Robot SLAM: Simultaneously Localizing a Robot and Mapping its Environment

Smruti Panigrahi, Ph.D.
Udacity Robotics Nandegree

***Abstract*—In this paper, a Gazebo/RViz simulation of Simultaneous Localization and Mapping (SLAM) of an environment using a mobile robot is presented. The robot model is created using Gazebo and is equipped with a depth-camera and a laser range finder. The task of the robot is to map an unknown environment using GraphSLAM algorithm. In particular, special type of Graph-SLAM called Real-time (RTAB) is used. RTAB-Mapping ROS package is used to map an environment. Two different environments are created using Gazebo to compare the performance between the mapping of two different environments using the RTAB-Mapping package. In addition, the effect of tuning different parameters that impact the accuracy of the mapped environment is discussed.**

Keywords—Robot Simulation, FastSLAM, Graph SLAM, Particle Filters, Monte Carlo Localization, ROS.

I. Introduction

Accurate localization is a challenging problem in autonomous vehicles and robotics applications. A lot of care is needed when designing a localization algorithm that performs robustly. Various parameters can affect how well the robot can localize itself in an environment. Robot localization mainly consists of its position coordinate and orientation angle and how well it correlates to the map. For wheeled robots, localization is achieved by analyzing sensor information such as Camera and LiDAR as well as wheel encoders. Various factors can affect the localization of a robot, such as sensor noise and environment noise. In order to overcome these challenges, robust filtering techniques and probabilistic models must be used. For the localization purpose, we investigate two state of the art algorithms based on Kalman filters and particle filters [1, 2].

As a roboticist or a robotics software engineer, building your own robots is a very valuable skill and offers a lot of experience in solving problems in the domain. Often, there are limitations around building your own robot for a specific task, especially related to available resources or costs.

That's where simulation environments are quite beneficial. Not only there is freedom over what kind of robot one can build, but also get to experiment and test different scenarios with relative ease and at a faster pace. For example, one can have a simulated drone to take in camera data for obstacle avoidance in a simulated city, without worrying about it crashing into a building!

In an unknown environment, it is extremely difficult for a robot to localize itself. Hence, simultaneously localizing and mapping the environment is essential for the robot to navigate in that environment. This problem is referred to as SLAM (simultaneous localization and mapping) or CLAM (concurrent localization and mapping). In this project a detailed comparison between various SLAM techniques is presented. A Graph-SLAM simulation using RTAB-Map is performed for a robot with no prior knowledge of its environment in order to map the environment. A custom indoor environment is created in Gazebo order to perform the mapping.

II. Background

While solving localization problems, it is assumed that map is known and the robot poses are estimated. Whereas in solving mapping problems, the exact robot poses are provided and the map of the environment is estimated. Combining the knowledge from both localization and mapping, one can attempt to solve the most fundamental problem in Robotics. In SLAM, the environment is mapped, given the noisy measurements and the robot is localized relative to its own map given the controls. This makes it a much more difficult problem than the localization only or mapping only problems, since both the map and the poses are unknown. In real-world environments, one would primarily be faced with SLAM problems it would be essential to estimate the map and localize the robot concurrently.

An example of a robot solving a SLAM problem is a Robotic vacuum cleaner that uses the measurements provided by its laser finder sensors and the data from the encoders to estimate the map and localize itself relative to it.

The inputs for a SLAM problem are the measurements and controls, and the output is a map along with the trajectory. Below (Fig. 1) is the graphical representation of a SLAM problem. Given the measurements z, and the controls u, the posterior map m along with the poses x is solved.

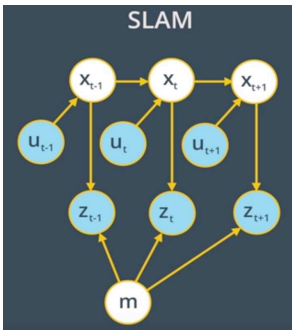

Fig. 1: SLAM graphical representation

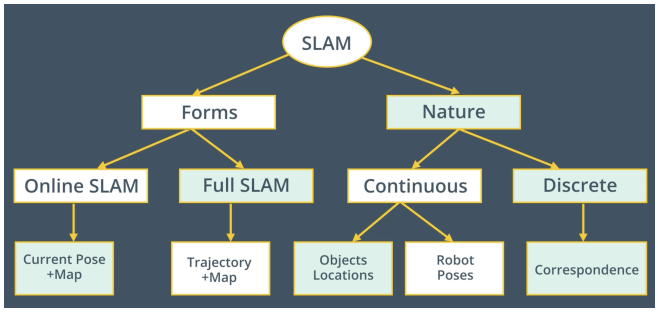

Fig. 2: Various forms and natures of SLAM

The SLAM problem comes in two different "forms", online SLAM and full SLAM (Fig. 2). Both forms are important in Robotics and have different application areas. The online SLAM problem computes a posterior over the current pose along with the map and the full SLAM problem computes a posterior over the entire path along with the map. Apart from different "forms" of SLAM the second key feature of the SLAM problem relates to its "nature". SLAM problems generally have a continuous and a discrete element as shown in Fig. 2 and is discussed in details in later sections.

A. *Online SLAM*

While solving the online SLAM problem, the current pose and map are estimated, given robot's current measurements and controls.

- At time t-1, the robot estimates its current pose x_{t-1} and the map m, given its current measurements z_{t-1} and controls u_{t-1}.
- At time t, the robot estimates its new pose x_t and the map m, given only its current measurements z_t and controls u_t. Notice how the previous measurements and controls are not taken into consideration when computing the new estimate of the pose and the map.
- At time $t+1$, the robot estimates its current pose x_{t+1} and the map m, given the measurements z_{t+1} and controls u_{t+1}. Thus with the online SLAM problem, we solve instantaneous poses independently of previous measurements and controls.

This problem can be modeled with the probability equation $p(x_t\ ,\ m,\ c_t\ |\ z_{1:t}\ ,\ u_{1:t})$ where the posterior represented by the instantaneous pose x_t, the map m, and the correspondence with previous time step c_t are estimated, given the measurements $z_{1:t}$ and controls $u_{1:t}$. Thus, with online SLAM one estimates the variables that occur at time t only (Fig. 3).

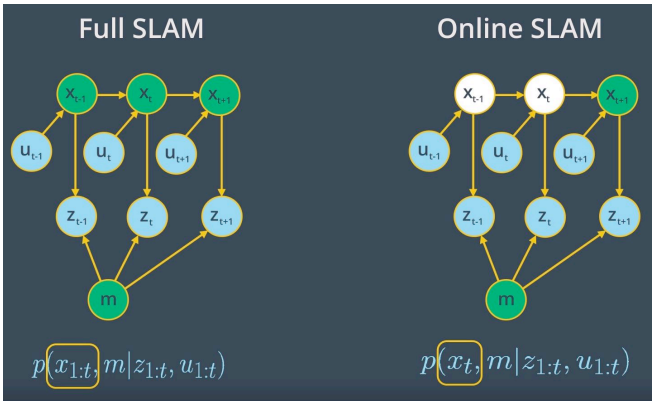

Fig. 3: Graphical representation of online SLAM and full SLAM.

B. *Full SLAM*

While solving the full SLAM problem, the entire path up to time t instead of an instantaneous pose is estimated, given all the measurements and controls.

- At time t-1, the robot estimates its current pose x_{t-1} and the map m, given its current measurements z_{t-1} and controls u_{t-1}.
- At time t, the robot estimates the entire path its new pose $x_{t-1:t}$ and the map m, given all the measurements $z_{t-1:t}$ and controls $u_{t-1:t}$.
- At time $t+1$, the robot estimates the entire path $x_{t-1:t+1}$ and the map m, given all the measurements $z_{t-1:t+1}$ and controls $u_{t-1:t+1}$.

This problem can be modeled with the probability equation $p(x_{1:t}\ ,\ m,\ c_{1:t}\ |\ z_{1:t}\ ,\ u_{1:t})$, where the posterior represented by the robot's entire path or trajectory $x_{1:t}$, the map m, and all of the correspondence values $c_{1:t}$ are estimated, given all the measurements $z_{1:t}$ and controls $u_{1:t}$. Thus, with full SLAM problem one estimates all the variables that occur throughout the robot travel time (Fig. 3).

C. *Adding Correspondece to Posterior*

The continuous component of the SLAM problem addresses the solution for a robot continuously collecting odometry information to estimate the robot poses and continuously sensing the environment to estimate the location of the object or landmark. Thus, both robots poses and object location are continuous aspects of the SLAM problem.

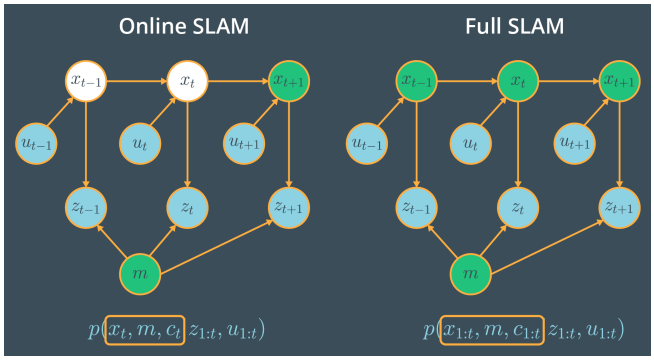

Fig. 4: Online SLAM vs Full SLAM with added correspondence.

In the discrete component of the SLAM, the robot has to identify if a relation exists between any newly detected and previously detected objects. As the robot continuously senses the environment to estimate the location of the objects, when doing so SLAM algorithms have to identify if a relation exists between any newly detected objects and previously detected ones. This helps the robot understand if it has been in this same location before. At each moment, the robot has to answer the question, "Have I been here before?". The answer to this question is binary - either yes or no - and that's what makes the relation between objects a discrete component of the SLAM problem. This discrete relation between objects is known by correspondence.

Since the correspondence is an important aspect of the SLAM, one must explicitly include this in the estimation problem, i.e. the posteriors should include the correspondence values in both online and full SLAM problem (Fig. 4). Hence, for the online SLAM problem, the posterior is estimated over the current pose, map and the current correspondence values, given the measurements and controls at time t. For full SLAM problem the posterior is estimated over the entire path, map and all of the correspondence values, given the measurements and controls up to time t. The advantage of adding the correspondence values to the posterior is to have the robot better understand

where it is located by establishing a relation between the objects. Hence, the new posteriors including the correspondence can be seen in Fig. 5.

D. Full SLAM vs Online SLAM

There is a mathematical relation between the online SLAM and full SLAM. The difference between the two SLAMs is in their poses; one represents the posterior over the entire path while the other represents a posterior over the current pose. By integrating the previous robot poses from the full SLAM problem once at a time and summing over each previous correspondence values, one can obtain the online SLAM posterior (Fig. 5).

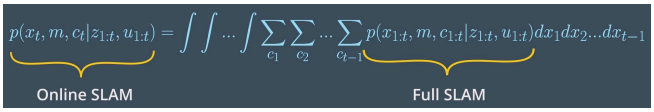

Fig. 5: Integration of the posterior of the full SLAM results in online SLAM.

Computing the full posterior composed of the robot pose, the map and the correspondence under SLAM poses a big challenge in robotics mainly due to the continuous and discrete portion.

The continuous parameter space composed of the robot poses and the location of the objects is highly dimensional. While mapping the environment and localizing itself, the robot will encounter many objects and have to keep track of each one of them. Thus, the number of variables will increase with time, and this makes the problem highly dimensional and challenging to compute the posterior.

Next, the discrete parameter space is composed out of the correspondence values, and is also highly dimensional due to the large number of correspondence variables. Not only that, the correspondence values increase exponentially over time since the robot will keep sensing the environment and relating the newly detected objects to the previously detected ones. Even if you assume known correspondence values, the posterior over maps is still highly dimensional as we saw in the mapping lessons.

One can now see why it is infeasible to compute the posterior under unknown correspondence. Thus, SLAM algorithms will have to rely on approximation while estimating a posterior in order to conserve computational memory.

While solving the localization problem we used different approaches to estimate the robot's pose inside the environment. The most robust approach was to use an AMCL algorithm. Each of the particles in the environment has the Robot's pose and weight. Now adding the map into this equation might seem like a good approach in solving the SLAM problem using particle filters. However, this approach will fail because the map is modeled with many variables resulting in high dimensionality. Thus, the particle filter approach to SLAM in this current form will scale exponentially and hence unusable for practical applications.

E. FastSLAM

A custom particle filter can be used to address this problem and is used in FastSLAM. The FastSLAM algorithm solves the Full SLAM problem with known correspondences. Using particles FastSLAM estimates the posterior over the robot path along with the map. Each of these particles hold the robot's trajectory which will give an advantage to SLAM to solve the problem of mapping with known poses. In addition to the robot's trajectory, each particle holds a map and each feature of the map is represented by a local Gaussian. With the FastSLAM algorithm, the problem is now divided into two separate independent problems, each of which aims to solve the problem of estimating the features of the map. To solve these independent mini problems, the FastSLAM will use a low-dimensional Extended Kalman Filter (EKF) algorithm. While map features are treated independently, dependency only exists between robot pose uncertainties.

- Estimating the Trajectory: FastSLAM estimates a posterior over the trajectory using a particle filter approach (i.e. MCL). This will give an advantage to SLAM to solve the problem of mapping with known poses.
- Estimating the Map: FastSLAM uses a low-dimensional EKF to solve independent features of the map, which are modeled with local Gaussian.

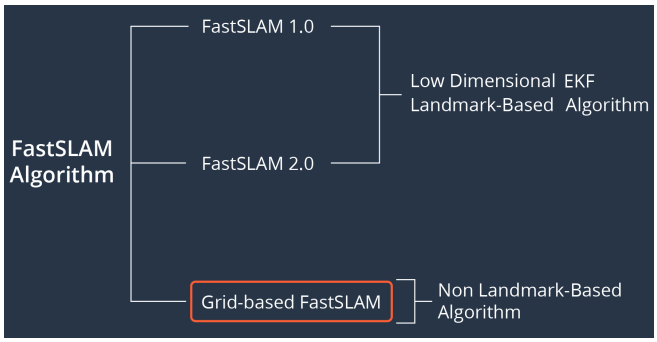

Fig. 6: Various FastSLAM Algorithms

This custom approach of representing the posterior with particle filter and Gaussian is known by the Rao-Blackwellized particle filter approach.

The FastSLAM algorithm can solve the full SLAM problem with known correspondences. Since FastSLAM uses a particle filter approach to solve SLAM problems, roboticists consider it as a powerful algorithm capable of solving both the Full SLAM and Online SLAM problems.

- FastSLAM estimates the full robot path, and hence it solves the Full SLAM problem.
- On the other hand, each particle in FastSLAM estimates instantaneous poses, and thus FastSLAM also solves the Online SLAM problem.

Three different instances of the FastSLAM algorithm exist (Fig. 6) as listed below.

- FastSLAM 1.0: The FastSLAM 1.0 algorithm is simple and easy to implement, but this algorithm is known to be inefficient since particle filters generate sample inefficiency. This algorithms use a large number of particles and a low-dimensional EKF to solve the SLAM problem.
- FastSLAM 2.0: The FastSLAM 2.0 algorithm overcomes the inefficiency of FastSLAM 1.0 by imposing a different distribution, which results in a low number of particles. This algorithm also uses a low-dimensional EKF to estimate the posterior over the map features.

- Grid-based FastSLAM: The third instance of FastSLAM is really an extension to FastSLAM known as the grid-based FastSLAM algorithm, which adapts FastSLAM to grid maps.

The main advantage of the FastSLAM 1.0 and 2.0 algorithms is that it uses a particle filter approach to solve the SLAM problem. Each particle will hold a guess of the robot trajectory, and by doing so, the SLAM problem is reduced to mapping with known poses. But, in fact, this algorithm presents a big disadvantage since it must always assume that there are known landmark positions, and thus with FastSLAM we are not able to model an arbitrary environment. Now, what if landmark positions are unavailable to us? Are we still able to solve the SLAM problem? We can solve this problem using grid-based FastSLAM as described below.

F. Grid-Based FastSLAM

With the grid-mapping algorithm we can model the environment using grid maps without predefining any landmark position. So by extending the FastSLAM algorithm to occupancy grid maps, the SLAM problem can be solved in an arbitrary environment (Fig. 7). While mapping a real-world environment, mobile robots equipped with range sensors are used. Then FastSLAM algorithm in terms of grid maps is solved.

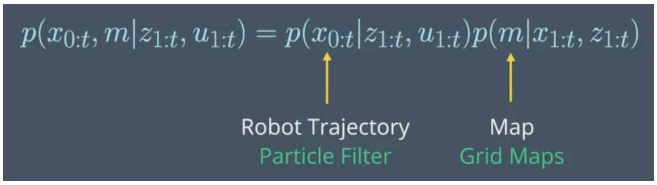

Fig. 7: Posterior of the grid-based FastSLAM

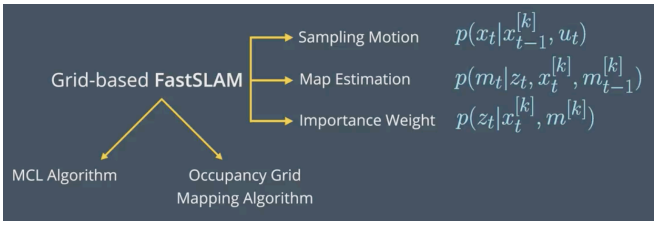

Fig. 8: Components of grid-based FastSLAM

Robot Trajectory: Just as in the FastSLAM algorithm, with the grid-based FastSLAM each particle holds a guess of the robot trajectory.

Map: In addition, each particle maintains its own map. The grid-based FastSLAM algorithm will update each particle by solving the mapping with known poses problem using the occupancy grid mapping algorithm.

To adapt FastSLAM to grid mapping, three different techniques are applied (Fig. 8):

1. **Sampling Motion**-$p(x_t | x_{t-1}^{[k]}, u_t)$: Estimates the current pose x_t, given the k^{th} particle's previous pose $x_{t-1}^{[k]}$ and the current controls u_t.
2. **Map Estimation**-$p(m_t | z_t, x_t^{[k]}, m_{t-1}^{[k]})$: Estimates the current map m_t, given the current measurements z_t, the current k^{th} particle pose $x_t^{[k]}$, and the previous k^{th} particle map $m_{t-1}^{[k]}$.
3. **Importance Weight**-$p(z_t | x_t^{[k]}, m^{[k]})$: Estimates the current likelihood of the measurement z_t, given the current k^{th} particle pose $x_t^{[k]}$ and the current k^{th} particle map $m^{[k]}$.

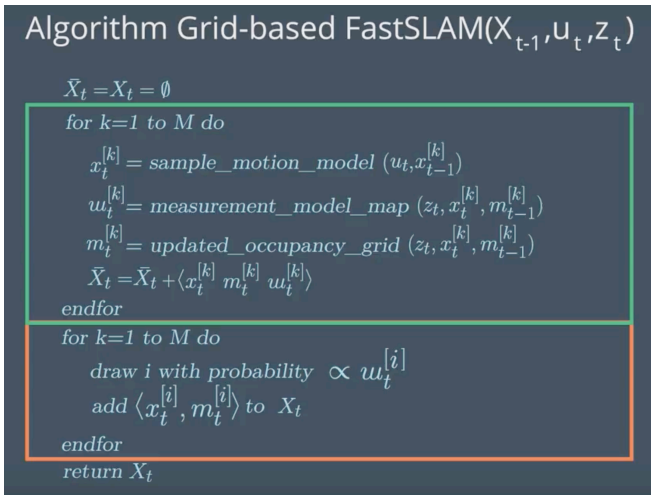

Fig. 9: Grid-based SLAM Algorithm

The sampling motion, map estimation and the importance weight are the essence of the grid-based SLAM algorithm. These are implemented to estimate both the map and the robot trajectory given the robot measurements and control. The grid-based FastSLAM is very similar to the MCL algorithm with additional term for the map estimation using occupancy-grid mapping algorithm.

As in MCL algorithm, the grid-based FastSLAM algorithm is composed of two sections represented by two for loops as shown in Fig. 9. The first section includes the motion, sensor measurements, and the map update steps. The second section includes the resampling process. At each iteration the algorithm takes the previous belief for the pose, the actuation command, and the sensor measurements as inputs. Initially the hypothetical belief is obtained by randomly generating M particles. Then in the first section, each particle implements the three techniques show above to estimate the k^{th} particle's current pose, likelihood of the measurement, and the map. Each particle begins by implementing the sampling technique in the sample motion model to estimate the current pose of the k^{th} particle. Next, in the measurement update step, each particle implements the importance weight technique in the measurement model map function to estimate the current likelihood of the k^{th} particle measurements. Next in the map update step, each particle implements the map estimation technique and updated the occupancy grid map function to estimate the current k^{th} particle map. This map estimation problem is solved under the occupancy grid mapping. Next, the newly estimated k^{th} particle's pose, map, and the likelihood of the measurements are all added to the hypothetical belief. Moving on to the second section, resampling process happens through the resampling step. Here particles with the measurement values close to the robot's measurement values survive and are redrawn in the next iteration while the others are discarded. The surviving particles poses and maps are added to the system belief. Finally the algorithms outputs the new belief and another cycle of iteration starts implementing the newly computed belief, the next motion, and the new sensor measurements

The grid-based FastSLAM algorithm implemented in ROS is called the Gmapping package [4]. Gmapping contains a single node called slam_gmapping. This node subscribes to the transform *tf*, relating the different frames of the laser, robot base and odometry as well as the robot laser scans. The slam_gmapping node publishes the occupancy grid map characteristics, the 2D occupancy grid map, and the entropy of distribution over the robot pose. Using slam_gmapping, a 2-D occupancy grid map of the environment (like a building floorplan) is created from laser and pose data collected by a mobile robot.

G. Graph SLAM

GraphSLAM is a SLAM algorithm that solves the full SLAM problem. This means that the algorithm recovers the entire path and the map instead of just the most recent pose and the map. This difference allows it to consider the dependencies between the current pose and previous poses. An example of where GrapSLAM will be applicable is in an underground minning. One way to map the space would be to drive a vehicle with a LiDAR and collects data about the surroundings. Then the data can be analyzed to create an accurate map of the envisronment. One benefit of the GraphSLAM is the reduced need for significant onboard processing capability. GraphSLAM has improved accuracy over FastSLAM. FastSLAM uses particles to estimate robot's most likely pose. However, at any point in time, it's possible that there isn't a particle in the most likely location. In large environments, the chances of having any particles in the most likely location is slim to none. Since GraphSLAM solve the full SLAM problemm, this means that it can work with all of the data at once to find the optimal solution. FastSLAM uses a titbits of information with a finite number of particles which mean there is room for error. Hence GraphSLAM will be used to avoid such errors. The following questions should be addressed in order to create a robust GraphSLAM:

1. How to construct the graphs?
2. How to optimize within their contraints to create the most likely set of poses and maps?
3. How to overcome some of GraphSLAM's limitations?

GraphSLAM uses graphs to represent the robot's poses and the environment. In GraphSLAM the robot's pose consisting of its position and orientation can be represented by a node. Usually, the first node is arbitrarily constrained to zero-zero or its equivalent in higher dimensions. The robot's pose at time step one can be represented by another node. And the two would be connected by an edge, sometimes called an arc. This edge is a soft spatial constraint between the two robot poses. These constraints are called soft because, as we have seen before, the motion and measurement data is uncertain. The constraints will have some amount of error present. Soft constraints come in two forms. Motion constraints between two successive robot poses and measurement constraints between a robot pose and a feature in the environment as shown in Fig. ?? below. The constraint seen between robot poses x_0 and x_1 is a motion constraint. If the robot were to sense its environment and encounter a feature m1, a soft measurement constraint m_1, would be added. The star in Fig. 10 represents a feature in the environment, which could be a landmark, specifically placed for the purpose of localization and mapping, or an identifiable element of the environment such as a corner or an edge. The solid edges represent the motion constraints and the dashed edges represent the measurement constraints. As the robot moves around, more and more nodes are added to the graph as shown in Fig. 11. And over time, the graph constructed by the mobile robot becomes very large in size. Luckily GraphSLAM is able to handle large number of features.

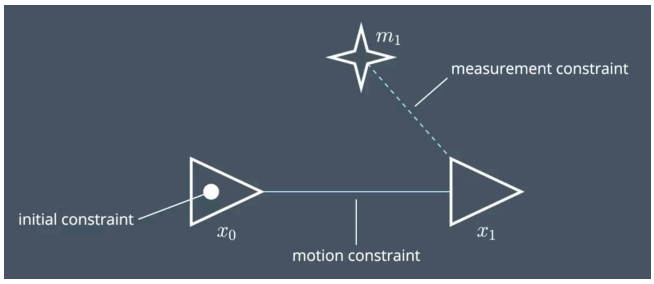

Fig. 10: A graph consisting of two nodes and two constraints.

Every motion or a measurement constraint pulls a system closer to that constraints desired state. Since the measurement and motions are uncertain, constraints will conflict with each other and there will always be some error present. In the end, the goal is to find the node configuration that minimizes the overall error present in all the constraints.

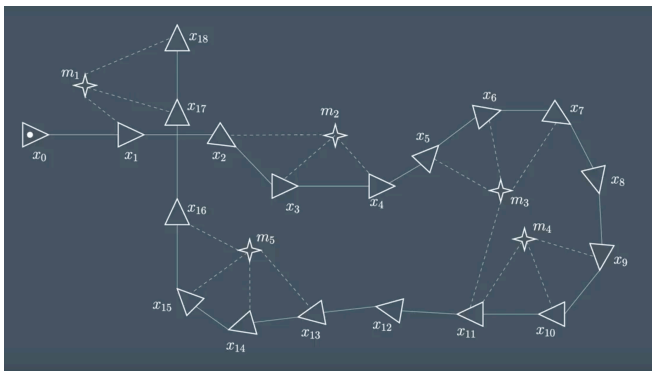

Fig. 11: Graph consisting of nodes and constraints.

The goal of GraphSLAM is to create a graph of all robot poses and features encountered in the environment to find the most likely robot's path and the map of the environment. This task can be divided into two sections; front-end and back-end.

The front-end of GraphSLAM looks at how to construct the graph using the odometry and sensory measurements collected by the robot. This includes, interpreting sensory data, creating the graph, and continuing to add nodes and edges to it as the robot continues to traverse the environment. Naturally, the front-end can differ greatly from application to application. Depending on the desired goal, including accuracy, the sensor used and other factors. For instance, the front-end of a mobile-robot performing SLAM in the office using a laser range finder, would differ greatly from the front-end for a vehicle operating on a large outdoor environment and using a stereo camera. The front-end of a GraphSLAM also has the challenge of solving the data association problem. In simpler term, this means accurately identifying whether features in the environment have been previously seen.

The input to the back-end of GraphSLAM is the completed graph with all the constraints. The output of the back-end is the most probable configuration of the robot poses map features. The back-end is an optimization process that takes all of the constraints and finds the system configuration that produces the smallest error. The back-end is a lot more consistent across applications. The front-end and the back-end can be complement is succession or can be performed iteratively with a back-end feeding an updated graph to the front-end for further processing.

At the core of GraphSLAM is graph optimization - the process of minimizing the error present in all of the constraints in the graph. The optimization applied is called the *maximum likelihood estimation* (MLE) to structure and solve the optimization problem for the graph. Likelihood is a complementary principle to probability. While probability tries to estimate the outcome given the parameters, likelihood tries to estimate the parameters that best explain the outcome. When applied to SLAM, likelihood tries to estimate the most likely configuration of state and feature locations given the motion and measurement observations.

The MLE can be solved numerically by applying an optimization algorithm. The goal of an optimization algorithm is to speedily find the optimal solution - in this case, the local minimum. There are several different algorithms that can tackle this problem; in SLAM, the gradient descent, Levenberg-Marquardt, and conjugate gradient algorithms are quite common.

In one-dimensional graphs, the robot's motion and measurements are limited to one dimension - they could either be performed either forwards or backwards. In such a case, each constraint could be represented in the following form,

- 1-D Measurement constraint: $(z_t-(x_t+m_t))^2 / \sigma^2$
- 1-D Motion constraint: $(x_t-(x_{t-1}+u_t))^2 / \sigma^2$

In multi-dimensional systems, we must use matrices and covariances to represent the constraints. This generalization can be applied to system of 2-dimensions, 3-dimensions, and any n-number of dimensions. The equations for the constraints would look like so,

- n-D Measurement constraint:$(z_t-h(x_t,m_j))^T Q_t^{-1}(z_t-h(x_t,m_j))$
- n-D Motion constraint: $(x_t-g(u_t,x_{t-1}))^T R_t^{-1}(x_t-g(u_t,x_{t-1}))$

Where $h()$ and $g()$ represent the measurement and motion functions, and Q_t and R_t are the covariances of the measurement and motion noise. These naming conventions should be familiar to you, as they were all introduced in the Localization module.

The multidimensional formula for the sum of all constraints is presented below.

$$J_{GraphSLAM} = x_0^T \Omega x_0 + \sum_t (x_t-g(u_t,x_{t-1}))^T R_t^{-1}(x_t-g(u_t,x_{t-1})) + \sum_t (z_t-h(x_t,m_j))^T Q_t^{-1}(z_t-h(x_t,m_j))$$

The first element in the sum is the initial constraint - it sets the first robot pose to equal to the origin of the map. The covariance, Ω_0, represents complete confidence. Essentially,

$$\Omega_0 = [\infty \;\; 0 \;\; 0; \;\; 0 \;\; \infty \;\; 0; \;\; 0 \;\; 0 \;\; \infty]$$

A more sophisticated data structure consisting of information matrix (Ω) and information vector (ξ) is used to work with multi-dimensional graphs and multi-dimensional constraints. The information matrix (Ω) is the inverse of the covariance matrix ($\sum$). This means that a higher certainty is represented by larger values in information matrix, the opposite of a covariance matrix where complete certainty is represented by zero. The matrix and vector hold over all of the poses and all of the features in the environment. Every off-diagonal cell in the matrix is a link between two poses, a pose and a feature or two features. When no information is available about a link, the cell has a value of zero. The information and information vector exploit the additive property of the negative log likelihood of constraints. For systems with linear measurements and motion models, the constraints can be populated into the information matrix and information vector in an additive manner. The information matrix and information vectors are considered to be populated, once every constraint has been added. An example is shown inf Fig. 12 below.

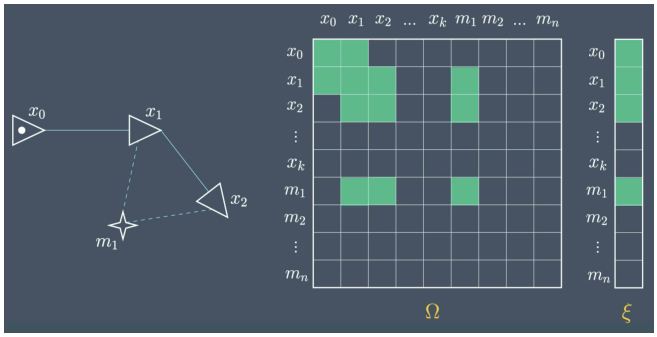

Fig. 12: An example of information matrix and vector containing three poses and one feature.

It is common for the number of poses and features to be in thousands, or even tens of thousands. The information matrix is considered sparse because most off-diagonal elements are zero, since there is no relative information to tie them together. This sparsity is very valuable when it comes to solving the system of equations that is embedded in the information matrix and vector. In summary,

- A motion constraint ties together two poses,
- A measurement constraint ties together the feature and the pose from which is was measured,
- Each operation updates 4 cells in the information matrix and 2 cells in the information vector,
- All other cells remain 0. Matrix is called 'sparse' due to large number of zero elements,
- Sparsity is a very helpful property for solving the system of equations.

Once the information matrix and information vector have been populated, the path and map can be recovered by the following operation,

$$\mu = \Omega^{-1}\xi$$

The result is a vector, μ defined over all poses and features, containing the best estimate for each. This operation is very similar to the simple one-dimensional case, with a bit of added structure. Completing the above operation requires solving a system of equations. In small systems, this is an easily realizable task, but as the size of the graph and matrix grows - efficiency becomes a concern. The efficiency of this

operation, specifically the matrix inversion, depends greatly on the topology of the system.

If the robot moves through the environment once, without ever returning to a previously visited location, then the topology is linear. Such a graph will produce a rather sparse matrix that, with some effort, can be reordered to move all non-zero elements to near the diagonal. This will allow the above equation to be completed in linear time.

A more common topology is cyclical, in which a robot revisits a location that it has been to before, after some time has passed. In such a case, features in the environment will be linked to multiple poses - ones that are not consecutive, but spaced far apart. The further apart in time that these poses are - the more problematic, as such a matrix cannot be reordered to move non-zero cells closer to the diagonal. The result is a matrix that is more computationally challenging to recover. However, a variable elimination algorithm can be used to simplify the matrix, allowing for the inversion and product to be computed quicker.

Variable elimination can be applied iteratively to remove all cyclical constraints. Just like it sounds, variable elimination entails removing a variable (ex. feature) entirely from the graph and matrix. This can be done by adjusting existing links or adding new links to accommodate for those links that will be removed.

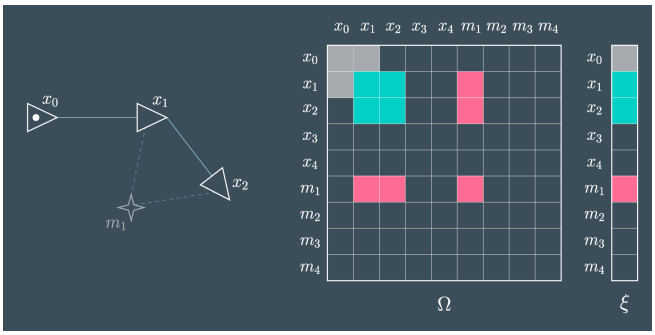

Fig. 13: Variable elimination from information matrix and vector

If we considered the constraints as springs, variable elimination removes features, but keeps the *net* forces in the springs unaltered by adjusting the tension on other springs or adding new springs where needed. This process is demonstrated in the image above (Fig. 13) that shows the elimination of m_1. In this process the matrix will have five cells reset to zero (indicated in red), and four cells will have their values adjusted (indicated in green) to accommodate the variable elimination. Similarly, the information vector will have one cell removed and two adjusted. This process is repeated for all of the features, and in the end the matrix is defined over all robot poses. At this point, the same procedure as before can be applied, $\mu=\Omega^{-1}\xi$.

Performing variable elimination on the information matrix/vector prior to performing inference is less computationally intense than attempting to solve the inference problem directly on the unaltered data.

In practice, the analytical inference method described above is seldom applied, as numerical methods are able to converge on a sufficiently accurate estimate in a fraction of the time. How are the nonlinear constraints handled in GraphSLAM?

The idea that a robot only moves in a linear fashion is quite limiting, and so it is important to understand how to work with nonlinear models. In localization, nonlinear models couldn't be applied directly, as they would have turned the Gaussian distribution into a much more complicated distribution that could not be computed in closed form (analytically, in a finite number of steps). The same is true of nonlinear models in SLAM - most motion and measurement constraints are nonlinear, and must be linearized before they can be added to the information matrix and information vector. Otherwise, it would be impractical, if not impossible, to solve the system of equations analytically.

Luckily, it is possible to linearize a nonlinear model using Taylor series expansion to linearize nonlinear constraints for SLAM. Taylor series approximates a function using the sum of an infinite number of terms. A linear approximation can be computed by using only the first two terms and ignoring all higher order terms. In multi-dimensional models, the first derivative is replaced by a Jacobian - a matrix of partial derivatives.

A linearization of the measurement and motion constraints is the following,

$$g(u_t, x_{t-1}) \simeq g(u_t, \mu_{t-1}) + G_t(x_{t-1}-\mu_{t-1})$$

$$h(y_t) \simeq h(\mu_t) + H_t^i(y_t-\mu_t)$$

To linearize each constraint, a value for μ_{t-1} or μ_t is needed to linearize about. This value is quite important since the linearization of a nonlinear function can change significantly depending on which value one chooses to do so about.

When presented with a completed graph of nonlinear constraints, only the motion constraints can be applied to create a pose estimate, $[x_0 \ldots x_t]^T$, and use this primitive estimate in place of μ to linearize all of the constraints. Then, once all of the constraints are linearized and added to the matrix and vector, a solution can be computed as before, using $\mu = \Omega^{-1}\xi$.

This solution is unlikely to be an accurate solution. The pose vector used for linearization will be erroneous, since applying just the motion constraints will lead to a graph with a lot of drift, as errors accumulate with every motion. Errors in this initial pose vector will propagate through the calculations and affect the accuracy of the end result. This is especially so because the errors may increase in magnitude significantly during a poorly positioned linearization (where the estimated μ_t is far from reality, or the estimated μ_t lies on a curve where a small step in either direction will make a big difference).

To reduce this error, the linearization process can be repeated several times, each time using a better and better estimate to linearize the constraints about.

The first iteration will see the constraints linearized about the pose estimate created using solely motion constraints. Then, the system of equations will be solved to produce a solution, μ. The next iteration will use this solution, μ, as the estimate used to linearize about. The thought is that this estimate would be a little bit better than the previous; after all, it takes into account the measurement constraints too. This process continues, with all consequent iterations using the previous solution as the vector of poses to linearize the

constraints about. Each solution incrementally improves on the previous, and after some number of iterations the solution converges.

In summary, nonlinear constraints can be linearized using Taylor Series, but this inevitably introduces some error. To reduce this error, the linearization of every constraint must occur as close as possible to the true location of the pose or measurement relating to the constraint. To accomplish this, an iterative solution is used, where the point of linearization is improved with every iteration. After several iterations, the result, μ, becomes a much more reasonable estimate for the true locations of all robot poses and features. More details can be found in [5, 6].

The workflow for GraphSLAM with nonlinear constraints is summarized below:

- Collect data and create graph of constraints.
- Until convergence:
  - Linearize all constraints about an estimate, μ, and add linearized constraints to the information matrix and vector.
  - Solve system of equations using $\mu = \Omega^{-1}\xi$.

H. RTAB-Mapping Using GraphSLAM

For applications, a specific GraphSLAM called, real-time appearance based (RTAB) mapping. Appearance map means that the algorithm uses data collected from vision sensors to localize the robot and map the environment. In appreance based method a process called loop clousure is used to determine wheter the robot has seen a location before. As the robot travels to new areas in its environment, the map is expanded, and the number of images that each new image must be compared to increses. This causes the loop closure to take longer with the complexity increasing linearly. The RTAB map is optimised for [11, 12] and long-term SLAM, by using multiple strategies to allow for loop closure to be done in real-time. In this context, this means that the loop closure is happening fast enough that the result can be obtained before the next camera images are aquired. The information flow in the RTAB map is shown in Fig. 14 below.

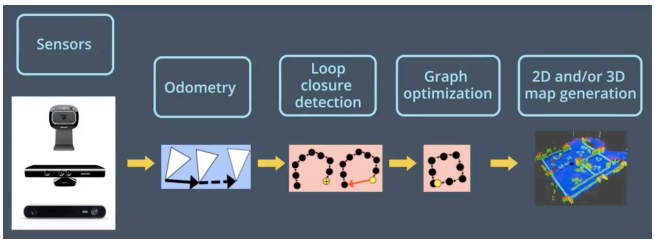

Fig. 14: An overview of the RTAB mapping flow.

The front-end of the RTAB-map focuses on sensor data used to obtain the constraints that are used for feature optimization approaches as shown in Fig. 15. Although, landmark constraints are used for other graphs, SLAM methods like 2D GraphSLAM, RTAB-map does not use them. Only odometry constraints and loop closure constraints are considered here. The odometry constraints can come from wheel encoders, IMUs, LiDAR or Visual Odometry. Visual odometry is accomplished using 2D features such as speeded up robust features (SURF).

The RTAB-map is apperance based with no metric distance information. RTAB-map can use a single monocular camera to detect loop closure. For metric GraphSLAM, RTAB-map requires a RGB-D camera or a stereo camera to compute the geometric constraints between the images of a loop clouser. A laser range finder can also be used to improve or refine the geometric constraint by providing a more precise localization. The front-end also involve graph management, which includes node creation and loop closure detection using bag-of-words. The back end of the RTAB-map (Fig. 16) includes graph optimization and an assembly of an occupancy-grid from the data of the graph.

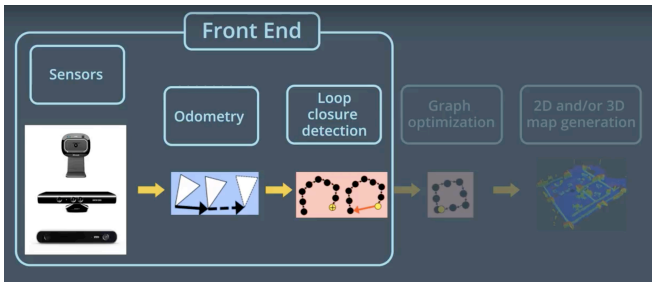

Fig. 15: The front end of a RTAB mapping. The sensor data is used to obtain the odometry and loop closure constraints.

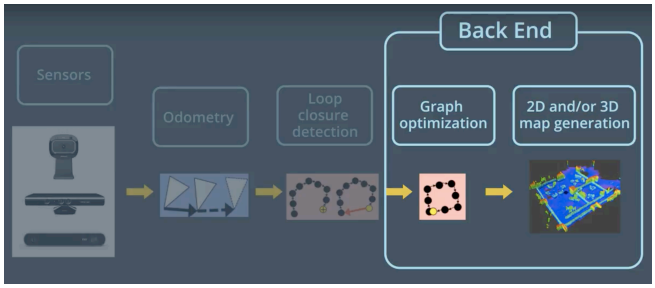

Fig. 16: The back end of the RTAB mapping where the graph optimization and the map output is generated.

Loop closure detection is the process of finding a match between the current and previously visited locations in SLAM. There are two types of loop closure detection; local and global loop closures. Many probabilistic SLAM methods use local loop closure detection where matches are found between a new observation and limited map region. The size and location this limited map region is determined by the uncertainty associated with robot's pose. This type of approach fails if the estimated pose is incorrect. It is likely that the events in the real world the robot is operating in will cause errors in the estimated position.

In a global loop closure approach, a new location is compared with previously viewed locations. If no match is found, the new location is added to the memory. As the map grows and the more locations are added, the amount of time to check whether the location has been previously seen increases linearly. If the time it takes to search and compare new images to the one stored in memory becomes larger than the acquisition time, the map becomes ineffective. RTAB-map uses a global loop closure approach combined with other techniques to ensure that the loop closure process happens in real time.

In RTAB-mapping loop closure is detected using a bag-of-words appraoch. Bag-of-words is commonly used in vision-based mapping. A feature is a very specific

characteristic of an image like a patch with a complex tecture, or a well-defined edge or corner. In RTAB-map, the default method for extracting features from an image is called Speeded Up Robust Feature or SURF. Each feature has a descriptor asscciated with it as shown in Fig. 17. A feature descriptor is a unique and robust representation of the pixels that make up a feature.

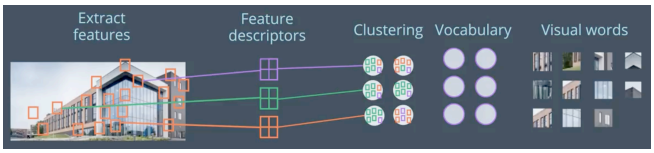

Fig. 17: SURF approach

In SURF, the point of interest where the feature is split into smaller square sub-regions. From these sub-regions, the pixel intensities in regions of regularly spaced sample points are calculated and compared. The differences between the sample points are used to categorize the sub-regions of the the image. Comparing feature descriptors directly is time consuming, so a vocabulary is used for faster comparision. This is where similar features or synonyms are clustered together. The collection of these clusters represent the vocabulary. When a feature descriptor is mapped to one in the vocabulary, it is called quantization. At this point, the feature is now linked to a word and can be referred to as a visual word. When all features in an image are quantized, the image is now a bag-of-words. Each word keeps a link to images that it is associated with, making image retrieval more efficient over a large data set. To compare an image with previous imaages, a matching score is given to all images containing the same words as shown in Fig. 18. Each word keeps track of which image it has been seen in, so similar images can be found. This is called an inverted index. If a word is seen inan image, the score of this image will increase. If an image shares many visual words with the query image, it will score higher. A Bayesian filter is used to evaluate the scores. This is the hypithesis that an image has been seen before. When the hypothesis reaches a predefined threshold H, a loop closure is detected.

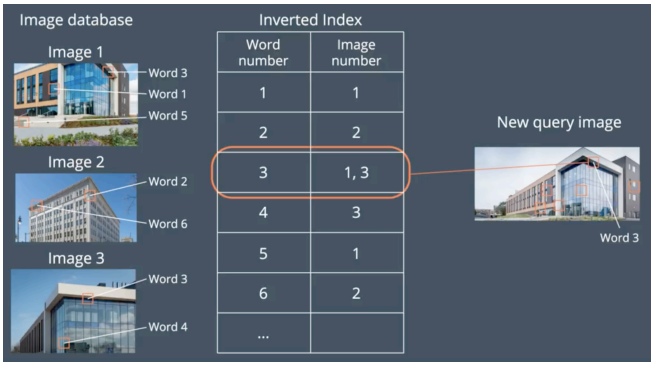

Fig. 18: An example of bag-of-words used in SURF.

RTAB-Map uses a memory management technique to limit the number of locations considered as candidates during loop closure detection. This technique is a key feature of RTAB-Map and allows for loop closure to be done in real time.

The overall strategy is to keep the most recent and frequently observed locations in the robot's Working Memory (WM), and transfer the others into Long-Term Memory (LTM) as shown in Fig. 19. Following steps are taken for the memory management.

- When a new image is acquired, a new node is created in the Short Term Memory (STM).
- When creating a node, recall that features are extracted and compared to the vocabulary to find all of the words in the image, creating a bag-of-words for this node.
- Nodes are assigned a weight in the STM based on how long the robot spent in the location - where a longer time means a higher weighting.
- The STM has a fixed size of S. When STM reaches S nodes, the oldest node is moved to WM to be considered for loop closure detection.
- Loop closure happens in the WM.
- WM size depends on a fixed time limit T. When the time required to process new data reaches T, some nodes of graph are transferred from WM to LTM - as a result, WM size is kept nearly constant.
- Oldest and less weighted nodes in WM are transferred to LTM before others, so WM is made up of nodes seen for longer periods of time.
- LTM is not used for loop closure detection and graph optimization.
- If loop closure is detected, neighbours in LTM of an old node can be transferred back to the WM (a process called retrieval).

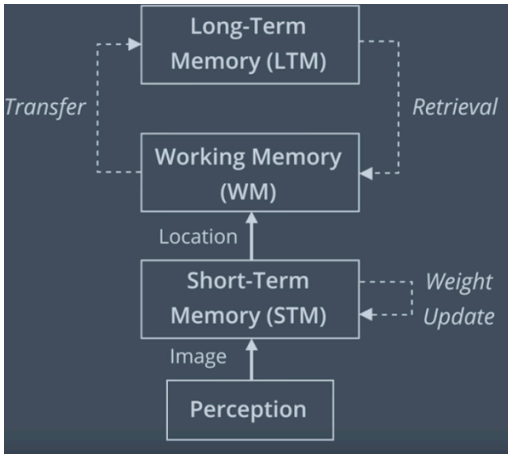

Fig. 19: RTAB-Mapping memory management.

When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map. RTAB-Map supports 3 different graph optimizations: Tree-based network optimizer (TORO), General Graph Optimization (G2O) and GTSAM (Smoothing and Mapping).

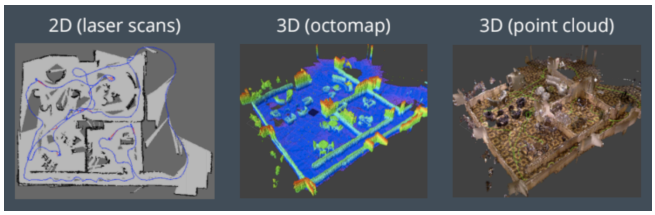

Fig. 20: An example of RTAB-map assembly and output.

All of these optimizations use node poses and link transformations as constraints. When a loop closure is detected, errors introduced by the odometry can be propagated to all links, correcting the map. The possible outputs of RTAB-Map are a 2d Occupancy grid map, 3d occupancy grid map (3d octomap), or a 3D point cloud as shown in Fig. 20.

Graph-SLAM complexity is linear, according to the number of nodes, which increases according to the size of the map. By providing constraints associated with how many nodes are processed for loop closure by memory management, the time complexity becomes constant in RTAB-Map (Fig. 21).

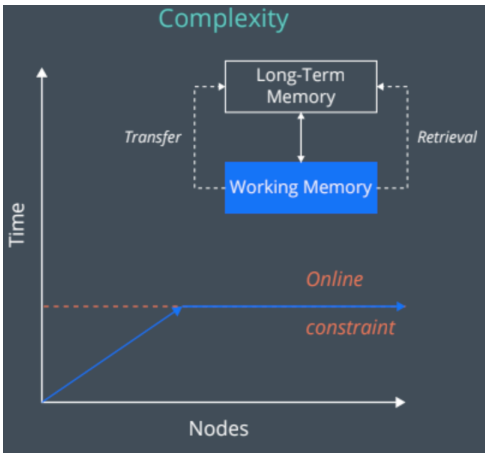

Fig. 21: RTAB-Map complexity.

III. Robot World ROS Package

A. Building ROS Package

The Robot Operating System (ROS) [3] is used to create ROS package for the simulation of the robot. Various ROS packages are used to accurately localize a mobile robot inside a provided map in Gazebo and RViz simulation environments. Steps used to create the simulation model is discussed below.

Several aspects of robotics is explored with a focus on ROS, including

1) Building a mobile robot, using Gazebo, for simulated tasks.
2) Creating a ROS package that launches a custom robot model in a Gazebo world.
3) Building a custom world in Gazebo for mapping simulation.
4) Adding sensors to the robot, and integrating it with Gazebo and RViz.
5) Integrating open-source ROS package RTAB mapping package allows the robot to map an unknown environment.
6) Parameter tuning in mapping.launch file in order for the RTAB-Map to be able to see the world around and map.

In this section, a brief instruction is provided to start developing ROS packages from scratch. After creating and initializing the "catkin_ws" workspace, start by navigating to the "src" directory and creating a new empty package called "RoboND-SLAM-Project".

Next, inside "RoboND-SLAM-Project" folder, create folders, "launch", "worlds", "urdf", "config", "meshes", "srcipts" that will further define the structure of the package. Then the worlds and urdf files are created from scratch. A mapping launch file along with robot_description.launch, rviz.launch, teleop.launch, and world.launch were added into the launch folder. The RViz configurations were saved in robot_slam.rviz file in the config folder. The meshes for the RGB-D camera and the LiDAR were added into the meshes folder. Then a python script was created where the launch of the various launch files are managed.

B. Robot World Creation in Gazebo

Two worlds were created and mapped in this project. The first was a world was a virtual environemnt representing a kitchen and dining room. For the second world, a bedroom, offiice-room and garden with one oak-tree was created. Then some tables, cabinets and bookshelfs were added in gazebo. The custom world called boring_room was created from scratch using Gazebo and inserting existing models found in the Gaezebo model database.

The robot was created with four wheels and was equipped with a LiDAR and an RGB-D camera. The addition of various features help the RTAB algorithm to identify consistent loop-closure constraints.

C. RViz Integration

While Gazebo is a physics simulator, RViz can visualize any type of sensor data being published over a ROS topic like camera images, point clouds, LiDAR data, etc. This data can be a live stream coming directly from the sensor or pre-recorded data stored as a bag file. RViz is the one-stop tool to visualize all the three core aspects of a robot: Perception, Decision Making, and Actuation.

The robot model is integrated into RViz to visualize data from the camera and laser sensors. In the launch file (robot_description.launch), two nodes need to be added. One node uses the package joint_state_publisher that publishes joint state messages for the robot, such as the angles for the non-fixed joints. The second node uses the robot_state_publisher package that publishes the robot's state to tf (transform tree). The robot model has several frames corresponding to each link/joint. The robot_state_publisher publishes the 3D poses of all of these links. This offers a very convenient and efficient advantage, especially for more complicated robots (like the PR2).

Another launch file was created just for the mapping purpose. This launch file contains various parameters for the RTAB-Map to accurately map the environment.

IV. Graph-SLAM Implementation

The GraphSLAM-based RTAB-Map is the best solution for SLAM to develop robots that can map environments in 3D. These considerations come from RTAB-Map's speed and memory management, its custom developed tools for information analysis and, most importantly, the quality of the documentation. For this project we will be using the rtabmap_ros package [7], which is a ROS wrapper (API) for interacting with rtabmap.

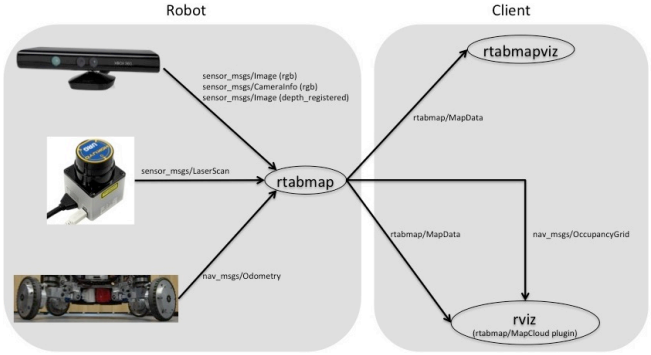

Fig. 22: RTAB-Map ROS package overview.

When building the ROS package proper care should be taken to make sure that the right information for RTAB-Map is published to successfully work. The Fig. 22 shows a visual overview of the high level connections enabling both mapping and localization.

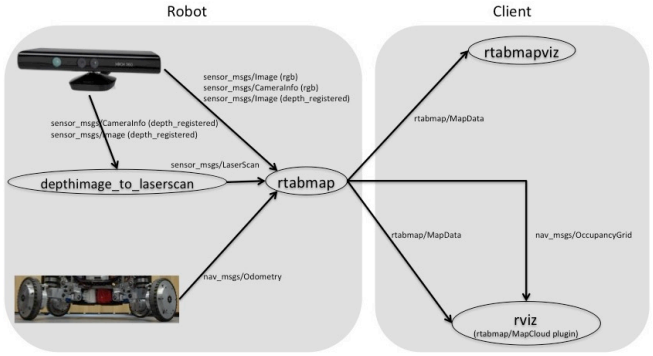

Fig. 23: Extracting laser scans data type from RGB-D depth camera to use with the RTAB-Map.

Removing the laser scanner does not mean that we no longer need the scan topic, but that we need to find a new way to generate this scan topic from the RGB-D camera images. This is where the depthimage_to_laserscan package [8] for the Kinect comes in. This package illustrates and provides a method of mapping the depth point cloud into a usable scan topic. This remapping can be viewed in the image in Fig. 23.

This is a convenient solution for those without a laser scanner on their robot. Since we are upgrading our robot from the previous project, we will elect to keep the laser range finder and the scan topic it provides and sticking with RTAB-Maps optimal configuration. Furthermore, it can be noted that the scan input topic is a convenient way to efficiently create an occupancy grid based on 2D ray tracing. However, the same results can be achieved with depth image, but it is more costly to generate.

While developing the robot's URDF file, several links and joints were created. Each link has its own coordinate frame, along with additional coordinate frames for the world (map) and the odometry frames. The tf library [9] helps keep track of all these different coordinate frames over time, and defines their relation with one another. The library allows one to debug the coordinate frames (or the tf tree) in several ways. view_frames will create a graphical representation of that tree, providing a broad view of how different frames (or links) in the setup connect to each other.

$ rosrun tf view_frames

The above will create a PDF file that will depict the tf tree as shown below. Here is an example of a tf tree for a sample robot for this project:

Another debugging tool under the tf library that is useful for debugging the ROS package setup is roswtf. The roswtf command will examine and analyze the setup or the graph above, including any running nodes and environment variables, and warn about any potential issues or errors. roswtf performs a lot of checks. For example, it checks if all running nodes are connected and communicating properly, or if any nodes have died.

Another tool that is used is **rtabmapviz**, which is an additional node for real time visualization of feature mapping, loop closures, and more. It's not recommended to use this tool while mapping in simulation due to the computing overhead. **rtabmapviz** is great to deploy on a real robot during live mapping to ensure that we are getting the necessary features to complete loop closures.

V. Results and Discussions

One physical robot model was created using Gazebo, as show in Fig. 15 and one custom world boring_room.world as shown in Fig. 16. They are very similar in shape and size except the my_bot has an extra mount for the LiDAR sensor in the front of the robot.

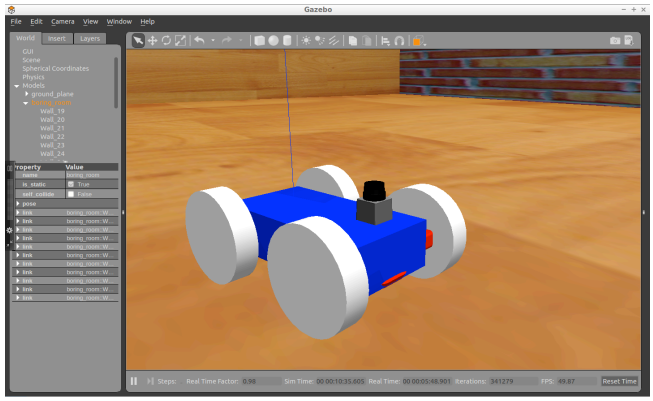

Fig. 15: my_bot robot model

Two sensors were added to the robot model, a LiDAR and a RGB-D camera sensor.

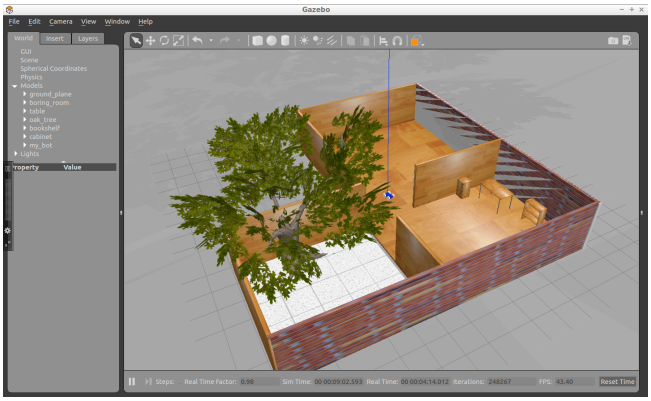

Fig. 16: boring_room gazebo world

The move_base package incorporates map, odometry and sensor data to evaluate the best path, and provides the robot with the associated velocity commands. It provides the interface between the path-planning nodes,

base_local_planner and base_global_planner and the costmap nodes, local_costmap and global_costmap.

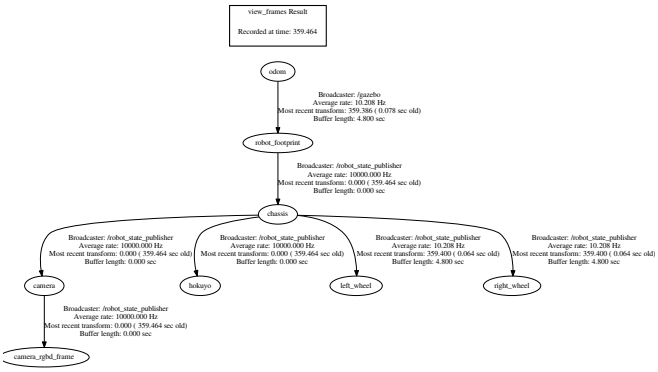

Fig. 17: tf-tree of my_robot

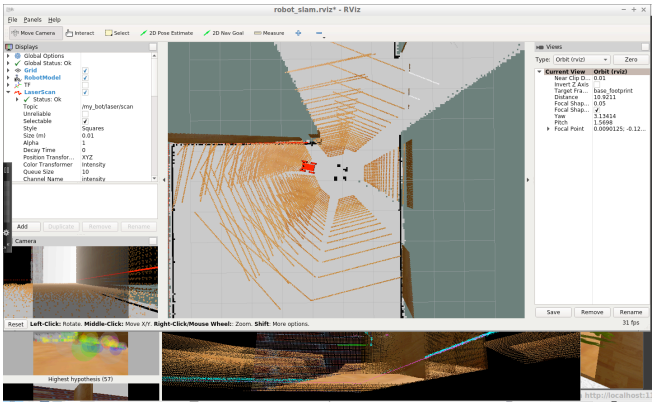

Fig. 18: Map as visualized in RViz.

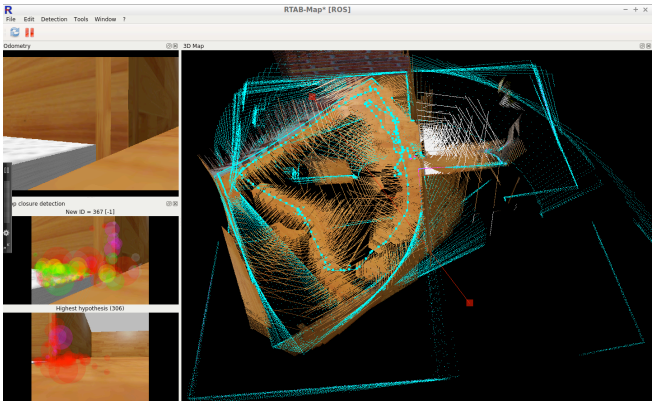

Fig. 19: 3D map as seen in the RTAB-Map.

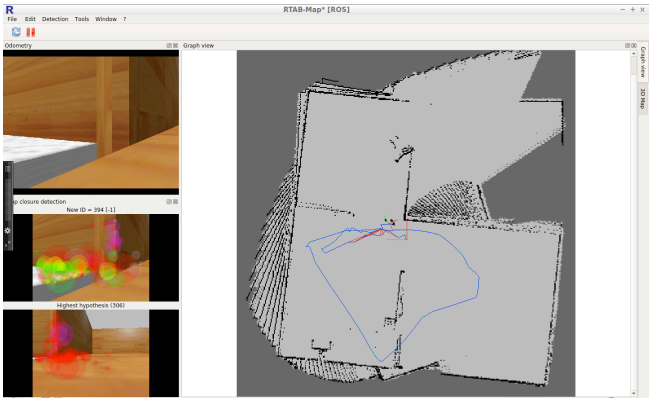

Fig. 20: Map as seen as a graph in RTAB-Map.

The transform tree (tf) for the robot model is shown in Fig. 17. The tf-tree shows whether all of the links in the robot model my_bot are properly connected.

The Fig. 18 shows the map in RViz. In order to see the RTAB-map visualize the images and laser scans, the scan frequency needs to be syncronyzed with the camera image frequency. Hence to facilitate this the following like was added in the mapping.launch file.

<param name="queue_size" type="int" value="30"/>

The value of the queue_size was adjusted in order to make sure that the images were broadcasted in the RTAB-Map. The robot was driven using keyboard to navigate in the environment. The outline of the environment can be seen in Fig. 19. Also the loop closures can be seen in one of the rooms in the boring_room.world. The Fig. 20 shows the graph visualization of the 3D map created by the RTAB-Map.

The kitchen dinning world was mapped using teleop keyboard controller. Fig. 21 and Fig. 23 show the Gazebo world and the RViz simulation environment.

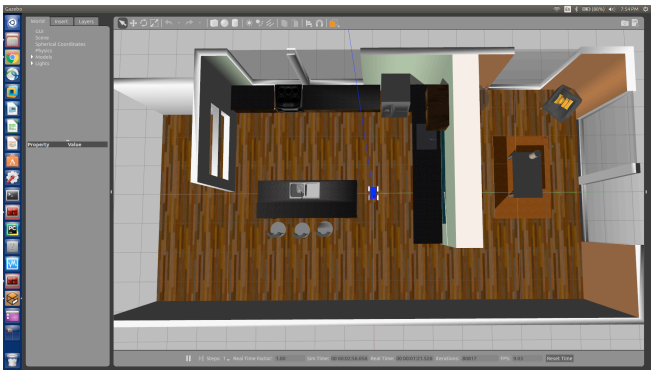

Fig. 21: Gazebo world of the kitchen_dining World.

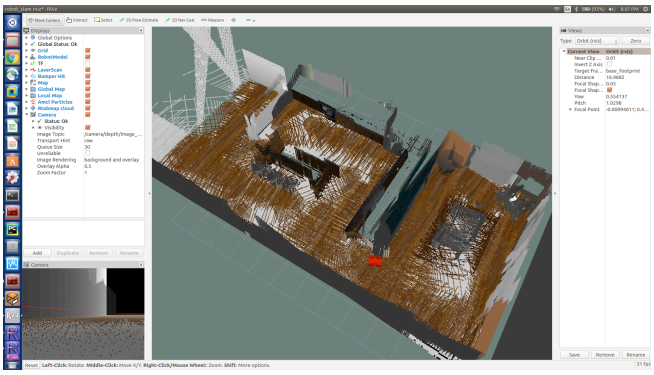

Fig. 22: kitchen_dining shown on RViz after my_bot completed mapping.

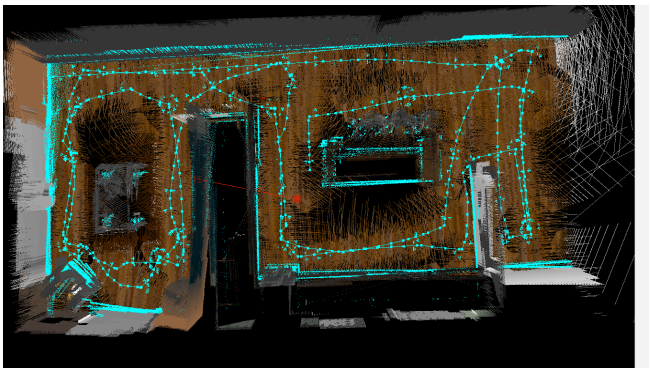

Fig. 23: RTAB 3D-Map after my_bot completed mapping the kitchen_dining world.

Fig. 22 and Fig. 23 show the RTAB-Map 3D and Graph. Due to complex features in the Kitchen dining world it needed to be run on a machine with NVidia GPU.

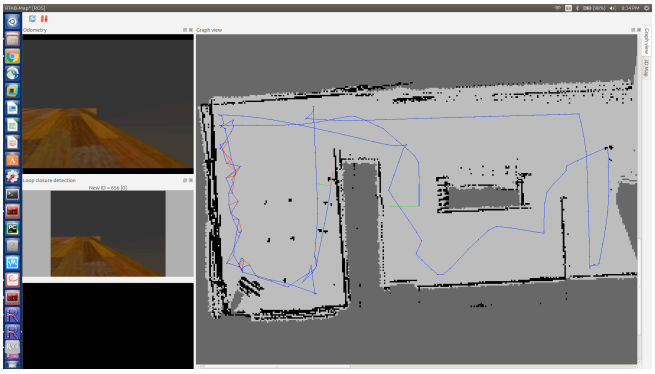

Fig. 24: RTAB Graph after for a second run of the my_bot mapping the kitchen_dining world.

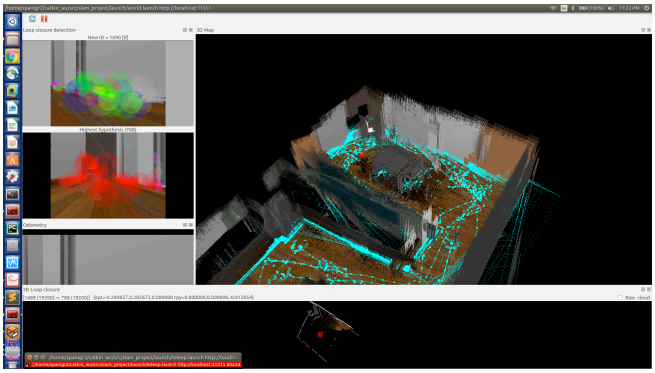

Fig. 25: RTAB Map showing loop closure detection

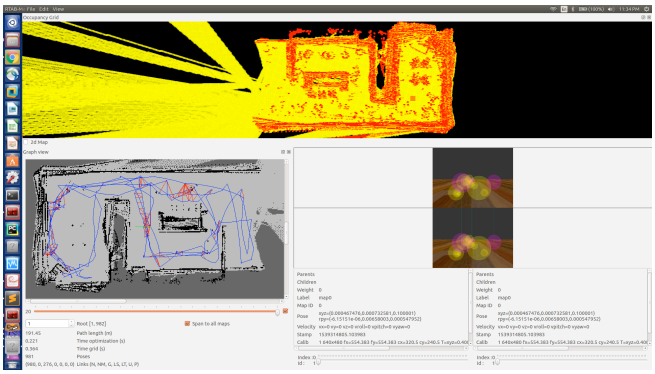

Fig. 26: Loop closure and 2D occupancy-grid maps shown in RTAB Database Viewer.

The 2D occupancy-grid and 2D graph is shown using the RTAB Database Viewer as shown in Fig. 25 and Fig. 26 below. The loop-closure map clearly shows 276 global loop closures for the kitchen_dining world. Due to high number of global loop closures, the rtabmap.db database file turned out to be more than 450MB in size.

VI. Conclusion

A custom robot model and custom world models were created in gazebo. The dimensions, inertia and other parameters of the robot chassis, wheels, sensors, joints etc. were defined in the .xacro file. The world was created using a floor plan. Then the model was uploaded in Gazebo and various elements were added to the rooms. Then the SLAM algorithm was performed using RTAB-Mapping in order to map the unknown environment in the custom world. Various parameters were added into mapping launch file to facilitate the RTAB-Mapping.

The mapping was performed manually via keyboard control in this project. However, next step in future work would be to add Navigation Stack in order to provide goal points for the robot. Using navigation stack will allow the robot to truly autonomously map and navigate the surrounding. As future work, it would be a natural next step to test the mapping using a RGBD camera on a real robot.

Acknowledgment

The authour would like to thank Udacity Inc. for providing some basic directions for RTAB-Mapping parameter tuning.

References

[1] Thrun, S. (2002, August). Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence* (pp. 511-518). Morgan Kaufmann Publishers Inc..

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

[3] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y., 2009, May. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software* (Vol. 3, No. 3.2, p. 5).

[4] http://wiki.ros.org/gmapping

[5] Thrun, S. and Montemerlo, M., 2006. The graph SLAM algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, *25*(5-6), pp.403-429.

[6] Grisetti, G., Kummerle, R., Stachniss, C. and Burgard, W., 2010. A tutorial on graph-based SLAM. *IEEE Intelligent Transportation Systems Magazine*, *2*(4), pp.31-43.

[7] http://wiki.ros.org/rtabmap_ros

[8] http://wiki.ros.org/depthimage_to_laserscan

[9] http://wiki.ros.org/tf2