



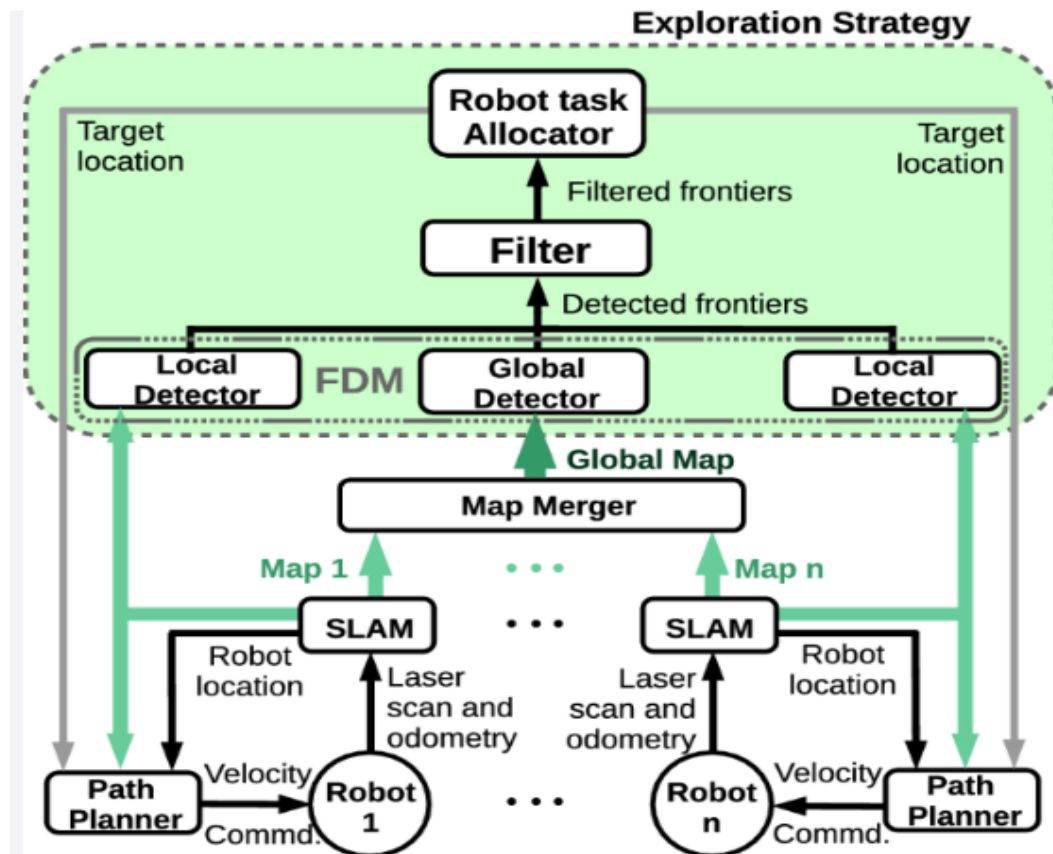
Robotics

Team: 7

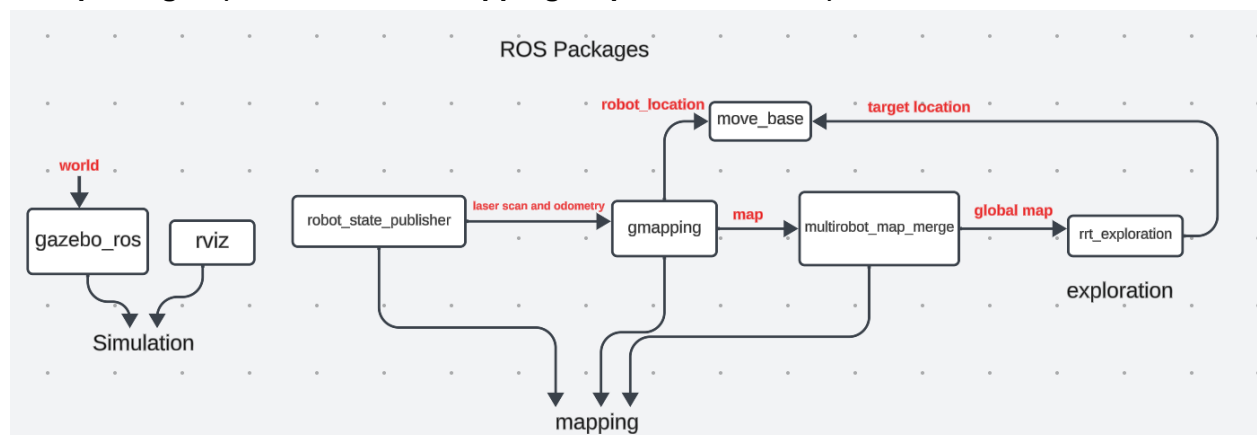
Name	ID	SEC	BN
Asmaa Adel Abdelhamed kawashty	9202285	1	13
Samaa Hazem Mohamed Abdel-latif	9202660	1	31
Norhan Reda Abdelwahed Ahmed	9203639	2	31
Hoda Gamal Hamouda Ismail	9203673	2	33

Supervisor: Eng. Mohamed Shawky

Block diagram of the system



ROS packages (for simulation, mapping, exploration ..., etc)



Note : the gmapping , move base , robot_state_publisher is duplicated for each robot

1-Gazebo Package:

(pkg="gazebo_ros")

- Responsible for simulating the environment.
- Launched using `gazebo_ros` **with a specified world file (house).**

The gazebo_ros package in ROS provides a Gazebo plugin that allows ROS nodes to communicate with the Gazebo simulator. It acts as a bridge between ROS, which is commonly used for robot control and simulation, and Gazebo, a widely-used open-source robotics simulator.

2-Turtlebot Packages (for each robot):

- **Robot State Publisher: Publishes the robot's state (Transform Frames).**

(pkg="robot_state_publisher")

is a component responsible for publishing the state of the robot to the ROS. It provides information about the robot's pose (position and orientation) and its joint states.

The Robot State Publisher is crucial for maintaining an accurate representation of the robot's state within the ROS framework. It publishes the robot's state as a `sensor_msgs/JointState` message, which includes the joint names, positions, velocities, and efforts.

By subscribing to the `sensor_msgs/JointState` message published by the Robot State Publisher, other ROS nodes can access and utilize the robot's state information.

TF frames, short for "Transform Frames," refer to a mechanism used in the ROS (Robot Operating System) ecosystem for managing coordinate transformations in a robotic system. The TF system allows you to keep track of the relationship between different coordinate frames in a robot, which is crucial for tasks such as sensor fusion, motion planning

- **Spawn Model: Spawns the robot model in Gazebo.**

`(pkg="gazebo_ros" type="spawn_model")`

The "Spawn Model" operation in Gazebo refers to the process of adding a robot or object model to the simulation environment. It allows you to introduce virtual models into the Gazebo simulation, such as robots, sensors, or any other objects you want to interact with or simulate.

- **SLAM Gmapping: Performs SLAM for mapping.**

`(pkg="gmapping" type="slam_gmapping")`

SLAM (Simultaneous Localization and Mapping) is a technique used in robotics to create a map of an unknown environment while simultaneously localizing the robot within that environment. Gmapping is a popular SLAM algorithm implemented in ROS.

- **Move Base: Provides navigation capabilities.**

`(ros_multi_tMove Base is a component in ROS)`

`pkg="move_base"`

Move Base is a component in ROS that facilitates autonomous navigation for mobile robots. It integrates **path planning**, obstacle avoidance, and control algorithms to enable a robot to navigate from one location to another in a given environment.b3)/launch/includes/move_base.launch)

3-Map Merging Package:

- **Merges maps generated by individual Turtlebots.**
- **Launched separately with parameters for each Turtlebot's position.**

`<include file="$(find ros_multi_tb3)/launch/includes/multi_tb3_mapmerge.launch">`

The map merging package in ROS is used to combine multiple maps generated by different robots or mapping runs into a single, coherent map of the environment. This is particularly useful in scenarios where multiple robots are exploring an area simultaneously or when mapping is done incrementally over time.

4-TF Transform Broadcasters:

- **world_to_mergedmap_tf_broadcaster:** Static transform from the world to the merged map.
- **world_to_<robot>_tf_broadcaster:** Static transforms from the world to each Turtlebot's map.

(pkg="tf" type="static_transform_publisher")

TF Transform Broadcasters are components in ROS that are responsible for publishing the transformations between coordinate frames. These transformations define the spatial relationships between different parts of a robot or objects in the environment. The TF Transform Broadcasters publish these transformations as TF messages, which are then used by other nodes to perform coordinate frame transformations.

5-RViz:

- **Visualizes the system using a specified RViz configuration file.**

6-exploration

(rrt_exploration)

It is a ROS package that implements a multi-robot map exploration algorithm for mobile robots. It is based on the Rapidly-Exploring Random Tree (RRT) algorithm. It uses occupancy grids as a map representation. The package has 5 different ROS nodes:

1. Global RRT frontier point detector node.
2. Local RRT frontier point detector node.

3. Filter node.
4. Assigner node.
5. opencv-based frontier detector node.

There are 3 types of nodes; nodes for detecting frontier points in an occupancy grid map, a node for filtering the detected points, and a node for assigning the points to the robots. The following figure shows the structure:

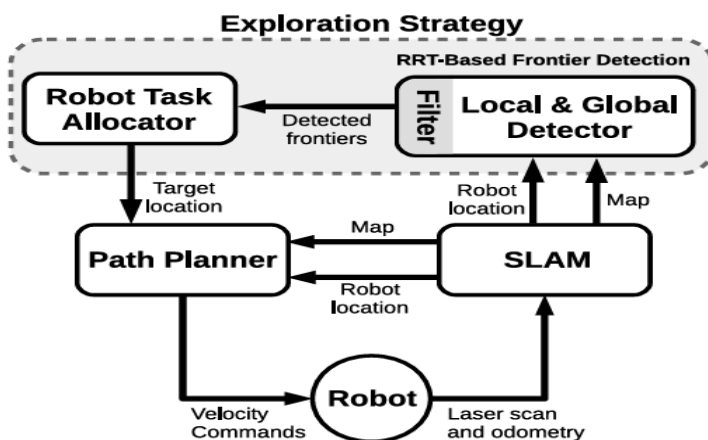
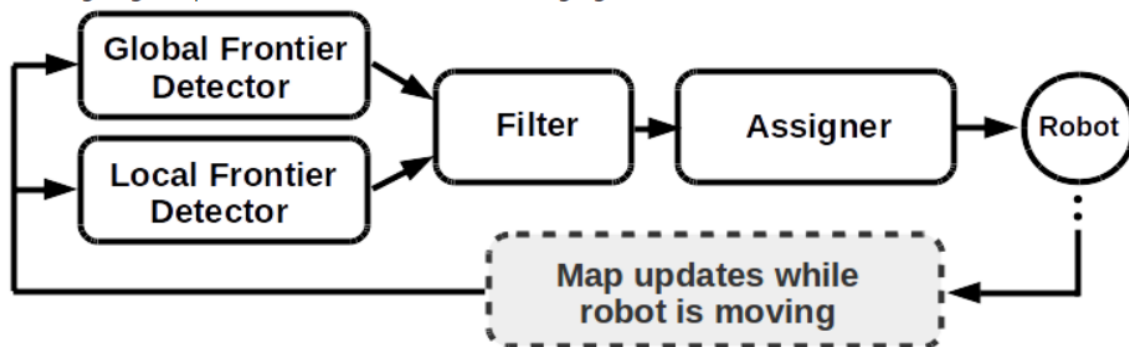


Fig. 6: Implementation diagram

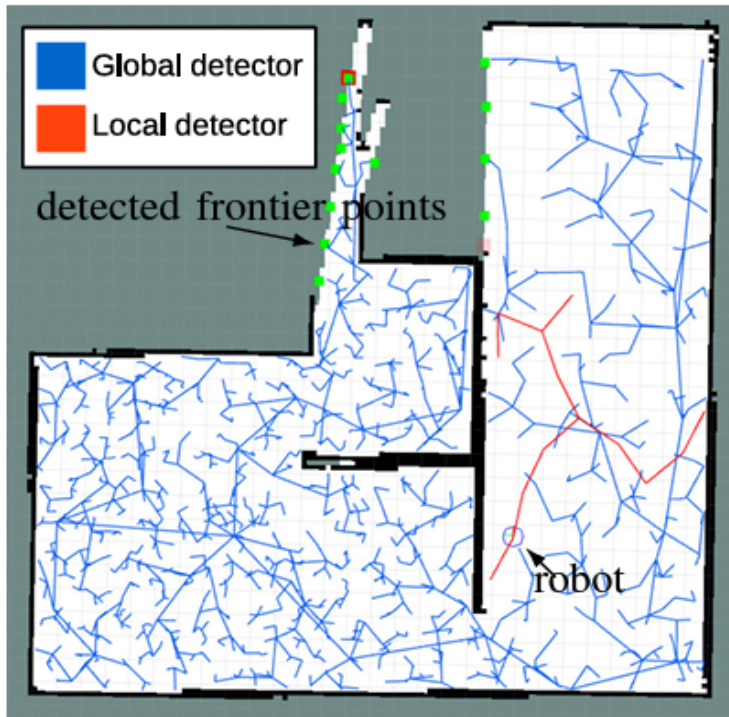


Fig. 3: Global and local frontier detectors

- **Global_rrt_frontier_detector**

The `global_rrt_frontier_detector` node takes an occupancy grid and finds frontier points (which are exploration targets) in it. It publishes the detected points so the filter node can process. In multi-robot configuration, it is intended to have only a single instance of this node running.

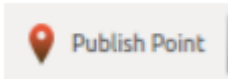
Running additional instances of the global frontier detector can enhance the speed of frontier points detection, if needed.

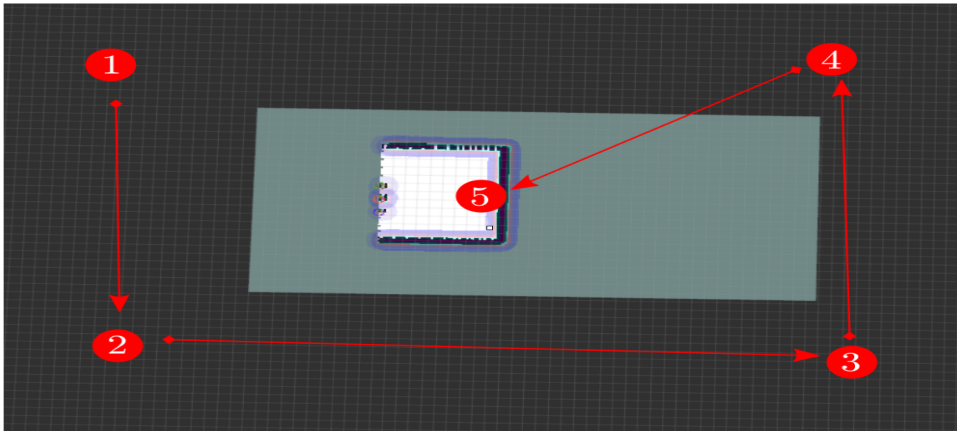
Parameters

1. **`~map_topic`** : This parameter defines the topic name on which the node will receive the map.
2. **`~eta`** : This parameter should be set according to the map size, a very large value will cause the tree to grow faster and hence detect frontier points faster, but a large growth rate also implies that the tree will be missing small corners in the map.

Subscribed Topics

1. **The map** (Topic name is defined by the `~map_topic` parameter)
2. **clicked_points**: The `global_rrt_frontier_detector` node requires that the region to be explored is defined. This topic is where the node receives five points that define the region. The first four points are four defining a square region to be explored, and the last point is the tree starting point. After publishing those five points on this topic, the RRT will start detecting frontier points. The five points are intended to be published from Rviz

using  button.



Published Topics

1. `detected_points` : The topic on which the node publishes detected frontier points.
2. `~shapes` : On this topic, the node publishes line shapes to visualize the RRT using Rviz.

● Local_rrt_frontier_detector

This node is similar to the `global_rrt_frontier_detector`. However, it works differently, as the tree here keeps resetting every time a frontier point is detected. This node is intended to be run alongside the `global_rrt_frontier_detector` node, it is responsible for fast detection of frontier points that lie in the close vicinity of the robot.

In multi-robot configuration, each robot runs an instance of the `local_rrt_frontier_detector`. So for a team of 3 robots, there will be 4 nodes for detecting frontier points; 3 local detectors and 1 global detector. Running additional instances of the local frontier detector can enhance the speed of frontier points detection, if needed.

All detectors will be publishing detected frontier points on the same topic (`/detected_points`).

Parameters

1. `~robot_frame`: The frame attached to the robot. Every time the tree resets, it will start from the current robot location obtained from this frame.
2. `~map_topic`: This parameter defines the topic name on which the node will receive the map.
3. `~eta`: This parameter controls the growth rate of the local RRT.

Subscribed Topics

1. **The map** (Topic name is defined by the `~map_topic` parameter)
2. `clicked_point`: The `local_rrt_frontier_detector` also subscribes to this topic similar to the `global_rrt_frontier_detector`.

Published Topics

1. `detected_points`: The topic on which the node publishes detected frontier points.
2. `~shapes`: On this topic, the node publishes line shapes to visualize the RRT using Rviz.

- **Frontier_opencv_detector**

This node is another frontier detector, but it is not based on RRT. This node uses OpenCV tools to detect frontier points. It is intended to be run alone, and in multi-robot configuration only one instance should be run (running additional instances of this node does not make any difference).

Originally this node was implemented for comparison against the RRT-based frontier detectors. Running this node alongside the RRT detectors (local and global) may enhance the speed of frontier points detection.

Note: You can run any type and any number of detectors, all the detectors will be publishing on the same topic which the filter node (will be explained in the following section) is subscribing to. On the other hand, the filter will pass the filtered frontier points to the assigner in order to command the robots to explore these points.

Parameters

- `~map_topic`: This parameter defines the topic name on which the node will receive the map.

Subscribed Topics

- **The map** (Topic name is defined by the `~map_topic` parameter)

Published Topics

- `etected_points` : The topic on which the node publishes detected frontier points.
- `shapes`: On this topic, the node publishes detected points to be visualized using Rviz.

● Filter

The filter nodes receives the detected frontier points from all the detectors, filters the points, and passes them to the assigner node to command the robots. Filtration includes the selection of old and invalid points, and it also discards redundant points.

Parameters

- `~map_topic`: This parameter defines the topic name on which the node will receive the map. The map is used to know which points are no longer frontier points (old points).
- `~costmap_clearing_threshold`: Any frontier point that has an occupancy value greater than this threshold will be considered invalid. The occupancy value is obtained from the costmap.
- `~info_radius`: The information radius used in calculating the information gain of frontier points.
- `~goals_topic` : defines the topic on which the node receives detected frontier points.
- `~n_robots`: Number of robots.
- `~rate`: node loop rate (in Hz).

Subscribed Topics

- **The map (Topic name is defined by the `~map_topic` parameter) .**
- `robot_x/move_base_node/global_costmap/costmap` : where x (in `robot_x`) refers to the robot's number.

The filter node subscribes for all the costmap topics of all the robots, the costmap is required therefore. Normally, costmaps should be published by the navigation stack (after bringing up the navigation stack on the robots, each robot will have a costmap). For example, if `n_robots=2`, the node will subscribe to: `robot_1/move_base_node/global_costmap/costmap` and `robot_2/move_base_node/global_costmap/costmap`. The costmaps are used to delete invalid points.

Note: Namespaces of all the nodes corresponding to a robot should start with `robot_x`. Again x is the robot number.

- The goals topic (Topic name is defined by the `~goals_topic` parameter)([geometry_msgs/PointStamped Message](#)): The topic on which the filter node receives detected frontier points.

Published Topics

- `frontiers` ([visualization_msgs/Marker Message](#)): The topic on which the filter node publishes the received frontier points for visualization on Rviz.
- `centroids` ([visualization_msgs/Marker Message](#)): The topic on which the filter node publishes only the filtered frontier points for visualization on Rviz.
- `filtered_points` ([PointArray](#)): All the filtered points are sent as an array of points to the assigner node on this topic.

• Assigner

This node receives target exploration goals, which are the filtered frontier points published by the filter node, and commands the robots accordingly. The assigner node commands the robots through the `move_base_node`. This is why you have brought up the navigation stack on your robots.

Parameters

- `~map_topic (string, default: "/robot_1/map")`: This parameter defines the topic name on which the node will receive the map. In the single robot case, this topic should be set to the map topic of the robot. In the multi-robot case, this topic must be set to a global merged map.
- `~info_radius(float, default: 1.0)`: The information radius used in calculating the information gain of frontier points.
- `~info_multiplier(float, default: 3.0)`: The unit is meter. This parameter is used to give importance to information gain of a frontier point over the cost (expected travel distance to a frontier point).
- `~hysteresis_radius(float, default: 3.0)`: The unit is meter. This parameter defines the hysteresis radius.
- `~hysteresis_gain(float, default: 2.0)`: The unit is meter. This parameter defines the hysteresis gain.
- `~frontiers_topic (string, default: "/filtered_points")`: The topic on which the assigner node receives filtered frontier points.
- `~n_robots(float, default: 1.0)`: Number of robots.

- `~delay_after_assignment(float, default: 0.5)`: The unit is seconds. It defines the amount of delay after each robot assignment.
- `~rate(float, default: 100)`: node loop rate (in Hz).

Subscribed Topics

- The map (Topic name is defined by the `~map_topic` parameter) ([nav_msgs/OccupancyGrid](#)).
- Filtered frontier points topic (Topic name is defined by the `~frontiers_topic` parameter) ([PointArray](#)).

Published Topics

The assigner node does not publish anything. It sends the assigned point to the `move_base_node` using Actionlib (the assigner node is an actionlib client to the `move_base_node` actionlib server).

Algorithms

Mapping

GMapping, also known as Grid-based FastSLAM, is a popular algorithm used for Simultaneous Localization and Mapping (SLAM) in robotics. It is particularly used for mapping environments using range sensor data, such as laser scanners.

Here's an overview of how GMapping works:

1. **Initialization:** GMapping starts with an initial occupancy grid map, representing the environment, where each cell is marked as either occupied or free. The robot's initial pose is also estimated.
2. **Motion Update:** As the robot moves, motion updates are performed to predict the robot's new pose based on odometry information. The motion model takes into account the robot's control commands and estimates the new pose using probabilistic filtering techniques, such as particle filters.
3. **Measurement Update:** When the robot receives range sensor data, such as laser scans, measurement updates are performed. GMapping uses the sensor data to update the occupancy grid map, refining the estimates of occupied and free cells.
4. **Mapping:** GMapping uses a grid-based representation to maintain the occupancy probabilities of each cell in the map. The algorithm updates the probabilities based on both motion and measurement updates. Cells with high probabilities are considered occupied, while cells with low probabilities are considered free.
5. **Loop Closure Detection:** GMapping detects loop closures, which occur when the robot revisits a previously visited location. Loop closure detection is crucial for correcting accumulated errors in the robot's pose estimate and map. Various techniques, such as scan matching and loop closure algorithms (e.g., the likelihood of matching features), are used for loop closure detection.
6. **Data Association:** GMapping associates the sensor measurements with the corresponding map features. It determines which map cells are likely to be responsible for the observed sensor readings. This step helps in aligning the sensor data with the map, improving the accuracy of the map.
7. **Particle Filtering:** GMapping employs a particle filtering approach to represent the belief distribution over the robot's pose and the map. Multiple particles (hypotheses) are maintained, each representing a possible pose and map configuration. The particles are updated and resampled based on the motion and measurement updates.
8. **Map Maintenance:** GMapping continuously updates and maintains the occupancy grid map as new sensor data is received. It refines the map by adjusting the probabilities of occupied and free cells based on the sensor measurements.

By iteratively performing motion updates, measurement updates, loop closure detection, and data association, GMapping builds an accurate map of the environment while simultaneously estimating the robot's pose. The resulting map can be used for localization, obstacle avoidance, and path planning.

Important things that affects slam

Loop Closure Detection and Scan Matching are two key components of Simultaneous Localization and Mapping (SLAM) algorithms used in robotics to improve mapping and localization accuracy. Here's an explanation of each:

Loop Closure Detection:

Loop closure detection refers to the process of identifying when a robot revisits a previously visited location or passes through a similar trajectory. It helps in detecting and correcting accumulated errors in the robot's pose estimate and map.

When a loop closure is detected, it indicates that the current pose estimate may be incorrect due to drift or errors in the odometry data. By detecting loop closures, SLAM algorithms can refine the robot's pose estimate and improve the consistency and accuracy of the map.

Loop closure detection is typically performed by comparing the current sensor data (e.g., laser scans, camera images) with the data collected during previous visits to the same or similar areas. Various techniques can be used for loop closure detection, such as feature-based matching, appearance-based methods, or geometric approaches. These techniques aim to find correspondences between the current sensor data and previously recorded sensor data to identify similarities or overlaps.

Scan Matching:

Scan matching is a technique used to align or match consecutive laser scans or other range sensor measurements to estimate the robot's pose change between the scans. It helps in estimating the robot's motion and refining the localization.

Scan matching algorithms aim to find the transformation (translation and rotation) that best aligns two consecutive scans. The goal is to minimize the differences or errors between corresponding points or features in the scans. By finding the optimal transformation, the algorithm estimates the robot's motion, compensating for noise, drift, and other factors.

There are different methods for scan matching, including iterative closest point (ICP), point-to-point matching, point-to-line matching, or feature-based matching. These methods use different metrics, optimization algorithms, and techniques to find the best alignment between scans.

Scan matching is crucial for accurate motion estimation and localization in SLAM. It helps in reducing the accumulation of errors over time and improves the quality of the estimated robot's trajectory and the resulting map.

Both loop closure detection and scan matching play important roles in SLAM algorithms, complementing each other to enhance localization accuracy and map consistency. Loop closure detection helps in correcting long-term errors and closing loops in the trajectory, while scan matching contributes to precise short-term motion estimation and alignment of consecutive scans.

Exploration

The Rapidly-exploring Random Tree (RRT) algorithm is a popular motion planning algorithm used in robotics to efficiently explore and navigate complex environments. The RRT algorithm is particularly useful for problems involving high-dimensional state spaces and environments with obstacles.

Here is a general overview of the RRT exploration algorithm:

1. Initialization:
 - Start with an initial configuration, representing the current pose of the robot or starting position.
 - Create an empty tree data structure, initially containing only the starting configuration.
2. Expansion:
 - Randomly sample a new configuration in the state space.
 - Find the nearest node in the existing tree to the sampled configuration.
 - Extend the tree by connecting the nearest node to the sampled configuration, considering constraints such as collision avoidance with obstacles.
 - Add the newly created node to the tree.
3. Repeat Expansion:
 - Repeat the expansion process by sampling new configurations and connecting them to the existing tree until a termination condition is met.
 - The termination condition could be reaching a desired goal configuration, exploring a specific area, or reaching a maximum number of iterations.
4. Path Generation:
 - Once the termination condition is met, a feasible path from the initial configuration to the goal configuration can be obtained by backtracking through the tree.
 - Starting from the goal node, trace the branch of the tree back to the initial configuration, forming the path.

The RRT algorithm is characterized by its randomness and exploration-oriented nature. It gradually builds a tree structure that grows towards unexplored areas of the environment. The randomness in sampling new configurations allows the algorithm to explore different regions, making it suitable for exploration tasks.

Several variants and enhancements to the basic RRT algorithm exist, such as RRT* and RRTConnect, which aims to improve the quality of the generated paths and optimize the exploration process.

Sample Odometry Motion Model

1. Algorithm **sample_motion_model**(u, x):

$$u = \langle \delta_{rot1}, \delta_{rot2}, \delta_{trans} \rangle, x = \langle x, y, \theta \rangle$$

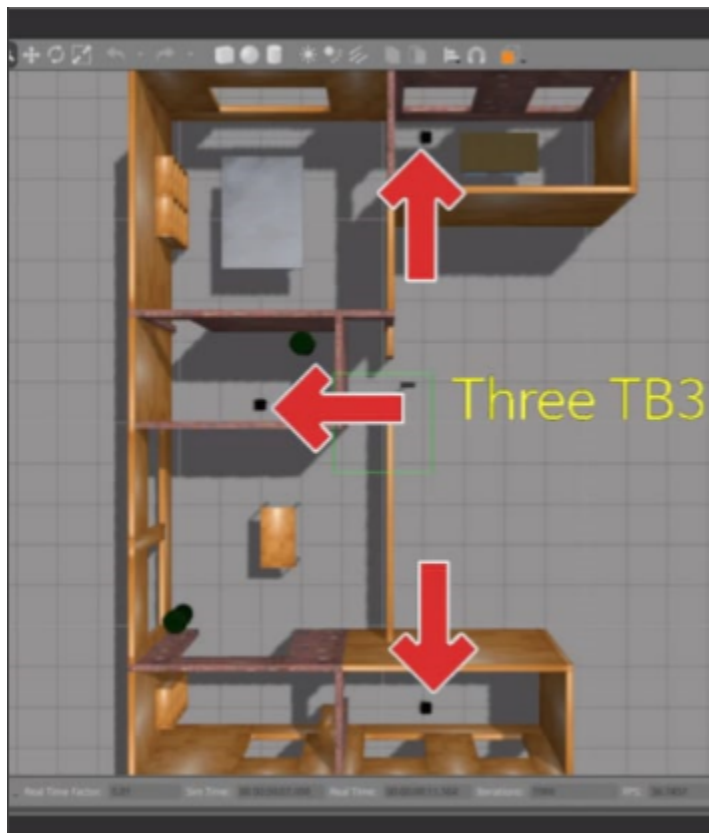
old pose

1. $\hat{\delta}_{rot1} = \delta_{rot1} + \text{sample}(\alpha_1 |\delta_{rot1}| + \alpha_2 \delta_{trans})$
2. $\hat{\delta}_{trans} = \delta_{trans} + \text{sample}(\alpha_3 \delta_{trans} + \alpha_4 (|\delta_{rot1}| + |\delta_{rot2}|))$
3. $\hat{\delta}_{rot2} = \delta_{rot2} + \text{sample}(\alpha_5 |\delta_{rot2}| + \alpha_6 \delta_{trans})$
4. $x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1})$
5. $y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1})$
6. $\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$

sample_normal_distribution

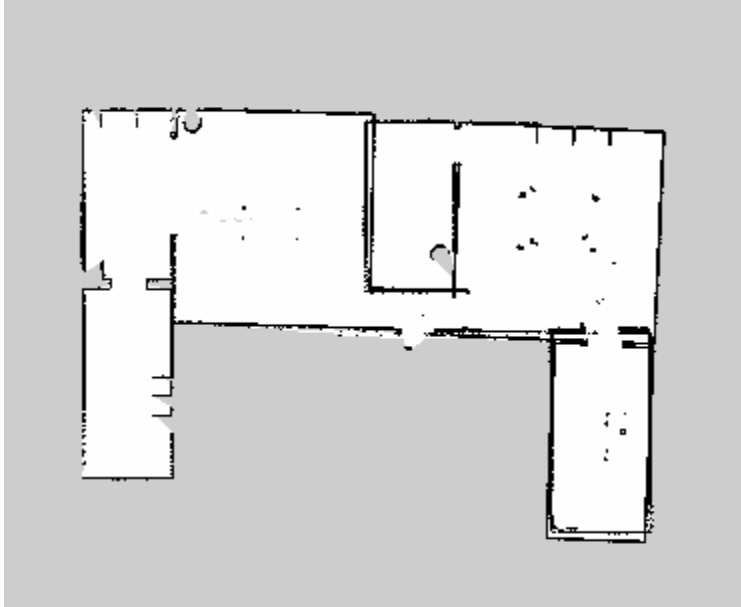
7. Return $\langle x', y', \theta' \rangle$

The world that we used

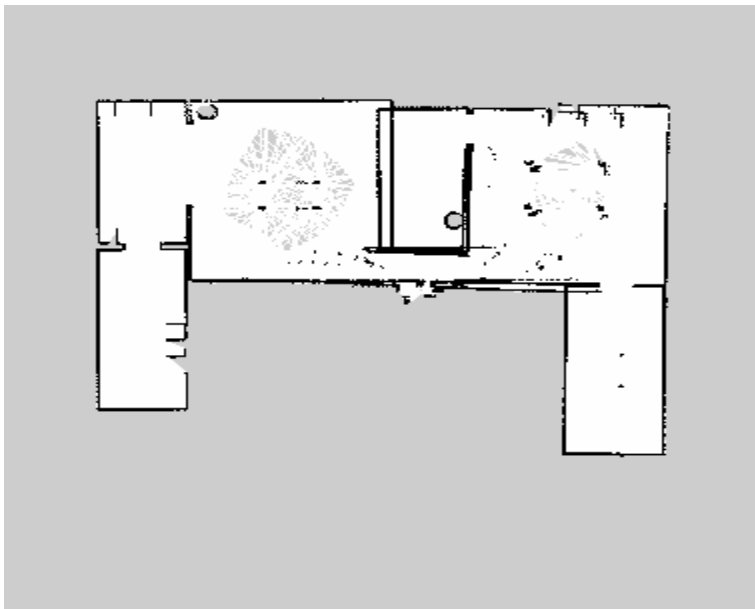


The result of our system

1-multi robot map exploration without odometry Noise And with sensor Noise



2-multi robot map exploration with odometry Noise And with sensor Noise



Note the odom noise cause map distortion a little bit

How to run the code

Open four terminals

1-terminal 1

```
# roscore
```

2- terminal 2

```
# source ./devel/setup.bash  
# export TURTLEBOT3_MODEL=waffle_pi  
# roslaunch ros_multi_tb3 multiple_tb3_house.launch
```

3-terminal 3

```
# source ./devel/setup.bash  
# export TURTLEBOT3_MODEL=waffle_pi  
# roslaunch rrt_exploration three_robots.launch
```

4- terminal4 (Saving Map)

```
# rosrun map_server map_saver -f mymap map:=/map
```

Good Resource for rrt exploration paper

<https://hasauino.github.io/assets/papers/rrtexploration.pdf>