel using a third interpolation in the z-axis as:

$$C = (1 - z_d)C_0 + z_d C_1 \tag{3.13}$$

One other interesting method to overcome the aliasing problem is the technique proposed by Paeth [25] and Tanaka et al. [26]. Which decomposes a rotation in the image plane into three consecutive shear transformations, expressed as:

$$T_{R(\alpha)} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\tan\frac{\alpha}{2} & 1 \end{bmatrix} \begin{bmatrix} 1 & \sin\alpha \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\tan\frac{\alpha}{2} & 1 \end{bmatrix} \tag{3.14}$$

Essentially, this transformation is a cascade of three shifts along the rows and columns of a 2D grid. In the first transformation, the x-coordinates of each row are shifted by:

$$x' = x - y\tan\frac{\alpha}{2} \tag{3.15}$$

Then the second shear is performed by shifting the y-coordinates of each column by:

$$y' = y + x \sin \alpha \tag{3.16}$$

And finally, the final shear is performed by the same shift described in Equation (3.15).

This method avoids the aliasing problem while still mapping source pixels to destination pixels because entire rows of the grid are shifted contiguously. Additionally since these shifts are performed on a raster grid, shifts are only expressed in integer values. Linear interpolation may then be used for the final value of the destination pixels.

This method has also been extended to three dimensions. Initially this was performed by simply performing nine consecutive shears, by expressing each rotation about a major axis as a product of three shears [27]. Subsequently this method was also improved by reducing the number of shear transformations required to perform a general rotation by performing shears along "slices" and "beams" of the volume [28].

For this work, trilinear interpolation is used. The choice of trilinear interpolation was chosen over consequtive shear transformations was due to ease of implementation. Initial success with shear transformations was acheived, however equivalent development effort with trilinear interpolation acheived an equivalent result at much less computational expense.

3.3 Improving the Transformation Estimate

The previously mentioned methods for merging maps apply when an exact transformation is known between the maps. Often this is not the case since obtaining the transformation frequently relies on noisy sensor measurements that in many cases are not accurate. When this is the case, data from the map itself is used to identify commonly mapped areas to improve the transform by finding the best alignment of those areas.

3.3.1 The Iterative Closest Point Algorithm

One method for aligning 3D environment data is the Iterative Closest Point (ICP) algorithm [29]. This algorithm uses two rigid bodies and an initial guess of their alignment and iteratively minimizes an error metric.

The algorithm operates by matching a "data" shape W to be in best alignment to a "model" shape X. The data and model representation may be in many forms, however points will be considered as the choice for this work. The first step of the algorithm is to determine a subset of points in X, Y that correspond to the points in W. The ICP algorithm assumes that this corresponding point in X is the point of least euclidean distance distance from the point considered in W. Where the euclidean distance from one point in W, $\boldsymbol{w}$, to another point in X, $\boldsymbol{x}$, is equal to:

$$d(\boldsymbol{w}, \boldsymbol{x}) = \|w - x\| \tag{3.17}$$

Now that a subset of points $Y \in X$ is selected, least squares registration is performed to determine the rigid transformation, consisting of a 3 x 3 rotation matrix R, and a translation vector $\boldsymbol{t}$ to minimize the mean squared distance, S, from the points in W to the corresponding points in Y, given by:

$$S(R, \boldsymbol{t}) = \frac{1}{N_w} \sum_{i=1}^{N_w} \|\boldsymbol{y}_i - R\boldsymbol{w}_i - \boldsymbol{t}\|_2^2 \tag{3.18}$$

Therefore, the parameters, R and $\boldsymbol{t}$ that yield the minimum mean squared distance must be determined using the least mean squares algorithm.

The obtained rotation matrix and translation vector to minimize the squared distance between corresponding points may be used to obtain a 4 x 4 transformation matrix given by:

$$T_i = \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.19}$$

This transformation is then applied then to W. The whole process of selecting corresponding points, and obtaining a transformation to minimize the mean square error

between corresponding points, is then repeated until the change in mean square error between iterations falls below an arbitrarily defined value. The final transformation refinement obtained from ICP registration can then be obtained by the product of the individual transformations from the n ICP iterations:

$$T_{ICP} = \prod_{i=i}^{n} T_i \tag{3.20}$$

3.3.2 3D Map Registration with the Iterative Closest Point Algorithm

There are works in the literature which discusses the use of ICP algorithm to improve the accuracy of registration of sensor observations into 3D maps [17, 18]. These works are often used during SLAM to improve the estimate of both the robot's relative pose in its surroundings as well as the sensor's pose relative to the robot's map. In these works typically point cloud representations of the environment are used and individual sensor observations are registered on a scan-by-scan basis.

One interesting work connects an octree subdivision of an environment to registration of point clouds [30]. In this work, the performance of traditional point cloud ICP registration is improved by creating an octree subdivision of the environment. This is done by using voxel centres of the most commonly observed voxels as the input points for the model and data sets. The authors demonstrate that this approach outperforms naively using the existing data for the data and model of the ICP algorithm.

While these works focus on the registration of consecutive sensor observations, the registration problem is closely related to the merging problem. For merging large point cloud maps the use of ICP would be plagued by the requirement to process a large number of points since nearly every observation is kept in the map as described earlier. However, since in this work an Octree representation is used for an efficient representation there is potential for the use of ICP to improve the transformation estimate between two Octree occupancy grid maps. This is especially true, since the performance of ICP with voxel centre points is shown to be strong [30].

Although registration and merging are similar problems, additional challenges to the merging problem are discussed in [31]. In order for registration to be successful there needs to exist both a source and template such that a transformation can be calculated to align the two data sets. In the context of mapping, this means there must be some portions of each map that represent the same parts of the environment for the use of the ICP algorithm to improve the transformation estimate. Another challenge discussed in [31] is the ability to identify regions of each map that represent the same parts of the environment. This being said, with some initial knowledge of a transformation between the two maps, an estimate of commonly mapped regions suitable for transformation improvement with ICP can be extracted from each map.

3.4 Conclusion

This chapter has presented an overview of the problem of merging maps from multiple robots into one global map. No previous knowledge about transformations between the reference frames of different robots has been shown to be the most flexible solution of multi-robot mapping. Subsequently the process for calculating the transformation estimate between the reference frames of two robots in a rendez-vous scenario is discussed. Once a transformation between reference frames is provided the problem of performing a transformation on discretized is explored. The use of the ICP algorithm on commonly mapped portions of the environment to refine this transformation estimate is also discussed. Now that a theoretical background on the problem of octree occupancy grid merging is presented, the software resources used to implement this theory in actual robots will be discussed in Chapter 4.

Chapter 4

Robotics Software Resources

This chapter describes the three principal software resources used to support this thesis, namely, ROS, Octomap and Point Cloud Library (PCL). These software resources facilitate the implementation of the theory described in Chapter 3 for merging of octree maps.

4.1 Robot Operating System

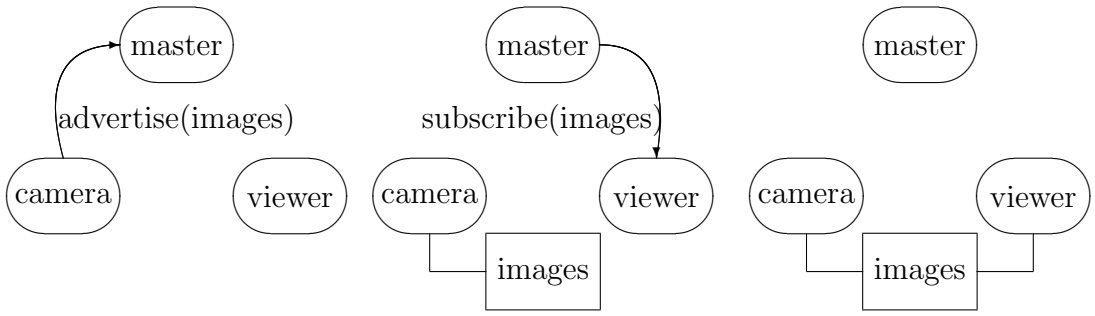
ROS is a middle-ware used to simplify the development of robotics software. ROS is an open source collection of tools, libraries, and community developed software that allows robotics software developers to be abstracted away from specific hardware. ROS accomplishes this abstraction through the following core components:

1. Communications Infrastructure
2. Robot Specific Features
3. Development Tools
4. Vast Software Ecosystem

These core components will now be discussed.

4.1.1 Robot Operating System Communications Infrastructure

A ROS system is composed of a number of nodes which are processes which perform various functions [32]. These nodes could be drivers for various sensors, tools provided by ROS itself, or even perform computation based on the data from other sensor driver nodes (i.e., path planning). These nodes are written in C++ or Python with the provided libraries, *roscpp* or *rospy* respectively. For inter-node communication,



(a) the camera node notifies the master it will publish images

(b) the viewer node notifies the master it will subscribe to images

(c) the viewer is receiving messages from the camera

Figure 4.1: Robot Operating System topic communication (used from [1] with permission).

ROS provides a message passing interface through the use of *topics* with an asynchronous publish/subscribe model or *services* using a synchronous request/response model. Messages are passed using pre-defined message formats, written in the message interface description language. To facilitate message-passing, there always exists one node on any ROS system called the master node. The master node manages communication between nodes by acting as a name server and directs the connections between nodes. The connections between nodes are most commonly Transmission Control Protocol (TCP) streams, allowing a ROS system to be distributed accross a network. The interaction with the master node for subscribing to a topic is shown in Figure 4.1 where the master node orchestrates the connection between nodes, but data transfer takes place between the nodes themselves. Multiple nodes in a system are also capable of subscribing to a topic. The master node also provides a parameter server which stores data available to every node in the ROS system.

4.1.2 Robot Operating System Robot Specific Features

ROS provides several robot-specific features to speed up the development time for robotics software. One of these features is that there exist several standard messages to cover the most commonly used use cases. These messages include, point-clouds,

poses, transforms, odometry, images, etc. By using these standard messages, a user's code can interoperate easily with previously written software contained in the ROS ecosystem.

Another useful feature provided by ROS is its description of Geometry with the *tf*, or transform, library. The *tf* library abstracts the transformation of coordinate frames to the user, while handling the fact that transform information is distributed across a network and comes from sources updated at various rates. ROS uses a tree representation to model transformations between coordinate frames in the system. In this representation each coordinate frame has one or zero parents, and zero to many children. With this representation, reference frames are represented as nodes in the tree, and the transformations between each reference frame are the branches of the tree. All ROS nodes that produce *tf* data broadcast this data on the same topic. Therefore all transforms between reference frames in the system are available by subscribing to the */tf* topic, allowing the tree structure to be built. An example of such a tree for the Turtlebot UGV is shown in Figure 4.2, where transformations between reference frames corresponding to various parts of the robot are shown.

Finally, ROS provides a robot description language that allows manufacturers and users wishing to customize their robot to describe the physical properties of the robot and locations of the sensors in an XML document, and have those properties reflected in the system with the *tf* library.

4.1.3 Robot Operating System Development Tools

ROS also provides several tools to facilitate debugging, plotting, and visualization. ROS provides the tool, *rxgraph* to introspect connections between nodes in the system, as well as *rviz*, a three dimensional visualization tool used to visualize many sensor data types as well as the robots themselves. A screenshot of *rxgraph* is shown in Figure 4.3 for a simulated mapping example. As well, a screen shot of *rviz* is shown in Figure 4.4, where a 3D map building process is visualized. There also exists several command line introspection tools to provide similar functionality without relying on a graphical environment. ROS also provides *rosbag*, a mechanism to save and playback

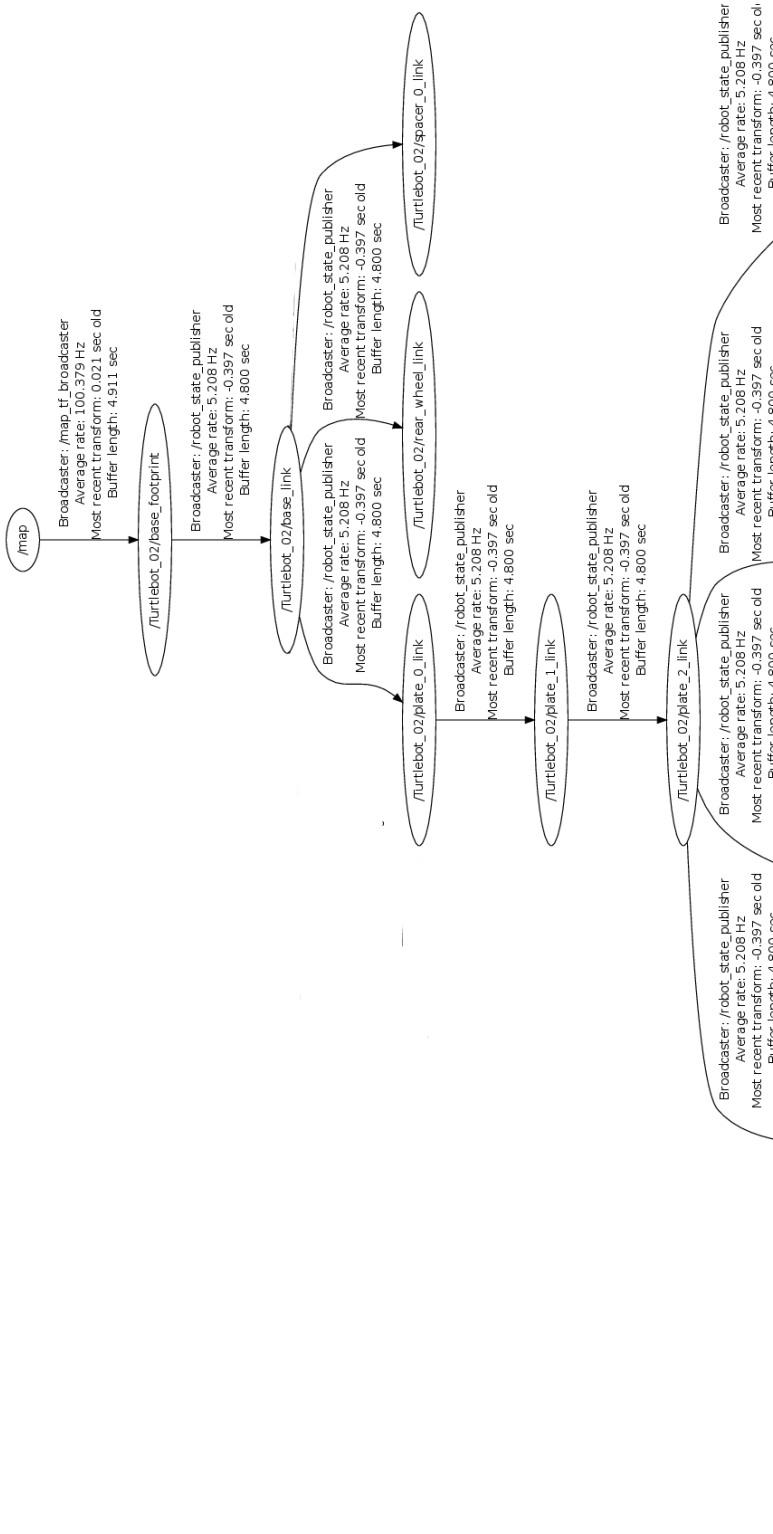


Figure 4.2: An example of the transform tree for the Turtlebot Unmanned Ground Vehicle (UGV) used in this work.

messages within the system. The *rosbag* tool is used in this research so that all sensor observations during a map building process may be saved so that the map building process may be altered offline. This allows the transform tree to be modified so that the transform of maps with a wide range of discretizations may be explored.

4.1.4 Robot Operating System Software Ecosystem

Given that ROS is community developed, there exists a vast ecosystem of existing software such that new developers may incorporate or build upon proven existing work. Some examples of existing software include implementations of navigation within an existing map, SLAM, path planning, etc. One of the most important existing implementations for this work is Octomap. This implementation is discussed further in Section 4.2

Several frequently used open source software projects are integrated into ROS. One of these projects is the Gazebo simulation environment which allows the simulation of robots described with the ROS robot description language. OpenCV as well as PCL are also fully integrated to ROS providing built-in computer vision functionality. PCL is discussed further in Section 4.3.

4.2 Octomap

Octomap is the open-source software implementation of Hornung et al.'s work in octree mapping [8]. Their implementation is written in C++ and freely available to build upon or modify. The discussion of Octomap is divided into the discussion of the self contained library, and the discussion of its implementation into the ROS environment.

4.2.1 The Octomap Library

Hornung et al.'s distribution of the Octomap library provides facilities for the creation, building, and modification of Octree occupancy grids as well as facilities to load and save these maps to disk. Their implementation consists of a series of *OcTree* types

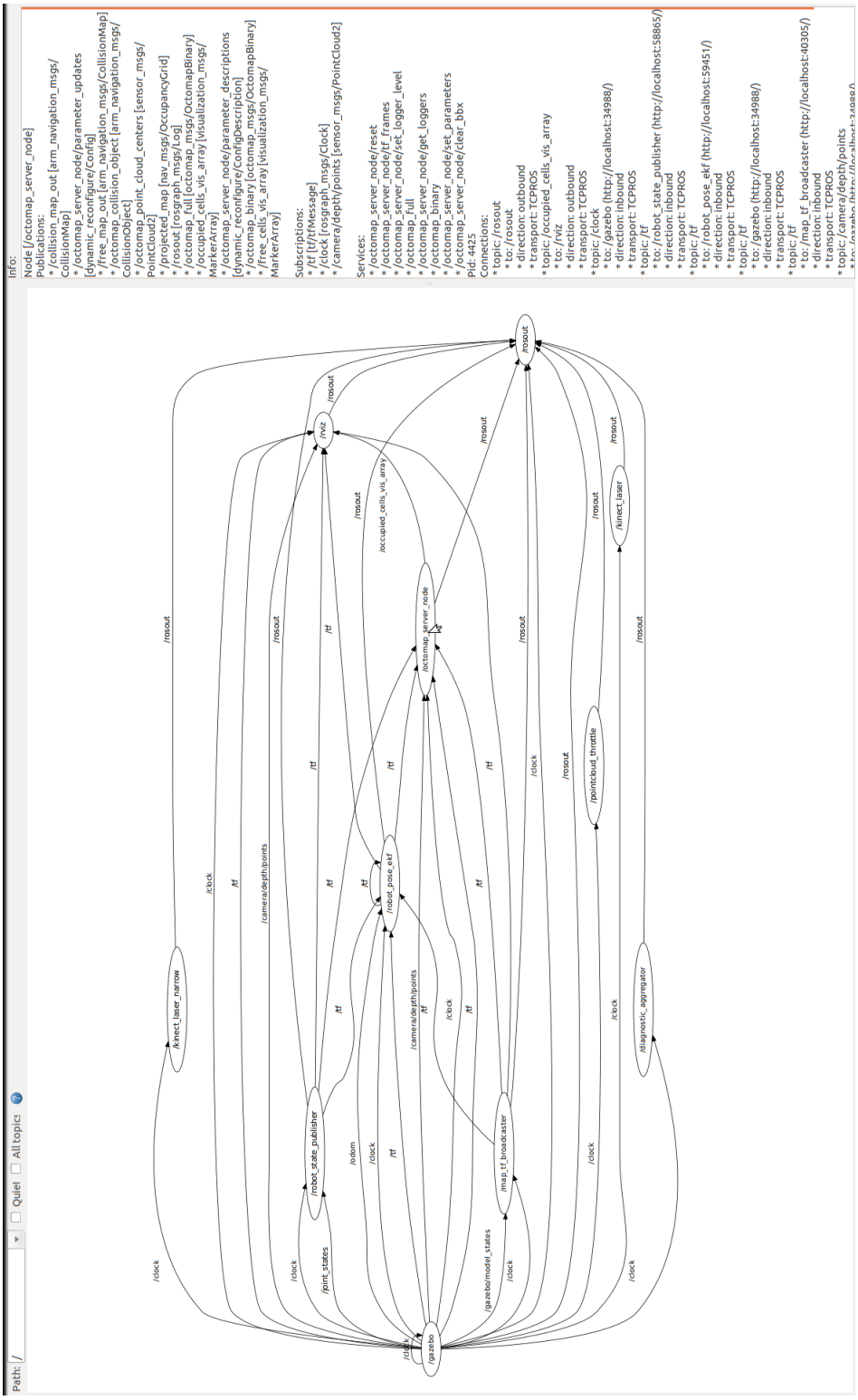


Figure 4.3: A screenshot of the *rxgraph* tool showing node and topic interconnections for simulated mapping.

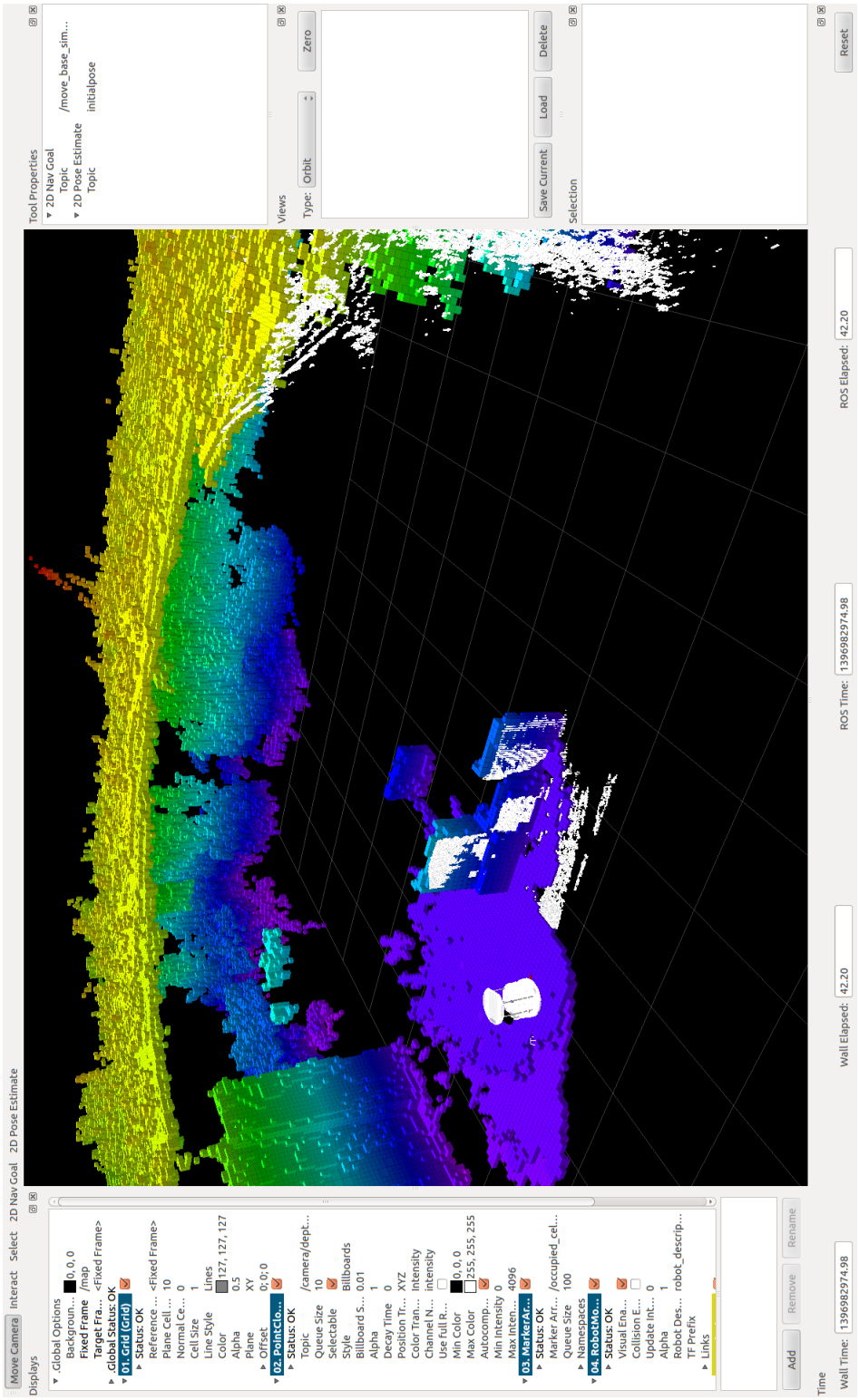


Figure 4.4: A screenshot of the *rviz* tool visualizing a map building process.

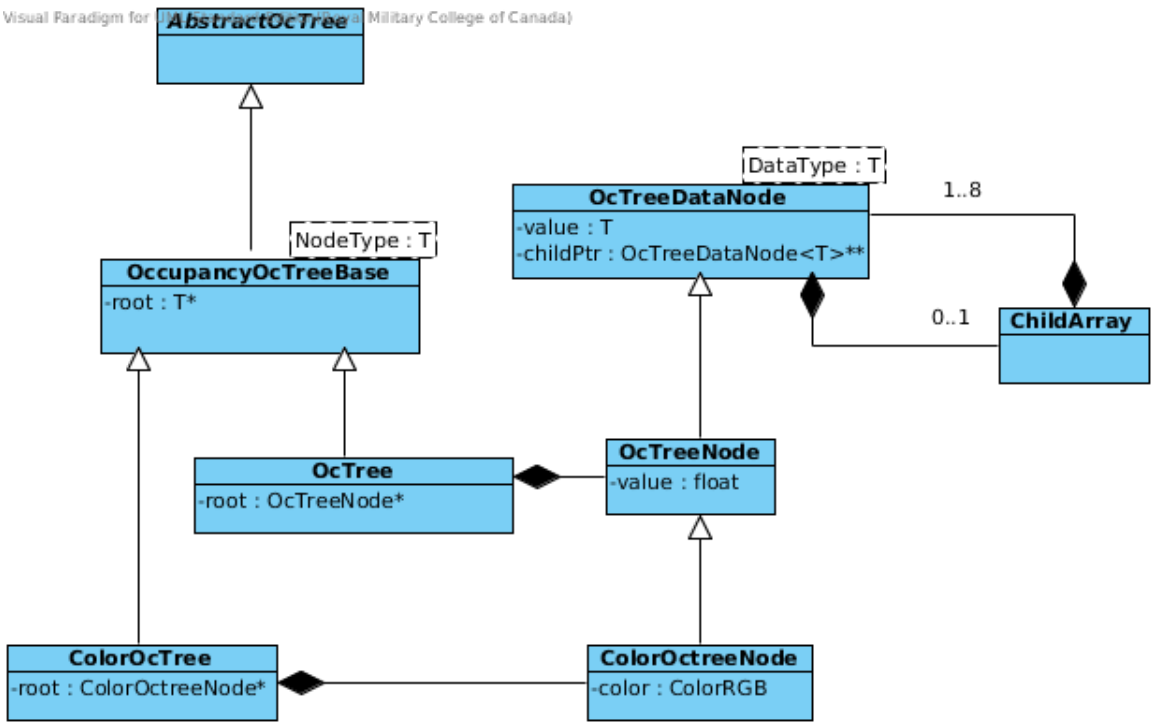


Figure 4.5: The class diagram of the Octomap library

which are a composition of a series of *Node* types. The class diagram used of the Octomap library is shown in Figure 4.5.

Inheritance is used across the Octree classes, where basic tree functionality is implemented in the *OcTreeBase* class, and mapping functionality is implemented in the *OccupancyOcTreeBase* class. The *OcTree* types are templated over the data type stored in the node. This strategy allows the same tree structure to be used while allowing the data stored in the node to be modified so that it is relevant to the mapping application.

The first node type *OcTreeDataNode*, includes a templated data value, as well as child and parent pointers. The node implementation is memory efficient in comparison to a naive implementation due the fact that nodes at the lowest level of the tree, or leaf nodes, do not allocate pointers to their children, since only a pointer to an array of children is allocated rather than the child pointers themselves. The main node type, *OcTreeNode* inherits from *OcTreeDataNode* and defines the data values' type.

The main type used for octree mapping is the *OcTree* class, which is a composition

of the *OcTreeNode* class. This strategy of templating the octree across node types allows the extensibility of the library to be used with additional data stored in each node, such as the *ColorOcTree* shown in Figure 4.5.

Included with the library is a visualization tool, *octovis*, which can load an octree map from disk and display it in 3D using OpenGL. Octovis allows users to verify the correctness of constructed maps. Additionally it is used in this work to present experimental results. Octovis is used throughout this work to present the validity of merging results. An example of an Octree occupancy grid as displayed by octovis is shown in Figure 4.6 where dark blue colours represent nodes with high occupancy probability. In Figure 4.6 a map which represents a small part of an environment containing the corner of a room and a doorway is shown.

Octomap does not however provide any functionality for transforming entire maps or merging multi-depth sources efficiently. However, given that the source code for the library is freely available it remains freely modifiable for contribution of the work of this thesis. The theoretical background for the addition of this functionality shall be discussed in Chapter 3.

4.2.2 Octomap Robot Operating System Implementation

Octomap is integrated into ROS such that 3D octree occupancy grid maps can be build from 3D point cloud data within a ROS environment. The developers of Octomap provides a map server for the ROS system that subscribes to a topic publishing point cloud messages and builds the map using localization from *tf* library. The connection of ROS topics for an example octomap mapping application is shown in Figure 4.7. This example considers a robot using external localization data from a motion capture system provided by the *mocap_node* ROS node and an additional node that incorporates the data published by *mocap_node* into the transform tree. A Microsoft Kinect sensor is used to provide 3D point cloud data to the octomap server node and is published in the */camera/depth/points* topic. Transforms between map and sensor frames are published on */tf* topics, which represent the transformation between reference frames pertaining to sensor relative to the map. The transform tree for this

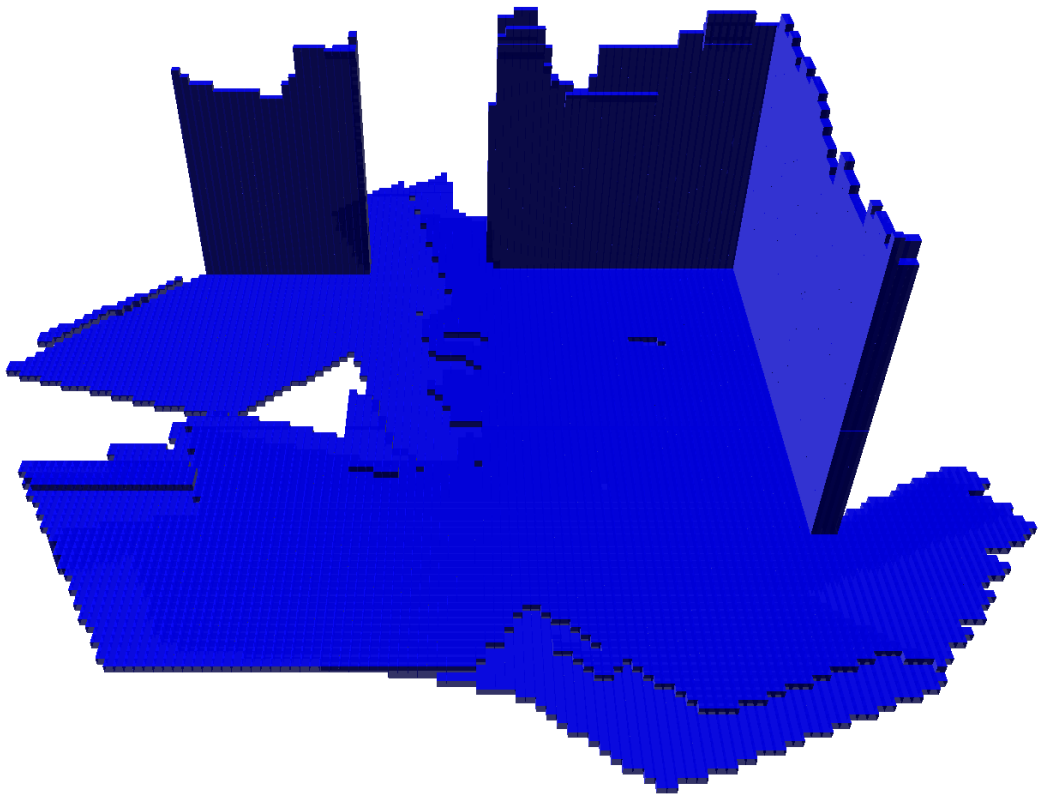


Figure 4.6: A visual representation of an Octree occupancy grid as displayed by the Octovis visualization tool.

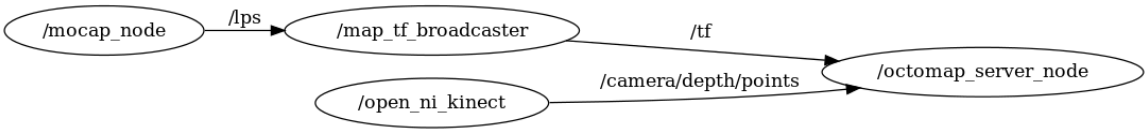


Figure 4.7: An example of the topic connections between nodes of a mapping application using Octomap.

mapping application is shown previously in Figure 4.2. The map is then accessible through a service and a custom map message. A ROS node for requesting this service and saving the map to disk is also provided.

4.3 Point Cloud Library

PCL is a software library for 3D image and point cloud processing [33]. The PCL library contains many algorithms for several aspects of point cloud processing including filtering noisy data sources, estimating features, reconstructing surfaces, segmentation, and most importantly for this work, registration. The registration component of PCL includes routines for performing alignment of 3D point clouds with the ICP algorithm. The library includes several optimizations to the original ICP algorithm to reduce computational complexity of the registration process. The registration component of the library is used throughout this work as an implementation of the ICP algorithm. PCL also provides visualization tools which are used in this thesis to provide an observation method to verify that the ICP registration routines converge to a desired result.

4.4 Conclusion

This chapter presented an overview of the existing software tools used to support this thesis. The Octomap library is used as the foundation of the work and provides the functionality that allows us to build and manipulate 3D octree occupancy grid maps. ROS provides us with a mechanism to interface the sensor observations of both simulated and physical robotics platforms to the Octomap library and build maps. Finally, the PCL library provides us with existing registration routines that may be

used to improve the transformation estimate between two merged maps. Each of these tools provide valuable functionality which is incorporated into the problem of merging 3D octree occupancy grid maps. This problem is formally defined in Chapter 5.

Chapter 5

Problem Formulation

In this chapter we outline the problem being investigated and introduce the notation that will be used throughout the work. We begin by describing the probabilistic representation of an octree map. We follow by exploring the problem of calculating a transformation between map reference frames from sensor observations of robots in a rendez-vous scenario. The use of ICP to reduce the error in initial transformation estimates is discussed. The problem of performing this transformation on an octree map is also explored. Finally, once maps are in a common reference frame, rules for incorporating data into a global map are introduced.

5.1 The Octree Map

In this work an octree map, M is defined as a set of m leaf nodes. Let n be a leaf node in the map.

$$|M| = m \tag{5.1}$$

$$n \in M \tag{5.2}$$

Each leaf node represents a volumetric subdivision of 3D space, and contains the log-odds probability of its occupancy, $L(n)$, a function of that volume's occupancy probability, $P(n)$:

$$L(n) = \log\left[\frac{P(n)}{1 - P(n)}\right] \tag{5.3}$$

5.2 Map Merging

Let a merged map M' be defined as the merger of one map M_2 into another map M_1 without loss of generality.

$$M' = M_2 \cup M_1 \tag{5.4}$$

Where,

$$n_1 \in M_1 \quad , \quad |M_1| = m_1 \tag{5.5}$$

$$n_2 \in M_2 \quad , \quad |M_2| = m_2 \tag{5.6}$$

It is important to note that M_1 and M_2 may not necessarily be of the same size as shown in Equation (5.5).

This merger requires three key steps to be performed:

1. Calculation of the transformation matrix, T, to transform M_2's data into the coordinate system of M_1;
2. The transformation of M_2's data into M_1's coordinate system.
3. Integration of data from M_2 into M_1.

These steps are now described.

5.2.1 Calculation of Transformation Matrix

The merger of two maps requires the calculation of a 4 x 4 transformation matrix, T from one map's coordinate system to the other. Homogeneous coordinates are used in the transformation matrices. The addition of another dimension to the transform matrices allows both rotation and translation to be performed in the same matrix multiplication. The transformation from M_2's coordinate system into M_1's coordinate system is denoted, T_1^2. This transformation can be used when merging data from M_2 into M_1. When merging data from a leaf node n_2 in M_2, centred at point $\boldsymbol{x_2}$ in M_2's coordinate system, the corresponding point in M_1's coordinate system, $\boldsymbol{x_1}$, is calculated as:

$$\boldsymbol{x_1} = T_1^2 \boldsymbol{x_2} \tag{5.7}$$

The transformed point can be used to look up the corresponding leaf node n_1 in the map, M_1 which encloses the point $\boldsymbol{x_1}$.

In order to calculate this matrix T_1^2, this work considers a merger when two robots are in a rendez-vous scenario where both can mutually observe each other. In this case the transformation matrix is calculated in two steps:

1. Calculation of initial transformation estimate from mutual pose observations.

2. Calculation of final transformation estimate from map data.

5.2.1.1 Calculation of Initial Transformation Estimate

This step of the transformation calculation obtains an initial transformation matrix from the two robots' mutual observations of one another. This step assumes that the robots are capable of mutually observing one another. This work follows the same framework for obtaining a transformation matrix as described by Dinnissen in [2]. Since the robots make observations of one another in their local frames of reference, the initial transformation matrix $T_{1\,i}^2$ is obtained from a product of three different component transformations. The first component, T_c , is the transformation matrix to convert points in the current robot's local frame of reference to its global frame of reference from its pose at the time of the merger. The second component, T_r , is the transformation matrix to convert points in the other robot's local frame of reference to the current robot's local frame of reference. This matrix is built using the other robot's sensor observations. The final component, T_o , is the transformation matrix to convert points in the other's robot's frame of reference to its global frame of reference from its pose at the time of the merger. The following equations show the calculation of $T_{1\,i}^2$ from each component. These equations are shown using two dimensional transforms for ease of demonstration, however it can be extended to three

dimensions without loss of generality.

$$
\begin{aligned}
&T^2_{1\,i} = T_c T_r T_0^{-1} \\
&= \begin{bmatrix} \cos\theta_c & -\sin\theta_c & t_{x_c} \\ \sin\theta_c & \cos\theta_c & t_{y_c} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_r & -\sin\theta_r & t_{x_r} \\ \sin\theta_r & \cos\theta_r & t_{y_r} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_o & -\sin\theta_o & t_{x_o} \\ \sin\theta_o & \cos\theta_o & t_{y_o} \\ 0 & 0 & 1 \end{bmatrix}^{-1}
\end{aligned} \tag{5.8}
$$

Where:

$$\theta_r = \pi - \theta_o + \theta_c \tag{5.9}$$

$$t_{x_r} = r\cos\theta_c \tag{5.10}$$

$$t_{y_r} = r\sin\theta_c \tag{5.11}$$

Figures 5.1 and 5.2 shows how all the component transformation matrices combine to form the desired transformation matrix as well as the meaning of the remaining variables in Equation (5.8).

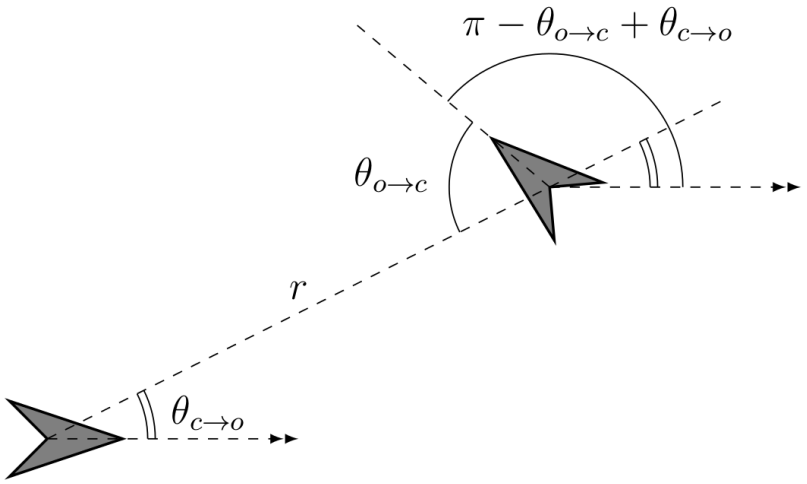


Figure 5.1: Diagram illustrating how relative equations are determined (Used from [2] with permission).

5.2.1.2 Improvement of Transformation Matrix

It is important to recognize that the initial transformation estimate is obtained from uncertain sensor observations. In addition, each robot's knowledge of their own position is uncertain given that it is subject to error in odometry and error in its previous

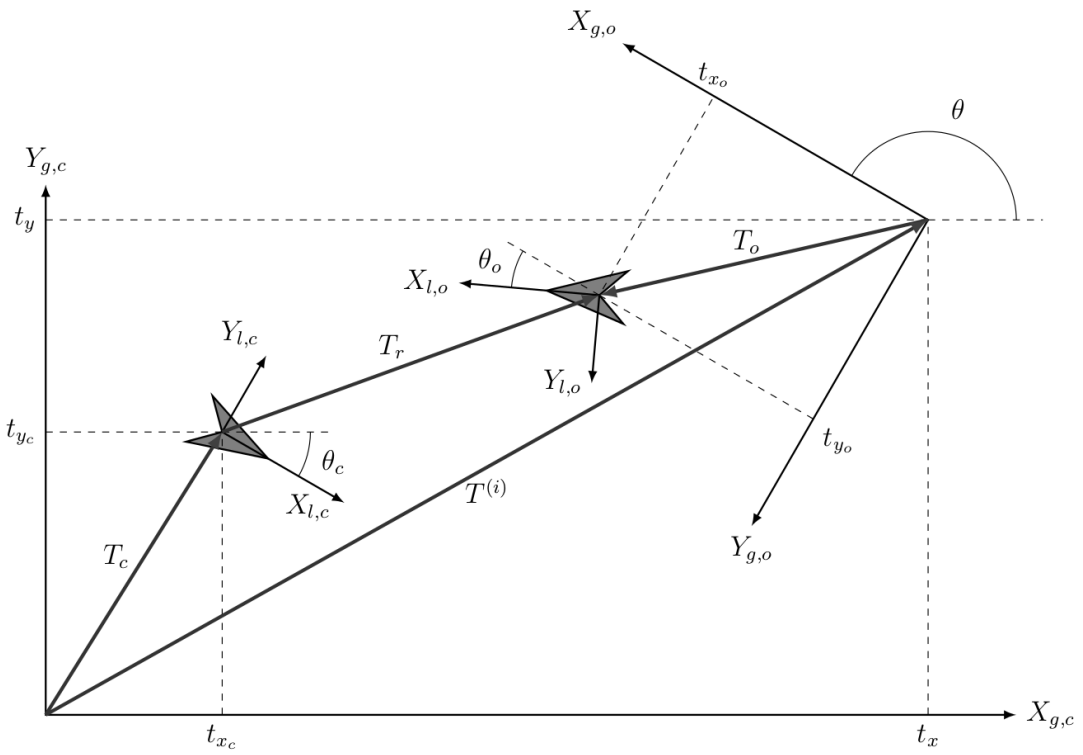


Figure 5.2: Diagram illustrating how transformation matrices and parameters are determined (Used from [2] with permission).

sensor observations. As such, while the initial transformation estimate does provide some useful information, it is often undesirable to rely on that information alone to perform map merging.

For the case of octree occupancy grid maps, the ICP algorithm is chosen as the method for improving the accuracy of the transform between maps. Although [29] describes ICP as an algorithm that may be used on arbitraty features, a point cloud representation will be generated from the maps M_1 and M_2 to be used with ICP as described in [30].

Point cloud generation is performed by adding the centre point of each occupied leaf node in the map to the point cloud. Once these point cloud sets are created the initial transform estimate as discussed in section 5.2.1.1 is applied to the second point cloud such that both point clouds are roughly in the same frame of reference. A Subset of each point cloud which represents commonly mapped territory is extracted. This is done by determining bounding boxes for each point cloud. Those points contained in the intersection of the two bounding boxes are added to the subset which represents commonly mapped territory.

The ICP algorithm is then performed on the point clouds which represent commonly mapped territory to obtain the registration vector, and subsequently a refinement to the transformation between the two frames of reference for each robot, T_{ICP}, which is then used to refine the overall transformation as:

$$T_1^2 = T_{ICP} T_{1\,i}^2 \tag{5.12}$$

Now that a sufficiently accurate transformation is obtained it is necessary to apply this transformation to the map.

5.2.2 Map Transformation

Once a sufficient estimate for the transformation between maps is known it is necessary to perform this transformation on M_2's data obtaining the transformed map, $M_2^{T_1^2}$.

This transformation is performed on the map using trilinear interpolation as described in Section 3.2. The bounding region is of the transformed map is initially determined using T_1^2 and the bounding region of M_2. The transformed map is then built by iterating through each leaf node, n , at the lowest level of the octree hierarchy in the bounding region of the transformed map and assigning an occupancy. This occupancy is determined by looking up the source point $\boldsymbol{x_s}$ from the centre point of the current voxel, $\boldsymbol{x_n}$ and the inverse transformation as:

$$\boldsymbol{x_s} = T_1^{2^{-1}} \boldsymbol{x_n} \tag{5.13}$$

Once $\boldsymbol{x_s}$ is known, the occupancy of the voxels whose centre points enclose $\boldsymbol{x_s}$ in a cubic lattice are used to calculate the final occupancy assigned to node n.

Once all nodes in the bounding region have been accounted for, a map will have been created entirely with leaf nodes. Map compression is then performed using the existing algorithms from the Octomap library as described in Section 2.4.

5.2.3 Integration of Map Data

The creation of this merged map, M', is then performed by updating M_1 with the data from $M_2^{T_1^2}$. Given the nature of the octree data hierarchy, commonly mapped portions of the environment may be mapped at different levels of the octree hierarchy. While this problem could be overcome by simply expanding all nodes in each map to the lowest level, this is inefficient since it requires the processing of additional nodes. As such, this work only performs local map expansion in cases of map-level conflict. This requires $M_2^{T_1^2}$'s data to be incorporated into M_1 differently corresponding to four different cases which are described below. Readers should note that clamping as described in Section 2.4 should be implemented for occupancy updates but is omitted here for simplicity.

5.2.3.1 Case 1 - New Data Is from a Volume that Is Not Already Mapped in M_1

If the new data to be integrated from $M_2^{T_1^2}$ does not correspond to an existing leaf node in M_1 after performing the transformation described in equation (5.7), a new leaf is then added to the octree. The leaf's occupancy probability in log-odds notation is then set to be equal to the log-odds occupancy of the node in $M_2^{T_1^2}$:

$$L(n_1|z_{1:t}) = L(n_2|z_{1:t-1}) \tag{5.14}$$

Where, $L(n_2|z_{1:t-1})$ represents the log-odds occupancy of the leaf node from $M_2^{T_1^2}$ and $L(n_1|z_{1:t})$ is the merged occupancy of the new leaf node at time t following the merger.

5.2.3.2 Case 2 - New Data is Already Mapped at the Same Level in M_1

If the new data to be integrated from $M_2^{T_1^2}$ corresponds to an existing leaf node in M_1's octree which is at the same level in the tree, the following update calculation applies:

$$L(n_1|z_{1:t}) = L(n_1|z_{1:t-1}) + L(n_2|z_{1:t-1}) \tag{5.15}$$

$L(n_1|z_{1:t-1})$ then represents the existing log-odds occupancy at leaf node n_1.

5.2.3.3 Case 3 - New Data corresponds to a Volume Already Mapped at a Coarser Resolution in M_1

If the new data to be integrated from $M_2^{T_1^2}$ corresponds to an area covered by a leaf node n_1, where n_1 is at a higher level in the octree, or a coarser resolution, the merger is performed by first adding a lower level to M_1's tree below n_1. This addition creates eight new leaf nodes, n_1^1 to n_1^8. This process is shown in Figure 5.3.

Where each sub-element of n_1 contains the log-odds occupancy of:

$$L(n_1^i) = L(n_1) \tag{5.16}$$

Following this step, n_2's data is then integrated to the sub-element of n_1 representing the same space by the formula:

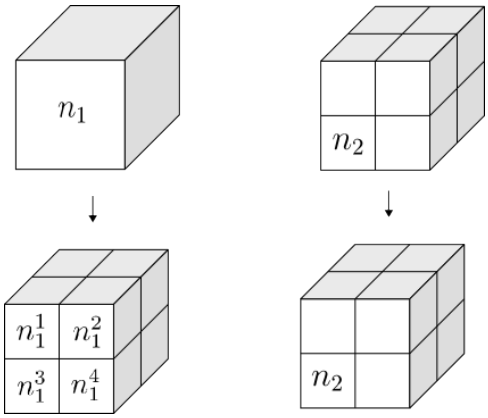


Figure 5.3: The addition of another level below n_1 prior to merging.

$$L(n_1^i|z_{1:t}) = L(n_1^i|z_{1:t-1}) + L(n_2|z_{1:t-1}) \tag{5.17}$$

5.2.3.4 Case 4 -New Data corresponds to a Volume Already Mapped at a Finer Resolution in the Existing Octree

This case describes the situation where data from a leaf node, $n_2 \in M_2^{T_1^2}$ maps a volume already mapped in M_1 and data from $M_2^{T_1^2}$ is at a coarser resolution than in the existing map, M_1. In this case, the integration of data from n_2 is performed very similar to 5.2.3.3. The integration of data is performed first by adding an additional level to $M_2^{T_1^2}$'s tree below n_2. This addition creates eight new leaf nodes, n_2^1 to n_2^8. This process is shown in Figure 5.4.

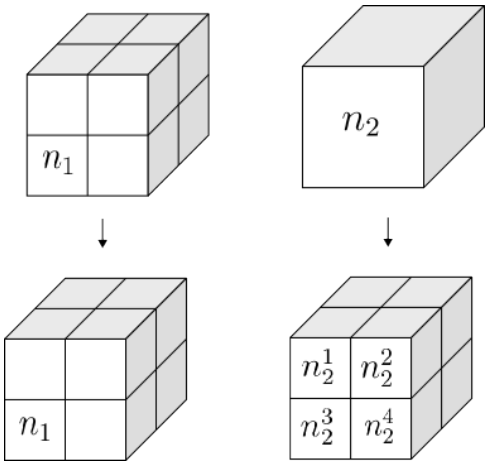


Figure 5.4: The addition of another level below n_2 prior to merging.

Where each sub-element of n_2 contains the log-odds occupancy of:

$$L(n_2^i) = L(n_2) \tag{5.18}$$

Following this step, each sub-element of n_2's data is then integrated in to the elements of M_1 representing the same space by the formula:

$$L(n_1|z_{1:t}) = L(n_1|z_{1:t-1}) + L(n_2^i|z_{1:t-1}) \tag{5.19}$$

Where n_1 represents any leaf node in M_1 which maps the same area as the sub-element of n_2 that we are merging.

5.3 Conclusion

This chapter has presented the developed theoretical components required for map-merging with unknown transformations between map reference frames. The implementation of this theory into software is described in Chapter 6.

Chapter 6

Implementation

This chapter discusses the implementation of the theory discussed in Chapter 5 into software algorithms. In this chapter the overall global map merging algorithm is divided into several sub-algorithms:

1. The creation of 3D point clouds from Octomap Octree maps such that the the ICP algorithm can be performed on map data to refine the initial transformation matrix within the context of PCL.

2. The extraction of intersecting volumes from each point cloud set.

3. The refinement of the initial transformation estimate between map reference frames using the ICP algorithm.

4. The application of the transformation to the second map to be merged such that each map is now in a common reference frame.

5. The integration of occupancy data from the transformed map into the first map to create a global map from the data of each robot.

The sequential execution of these algorithms results in a merged octree occupancy grid that is robust to error in the transformation estimate between the maps. A detailed description for each sub-algorithm is given in addition to pseudo code notation.

6.1 Creation of Point Clouds from Octomap Octree Maps

In order to use the available ICP algorithms from PCL it is necessary to translate map data into point clouds. This is done for both maps to be merged. This translation is performed by iterating through each leaf node, n_{leaf}, in a map, M, and adding the node's centre point to the Point Cloud if the node is occupied. This node uses the same definition of occupied as [8] where the node's log-odds occupancy must have

reached the clamping threshold, l_{max}, to be considered occupied. Pseudo code for this algorithm is shown in Algorithm 1. The C++ Implementation of this algorithm

Algorithm 1: Creation of point cloud sets from Octomap octree maps

```
Input: An Octomap M
Output: A Point Cloud P_Cl
begin
    P ⟵ ∅;
    foreach n_leaf ∈ M do
        if L(n) ≥ l_max then
            P_Cl = P_Cl + CentrePoint(n);
        end
    end
end
```

with PCL and Octomap is shown in Appendix A.1.

6.2 Extraction of Intersecting Volumes from Point Cloud Sets

In order to use the ICP algorithm successfully, one must have two feature sets which represent the same rigid body as either the data set, W or the model set, X. To extract these sets, this work first applies the initial transform estimate to the second map's data point cloud set, W. Those points representing intersecting volume of the two sets are then extracted by initially extracting the points in W that are contained within the bounding cubic lattice of the entire set X. This selection of points from W is denoted W_{sel}. Secondly, those points from X contained in W_{sel}'s bounding cubic lattice are extracted to obtain, X_{sel}. Pseudo code for this algorithm is given in Algorithm 2. It is important to note that the location of the bounding boxes for each point cloud set are determined using utility functions, $GetMetricMin()$ and $GetMetricMax()$. These functions return a vector where each component represents the lowest and highest displacement from the origin along each axis in the point cloud set. The C++ Implementation of this algorithm with PCL is shown in Appendix A.1.

Algorithm 2: Extraction of Point Clouds from Intersecting Volumes

Inputs : Point Clouds, W and X, Transformation Matrix $T^2_{1\,i}$
Outputs: Point Clouds, W_{sel} and X_{sel}
begin
  $W_{sel} \longleftarrow \emptyset$;
  $X_{sel} \longleftarrow \emptyset$;
  $\boldsymbol{x}_{max} = \mathbf{0}$;
  $\boldsymbol{x}_{min} = \mathbf{0}$;
  $TransformPointCloud(P, T^2_{1\,i})$;
  $\boldsymbol{x}_{min} = GetMetricMin(X)$;
  $\boldsymbol{x}_{max} = GetMetricMax(X)$;
  foreach $\boldsymbol{w} \in W$ **do**
    if $x_{minx} < w_x < x_{maxx}$ *and* $x_{miny} < w_y < x_{maxy}$ *and* $x_{minz} < w_z < x_{maxz}$ **then**
      $W_{sel} = W_{sel} + \boldsymbol{w}$;
    end
  end
  $\boldsymbol{w}_{min} = GetMetricMin(W)$;
  $\boldsymbol{w}_{max} = GetMetricMax(W)$;
  foreach $\boldsymbol{x} \in X$ **do**
    if $w_{minx} < x_x < w_{maxx}$ *and* $w_{miny} < x_y < w_{maxy}$ *and* $w_{minz} < x_z < w_{maxz}$ **then**
      $X_{sel} = X_{sel} + \boldsymbol{x}$;
    end
  end
end

6.3 Refinement of Initial Transform with the Iterative Closest Point Algorithm

In merging situations prone to error in the initial transformation estimate between map reference frames, it is necessary to perform map alignment using data from commonly mapped territory. In the previous section, one method for extracting a commonly mapped portion of the environment is introduced. Refinement of the initial transformation is performed by iteratively performing the ICP algorithm on the data and model sets W and X respectively. Each incremental transformation is then incorporated into the overall refined transformation by post multiplication after each ICP iteration.

It is important to note that the ICP algorithm described in Section 3.3.1 would be computationally expensive to compute due to the requirement to determine the set of points from the model that correspond to the target. In the PCL library, several parameters exist to reduce the cost of determining this correspondence set. Most importantly for this use of ICP are:

- Maximum correspondence distance;
- Maximum transformation ϵ.

The maximum correspondence distance reduces the search space for correspondences to be found in the model set by only searching for correspondences in a sphere centred at a point in the data set with a radius equal to the maximum correspondence distance. The maximum transformation ϵ reduces the search space for transforms that minimize the mean square error. The maximum transformation ϵ is a measure of the maximum allowable transformation between ICP iterations, defined as the sum of all elements in the difference between the transformation matrices from two consecutive ICP iterations. Use of these parameters allows the ICP algorithm to begin with a large correspondence distance to make large scale transformations initially. Then, once the ICP algorithm has converged with its initial maximum correspondence distance, the maximum correspondence distance can be decreased to make further small scale refinements to the transformation.

Another important observation is that the ICP algorithm performs better when there is a larger number of points in the model set , X, than the data set, W. For continuous sensor registration applications this is usually the case. However, for map merging algorithms it may not always be the case that M_1 has more nodes in the common portion of the environment than M_2. Since both the generated point clouds from M_1 and M_2 represent the same portion of the environment, the point cloud of largest size is used as the model set, with the point cloud of smallest size used as the data set. If point cloud sets are interchanged, then the inverse of the refinement transformation matrix is returned.

The details of ICP refinement algorithm are also described further in Algorithm 3 with the C++ implementation using PCL shown in Appendix A.1.

6.4 Execution of Refined Transform

After obtaining a final transformation with ICP refinement, the overal transformation matrix to obtain $M_2^{T_1^2}$, T_1^2, is obtained by multiplying the original transformation matrix estimate by the ICP refinement. This transformation is then performed on the map as described in Section 3.2. The process for performing this transformation is described further in Algorithms 4 to 7. The high level transformation with pseudo code shown in Algorithm 7, begins by first determining a cubic bounding region of the transformed map. This is done by first determining the metric minimum and maximum points of the map using functions from the Octomap Library. These points are then used to determine the eight vertices of the bounding cubic region for M. Subsequently each of these points are transformed with T_1^2. The minimum and maximum x, y, and z component are then extracted to obtain $\boldsymbol{p}_{min}$ and $\boldsymbol{p}_{max}$. Pseudo code for this process of extracting a bounding region of the transformed map is shown in Algorithm 4. It is important to note that the bounding region for the transformed map is determined using the, $GetMetricMin()$ and $GetMetricMax()$ utility functions from the Octomap library. These functions return a vector where each component represents the lowest and highest displacement from the origin along each axis in the map specified as a parameter. The extracted points are subsequently used to iterate over

Algorithm 3: Iterative Closest Point Map Transformation Refinement

Inputs : Point Clouds, P_{sel} and X_{sel}, Resolution of the Octree Map, $mapRes$
Output: Refined Transformation Matrix T_{ICP}
begin
  $SetICPMaxEpsilon(mapRes/60)$;
  $SetICPMaxCorrespondenceDistance(10mapRes)$;
  $T_{ICP} = I$;
  $T_i = I$;
  $T_{i-1} = I$;
  $P_{result} \longleftarrow \emptyset$;
  if $|P_{sel}| < |X_{sel}|$ **then**
    $P_{result} = P_{sel}$;
    $SetICPModel(X_{sel})$;
    for $i = 0$ **to**$maxICPIterations$ **do**
      $SetICPData(P_{result})$;
      $T_{i-1} = T_i$;
      $DoICPAlignment(P_{result}, T_i)$;
      $T_{ICP} = T_{ICP}T_i$;
      if $\sum_{j=1}^{4}\sum_{k=1}^{4} T_{ijk} - T_{i-1jk} < GetICPMaxEpsilon()$ **then**
        $DecreaseICPMaxCorrespondenceDistance()$;
      end
    end
  else
    $P_{result} = X_{sel}$;
    $SetICPModel(P_{sel})$;
    for $i = 0$ **to**$maxICPIterations$ **do**
      $SetICPData(P_{result})$;
      $T_{i-1} = T_i$;
      $DoICPAlignment(P_{result}, T_i)$;
      $T_{ICP} = T_{ICP}T_i$;
      if $\sum_{j=1}^{4}\sum_{k=1}^{4} T_{ijk} - T_{i-1jk} < GetICPMaxEpsilon()$ **then**
        $DecreaseICPMaxCorrespondenceDistance()$;
      end
    end
    $T_{ICP} = {T_{ICP}}^{-1}$;
  end
end

the voxels in the bounding box of the transformed map and look up the occupancy of the nodes of the original map. Using utility functions from the Octomap library, the leaf node nearest to the point generated by the inverse transform with the location of the transformed map's voxel centre point is determined. The offset of the generated source point and the centre point of the nearest neighbour node is then used to determine the occupancy of the interpolated point with trilinear interpolation as described in Algorithms 5 and 6. A leaf node is then added to the transformed octree with the interpolated occupancy. Once all voxels in the transformed map's bounding box are addressed, the transformed map is then compressed with the Octomap library. The overall algorithm for transforming octree maps is shown in Algorithm 7 with its C++ implementation shown in Appendix A.2.

6.5 Integration of Transformed Map into Global Map

Now that both maps are in a common reference frame, data from $M_2^{T_1^2}$ can be integrated into M_1 to create a global map M. This global map is created by iterating over each node in $M_2^{T_1^2}$ and updating M_1 with its data according to the rules described in Section 5.2.3. At each step of the iteration the location of n_2 is searched in M_1 to determine if a corresponding node n_1 exists. If this is not the case, Case 1 from Section 5.2.3.1 applies. If n_1 does exist, it is required to determine the depth difference of n_2 within $M_2^{T_1^2}$'s tree and n_1 within M_1's tree. If there is no depth difference, Case 2 from Section 5.2.3.2 applies. If the depth difference is positive, Case 3 from Section 5.2.3.3 applies. This scenario requires children to be added to n_1. The child corresponding to the same volume as n_2 is then updated. Finally, if the depth difference is negative, Case 4 from Section 5.2.3.4 applies. This scenario requires children to be added to n_2. Each added child is then used to update the corresponding nodes in M_1. Given that this algorithm is written with the intent to be implemented with the Octomap library and the C++ programming language, a *for* loop may be used to integrate the added children to n_2 on subsequent loop iterations. This is due to the specific implementation of the Octomap node iterator class. With this implementation, children are simply added to n_2 and Case 2 updates are executed in later loop iterations for

Algorithm 4: Extraction of Transformed Map's Bounding Box

Inputs : Octree Map M, Transformation Matrix T_1^2
Output: Points $\boldsymbol{p}_{min}$ and $\boldsymbol{p}_{max}$
begin
$P_{BoundingBox} \longleftarrow \emptyset$;
$\boldsymbol{p}_{min} = GetMetricMin(M)$;
$\boldsymbol{p}_{max} = GetMetricMax(M)$;
$P = P + \begin{bmatrix} p_{min_x} & p_{min_y} & p_{min_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{min_x} & p_{min_y} & p_{max_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{min_x} & p_{max_y} & p_{min_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{min_x} & p_{max_y} & p_{max_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{max_x} & p_{min_y} & p_{min_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{max_x} & p_{min_y} & p_{max_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{max_x} & p_{max_y} & p_{min_z} \end{bmatrix}^T$;
$P = P + \begin{bmatrix} p_{max_x} & p_{max_y} & p_{max_z} \end{bmatrix}^T$;
foreach $\boldsymbol{p} \in P$ **do**
  $\boldsymbol{p} = T_1^2 \boldsymbol{p}$
end
foreach $\boldsymbol{p} \in P$ **do**
  if $p_x < p_{min_x}$ **then**
    $p_{min_x} = p_x$;
  end
  if $p_y < p_{min_y}$ **then**
    $p_{min_y} = p_y$;
  end
  if $p_z < p_{min_z}$ **then**
    $p_{min_z} = p_z$;
  end
  if $p_x < p_{max_x}$ **then**
    $p_{max_x} = p_x$;
  end
  if $p_y < p_{max_y}$ **then**
    $p_{max_y} = p_y$;
  end
  if $p_z < p_{max_z}$ **then**
    $p_{max_z} = p_z$;
  end
end
end

Algorithm 5: Trilinear Interpolation

Inputs : Axis-Offsets x_d, y_d, and z_d, Interpolation points c_{000} to c_{111}
Output: Interpolated Occupancy, c
begin
$c_{00} = (1 - |x_d|)c_{000} + |x_d|c_{100}$;
$c_{01} = (1 - |x_d|)c_{001} + |x_d|c_{101}$;
$c_{10} = (1 - |x_d|)c_{010} + |x_d|c_{110}$;
$c_{11} = (1 - |x_d|)c_{011} + |x_d|c_{111}$;
$c_0 = (1 - |y_d|)c_{00} + |y_d|c_{10}$;
$c_1 = (1 - |y_d|)c_{01} + |y_d|c_{11}$;
$c = (1 - |z_d|)c_0 + |z_d|c_1$
end

the added children. Pseudo code for this algorithm is shown in Algorithm 8 and its C++ implementation is shown in Appendix A.3.

6.6 Conclusion

This chapter has presented the implementation of the theory discussed in Chapter 5 into algorithms that may be implemented with PCL and Octomap. Each algorithm was described in detail with pseudo code notation given for each and C++ source code referenced in Appendix A. Now that a software implementation for merging octree occupancy grids is given, experimental results of successful map mergers for both simulated and true environments will be presented in Chapter 7.

Algorithm 6: Trilinear Interpolation for Octree Occupancy Grids

Inputs : Octree Map M, Nearest-Neighbour Node $n_{c_{000}}$, Axis-Offsets x_d, y_d, and z_d

Output: Interpolated Occupancy, c

begin

$res = GetResolution(M)$;

$c_{000} = P(n_{c_{000}})$;

$c_{001} = 0$;

$n_{c_{001}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}0 & 0 & res \times Sign(z_d)\end{bmatrix}^T)$

if $n_{c_{001}} \in M$ **then**

$c_{001} = P(n_{c_{001}})$;

end

$c_{010} = 0$;

$n_{c_{010}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}0 & res \times Sign(y_d) & 0\end{bmatrix}^T)$

if $n_{c_{010}} \in M$ **then**

$c_{010} = P(n_{c_{010}})$;

end

$c_{011} = 0$;

$n_{c_{011}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}0 & res \times Sign(y_d) & res \times Sign(z_d)\end{bmatrix}^T)$

if $n_{c_{011}} \in M$ **then**

$c_{011} = P(n_{c_{011}})$;

end

$c_{100} = 0$;

$n_{c_{100}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}res \times Sign(x_d) & 0 & 0\end{bmatrix}^T)$

if $n_{c_{100}} \in M$ **then**

$c_{100} = P(n_{c_{100}})$;

end

$c_{101} = 0$;

$n_{c_{101}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}res \times Sign(x_d) & 0 & res \times Sign(z_d)\end{bmatrix}^T)$

if $n_{c_{101}} \in M$ **then**

$c_{101} = P(n_{c_{101}})$;

end

$c_{110} = 0$;

$n_{c_{110}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} + \begin{bmatrix}res \times Sign(x_d) & res \times Sign(y_d) & 0\end{bmatrix}^T)$

if $n_{c_{110}} \in M$ **then**

$c_{110} = P(n_{c_{110}})$;

end

$c_{111} = 0$;

$n_{c_{111}} = LookupNode(M, \boldsymbol{p}_{n_{c_{000}}} +$

$\begin{bmatrix}res \times Sign(x_d) & res \times Sign(y_d) & res \times Sign(z_d)\end{bmatrix}^T)$

if $n_{c_{111}} \in M$ **then**

$c_{111} = P(n_{c_{111}})$;

end

$c = TrilinearInterpolation(x_d, y_d, z_d, c_{000}, c_{001}, c_{010}, c_{011}, c_{100}, c_{101}, c_{110}, c_{111})$

end

Algorithm 7: Octree Map Transformation with Trilinear Interpolation

Inputs : Octree Map M, Transformation Matrix T_1^2, Points $\boldsymbol{p}_{min}$ and $\boldsymbol{p}_{max}$
Output: Transformed Map M'
begin
  $\boldsymbol{p}_{min} = \mathbf{0}$;
  $\boldsymbol{p}_{max} = \mathbf{0}$;
  $GetMapBoundingBox(M, T_1^2, \boldsymbol{p}_{min}, \boldsymbol{p}_{max})$;
  $x = p_{min_x}$;
  $y = p_{min_y}$;
  $z = p_{min_z}$;
  $res = GetResolution(M)$;
  while $z < p_{max_z}$ **do**
    while $y < p_{max_y}$ **do**
      while $x < p_{max_x}$ **do**
        $\boldsymbol{p}_{dest} = \begin{bmatrix} x & y & z \end{bmatrix}^T$;
        $\boldsymbol{p}_{source} = T_1^{2^{-1}} \boldsymbol{p}_{dest}$;
        $n_{c_{000}} = LookupNode(M, \boldsymbol{p}_{source})$;
        if $n_{c_{000}} \in M$ **then**
          $\boldsymbol{p}_{n_{c_{000}}} = CentrePoint(n_{c_{000}})$;
          $x_d = (p_{source_x} - p_{n_{c_{000}\,x}})/res$;
          $y_d = (p_{source_y} - p_{n_{c_{000}\,y}})/res$;
          $z_d = (p_{source_z} - p_{n_{c_{000}\,z}})/res$;
          $TrilinearOctreeInpterpolation(M, n_{c_{000}}, x_d, y_d, z_d)$;
          $AddLeafToTree(M', \boldsymbol{p}_{dest}, c)$;
        end
        $x = x + res$;
      end
      $y = y + res$;
    end
    $z = z + res$;
  end
  $CompressMap(M')$;
end

Algorithm 8: Octree Map Merging of Transformed Maps

Inputs : Octree Maps M_1 and M_2'
Output: Merged Map M
begin
  foreach $n_2 \in M_2'$ **do**
    $n_1 = SearchMap(M_1, CentrePoint(n_2))$;
    if $n_1 \in M_1$ **then**
      $depthDifference = NodeDepth(n_2) - NodeDepth(n_1)$;
      if $depthDifference = 0$ **then**
        $UpdateNode(n_1, LogOdds(n_2))$;
      else if $depthDifference > 0$ **then**
        $AddChildrenToNode(n_1, depthDifference)$;
        $n_1' = SearchMap(M_1, CentrePoint(n_2))$;
        $UpdateNode(n_1', LogOdds(n_2))$;
      else if $depthDifference < 0$ **then**
        $AddChildrenToNode(n_2, depthDifference)$;
      end
    else
      $AddNodeToTree(M_1, CentrePoint(n_2)), LogOdds(n_2))$;
    end
  end
end

Chapter 7

Results

In this chapter, the results of successful map mergers are presented from both simulated and real-world environments. In each of these cases, the ROS system used for building each map is described as well as the use of the implemented software to merge the maps together. The results of each merger are then presented and discussed.

7.1 Simulated Map Merging

In this section the merger of two maps built within a simulated environment is presented. This experiment uses the Gazebo simulation environment that is integrated into ROS. This work uses the Fuerte release of ROS as well as the Turtlebot stack included in the ROS Fuerte repositories.

7.1.1 Gazebo Simulation Environment

For this experiment, ROS' integrated simulation environment, Gazebo, is used to allow a simulated Turtlebot UGV with an artificial Microsoft Kinect Sensor to traverse a synthetic environment and build an octree occupancy grid map of a portion of that environment. Another simulated robot then maps a different part of that environment while mapping a small portion of the same area covered by the first robot. The simulated environment and robot are shown in Figures 7.1 and 7.2 respectively.

7.1.2 Map Building

To build each map, an instance of the Gazebo simulator is launched. The environment is then loaded with the *wg_collada_world* model included with the simulator package. A simulated Turtlebot UGV is then spawned using the *turtlebot_gazebo* package. This package includes facilities to visualize the simulated robot from the

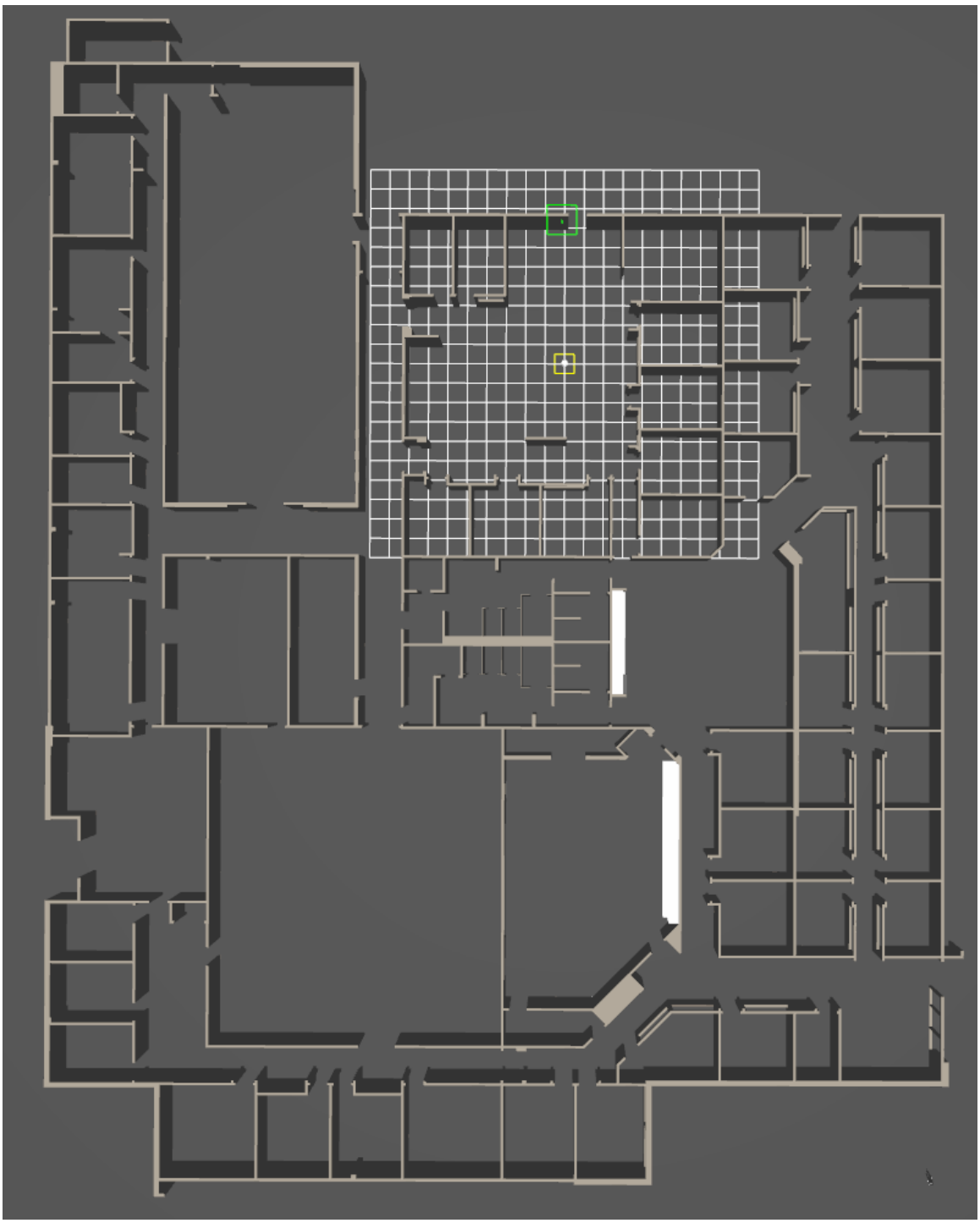


Figure 7.1: The simulated environment to be mapped.

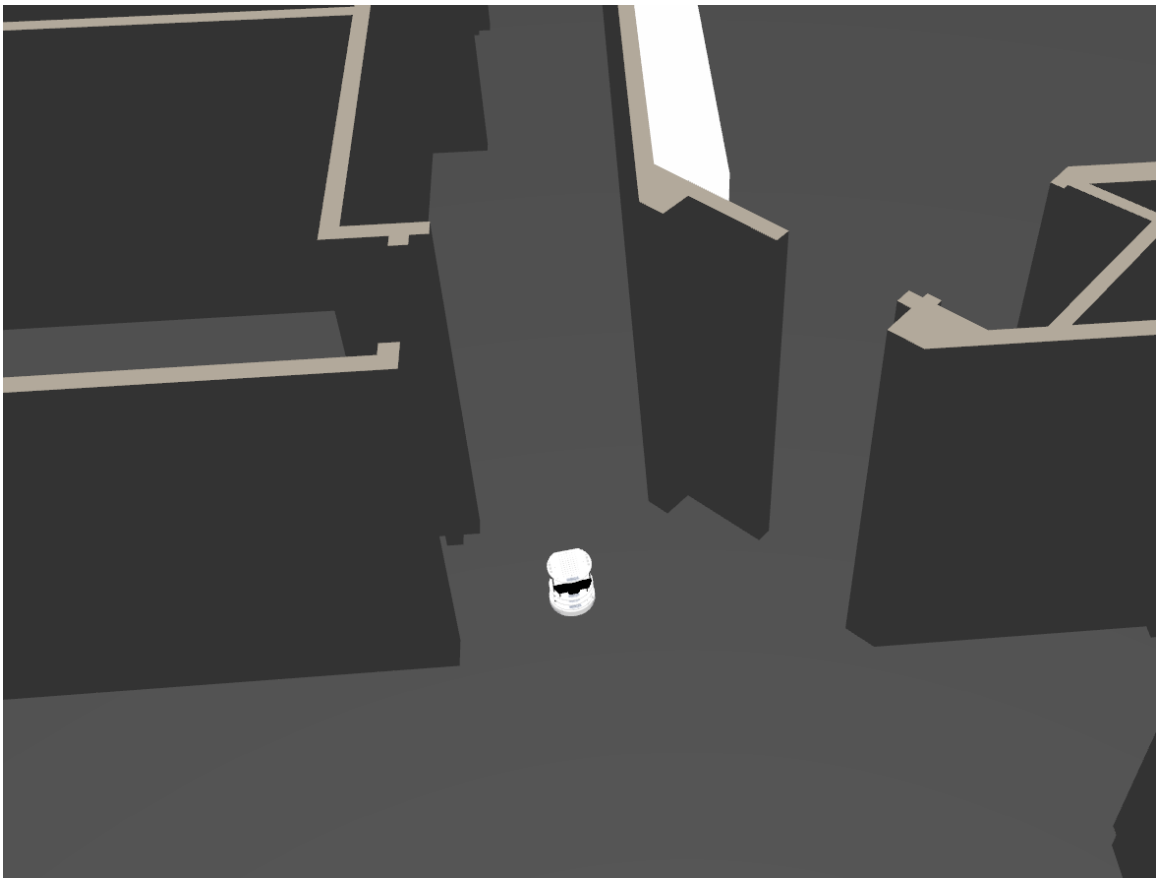


Figure 7.2: The simulated robot in its environment.

simulator's GUI and emulate sensor observations as if the robot were physically in the same environment.

To provide localization for mapping, a transform broadcaster has been written to publish *tf* data from Gazebo's */gazebo/model_states* topic. This transform broadcaster, named the */map_tf_broadcaster* node, subscribes to the *model_states* topic. This topic contains state data about each simulated model in Gazebo. This data includes the pose of the robot relative to the environment. As such, the */map_tf_broadcaster* node extracts the pose of the robot relative to the environment and adds a frame to the transform tree called */map* which is a parent of the robot's */base_footprint* frame. This transform tree is shown in Figure 7.3.

Finally, to provide map serving capabilities, the */octomap_server* node from [8] is run. This node subscribes to the */camera/depth/points* topic to provide 3D sensor observations of the environment as well as the */map_tf_broadcaster* node's */tf* topic to provide localization and subsequently uses the sensor observations to build and host a map. The node and topic connections for this system is shown in Figure 7.4.

In order to build two separate maps to be merged, two separate mapping runs are completed. In the first run, the robot is spawned using the */gazebo/model_states* topic directly for localization. The robot is then tele-operated through the environment until it reaches a rendez-vous point with the second robot and the map is saved to disk. In the second mapping run the robot is spawned in a different location. For localization, $\pi/4$ is added to yaw angle of the */gazebo/model_states* topic's orientation so that maps are built in different voxel discretizations. The individual maps as well as the combination of the two maps prior to transformation are shown in Figure 7.5.

7.1.3 Map Merging Results

The two built maps are then passed to the implemented map merging software as command-line arguments along with an initial guess of the transformation between map reference frames. The software as described in Chapter 6 converts these maps to point cloud representations and performs ICP transform refinement to obtain an improved transform between each map. The implemented software also includes

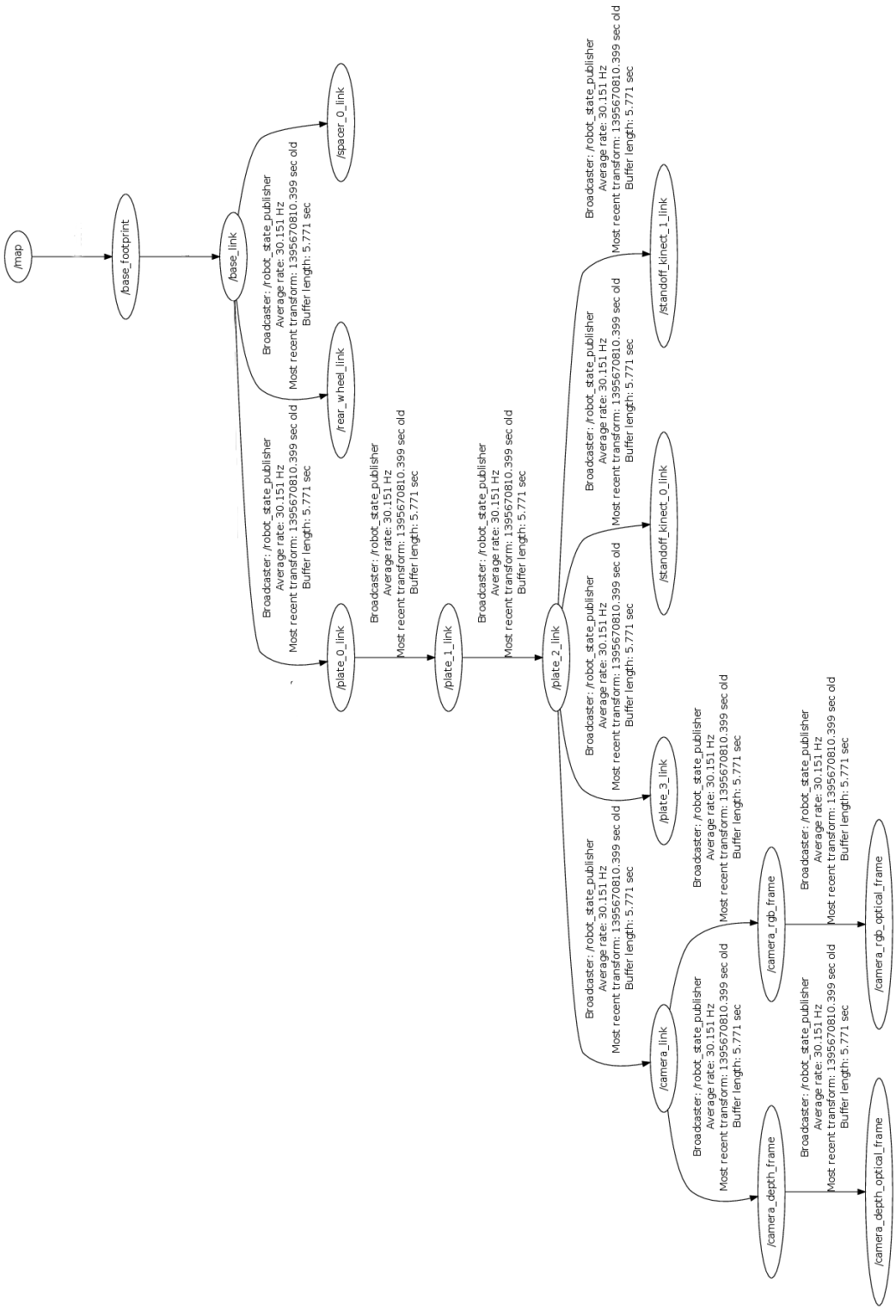


Figure 7.3: The transform tree for mapping a simulated environment.

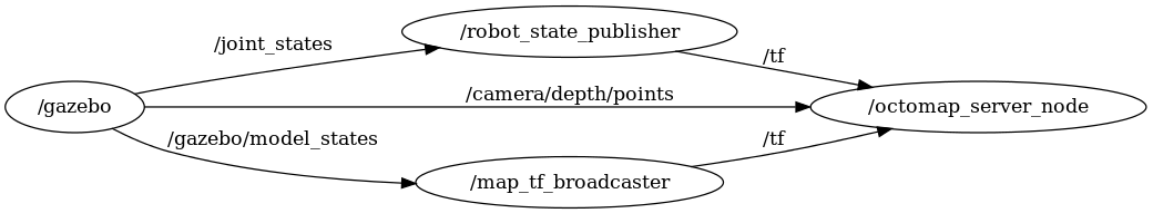


Figure 7.4: The node and topic connections for mapping a simulated environment.

visualization for the ICP alignment. As previously stated, maps are built with a $\pi/4$ reference frame rotation. Therefore they must be transformed with a $-\pi/4$ rotation to be merged correctly. Therefore erroneous initial transform estimates near the desired transformation are artificially induced and passed as command-line arguments to test the ICP refinement process. The ICP refinement is shown visually for commonly mapped portions of the simulated environment for a transform estimate error in the yaw angle of 0.265 radians in Figure 7.6 with the first map shown in red and the second in green. Figure 7.6 shows the alignment results before and after ICP refinement using the point clouds extracted from commonly mapped regions of the map

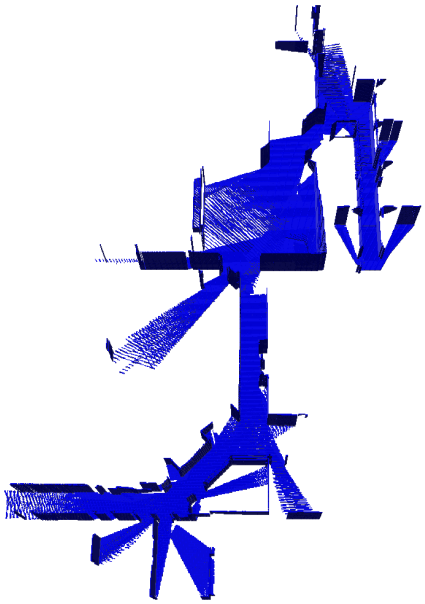
Since the transformation between both maps is known exactly, this knowledge allows us to evaluate how well the ICP refinement restored an erroneous transformation estimate. This is done using two error metrics. The first, ϵ_r, evaluates the difference of the rotation parts of the correct transformation matrix and the transformation matrix obtained by the product of the initial transformation estimate and the ICP refinement transformation matrix. The absolute value of each element of the difference of the rotation part of the two matrices is then summed to obtain ϵ_r. Where the correct matrix is:

$$T^{exact} = \begin{bmatrix} \cos\frac{\pi}{4} & \sin\frac{\pi}{4} & 0 & 0 \\ -\sin\frac{\pi}{4} & \cos\frac{\pi}{4} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

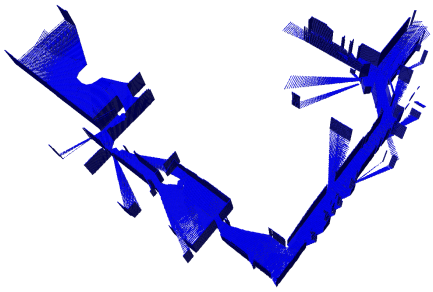
And,

$$\epsilon_r = \sum_{j=1}^{3}\sum_{k=1}^{3} |T_{jk}^{exact} - T_{1}^{2}{}_{i} T_{ICP}{}_{jk}| \tag{7.2}$$

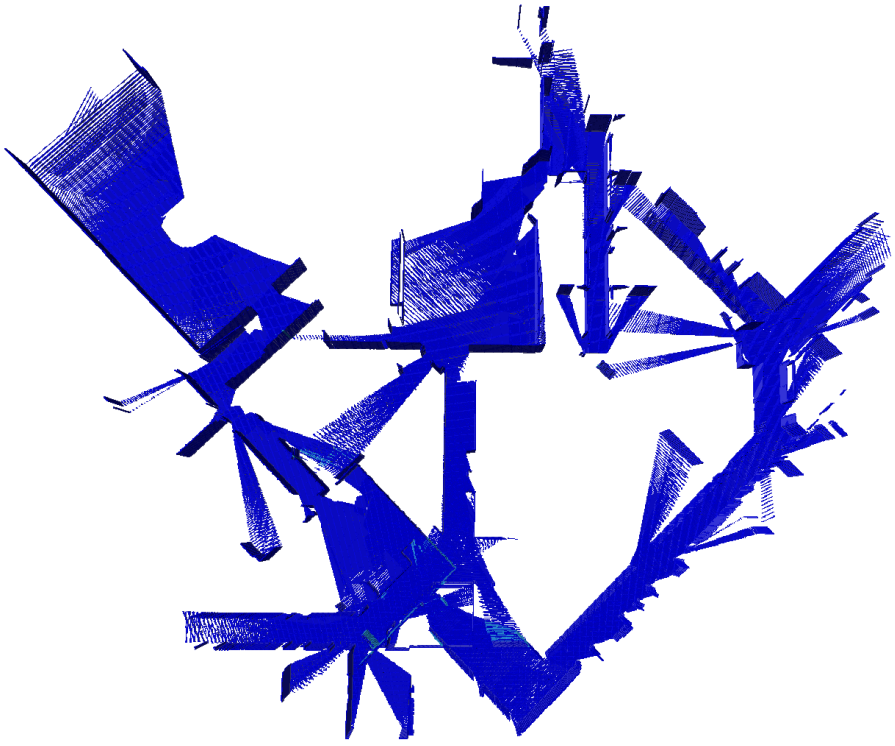
The other metric, ϵ_t, evaluates the difference of the translation parts of the correct transformation matrix and the obtained transformation matrix. The absolute value



(a) The map of the first robot.

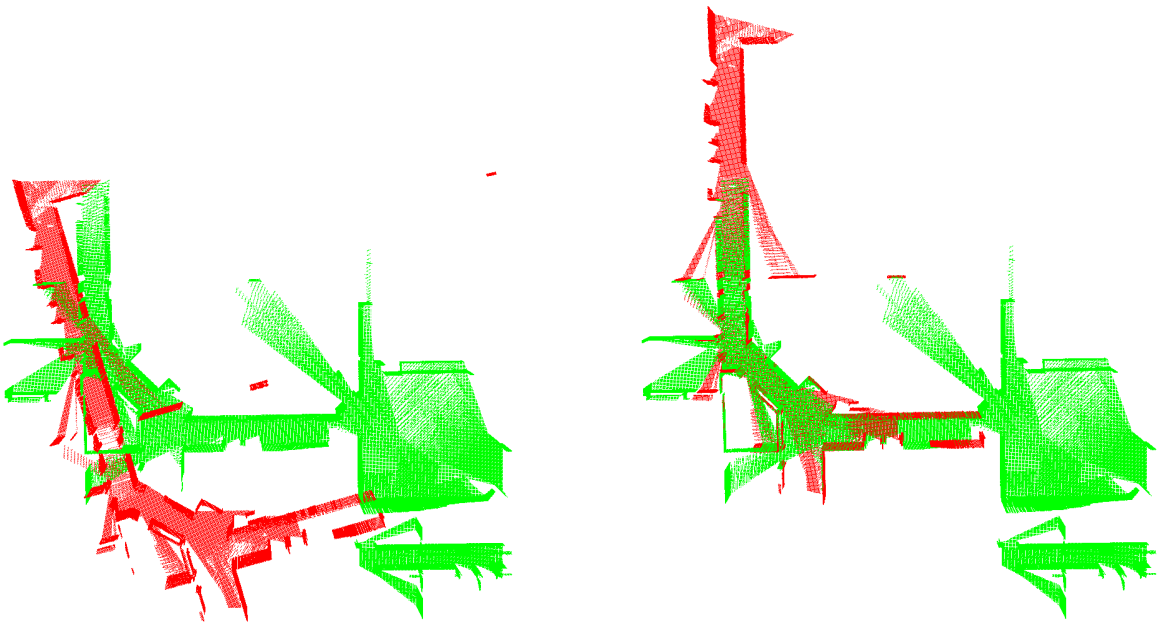


(b) The map of the second robot.



(c) The two maps prior to transformation.

Figure 7.5: Built octree occupancy grid maps of the simulated environment prior to merging.



(a) Point cloud representation of common territory with initial transform error.

(b) Point cloud representation of common territory with ICP refined transform.

Figure 7.6: The results of ICP alignment of commonly mapped territory of the simulation environment for yaw-angle initial transformation error.

of each element of the difference of the translation part of the two matrices is then summed to obtain ϵ_t. Where,

$$\epsilon_t = \sum_{j=1}^{4} |T_{jk}^{exact} - T_{1\,i}^{2} T_{ICP\,jk}|_{k=4} \tag{7.3}$$

Two transformation metrics are used rather than one single metric for the complete transformation matrix due to the fact that small orientation (rotation) errors can have a drastic effect on map alignment in comparison to moderate translation errors.

For this experiment, the transformation from the first map to the second map is a rotation about the z-axis of $\pi/4$, therefore the transformation error metric, ϵ, for several different uncertain transformations is evaluated in Table 7.1 with the erroneous transformation estimates passed to the merging software as well as the values of ϵ_r and ϵ_t before and after ICP refinement. Refinement attempts where the error in initial transform estimate was too large for ICP refinement to converge to a desired result are highlighted in red.

After observation, using visualization throughout the ICP refinement process, it

Translation (x, y, z)	Rotation $(roll, pitch, yaw)$	ϵ_r pre ICP	ϵ_t pre ICP	ϵ_r post ICP	ϵ_t post ICP
$(0, 0, 0)$	$(0, 0, -0.5)$	0.796312	0	2.18382	20.3253
$(0, 0, 0)$	$(0, 0, -0.52)$	0.741876	0	0.0255836	0.11396
$(0, 0, 0)$	$(0, 0, -0.6)$	0.521384	0	0.00492871	0.0659221
$(0, 0, 0)$	$(0, 0, -\pi/4)$	0	0	0.0677813	0.795031
$(0, 0, 0)$	$(0, 0, -0.8)$	0.0413005	0	0.0961062	1.02546
$(0, 0, 0)$	$(0, 0, -0.87)$	0.239007	0	0.164128	1.82668
$(0, 0, 0)$	$(0, 0, -0.9)$	0.323436	0	0.658751	6.70871
$(1, 0, 0)$	$(0, 0, -\pi/4)$	0	1	0.0605207	0.712245
$(0, 1, 0)$	$(0, 0, -\pi/4)$	0	1	0.0753039	0.884319
$(1, 1, 1)$	$(0, 0, -0.6)$	0.521384	3	0.0274716	0.357402
$(1, 1, 1)$	$(0, 0, -0.8)$	0.0413005	3	0.0410986	0.469403
$(0, 0, 0)$	$(-0.1, 0, -\pi/4)$	0.25308	0	0.448877	0.0352201
$(0, 0, 0)$	$(0, -0.1, -\pi/4)$	0.25308	0	1.17584	8.70397
$(0, 0, 0)$	$(-0.1, -0.1, -\pi/4)$	0.371481	0	0.0762914	0.794496
$(1, 1, 0)$	$(-0.1, 0, -\pi/4)$	0.25308	2	0.0150514	0.187039
$(0, 0, 1)$	$(-0.1, 0, -\pi/4)$	0.25308	1	21.5134	3.30602
$(1, 1, 0)$	$(0, -0.1, -\pi/4)$	0.25308	2	8.64066	1.16931
$(0, 0, 1)$	$(0, -0.1, -\pi/4)$	0.25308	1	0.24857	1.18748

Table 7.1: ICP alignment error evaluation for simulated map merging.

was observed that ϵ_r values less than 0.25 and ϵ_t values less than 0.9 yielded coherent merging results. As seen in Table 7.1 the ICP refinement implementation allows transformation correction for a wide range of transformation errors. The ICP refinement algorithm was able to correct yaw-angle errors up to 0.265 radians as shown in the second row of Table 7.1. However for angles larger than this such as an erroneous yaw rotation estimate of -0.5 (row 1 of Table 7.1), the ICP refinement does not converge to a desired result. This is often due to the fact that the distance between corresponding points is so great that the ICP algorithm converges to local minima rather than the absolute minima. One thing that is important to note is that the ICP refinement strategy is often not effective in the presence of error in roll and pitch angles. This is due to the fact that when intersecting bounding boxes of each are used with the initial transformation estimate to determine commonly mapped territory, the extraction fails to produce sufficient data sets for registration. This is shown in rows 13, 14, 17, and 18 of Table 7.1.

Once these refined transformations are obtained, the transformations are then applied to the maps then merged. Using the *octovis* visualization to the coherency of the maps are qualitatively evaluated. Figure 7.7 shows the result of a map merger with exact knowledge of the transformation between the two maps. Dark blue voxels correspond to high occupancy where as lighter shades of blue represent lower occupancy. It can be seen that data from the second map is interpreted as lower occupancy, however this is due to the fact that data from the edge of occupied territory is spread over several voxels due to the $\pi/4$ rotation. This being said, after being transformed to the other robot's reference frame the data is still coherent, as free and occupied space, remains as such and additional information is added to the map.

Figure 7.8 also shows the results of a merger with an erroneous initial transform. This figure compares the coherency of map mergers with and without ICP transform refinement. The merger in this example has a yaw-angle error of 0.265 radians (row 1 of Table 7.1). Figure 7.8 demonstrates that the map is satisfactorily aligned by the ICP refinement, allowing the implemented map merging algorithm to be a suitable candidate for merging when transformation estimates are obtained from uncertain

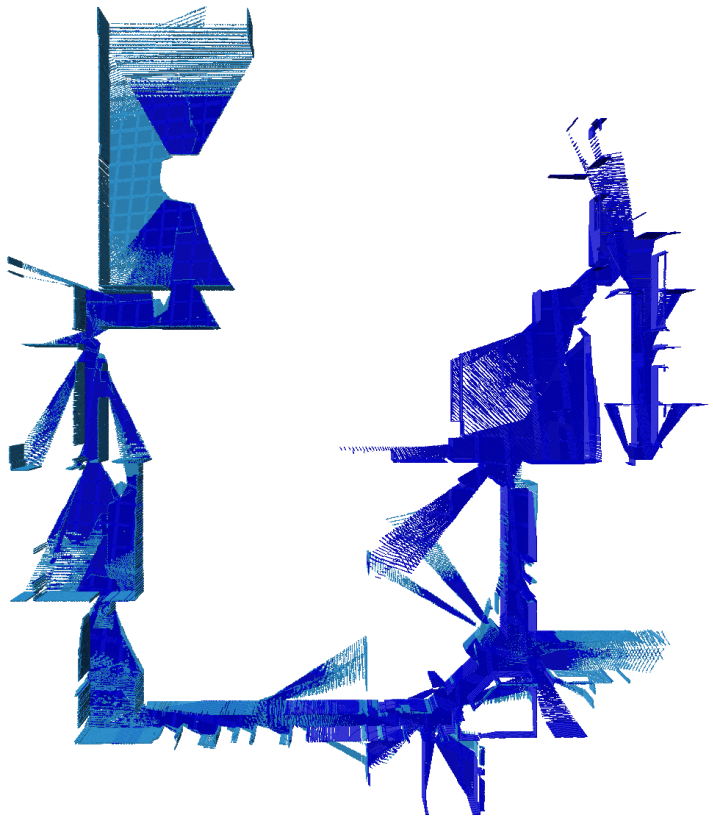


Figure 7.7: The merged map of the simulated environment with exact transformation knowledge

sensor observations.

7.2 Real-World Map Merging

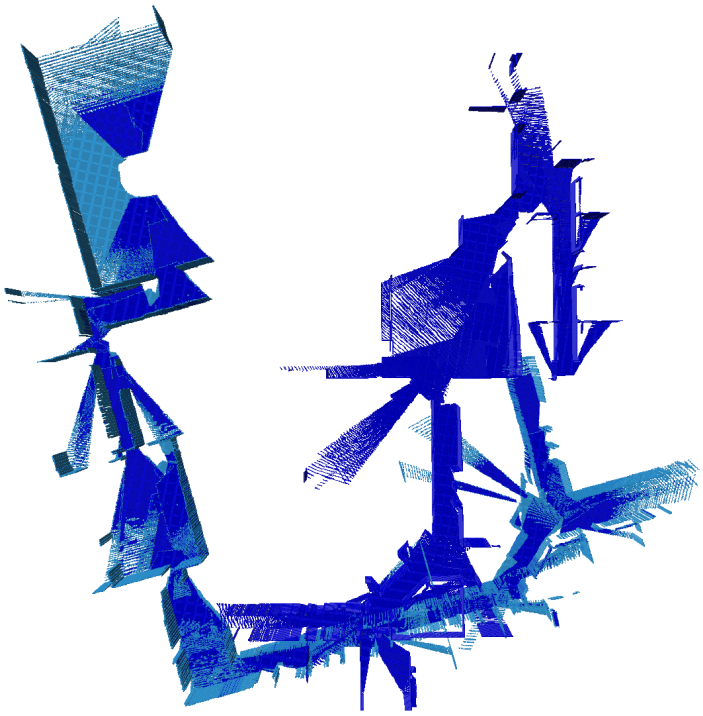
In this section the merger of two maps built within a laboratory environment is presented. In this experiment, an environment is set up for a Turtlebot to traverse and build maps, each with a focus on different landmarks so that the merger of the two maps will provide additional data for each of these landmarks. In the first mapping run the robot circles the box formation to obtain information about both sides of the box formation while omitting detailed observations about the hollow square wood structure. In the second mapping run the robot makes observations about the edges of the hollow square structure while omitting the far side of the box structure. This allows the merged map to contain detailed information about both the box formation and the perimeter of the the hollow square structure.

7.2.1 Mapping Environment

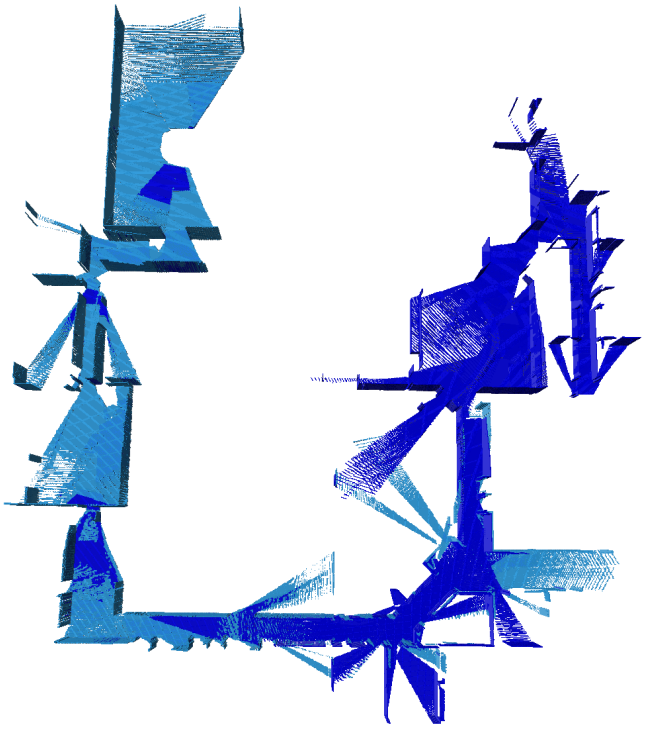
For this experiment, a Turtlebot UGV explores a physical environment within the RMCC robotics laboratory. The robot then uses a Microsoft Kinect sensor to extract 3D observations of this environment. The environment is shown in Figure 7.9 with two landmarks. These two landmarks are set up such that two separate mapping runs obtain detailed information about each landmark.

7.2.2 Map Building

To build the first map, the robot is placed to the left of the boxes in the left of Figure 7.9. The robot then follows a path around the boxes on the left of Figure 7.9 making detailed observations about the boxes. The path of the first robot is also shown in yellow. In the second mapping run, the robot travels the perimeter of the wooden fence like structure in the right of the image to build a detailed map of the wooden structure. The path of the second robot is shown in red.



(a) The merged result in simulation without ICP refinement.



(b) The merged result in sumulation with ICP refinement.

Figure 7.8: Built octree occupancy grid maps of the simulated environment after merging with and without ICP refinement.

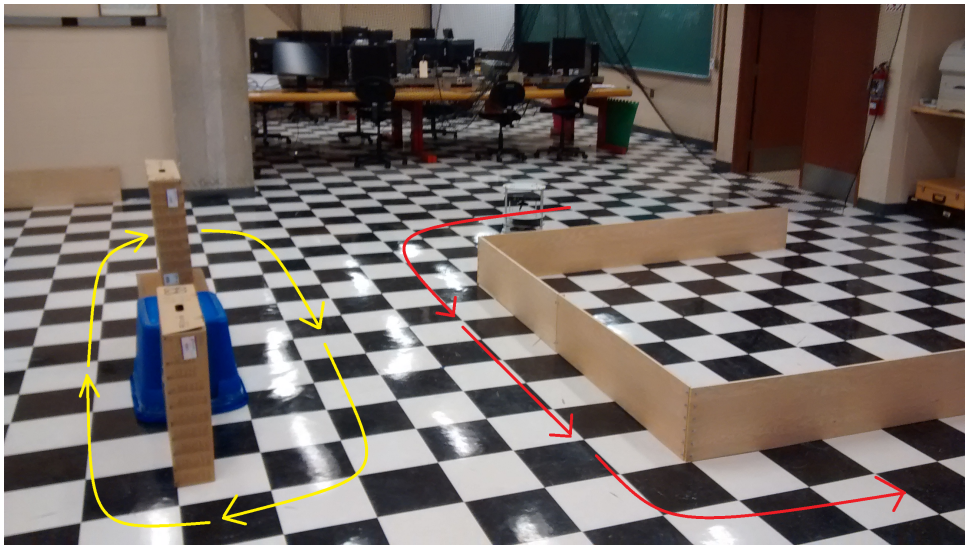


Figure 7.9: The experimental mapping environment.

In each mapping run the Turtlebot UGV is booted with a provided system image which automatically provides the majority of needed sensor drivers. A developed launch script is then used to run the additional nodes required for mapping. This includes the driver for the Microsoft Kinect which publishes point cloud data on the */camera/depth/points* topic. Additionally, a natural point "Optitrack" motion capture system is used to provide localization. The "Optitrack" system is an optical motion capture system which is able to accurately track in 6DOF the position and orientation of a predefined constellation of reflectors. A set of these reflectors is attached to the Turtlebot UGV such that it may be tracked within the environment. The cameras are then connected to a server which streams the robot's pose information Using the *NatNet* protocol to the ROS system. The */mocap_node* is then used to receive this information and publish pose messages in a ROS topic, */lps*. A similar node to the */map_tf_broadcaster* from the simulation is then used to publish *tf* data by subscribing to the */lps* topic. Finally, a *rosbag* recorder is used to record all publications to sensor observation and localization topics so that maps can be built offline using any desired discretization.

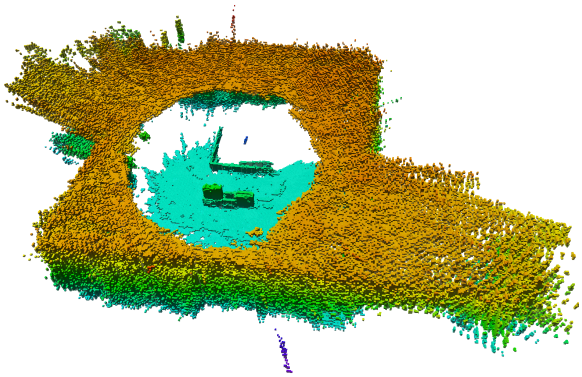
Offline, the Octomap server is run to provide map serving capabilities. The sensor observations and localization are then replayed using the *bag* file obtained from the

first mapping run. The server then subscribes to the point cloud topics as well as required */tf* topics for appropriate map building. The */octomap_saver* node is then run which saves the built map to a file. For the second map, the same process is repeated with the exception that a simple transform broadcaster is used to add an additional transform to the system which is a parent of the transform provided by the Optitrack system. This frame is rotated by $\pi/4$ from the Optitrack frame so that the two maps are not built in common voxel discretizations. The node and topic interconnections for the experimental mapping runs are shown in Figure 7.11, while the transform tree remains the same. The individual maps as as well as the maps shown together prior to transformation are shown in Figure 7.10 with height color encoding to make interpretation of 3D shapes easier.

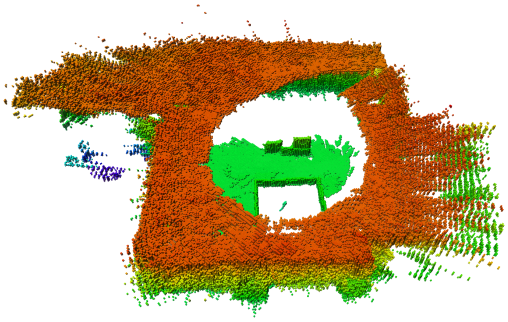
7.2.3 Map Merging Results

Once again, the maps are then passed to the implemented map merging software as command-line arguments along with an initial guess of the transformation between map reference frames. Several different initial guesses for the transformation are used to evaluate the effectiveness of ICP transform refinement for erroneous initial guesses using the same ϵ_r and ϵ_t as described in Section 7.1.3. The results of ICP refinement for each transform estimate is shown in Table 7.2 with the erroneous transformation estimates passed to the merging software as well as the values of ϵ_r and ϵ_t before and after ICP refinement. Once again, refinement attempts where the error in initial transform estimate was too large for ICP refinement to converge to a desired result are highlighted in red.

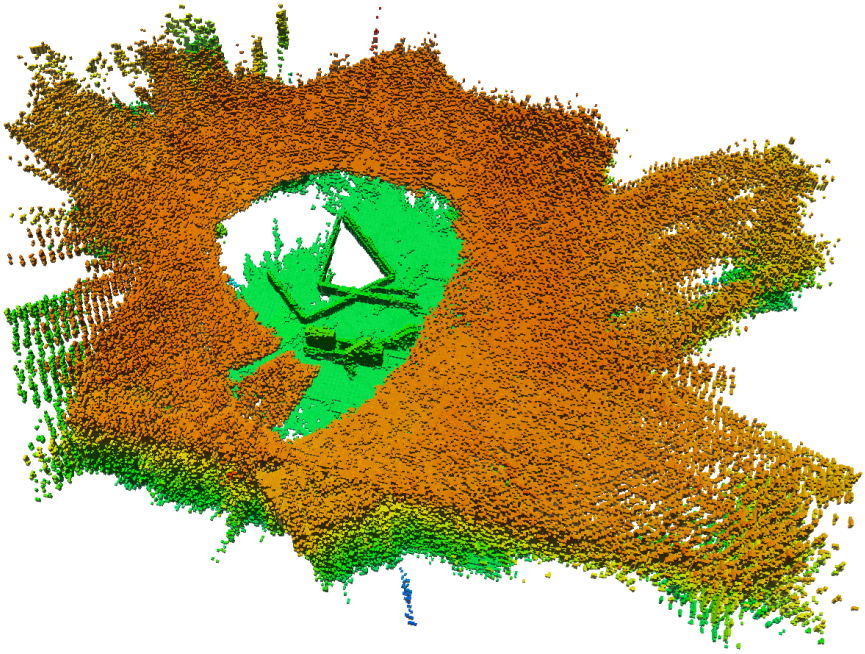
As predicted, ICP remains a suitable candidate to refine transformation estimates for uncertain transformations between maps. In the provided example a vast range of transformation errors were able to be corrected including yaw angles of up to 0.685 radians of error (row 2 of Table 7.2), where the refinement process is shown in Figure 7.12. For several erroneous transformation estimates ϵ_r and ϵ_t converged to desired results. Additionally, the magnitude of the error that the ICP refinement algorithm was able to reduce was far greater. This is due to the fact that the maps of the



(a) The map of the first robot.



(b) The map of the second robot.



(c) The two maps prior to transformation.

Figure 7.10: Built octree occupancy grid maps of the real-world laboratory environment prior to merging.

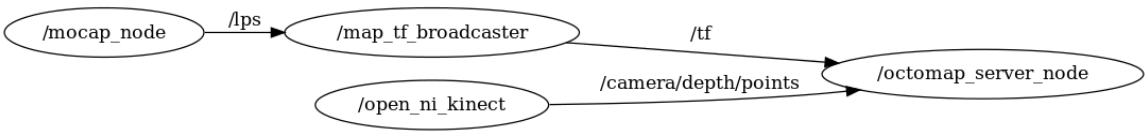
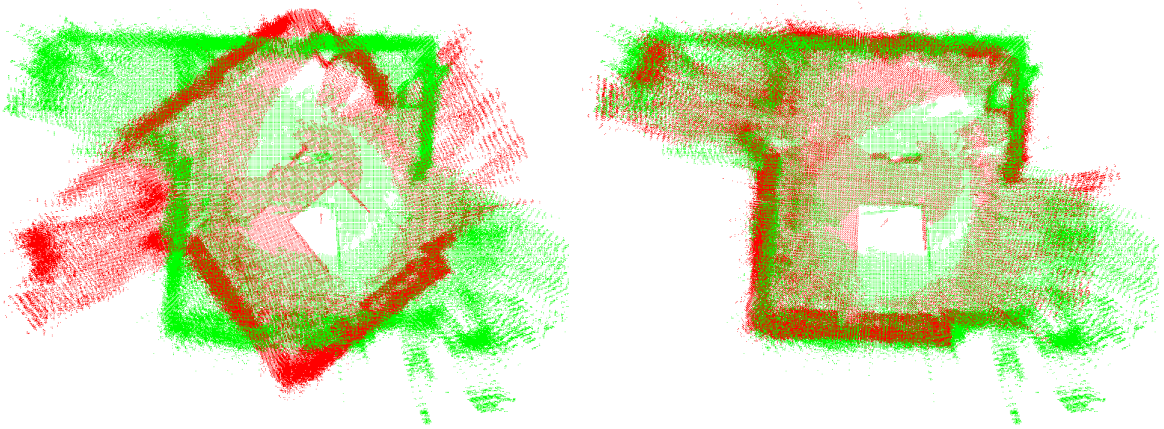


Figure 7.11: The node and topic connections for mapping a real-world environment

Translation (x, y, z)	Rotation $(roll, pitch, yaw)$	ϵ_r pre ICP	ϵ_t pre ICP	ϵ_r post ICP	ϵ_t post ICP
$(0, 0, 0)$	$(0, 0, -0.07)$	1.85521	0	2.695	1.07009
$(0, 0, 0)$	$(0, 0, -0.1)$	1.79034	0	0.121203	0.153779
$(0, 0, 0)$	$(0, 0, -0.5)$	0.796312	0	0.121749	0.166057
$(0, 0, 0)$	$(0, 0, -\pi/4)$	0	0	0.207669	0.1337277
$(0, 0, 0)$	$(0, 0, -0.8)$	0.0413005	0	0.123183	0.196204
$(0, 0, 0)$	$(0, 0, -1.5)$	1.85352	0	0.138249	0.133851
$(0, 0, 0)$	$(0, 0, -1.7)$	2.24102	0	2.88717	0.683354
$(1, 0, 0)$	$(0, 0, -\pi/4)$	0	1	0.135229	0.198153
$(0, 1, 0)$	$(0, 0, -\pi/4)$	0	1	0.158971	0.228493
$(1, 1, 1)$	$(0, 0, -0.6)$	0.521384	3	0.188403	0.272216
$(1, 1, 1)$	$(0, 0, -0.8)$	0.0413005	3	0.202992	0.202992
$(0, 0, 0)$	$(-0.3, 0, -\pi/4)$	0.821276	0	0.17818	0.235683
$(0, 0, 0)$	$(0, -0.3, -\pi/4)$	0.821276	0	0.103531	0.163994
$(0, 0, 0)$	$(-0.3, -0.3, -\pi/4)$	0.371481	0	0.0762914	0.794496
$(1, 1, 0)$	$(-0.3, 0, -\pi/4)$	0.821276	2	0.240368	0.32025
$(0, 0, 1)$	$(-0.3, 0, -\pi/4)$	0.821276	1	0.259235	0.41867
$(1, 1, 0)$	$(0, -0.3, -\pi/4)$	0.821276	2	0.180602	0.250326
$(0, 0, 1)$	$(0, -0.3, -\pi/4)$	0.821276	1	0.104994	0.218853

Table 7.2: ICP alignment error evaluation for real-world map merging.



(a) Point cloud representation of common territory with initial transform error.

(b) Point cloud representation of common territory with ICP refined transform.

Figure 7.12: The results of ICP alignment of commonly mapped territory of the real-world laboratory environment for yaw-angle initial transformation error.

experimental environment had a far greater portion of commonly mapped territory than the simulated environment. As well, the experimental sensor had a greater range than the simulated sensor. This being said, some failures were observed in rows 1 and 7 of Table 7.2 due to the fact that the ICP algorithm once again converged to local minima, rather than absolute minima.

Given that a reliable transformation refinement process is in place, this allows us to merge the results from one mapping run with another to have detailed data about both landmarks in the environment. The merged map with exact knowledge of the correct transform between maps is shown in Figure 7.13. Additionally the contrast of the merged map with an erroneous yaw angle estimate of 0.685 radians (row 2 of Table 7.2) with and without ICP refinement is shown in Figure 7.14.

7.3 Conclusion

This chapter has presented success in map merging for both simulated and real-world environments. This success has shown that the theory for merging octree occupancy grids as presented throughout this work is indeed valid. The results presented in this chapter will be brought into context with the complete work in Chapter 8. Addition-

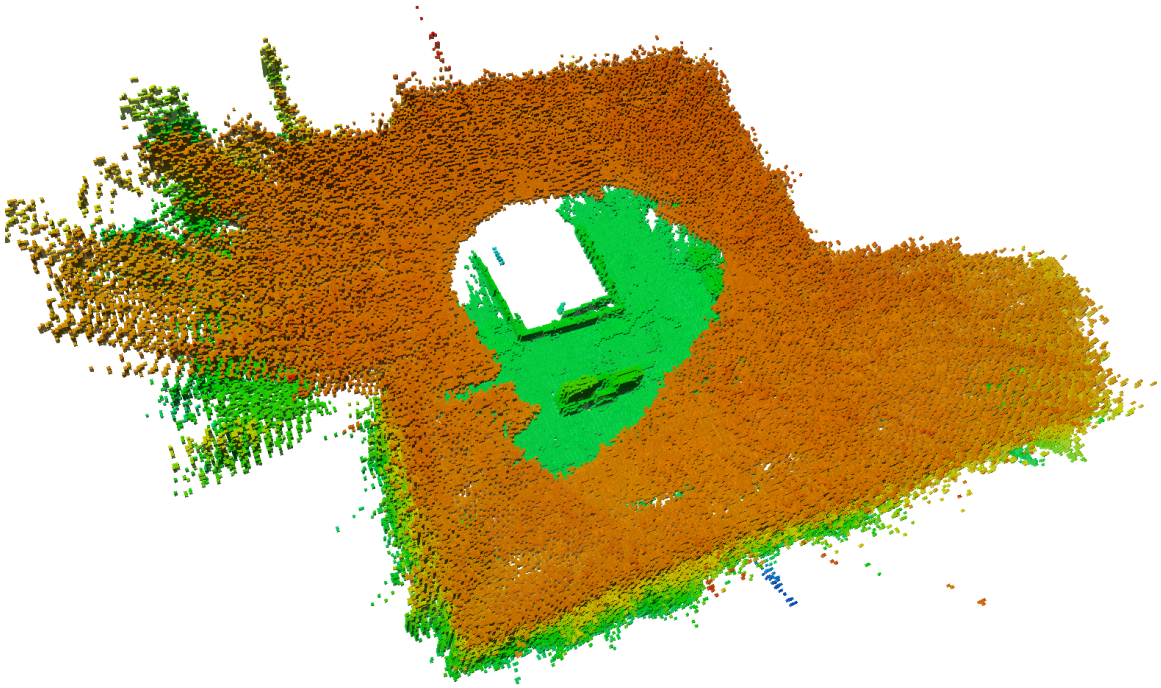
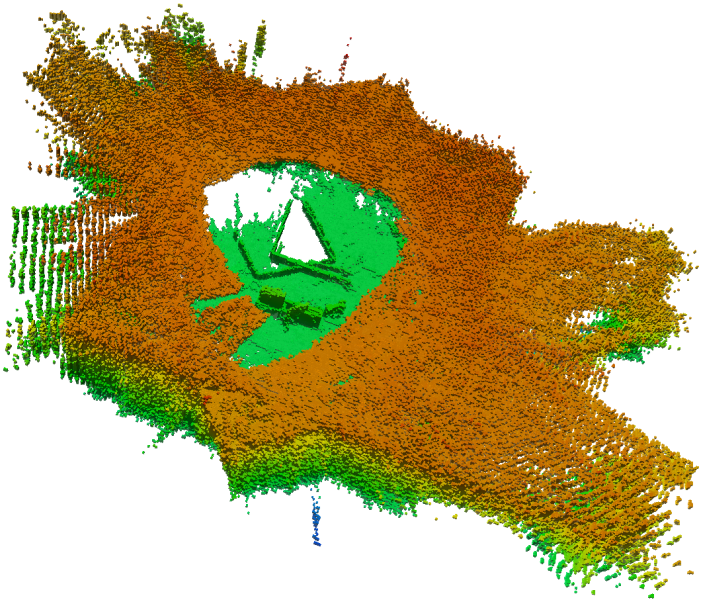
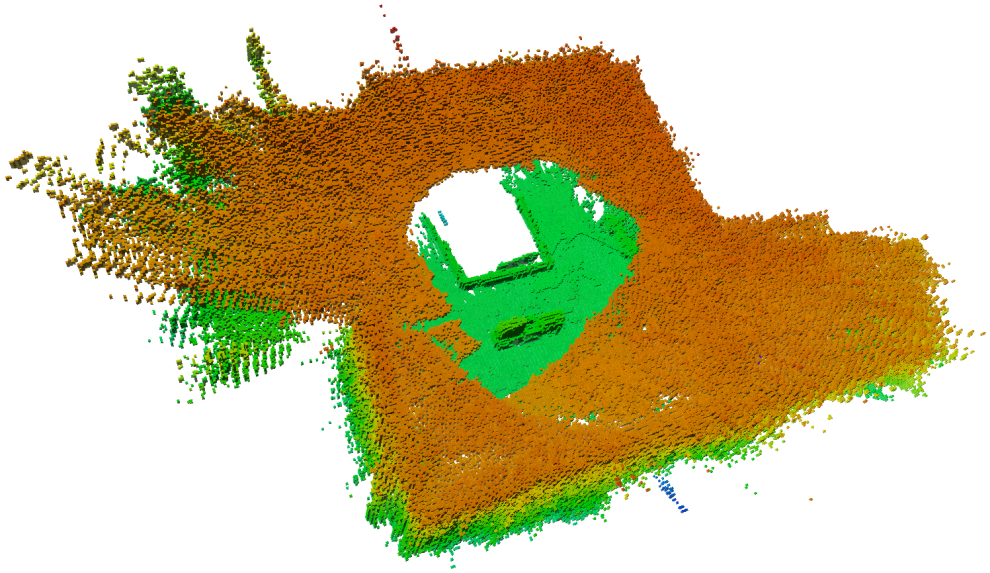


Figure 7.13: The merged map of the real-world laboratory environment with exact transformation knowledge

ally, Chapter 8 will also explore potential avenues for improvement of this research through future work.



(a) The merged real-world result without ICP refinement.



(b) The merged real-world result with ICP refinement.

Figure 7.14: Built octree occupancy grid maps of the real-world laboratory environment after merging with and without ICP refinement.

Chapter 8

Conclusion

Our research was motivated by the lack of a suitable 3D mapping framework for multi-robot applications.

In Chapter 2 we motivated our research by presenting a summary of robotic mapping techniques as well as recent developments in 3D map representations.This chapter also included a description of the octree occupancy grid map and the advantages of its use in 3D mapping.

In Chapter 3 we continued to motivate our research by reviewing and summarizing the problem of multi-robot map merging. This chapter introduced the concept of transforming one map into another's frame of reference as well as ways to implement this transform when a discretized environment model is used. This chapter also introduced the use of registration techniques with commonly mapped territory as a way to improve transform estimates between maps.

In Chapter 4 we introduced the software resources that would be used to support this research. This included a description of ROS, the middle-ware used in the deployment of the simulated and real-world robots, Octomap, the library and map-building frameword for 3D mapping with 3D occupancy grids, as well as PCL, the library used for transform refinement.

In Chapter 5, the strategies used to overcome the problem of merging octree occupancy grids were described. These strategies overcame the problem of transforming maps from one reference to the other despite the volume discretization, as well as the problem of merging maps together with volumes mapped at different levels of the tree hierarchy. Subsequently, in Chapter 6 the implementation of each of these strategies in to software algorithms was presented

In Chapter 7 the proposed strategies for octree occupancy grid merging were performed and validated with both simulated and experimental results. For each environment, two maps were built independently of one another and successfully

merged with both exact and uncertain transformation estimates between each map.

8.1 Contributions

This thesis made several contributions to the field of autonomous robotics:

1. The proposal of a valid solution for merging data from two independent octree occupancy grids where data is mapped at different depths in the octree tree hierarchy. The solution uses local map expansion to minimize the computational complexity of the overall merger in comparison to complete map expansion.

2. The proposal of a valid solution for transforming maps with a octree occupancy grid representation. The solution determines the bounding volume of the transformed map and subsequently looks up the correct occupancy for each transformed voxel using trilinear interpolation with the source voxels.

3. The proposal of a solution for refining transformation estimates between the reference frames of two maps using registration techniques with commonly mapped portions of the environment. The solution determines a subset of each map using commonly mapped parts of the environment with intersecting bounding volumes. A point cloud representation of map subset is then created and ICP registration is performed to obtain a refined transform between each map's frame of reference.

4. The verification and validation of each proposed solution was subsequently implemented in both simulated and real-world environment. Each proposed solution was shown to be successful in the creation of coherent maps which closely represented the environment.

8.2 Future Work

Although the use of intersecting bounding volumes produced suitable extraction of commonly mapped parts of the environment for ICP registration for a wide variety

of transform errors, there were certain transform errors such as large errors in roll and pitch angles that caused this strategy to fail to produce point clouds suitable for ICP convergence. As such, an area that may be improved is the extraction of commonly mapped volumes in the environment. Alternatives to the technique proposed in this work could be explored for better results. Currently PCL includes several segmentation algorithms that could be explored to extract common shapes in each map regardless of position as determined by the initial transform estimate.

In this work, the only registration technique that was used was the ICP algorithm with point clouds extracted using the centre of each occupied leaf node in the maps. While this technique is useful for a large range of transformation errors this range could be improved by additional registration techniques. One possible avenue to explore is the extraction of feature descriptors on each generated point cloud and to perform initial alignment based on correspondence matching. ICP could subsequently be used to further improve the transformation after an initial alignment using a feature based registration technique.

The full integration of this work into ROS would make the success of this work more suitable for others in the autonomous robotics community. Currently, the map merging software runs offline using maps saved to disk and the transform estimate given as a command-line argument. The full integration of this software into ROS would consist of a node which requests maps as a service from other map servers much like the *octomap_saver* node included with the *octomap_mapping* package. This node could then use localization data from the ROS system as initial transformation data and load the merged map into a local Octomap server.

8.3 Conclusion

Our work has shown that an Octree occupancy grid representation of the environment is not only a suitable candidate for 3D mapping with individual robots, but for multi-robot mapping applications as well. Our work has also shown that existing techniques used for multi-robot mapping and map merging in 2D may be used for multi-robot mapping, however with the added complexity of the additional 3 DOF in a

3D system. This leads us to conclude that Octree occupancy grid map representations combined with the contributions of this work have exciting potential for use in future applications.

References

[1] Jay Thor Turner. A real-time implementation of a subsumtion based robot control system. Master's thesis, Royal Military College of Canada, Kingston, Canada, 2013.

[2] Pierre Dinnissen. Using reinforcement learning in multi-robot slam. Master's thesis, Carleton University, Ottawa, Canada, 2011.

[3] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. *Autonomous robot vehicles*, 1:167–193, 1990.

[4] Pierre Payeur, Patrick Hébert, Denis Laurendeau, and Clément M Gosselin. Probabilistic octree modeling of a 3d dynamic environment. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 1289–1296, Albequerque, United States, 1997. IEEE.

[5] Jonathan Fournier, Benoit Ricard, and Denis Laurendeau. Mapping and exploration of complex environments using persistent 3d model. In *Fourth Canadian Conference on Computer and Robot Vision*, pages 403–410, Montreal, Canada, 2007. IEEE.

[6] Kaustubh Pathak, Andreas Birk, Jann Poppinga, and Sören Schwertfeger. 3d forward sensor modeling and application to occupancy grid based sensor fusion. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2059–2064, San Diego, United States, 2007. IEEE.

[7] Nathaniel Fairfield, George Kantor, and David Wettergreen. Real-time slam with octree evidence grids for exploration in underwater tunnels. *Journal of Field Robotics*, 24(1-2):03–21, 2007.

[8] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013.

[9] Armin Hornung, Kai M. Wurm, and Maren Bennewitz. Humanoid robot localization in complex indoor environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, 2010.

[10] Kevin Murphy et al. Bayesian map learning in dynamic environments. *Advances in Neural Information Processing Systems (NIPS)*, 12:1015–1021, 1999.

[11] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the National conference on Artificial Intelligence*, pages 593–598, Edmonton, Alberta, 2002.

[12] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *International Joint Conference on Artificial Intelligence*, volume 18, pages 1151–1156, San Francisco, United States, 2003. LAWRENCE ERLBAUM ASSOCIATES LTD.

[13] Kai M Wurm, Cyrill Stachniss, and Giorgio Grisetti. Bridging the gap between feature-and grid-based slam. *Robotics and Autonomous Systems*, 58(2):140–148, 2010.

[14] Hans Moravec and Alberto Elfes. High resolution maps from wide angle sonar. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 2, pages 116–121, San Francisco, United States, 1985. IEEE.

[15] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabolistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[16] Yuval Roth-Tabak and Ramesh Jain. Building an environment model using depth information. *Computer*, 22(6):85–90, 1989.

[17] David M Cole and Paul M Newman. Using laser range data for 3d slam in outdoor environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1556–1563, Orlando, United States, 2006. IEEE.

[18] Andreas Nüchter, Kai Lingemann, Joachim Hertzberg, and Hartmut Surmann. 6d slam—3d mapping outdoor environments. *Journal of Field Robotics*, 24(8-9):699–722, 2007.

[19] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[20] Wolfram Burgard, Mark Moors, Dieter Fox, Reid Simmons, and Sebastian Thrun. Collaborative multi-robot exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 476–481, San Francisco, United States, 2000. IEEE.

[21] N Ergin Özkucur and H Levent Akın. Cooperative multi-robot map merging using fast-slam. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 449–460. Springer, 2010.

[22] Andreas Birk and Stefano Carpin. Merging occupancy grid maps from multiple robots. *Proceedings of the IEEE*, 94(7):1384–1397, 2006.

[23] Pierre Dinnissen, Sidney N Givigi, and Howard M Schwartz. Map merging of multi-robot slam using reinforcement learning. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 53–60, Seoul, Korea, 2012. IEEE.

[24] George J Grevera and Jayaram K Udupa. *IEEE Transactions on Medical Imaging*, 17(4):642–652, 1998.

[25] Alan W Paeth. A fast algorithm for general raster rotation. In *Graphics Interface*, volume 86, pages 77–81, Vancouver, Canada, 1986.

[26] At Tanaka, M Kameyama, S Kazama, and O Watanabe. A rotation method for raster image using skew transformation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 272–277, Miami, United States, 1986.

[27] Peter Schröder and James B Salem. Fast rotation of volume data on data parallel architectures. In *Proceedings of the 2nd conference on Visualization*, pages 50–57, San Diego, United States, 1991. IEEE Computer Society Press.

[28] Baoquan Chen and Arie Kaufman. 3d volume rotation using shear transformations. *Graphical Models*, 62(4):308–322, 2000.

[29] Paul J Besl and Neil D McKay. A method for registration of 3-d shapes. *IEE Transactions on Pattern Analysis and Machine Intelligence*, pages 239–256, 1992.

[30] Marcus Strand, Frank Erb, and Rüdiger Dillmann. Range image registration using an octree based matching strategy. In *International Conference on Mechatronics and Automation*, pages 1622–1627, Harbin, China, 2007. IEEE.

[31] Andreas Birk and Stefano Carpin. Merging occupancy grid maps from multiple robots. *IEEE Proceedings, special issue on Multi-Robot Systems*, 94(7):1384–1397, 2006.

[32] Robot operating system. http://www.ros.org.

[33] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, 2011.

Appendices

Appendix A

Source Code for Map Merging Algorithms

A.1 Source Code for Algorithms 1 to 3

```
#include <Eigen/SVD>
#include <pcl/common/common.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/filter.h>
#include <pcl/features/normal_3d.h>
#include <pcl/registration/icp.h>
#include <pcl/registration/icp_nl.h>
#include <pcl/registration/transforms.h>

//convenient typedefs
typedef pcl::PointXYZ PointT;
typedef pcl::PointCloud<PointT> PointCloud;
typedef pcl::PointNormal PointNormalT;
typedef pcl::PointCloud<PointNormalT> PointCloudWithNormals;

#define MAX_ITER 500

void tree2PointCloud(OcTree *tree,
                     pcl::PointCloud<pcl::PointXYZ>& pclCloud) {
    // now, traverse all leafs in the tree:
    for (OcTree::leaf_iterator it = tree->begin_leafs(),
         end = tree->end_leafs(); it != end; ++it)
    {
        if (tree->isNodeOccupied(*it)){
            pclCloud.push_back(
                        pcl::PointXYZ(it.getX(),
                                      it.getY(),
                                      it.getZ()
                                      )
                        );
        }
    }

}


bool pointInBBox(pcl::PointXYZ& point,
                 pcl::PointXYZ& bboxMin,
                 pcl::PointXYZ& bboxMax){

    return (point.x < bboxMax.x && point.x > bboxMin.x) &&
           (point.y < bboxMax.y && point.y > bboxMin.y) &&
           (point.z < bboxMax.z && point.z > bboxMin.z);
```

```
}

Eigen::Matrix4f getICPTransformation(
        pcl::PointCloud<pcl::PointXYZ>& cloud1,
        pcl::PointCloud<pcl::PointXYZ>& cloud2,
        Eigen::Matrix4f& tfEst,
        double mapRes) {

    //apply the tfEst to cloud 2

    pcl::transformPointCloud(cloud2,cloud2,tfEst);

    //get the bounding region of cloud1 to
    //extract the points from cloud 2 contained in the region
    pcl::PointXYZ minCloud1; pcl::PointXYZ maxCloud1;
    pcl::getMinMax3D(cloud1,minCloud1,maxCloud1);

    //filter out the points in cloud 2 that are not in cloud 1's range
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud2filtered (
                new pcl::PointCloud<pcl::PointXYZ>);

    for(pcl::PointCloud<pcl::PointXYZ>::iterator it=cloud2.begin();
        it!=cloud2.end(); it++){

        if(pointInBBox(*it,minCloud1,maxCloud1)) {
            cloud2filtered->push_back(*it);
        }
    }

    //filter out the points in cloud 1 that are not in cloud 2's range
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud1filtered (
                new pcl::PointCloud<pcl::PointXYZ>);

    //same for other cloud
    pcl::PointXYZ minCloud2filtered; pcl::PointXYZ maxCloud2filtered;
    pcl::getMinMax3D(*cloud2filtered,minCloud2filtered, maxCloud2filtered);

    minCloud2filtered = pcl::PointXYZ(
                minCloud2filtered.x-1,
                minCloud2filtered.y-1,
                minCloud2filtered.z-1
                );

    maxCloud2filtered = pcl::PointXYZ(
                maxCloud2filtered.x+1,
                maxCloud2filtered.y+1,
                maxCloud2filtered.z+1
                );

    for(pcl::PointCloud<pcl::PointXYZ>::iterator it=cloud1.begin();
        it!=cloud1.end(); it++){

        if(pointInBBox(*it,minCloud2filtered,maxCloud2filtered)) {
            cloud1filtered->push_back(*it);
```

```
    }
}

// Downsample for consistency and speed
PointCloud::Ptr src (new PointCloud);
PointCloud::Ptr tgt (new PointCloud);
pcl::VoxelGrid<PointT> grid;
grid.setLeafSize (10*mapRes, 10*mapRes, 10*mapRes);
grid.setInputCloud (cloud1filtered);
grid.filter (*tgt);

grid.setInputCloud (cloud2filtered);
grid.filter (*src);

// Align
pcl::IterativeClosestPointNonLinear<PointT, PointT> reg;
reg.setTransformationEpsilon (mapRes/60);
reg.setMaxCorrespondenceDistance (10*mapRes);

Eigen::Matrix4f Ti = Eigen::Matrix4f::Identity (), prev;
PointCloud::Ptr reg_result;

if(src->size() < tgt->size()){

    reg.setInputCloud (src);
    reg.setInputTarget (tgt);

    // Run the same optimization in a loop and visualize the results
    reg_result = src;
    reg.setMaximumIterations (2);
    for (int i = 0; i < MAX_ITER; ++i)
    {

        // save cloud for visualization purpose
        src = reg_result;

        // Estimate
        reg.setInputCloud (src);
        reg.align (*reg_result);

        //accumulate transformation between each Iteration
        Ti = reg.getFinalTransformation () * Ti;

        //if the difference between this transformation and the previous one
        //is smaller than the threshold, refine the process by reducing
        //the maximal correspondence distance
        if(reg.getMaxCorrespondenceDistance () > 0.2){
            if (fabs((reg.getLastIncrementalTransformation() - prev.sum()))
                    < reg.getTransformationEpsilon())
                reg.setMaxCorrespondenceDistance (
                            reg.getMaxCorrespondenceDistance () - 0.1
                            );
        }
        else if(reg.getMaxCorrespondenceDistance () > 0.002) {
```

```
            if (fabs((reg.getLastIncrementalTransformation() - prev.sum()))
                < reg.getTransformationEpsilon())
                reg.setMaxCorrespondenceDistance (
                            reg.getMaxCorrespondenceDistance () - 0.001
                            );
        }

        prev = reg.getLastIncrementalTransformation ();
    }
}
else {

    reg.setInputCloud (tgt);
    reg.setInputTarget (src);

    // Run the same optimization in a loop and visualize the results
    reg_result = tgt;
    reg.setMaximumIterations (2);
    for (int i = 0; i < MAX_ITER; ++i)
    {

        // save cloud for visualization purpose
        tgt = reg_result;

        // Estimate
        reg.setInputCloud (tgt);
        reg.align (*reg_result);

        //accumulate transformation between each Iteration
        Ti = reg.getFinalTransformation () * Ti;

        //if the difference between this transformation and the previous one
        //is smaller than the threshold, refine the process by reducing
        //the maximal correspondence distance
        if(reg.getMaxCorrespondenceDistance () > 0.2){
            if (fabs((reg.getLastIncrementalTransformation() - prev.sum()))
                < reg.getTransformationEpsilon())
                reg.setMaxCorrespondenceDistance (
                            reg.getMaxCorrespondenceDistance () - 0.1
                            );
        }
        else if(reg.getMaxCorrespondenceDistance () > 0.002) {
            if (fabs((reg.getLastIncrementalTransformation() - prev.sum()))
                < reg.getTransformationEpsilon())
                reg.setMaxCorrespondenceDistance (
                            reg.getMaxCorrespondenceDistance () - 0.001
                            );
        }

        prev = reg.getLastIncrementalTransformation ();
    }
    Ti = Ti.inverse();
}
```

```
    return Ti*TfEst;

}
```

A.2 Source Code for Algorithms 4 to 7

```
void transformTree(OcTree *tree, Eigen::Matrix4f& transform) {

    double treeRes = tree->getResolution();
    OcTree* transformed = new OcTree(treeRes);

    //build inverse transform
    Eigen::Matrix3f rotation;
    Eigen::Matrix3f invRotation;
    Eigen::Matrix4f invTransform;
    rotation << transform(0,0), transform(0,1), transform(0,2),
            transform(1,0), transform(1,1), transform(1,2),
            transform(2,0), transform(0,2), transform(2,2);
    invRotation = rotation.transpose();
    invTransform <<
            invRotation(0,0), invRotation(0,1), invRotation(0,2), -transform(0,3),
            invRotation(1,0), invRotation(1,1), invRotation(1,2), -transform(1,3),
            invRotation(2,0), invRotation(2,1), invRotation(2,2), -transform(2,3),
            0, 0, 0, 1;

    //size in each coordinate of each axis.
    double minX, maxX, minY, maxY, minZ, maxZ;

    //get the minimum and max in y so we can step along each row
    tree->getMetricMin(minX,minY,minZ);
    tree->getMetricMax(maxX,maxY,maxZ);

    //get a Look up table
    OcTreeLUT ocTreeLUT(treeRes);

    //allocate a vector of points
    std::vector<point3d> points;

    //make 8 points to make a map bounding box, performing the tf on them
    //to get the range of values in the transformed map
    points.push_back(point3d(maxX,minY,minZ));
    points.push_back(point3d(minX,minY,minZ));
    points.push_back(point3d(minX,maxY,minZ));
    points.push_back(point3d(maxX,maxY,minZ));
    points.push_back(point3d(maxX,minY,maxZ));
    points.push_back(point3d(minX,minY,maxZ));
    points.push_back(point3d(minX,maxY,maxZ));
    points.push_back(point3d(maxX,maxY,maxZ));

    //transform the points
    for(unsigned i = 0; i<points.size(); i++){
        Eigen::Vector4f point(points[i].x(),points[i].y(),points[i].z(),1);
        point = transform * point;
        points[i] = point3d(point(0),point(1),point(2));
```

```
}

//go through tf'd points to get a new bbox
minX = points[0].x(); minY = points[0].y(); minZ = points[0].z();
maxX = points[0].x(); maxY = points[0].y(); maxZ = points[0].z();
for(unsigned i=0; i<points.size(); i++){
    minX = (points[i].x() < minX) ? points[i].x() : minX;
    minY = (points[i].y() < minY) ? points[i].y() : minY;
    minZ = (points[i].z() < minZ) ? points[i].z() : minZ;
    maxX = (points[i].x() > maxX) ? points[i].x() : maxX;
    maxY = (points[i].y() > maxY) ? points[i].y() : maxY;
    maxZ = (points[i].z() > maxZ) ? points[i].z() : maxZ;
}

//go through the possible destination voxels on a row by row basis
//and calculate occupancy from source voxels with inverse tf

for(double z = minZ -treeRes/2; z<(maxZ + treeRes/2); z+=treeRes) {
    for(double y = minY -treeRes/2; y<(maxY + treeRes/2); y+=treeRes) {
        for(double x = minX -treeRes/2; x<(maxX + treeRes/2); x+=treeRes) {
            OcTreeKey destVoxel = transformed->coordToKey(
                        point3d(x,y,z)
                        );
            Eigen::Vector4f point(x,y,z,1);
            point = invTransform * point;
            point3d sourcePoint = point3d(point(0),point(1),point(2));
            OcTreeKey sourceVoxel = tree->coordToKey(sourcePoint);
            point3d nn = tree->keyToCoord(sourceVoxel);

            //use nearest neighbour to set new occupancy
            //in the transformed map
            OcTreeNode *oldNode = tree->search(sourceVoxel);

            //Occupancies to interpolate between
            double c000, c001, c010, c011, c100, c101, c110,
                    c111, c00, c01, c10, c11, c0, c1;
            double xd, yd, zd;

            //differences in each direction between next closest voxel
            xd = (sourcePoint.x() - nn.x())/treeRes;
            yd = (sourcePoint.y() - nn.y())/treeRes;
            zd = (sourcePoint.z() - nn.z())/treeRes;

            if(oldNode != NULL){

                c000 = oldNode->getOccupancy();
                OcTreeNode *node;

                //c001
                if((node = tree->search(
                        point3d(nn.x(),nn.y(),nn.z() +
                                getSign(zd)*treeRes)))
                        != NULL) {
                    c001 = node->getOccupancy();
```

```
} else
    c001 = 0;

//c010
if((node = tree->search(
        point3d(nn.x(),
                nn.y() + getSign(yd)*treeRes,
                nn.z())))
        != NULL) {
    c010 = node->getOccupancy();
} else
    c010 = 0;

//c011
if((node = tree->search(
        point3d(nn.x(),
                nn.y() + getSign(yd)*treeRes,
                nn.z() + getSign(zd)*treeRes )))
        != NULL) {
    c011 = node->getOccupancy();
} else
    c011 = 0;

//c100
if((node = tree->search(
        point3d(nn.x()+ getSign(xd)*treeRes,
                nn.y(),
                nn.z())))
        != NULL) {
    c100 = node->getOccupancy();
} else
    c100 = 0;

//c101
if((node = tree->search(
        point3d(nn.x()+ getSign(xd)*treeRes,
                nn.y(),
                nn.z() +getSign(zd)*treeRes)))
        != NULL) {
    c101 = node->getOccupancy();
} else
    c101 = 0;

//c110
if( (node = tree->search(
         point3d(nn.x()+ getSign(xd)*treeRes,
                 nn.y() +getSign(yd)*treeRes,
                 nn.z())))
        !=NULL) {
    c110 = node->getOccupancy();
} else
    c110 = 0;

//c111
```

```
                        if( (node = tree->search(
                                point3d(nn.x()+ getSign(xd)*treeRes,
                                        nn.y() +getSign(yd)*treeRes,
                                        nn.z()+getSign(zd)*treeRes)))
                                != NULL) {
                            c111 = node->getOccupancy();
                        } else
                            c111 = 0;

                        //Interpolate in x
                        c00 = (1-fabs(xd))*c000 + fabs(xd)*c100;
                        c10 = (1-fabs(xd))*c010 + fabs(xd)*c110;
                        c01 = (1-fabs(xd))*c001 + fabs(xd)*c101;
                        c11 = (1-fabs(xd))*c011 + fabs(xd)*c111;

                        //interpolate in y
                        c0 = (1-fabs(yd))*c00 + fabs(yd)*c10;
                        c1 = (1-fabs(yd))*c01 + fabs(yd)*c11;

                        //now let's assign the new node value
                        OcTreeNode *newNode = transformed->updateNode(
                                    destVoxel,true
                                    );
                        newNode->setLogOdds(
                                    logodds((1-fabs(zd))*c0 + fabs(zd)*c1)
                                    );
                    }
                }
            }
        }

        tree->swapContent(*transformed);

        delete transformed;
    }
```

A.3 Source Code for Algorithm 8

```
#include <octomap/octomap.h>
#include <octomap/OcTreeLUT.h>
#include <fstream>
#include <iostream>
#include <string.h>
#include <stdlib.h>
#include <list>
#include <cmath>

using std::cout;
using std::endl;
using namespace octomap;
using namespace octomath;


void expandLevel(std::vector<OcTreeNode *> *nodePtrs) {
```

```
    unsigned size = nodePtrs->size();

    for(unsigned i = 0; i<size; i++) {
        OcTreeNode *parent = nodePtrs->front();
        parent->expandNode();
        nodePtrs->erase(nodePtrs->begin());
        for(unsigned j=0; j<8; j++) {
            nodePtrs->push_back(parent->getChild(j));
        }
    }
}

unsigned expandNodeMultiLevel(OcTree *tree, OcTreeNode *node, unsigned currentDepth
    if(currentDepth == (int) tree->getTreeDepth()) {
        return 0;
    }

    int levelsCounter = 0;
    std::vector<OcTreeNode *> nodePtrs;
    nodePtrs.push_back(node);

    for(unsigned i = 0; i<levels; i ++) {
        if(currentDepth == (int) tree->getTreeDepth()) {
            return levelsCounter;
        }
        expandLevel(&nodePtrs);
        levelsCounter++;
        currentDepth++;
    }

    return levelsCounter;

}


/*
* Searches for a node at a given point
* and returns the depth in the tree of that node
* Assumes you have called search before and
* know its actually there.
* Returns -1 if it couldn't find anything
*/

int getNodeDepth(OcTree* tree, point3d& point, OcTreeNode* node) {
    for(int depth = tree->getTreeDepth(); depth > 1; depth--) {
        if(tree->search(point,depth) == node)
            return depth;
    }

    return -1;
}

int main(int argc, char** argv) {
```

```
std::string filename1 = std::string(argv[1]);
std::string filename2 = std::string(argv[2]);
std::string outputFilename = std::string(argv[3]);

cout << "\nReading octree files...\n";

double roll, pitch, yaw;

point3d translation;
if(argc == 7 || argc == 10) {
    translation = point3d(atof(argv[4]),atof(argv[5]),atof(argv[6]));
}
if(argc == 10) {
    roll = atof(argv[7]);
    pitch = atof(argv[8]);
    yaw = atof(argv[9]);
}
else {
    roll = 0;
    pitch = 0;
    yaw = 0;
}

Pose6D pose(translation.x(),
            translation.y(),
            translation.z(),
            roll,pitch,yaw);
//build a transform matrix
Eigen::Matrix4f transform;
std::vector<double> coeffs;
pose.rot().toRotMatrix(coeffs);

transform << coeffs[0], coeffs[1], coeffs[2], translation.x(),
        coeffs[3], coeffs[4], coeffs[5], translation.y(),
        coeffs[6], coeffs[7], coeffs[8] , translation.z(),
        0, 0, 0, 1;

OcTree* tree1 = dynamic_cast<OcTree*>(OcTree::read(filename1));
OcTree* tree2 = dynamic_cast<OcTree*>(OcTree::read(filename2));

//initial TF Matrix
std::cout << transform << std::endl;

cout << "Registering map to Improve TF Estimate" << endl << endl;

//make point clouds from each map
pcl::PointCloud<pcl::PointXYZ> tree1Points;
tree2PointCloud(tree1, tree1Points);
pcl::PointCloud<pcl::PointXYZ> tree2Points;
tree2PointCloud(tree2, tree2Points);

//get refined matrix
transform = getICPTransformation(tree1Points,tree2Points,transform);
```

```
if(roll != 0 ||
        pitch != 0 ||
        yaw != 0 ||
        translation.x() != 0 ||
        translation.y() != 0 ||
        translation.z() != 0) {
    transformTree(tree2,transform);
}


//begin merging algorithm
//traverse nodes in tree 2 to add them to tree 1
for (OcTree::leaf_iterator it = tree2->begin_leafs();
     it != tree2->end_leafs();
     ++it)
{
    if(tree2->isNodeOccupied(*it)){
        it->setLogOdds(logodds(0.6));
    }

    //find if the current node maps a point in map 1
    OcTreeNode *nodeIn1 = tree1->search(it.getCoordinate());
    OcTreeKey nodeKey = tree1->coordToKey(it.getCoordinate());
    point3d point = it.getCoordinate();
    if(nodeIn1 != NULL) {
        //get the depth of already mapped space in 1 and compare to 2
        int depthIn1 = getNodeDepth(tree1, point, nodeIn1);
        if(depthIn1 != -1) {
            int depthDiff = it.getDepth() - depthIn1;
            if(depthDiff == 0) {
                tree1->updateNode(nodeKey,it->getLogOdds());
            }
            else if(depthDiff > 0) {
                //map 2 is lower depth, add children to 1 if it's not a leaf
                for(int i=0; i<depthDiff; i++) {
                    if(depthIn1 == (int) tree1->getTreeDepth()) {
                        break;
                    }
                    nodeIn1->expandNode();
                    nodeKey = tree1->coordToKey(point);
                    depthIn1++;
                }
                nodeIn1->setLogOdds(
                            logodds(nodeIn1->getOccupancy()+
                                    it->getOccupancy()));
            }
            else if(depthDiff < 0) {
                //map 1 is lower depth, add children to 2
                expandNodeMultiLevel(tree2,tree2->search(point),
                                     it.getDepth(), abs(depthDiff));
                //now that we are expanded the other
                //expanded nodes will be handled in subsequent loop
                //iterations
            }
```

```
            }
        } else {
            OcTreeNode *newNode = tree1->updateNode(point,true);
            newNode->setLogOdds(it->getLogOdds());
        }
    }

    std::cout << "Compressing merged result\n";
    tree1->prune();
    //tree1 is now the compressed merged map

    //write merged map to file
    tree1->write(outputFilename);

    delete tree1;
    delete tree2;
}
```

Curriculum Vitae

Curriculum Vitae

James Jessup joined the Canadian Forces in 2008 as a Naval Combat Systems Engineering Officer (NCSEng). After completing his B.Eng. in Electrical Engineering at the Royal Military College of Canada in 2012, he was posted Kingston, Ontario, to complete his Master's of Applied Science on a scholarship from DRDC.