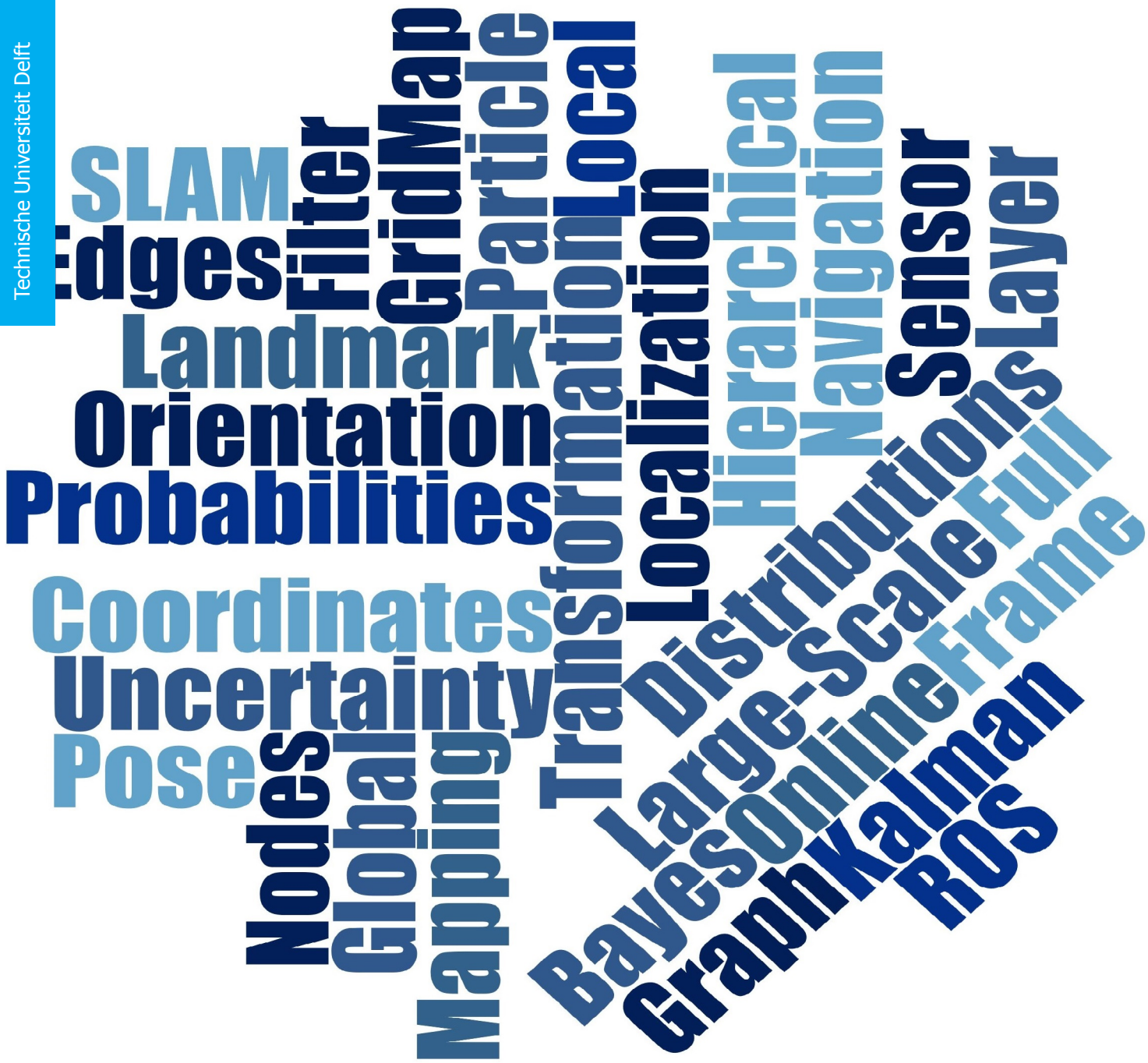
Hierarchical Mapping

A Rao-Blackwellized Particle Filter based hierarchical SLAM framework

E. Giannopoulos

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Hierarchical Mapping

A Rao-Blackwellized Particle Filter based hierarchical SLAM framework

by

E. Giannopoulos

in partial fulfillment of the requirements for the degree of

Master of Science
in Embedded Systems

at the Delft University of Technology,
to be defended publicly on Wednesday November 29, 2017 at 09:00 AM.
MSc. thesis number: CE-MS-2017-19

Supervisors:	dr. ir. A. J. van Genderen,	D EEMCS CE
	dr. ir. C. J. M. Verhoeven,	UHD EEMCS ELCA
	dr. ir. E. A. Hakkennes,	Senior Consultant TECHNOLUTION
	ir. D. Chronopoulos,	Software Developer at Robot Care Systems
Thesis committee:	dr. ir. S. D. Cotofana,	UHD EEMCS CE
	dr. ir. A. J. van Genderen,	D EEMCS CE
	dr. ir. C. J. M. Verhoeven,	UHD EEMCS ELCA
	dr. ir. E. A. Hakkennes,	Senior Consultant TECHNOLUTION
	ir. D. Chronopoulos,	Software Developer at Robot Care Systems

This thesis is confidential and cannot be made public until November 29, 2019.

TU Delft Delft University of Technology

ROBOT CARE SYSTEMS RCS

Preface

This document concludes the work that I conducted in order to obtain my Master of Science in Embedded Systems. Conducting my master thesis was a really challenging process, full of obstacles and difficulties that I had to face, but in the same time full of creative and memorable moments. During the conductance of my thesis I had to deal with a totally new field to me, I had to adapt and learn quickly and in the same time I had to contribute with what I had already learned and obtained during my master courses. As the journey of attending the Embedded Systems master's programme comes to an end, I feel that I gained experience academically, professionally and socially, exploring new aspects in every part.

I would like to thank many people that contributed in their own way towards the completion of this work and the completion of my studies in general. First of all I would like to thank RCS for giving me the opportunity to conduct my thesis in their offices. Especially, I would like to thank my supervisor ir. Dimitrios Chronopoulos for his everyday guidance throughout this process and for his trust on me. I would also like to thank Dr. ir. Alejandro Lopez, Dr. ir. Aswin Chandarr and my fellow student Thomas Horstink for their support during the project. In addition, I would like to thank my university supervisors Dr. ir. Arjan van Genderen, Dr. ir. Edwin Hakkennes and Dr. ir. Chris Verhoeven for accepting me as their student and for giving me their helpful feedback throughout the whole process. Furthermore, I would like to thank my family for supporting my decision on studying abroad and giving me this opportunity. Finally, I would like to thank my fellow student and friend Giannis with whom we followed together the same master programme and helped each other, my roommates Hary, Jordan, Pavlos and Vangelis for encouranging me daily, as well as my friend Aimilia for always being there for me.

E. Giannopoulos
Delft, November 2017

Abstract

Mobile robots need to be fully autonomous in order to perform their tasks inside their environment. To do that, robots need to have an understanding of their environment, so that they can successfully localize and navigate themselves within it. The understanding of the robots' environment is created by solving the SLAM (Simultaneous Localization And Mapping) problem. The SLAM problem is usually approached using probabilistic laws; both the robot's path and the environment are estimated and represented by probability distributions.

The estimate of the environment is incorporated into a map, which could either be one global map or a network of smaller connected local maps. In small areas where the sensors' errors and inaccuracies remain low, the creation of one global map is preferred. However, as the area grows larger, the global map becomes inaccurate due to accumulated errors. Therefore the strategy of creating a network of local maps is employed. Even though this strategy produces low error results, there is still a limit beyond which it does not scale. Specifically, as the mapping area grows even wider, the number of local maps increases which in turn causes the amount of required memory to increase. In addition, when it comes to loop closure or in other words the procedure of estimating whether the robot returned to a previously visited point in the map, the required computational power increases linearly or sometimes quadratically with respect to the number of local maps.

Consequently, SLAM algorithms suffer from scalability problems, especially when the mapping area becomes too large. Since robots are real-world devices, mapping solutions should be efficient and applicable to the robot's resources, even when these resources are limited.

In this thesis we propose a framework which addresses this scalability issue, aiming to reduce the computational needs and memory usage during the SLAM procedure. Additionally, our framework intends to facilitate loop closure, which can become troublesome in large environments.

Our framework uses multiple sensors and creates a network of hierarchically structured local maps. In order to assess our framework, we compare it to a technique which produces a network of non-hierarchically structured local maps. Our experiments show that in large areas our framework works ideally; the speedup is encouraging and the RAM memory overhead is negligible or conditionally lower.

Contents

1

Introduction

The advancement of technology has boosted the robotic sector. Nowadays robotic applications are almost everywhere, steadily becoming an integral part of our life. Examples of robotic applications can be found in different sectors such as military, health-care or transportation. Recently, mobile robots have started becoming popular, with smart cars, drones or smart vacuum cleaners being prominent examples of mobile robots.

Mobile robots need to perform their designated tasks inside their environment. Ideally, mobile robots need to be fully autonomous, having an understanding of their environment and navigating themselves inside it. The understanding of the robots' environment is created by solving the SLAM (Simultaneous Localization And Mapping) problem.

1.1. Simultaneous Localization And Mapping

SLAM describes the problem of estimating the robot's position and the robot's environment. SLAM is often referred to as a "chicken or egg" problem; the robot need to know its position in order to create the map, but in order to know its position, it needs a map.

The robot uses its range sensors and its motion sensors in order to solve the SLAM problem. Range sensors are used so that the robot can locate objects around it. Then the robot needs to express the objects' position in respect to its own coordinate frame. Therefore, the robot needs to know its own location. This information comes from the motion sensors, which are also called odometry. Odometry is used by the robot in order to keep track of its own movement [1].

However, both the range sensors and odometry are noisy. Thus, there is always uncertainty regarding the map and the robot's position in it. This yields that both the robot's position and the map are approximated, trying to keep the errors as low as possible. Specifically, SLAM is solved using probabilistic laws and the robot's position and the map are estimated and represented by probability distributions. This estimate is done recursively using a filter. At each time step the robot's pose is corrected based on the surroundings, while the surroundings are updated based on the new robot pose [2].

The ultimate result of the SLAM process is a map. The map represents the robot's perception of the world and is used by the robot in order to autonomously navigate itself. SLAM algorithms produce either one global map or a network of smaller connected local maps. Algorithms that create one global map are usually employed in small areas, where the sensor errors and inaccuracies remain low. However, as the area grows wider the global map gets inaccurate due to accumulated errors. Therefore, in wider areas SLAM algorithms that produce a network of connected local maps are preferred since the local maps are small enough in order to be as accurate as possible. Even though, this technique produces low error results, there is a limit beyond which, this technique does not scale.

The scalability problem lies in the fact that as the area grows pretty large the amount of memory and computational power gets too demanding. Specifically, as the mapping area grows even larger, the amount of the local maps increases. Thus, the amount of required memory grows linearly to the number of local maps. In addition, while addressing loop closure or in other words the procedure of estimating whether the robot returned to a previously visited point in the map, the number of local

maps affects the performance remarkably. Specifically, the required computational power in order to perform loop closure increases linearly and in some cases quadratically to the number of local maps.

1.2. Thesis Statement

In this thesis we propose a mapping framework which mainly addresses this scalability issue. Our framework intends to be ideal for large-scale areas by decreasing the mean computational power and especially the computational power which is required for performing loop closure. In addition, it introduces a mechanism which aims to decrease the memory consumption. Furthermore, it tries to facilitate the loop closure procedure which can become troublesome in large environments. Finally, our framework produces a network of maps where navigation can be optimized, gaining speedup during the process.

The key insight behind our framework is the use of multiple sensors and the production of an hierarchically structured network of local maps. The implementation of our framework is generic, which means that it can be used for an arbitrary number of sensors and it can be configured per one's implementation.

1.3. Thesis Outline

The rest of this thesis report is structured as follows:

- Chapter 2: The basic SLAM notions and the related work is discussed.
- Chapter 3: The theoretical analysis of our framework is explained.
- Chapter 4: The implementation of our framework is presented.
- Chapter 5: The results of our experiments are presented.
- Chapter 6: We conclude on our framework.
- Chapter 7: Future recommendations about optimizing the framework are discussed.
- Appendices: Useful algorithms for our framework are presented.

2

Related Work

In this chapter we present all the required information about the SLAM problem in order to provide the reader with the appropriate background, before describing the theory and the implementation of our mapping scheme. Firstly, we give the definition of the SLAM problem. Then we present the most used methods for metric SLAM, as well as hybrid metric topological approaches to SLAM. Finally, we discuss about Range-Only SLAM.

2.1. SLAM Problem Definition

As we discussed in Chapter 1, the SLAM describes the problem of a robot creating the map of its unknown environment, while in the same time keeping track of its own pose inside the map. Therefore, the robot needs to estimate the map (m), which in practice consists of landmarks[1] observed and placed at specific coordinates. In addition, the robot needs to estimate its path ($x_{1:t}$). The robot's path is the set of the robot's states/positions/poses/locations $x_1, x_2, x_3 ... x_t$, which are estimated at each time step. The aforementioned quantities are estimated, given a set of observations $z_{1:t}$, which are obtained by the robot's range sensors (e.g. laser scanner, camera, ultrasonic sensor etc). Additionally, the set of the input controls $u_{1:t}$ is taken into account. The input controls is practically odometry information, which is derived by the robot's motion sensors. [3]

However, since the robot's sensors are noisy, there is uncertainty regarding their measurements. Therefore, the SLAM problem is usually solved using probabilistic laws and the robot's pose and the map are estimated and represented by probability distributions.[3]

There exist two distinguishable cases of the SLAM problem; the online SLAM problem and the full SLAM problem. The online SLAM problem is described by Equation 2.1. It involves the calculation of the posterior over the momentary pose and the map. It is called online SLAM, since it involves the calculation of the variables that persist at time t. The online SLAM posterior is estimated in an incremental way by discarding the past observations and controls once they are processed. Figure 2.1 depicts the graph model of the online SLAM problem [3].

$$p(x_t, m | z_{1:t}, u_{1:t}) \tag{2.1}$$

The full SLAM problem is described by Equation 2.2. Full SLAM problem involves the calculation of the posterior over the robot path and the map. In this case past measurements are not discarded and at each time step an estimate for the whole robot path is performed. Figure 2.2 depicts the graph model of the full SLAM problem [2].

$$p(x_{1:t}, m | z_{1:t}, u_{1:t}) \tag{2.2}$$

The different structure between the online SLAM and the full SLAM posterior causes interesting changes in the dependencies of the SLAM variables. Various techniques have been explored by academics to address both SLAM problems, each handling those dependencies in a different way [2].

[1] Landmarks, which are also referred to as features, are the objects that the robot locates around it.

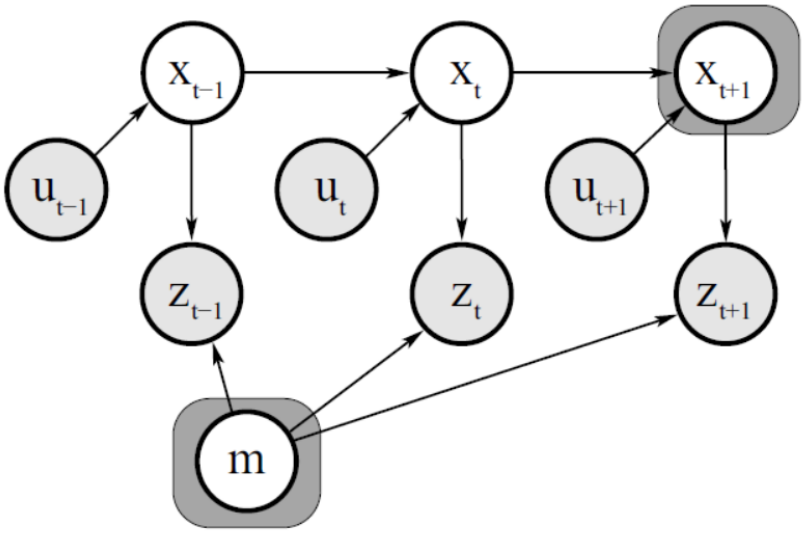


Figure 2.1: The graph model for online SLAM. The x variable stands for the robot pose, the z variable stands for the observations, the u variable stands for the input control and the m variable is the map. The subscript at x, z and u represents the time step. Only the last pose and the map are estimated, as shown by the gray-shaded squares [2].

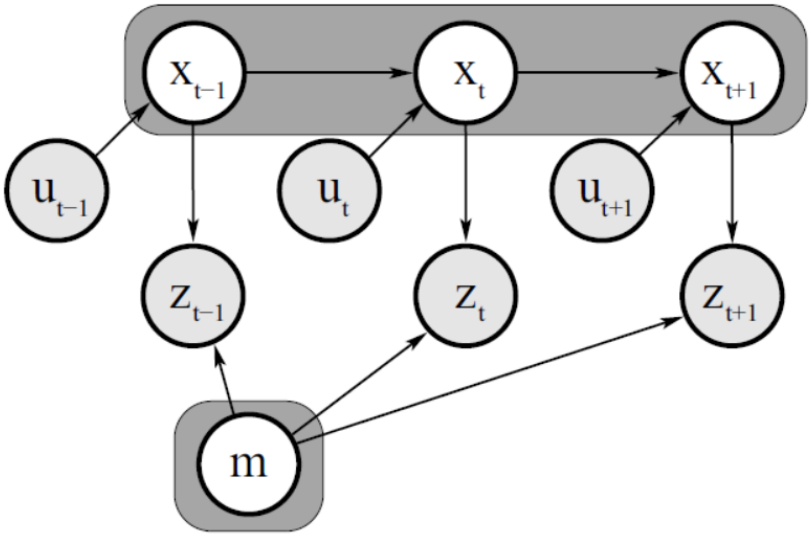


Figure 2.2: The graph model for full SLAM. The x variable stands for the robot pose, the z variable stands for the observations, the u variable stands for the input control and the m variable is the map. The subscript at x, z and u represents the time step. The whole robot path and the map are estimated at each time step, as shown by the gray-shaded squares [2].

One important aspect in solving both the online SLAM problem and the full SLAM problem is the pose correction. The motion sensor noise corrupts the estimate of the robot pose, whereas both the range sensor noise and the motion sensor noise corrupts the estimate of the landmarks' location. Therefore, the robot pose needs to be estimated and corrected constantly in order to produce as accurate results as possible. Figure 2.3 depicts an example where the robot pose is not corrected and as a result the map is inaccurate. On the contrary, Figure 2.4 depicts the resulting map when pose correction takes place. The result is much more accurate than before and the error on the map remains low [1].

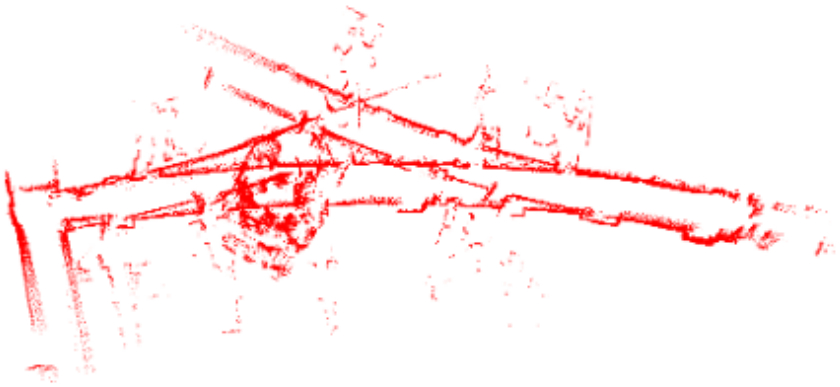


Figure 2.3: The resulted map of a process where no pose correction takes place. In this example the robot uses its range measurements in order to place the landmarks in the map, but it doesn't correct the pose estimate. The resulted map is inaccurate, so we can clearly notice the importance of correcting the robot pose [1].

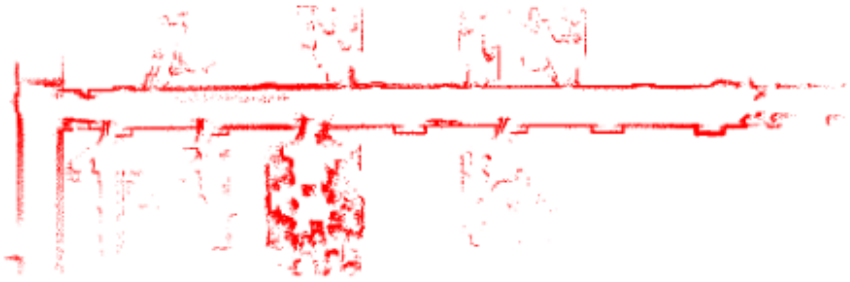


Figure 2.4: The resulted map of a process where pose correction takes place. In this example the robot corrects its pose at each time step of the SLAM process [1].

When the robot corrects its pose, it takes into account the current observation and the already constructed map. The robot uses the constructed map in order to calculate the expected observation. In order to calculate the expected observation it needs to know to which landmarks the current observation corresponds. Then it uses the difference between the expected observation and the current observation as the metric to correct the pose.

It can be inferred that the quality of the robot pose correction heavily depends on the environment. Environments that place unique characteristics have a better effect on the robot pose correction, since the robot is able to efficiently correspond the current observation to the mapped landmarks and calculate the expected observation [2].

2.2. Metric SLAM

The ultimate goal of solving the SLAM problem is to produce a map. The map keeps the robot's perception of the world, namely the locations of the objects around it. In addition, a map can provide with information of the shape or size of objects. These maps are often referred to, as metric maps. The SLAM techniques that involve the production of metric maps fall under the category of metric SLAM.

One common type of a metric map is a landmark map. A landmark map holds the position of all estimated landmarks. The position of the landmarks is expressed by a mean value vector and a

covariance matrix. This means that for each landmark there is an associated uncertainty for its true coordinates. This uncertainty is derived by the covariance matrix and it is visualized as an ellipse around the mean location of the landmark. Figure 2.5 depicts such an example.

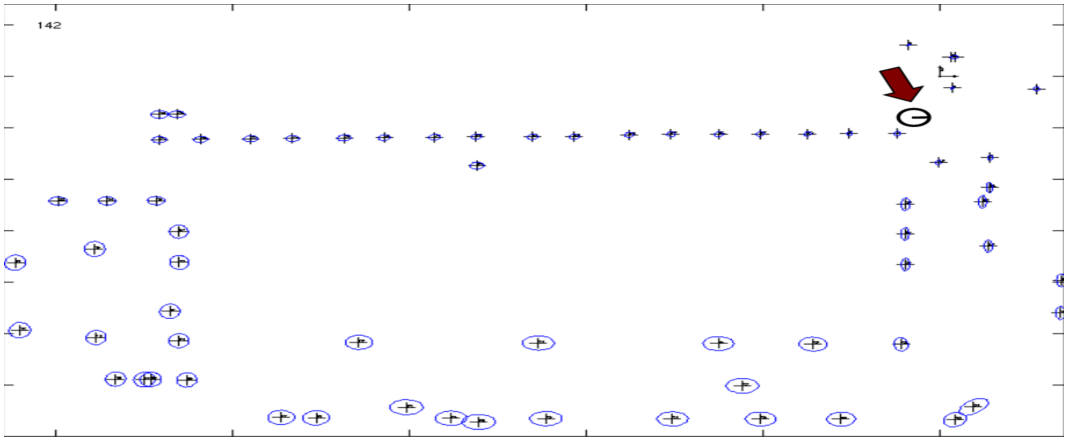


Figure 2.5: A map of landmarks: The red arrow represents the robot's location, the crosses represent the mean value of each landmark's position and the ellipses represent the uncertainty associated to each landmark position. The landmarks that are closer to the current robot's location have smaller uncertainty [3].

Another type of metric map is the occupancy grid map. This map consists of grid cells which represent a square plane in space. Each cell takes a value which corresponds to the cell state; free, occupied or unknown. Free cells represent the area where the robot can move, occupied cells represent area where landmarks have been placed, while unknown cells represent area which is still unexplored and unknown to the robot. The size of each cell defines the level of detail in the produced map, or in other words the map's resolution. Figure 2.6 depicts an example of an occupancy grid map [3].

Figure 2.6: An occupancy grid map: the black cells represent detected objects, the white cells represent free space and the grey cells represent unknown/unexplored area [4].

Landmark maps and grid maps are the two extreme cases of types of maps. Landmark maps are simple, whereas grid maps provide more details about the environment. The choice of the type of map depends on the application; how much detail of the environment does the robot needs to know?

2.2.1. EKF SLAM

The older known solution to the SLAM problem is the Extended Kalman Filter. Specifically, the EKF is used in order to estimate the *online* SLAM posterior. The state of the robot and the landmarks' locations are estimated simultaneously. Both the robot pose and the location of each landmark follow Gaussian estimates and they are represented by a mean value and a corresponding covariance. The mean value is kept by the state vector which is represented as μ and the covariance is kept by the covariance matrix which is represented as Σ. The state vector and the corresponding covariance matrix are structured as shown in Figure 2.7. It should be noted that the structure of the covariance matrix implies dependency among all the estimated variables; the robot's position and the landmarks' position [2].

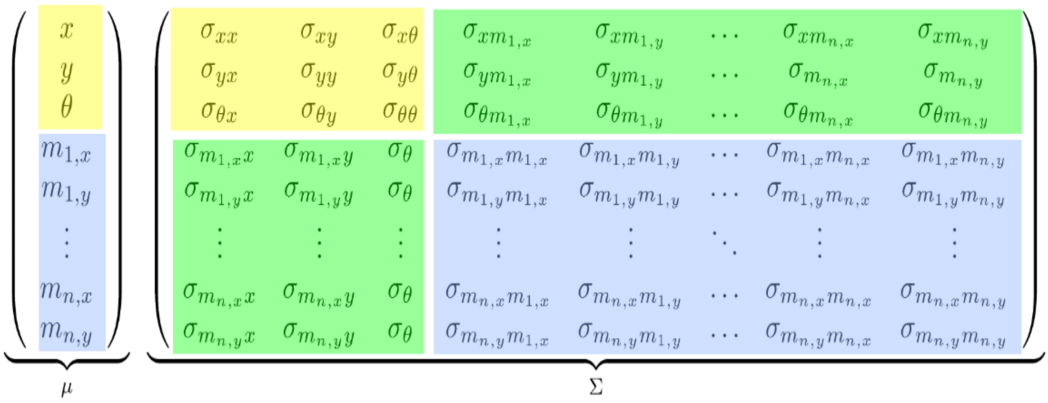


Figure 2.7: State vector and covariance matrix of the EKF. The state vector holds the mean value of all the estimated variables (robot pose and landmarks' position), while the covariance matrix keeps the covariance of each calculated variable [3].

There exist two models that are of great importance within the SLAM process. These models are the motion model (Equation 2.3) and the observation model (Equation 2.4). The motion model describes how the robot gets to the current state given the previous state and the control, whereas the observation model outputs the expected range measurement given the state of the robot [2].

$$x_t = g(x_{t-1}, u_t) + \varepsilon_t \tag{2.3}$$

$$z_t = h(x_t) + \delta_t \tag{2.4}$$

The EKF technique uses both of the aforementioned models in order to solve the SLAM problem. Since both of these models are not linear, a linearization is performed at the operating point using a first order Taylor expansion. The EKF procedure consists of two steps; the prediction step and the correction step.

During the prediction step, the state of the robot (position and orientation) is predicted using the motion model, given the previous state and the odometry. The corresponding values in the state vector (x, y, θ) are updated, alongside the part of the covariance matrix that involves correlation with the pose variables.

In the correction step, given the predicted robot pose, the observation model is applied for each observed landmark. The difference between the expected observation of the landmark (derived by the observation model) and the actual observation of the landmark (obtained by the sensors) is the metric for correcting the predicted pose and for updating the landmark's location. During the correction step, the entire state vector and the entire covariance matrix is updated.

It should also be noted that during the correction step, an incremental maximum likelihood estimator is applied in order to create correspondences among the current observation and the already stored landmarks. In other words, the robot has to match what it currently observes with what it has stored in memory. If the current observation, matches a stored landmark, then that landmark is updated. If the current observation doesn't match any stored landmark, then a new landmark is initialized [2].

1: **Algorithm EKF_SLAM_known_correspondences**$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, c_t)$:

2: $F_x = \begin{pmatrix} 1 & 0 & 0 & \underbrace{0 \cdots 0}_{3N} \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \end{pmatrix}$ //point to pose elements of state vector

3: $\bar{\mu}_t = \mu_{t-1} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} - \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}$ //predict pose and //update state vector

4: $G_t = I + F_x^T \begin{pmatrix} 0 & 0 & -\frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & 0 \end{pmatrix}$

5: $\bar{\Sigma}_t = G_t\,\Sigma_{t-1}\,G_t^T + F_x^T\,R_t\,F_x$ //update covariance matrix based on the predicted pose

6: $Q_t = \begin{pmatrix} \sigma_r^2 & 0 & 0 \\ 0 & \sigma_\phi^2 & 0 \\ 0 & 0 & \sigma_s^2 \end{pmatrix}$ //measurement covariance matrix, //representing the noise on measurements

7: *for all observed features* $z_t^i = (r_t^i \;\; \phi_t^i \;\; s_t^i)^T$ *do*

8: $j = c_t^i$

9: *if landmark* j *never seen before*

10: $\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \\ \bar{\mu}_{j,s} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \\ s_t^i \end{pmatrix} + \begin{pmatrix} r_t^i\cos(\phi_t^i + \bar{\mu}_{t,\theta}) \\ r_t^i\sin(\phi_t^i + \bar{\mu}_{t,\theta}) \\ 0 \end{pmatrix}$ //initialize landmark location

11: *endif*

12: $\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}$

13: $q = \delta^T\delta$ //calculate expected observation

14: $\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \mathrm{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \\ \bar{\mu}_{j,s} \end{pmatrix}$

15: $F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & \underbrace{0\cdots0}_{3j-3} & 0 & 0 & 0 & \underbrace{0\cdots0}_{3N-3j} \\ 0 & 1 & 0 & & 0 & 0 & 0 & \\ 0 & 0 & 1 & & 0 & 0 & 0 & \\ 0 & 0 & 0 & & 1 & 0 & 0 & \\ 0 & 0 & 0 & & 0 & 1 & 0 & \\ 0 & 0 & 0 & & 0 & 0 & 1 & \end{pmatrix}$ //point to the pose //elements and current //landmark elements of //state vector

16: $H_t^i = \frac{1}{q}\begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & \sqrt{q}\delta_y & 0 \\ \delta_y & -\delta_x & -q & -\delta_y & +\delta_x & 0 \\ 0 & 0 & 0 & 0 & 0 & q \end{pmatrix} F_{x,j}$ //calculate Jacobian in order to //linearize, at the operating point

17: $K_t^i = \bar{\Sigma}_t\,H_t^{iT}(H_t^i\,\bar{\Sigma}_t\,H_t^{iT} + Q_t)^{-1}$ //calculate Kalman gain

18: $\bar{\mu}_t = \bar{\mu}_t + K_t^i(z_t^i - \hat{z}_t^i)$ //update state vector (pose correction and landmark coordinate update)

19: $\bar{\Sigma}_t = (I - K_t^i\,H_t^i)\,\bar{\Sigma}_t$ //update covariance matrix

20: *endfor*

21: $\mu_t = \bar{\mu}_t$

22: $\Sigma_t = \bar{\Sigma}_t$

23: *return* μ_t, Σ_t

Figure 2.8: Detailed description of the EKF SLAM algorithm with known correspondences. Given the state vector of the previous time step μ_{t-1}, the covariance matrix of the previous time step Σ_{t-1}, the current odometry u_t, the current observation z_t and the current correspondences c_t, compute the new state vector μ_t and the new covariance matrix Σ_t. Lines 2-6 account for the prediction step, whereas lines 7-22 account for the correction step [2].

Figure 2.8 depicts the EKF procedure. In this case known correspondences between the observation and the landmarks is assumed. In other words, the robot knows a-priory which landmarks it observes at each time step, so it does not apply any likelihood estimator.

However, in most of the cases the correspondences are not known and the aforementioned incremental maximum likelihood estimator is applied. The success of the likelihood estimator lies on the nature of the landmarks. Landmarks should be distinct enough in order for the likelihood estimator to perform accurately. Thus, EKF can provide with good quality results in areas that pose distinct

characteristics [2].

Finally, the complexity of the EKF algorithm grows quadratically to the number of observed landmarks, which makes the EKF not suitable for large areas. Thus, EKF mapping is not really used nowadays, except for special cases, where it can work efficiently [2].

2.2.2. RBPF SLAM

Another widely used SLAM method is the Rao-Blackwellized Particle Filter, which addresses the *full SLAM problem*. In this section we present the main idea behind this method.

As previously discussed, Equation 2.2 involves estimating the whole robot path alongside the map. Equation 2.2 can take the following form:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t}) = p(x_{1:t}|z_{1:t}, u_{1:t})p(m|z_{1:t}, u_{1:t}, x_{1:t}) \tag{2.5}$$

The fact that the whole robot path is involved can yield an interesting property. Specifically, if we know the whole robot path, estimating the location of all features in the map can be done independently to each other. In other words, the features are independent to each other, under the condition that the robot's path is known. This property allows for factorizing the problem of estimating the map into N different problems, for each one of the N different landmarks [2]. Under this assumption, Equation 2.5 can be expressed as:

$$\begin{aligned} &p(x_{1:t}|z_{1:t}, u_{1:t})p(m|z_{1:t}, u_{1:t}, x_{1:t}) = \\ &p(x_{1:t}|z_{1:t}, u_{1:t}) \prod_{i=1}^{N} p(m_i|z_{1:t}, u_{1:t}, x_{1:t}) \end{aligned} \tag{2.6}$$

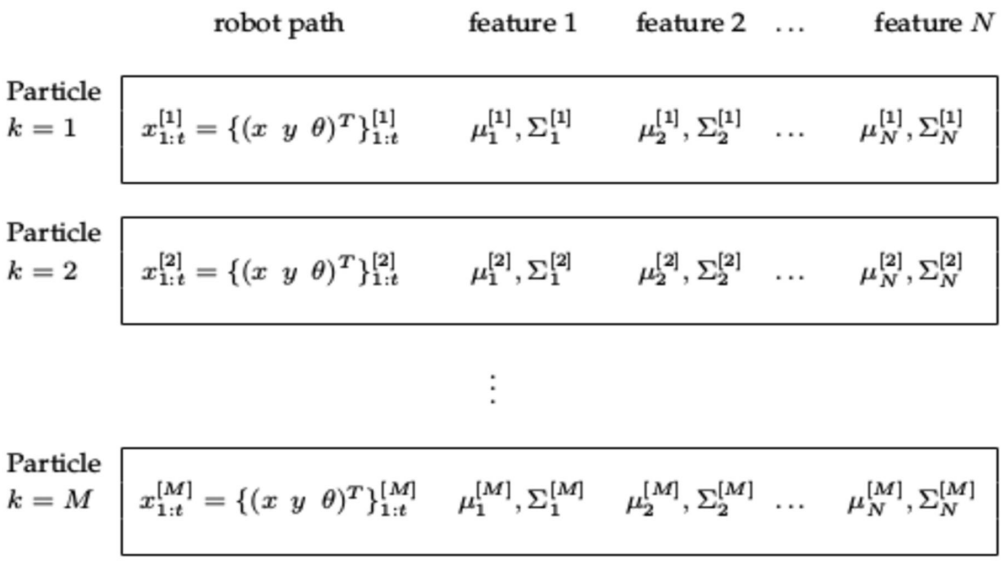


Figure 2.9: Each particle sample maintains its own path hypothesis. Since the map is conditioned on the robot path, a different map of landmarks is assigned per particle sample. In total there exist 1 filter for the robot path (particle filter) and MN filters for all features (EKFs) [2].

This factorization of the full SLAM problem allows for using a version of particle filters known as Rao-Blackwellized Particle Filters. RBPFs use particles in order to represent the posterior over some variables alongside Gaussians that represent the rest of the variables.

1: **Algorithm FastSLAM 1.0_known_correspondence**(z_t, c_t, u_t, Y_{t-1}):

2: *for* $k = 1$ *to* M *do* // *loop over all particles*

3: *retrieve* $\left\langle x_{t-1}^{[k]}, \left\langle \mu_{1,t-1}^{[k]}, \Sigma_{1,t-1}^{[k]} \right\rangle, \ldots, \left\langle \mu_{N,t-1}^{[k]}, \Sigma_{N,t-1}^{[k]} \right\rangle \right\rangle$ *from* Y_{t-1}

4: $x_t^{[k]} \sim p(x_t \mid x_{t-1}^{[k]}, u_t)$ // *sample pose*

5: $j = c_t$ // *observed feature*

6: *if feature* j *never seen before*

7: $\mu_{j,t}^{[k]} = h^{-1}(z_t, x_t^{[k]})$ // *initialize mean*

8: $H = h'(x_t^{[k]}, \mu_{j,t}^{[k]})$ // *calculate Jacobian*

9: $\Sigma_{j,t}^{[k]} = H^{-1}\ Q_t\ (H^{-1})^T$ // *initialize covariance*

10: $w^{[k]} = p_0$ // *default importance weight*

11: *else*

12: $\hat{z} = h(\mu_{j,t-1}^{[k]}, x_t^{[k]})$ // *measurement prediction*

13: $H = h'(x_t^{[k]}, \mu_{j,t-1}^{[k]})$ // *calculate Jacobian*

14: $Q = H\ \Sigma_{j,t-1}^{[k]}\ H^T + Q_t$ // *measurement covariance*

15: $K = \Sigma_{j,t-1}^{[k]}\ H^T\ Q^{-1}$ // *calculate Kalman gain*

16: $\mu_{j,t}^{[k]} = \mu_{j,t-1}^{[k]} + K(z_t - \hat{z})$ // *update mean*

17: $\Sigma_{j,t}^{[k]} = (I - K\ H)\Sigma_{j,t-1}^{[k]}$ // *update covariance*

18: $w^{[k]} = |2\pi Q|^{-\frac{1}{2}}\ \exp\left\{-\frac{1}{2}(z_t - \hat{z}_n)^T\ Q^{-1}\ (z_t - \hat{z}_n)\right\}$ // *importance factor*

19: *endif*

20: *for all other features* $j' \neq j$ *do* // *unobserved features*

21: $\mu_{j',t}^{[k]} = \mu_{j',t-1}^{[k]}$ // *leave unchanged*

22: $\Sigma_{j',t}^{[k]} = \Sigma_{j',t-1}^{[k]}$

23: *endfor*

24: *endfor*

25: $Y_t = \emptyset$ // *initialize new particle set*

26: *do* M *times* // *resample* M *particles*

27: *draw random* k *with probability* $\propto w^{[k]}$ // *resample*

28: *add* $\left\langle x_t^{[k]}, \left\langle \mu_{1,t}^{[k]}, \Sigma_{1,t}^{[k]} \right\rangle, \ldots, \left\langle \mu_N^{[k]}, \Sigma_N^{[k]} \right\rangle \right\rangle$ *to* Y_t

29: *endfor*

30: *return* Y_t

Figure 2.10: The FastSLAM algorithm for known correspondences. Given the current observation z_t, the current correspondences c_t, the current odometry u_t and the previous time step particle set Y_{t-1}, compute the new particle set Y_t. Each individual landmark is stored in memory using a 2x2 EKF, described by the state vector and the covariance matrix [2].

The most known implementation of the RBPF SLAM technique is the family of the FastSLAM algorithms [1], [5]. FastSLAM uses a particle filter[2] to compute the posterior over paths, whereas for each landmark (or feature) in the map it uses a separate low-dimensional 2x2 EKF. The feature estimators are conditioned on the robot path, which means that for each path hypothesis a different estimate of the features' location can exist. Since each particle holds a different path hypothesis, an individual map of landmarks is assigned per particle. In case that M particles are used in order to estimate the robot

[2]Appendix C

path and N landmarks have been discovered in the map, the total number of filters would be $1 + MN$. Figure 2.9 depicts the concept of Fast SLAM or the RBPF based techniques in general. It should also be noted that the aforementioned concept can be generalized for grid maps as well. So, for each particle sample, instead of keeping a set of individual 2x2 EKFs, a separate grid map can be kept [6].

In Figure 2.10 the algorithm for a map of landmarks, where each landmark is represented by an EKF is shown. The Figure describes the FastSLAM 1.0 algorithm for the simple case of known correspondences between observation and landmarks. However, in practice, a likelihood field model is applied per particle in order to account for the correspondences, in the same manner that we described in the EKF subsection. Figure 2.11 depicts the FastSLAM algorithm for grid maps.

1: **Algorithm FastSLAM_occupancy_grids**$(\mathcal{X}_{t-1}, u_t, z_t)$**:**

2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$

3: *for* $k = 1$ *to* M *do*

4: $x_t^{[k]} = \textbf{sample\_motion\_model}(u_t, x_{t-1}^{[k]})$

5: $w_t^{[k]} = \textbf{measurement\_model\_map}(z_t, x_t^{[k]}, m_{t-1}^{[k]})$

5: $m_t^{[k]} = \textbf{updated\_occupancy\_grid}(z_t, x_t^{[k]}, m_{t-1}^{[k]})$

6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[k]}, m_t^{[k]}, w_t^{[k]} \rangle$

7: *endfor*

8: *for* $k = 1$ *to* M *do*

9: *draw* i *with probability* $\propto w_t^{[i]}$

10: *add* $\langle x_t^{[i]}, m_t^{[i]} \rangle$ *to* $\mathcal{X}_t$

11: *endfor*

12: *return* $\mathcal{X}_t$

Figure 2.11: The FastSLAM algorithm for grid maps. In this case a grid map is assigned per particle sample. Lines 3 to 7 depict the main procedure of updating the map and the path posterior. Lines 8 to 11 depict the resampling step [2].

Shortly, the RBPF SLAM procedure for the EKF case is as follows:

1. Given the particle set, for each particle:
   (a) A pose is sampled based on the previous state and the input control.
   (b) The particle's associated map is updated. Specifically, given the new sampled pose each observed landmark's location is updated. In this step, a likelihood field model is applied in order to solve the correspondences problem. This happens in the same manner as we discussed in the EKF subsection.
   (c) The particles' weight is calculated based on the sampled pose and the difference between the expected observation and the real observation. Particles that sampled a pose closer to the real one will get a higher score than those that sampled a pose further away.
2. A new particle set is created by performing resampling. It should be noted that the resampling step doesn't happen at every iteration like in the common particle filter technique. On the contrary, it happens at specific strategic moments, when the variance of the particles' weights increases above a threshold. The choice of the threshold regulates the trade-off execution time - robustness. Equation 2.7 states the metric which is used for deciding on performing resampling.

$$N_{eff} = \frac{1}{\sum_{k=1}^{n}(w_k^2)} \tag{2.7}$$

In the grid map case, the procedure is the same. The only difference is the way that the particle's associated map is updated [7].

The RBPF SLAM technique comes with great advantages compared to previous ones. Firstly, the algorithm's time complexity is logarithmic to the number of features, so it poses a computational advantage over EKF. Secondly, the data association decisions are made per particle basis and the filter maintains posteriors over multiple data associations instead of maintaining the most likely one. This ability to pursue multiple data association provides with robustness when compared to techniques that are based on an incremental maximum likelihood association. Finally, particle filters can deal with non linear motion models. Thus, linear approximations do not exist and in cases where the models are highly non-linear or the pose uncertainty is high, the results are encouraging [2].

2.3. Hybrid Metric-Topological SLAM

Up until this point, the presented SLAM techniques involve the estimation of one global metric map. This map represents the world as it is perceived by the robot's sensors; given global coordinates for each observed object. In addition this global map provides information about the shape and size of the objects.

However, there exists the notion of topological mapping as well. In contrast to metric mapping, topological mapping is not based in the metric properties of the environment like distances or shapes of the objects. Topological mapping is based on the topology of the environment. A topological map is in practice a graph. It consists of nodes and edges. The nodes represent areas that the robot could be at, while the edges represent navigability between the nodes that they connect. An example of a topological map is a metro map, such as the map depicted in Figure 2.12.

Figure 2.12: The metro map of the Rotterdam area. The metro map is a good example of a topological map [8].

Global metric maps tend to become inconsistent in large areas. The accumulation of the odometry error combined with error in the sensors' measurements leads to a deformed global map. A common practice in order to avoid the aforementioned problems and in order to have more consistent and accurate mapping is the hybrid metric-topological mapping (HMT SLAM).

In HMT SLAM the global map is represented by a topological graph. Each node of the topological graph stands for a local metric map that is defined with its own local coordinate frame. In other words the global metric map is segmented into multiple local maps that are connected to each other as the topological graph dictates. The local metric map is chosen to be as small as necessary, so that

mapping strategies can be accurate enough, producing usable results. The global map's inaccuracies are captured by the topological graph. Each edge of the graph, that connects two local maps, holds the information of the uncertainty of how these maps are placed next to each other. Specifically, the edge that connects two local maps, actually carries the transformation between the two local coordinate frames alongside the uncertainty of the transformation. Therefore, in HMT SLAM local areas are mapped with low error, whereas in the global scope there is associated uncertainty of where the local maps are placed.

Extensive work has been done in the field of HMT SLAM and numerous algorithms have been proposed [9], [10], [11], [12], [13]. In this section two HMT SLAM works are presented. These works are characteristic examples of how to treat dependencies between local maps by treating those dependencies in totally different way.

In the work of Blanco et al. [12] an HMT algorithm based on the RBPF SLAM technique is proposed. The robot pose is regarded as a hybrid pose consisting of a tuple between a metric pose and a topological pose. The global map is also defined as a tuple between all local maps and their coordinate frame transformations. The robot's paths inside the local maps are assumed to be conditionally independent, given the fact that the relationship between the starting poses in the local maps is known. This relationship is captured by the edges that connect the local maps, therefore the loss of information, due to the aforementioned assumption, is diminished. The fact that the robot's path are conditionally independent, yields that the local maps are conditionally independent as well. Figure 2.13 depicts the graphical model of this method.

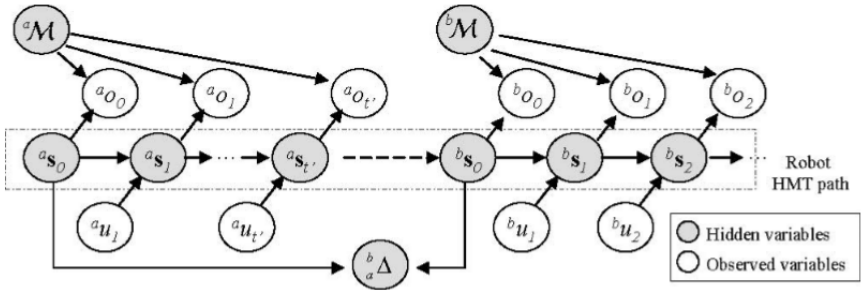


Figure 2.13: The graphical model described in Blanco et al. [12]. s is the hybrid pose, o is the observation, u is the input control, M is a local map and Δ is the edge connecting two local maps. The edge represents the relationship between the starting poses at each map. Thus, the segments of the robot path are conditionally independent, given the edge. Consequently, the local maps are conditionally independent as well, given the robot path.

In the work of Pinies et al. [9] independence between submaps is not regarded. Common features between submaps are placed in both maps. When the robot decides to create a new submap, it records all the features related to its last pose. These features are the common features between the two submaps, whereas the rest features of the old submap are the non-common features. The new submap is initialized by setting the new pose to $(x = 0, y = 0, \theta = 0)$ and expressing the common features with respect to the new pose. Then the robot continues mapping until it decides to create a new submap and repeat the same procedure. Figure 2.14 depicts the graphical model of this method.

Another interesting work on HMT SLAM is the ForMeT framework [14]. ForMeT stands for Forced Resampling Metric Topological. This work was conducted during a master thesis at RSS (Robot Security Systems).

ForMeT is a HMT SLAM algorithm based on the RBPF technique. ForMeT, therefore, uses a particle filter in order to estimate the robot's path and assigns a topological map per particle. The size of each local map is fixed and each time a particle samples a pose outside the boundaries of a local map, a new map is added for that particle. ForMeT incorporates the work of previous HMT SLAM works and contributes with a new mechanism called forced resampling.

The novel idea in ForMeT is that particles should maintain a different path hypothesis up to a specific time in the past. Before that point there should exist a unique path hypothesis for all particles. Therefore, particles decide on a common past path hypothesis up to a point and then they maintain their own different path hypothesis. Since local maps are conditioned on the estimated path, this

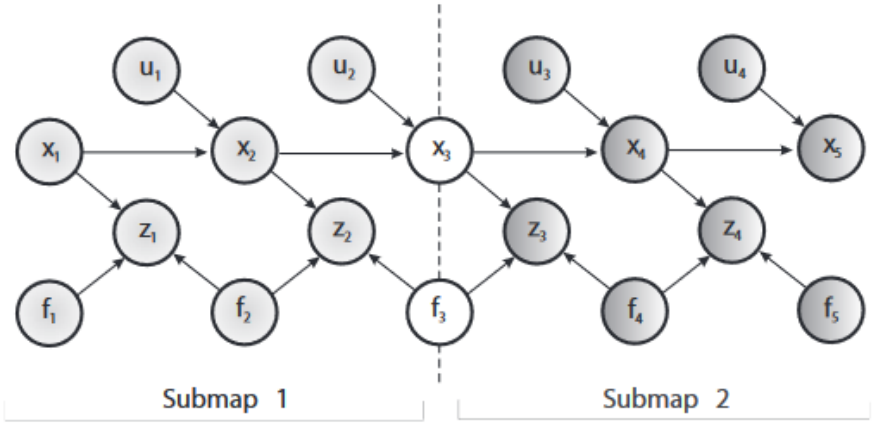


Figure 2.14: The graphical model of Pinies et al. [9]. The global map is partitioned in smaller submaps which are not independent. The last pose of the first submap is the pose x_3. The feature related to that pose is feature f_3 which is the one that the two submaps share, as a common feature. (Note: x is the pose, z is the observation, u is the input control and f are the features.)

means that the estimated map is the same up to the decided point for all particles and after that point there is a different map assigned per particle.

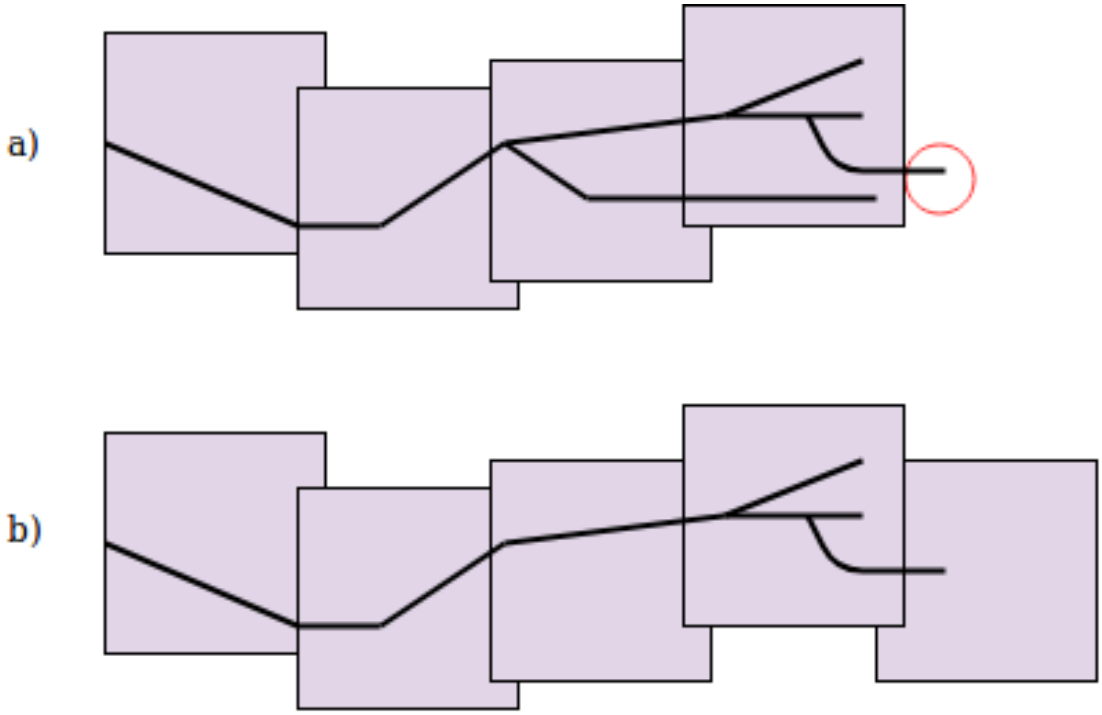


Figure 2.15: ForMeT's concept of shared path. (a) Particles share a common path at the first two maps. In the third map there exist two path hypotheses, whereas in the fourth (current map) there exist four path hypotheses. The red circle indicates that at least one particle sampled a pose outside the designated area of the fourth map. Therefore a fifth map needs to be constructed. (b) Before creating the fifth map, forced resampling took place and the particles decided on a common path (one out of the two) in the third map. The discarded path is no longer visible.

In ForMeT the particles maintain different path hypothesis in the last two local maps. When a particle samples a pose outside the designated area of the current local map, forced resampling is performed. Particles decide on a common path at the previous map and then the previous map is added in the shared topology, since it is the same for all the particles. Figure 2.15 depicts the concept in ForMeT.

The existence of the shared topology makes ForMeT a lightweight algorithm, when it is compared to

existing HMT SLAM techniques, both in terms of memory usage and execution time. Existing HMT SLAM techniques either keep the set of local maps into memory for each particle or they keep the collection of sensor measurements and create the maps on demand trying to keep memory consumption low. In the first case they require extended amount of memory, whereas in the second case they increase the execution time. However, ForMeT can keep all the maps loaded into memory; the memory usage is still low and there is no additional execution time involved. For example, let's assume an RBPF algorithm which uses 30 particles and has created 30 maps for each particle. In this case, the total amount of local maps in memory would be $30*30 = 900$. However in the ForMeT case it would be $28+2*30 = 88$ local maps in memory.

2.4. Range-Only SLAM

As we discussed at the beginning of this chapter, pose correction is necessary in order to perform quality mapping. While correcting the pose, the robot takes into account its surroundings. However, there can exist areas with many similar features or there can exist areas that do not pose unique characteristics. These environments affect the pose correction process and can yield poor quality map. When considering large scale environments, there is even higher possibility of meeting an area with the aforementioned characteristics.

In order to solve the aforementioned problem, the usage of radio sensors has been introduced in the SLAM process. One of the most common used radio sensors in SLAM is the Ultra Wideband (UWB) beacons transceiver. UWB beacons transceivers are range sensors that use *time-of-flight* in order to do ranging and determine their relative location. These sensors are accurate enough with their ranging function, determining the distances with low errors. In addition UWB beacons can self identify themselves by being assigned a unique identification tag. Therefore, UWB beacon transceivers can be used in SLAM in order to solve the data association problem and correct accurately the estimated robot pose.

The usage of radio sensors for performing SLAM introduces the concept of Range-Only SLAM (RO-SLAM). RO-SLAM refers to the problem of building maps with range-only sensors. These sensors do not provide any information about the direction of the measurement, therefore the procedure of discovering landmarks is different to what it is described up until now.

RO-SLAM has to deal with problems that are not present in traditional SLAM. Firstly, given just one sensor observation, there is a large part of an environment that a landmark could be. In addition, there is high possibility that multiple plausible hypotheses exist about the landmark's position. Finally, range-only sensors produce outlier values.

There exist various RO-SLAM techniques, but their main difference lies on the initialization phase. Specifically, given a range measurement r from a particular newly observed beacon, the possible location of the beacon could be anywhere on the circle defined with center the current pose of the robot and with radius the range measurement r. Therefore, the possible locations of a newly observed beacon are numerous.

In the work of Olson et al. [15] an initial estimation of the possible location of the beacon is computed using a voting scheme over a 2D grid. Specifically, as the robot moves and acquires range measurements, it updates a grid map. The grid cell that scores higher after some time, having more occupied values than any other grid cell is voted as the winner cell and its location is marked as a possible beacon location. Then an EKF is applied in order to refine the estimate of the beacon's location.

In the work of Blanco et al. [16] a Sum of Gaussians (SOGs) scheme is employed in order to approximate the aforementioned circle that represents the possible locations of a beacon. A number of equally spaced Gaussians, representing beacon's locations, are generated on the circle circumference. The density of the Gaussians is a designer's choice; the more the Gaussians, the more accurate the result but the higher is the increase on computation time. Each one of these Gaussians is described by a mean value, a covariance and a weight factor. The weight factor represents the likelihood that the specific Gaussian is the correct one. The robot moves and a pose estimate is performed. Then, the Gaussians that represent the location of a specific beacon are weighted based on the difference between the expected measurement and the real measurement. After the robot has moved for some meters the weight of most of the Gaussians should have become negligible. Finally, only one has considerable weight and it actually represents the location of the beacon. In Figure 2.16 the afore-

mentioned technique is shown. It can be observed that the convergence of this technique depends highly on the travelled path.

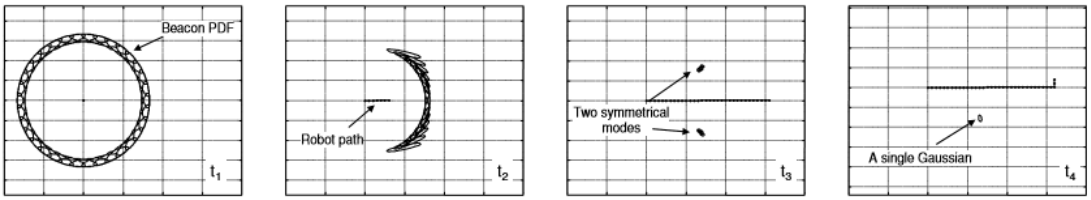


Figure 2.16: At time t_1 the possible locations of the beacon are described by Gaussians equally spaced over the circumference of the circle. As the robot moves, more and more Gaussians are discarded and finally one is the winner [16].

2.5. Summary and Motivation

The SLAM problem is usually estimated using probabilistic laws. Many SLAM techniques have been introduced as a solution to the problem, producing different kind of maps and using different methods. Metric SLAM techniques such as FastSLAM or EKF SLAM that provide one global metric map are accurate in small areas, but they are inaccurate in large areas. This happens mainly due to the accumulated error on the robot's sensors. Therefore, in larger areas HMT SLAM techniques are preferred [9], [10], [11], [12], [13]. HMT SLAM techniques segment the global map into multiple connected local maps. The global map is represented by a graph, consisting of nodes and edges. Each node of the graph represents a local map, whereas the edges represent navigability between the maps. In HMT SLAM the local maps are small enough, so mapping strategies are accurate inside them, whereas he global map's inaccuracies are captured by the graph.

However, as the mapping area grows even larger, the number of nodes which represent the local areas increases. As the number of nodes increases execution time and the memory consumption of these HMT SLAM algorithms increases as well. Therefore, HMT SLAM techniques scale up to a certain area. In addition, when mapping large scale environments, pose correcting which is of great importance for the quality of the mapping process, may get troublesome. In large environments there might exist many similar features or subareas that don't pose unique characteristics. This could lead to poor estimation and correction of the robot pose, which could lead to inaccuracies of the map. Therefore, combining traditional SLAM techniques with RO-SLAM techniques might prove beneficial for the quality of the map in large environments, since RO-SLAM involves the discovery of distinguishable landmarks that facilitate the pose correction process.

3

Theoretical Analysis of the Hierarchical Framework

As discussed in the previous chapter, a common strategy to achieving consistent maps is to solve the SLAM problem, using a hybrid metric topological scheme. Even though this strategy produces an accurate map, it proves to be non-scalable when the mapping area grows significantly large.

As the mapping area becomes larger, the number of nodes, which represent smaller local areas in the global map, increases as well. Consequently, the complexity of calculating paths between nodes, which can be used for loop closure or navigation purposes, increases linearly or sometimes quadratically to the number of nodes. Linear or quadratical increase on the complexity yields linear or quadratical increase on the execution time. In addition, memory usage grows linearly to the number of nodes. Finally, when it comes to loop closing, it is evident that in a large area there could be many similar features, so the probability for performing a wrong loop closure is high.

In order to address the aforementioned problems an hierarchical mapping scheme was designed and implemented. The details of this scheme are discussed in the following sections of this chapter.

3.1. Hierarchical Model Assumptions

Let's assume a global metric map. We could partition it in smaller connected local areas, called nodes, thus creating a topological graph. Then each one of these nodes (representing a map), could be further partitioned creating another topological graph. This procedure could continue for an arbitrary number of times, until the global map is fragmented into many layers. This notion is depicted in Figure 3.1, where the hierarchical connection among the graphs is shown. We call this a multi-layer hierarchical topological graph. In order to further analyze our scheme, we should first make some assumptions that create its basis:

1. Given L+1 sensors an hierarchical graph of a total of L+1 layers can be created
2. Each graph lies on a unique hierarchy layer (l).
3. The lowest in hierarchy graph lies on layer 0.
4. The highest in hierarchy graph lies on layer L.
5. Each sensor is associated with only one topological layer and vice versa.
6. Layer 0 nodes represent fixed size local maps that hold metric information which is the robot's perception of the world.
7. Each node of layers 1 to L represents an area, containing one or more layer-0 nodes.
8. The higher the layer is in the hierarchy, the wider should be the area that its nodes represent.
9. The higher the layer is in the hierarchy, the longer is the range of the sensor associated with it.

10. All nodes can have multiple children but only one parent. Layer 0 nodes are an exception since they don't have children and they have one or more parents.

11. Each node of layer L is characterized by a unique piece of information. That piece of information is derived from the layer's associated sensor and the map. That piece of information is recognizable inside an area in the map.

12. Nodes of layer 1 to $L-1$ that share the same parent are characterized by a piece of information that is unique among them, derived from the layer's associated sensor and the map. Nodes that have different parent can be characterized by the exact same piece of information.

13. At layer 0 a new node is created, as soon as the sampled pose falls outside the designated borders.

14. At layers 1 to L a new node is created, as soon as the current piece of information has changed.

15. If a new node is created to one of the layers 2 to L, then new nodes are created to all the lower layers, except for layer-0.

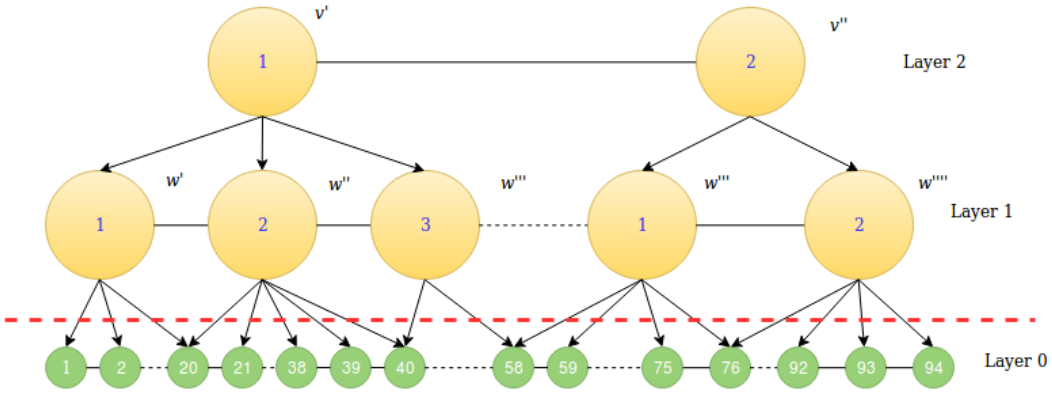


Figure 3.1: Example of an hierarchical graph: We observe three topological graphs connected in an hierarchical way, following a parent-child structure. layer-0 nodes represent fixed size local maps, while layer-1 and layer-2 nodes represent wider areas in the map. This can also be inferred from the parent child structure. Layer-1 and layer-2 nodes are characterized by a piece of information ($v', v'', w', w'', w''', w''''$), which is distinct in the wider area that these nodes represent.

3.2. Hierarchical Model Function

The application of our hierarchical graph during the mapping process mainly aims to facilitate the node traversal procedure, which happens during the topological loop closure.

Before discussing how our framework achieves that optimization, let's discuss how topological loop closure happens in a non hierarchical HMT mapping scheme. In general, the robot moves on a trajectory while creating connected local maps. Each time that the robot decides to exit the current node which represents the current local map, it has to either create a new node (local map) or traverse to a known node (local map). In order to do so it performs a series of steps:

1. A path planning algorithm is executed on the current graph structure, in order to return paths, starting from the current node towards each other node of the graph. Usually, shortest path planning algorithms are used in order to return optimal paths which are the ones that hold the least uncertainty.

2. Given the solution of the path planning algorithm, the transformation and the uncertainty projection towards all other nodes is calculated.

3. If the robot's pose falls into the uncertainty area of one of the target nodes' map, then that node is eligible for traversal and a map matching technique is applied.

4. Depending on the map matching result, a new node is created or traversal to a known node takes place.

However, as the mapping area becomes bigger the resulting graph becomes bigger and more interconnected, slowing down the performance of path planning algorithms. Our hierarchical mapping scheme tries to solve this problem as descibed below.

3.2.1. Tree Path Computation

The general idea is that as the robot moves inside the mapping area, it creates fixed size local maps, represented by layer-0 nodes. The robot groups these local maps into an hierarchical graph that is based on the robot's surroundings. At each time step the robot localizes itself by calculating the current tree path based on the hierarchical graph structure. In other words the robot decides at which node of each layer it is currently at.

Specifically, starting from layer L and moving downwards to layer-1, for each layer the robot takes into account the associated layer's sensor and the map, in order to decide the current node. Once it decides on a node of a layer, it then looks into the children of the decided node. If during this process, a new node needs to be created at one of these layers, because a new piece of information has been detected (one that is not assigned to any node so far), new nodes are created to all the lower layers, except for layer-0 (Assumption 15). The new nodes that need to be created at the lower layer are characterized by the piece of information which the layer currently possesses. If the piece of information that the lower layer currently possesses is the same as in the previous time step, the new node is a duplicate of the old one. This will not create any issues since the new node shares different parent than the old node (Assumption 12). Via this method, a specific area consisting of one or more layer-0 nodes is estimated at each time step. This estimation is of great importance as we explain further below.

3.2.2. Hierarchical Edge

Another key aspect of the hierarchical graph is the *hierarchical edges* mechanism. Firstly let's define the hierarchical edge. In order to do so, let's remember that the nodes of the lowest layer in our framework are connected through edges, which carry the transformation from the local coordinate frame of the source node (local map), towards the local coordinate frame of the target node(local map), along with the uncertainty of the transformation. Each node that belongs to a layer, apart from layer-0, is ultimately containing layer-0 nodes and edges, practically defining a wider area in the map than each node of layer-0. **Let's consider all layer-0 nodes that exist on the boundaries of higher layer nodes.** We will call these nodes, *gateway nodes*. We can then calculate the accumulated transformation and the accumulated uncertainty among these gateway nodes and store this result into memory as an hierarchical edge.

3.2.3. Node Traversal

Both the tree path computation, as described in subsection 3.2.1, and the hierarchical edge mechanism are applied in order to optimize the layer-0 node traversal procedure. We will proceed with an example in order to better explain how the node traversal procedure is performed within our framework.

Let's assume a large area which consists of four distinguishable areas A, B, C and D (Figure 3.2). These areas represent layer-1 nodes, whereas the numbered circles represent layer-0 nodes. The robot followed a trajectory starting from area A, traversing to area B, then area C, then area D and then it returned back to area B when exiting area D. Figure 3.3 depicts the equivalent hierarchical graph.

While being inside area A the robot creates fixed size layer-0 local maps alongside its trajectory. Specifically, each time that it samples a pose outside the designated boundaries, it checks if it should traverse to a known layer-0 node or if it should create a new layer-0 node. In order to do so, the robot follows the same procedure which we described at the beginning of this section. When the robot is at layer-0 node 38, it senses that it enters area B. At that point it calculates and stores into memory the hierarchical edge between layer-0 nodes 1 and 38 (Figure 3.2(a)).

The robot continues inside area B following the depicted trajectory. The procedure is the same as before, with one exception; when layer-0 node traversal takes place, the robot considers possible traversal candidates only layer-0 nodes that are contained in area B. Therefore, it has to plan on less nodes than a non-hierarchical schemes does. In the same manner as before, while the robot is at node

89, it senses that it enters area C and it calculates the hierarchical edge between layer-0 nodes 89 and 38 (Figure 3.2(b)).

Then the robot continues moving inside area C. As before, while being inside this area, it considers only the layer-0 nodes that are contained by area C, until it reaches node 136. Then the robot senses that it enters area D and calculates the hierarchical edge between layer-0 nodes 89 and 136 (Figure 3.2(c)).

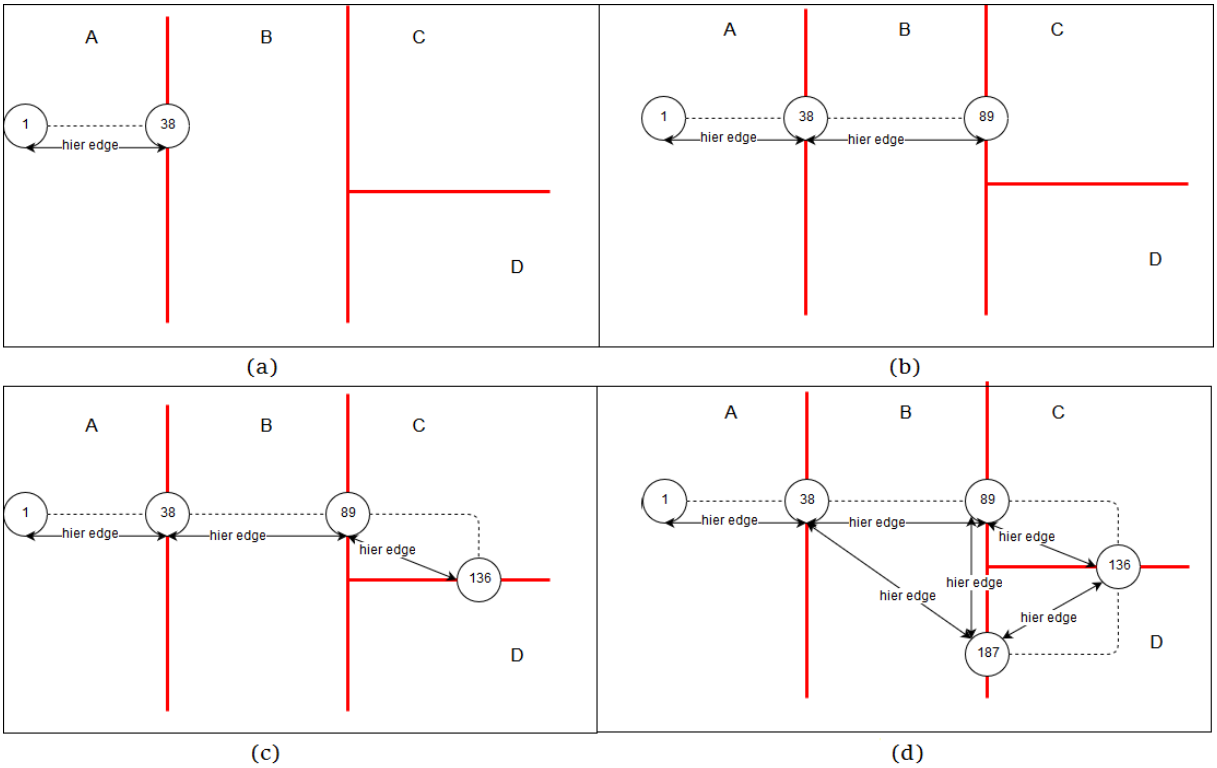


Figure 3.2: Travelled path of a robot inside 4 recognizable areas. The robot moved for a total trajectory upon which it create 187 fixed size local maps (layer-0 nodes). During its trajectory it traversed among 4 recognizable areas (layer-1 nodes).

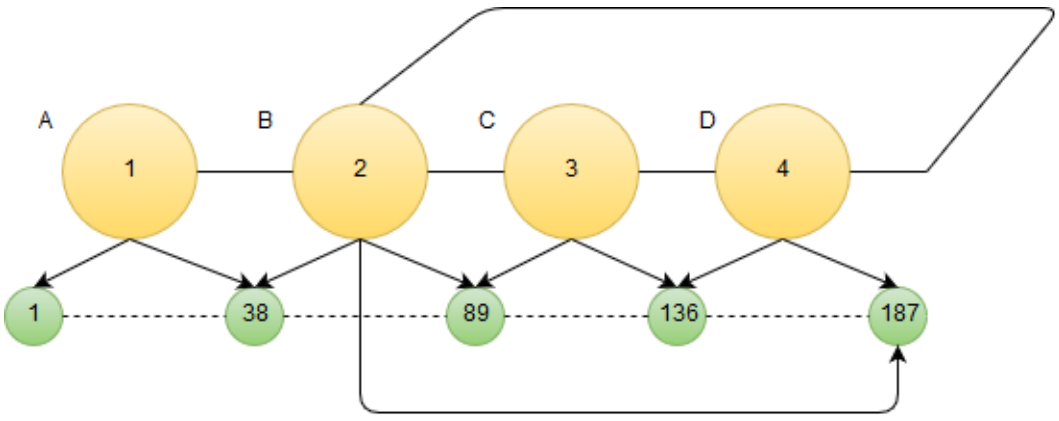


Figure 3.3: Equivalent hierarchical graph of Figure 3.2 example. We observe a loop closure at layer-1, where from node 4 the robot returned back to node 2. That happened while being on node 187, thus that node shares two parents.

The robot continues moving into area D, behaving in the same manner as the previous areas and eventually reaches node 187. Then it senses that it returned back to area B and thus it calculates the hierarchical edge 136 to 187. At that point the robot also needs to calculate the hierarchical edges 38 to 187 and 89 to 187. The reason for computing the edges 38 to 187 and 89 to 187 lies in the fact that

the graph, which consists of layer-0 nodes contained by area B, needs to be fully connected (Figure 3.2(d)). As the robot moves inside area B, each time that it has to check for traversal to a known node, it plans on area B contained nodes, executing path planning algorithms.

On one hand the hierarchical edge mechanism speeds up the computations and connects areas that normally would not be connected. When calculating the hierarchical edges (38 to 187 and 89 to 187), the already built hierarchical edges were taken into account, keeping the computations low and resulting in all nodes in area B being connected. On the other hand computing the current tree path diminishes the amount of candidate layer-0 nodes. Taking into account the aforementioned mechanisms, we expect time efficient results as well as a more successful loop closure.

Finally, we should note that the hierarchical edges can be used for navigation purposes in the long-term, after the mapping procedure has completed. In a large scale area that consists of thousands or millions of nodes, path planning algorithms can get more efficient by using the hierarchical edges, practically diminishing the size of the graph.

3.3. SLAM Posterior Distribution

Given the assumptions and the function of our model, we proceed further in order to describe it mathematically. The mathematical description is used in order to continue with our model's implementation.

As discussed in Chapter 2, the Full SLAM problem is described by Equation 2.2. This equation expressed for the example of Figure 3.1, incorporating the extra sensor observations, takes the form:

$$p(x_{1:t}, m | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) \tag{3.1}$$

As we have discussed the full SLAM problem yields an interesting property; if we the robot path is known, the estimation of the individual landmarks is conditionally independent to each other. This yields that the SLAM problem is split into two different estimation problems, one for the robot path and one for the map. The map posterior can then be factorized into L posteriors for each landmark. So, Equation 3.1 can be expressed as:

$$\begin{aligned} p(x_{1:t}, m | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) &= \\ &= p(x_{1:t} | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) p(m | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}, x_{1:t}) \\ &= p(x_{1:t} | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) \prod_{l=1}^{L} p(m_l | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}, x_{1:t}) \end{aligned} \tag{3.2}$$

We will now further analyze the leftmost term in the right hand side of Equation 3.2, which represents the posterior distribution over the robot's path. Applying the Bayes rule, the posterior distribution over the robot's path can be expressed as:

$$\begin{aligned} &p(x_{1:t} | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) = \\ &\quad p(z_t, w_t, v_t, | x_{1:t}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) p(x_{1:t} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) \end{aligned} \tag{3.3}$$

Let's examine each term of the right hand side of Equation 3.3 separately. Looking at the rightmost term, we can apply the multiplication rule, which yields that:

$$\begin{aligned} &p(x_{1:t} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) = \\ &\quad p(x_t | x_{1:t-1}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}, m) p(x_{1:t-1} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) \end{aligned} \tag{3.4}$$

Equation 3.4 can be further simplified by applying the Markov assumption:

$$p(x_{1:t} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) = p(x_t | x_{t-1}, u_t) p(x_{1:t-1} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \tag{3.5}$$

The rightmost term at the right hand side of Equation 3.5 is the posterior over the robot's path, just a time step earlier. This verifies that the robot path is recursively estimated.

Now let's examine the left most term of the right hand side of Equation 3.3. Applying the total probability theorem, we can incorporate the map variable. Subsequently we can apply the Markov

assumption and finally we can get that:

$$p(z_t, w_t, v_t | x_{1:t}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) = \int p(z_t, w_t, v_t | x_{1:t}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}, m) p(m | x_{1:t}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) dm = \int p(z_t, w_t, v_t | x_t, m) p(m | x_{1:t-1}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) dm \tag{3.6}$$

We denote n_l^i as one node of the graph; l refers to the node's layer whereas i refers to the node's index. These nodes represent an area in the map. Each node of the layer-0 (n_0^i) can be uniquely characterized by the junction of all its parent nodes that create a tree path, in the sequence $n_L^b \cap n_{L-1}^i \cap \ldots \cap n_1^j \cap n_0^k$.

For example, taking into account the graph of Figure 3.1, the global map is split in unique areas by computing each possible tree path and denoting by the junction of the nodes of each layer as $A_{i,j,k} = (n_0^k, n_1^j, n_2^i)$.

Thus, by applying the total probability theorem in the left most term of the right hand side of Equation 3.6, it is derived that:

$$p(z_t, w_t, v_t, | x_t, m) = \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} p(z_t, w_t, v_t, | x_t, m, A_{i,j,k}) p(A_{i,j,k} | x_t, m) = \sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} p(z_t, w_t, v_t, | x_t, m, n_0^k, n_1^j, n_2^i) p(n_0^k, n_1^j, n_2^i | x_t, m) \tag{3.7}$$

We now can apply subsequently, the multiplication rule at the right most term of the right hand side of Equation 3.7 and get:

$$p(n_0^k, n_1^j, n_2^i | x_t, m) = p(n_0^k | x_t, m, n_1^j, n_2^i) p(n_1^j | x_t, m, n_2^i) p(n_2^i | x_t, m) \tag{3.8}$$

Thus substituting Equation 3.8 at Equation 3.7, Equation 3.7 takes the form:

$$p(z_t, w_t, v_t | x_t, m) = \sum_{i=1}^{I} \left[p(n_2^i | x_t, m) \sum_{j=1}^{J} \left[p(n_1^j | x_t, m, n_2^i) \sum_{k=1}^{K} \left[p(n_0^k | x_t, m, n_1^j, n_2^i) p(z_t, w_t, v_t | x_t, m, n_0^k, n_1^j, n_2^i) \right] \right] \right] \tag{3.9}$$

Assuming that the sensors' observations are independent to each other, Equation 3.9 can be further simplified to:

$$p(z_t, w_t, v_t, | x_t, m) = = \sum_{i=1}^{I} \left[p(n_2^i | x_t, m) \sum_{j=1}^{J} \left[p(n_1^j | x_t, m, n_2^i) \sum_{k=1}^{K} \left[p(n_0^k | x_t, m, n_1^j, n_2^i) \right. \right. \right. \left. \left. \left. p(z_t | x_t, m, n_0^k, n_1^j, n_2^i) p(w_t | x_t, m, n_0^k, n_1^j, n_2^i) p(v_t | x_t, m, n_0^k, n_1^j, n_2^i) \right] \right] \right] \tag{3.10}$$

Equation 3.10 depicts the segmentation of the global map into areas in every possible tree path combination, starting from layer-2 and reaching layer-0. It is also implied that for each tree path, the probability to be at the specific tree path, is calculated by starting at the root node and moving downwards. We now substitute Equation 3.10 back to Equation 3.6. Therefore, Equation 3.6 can be

expressed as:

$$
\begin{aligned}
&p(z_t, w_t, v_t | x_{1:t}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t}) = \\
&\int \Bigg[\sum_{i=1}^{I} \Bigg[p(n_2^i | x_t, m) \sum_{j=1}^{J} \Bigg[p(n_1^j | x_t, m, n_2^i) \sum_{k=1}^{K} \Bigg[p(n_0^k | x_t, m, n_1^j, n_2^i) p(z_t | x_t, m, n_0^k, n_1^j, n_2^i) \\
&p(w_t | x_t, m, n_0^k, n_1^j, n_2^i) p(v_t | x_t, m, n_0^k, n_1^j, n_2^i) \Bigg] \Bigg] \Bigg] p(m | x_{1:t-1}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \Bigg] dm
\end{aligned}
\tag{3.11}
$$

Finally, substituting 3.11 and 3.5 back to 3.3 and then 3.3 back to 3.2, the SLAM problem can be described by:

$$
\begin{aligned}
&p(x_{1:t}, m | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}) = \\
&\int \Bigg[\sum_{i=1}^{I} \Bigg[p(n_2^i | x_t, m) \sum_{j=1}^{J} \Bigg[p(n_1^j | x_t, m, n_2^i) \sum_{k=1}^{K} \Bigg[p(n_0^k | x_t, m, n_1^j, n_2^i) \Bigg[p(z_t | x_t, m, n_0^k, n_1^j, n_2^i) \\
&p(w_t | x_t, m, n_0^k, n_1^j, n_2^i) p(v_t | x_t, m, n_0^k, n_1^j, n_2^i) \Bigg] \Bigg] \Bigg] p(m | x_{1:t-1}, z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \Bigg] dm \\
&p(x_t | x_{t-1}, u_t) p(x_{1:t-1} | z_{1:t-1}, w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \prod_{l=1}^{L} p(m_l | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}, x_{1:t})
\end{aligned}
\tag{3.12}
$$

Now we can generalize for an arbitrary number of layers. This means that we incorporate additional sensor information $(z, y, ..., w, v)$ and consequently result to Equation 3.13.

$$
\begin{aligned}
&p(x_{1:t}, m | z_{1:t}, y_{1:t} ... w_{1:t}, v_{1:t}, u_{1:t}) = \\
&\int \Bigg[\sum_{i=1}^{I} \Bigg[p(n_L^i | x_t, m) \sum_{j=1}^{J} \Bigg[p(n_{L-1}^j | x_t, m, n_L^i) ... \sum_{n=1}^{N} \Bigg[p(n_1^n | x_t, m, n_2^a ..., n_{L-1}^j, n_L^i) \\
&\sum_{k=1}^{K} \Bigg[p(n_0^k | x_t, m, n_1^n, ..., n_{L-1}^j, n_L^i) p(z_t | x_t, m, n_0^k, n_1^n, ..., n_{L-1}^j, n_L^i) p(y_t | x_t, m, n_0^k, n_1^n, ..., n_{L-1}^j, n_L^i) \\
&...p(w_t | x_t, m, n_0^k, n_1^n, ..., n_{L-1}^j, n_L^i) p(v_t | x_t, m, n_0^k, n_1^n, ..., n_{L-1}^j, n_L^i) \Bigg] \Bigg] ... \Bigg] \Bigg] \\
&p(m | x_{1:t-1}, z_{1:t-1}, y_{1:t-1}, ... w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \Bigg] dm \;\; p(x_t | x_{t-1}, u_t) \\
&p(x_{1:t-1} | z_{1:t-1}, y_{1:t-1}, ... w_{1:t-1}, v_{1:t-1}, u_{1:t-1}) \prod_{l=1}^{L} p(m_l | z_{1:t}, y_{1:t}, ... w_{1:t}, v_{1:t}, u_{1:t}, x_{1:t})
\end{aligned}
\tag{3.13}
$$

Given the aforementioned Equation we can conclude that:

1. The SLAM posterior is split into a localization posterior and a map posterior.
2. The map posterior is factorized into L different posteriors, one for each of the L landmarks. In order to calculate the map posterior at timestep t, the path posterior needs to be calculated first.
3. In order to calculate the path posterior a series of posteriors are calulated:
   (a) We first predict the current state of the robot, given the previous state and the input control.
   (b) Based on the predicted current state and the estimated map of the previous time step, we predict the current tree path.
   (c) Then we correct the prediction by applying the different observation models. Since the prediction of being at a node is based on the associated observation, applying the correspondent observation model accounts for correcting our prediction.

(d) After applying the correction step, we update the robot path with the latest state.

4. We use the known robot path as input for estimating the map.

5. The procedure repeats itself in a recursive manner.

To conclude, we inserted the notion of hierarchical nodes in the mapping equation and we can now move further into discussing the implementation details.

4

Implementation of the Hierarchical Framework

The previously theoretical analysis concludes in Equation 3.13 which describes our hierarchical mapping scheme. In this chapter we present our implementation; describing the basic components that we used and the way they function. The implementation of the framework is done in C++ and the final deliverable consists of a package compatible with the Robot Operating System (ROS). If the reader is not familliar with ROS, it is reccommended to go through Appendix D and the related sources before proceeding with this chapter.

4.1. SLAM Implementation

The final form of Equation 3.13 allows us to implement the equation using the Rao Blackwellized Particle Filter (RBPF) technique. As a reminder, RBPF uses a particle filter in order to estimate the robot path and Gaussian estimators in order to estimate the location of the landmarks. Practically, a different map is associated per particle, since each particle represents a different path hypothesis.

In our implementation the output consists of a network of local maps connected in an hierarchical graph. Thus, instead of associating one global map per particle sample, we associate one hierarchical graph per particle. The original RBPF algorithm had to be modified. As mentioned in subsection 2.2.2, the original algorithm consists of (a) sampling a pose for each particle, (b) weighting the particles based on the sampled pose, the expected and the real observation, (c) updating the map and (d) resampling. Our modified version is depicted in Figure 4.1. In our modified version there is an extra step. After sampling a pose for each particle, the topology of the graph is updated for each particle. The rest steps follow the original RBPF algorithm.

So our hierarchical mapping algorithm initializes the particle set and the associated maps and then repeats the following recursive procedure:

1. For each particle:
   (a) Given the previous pose and the current odometry, a new pose is sampled.
   (b) Given the new sampled pose, the new set of sensor observations and the map of the previous time step, the topology of the graph is updated as described in section 3.2.
   (c) Given the new sampled pose, the map of the previous time step and the observations, the weight of the particle is calculated.
   (d) Given the new sampled pose, the map of the previous time step and the observations, the associated map is updated
2. For the entire particle population, particles are resampled if the variance of their weights has crossed a threshold.

Algorithm 1 Hierarchical Mapping

1: **for** k=1 to M **do**
2: $initialize\_all\_particles\_at\_(x, y, \theta) = (0, 0, 0)$
3: $initialize\_map$
4: **end for**
5: **while** $mapping$ **do**
6: **for** $k = 1$ to M **do** // for all particles
7: $x_t^{[k]} \approx p(x_t | x_{t-1}^{[k]}, u_t)$ // sample a pose
8: $nodes = \{all\_tree\_root\_nodes\}$ // consider root nodes
9: $parents = empty$
10: $new\_node\_layer = -1$
11: **for** $l = L$ to 1 **do** // for all layers
12: $n_l^{i[k]} \approx p(n_l^i | x_t^{[k]}, m_{t-1}, parents)$ //compute the i-th node where the robot is at layer l
13: **if** $n_l^{i[k]}$ in nodes **then** //if is the current or known node
14: $nodes = \{children\_of(n_l^{i[k]})\}$ //consider its children nodes
15: $add\ n_l^{i[k]}\ to\ parents$ //add the decided node of layer l to parent set
16: **else** // if a new node needs to be created
17: $new\_node\_layer = l$ //indicate the layer that this happened
18: break
19: **end if**
20: **end for**
21: **if** $new\_node\_layer \neq -1$ **then** //if new node created at a layer
22: **for** $l = new\_node\_layer$ to 1 **do** //create new nodes to all lower layers
23: $create\_new\_node\_at\_l$
24: **end for**
25: **for** $l = 1$ to new_node_layer **do** //create hierarchical edges
26: $create\_hierarchical\_edge$
27: **end for**
28: **end if**
29: $nodes = \{children\_of(n_1^{i[k]})\}$ //consider layer-0 nodes that belong to the tree path
30: **if** $x_t^{[k]} \geq THRESHOLD\_DISTANCE$ **then** // if pose outside the desgnated area
31: $n_0^{i[k]} \approx p(n_0^i | x_t^{[k]}, m_{t-1}, n_1^{i[k]}, \ldots, n_L^{i[k]})$ //decide traversal to known or creation of new
32: **if** $n_0^{i[k]}$ in nodes **then**
33: $traverse\_to\_node$
34: $update\_edge$
35: **else**
36: $create\_new\_node$
37: $create\_new\_edge$
38: **end if**
39: **end if**
40: $weight^{[k]} \approx p(z_t | x_t^{[k]}, m_{t-1}) p(w_t | x_t^{[k]}, m_{t-1}) p(v_t | x_t^{[k]}, m_{t-1})$ //calculate weights
41: $m_t^{[k]} \approx p(m | z_{1:t}, w_{1:t}, v_{1:t}, u_{1:t}, x_{1:t}^{[k]})$ //update layer 0 local map
42: **end for**
43: **if** $N_{eff} \leq THRESHOLD$ **then** //resample
44: $resample\_particles$
45: **end if**
46: **end while**

Figure 4.1: The modified version of the original RBPF algorithm. After sampling a new pose, an update of the graph is performed (Lines 8 to 39).

4.2. Framework Architecture

After deciding on the RBPF SLAM method as the basis of our framework, we proceeded with the implementation details.

We defined the configurability of our framework as the main design requirement. We proceeded on this decision since we require our framework to be applicable to different scenarios. Depending on the robot's resources, the robot's sensors and the robot's environment a different configuration should be applied in order to achieve better performance. This means that:

1. The number of sensors and the number of layers that the hierarchical graph consists of can be chosen arbitrarily.

2. One should provide with the mathematical components of the RBPF which are related to each sensor:
   (a) sampling pose
   (b) particle weighting
   (c) map updating
   (d) map matching technique

As we have already discussed in Chapter 3, the proposed framework consists of $L + 1$ graphs connected in an hierarchical way. The nodes of the lowest hierarchically graph represent fixed size local maps. These local maps hold metric information. Thus, one can safely assume that for layer-0 nothing more is needed other than implementing a Hybrid Metric-Topological (HMT) algorithm. Since we decided to implement our mapping scheme using RBPF (as shown in Figure 4.1), we could use an RBPF based HMT mapping algorithm for layer-0 and then expand it in the hierarchical form, taking into account the remaining higher layers.

Therefore our implementation mainly consisted of two steps:

1. Implementing a generic RBPF based HMT SLAM algorithm.
2. Expanding the HMT SLAM algorithm to have an hierarchical form.

4.3. Implementing a generic RBPF based HMT SLAM algorithm

In Chapter 2, we described the ForMeT framework which constitutes a lightweight hybrid metric-topological SLAM algorithm. ForMeT follows the RBPF technique, but it is also enhanced with an extra mechanism in order to provide with low memory consumption and with fast execution time. Therefore, we decided that ForMeT is one ideal candidate for our layer-0 implementation and that we could built on top of it, in order to create our hierarchical mapping scheme.

4.3.1. Original ForMeT implementation

Even though there exists a ROS implementation of ForMeT, it is not directly applicable in our framework. The reason is that the original ForMeT implementation is not generic and consequently it doesn't meet our requirements. The main problems are the following:

1. ForMeT assumes the existence of only laser scanner data. Our goal is to create a framework where multiple sensors can be used to discover landmarks which in turn are used to update the map's estimate.
2. In ForMeT each node is associated with one local map, which is represented by a grid map. Since we need to use multiple sensors, we also need to allow associating any possible type of map and an arbitrary number of maps to each layer-0 node.
3. In ForMeT, the basic building blocks of the RBPF technique, namely the pose sampling, the particle weighting, the map updating, as well as the map matching for addressing node explicit loop closing, are tightly connected to the laser scanner data and to the resulting grid map. However, since our framework supports different sensors per implementation, these functions should adhere to a certain prototype, based on one's implementation and according to each sensor used.
4. In ForMeT all the range measurements that fall outside the grid map's boundaries are discarded. Therefore, each grid map holds landmarks that are not observable from any other grid map. However, since we aim to generalize ForMeT's concept, there can also be a case where landmarks are observed from two or more different local maps.

4.3.2. Re-implementation of ForMeT

To resolve the aforementioned problems of the original ForMeT implementation we proceeded with a re-implementation of ForMeT from the very beginning. The new implementation is sensor based and generic. The delivered framework consists of two parts. The first part is defined per user implementation, whereas the second part is fixed and operates in the same manner regardless of the specific application.

The configurable part concerns the number of sensors that are used, the number and the type of the maps that are associated to each sensor and to each node, as well as the way that these maps are updated, matched and used for particle scoring. In addition, the configurable part involves specifying the ”sampling pose” function.

In order to implement the configurable part, we used class inheritance techniques. Firstly, we defined the base parent class of sensors that holds the common information for all sensors. Then we defined the base parent class of maps that holds the common information for all maps. Per one's implementation the appropriate sensor classes and maps classes that inherit from the parent classes need to be implemented, with all the inherited methods defined. The main steps that must be followed per one's implementation are:

1. The number and type of sensors are specified. The framework is then subscribed to all the associated ROS topics that publish sensory information.
2. A function that processes the input sensory data per sensor is defined. After acquiring the data from the subscribed ROS topic, this data may need to be filtered or otherwise processed. For example, data of a laser scanner that exceed a threshold distance could be filtered out, in order to achieve faster computation and more accurate mapping.
3. The type of map that each sensor is associated with is specified. There can be one type of map associated to multiple sensors, but there can also be different type of map per sensor.
4. Per type of map the following functions are specified:
   - How the map is updated given the associated sensor(s) to it.
   - How a score of similarity is extracted between the sensor observation and the constructed map. This score is used for weighting the particles.
   - How local maps are matched. This is used on topological loop closure.
   - The way that the location of the landmarks is translated among the maps. This happens in case landmarks are observable from multiple different local maps.
5. We finally define the motion model, which is used by the particle filter in order to sample a pose for each particle. We can also choose if we will take into account the sensors' associated maps, in order to generate particles based on the robot's surroundings.

The fixed part concerns the update of the topological graph, taking into account the results of the configurable part. Therefore, for each particle:

1. A new pose is sampled based on the specified motion model.
2. Given the current node of the graph a score for each associated local map is calculated. The score is calculated, given the sensor observation related to each local map and the features of each local map, (the scoring function is specified in the configurable part). The final score which is used in order to weight the particle is the product of all individual scores that come from each map.
3. If the new sampled pose falls outside of the designated boundaries of the graph's node, the node traversal procedure takes place. In order to decide if traversal to known node will take place and consequently a topological loop closure will be performed, a series of steps are followed:
   (a) Dijkstra's shortest path algorithm is executed in order to return the optimal path, starting from the current node towards each other node of the graph. The optimal path is the one that holds the least uncertainty.
   (b) Given the solution of the Dijkstra's algorithm, the transformation and the uncertainty projection towards all other nodes is calculated.
   (c) If the particle's sampled pose falls into the uncertainty area of one of the target nodes' map, then that node is eligible for traversal. Then map matching for all the associated maps of that node will take place, as we specified in the configurable part.

(d) Depending on the map matching result[1], a new node is created or traversal to a known node takes place. In case that landmarks are observable among maps, then the landmarks are transferred to the new or known map, based on the specified function of the configurable part.

4. For each map that is associated to the current node of the topological graph, an update is performed. The update is performed based on the specified update function for that map.

In Figure 4.2 the software blocks of our re-implementation are shown. The green colored blocks are fixed and they work as we have specified. However, the yellow colored blocks are defined per implementation.

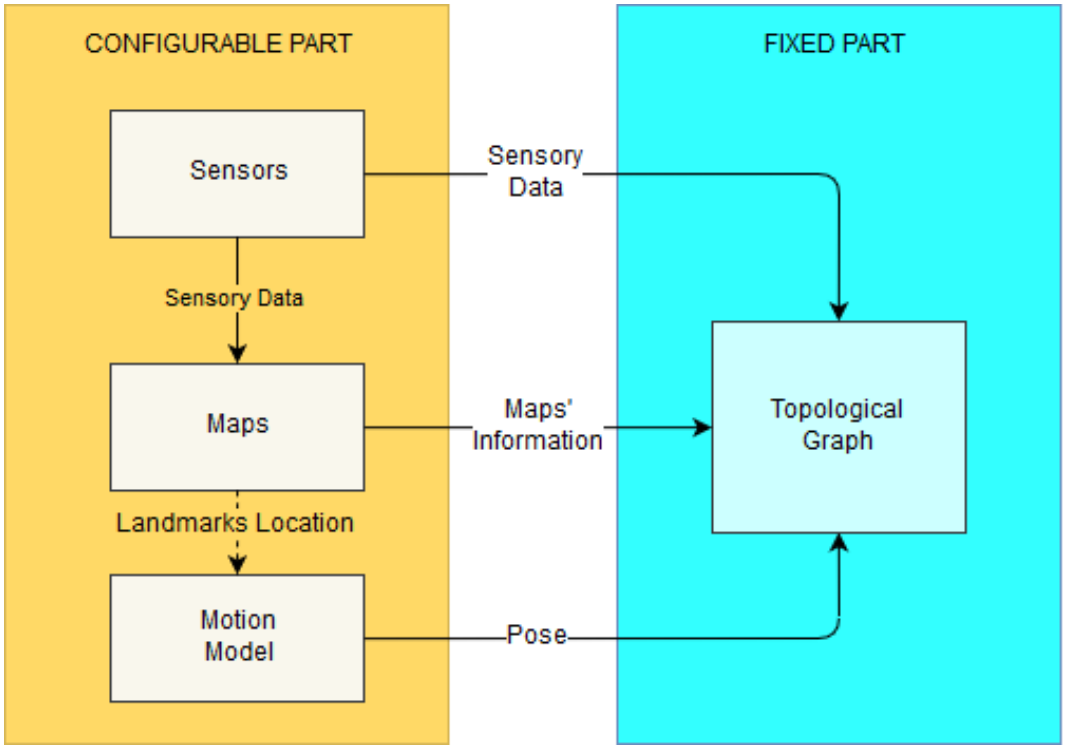


Figure 4.2: These are the software blocks of the generic implementation of ForMeT. The configurable part consists of defining the sensors, defining the maps and defining the motion model. The blocks of the configurable part communicate among them and they also transmit information to the fixed part. The fixed part concerns the update of each topological graph assigned to each particle's path hypothesis.

4.4. Expanding our generic RBPF based HMT SLAM algorithm

Up to this point, we have a hybrid metric-topological SLAM algorithm, which is RBPF based and it is generic to the number of sensors, the number and type of maps associated to each node and the mathematical models that are associated to each map. However, we have to expand our implementation in order for it to work with multiple layers on top and take our desired hierarchical form. We once again provided an implementation that consists of a configurable part, which is defined per one's implementation and a fixed part that always operates in the same manner, producing different results based on the implementation of the configurable part.

The configurable part concerns the number of additional layers, the sensor that each hierarchical node is assigned to, the local map that each hierarchical node is assigned to as well as the type of recognizable information that each hierarchical node carries in order to define an area in the map. In order to implement the configurable part, we used template class techniques. We defined the template class of hierarchical nodes, which holds the appropriate information. Per one's implementation the form

[1]The map matching function varies per implementation, since it belongs to the configurable part. A threshold is decided to account for positive or negative matching. That threshold is given as a tuning parameter to the user.

of the template fields need to be specified as well as the layer that each node lies at. The main steps that are followed are:

1. The total number of additional layers in the topological graph are specified.
2. The sensor that is connected to each additional layer is decided. Then the associated sensor's layer-0 local map is connected to each layer.
3. The distinguishable information that each hierarchical node holds is decided, based on the associated sensor and the associated map. This distinguishable information should be able to define an area in the map.
4. The way that each hierarchical node's distinguishable information is updated/processed at each time step is specified, given the associated sensor and the associated map.
5. The scoring function of each layer is specified. The scoring function returns the most probable hierarchical node that the robot is at each layer, given the hierarchical nodes, the surroundings, the sensor information and the sampled pose.

The fixed part concerns the update of the hierarchical topological graph when taking into account the configurable part's output information. The fixed part addresses the tree path computation, the calculation of the hierarchical edges and the node traversal procedure. The fixed part functions as we specified in Chapter 3. For each particle starting from the highest hierarchical layer and reaching layer 1, the current tree path is decided, according to each particle's path estimate. Given each hierarchical graph assigned to each path hypothesis of each particle, the hierarchical edges are also calculated.

4.5. Proof Of Concept Implementation

After implementing the generic form of our hierarchical multi-layer topological graph, we applied a proof of concept implementation in order to verify the correctness of its function. Specifically, we created three topological layers, associated to a laser scanner and two UWB beacon transceivers. The first UWB beacon transceiver is assumed to have a weaker antenna than the second, therefore having shorter ranging capabilities. Each of the UWB transceivers is part of a different UWB transceiver network. In the following subsections, we present the implementation regarding these three sensors.

4.5.1. Laser Scanner

The first sensor of our implementation is a laser scanner. The implementation of the laser scanner which is discussed in this subsection is the same as in ForMeT [14]. As discussed above, after we defined our sensor we also defined the type of map the laser scanner data are updating. In this case an occupancy grid map is updated.

Firstly, we specified the way that the grid map is updated. Each cell of the grid map is characterized by two values the number of observations n, as well as the ratio of occupied observations α. Each time that the cell is visited, either because the end-point of a beam reached it or because a beam passed through it, the n variable is incremented, whereas the α variable is updated following Equation 4.1. The obs_n variable is the current time step's observation and it is either 0 for free value or 1 for occupied value.

$$\alpha_n = \frac{(n-1) * \alpha_{n-1} + obs_n}{n} \tag{4.1}$$

Finally, a threshold value is assigned to the grid cells and those cells that contain an α value above that threshold value are regarded occupied.

Let's give a numerical example in order to make things clearer. Let's assume a cell of a grid map. The first time the cell was visited an occupied value was placed, the second time a free value was placed and the third time an occupied value was placed again. So:

$\alpha_0 = -1,\ n = 0$

$\alpha_1 = \frac{(1-1)*\alpha_0+1}{1} = 1\ ,\ n = 1$

$\alpha_2 = \frac{(2-1)*\alpha_1+0}{2} = 0.5,\ n = 2$

$\alpha_3 = \frac{(3-1)*\alpha_2+1}{3} = 0.67, n = 3$

If we specify a threshold value of 0.7, the cell is regarded free but if we specify a threshold value of 0.5 the cell is regarded occupied.

Next, we implemented the function that calculates the score which is used to weight the particles. The score is calculated using the likelihood field model, which provides a metric of how well the current laser scan fits into the map, if it is projected from the pose of the particle. Firstly, the pose of the particle and the laser measurement are taken into account. Given the inverse sensor model we calculate the hit-point of a laser scan beam. Then we assign a kernel window around the hit-point and search is done inside this kernel window for the closest matchable cell of the grid map which is constructed by the laser scanner data. This procedure is applied for all or some of the beams of the laser scanner, depending on the trade off of accuracy versus execution time. The result is two sets of points, one representing the matched grid cells and the second representing the hit-points of each beam. For each beam, the distance from the hit-point to the center of its matched grid cell is denoted as d. This distance is fed to Equation 4.2 in order to retrieve the score of each beam and consequently the score for that laser scan. The maximum score for each beam occurs when the distance d is 0. The final score for a specific laser scan and consequently for each particle is the product of all beams score. Figure 4.3 depicts the aforementioned procedure.

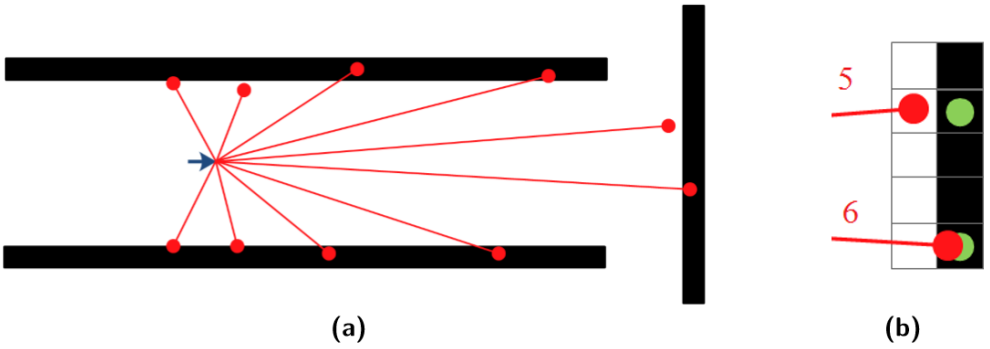


Figure 4.3: Example of calculating the likelihood of the laser scan in the grid map. (a) The laser scan beams that are used for weight calculation. (b) Beams 5 and 6; the distance between the red dot (endpoint of beam) and the green dot (matched grid cell) is used in Equation 4.2

$$s = \exp \frac{-d^2}{\sigma} \tag{4.2}$$

We further refine the procedure using the Iterative Closest Point (ICP) algorithm. Both sets of previously mentioned points are fed as input to the ICP algorithm. The ICP algorithm takes an alignment point-set A and a reference point-set R as input and returns the optimal transform which would align A to R. Therefore, in our case the point-set A corresponds to the set of the hit-points, whereas the point-set R corresponds to the matched grid cells. Since hit-points are expressed with reference to the sampled pose, the optimal transform which is returned by the ICP algorithm is used in order to transform the sampled pose to a new optimized pose. Then using the new optimized pose, the laser scan hit-points are recalculated and a search window is applied in order to find the new grid cells that match the hit-points. These two set of points are again fed into the ICP and the procedure repeats for a given number of iterations or until convergence is reached. That way pose optimization is performed, given the surroundings of the robot. In addition a better and a more representative score is calculated.

Finally, we defined the function that is used for scan matching on the grid maps constructed by the laser scanner. We followed the exact same procedure used for calculating the score for weighting the particles, but with one difference: we define a bigger kernel window around the hit-point of the beam. That way we increase the chances to find a grid cell that matches the hit-point. Therefore, when a map is eligible for traversal, we apply the aforementioned procedure using the ICP algorithm and through

the likelihood field model we get a score which represents how much the current observation fits on the target map. That score is proportional to the probability that the target map is the traversal map. This scan matching technique heavily depends on the initialization. A good initialization pose, close to the real pose, is required or otherwise the ICP can be trapped into the first local minimum, yielding a useless solution [17].

4.5.2. UWB Beacon Transceiver

As we mentioned earlier, we also defined two UWB beacon transceivers with different ranging capabilities in order to create our hierarchical graph. Both these sensors are defined in the same way. In this section we present their implementation.

The beacon discovery procedure follows the SOGs method presented in section 2.4. The possible beacon's locations are represented by a set of 2D Points. Thus, in this case a map of landmarks is constructed. Each 2D point is represented by a vector holding its mean value and by a matrix holding its covariance. The estimate of the beacon location is updated using an EKF filter. The beacon's location is expressed according to the current's local map coordinate frame, which is the same frame as the current layer-0 node.

The particle's weight is calculated using the likelihood field model once more. For each particle, an expected range measurement for each beacon id is calculated applying the observation model, given the particle's sampled pose. Then, the difference between the expected range measurement and the actual range measurement is calculated. Applying Equation 4.3, where d is the difference in the range measurements and Q is the measurement covariance, a score is calculated and it is used in order to weight the particle.

In case that a beacon location is not yet discovered and multiple locations are still considered, Equation 4.3 is applied for all of these locations and the final score for the particular beacon is the summation of all scores. The final score for the particle is the product of all scores that are calculated for each beacon.

$$s = \exp \frac{-0.5d^2}{Q} \tag{4.3}$$

Finally, we specified that each time a new node is created or traversal to known node takes place, the location of the beacons of the previous local map that are also visible in the current local map is translated to the new map using the transform of the associated edge.

4.5.3. Creating Hierarchical Nodes

The final step to complete our proof of concept implementation consisted of defining the information maintained by every layer-1 and layer-2 node.

Layer-1 is associated to the UWB beacon transceiver of the short range and its produced map, whereas layer-2 is associated to the UWB beacon transceiver of the long range and its produced map. We decided to segment the map into Voronoi cells both in layer-2 and layer-1, based on the discovered beacon locations. Each node of layer-2 and layer-1 is characterized by a dominant beacon[2] and defines an area (Voronoi cell) in the map.

Next we specified the probability of the robot belonging to an hierarchical node of each layer, given the sampled pose and the map. This probability was introduced in Chapter 3 and it is denoted as $p(n_l^i | x_t^{[k]}, m)$. In our case we define the probability of belonging to a Voronoi cell, given the discovered locations of the beacons and the sampled pose of each particle, following the equation:

$$p(n_l^i | x_t^{[k]}, m) \approx \frac{1}{N} - \frac{1}{N} \frac{|x_t^{[k]} - x_i|}{\sum_i^N |x_t^{[k]} - x_i|} \tag{4.4}$$

where x_i stands for the discovered location of each beacon, $x_t^{[k]}$ is the sampled pose of the $k-th$ particle at time t and N is the total number of observed beacons at time t.

Given each particle's sampled pose and each particle's constructed map of beacons, Equation 4.4 is applied for each discovered beacon location. The beacon id that scores higher wins, thus the Voronoi

[2]The beacon that defines the current Voronoi cell

cell and consequently the node where the robot might be is decided. This practically means that each node of the layer-1 and layer-2 of the hierarchical graph is practically characterized by the beacon id of the constructed Voronoi cell.

In order to better explain the aforementioned procedure, we give an example. Let's assume the scenario depicted by Figure 4.4. The red dots represent discovered beacon locations (in this case they exist at the same coordinates for both particles' associate map), the black dot is the real pose of the robot, whereas the blue dot is a pose sampled by particle 1 and the green dot is a pose sampled by particle 2. The dashed line represents the splitting into two areas, based on the beacons. These areas represent layer 1 nodes.

Given Equation 4.4 and particle 1 sampled pose we calculate:

$p(n_1^1) \approx \frac{1}{2} - \frac{1}{2}\frac{9.5}{10.5+9.5} \approx 0.2625$

$p(n_1^2) \approx \frac{1}{2} - \frac{1}{2}\frac{10.5}{10.5+9.5} \approx 0.2375$

Given Equation 4.4 and particle 2 sampled pose we calculate:

$p(n_1^1) \approx \frac{1}{2} - \frac{1}{2}\frac{11}{11+9} \approx 0.225$

$p(n_1^2) \approx \frac{1}{2} - \frac{1}{2}\frac{9}{11+9} \approx 0.275$

Therefore in the associated graph of particle 1, node 1 is the winning node and in the associated graph of particle 2, node 2 is the winning node. Based on each sampled pose a different hypothesis for the hierarchical node is formulated. During the weighting process (when the actual range measurement is taken into account), the particles that sampled a pose closer to the real one will get a higher score and this will account for taking the right decision.

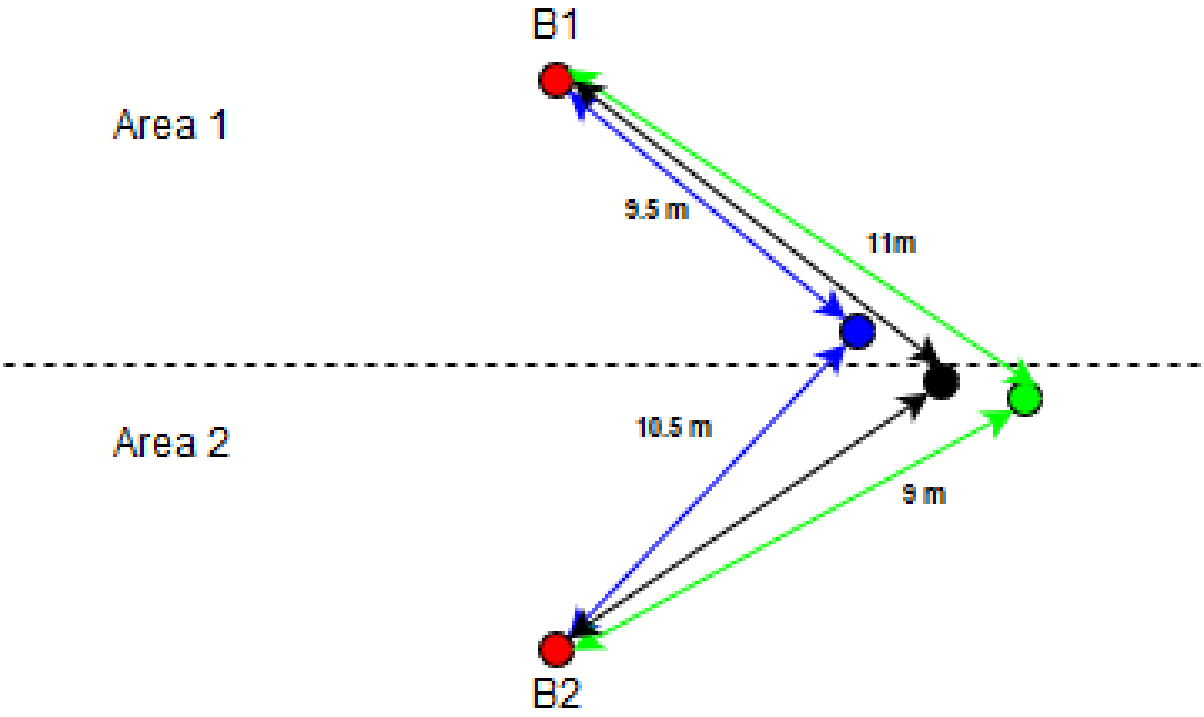


Figure 4.4: The large area is split into two sub-areas based on the discovered beacon locations (red dots). Each particle (blue and green dot) has sampled a different pose, trying to approximate the real pose of the robot (black dot). Based on each particle's sampled pose and the constructed map, a different decision on the area that the robot is at, is made.

5

Experiments

We have discussed the theoretical aspects of our framework, its architecture and our own implementation which was created mainly as proof of concept. We now present an evaluation of our proposed framework. The experiments performed, test our framework in terms of (a) loop closure, (b) execution time and (c) memory consumption. In the following sections we present our experimental setup and each one of the individual experiments. We aim to prove that our framework generates a high quality map, providing useful information for navigation, while it keeps the robot resources (CPU usage and memory usage) at low consumption levels.

5.1. Experimental Setup

We compare our hierarchical framework against a non-hierarchical one. The proof of concept implementation described in section 4.5 is used for this purpose. The non-hierarchical framework is emulated by discarding the UWB beacon information and limiting the number of layers in the hierarchical graph to one. This means that the non-hierarchical framework is a simple HMT SLAM technique.

We performed the experiments using the V-Rep simulation environment [18]. V-Rep is used in order to simulate real world scenarios of robots interacting with their environment or other robots. V-Rep runs on a physics engine which provides with real world properties, whereas each scenario is created by defining the behavior of all involved objects through scripts.

Our simulation environment consists of:

1. An area consisting of cuboids that create rooms and hallways.
2. Our robot model.
3. UWB beacon transceivers, spread throughout the whole area.

In order to create the simulation environment a series of steps was performed:

1. We decided on the robot model to use. We chose the Pioneer P3DX robot model which is provided by V-Rep. In order for the robot to perform in our desired way, we programmed its associated script using the LUA script language. Inside the Pioneer P3DX script we assigned a subscription to a ROS topic which publishes motion commands. The motion commands are given through the keyboard. Based on these commands the velocity of the robot's wheels is updated and the robot moves in the simulation environment. In the same script we also provided with the appropriate code for publishing to the odometry topic. Odometry messages are computed based on the robot's movement and they are continuously published to the odometry topic.
2. We decided on the laser scanner model to use. In this case we chose the Hokuyo URG-04LX-UG01 laser scanner model that is provided by V-Rep. We placed the laser scanner model inside the P3DX model as a component of the latter. Once more we had to provide the laser scanner model with a LUA script that defines its behavior. Our script practically gets the readings of the laser scanner sensor and publishes them to the appropriate ROS topic. The mapping algorithm subscribes to that topic in order to get the laser data.

3. We emulated the UWB beacon transceiver by using the transceiver model of V-Rep. We once more placed the transceiver model as a component of the P3DX model and we associated scripts to the transceiver model in order to specify its behavior. For these experiments we placed a transceiver model on the moving robot and we scattered a number of transceiver models in the simulation environment. The transceivers that are not placed on the robot transmit their location in the simulated world as global coordinates. The transceiver that is placed on the robot receives the coordinates of the rest scattered transceivers, as long as the robot is inside their range area and calculates the distance to each one of them. The calculated distance is distorted by applying Gaussian noise drawn from the $N(\mu = 0, \sigma^2 = 0.05)$. Then the model that is placed on the robot publishes the id of the beacons that it "hears" alongside the distance to them, in the appropriate ROS topic.

4. Finally, we had to create the simulation environment. In order to do so, we used a python script that generates cuboids in random locations inside the V-Rep environment. These cuboids create rooms and hallways of various size.

In Figure 5.1 our simulation environment is depicted. We can see the area which consists of cuboids, as well as our robot and the UWB beacon transceivers that are spread throughout the environment.

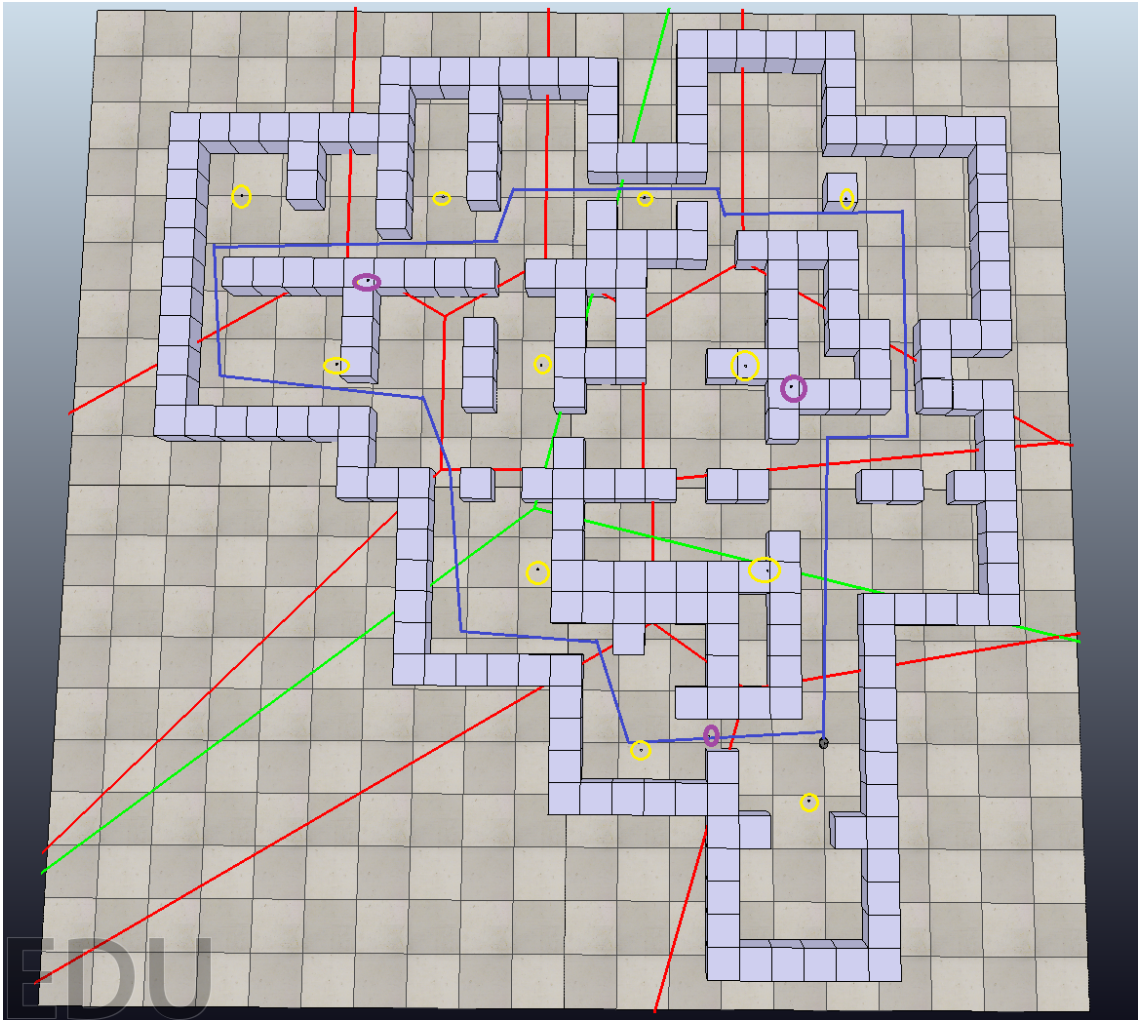


Figure 5.1: The simulation environment. The green lines represent the segmentation that occurs on layer 2, while the red lines represent the segmentation that occurs on layer 1. The blue line is the robot trajectory. The 11 scattered dots, circled in yellow color, inside the environment stand for the UWB beacon transceivers associated with layer-1. The 3 scattered dots, circled in purple are the UWB beacon transceivers associated with layer-2. The model of the robot is shown at the lower right part of the cuboid based environment.

5.2. Loop Closure Experiment

In the previous chapters we discussed that the fragmentation of the map into distinguishable local areas as well as the usage of sensors that provide with identifiable landmarks can be of great advantage when it comes to closing a loop. In order to verify this assumption, we performed a loop closing experiment that we will discuss in this section.

We used the experimental setup depicted on Figure 5.1. The green lines represent the segmentation on the map which occurs by layer 2, whereas the red lines represent the segmentation which occurs by the layer 1. The blue line is the robot trajectory, which is approximately 100 meters.

At first, we used a non hierarchical mapping scheme and the resulting map is depicted by Figure 5.2. It can be seen that at some point particles that sampled a pose wrongly (local map 11, local map 18 and local map 19) finally won, thus an inaccurate map was constructed.

Figure 5.2: Resulting map of the non-hierarchical scheme. When the robot left node 23, it should had traversed back to node 1 but instead it created node 24. Thus, the loop was not closed.

If loop closing had been successful, when the robot left node 23 and created node 24, it should have traversed back to node 1. However, when it calculated the transformation towards node 1 local coordinate frame and transformed its sampled pose to that reference frame taking into account the associated uncertainty, it concluded that this map is ”too far”, so it is not eligible for traversal. Thus our ICP scan matching technique with that node’s map was never performed. Therefore, the loop was never closed.

To evaluate if our framework could perform better, we used our hierarchical scheme of three layers in order to perform the same experiment. The resulting map is depicted on Figure 5.3. As we can see this time the loop is closed and the result is successful. In this case, the robot also identified that it returned back to the original Voronoi cell and took into account only those nodes that also belong to that cell (nodes 1 and 2).

Thus, the loop closure procedure is improved and the resulting map is more accurate. We next provide some insight of how this result was achieved for this particular scenario:

1. The usage of UWB beacons improved the quality of the map. UWB beacons are self identifiable, helping the robot to perform quality pose correction. The simulated environment consists of hallways and rooms that don't possess unique characteristics, therefore we can see the advantage of using UWB beacons in such areas.

2. The fact that the robot corrects its pose efficiently, leads to closing the loop implicitly in the global scope. As we can see in Figure 5.3 less nodes than before were created since the pose was estimated better (22 local maps). When the robot left node 22, it took into account node 1 as possible traversal candidate, since the translated pose into node 1 coordinate frame fell inside the node 1 boundaries. The ICP scan matching technique was also successful, given the correct estimated robot pose.

3. The segmentation of the global map into Voronoi cells, filtered out those local maps that didn't belong to the original Voronoi cell. Thus, planning was performed into a subarea of the global map and could finish in time. This is important for addressing real-time constraints. In a scenario that the robot moves really fast, traversing among local maps the mapping procedure should be efficient in terms of execution time. In the next section, we perform an execution time experiment and we better show this advantage.

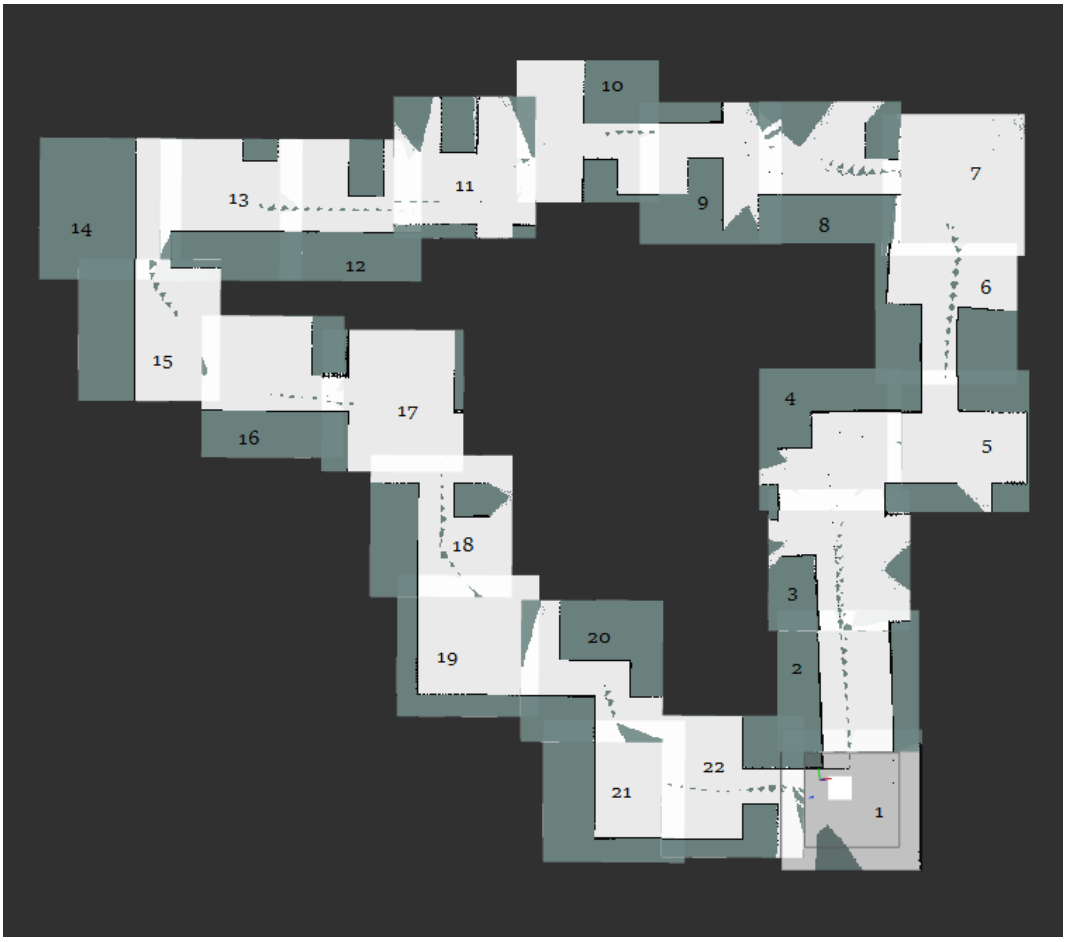


Figure 5.3: Resulting map of the hierarchical scheme. The loop is successfully closed. The robot returned back to its original position both in reality (Figure 5.1) and in the depicted resulting map.

5.3. Execution Time Experiment

The next series of experiments were meant to evaluate the main loop's execution time. The main loop of our framework consists of what we described in Figure 4.1; sampling pose for each particle, graph updating for each particle, weighting each particle, local map updating for each particle and resampling for the particle set.

In order to perform these experiments, we created a fake data-set of an already created graph. In other words, we loaded fake maps into the memory and we assumed that the robot had already followed a trajectory, when we started our simulation. Creating a fake data-set was done for both the non-hierarchical and the hierarchical case. It should be noted that throughout these experiments, when addressing the data-set of the hierarchical case, we created a 3 layer graph and we followed the principle that each node contains 50 nodes.

In the first experiment, we pre-loaded 100 layer-0 nodes for both the hierarchical and the non-hierarchical case. Then we started the simulation and we drove the robot forward for a specific distance until it created two more layer-0 nodes. Each time that main loop was executed, we measured its latency. Figure 5.4 depicts the result of the experiment and we see that there is almost no difference between the non-hierarchical and the hierarchical case. The number of layer-0 nodes is relatively small, therefore considering the hierarchical case, the hierarchical graph consists of few nodes, adding almost 0 overhead in execution time. Table 1 summarizes what is depicted in Figure 5.4.

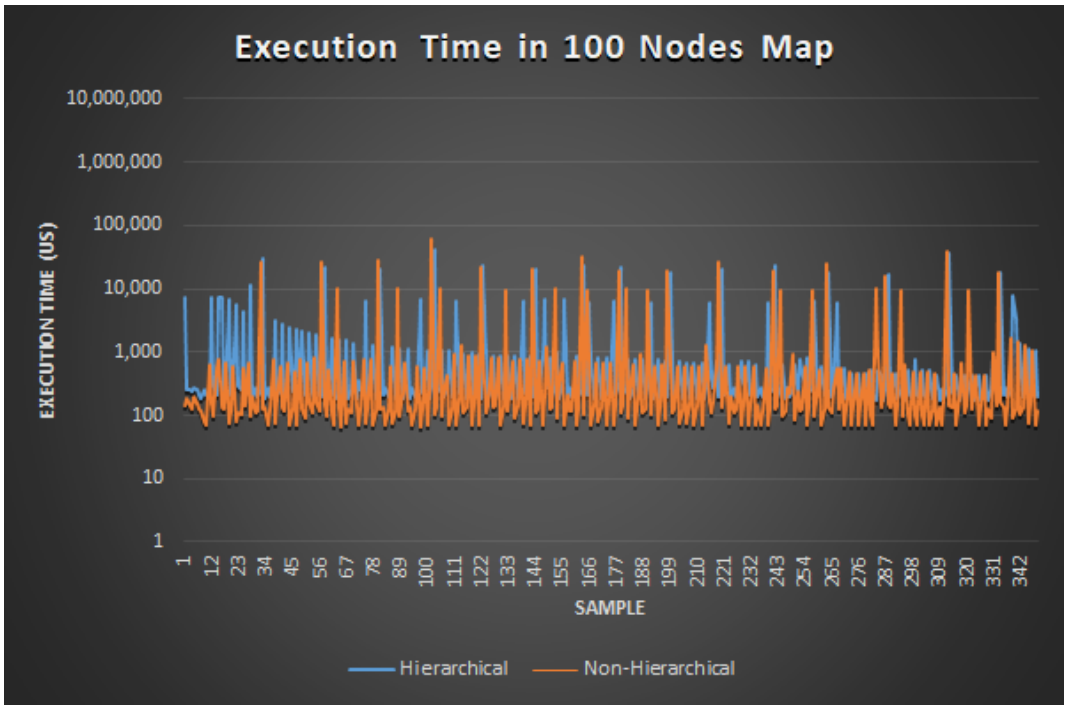


Figure 5.4: Main loop execution time comparison in 100 nodes map. In both the hierarchical and the non-hierarchical case the execution time is approximately the same. We can also see that in the hierarchical case, there is increased execution time at the beginning (samples 1 to 100) and it slowly decreases. This happens due to the beacon discovery process. At the beginning, multiple locations for the observed beacons are considered increasing the execution time. As the beacons are being discovered, the execution time decreases.

Table 1: Execution time in 100 nodes map

Case	Mean (us)	Standard Deviation	Max (us)	Min (us)
Hierarchical	1815.76	5126.8	42044	172
Non-Hierarchical	1747.08	5991.03	61912	66

In the second experiment, we pre-loaded 3000 layer-0 nodes into the memory and we followed the same procedure as before. In this case the result is depicted in Figure 5.5. Table 2 summarizes what is depicted in Figure 5.5.

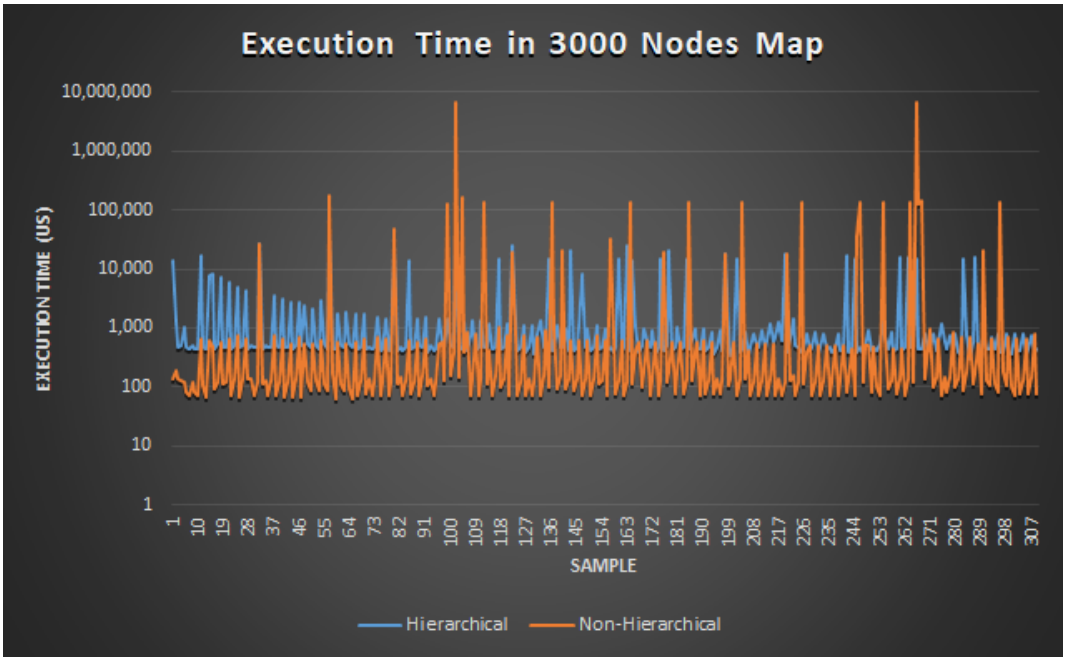


Figure 5.5: Main loop execution time comparison in 3000 nodes map. The hierarchical case has an overhead which is related to the fact that more than one layers are taken into account. However, the hierarchical case keeps the execution time spikes low, in contrast to the non-hierarchical case. In the non-hierarchical case when node traversal happens the execution time grows at almost 10 seconds.

In the hierarchical case there is an added overhead which is much more observable than in the 100 layer-0 nodes case. Specifically, the minimum execution time recorded for the hierarchical case is 303 us whereas the minimum execution time for the non-hierarchical case is 63 us. This happens since this time the 3000 layer-0 nodes are grouped in an hierarchical graph consisting of many hierarchical nodes that need to be checked. Even though the overhead is more than before, the difference is almost negligible.

The big difference occurs at the node traversal action. In the non-hierarchical case we see two large spikes that reach up to approximately 7 seconds, whereas in the hierarchical case these spikes remain low, reaching up to no more than ten milliseconds.

Table 2: Execution time in 3000 nodes map

Case	Mean (us)	Standard Deviation	Max (us)	Min (us)
Hierarchical	2454.6	5756.4	58812	303
Non-Hierarchical	52445.8	520504.6	6906936	63

Finally, we performed a third experiment comparing the execution time, only at the node traversal action part of the algorithm for both the hierarchical and the non-hierarchical case. We measured the time as we increased the number of layer-0 nodes and we obtained the results depicted in Figure 5.6.

We see that in the hierarchical case the curve follows a logarithmic growth to the number of nodes, whereas in the non-hierarchical case, it follows a linear growth to the number of nodes. Mainly, the traversal action consists of finding paths that hold the least uncertainty between nodes. It should be noted that this path finding procedure also occurs when it comes to navigating at a fully created map. Thus, we expect that there is also similar benefit, when it comes to navigation.

With these experiments we believe we have proved that our framework is better than non-hierarchical frameworks in terms of execution time, decreasing both the maximum value of execution time and the mean value of execution time sufficiently.

5.4. Memory Usage Experiment

Our last experiments are about the memory usage of our framework. We pre-loaded fake data-sets in both cases, varying from 100 to 10000 layer-0 nodes. In our first memory experiment all the associated maps are loaded into the memory and the results are presented in Table 3. The memory consumption

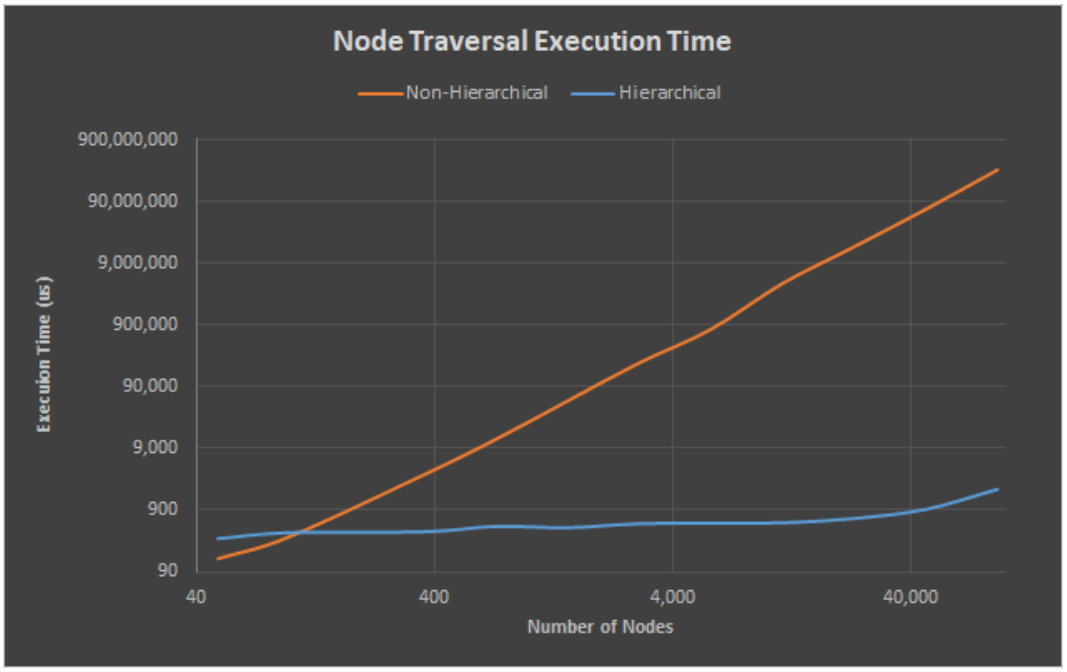


Figure 5.6: Node traversal execution time vs number of nodes. In the hierarchical case there is a logarithmic growth of execution time with respect to the number of nodes. In the non-hierarchical case there is a linear growth of execution time with respect to the number of nodes. (The graph follows a logarithmic scale in both y-axis and x-axis)

is almost the same in both cases. The hierarchical case adds a negligible overhead; in Table 3 we see that the ratio (memory consumption of hierarchical case to memory consumption of non-hierarchical case) is approximately 1.0.

Table 3: Memory Usage (All maps loaded)

Number of Nodes	Non-Hierarchical Case (Bytes)	Hierarchical Case (Bytes)	Ratio (Hier/Non-Hier)
97	14,434,727	14,436,097	1.00009491
361	45,825,325	45,846,597	1.0004641975
1441	174,341,063	174,422,459	1.000466878
2881	345,916,836	346,017,826	1.0002919488
5761	688,442,205	688,745,207	1.000440127

In order to verify that our framework indeed adds negligible overhead, we performed one more experiment where we implemented a scheme more favorable to the non-hierarchical case. We again pre-loaded fake data-sets for both cases, varying from 100 to 10000 layer-0 nodes. However, we deleted from memory any associated grid map or landmark map, actually keeping the graphs loaded into the memory. The result is again the same as before as it is also depicted in Table 4. The hierarchical case adds a small overhead but it is still negligible; ratio (memory consumption of hierarchical case to memory consumption of non-hierarchical case) is around 1.02.

Table 4: Memory Usage (No maps loaded)

Number of Nodes	Non-Hierarchical Case (Bytes)	Hierarchical Case (Bytes)	Ratio (Hier/Non-Hier)
97	3,152,366	3,217,192	1.0205642365
361	3,836,332	3,913,732	1.0201755218
1441	6,734,030	6,883,394	1.0221804774
2881	10,819,083	11,057,161	1.022005377
5761	18,363,012	18,741,342	1.0206028292

Therefore, we compared the two extreme cases (all maps loaded vs no maps loaded) and we conclude that the memory consumption is not becoming significantly worse when using an hierarchical

scheme. We have to note that the second scenario is not a realistic one, because the robot needs to have loaded some of the local maps into the memory in order to perform its tasks. However, this experiment proves that the memory overhead added by our framework can be considered negligible. Memory consumption could even get better, as we discuss in the following two last Chapters.

Finally, in Figure 5.7 the relationship of the memory overhead with respect to the number of nodes is shown. It can be seen that the overhead increases linearly as the area grows larger, but this is not significant if one takes into account the total memory usage. For example in the 5761 nodes case, the added overhead is around 380KB whereas the total memory consumption is around 18.7MB.

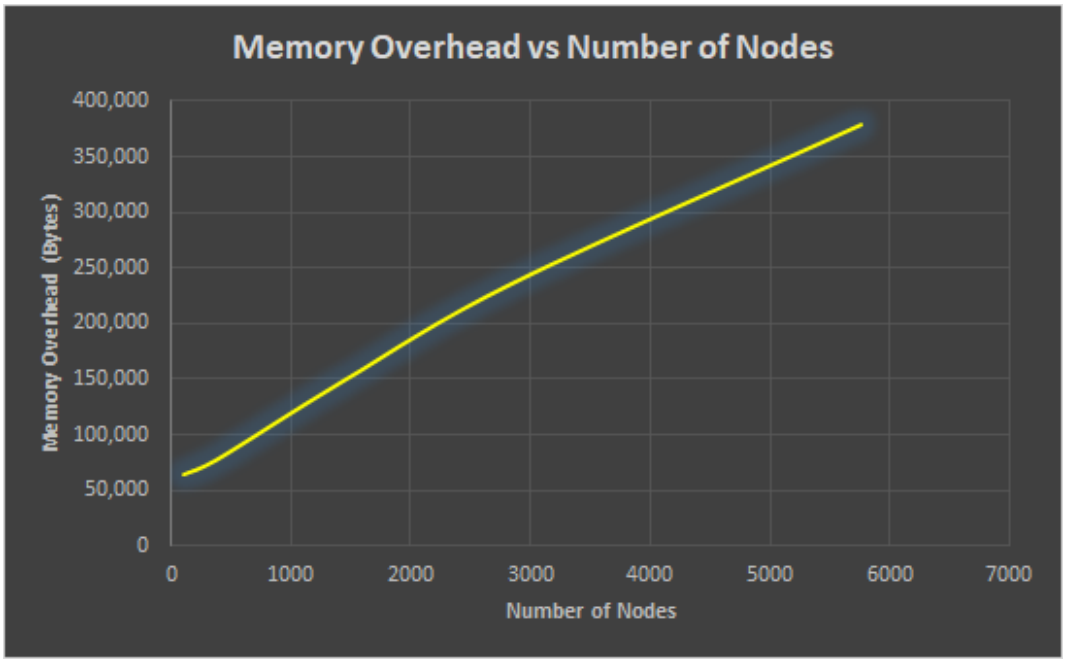


Figure 5.7: Memory overhead with respect to the number of nodes. This overhead is recorded in the extreme scenario of "no maps loaded", which is favorable for the non-hierarchical case. The total memory overhead added by our framework is captured.

6

Conclusion

In this thesis we discussed our hierarchical mapping framework. Our framework intends to map vast areas, where existing SLAM methods were found lacking. Specifically, in wide areas, even HMT-SLAM methods are demanding with respect to the robot's resources (CPU usage and memory consumption). In addition, existing SLAM methods fail to accurately estimate the original robot trajectory as it grows in length. This phenomenon is even more prominent inside environments that don't pose unique characteristics. Finally, in large areas the produced map of existing HMT-SLAM methods can become a huge interconnected graph, where path planning techniques which are also used for navigational purposes get inefficient. Having presented our framework's theoretical model, its implementation as well as the conducted experiments, we conclude that:

1. As far as the *main loop's execution time* is concerned, our framework adds a small overhead, when compared to non-hierarchical mapping schemes. This is reasonable since it uses multiple sensors in order to update the map and it maintains additional layers in the topological graph. However, the overhead is negligible, if the construction of the hierarchical graph is chosen according to the mapping area. For example, it is not wise to add 10 layers when mapping a $20m^2$ room, since the execution time would increase causing no benefit. In our experiments we showed that in the case of 3000 layer-0 nodes the overhead added was 250 us.

2. As far as the *execution time of the traversal action*, when topological loop closure happens, our framework provides with encouraging results. As the mapping environment grows large and the number of local nodes increases, the execution time of the traversal action grows logarithmically to the number of nodes. Compared to non-hierarchical schemes where the traversal's action execution time grows linearly to the number of nodes, this is a significant improvement.

3. As far as the *memory usage* is concerned, our framework adds small overhead when it is compared to a non-hierarchical mapping scheme. The overhead added is less than 2% in our experiments. However, our framework adds extra information on top of the local maps which could be used for optimizing the memory consumption as discussed in Chapter 7.

4. As far as the *quality of the produced map* is concerned, our framework provides with encouranging results. Adding extra recognizable features during the mapping procedure, accounts for a better estimated and corrected robot pose. Consequently, a better estimated and corrected robot pose yields a better quality map.

5. As far as the ultimate goal of *navigating in the constructed map* is concerned, our framework is efficient and fast. A very important part of navigation is path planning. Path planning also happens during the layer-0 node traversal procedure. We showed that during the layer-0 node traversal procedure our framework performs faster than non-hierarchical schemes, so we can assume that similar results are going to be obtained, during navigation as well.

6.1. Contributions

Our main contributions can be summarized as follows:

1. We provided with a framework which is a combination of Hybrid Metric Topological (HMT) SLAM and Range-Only SLAM (RO-SLAM). HMT SLAM provides accurate local maps, while the global map's inconsistencies are captured by the graph. As the area grows larger these inconsistencies are getting worse. However, RO-SLAM provides with identifiable landmarks that can be used for quality pose correction and thus reducing the inconsistencies on both the local and global scope. Combining these methods accounts for a better quality map both locally and globally.

2. We inserted the notion of hierarchy in the topological graph, by taking into account multiple information from multiple sensors, achieving encouraging results when it comes to utilizing the robot's resources (CPU usage and memory consumption).

3. We implemented our framework in a ROS compatible package. In contrast to different SLAM implementations that are intended to be used with specific sensors, our implemented framework is highly configurable. Per one's implementation the required code for any additional sensor must be provided in order to define its function. The framework can then manage the output of the sensors' implementation and produce the hierarchically structured global map.

7

Future Work

In this thesis we designed and implemented an hierarchical mapping framework. We proved that our framework is scalable and thus a suitable solution for mapping large-scale environments. However, there is still room for research and improvements.

1. Our framework ultimately aims to facilitate the navigation process in the already built maps. In order to achieve optimal navigation it could exploit the inherent arrangement of local maps into an hierarchical graph. Using this graph, navigation could be performed starting from the highest in hierarchy layer and moving to the lower layers, planning on less nodes and finding routes much faster than existing methods. As the navigation needs were always considered during this thesis, our mapping algorithm already creates the appropriate structures that can be used in navigation. The next step should be the implementation of a complete navigation/localization/mapping suite that works on the maps that our mapping framework produces.

2. In this work we evaluated our framework by performing simulations. During our simulations we approximated real world conditions modelling our robot and its sensors and using the appropriate error models for the acquisition of sensor data. The results that we obtained are more than encouraging. However, our framework was not verified against real world conditions. A real world experiment would be the final assessment for our framework, guaranteeing its correct functionality.

3. One necessary extension for our framework and matter of future research lies in the field of continuous and dynamic mapping. The current implementation, assumes a static map. However, in real world conditions the environment changes over time. The robot should be able to update each local map upon revisiting it. The robot should be able to recognize the dynamic elements of the environment and keep track of them in order to efficiently update each map, when necessary. Practically the mapping process should be continuously running and a mechanism for updating the local maps has to be implemented. The fact that our framework is lightweight is an advantage to executing the mapping process in parallel with navigation. The localization part could be shared between the mapping process and the navigation process, whereas the paths that are calculated during the mapping process could be cached and used for navigation. Currently our framework works assuming that all the associated sensors are constantly working and providing data. However, in real world conditions, hardware could get faulty or could stop working. The fact that additional sensors are used should be of advantage in such a case. The framework should be able to continue its mapping process using the rest sensors and changing the graph structure, being fault tolerant and re-configurable.

4. The number of local maps created by hybrid metric-topological algorithms increase as the mapping area grows. Therefore, a mechanism for storing and loading the maps into memory should be created. In plain HMT techniques one metric for creating such a mechanism could be the distance. The robot could load into memory the maps that it currently needs or it would need in the

imminent future based on how far from its current position those maps are. The distance, however, is derived based on the estimated trajectory and in large-scale environments this estimation is not trustworthy. Our framework structures the maps hierarchically based on the sensor's observations. This could be of help into creating such a mechanism for memory management. The robot could load and store the maps based on its observed surroundings. That would decrease the memory usage of the algorithm to a great extent.

5. Our framework is implemented using the Rao Blackwellized Particle Filter SLAM technique. RBPF follows the filter based approach to SLAM, using a Particle Filter and Extended Kalman Filters in order to estimate the map. RBPF is a widely used technique which can create accurate and low error results while in the same time it can be efficient concerning the robot's resources. However in recent years there has been improvement in the Graph SLAM field. In Graph SLAM the problem of simultaneous localization and mapping is treated as a nonlinear parameter optimization problem. Specifically, factor graphs are used setting constraints among recorded robot poses and robot observations and optimizing the graph based on these constraints. Results are more than encouraging in the field of Graph SLAM making it a more modern SLAM method [19]. One possible future version of our framework could be implemented based on the Graph SLAM technique.

6. Our framework is configurable and can be used with different sensors and with arbitrary number of graph layers. At this point the only sensors that are implemented are those described in our proof of concept implementation, discussed in section 4.5. The framework should be further expanded by implementing the functionality of some commonly used sensors (e.g. GPS, camera etc).

A

Basic Probabilistic Notions

In this section we present some basic probabilistic notions as well as basic probability properties that are of importance in probabilistic robotics [2]. Let's denote as X a random variable and as x a specific value that X might get. We then can express the probability that random variable X takes the value x by:

$$p(X = x) \tag{A.1}$$

The aforementioned notation can be simplified to $p(x)$ instead of $p(X = x)$. Random variables can possess probability density functions (PDFs). An example of a PDF is the normal distribution described as:

$$p(x) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp(-\frac{(x-\mu)^2}{2\sigma^2}) \tag{A.2}$$

In the case when a random variable takes discrete values, it holds that:

$$\sum_x p(x) = 1 \tag{A.3}$$

In the case when a random variable takes continuous values, it holds that:

$$\int p(x)dx = 1 \tag{A.4}$$

The joint distribution of two random variables X and Y is given by

$$p(x, y) = p(X = x \; and \; Y = y) \tag{A.5}$$

Equation A.5 can be further expanded by the following equation, which is called the multiplication rule.

$$p(x, y) = p(x|y)p(y) \tag{A.6}$$

The leftmost term of the right hand side of Equation A.6 is called conditional probability. Sometimes, a random variable may carries information about an other random variable. If we would like to know the probability that X's value is x conditioned on the fact that Y's value is y we use the aforementioned notation.

One important rule that relates the conditional probability to its inverse is the Bayes rule. Bayes rule is often used in probabilistic robotics and is of fundamental value in order to estimate different desired variables. Equation A.7 describes this rule.

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{A.7}$$

Finally the Total Probability Theorem, which expresses the total probability of an outcome which can be realized via several distinct events is given by Equation A.8 for the discrete case and by Equation A.9 for the continuous case. This formula relates the marginal probabilities to conditional probabilities.

$$p(x) = \sum p(x|y)p(y) \tag{A.8}$$

$$p(a) = \int p(x|y)p(y)dy \tag{A.9}$$

B

The Bayes Filter

A key concept in probabilistic robotics is the belief. The belief represents the robot's hypothesis of the environment. The robot cannot directly measure the state of the environment but it can formulate an assumption of what that state might be. For example, the pose of the robot at time t might be $x = 22.5, y = 30.5, \theta = 28°$ in a given 2D global coordinate frame. However, the robot cannot surely know the pose. Instead it infers its pose from data.

The belief is represented through conditional probability distributions. Each plausible hypothesis of the true state is assigned with a probability, which is given by the belief distribution. Belief distributions are posterior probabilities over state variables conditioned on the input data. A belief distribution over a state variable x_t is denoted as $bel(x_t)$. Equation B.1 shows the belief distribution over the state x_t given the measurements $z_{1:t}$ and the controls $u_{1:t}$.

$$bel(x_t) = p(x_t | z_{1:t}, u_{1:t}) \tag{B.1}$$

The most basic and general algorithm that calculates beliefs is the Bayes Filter algorithm. The Bayes Filter calculates the belief distribution given measurement and control data. Figure B.1 depicts the algorithmic procedure of the Bayes Filter algorithm. The Bayes Filter algorithm is executed recursively. At each time step the previous belief $bel(x_{t-1})$, the most recent measurement z_t and the most recent control u_t are fed as input to the algorithm. The output at each time step is the current belief $bel(x_t)$. The Bayes Filter algorithm possesses two main steps; the control update (prediction) and the measurement update.

1: **Algorithm Bayes_filter**$(bel(x_{t-1}), u_t, z_t)$:
2: for all x_t do
3: $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, dx_{t-1}$
4: $bel(x_t) = \eta\, p(z_t \mid x_t)\, \overline{bel}(x_t)$
5: endfor
6: return $bel(x_t)$

Figure B.1: The Bayes Filter algorithm for estimating the belief distribution. Line 3 depicts the prediction whereas line 4 depicts the measurement update [2].

During the control update step Bayes Filter algorithm incorporates the latest control u_t. The output of this step is a predicted belief $bel(x_t)$. The predicted belief is the integral of the product of two

distributions; the previous belief $bel(x_{t-1})$ and the probability that state x_t is reached given the previous state x_{t-1} and the control u_t.

During the measurement update step the final belief $bel(x_t)$ is calculated. In this step the latest measurement z_t is incorporated. The predicted belief is multiplied by the probability that measurement z_t occurs, given the predicted state x_t. This product is not a probability because it cannot integrate to 1. Therefore, the aforementioned product is further multiplied by a normalizing constant η.

Finally, in order to perform the calculations one should provide with the initial belief $bel(x_0)$. The initial belief could be a point mass distribution in the case that the initial state is totally known or it could be a uniform distribution over the domain of x_0 in the case that the initial state is totally unknown or it could be a non-uniform distribution in the case that the initial state is partially known.

The Bayes Filter algorithm can be implemented in the previously discussed form and it can be used for simple estimation problems. Specifically, the integration in line 3 should be carried out, alongside the multiplication in line 4 in closed form. In any other case, there should be a restriction of the state space, so that the integral of line 3 sums up to 1 [2].

C

The Particle Filter

The particle filter constitutes a non-parametric implementation of the Bayes filter. The idea of particle filter is to represent the posterior belief distribution $bel(x_t)$ by a set of random state samples drawn from this distribution. The samples of the posterior distribution are called particles. Each particle is a hypothesis of the true state of the world.

The belief $bel(x_t)$ is approximated by a set of weighted particles; $X_t = \{x^{[j]}, w^{[j]}\}_{j=1,2,..M}$. Here X_t is the particle set at time t, $x^{[j]}$ is the $j-th$ particle's state hypothesis, $w^{[j]}$ represents the $j-th$ particle's weight and M is the total number of particles. The weight of the particle is a metric that indicates the plausibility of the particle's assumption. The particle's weight is derived by the represented distribution. The amount of particles signifies the level of approximation of the distribution. Figure C.1 depicts the particle filter representation of a Gaussian distribution. Figure C.2 depicts the particle filter representation of an arbitrary distribution. The latter Figure depicts that the particle set can take on different distributional shapes and consequently represent a broader space of distributions.

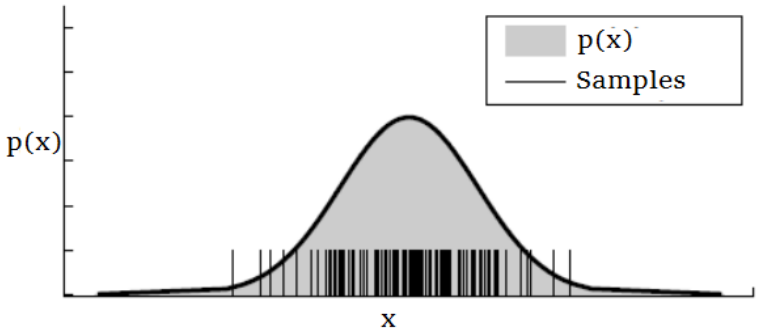


Figure C.1: Particle filter representation of a Gaussian distribution. The distribution is approximated by particles. The area where the particle population is dense depicts high probability of occurrence [2].

Figure C.3 depicts the algorithmic idea of the particle filter. At each time step the belief is calculated, given the previous belief, the control and the measurement. The algorithm firstly formulates a prediction about the state. The belief's prediction is represented by a temporary set of particles, denoted as X_t. Then, the posterior belief is approximated given the previously constructed temporary set.

Specifically, given the previously constructed particle set X_{t-1}, for each of the M particles that the set consists of, the following steps are involved:

1. Firstly, a hypothetical state is generated based on the particle's previous state and the control. This happens by sampling from the state transition model $p(x_t|x_{t-1}, u_t)$. In order to implement this step, one should be able to sample from the state transition model.

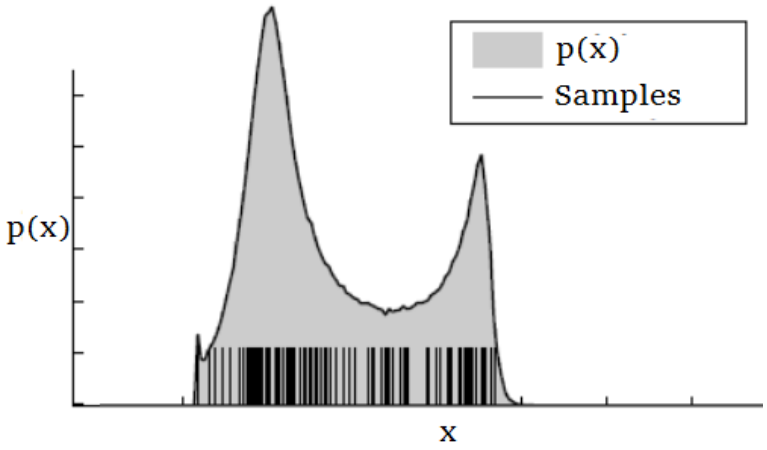


Figure C.2: Particle filter representation of an arbitrary distribution. The distribution is approximated by particles. The area where the particle population is dense depicts high probability of occurrence. This figure also depicts the ability of the particle set to take on different distributional shapes [2].

2. Secondly, the weight of the particle is calculated. In this step the measurement is incorporated. The weight represents the probability that the measurement z_t occurs, given each state hypothesis of each particle.

3. Thirdly, the particle is added to the temporary set X_t.

The temporary set X_t, represents the predicted belief of the state. While the measurement was taken into account in order to calculate the weight, it has not yet take effect into updating the prediction. The update of the prediction happens at the resampling step. During the resampling step, M particles are drawn with replacement out of the temporary set X_t and they are placed to the new set X_t. The probability of drawing a particle is proportional to the particle's weight. The newly constructed set approximates the belief at time t.

The non-parametric representation actually approximates the original distribution, but it poses the advantage of representing a broader space of distributions. One additional advantage of particle filter is the ability to model non-linear transformations of the random variables [2], [3].

1: **Algorithm Particle_filter**($\mathcal{X}_{t-1}, u_t, z_t$):
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: *for* $m = 1$ *to* M *do*
4: *sample* $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: *endfor*
8: *for* $m = 1$ *to* M *do*
9: *draw* i *with probability* $\propto w_t^{[i]}$
10: *add* $x_t^{[i]}$ *to* $\mathcal{X}_t$
11: *endfor*
12: *return* $\mathcal{X}_t$

Figure C.3: The particle filter algorithm for estimating the belief distribution. Lines 3 to 7 concern the formulation of the predicted belief, whereas lines 8 to 11 concern the prediction update [2].

D

The Robot Operating System (ROS)

The Robot Operating System (ROS) is an open source meta-operating system that is used in order to create robotic applications. It consists of a set of software libraries and tools that facilitate the building of robotic applications. ROS functions as any operating system providing with services, such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management. In addition, it provides with tools and libraries for obtaining, building, writing and running code across multiple computer platforms.

ROS comes along with a number of concepts that describe its function. The ROS concepts can be divided in three different levels: the ROS computation graph level, the ROS filesystem level and the ROS community level. In this Appendix we provide with a short description of the concept contained in the ROS computation graph level, while we shortly mention the concepts on the ROS filesystem level and the ROS community level. For more information about ROS, one can visit the official website [20], where various tutorials and examples exist.

The ROS computation graph is a peer-to-peer network of ROS processes which are used in order to process data. Figure D.1 depicts the basic structure of the ROS communication graph. In the ROS computation graph level, the main concepts which are involved are the following:

1. Nodes: The nodes are processes which are used in order to perform computations. In a robot control system there exist many nodes that are running. For example, one node may control the wheel motors, one other node may perform mapping, one other node may control an ultrasonic sensor etc.

2. Master: The ROS master is necessary, so that nodes can find each other, exchange messages and invoke services. It provides with name registration and lookup to the rest of the computation graph.

3. Messages: Messages are used for communication between nodes. Messages are structures of data, that carry information for a specific functionality.

4. Topics: Topics are used in order to perform communication among the nodes. Nodes can either publish or subscribe to a ROS topic. For example, a node that performs mapping is subscribed to the odometry topic receiving odometry messages. In the same time the motion sensor is publishing odometry information on the odometry topic. There can exist multiple publisher and multiple subscribers to a ROS topic. In addition, a ROS node can be subscribed to many topics as well as publish to many topics.

5. Services: In contrast to the publish/subscribe policy of ROS topics, ROS services follow a request/reply policy. One node can provide a service and reply back to another node, upon request.

6. Bags: Bags are a format for recording and playing ROS message data. Sensor data can be stored and played back. This facilitates the procedure of building and testing algorithms.

The basic concepts used on the ROS filesystem level are:

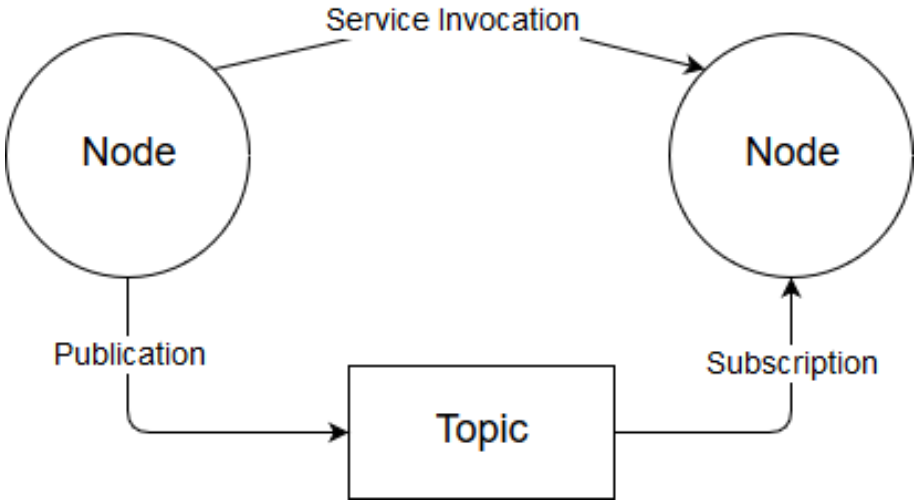


Figure D.1: The basic structure of the ROS communication graph. ROS nodes communicate to each other either through publishing/subscribing to ROS topics or through service invocation to ROS services.

1. Packages
2. Meta-packages
3. Package manifests
4. Repositories
5. Message types
6. Service types

The basic concepts used on the ROS community level are:

1. Distributions
2. Repositiories
3. The ROS Wiki
4. Mailing lists

Bibliography

[1] M. Montemerlo, *FastSLAM: A factored solution to the Simultaneous Localization and Mapping Problem With Uknown Data Association*, Ph.D. thesis, The Robotics Institute Carnegie Mellon University (2003).

[2] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics* (MIT Press, 2005).

[3] C. Stachniss, *Course material for slam course of university of freiburg,* .

[4] Howard, Parker, and Sukhatme, *The sdr experience: Experiments with a large-scale heterogeneous mobile robot team,* .

[5] M. Montemerlo and S. Thrun, *Fastslam 2.0, fastslam: A scalable method for the simultaneous localization and mapping problem,* , 63 (2007).

[6] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, *Fastslam a factored solution to the simultaneous localization and mapping problem,* Aaai/iaai , 593 (2002).

[7] G. Grisetti, C. Stachniss, and W. Burgard, *Improved techniques for grid mapping with rao-blackwellized particle filters,* IEEE Transactions on Robotics , 34 (2007).

[8] *Metro map of rotterdam,* .

[9] J. T. P. Pinies, *Large-scale slam building conditionally independent local maps: Application to monocular vision,* IEEE Transactions on Robotics **24**, 1094 (2008).

[10] T. kyeong Lee, S. Lee, and S. young Oh, *A hierarchical rbpf slam for mobile robot coverage in indoor environments,* Intelligent Robots and Systems (IROS) , 841 (2011).

[11] D. Schleicher, L. M. Bergasa, M. Ocana, R. Barea, and M. E. Lopez, *Real-time hierarchical outdoor slam based on stereovision and gps fusion,* IEEE Transactions on Intelligent Transportation Systems , 440 (2009).

[12] J.-L. Blanco, J. A. F. Madrigal, and J. Gonzalez, *A new approach for large-scale localization and mapping: Hybrid metric-topological slam,* Proceedings 2007 IEEE International Conference on Robotics and Automation , 2061 (2007).

[13] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller, *An atlas framework for scalable mapping,* IEEE International Conference on Robotics and Automation , 1899 (2003).

[14] R. Roozendaal, *Local Accuracy in Global Uncertainty*, Master's thesis, TU Delft (2016).

[15] E. Olson, J. J. Leonard, and S. Teller, *Robust range-only beacon localization,* IEEE Journal of Oceanic Engineering , 949 (2006).

[16] J. L. Blanco, J. A. F. Madrigal, and J. Gonzalez, *Efficient probabilistic range-only slam,* Intelligent Robots and Systems , 1017 (2008).

[17] P. Besl and N. D. McKay, *A method for registration of 3-d shapes,* IEEE Transactions on Pattern Analysis and Machine Intelligence , 239 (1992).

[18] *V-rep official website,* .

[19] W. Hess, D. Kohler, H. Rapp, and D. Andor, *Real-time loop closure in 2d lidar slam,* IEEE International Conference in Robotics and Automation (ICRA) , 1271–1278 (2016).

[20] *Ros official website,* .