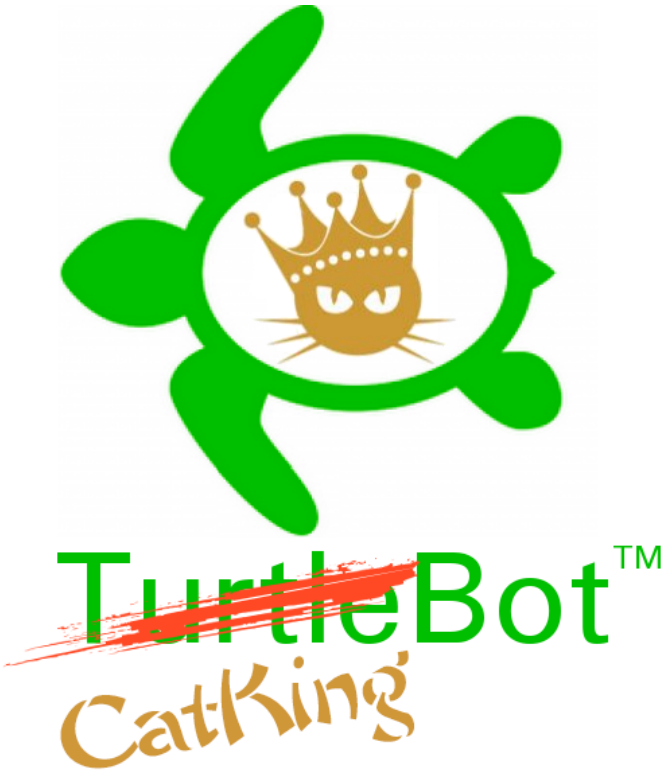
UNIVERSITY OF BURGUNDY



Robotics Project Report

Devesh ADLAKHA
S.Mehdi MOHAIMENIANPOUR
Diego NUNES DE ALMEIDA

Bachelor in Computer Vision

May 11, 2015

Abstract

This report presents CatKing, our project as part of the Robotics Engineering 2 course module in the second semester of study in the Bachelor in Computer Vision program at the University of Burgundy, France. The course objective was an introduction to the Robot Operating System (ROS) using the TurtleBot 2 running on the Hydro Medusa distribution. The project requirements were: motion control, integration of a planar laser range finder sensor, and navigation and localization. The task capabilities of CatKing are modeled after a home robot application, influenced particularly by the RoboCup@Home league [1]. The functionalities include: surveillance, global localization, virtual joystick control, QR bar code reading, face detection and recognition, and speech synthesis. These tasks and their implementation details are presented independently, followed by their state machine integration. The challenges faced and our solutions are described, followed by a discussion of the future work for this project. The source code can be found in the appendix, and is available as open-source in the following GitHub repository: https://github.com/Voidminded/CatKing.

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

The objective of this project assignment was to introduce ubiquitous tools in robotics, the TurtleBot 2 and ROS. The project requirements comprised fundamental tasks in robotics:

1. Motion control.
2. Integration of an external sensor (the RPLIDAR planar laser range finder [2]).
3. Simultaneous localization and mapping (SLAM), path-planning and navigation.

Beyond these requirements, our objective was to investigate additional tasks, explore existing packages and gain familiarity with the ROS framework. The RoboCup@Home league for domestic application robots served as the primary reference in determining tasks. The 2015 edition of the competition includes the following challenges:

- Human-robot interaction: speech recognition and synthesis.
- Object detection/recognition and manipulation.
- Navigation: path planning and navigation in a dynamic environment, with feedback of success and failure. The obstacle type may be inferred or recognized to effectively adjust navigation behavior, for example, a human could be asked to clear the path, whereas a door could be opened to facilitate navigation.
- Person recognition: learning algorithms for face/gender/pose recognition, and extending this functionality to a crowd of people.
- Entertainment: amusing features such as dancing.
- Open challenges: a set of complex high-level tasks, such as cooking a meal, ironing clothes, pushing a wheelchair, among others.

Based on these references, and upon surveying the existing packages in ROS, the following tasks were selected, which constitute the capabilities of CatKing:

1. Surveillance: patrol the arena following a defined sequence of goal poses, collecting data of interest, such as camera images and microphone audio.
2. Global localization: achieve localization autonomously within the map.

3. Virtual joystick: a novelty feature for manual control.
4. QR barcode reading: decode messages, and as a means to change state in the state machine implementation.
5. Face detection and recognition: as part of the human-robot interaction.
6. Speech synthesis: communication of internal state and feedback from processes.

The project development was carried out on a TurtleBot 2 running on the ROS Hydro Medusa distribution.

CHAPTER 2

SLAM AND NAVIGATION

Simultaneous localization and mapping (SLAM), path planning and navigation are essential to an autonomous robot. This chapter presents their incorporation and configuration in our project.

2.1 MOTION CONTROL

The hierarchy of motion control in ROS was studied following [3] , and is depicted in Figure 2.1. The results and comparisons of the levels of motion control are well documented, and the emphasis throughout this chapter is on our configuration of the localization and navigation packages.

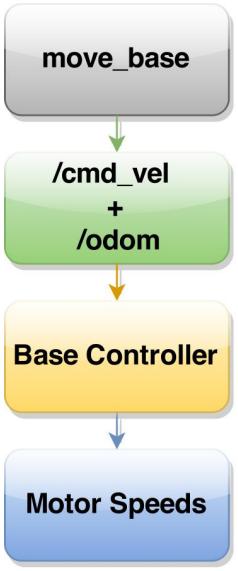


Figure 2.1: Hierarchy of motion control

2.2 INTEGRATION OF THE RPLIDAR

The RPLIDAR [2] is a low cost planar laser scanner (LIDAR) developed by RoboPeak. It offers a working range of 6 meters in the [0 360]° angular space. Its primary purpose in our project is in SLAM and navigation.

The integration was successfully done by following the procedure available in the Robotics Lab GitHub repository (https://github.com/roboticslab-fr/rplidar-turtlebot2) and it is shown in Figure 2.2.

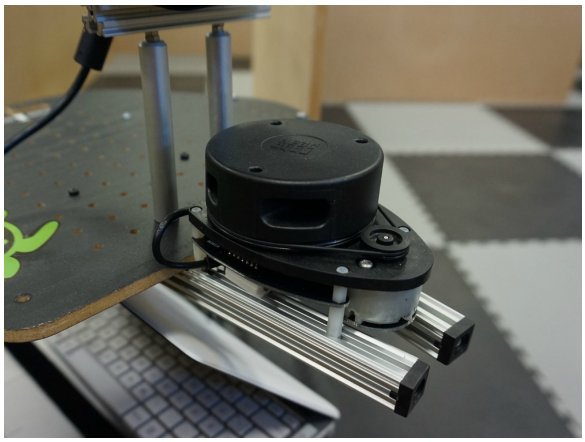


Figure 2.2: RP-LIDAR

2.3 Mapping

2.3.1 Arena

We proposed a design for the arena, which was approved through a consensus with the other groups. The design was created with the following objectives:

1. Requires fine-tuning of navigation parameters for robust autonomous navigation. The construction of the arena with relatively tight spaces and corridors, and multiple obstacles necessitates proper selection of the navigation parameters.
2. Contains sufficient open space to show navigation results (such as moving in a square).
3. Boundaries are asymmetrical or distinguishable to assist in localization.
4. Resembles a domestic environment with areas identifiable as rooms, corridors, and free space.

The original design for the arena is shown in Figure 2.3. This design was modified during its physical assembly to ensure ample open space, in keeping with the second design objective. The arena in use is shown in Figure 2.4.

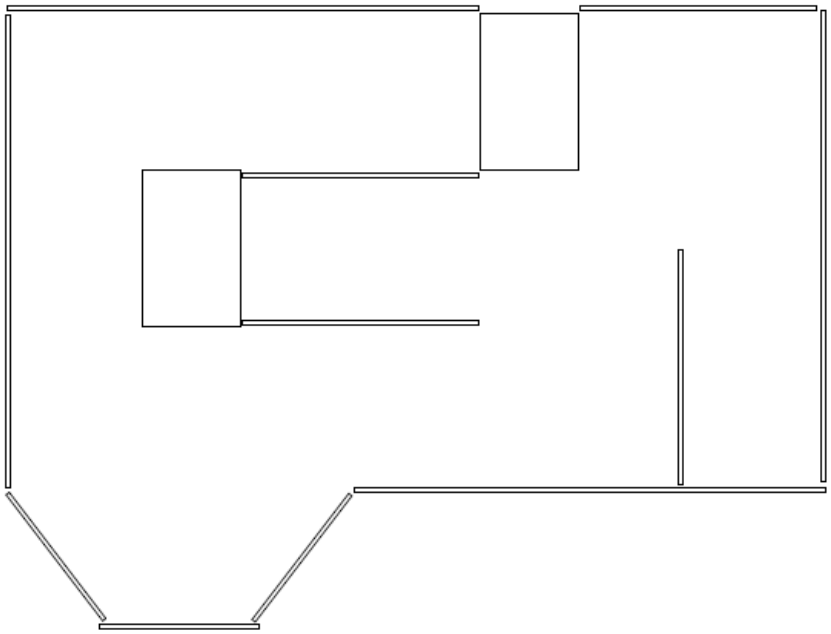


Figure 2.3: First design of the arena

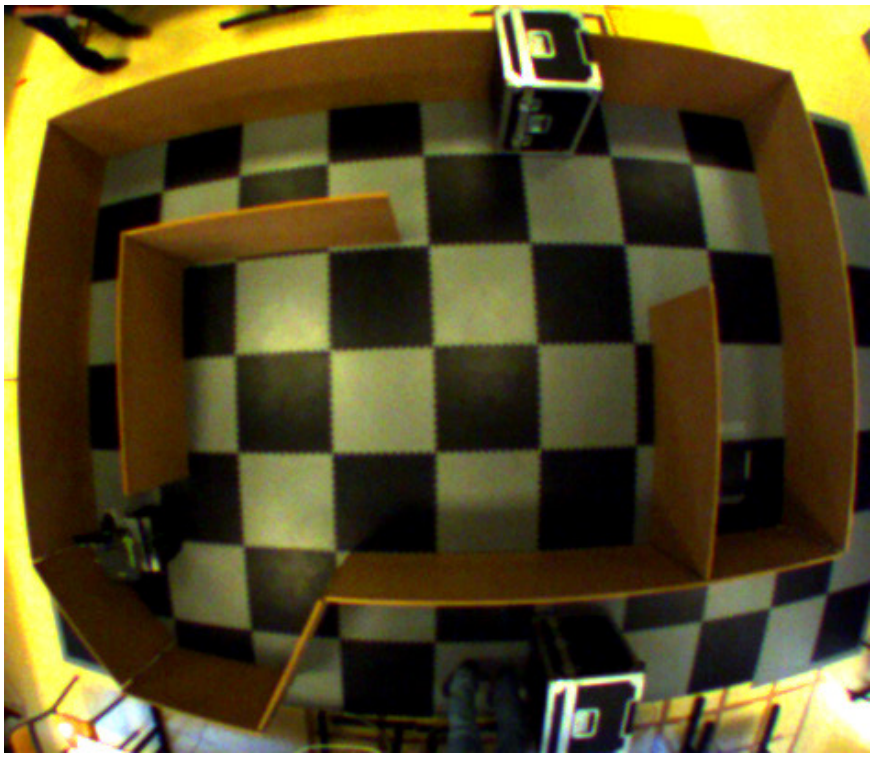


Figure 2.4: Arena in use

2.3.2 Map Creation

Several packages implementing SLAM exist, and the `rplidar_gmapping` package from [3] was used to create a map of the arena.

The `gmapping` (and by extension, `rplidar_gmapping`) package is well established, requiring laser scan and odometry data around the desired area to map. Teleoperation with the keyboard was used to maneuver the TurtleBot around the arena. Since the generated map was to be processed before use, the map creation process was not required to be meticulous.

The following package parameters were tuned:

- Minimum score: increased to a high value of 10,000 to prevent a particle with low confidence to be accepted.
- Map update interval: reduced from 30 to 2 Hz. Though a high update frequency is desirable, it was reduced empirically to comply with the processing power of the netbook on the Turtlebot.
- Particles: restored to the default value of 30 (from 1 in the launch file).
- Delta: the resolution of the map was reduced by a factor of 5 to 0.01 expecting finer cells in the cost grid to supplement navigation.

Figure 2.5 shows the resultant map, and Figure 2.6, the result after processing in Photoshop.

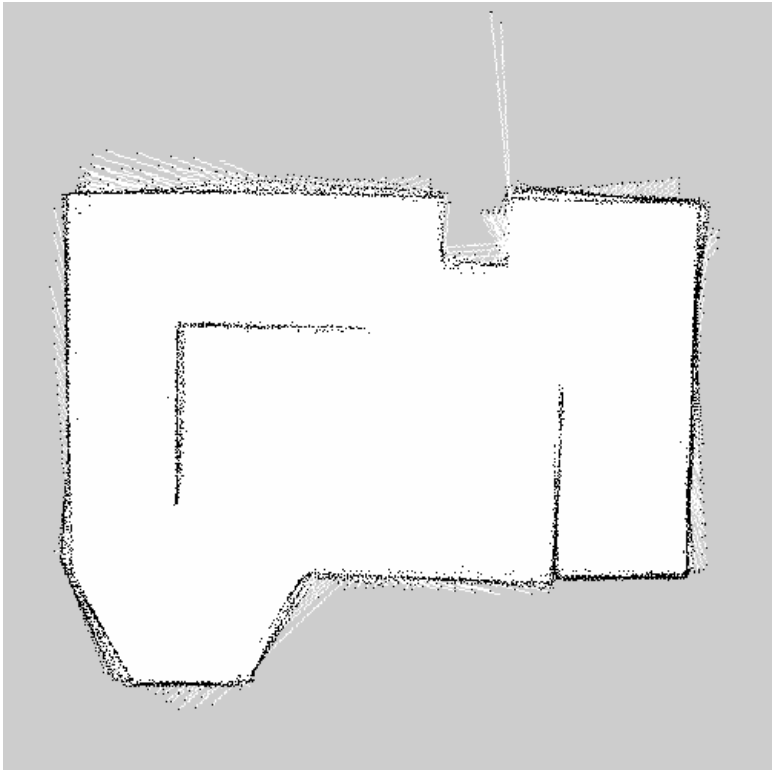


Figure 2.5: Resultant map from `gmapping`

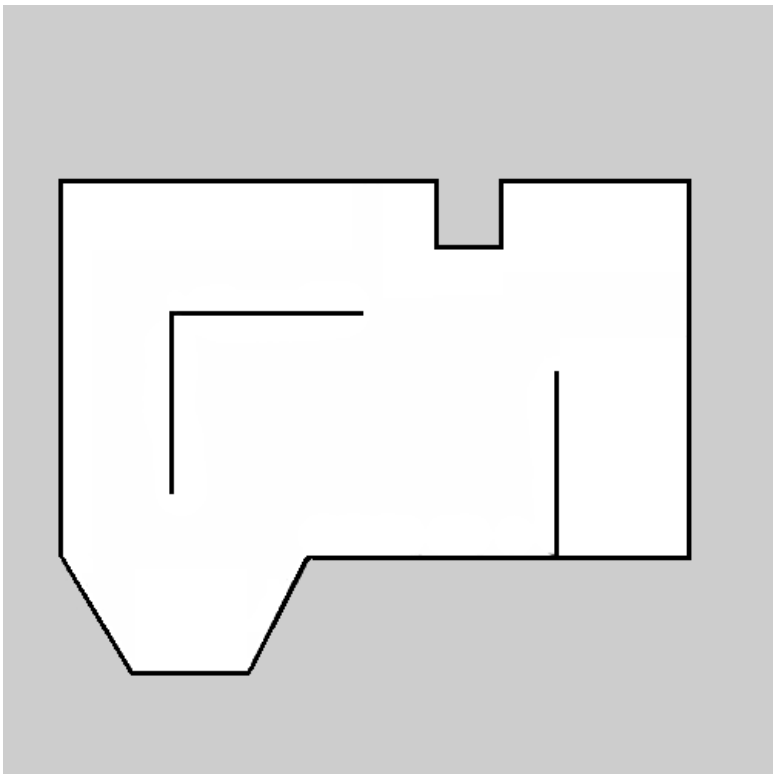


Figure 2.6: Map after processing in Photoshop

2.4 LOCALIZATION

The `amcl` package from [3] was used for localization within the map. The Adaptive Monte Carlo Localization (AMCL) algorithm maintains multiple hypothesis of the robot pose as particles within the map. Using laser scan data as its measurement model and odometry as its motion model, these particles of the filter may converge, leading to localization. As the SLAM package, it is pretty standard in robotics and in ROS, and was used with its default configuration.

Figure 2.7 shows the localization process. The particles convergence upon motion while receiving laser scan data. The motion may be through manual control or a higher level path planner.

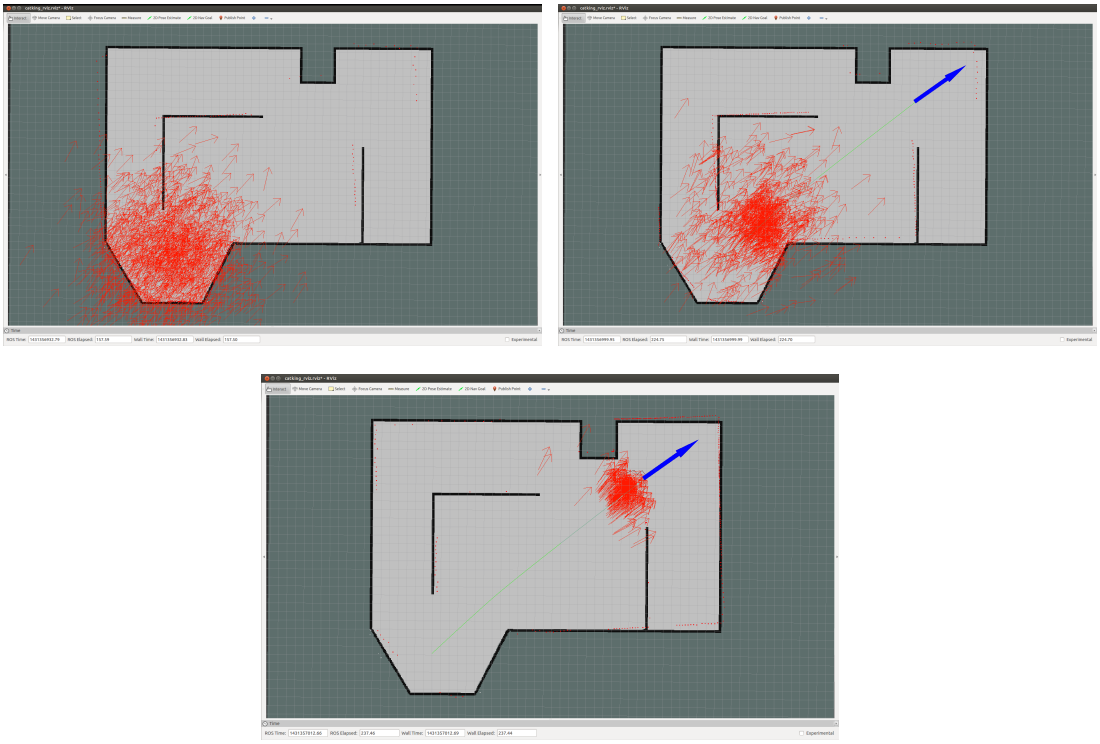


Figure 2.7: Convergence in localization

2.5 Path-planning and Navigation

The following requirements were established for navigation in decreasing order of priority:

1. Safety: involves collision free motion, from the static arena as well dynamically added or moving objects.
2. Smooth trajectories: preferring linear motion, and reducing tendencies to rotate or oscillate.

The `move_base` package manages path planning and navigation to a goal pose. Figure 2.8 depicts the navigation stack, centered around `move_base`.

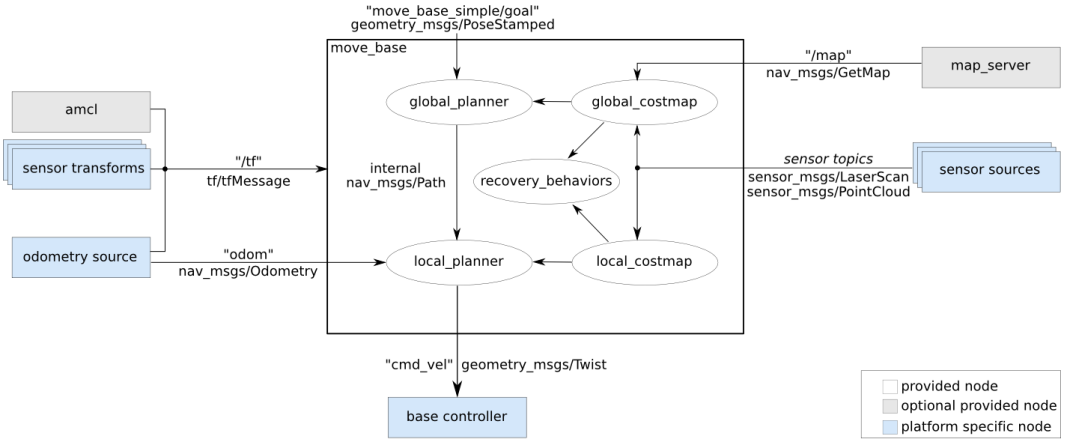


Figure 2.8: Navigation stack

Three plans are maintained in the path planning algorithm:

- Planner plan: the complete plan generated by the global planner.
- Global plan: part of the planner plan till a waypoint.
- Local plan: trajectory followed due to the velocity commands of the local planner.

As such, the global planner determines the overall plan to the goal pose, while the local planner directs motion to a waypoint in this plan, while accounting for obstacles through the laser scan data.

Four configuration files govern the motion behavior of the TurtleBot:

1. `costmap_comom_params`
2. `global_costmap_params`
3. `local_costmap_params`
4. `base_local_planner`

The modification of these parameters and their justification to suit our navigation objectives is presented, organized by their configuration files:

1. `costmap_comom_params`: these parameters are used for both the local and global cost maps.
    - Robot footprint: the model of the TurtleBot was modified to better reflect its actual structure. Figure 2.9 shows the two footprints we tested. While, the footprint above has a large safety margin and performed very safe navigation within the map, tight spaces could lead to oscillation or difficulty in rotating in-place. Thus, we progressed to the latter footprint, which performed better in these two situations.

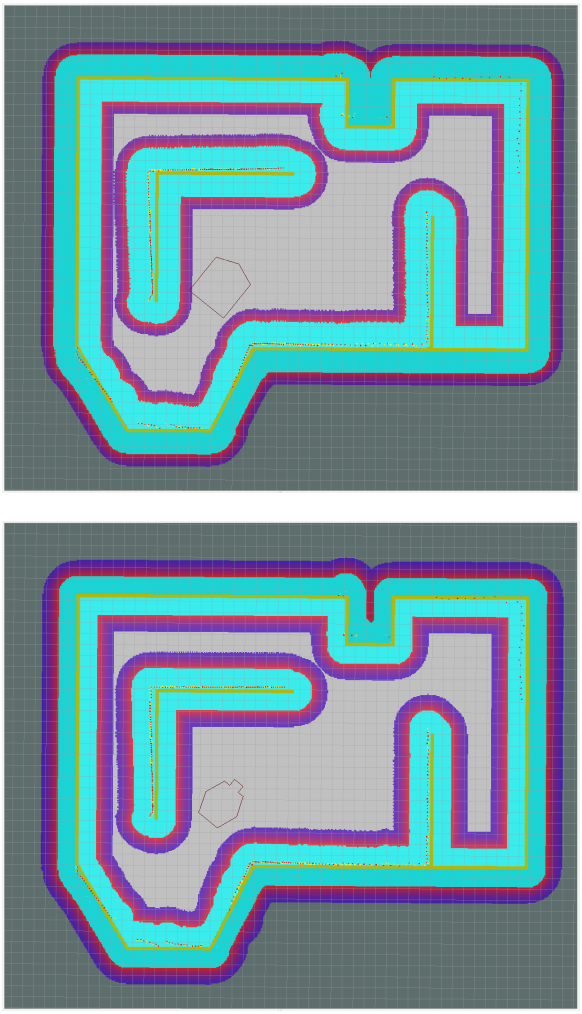


Figure 2.9: Footprints tested

- Inflation radius: set to 0.3. It represents the radius till which a cost is propagated in the cost map. As such, a higher cost was set close to the obstacle, which is particularly useful in tight corridors.
- Scan observation persistence: set to 0.0. This maintains only the current laser scan data in the cost map, which is exactly desirable in this project.

2. `global_costmap_params`:

   - Update and publish frequency: set to an empirically determined 3.0 Hz, which provided the netbook with sufficient time (in general) to process and update the map, and to publish the result for display in Rviz.
   - Resolution: set to 0.01 corresponding to the resolution of the map.

3. `local_costmap_params`:

   - Update and publish frequency: set to empirically determined 5.0 Hz, which provided the netbook with sufficient time (in general) to process and update the map and publish the result for display in Rviz visualization.
   - Resolution: set to 0.01 corresponding to the resolution of the map.

- Width, height: set to 1.75, refer section (surveillance) for details.

4. `base_local_planner`:

   - Recovery behavior enabled: true.
   - Clearing rotation enabled: true (refer to Global Localization section).
   - Max velocity x: 0.2.
   - Max rotational velocity 0.5.
   - Acceleration limit x: 1.0.
   - Acceleration limit theta: 1.0.
   - Publish cost grid: true. This allows visualization of the cost grid, and was not particularly used.

The limits on the velocity and acceleration led to following the global plan more closely. This was desirable since the local planner often led to motions with significant drift from the global plan, potentially forcing the robot to areas of high cost. Ultimately, this could lead to oscillations or even erratic motion.

As shown in Figure 2.10, reducing the motion of the TurtleBot led to smoother trajectories close to the global plan.

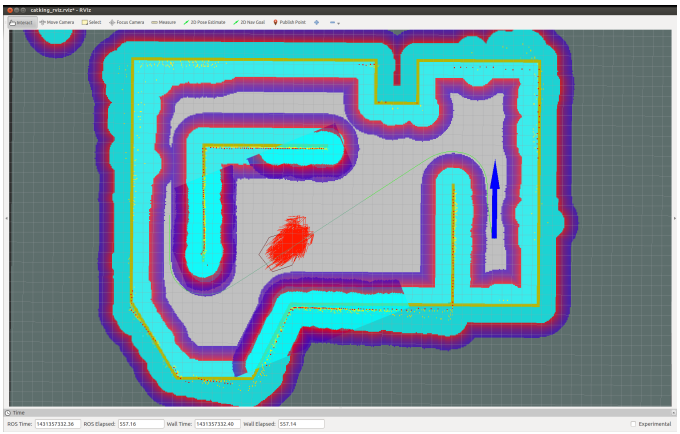


Figure 2.10: Global plan trajectory following with reduced motion

2.6 Localization and Navigation

The `amcl` node is used in conjunction with `move_base` for navigation within the map. Given an initial pose estimate (for instance, through Rviz), a goal pose to `move_base` initiates motion, which aids in localization, and thereby navigation.

Accurate localization is essential for safe navigation, and errors in localization may be accounted for in the footprint of the robot. However, given the narrow spaces in the map (the tightest section being around 70 cm), the localization errors could not really be accounted for within the footprint, and safe navigation relies on accurate localization.

2.7 RVIZ CONFIGURATION

Rviz was custom configured following the tutorial in [4] to display the footprint and all path planner plans, as shown in Figure 2.11. This configuration file `catking_rviz.rviz` is used throughout the project.

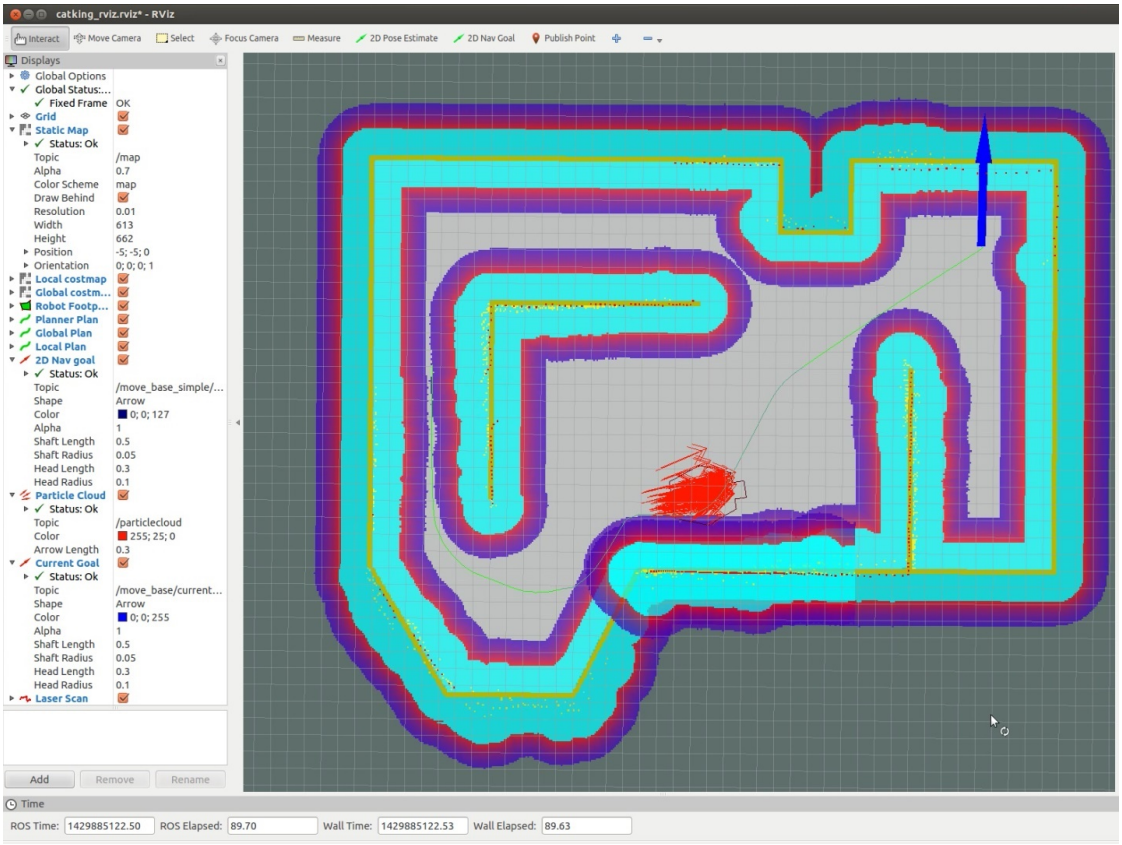


Figure 2.11: Rviz custom configuration

CHAPTER 3

FEATURES

The task capabilities of CatKing are presented, followed by the state machine integration.

3.1 SURVEILLANCE

Surveillance involves patrolling the arena in a defined sequence of locations, and collecting data of interest such as camera images and microphone audio. The `rbx1_nav` package [3] includes a node implementing patrol, which was modified to follow a defined (rather than random) sequence of locations for our map.

Figure 3.1 shows the six locations selected in the map for patrolling:

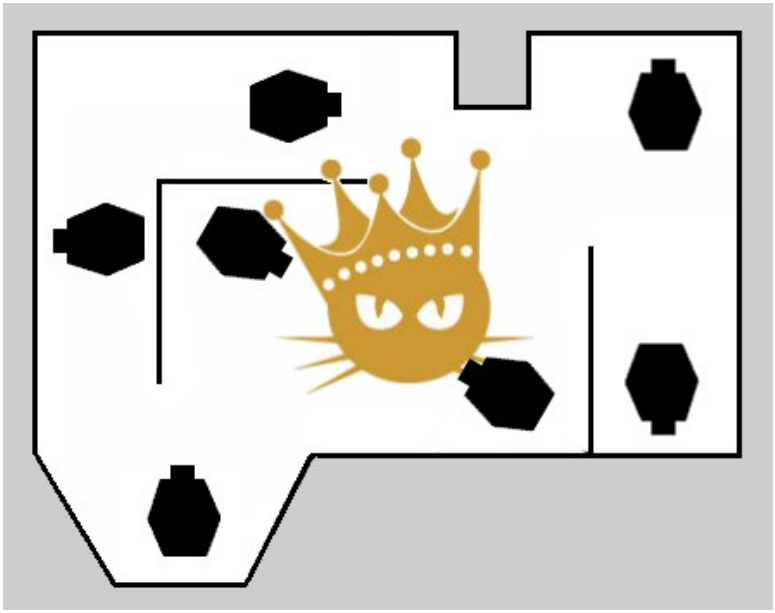


Figure 3.1: Patrol goal poses

The patrol node launches two other nodes: `amcl` for localization, and `move_base` for path planning and navigation. The initial pose estimate required for localization and initiating patrol may be provided interactively through Rviz. The path planner then determines a path to the first goal pose, and given a reasonable initial pose estimate, subsequent motion leads to quick convergence in localization. The patrol mode is the continuation of path planning and navigation to successive goal locations determined

using the localized pose.

Figure 3.2 shows the result of patrol between two of the selected goal locations through the Rviz visualization.

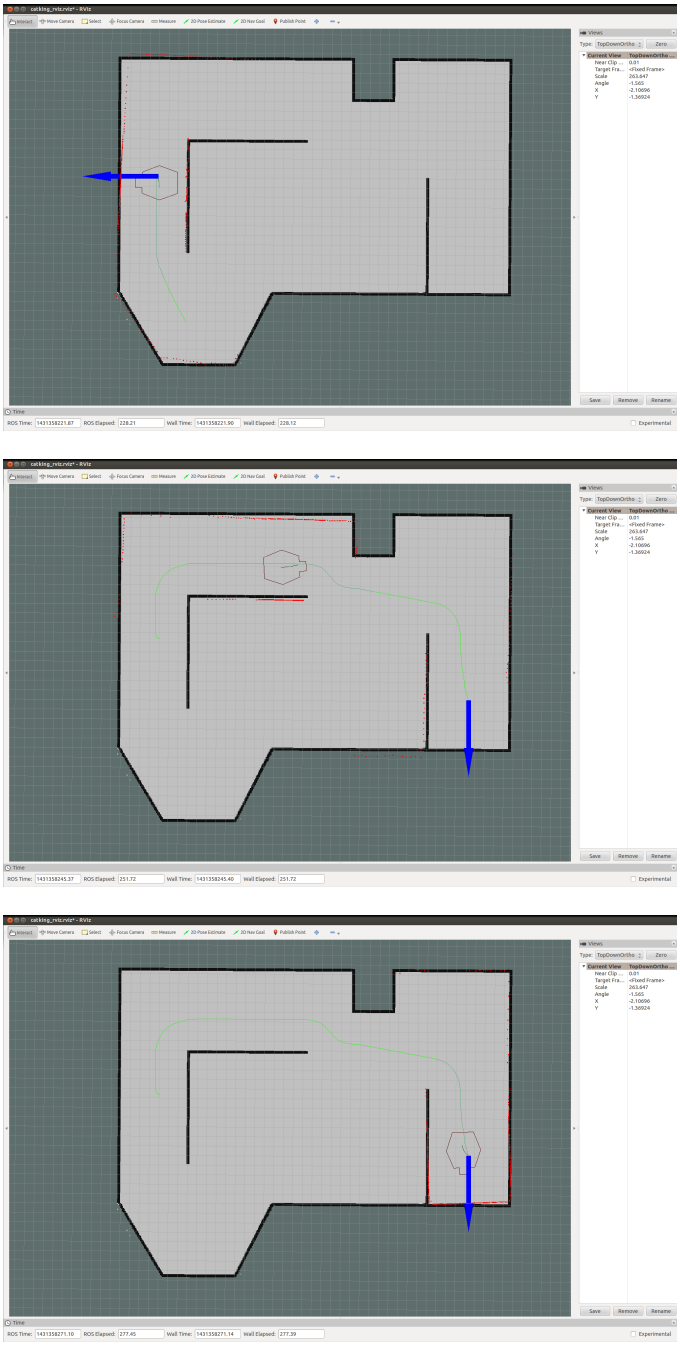


Figure 3.2: Patrol results between two goal poses

An attained goal pose is maintained for a configurable rest period of ten seconds. The node provides feedback with statistics such as the running time, distance covered, goal sequence and success rate, as shown in Figure 3.3.

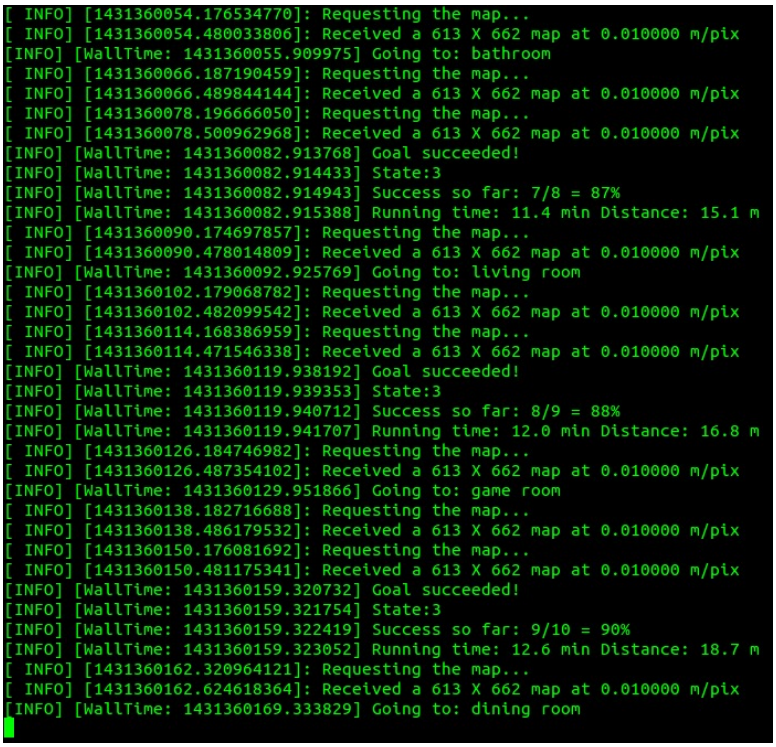


Figure 3.3: Patrolling statistics

Images from the kinect and audio from the microphone of the netbook are recorded in a ros bag, named with the timestamp of the execution. Storing the data in a bag file presents the following benefits:

- Scalable: simple addition/removal of topics.
- ROS abstraction: the agnostic nature of ROS permits to use the bag file as the source of data in place of a system execution. Although the recorded image and audio data is of little interest in this regard, additional topics or even the entire set of available topics may be recorded.
- Managing data formats: no format specification is required as ROS manages the different data types.

However, bag files tend to be large, for instance, a five minute run contained over 150 MB of data. A reduction in the frame rate of the camera to skip 10 frames after each frame significantly reduced the size of the file to less than 50 MB for a run of an equivalent duration. The resolution was not decreased due to the bar code reading and face recognition processes.

An alternative approach could be a node subscribing to the desired topics, and saving the data in selected formats according to their data type. While this enables control over the management of the data, the benefits of bag files were found to outweigh the size cost.

3.1.1 Clearing cost maps

Laser scan data modifies the obstacle cost maps by either marking obstacles or clearing free space through ray tracing (provided the marking and clearing parameters are set). A marked obstacle in the cost map, which cannot be ray traced (for instance, due to the range or position of the sensor), will remain in the cost map even if the space is free. While this may be considered as a safety feature, it presented a hindrance to patrol. Such markings appear as yellow pixels in the cost map, as shown in Figure 3.4, and often led to failures in navigation when the obstacle marking prevented a path to be generated.

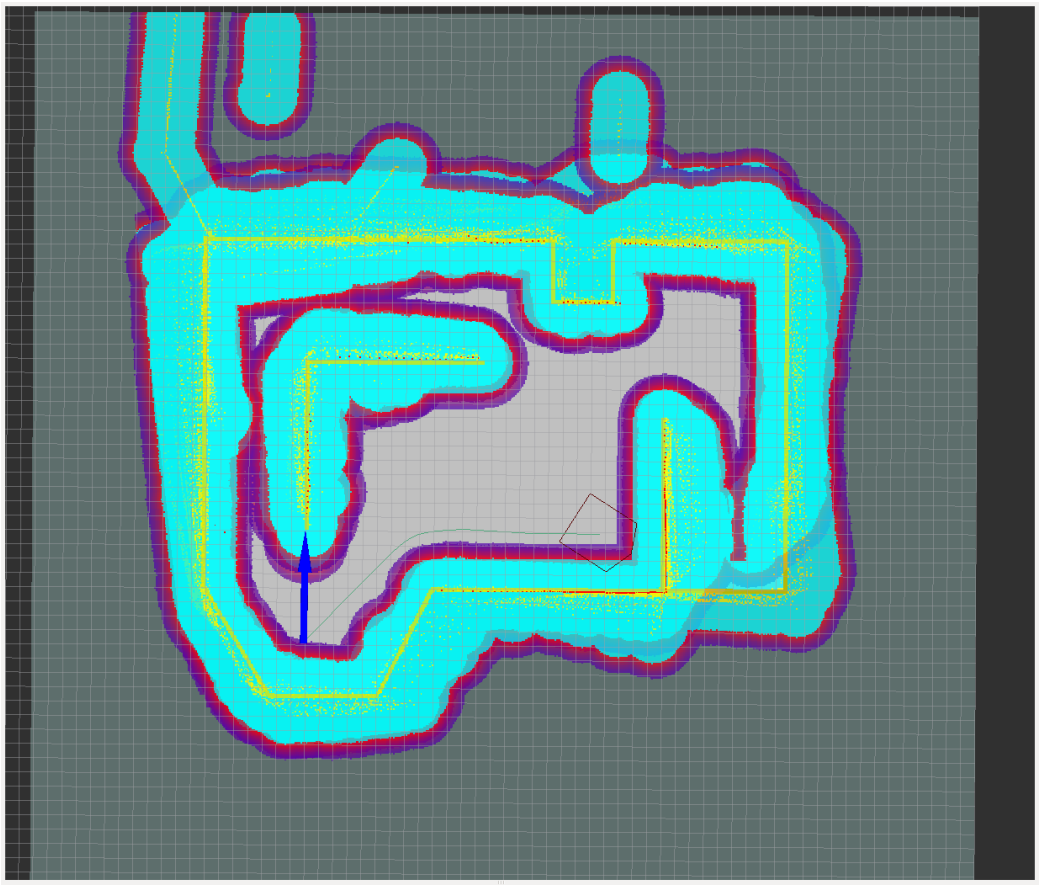


Figure 3.4: Accumulation of obstacle markings in the cost maps

While dynamic objects in the arena could cause this behavior, the results in the static environment were due to the rotational drift. As shown in Figure 3.5, the imperfect rotational odometry causes the misalignment of the laser scans, potentially leading to an accumulation of obstacle markings in the cost map.

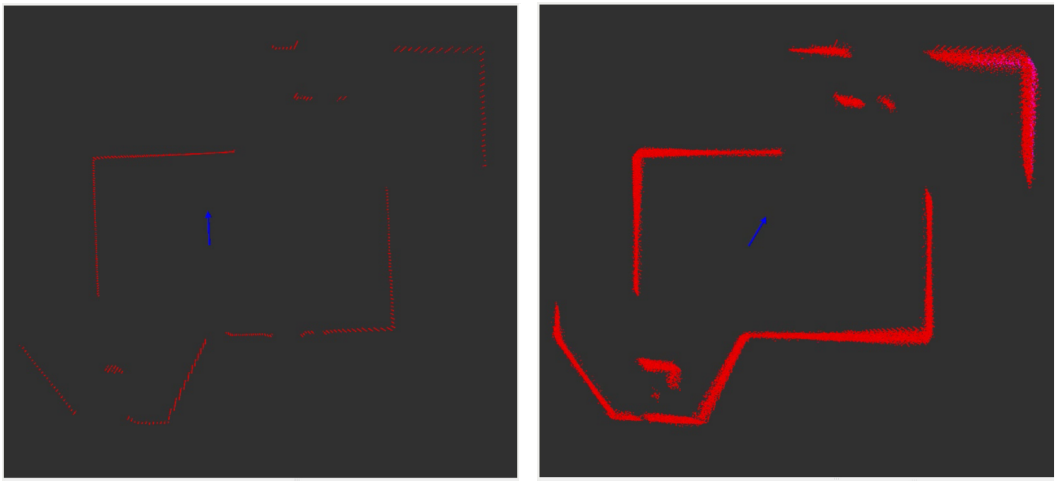


Figure 3.5: Misalignment of the laser scans

As mentioned in section 2.5, the observation persistence parameter was set to 0.0 to maintain only the current laser scan data in the cost map. However, the lack of ray-tracing could not be accounted for.

Two techniques were combined as a solution:

1. The local cost map was reduced to a size where ray tracing was generally possible. Given the configured slow motion of the TurtleBot, and the frequency of the laser scan and cost map updates, the motion remained safe in general. However, the global cost map maintained the markings, requiring the fix explained below.
2. The cost maps were routinely cleared using a service of the `move_base` package. This service is called in the state machine implementation (section 3.7).

This is clearly a radical solution, with potential for erroneous navigation results, whereby the navigation plan may be generated using the cleared cost map. This solution is maintained as a compromise on safety to ensure patrol.

3.2 Global Localization

Global localization, or the kidnapped robot problem is classical in robotics. In order to achieve autonomous navigation, the TurtleBot must be capable of localizing autonomously. The theory of Probabilistic Robotics suggests the particle filter as a candidate solution, as it maintains multiple hypotheses in the search space. The `amcl` node utilizes particle filtering, and provides a global localization service to uniformly sample the robot pose in the map, signifying complete ignorance of the robot pose in the arena.

Our approach is based on the following experiment: the TurtleBot was placed in an arbitrary position in the arena, and the global localization service was called. Motion commands were then given through teleoperation. The results of this experiment are

shown in Figure 3.6. Evidently, localization was achieved through motion in a short period.

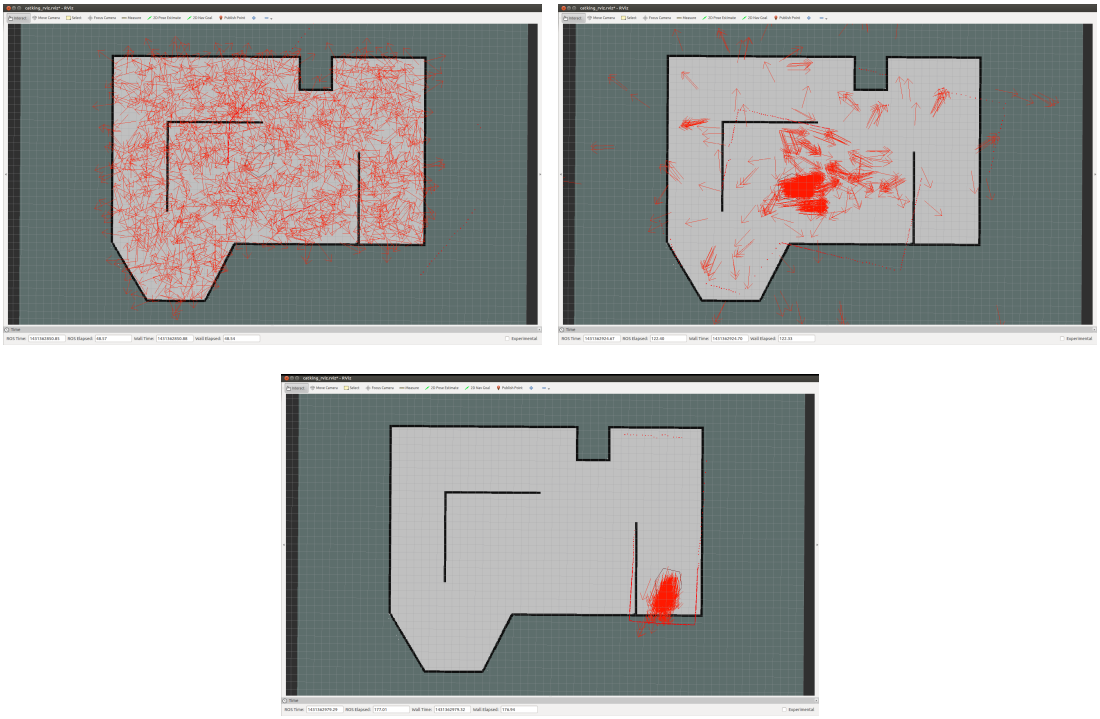


Figure 3.6: Global localization experiment results

The experiment indicated the feasibility of global localization, and autonomous motion being the requirement for achieving it autonomously. The patrol node already provided goal poses for `move_base` to follow, which could be utilized, as the destination and path were largely insignificant. However, with the robot pose unknown, a path could not be followed to the goal pose. Consequently, the recovery behaviors of `move_base` served as the means for initiating motion to the goal pose. As shown in Figure 3.7, the recovery behaviors initiate rotation upon a path planning failure, which given the field of view and range of the sensor, as well as distinguishable features in the map, are generally sufficient to initiate localization.

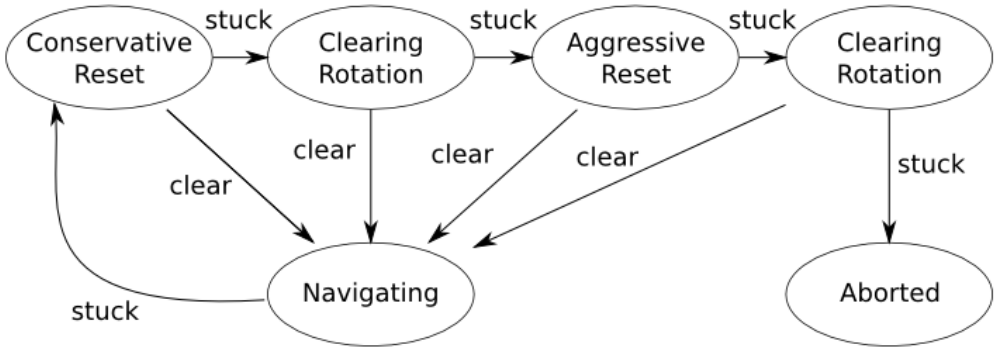


Figure 3.7: Recovery behaviours of `move_base`

Localization due to the recovery behaviors allows a plan to the goal pose to be generated and followed, resulting in convergence in localization. As anticipated, however, laser scan data from similar regions led to potential for incorrect localization, such as in Figure 3.8.

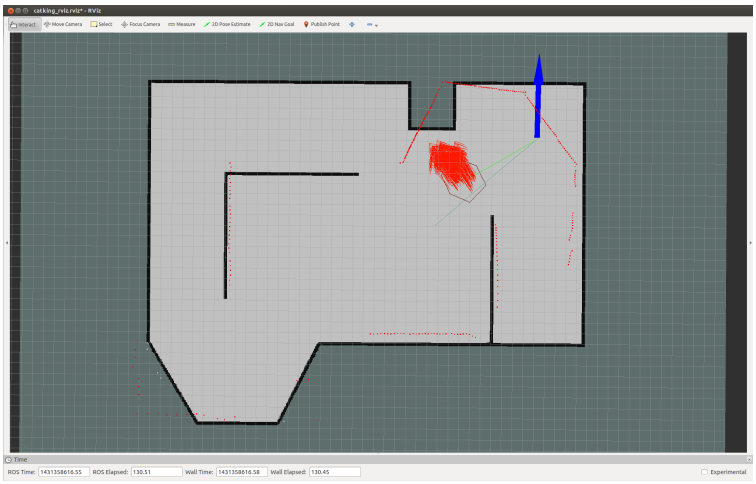


Figure 3.8: Erroneous localization

Although a fairly rare occurrence in our trials, once localized, potential inaccuracies are not accounted for disturbing navigation The theory covers this problem, and suggests adding random hypotheses periodically to the search space as a solution. We tried tuning the `amcl` node in this way, but could not spend sufficient time to succeed. Instead, the following two measured were combined to determine inaccuracy in localization:.

1. Localization covariance: the `amcl` node provides a $6x6$ covariance matrix of the robot pose (location and orientation). Perfect localization implies convergence of all pose vectors, whereby the elements of the covariance matrix tend to 0. Differences between the laser scan data and the cost maps increases the covaraince values. The sum of these elements is used as a naive measure of incorrect localization with emprically determined thresholds.
2. Comparison of local and global cost maps: in a static environment, the local and global cost maps align well in case of accurate localization. A pixel-based difference between the narrow local cost map and the corresponding region of the global cost map constitutes the second measure.

The above two measures are combined in a weighted sum, using empirically determined weights in determining inaccuracy in localization. Clearly, this approach is susceptible to external obstacles and dynamic objects in the arena, whereby the global localization process may be repeated unnecessarily.

3.3 Virtual Joystick

The virtual joystick is a means to assume manual control of the TurtleBot. The `visp_auto_tracker` package [6] wraps an automated pattern-based tracker from the `visp` library,

and provides the 3D pose of and identifiable pattern. A 3D model of the object to be tracked must be provided, which is pre-configured for QR bar codes. Once the pattern is detected, tracking is carried out by the hybrid model-based tracker from `visp` using moving-edges and keypoint features.

The pose vector is published as a `geometry_msgs::PoseStamped` message on the `/object_position` topic. This message is converted to velocity commands through a pre-defined coefficient, and published on the `/cmd_vel_mux/input/teleop` topic to move the TurtleBot.

A static camera is ideal for this feature, and we used an out-dated iBot firewire camera for testing. Figure 3.9 shows the 3D pose obtained from the `visp_auto_tracker` node used to manually steer the TurtleBot.

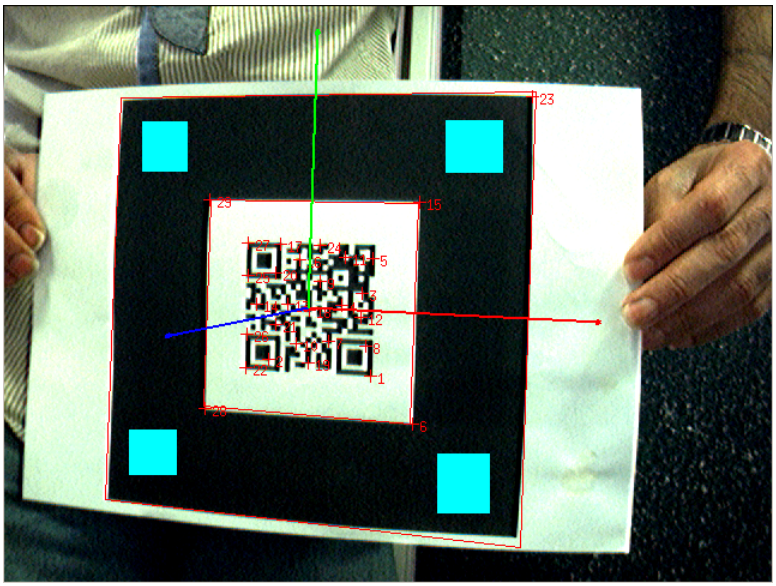


Figure 3.9: 3D pose output from `visp_auto_tracker`

3.4 Face Detection and Recognition

The following criteria were used in selecting a package for face detection and recognition:

- Real-time performance.
- Online training.
- Face detection output for each frame (to determine the presence or absence of a face for each frame).

The ProcRob `face_recognition` package [7] was selected as it provides online training and real-time performance. It is based on the well established Eigenfaces (PCA) face recognition method, and consists of two nodes:

- **FServer**: provides a simple `actionlib` server interface for performing different face recognition functionalities in a video stream.

- **FClient**: implements an `actionlib` client for `face_recognition` simple `actionlib` server (*i.e.* 'FServer').

The FServer publishes a `actionlib::SimpleActionServer<face_recognition::FaceRecognitionAction>` message if a face is recognized. The FClient is provided for demonstration and testing, and simply displays the output if a face is successfully recognized.

This package did not meet the third requirement of providing face detection feedback, which is useful to distinguish the case of the absence of a face from the absence of a recognizable face. Thus, the first modification to the source code of this package was add a publisher to `Fserver` to publish a message for face detection for each frame. The package already uses the face detector of OpenCV, which was used to publish on the `/faceDetected` topic as follows:

- 0: no face detected.
- 1: face detected, but not recognized.
- 2: face recognized.

The second modification consisted in using the `actionlib` message from `face_recognition` message to extract the name of the recognized face. This part required creating an instance of `actionlib::SimpleActionClient<face_recognition::FaceRecognitionAction>` to send a goal message to FServer, and in the event of receiving a message from FServer (implying face recognized), extracting the name from `feedbackCb` part of `actionlib`. However, implementing this in the driver node (section 3.7, it was found to be process consuming, and the waiting for the `Fserver` output could interfere with the working of the driver node. Another thread could have been created in the driver node to avoid such an interference, but we opted to implement it in the `FClient` node and add another publisher to that node. This way, if the FClient receives feedback from FServer, it would extract the name of the face from `feedbackCb` part of `actionlib` message, and consequently publish it as a separate message on the `/faceName` topic.

Figure 3.10 shows the successful results in face recognition, and further tests must be conducted to determine its accuracy.

Figure 3.10: Face recognition output

3.5 Barcode Reading

User input to the robot is essential in human-robot interaction, and the lack of success with speech recognition prompted the use of bar codes with encoded messages. We used ZBar, which is an open source software suite for reading bar codes from various sources, such as video streams, image files and raw intensity sensors. It supports many popular symbologies including EAN-13/UPC-A, UPC-E, EAN-8, Code 128, Code 39, Interleaved 2 of 5 and QR Code. `zbar_ros` package [8] is a lightweight wrapper around the Zbar barcode processing library. It provides a node and a nodelet with equivalent interface. The image topic is a lazy subscription that is active only when the barcode topic has a client.

The `barcode_reader_node` listens to the image data from the Kinect RGB image on the incoming topic and provides the contents of a detected barcodes. THe `zbar_ros` package originally publishes the barcode information as a standard `tf::tfMessage` on `/tf` topic, but as the TurtleBot already has a `/tf` topic for publishing its transfer function, there is potential for interference. In fact, since the driver node (section 3.7) is designed to speak a barcode message not identified as a command, the TurtleBot was noticed to read the `tf` messages due to this interference, and repeat words such as "odom". Thus,

the `zbar` package source code was edited to publish the barcode messages in separated topic named `/barcode` to avoid any interference.

3.6 KINECT CONTROL

The Kinect was used in surveillance, face detection and recognition, and bar code reading, but its low placement was not ideally suited to these purposes. As a result, we repositioned it to the top shelf, as shown in Figure 3.11. However, we desired tilting as the faces to be detected would generally be out of the field of view with the configuration for surveillance and bar code reading. The solution was to use the Kinect motor for tilting when needed.

Figure 3.11: Kinect installation on Robot

The `kinect_aux` package [9] was used, which provides access to additional features of the sensor, such as the accelerometer and LEDs. In addition, we used it in parallel with the `openni_camera` driver. This package was written for and older Kinect model (1414) and could not control the newer Xbox 360 Kinect model (1473) due to the change Microsoft had made for driver protocol. In Kinect 1414 the motor is an individual device and is recognised separately from other components (camera, audio device, etc). But in Kinect 1473 and newer models the motor is a subdevice of the audio control module and yet there is no driver for Linux. The first solution we found for was to downgrade the Kinect firmware to 1414 firmware. It was a tedious activity, but it worked! But then, we faced another problem: whenever the Kinect turned off and on again, it automatically would go back to the original firmware and, again, the motor would not work. The next solution was writing a script to automatically downgrade the firmware whenever the Kinect is attached to the system. That would be enough, but fortunately we found an older Kinect (1414) in the lab and used this one.

The Driver node controls the Kinect motor, tilting up for face detection and recognition, and down for barcode reading and surveillance.

3.7 DRIVER NODE

A state machine was required to integrate the different functionalities of CatKing. Rather than using a standard ROS package, such as `smach`, we decided to use simple conditional statements in a node implementation.

This node subscribes to 8 different topics, and publishes 5 different topics. In some cases, the system command is used to publish the required message in the same terminal that runs the node, as publishing a message on a topic once latches the system for 3 seconds. This way, the message is published completely leading to its corresponding action is done and during this publish time the programs state and decision has not been changed since it was paused by system.

A brief explanation of the working of this node is given below:

1. Global localization: initiates global localization when an inaccuracy is determined, as explained in section3.2.
2. Clearing cost maps: the global and local cost maps are cleared at intervals of 12 seconds using a timer to resolve the accumulation of obstacle markings, as explained in section 3.1.1.
3. Patrol: the patrol node is halted to execute another task, and resumed upon entering the patrol state. This is accomplished by publishing a boolean message, `std_msgs::Bool` every second on the `/startPatrol` topic which activates patrol.
4. Virtual Joystick: the conversion from `geometry_ms gs::PoseStamped` to `geometry_msgs:: Twist` velocity command with the corresponding coefficients, as mentioned in section 3.3 is carried out only in the virtual joystick state when patrol is paused to avoid conflicts in motion.
5. Speaking: yhe `soundplay` node and the `say.py` script are used to read strings from the `/sayThis` topic as audio feedback from the TurtleBot.
6. Controlling the Kinect sensor: the driver node is responsible for tilting the Kinect according to the current state. As mentioned in section 3.6, the `kinect_aux` node controls the tilt angle of the Kinect sensor. This node subscribes to `/tilt_angle` topic which receives a `float64` message as the desired tilt angle in the range $[-31.0 \quad 31.0]°$. The desired tilt angle is published on the `/tilt_angle` topic when required. This node also controls the Kinect LED state by publishing the state number (an integer in the range $[0 \quad 7]$) on `/led_option` topic. The LED states are used to signal different states or tasks in action.
7. Face detection: the aforementioned `/faceName` and `/faceDetected` topics published by the face recognition node (section 3.4) to determine if the face is detected in the face recognition state.
8. State machine: as mentioned in section 3.5, bar codes are used in commanding inputs and altering the TurtleBot state. The bar code messages are compared with the following predefined commands:

Figure 3.12: Predefined QR codes for state machine

- `<CMD_MoveUp>`: pauses the patrol after reaching the currently pursued goal, tilts the Kinect upward waits for the next user command.
- `<CMD_MoveDown>`: tilts the Kinect down to its default position, resumes patrol.
- `<CMD_VisServo>`: enters Virtual Joystick state, overriding robot motion.
- `<CMD_Recognize>`: commands to recognize a face.

  If the information in the QR tag is one of these predefined commands, the state machine will change the state as in shown in Figure 3.13 otherwise the information is simply read using the `sound_play` node.

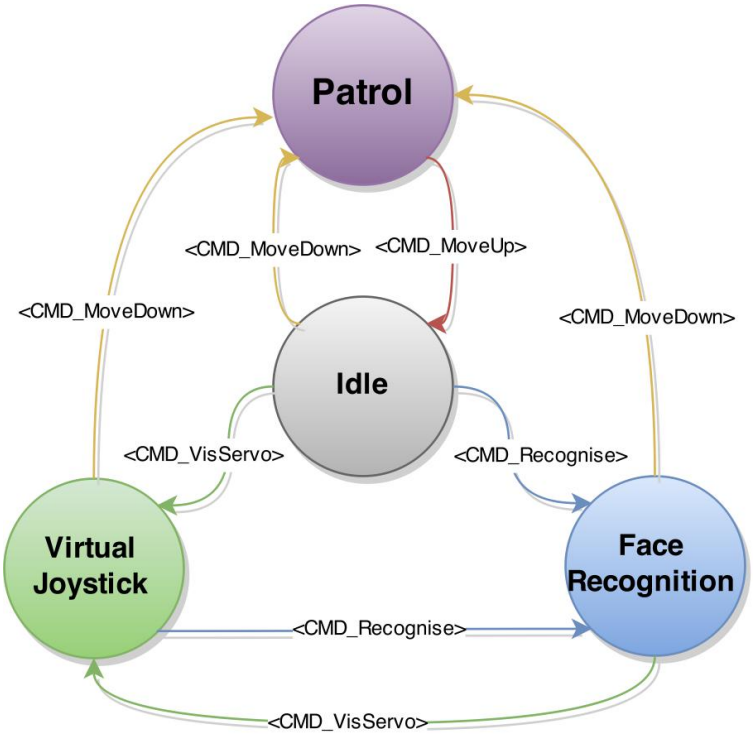


Figure 3.13: Driver node's State Machine

9. Terminate Signal handler: at last in the Driver node we handle the program closing signal `SIGTERM` to know when the program is closing and initiate the shutdown protocol, such as saying "goodbye".

CHAPTER 4

DISCUSSION

4.1 CHALLENGES FACED

Some notable challenges encountered in the project are described along with our solutions:

1. Mapping: although the SLAM package, `gmapping`, is well established and documented, and it worked well in previous experience, it performed poorly using the RP-LIDAR sensor in the Groovy distribution. There were frequent jumps in the pose estimate, leading to inaccurate mapping. Tuning the package parameters presented no apparent improvements, though tests with the kinect data as laser scan data performed normally. `hector_mapping` [10] produced a seemingly correct map, but it could not be saved. Finally, upgrading to the Hydro Medusa distribution led to the presented results using the `gmapping` package.

2. Incorrect final orientation in patrol: the position of the TurtleBot corresponded to the goal location, but the orientation was arbitrary. The patrol node was found to utilize a secondary configuration file, which overwrote the `move_base` parameters for the goal pose tolerance.

3. `move_base` node crash: in the state machine execution, the `move_base` node died often, halting the patrol. The reason behind this could not be determined, and the log file of the crash could not be found. The fix used was to respawn the node upon termination through its corresponding flag. A goal location could be skipped in the patrol, as a result, however.

4. Inflation radius: was misunderstood to be the distance to inflate the obstacle by, instead of the distance beyond the footprint, as shown in Figure 4.1. An inflation radius lower than the robot radius erroneously inflated the obstacles with this value rather than the sum of the robot and inflation radius. As result, the visualization in Rviz was constant, despite varying these parameters, and the robot motion was unsafe with frequent collisions.

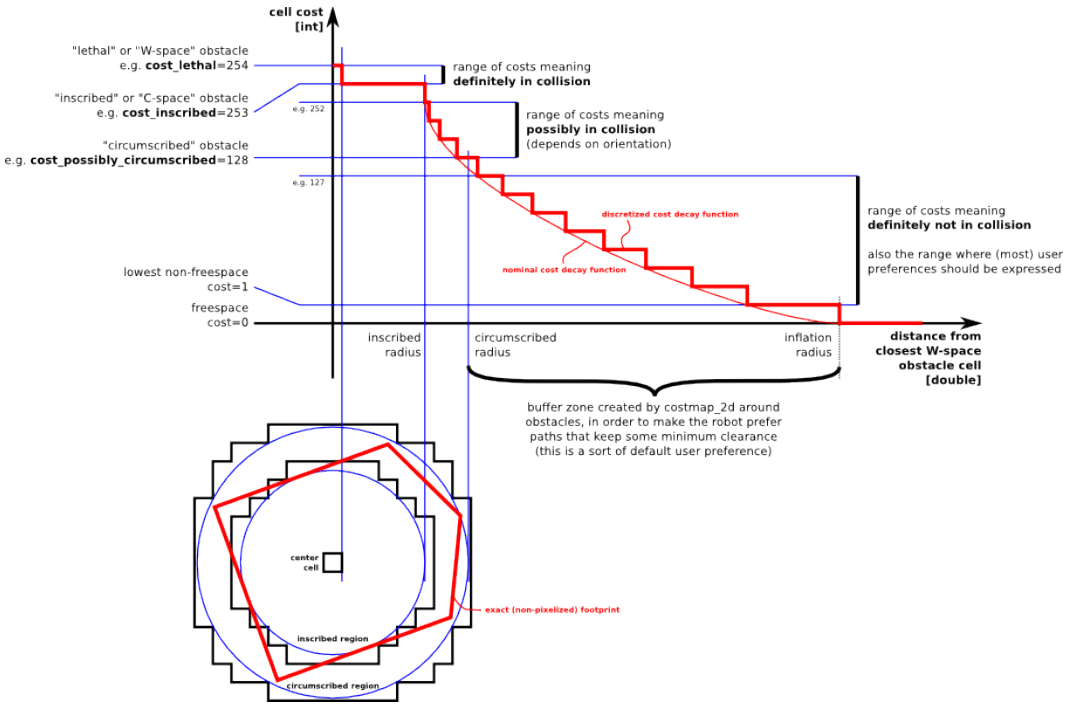


Figure 4.1: Inflation radius

5. Netbook battery heating: heating often caused the netbook to abruptly power off, despite having charge. The netbook was moved to the middle shelf, as shown in Figure 4.2, where it could be kept open, without interfering with the laser scanner.

Figure 4.2: Netbook placement to avoid heating

6. Documentation: since existing packages were used exclusively, we relied on their documentation for our understanding and implementation. Whie most of the packages we used are well established, documented and maintained, some aspects were ambiguous, as shown in Figure 4.3. Whereas, in cases such as the navigation stack, the documentation is quite dense, and required a thorough understanding for use, as illustrated in the inflation radius example above.

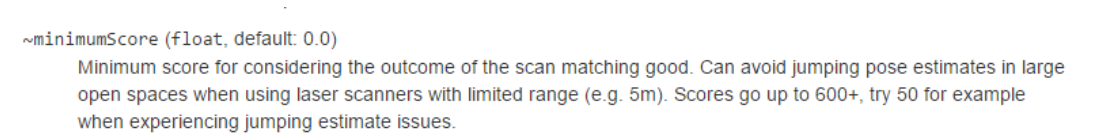


Figure 4.3: Ambiguous documentation

7. Second workstation: as the CatKing project grown bigger and bigger, handling everything in one workstation and by one person became harder and harder. We had several nodes that we had to track the behaviour of, Rviz for monitoring robot's motion, 2 output cameras, etc. Therefore, we have decided to add another work station which is one of our laptops. This would divide the process between two computers and also the network load. Handling two camera (one RGB-D and one RGB) data, laser scanner data, and so many other data was creating some delay. As the router installed on Robot is a dual band router, we used the second band for the second work station and the network load was controlled by this one.

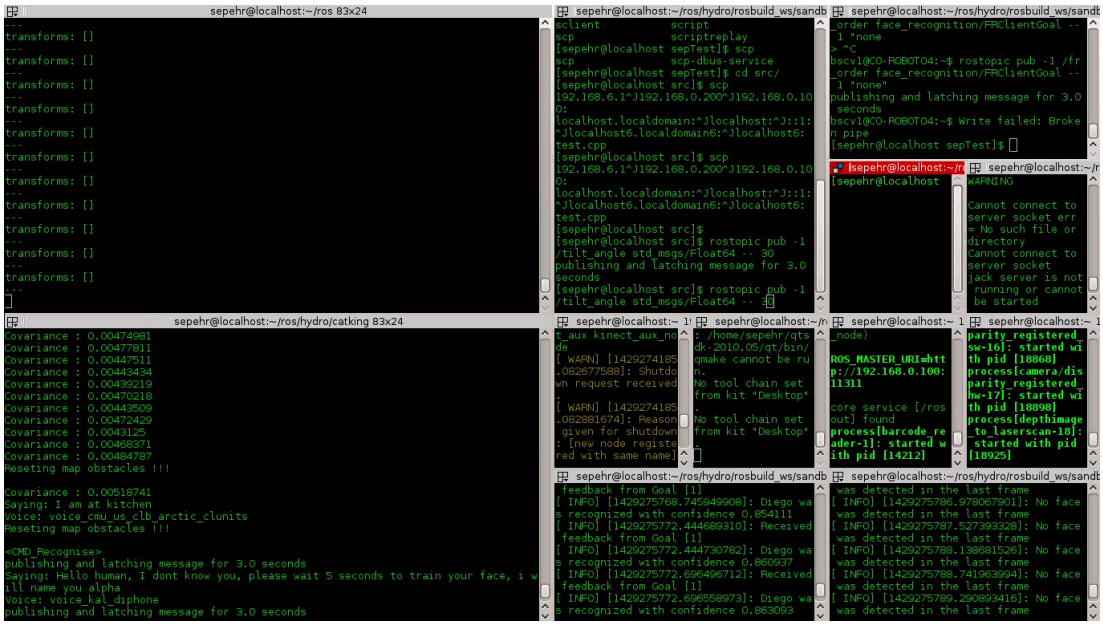


Figure 4.4: Second workstation

8. iBot firewire camera: for implementing the virtual joystick we needed a camera for our workstation, since it did not have any webcam. We just found an old iBot Firewire iEEE 1394 camera in the lab. As this webcam was designed for Mac computers, it did not have a proper driver for Linux. So we had the challenge to

make it work with Linux. This was done before adding the second workstation and now we can also use our second workstation's webcam for this purpose.

9. Using one single launch file for the project: It seems to be easier to create a launch file to run all the nodes together. However, we faced some issues with this kind of implementation. The most important issues were nodes initialization time and unexpected error that might happen during run time. The first issue could be explained as some of the nodes had to be run when another node which it depends on is already running and initialized completely otherwise if some particular nodes run before complete initialization of its dependency it would throw an exception and close or create a malfunction in system. For instance if the driver node runs before the complete initialization of the navigation and AMCL packages, the auto localization service which will be called at the start of driver node will be missed and the robot will not start the localization. The second issue is due to the fact that when one node stops working for any reason, if this node is being run in separate terminal we can easily see the problem and fix it or re-launch the node without killing the whole system. It also should be mentioned that we prefered to monitor the state of every single node. Thus we decided not to use a global launch file and run everything sequentially in correct order.

4.2 Future Work

- Global localization: the formulation of the particle filter and AMCL requires further investigation to periodically add random hypotheses within the map in order to account for incorrect localization.
- The abrupt crashing of the `move_base` node during the state machine run has to be fixed.
- Speech recognition is key in robot-human interaction, and existing packages ought to be tested for reliable and reproducible performance.

Chapter 5

Conclusion

CatKing was modeled after a home robot application, with task capabilities influenced from the RoboCup@Home league. The implementation is a state machine integrating tasks performed through several existing ROS packages, whereby our contribution is mostly in their modification and configuration. The aspiration is to build on this design to be capable of participation in such a league. Scalability is critical for this purpose, and a systematic state machine implementation would be required to incorporate additional tasks of interest.

This project served as a solid introduction to ROS. We experienced both the Groovy and Hydro Medusa distributions, and worked with `rosbuild` as well as `catkin` build systems. The learning curve was slow, and we found a thorough understanding of the documentation to be valuable at times, necessary for implementation. We are grateful for the open access to the laboratory, and for the support available throughout the semester.

REFERENCES

[1] *The RoboCup@Home league* http://www.robocupathome.org/

[2] *RPLIDAR planar laser range finder* http://rplidar.robopeak.com

[3] Goebel, R. Patrick; *ROS By Example for ROS Hydro Volume 1*, January 2014.

[4] *Using Rviz with the Navigation Stack* http://wiki.ros.org/navigation/Tutorials/

[5] *Move base recovery behaviors* http://wiki.ros.org/move_base

[6] *Visp auto-tracker* http://wiki.ros.org/visp_auto_tracker

[7] *Face recognition* http://wiki.ros.org/face_recognition

[8] *Zbar* http://wiki.ros.org/zbar_ros

[9] *Kinect aux* http://wiki.ros.org/kinect_aux

[10] *Hector mapping* http://wiki.ros.org/hector_mapping

APPENDIX A

PROJECT MANAGEMENT

A.1 TOOLS

A variety of management tools were tried in this project:

1. Trello: task managing and progress monitoring.
2. Google drive: file sharing for this report.
3. Facebook messenger: communication.
4. Slack: sparingly used for communication, file sharing, monitoring progress (through link with Trello).
5. Overleaf: collaborating on LaTeX for this report.
6. Github: maintaining the project as open-source.

A.2 DEVELOPMENT

The first few weeks were dedicated to tutorials, followed by independent project development. Soon upon commencing the project, we realized we could not schedule the tasks in a timeline, as the time required for each task was unpredictable. For instance, SLAM, which is generally straightforward, took weeks to achieve. Similarly, installing the `visp` auto tracker [6] package required a significant amount of time, whereas the seemingly laborious task of a node for the virtual joystick was completed very fast. Thus, we resorted to a task-based approach of following tasks successively and working additional hours. Working in parallel aided in completing the selected number of tasks.

Navigation and localization were continued throughout the project, whereas other tasks were incrementally added as explained above.

APPENDIX B

CATKING ORIGINS

The project name, `CatKing`, is the result of a minor yet amusing typographical error, during the creation of a distinct workspace for the catkin build system.

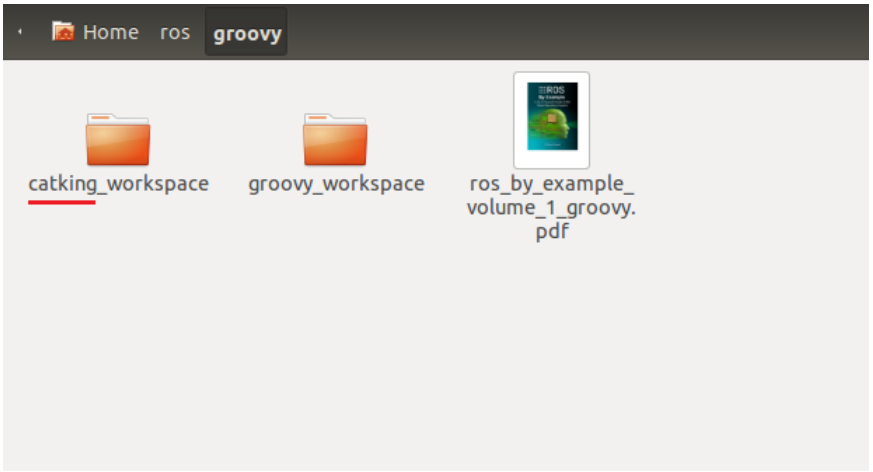


Figure B.1: CatKing origins

APPENDIX C

CODE

THE DRIVER NODE

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_msgs/Float64.h"
#include "std_msgs/UInt16.h"
#include "std_msgs/Bool.h"
#include "geometry_msgs/Twist.h"
#include "geometry_msgs/PoseStamped.h"
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <tf/tfMessage.h>
#include <geometry_msgs/TransformStamped.h>
#include <iostream>
#include <string>
#include <signal.h>

using namespace std;

ros::Publisher pubBarcode;
ros::Publisher pubKinectTilt;
ros::Publisher pubKinectLED;
ros::Publisher pubStartPatrol;
ros::Publisher pubVisServ;
std_msgs::Bool patAllowance;
string ss;
string name;
string faceState;
bool inited = false;

void faceDataExtractor(const std_msgs::String::ConstPtr&
   msg)
{
```

```
    name = msg->data.c_str();
    cerr << msg->data.c_str() << endl;
}

void faceStateExtractor(const std_msgs::String::ConstPtr&
    msg)
{
    faceState = msg->data.c_str();
}

void speaking(const std_msgs::String::ConstPtr& msg)
{
    string toSay = "rosrun sound_play say.py \"I am at "
       + msg->data +"\" voice_cmu_us_clb_arctic_clunits";
    system(toSay.c_str());
}

void infoExtractor(const tf::tfMessage::ConstPtr& msg)
{
    if(!msg->transforms.empty())
    {
        cout << msg->transforms.front().child_frame_id <<
            endl;
        ss = msg->transforms.front().child_frame_id.c_str
           ();
        std_msgs::String str;
        if(ss == "<CMD_MoveUp>")
        {
            patAllowance.data = false;
            pubStartPatrol.publish(patAllowance);
            std_msgs::Float64 angle;
            angle.data = 30;
            pubKinectTilt.publish(angle);
            ros::Duration(3).sleep();
            name = "";
            std_msgs::UInt16 LED;
            LED.data = 1;
            pubKinectLED.publish(LED);
        }
        else if(ss == "<CMD_MoveDown>")
        {
            patAllowance.data = true;
            pubStartPatrol.publish(patAllowance);
            std_msgs::Float64 angle;
            angle.data = -15;
            pubKinectTilt.publish(angle);
```

```
            ros::Duration(3).sleep();
            name = "";
            std_msgs::UInt16 LED;
            LED.data = 2;
            pubKinectLED.publish(LED);
        }
        else if(ss == "<CMD_VisServo>")
        {
            patAllowance.data = false;
            std_msgs::UInt16 LED;
            LED.data = 3;
            pubKinectLED.publish(LED);
        }
        else if(ss == "<CMD_Recognise>")
        {
            std_msgs::UInt16 LED;
            LED.data = 6;
            pubKinectLED.publish(LED);
            cerr << "+++++name : " << name << endl;
            if( faceState == "2")
            {
                string toSay = "rosrun sound_play say.py
                   \" Hi " + name + " What can I do for
                   you\" voice_cmu_us_clb_arctic_clunits"
                   ;
                system(toSay.c_str());
                name = "";
            }
            else if( faceState == "1")
            {
                string toSay = "rosrun sound_play say.py
                   \"Hello human, I dont know you\"
                   voice_cmu_us_clb_arctic_clunits";
                system(toSay.c_str());
            }
            else
            {
                string toSay = "rosrun sound_play say.py
                   \"There is no one here\"
                   voice_cmu_us_clb_arctic_clunits";
                system(toSay.c_str());
            }
        }
        else
        {
            std_msgs::UInt16 LED;
```

```
            LED.data = 4;
            pubKinectLED.publish(LED);
            string toSay = "rosrun sound_play say.py \""
               + ss + "\" voice_cmu_us_clb_arctic_clunits
               ";
            system(toSay.c_str());
            name = "";
        }
        str.data = ss;
        pubBarcode.publish(str);
    }
}

void mapReset(const ros::TimerEvent&)
{
    cerr << "Reseting map obstacles !!!" << endl;
    system("rosservice call /move_base/clear_costmaps");
}

void patSend(const ros::TimerEvent&)
{
    pubStartPatrol.publish(patAllowance);
}

void visualServoing(const geometry_msgs::PoseStamped::
   ConstPtr& msg)
{
    static double xx = 0,yy = 0;
    if(yy != msg.get()->pose.position.y
            || xx != msg.get()->pose.position.x)
    {
        geometry_msgs::Twist cmd;
        cmd.linear.x = msg.get()->pose.position.y * -6;
        cmd.angular.z = msg.get()->pose.position.x * 6;
        pubVisServ.publish(cmd);
        yy = msg.get()->pose.position.y;
        xx = msg.get()->pose.position.x;
    }
}

void localization(geometry_msgs::
   PoseWithCovarianceStampedConstPtr msg)
{
    double conv = 0;
    for(int i = 0; i < msg.get()->pose.covariance.size();
        i++)
```

```
        conv += fabs(msg.get()->pose.covariance.at(i));
    cerr << "Covariance : " << conv << endl;
    if(!inited)
    {
        if(conv < 0.01)
        {
            inited = true;
            string toSay = "rosrun sound_play say.py \"
               Localization Finished, I am calibrated now
               \" voice_cmu_us_clb_arctic_clunits";
            system(toSay.c_str());
        }
    }
    else if(conv > 0.03)
    {
        inited = false;
        system("rosservice call /global_localization");
        string toSay = "rosrun sound_play say.py \"I feel
            something is wrong with my localization,
           going back to calibrating phase\"
           voice_cmu_us_clb_arctic_clunits";
        system(toSay.c_str());
    }
}


void mySigintHandler(int sig)
{
    patAllowance.data = false;
    cerr << "Shutting Down" << endl;
    string toSay = "rosrun sound_play say.py \"Shutting
       Down. Goodbye everyone.\"
       voice_cmu_us_clb_arctic_clunits";
    system(toSay.c_str());
    ros::shutdown();
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "Driver_module");
    ros::NodeHandle n;
    signal(SIGINT, mySigintHandler);
    std::cerr << " inited " << std::endl;
    ros::Subscriber subFaceName = n.subscribe("/faceName"
       , 1, faceDataExtractor);
```

```
    ros::Subscriber subFaceState = n.subscribe("/
       faceDetected", 1, faceStateExtractor);
    ros::Subscriber subVIsServ = n.subscribe("/
       object_position", 1, visualServoing);
    ros::Subscriber subSpeaker = n.subscribe("sayThis",
       1, speaking);
    ros::Subscriber subLocalizer = n.subscribe("amcl_pose
       ", 1, localization);
    ros::Subscriber subBarcode = n.subscribe("/barcode",
       5, infoExtractor);
    ros::Timer mapResetTimer = n.createTimer(ros::
       Duration(12), mapReset);
    ros::Timer patrolTimer = n.createTimer(ros::Duration
       (1), patSend);
    system("rostopic pub -1 /fr_order face_recognition/
       FRClientGoal -- 1 \"none\"");
    system("rosservice call /global_localization");
    pubBarcode = n.advertise<std_msgs::String>("
       barcodeInfo", 100);
    pubKinectTilt = n.advertise<std_msgs::Float64>("
       tilt_angle",3);
    pubKinectLED = n.advertise<std_msgs::UInt16>("
       led_option",3);
    pubStartPatrol = n.advertise<std_msgs::Bool>("
       startPatrol",9);
    pubVisServ = n.advertise<geometry_msgs::Twist>("
       cmd_vel_mux/input/teleop",6);
    patAllowance.data = true;
    std::cerr << " blah " << std::endl;
    //    tf::TransformListener tf(ros::Duration(10));
    //    costmap_2d::Costmap2DROS costmap("/move_base/
       local_costmap/costmap", tf);
    ros::spin();

    return 0;
}
```

PATROL SCRIPT

```
#!/usr/bin/env python

""" nav_test.py - Version 1.1 2013-12-20

    Command a robot to move autonomously among a number
       of goal locations defined in the map frame.
    On each round, select a new random sequence of
       locations, then attempt to move to each location
    in succession.  Keep track of success rate, time
       elapsed, and total distance traveled.

    Created for the Pi Robot Project: http://www.pirobot.
       org
    Copyright (c) 2012 Patrick Goebel.  All rights
       reserved.

    This program is free software; you can redistribute
       it and/or modify
    it under the terms of the GNU General Public License
       as published by
    the Free Software Foundation; either version 2 of the
        License, or
    (at your option) any later version.5

    This program is distributed in the hope that it will
       be useful,
    but WITHOUT ANY WARRANTY; without even the implied
       warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
        See the
    GNU General Public License for more details at:

    http://www.gnu.org/licenses/gpl.html

"""

import rospy
import actionlib
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose,
   PoseWithCovarianceStamped, Point, Quaternion, Twist
from std_msgs.msg import Bool
from move_base_msgs.msg import MoveBaseAction,
   MoveBaseGoal
```

```
from random import sample
from math import pow, sqrt
from std_msgs.msg import String

class NavTest():
    def __init__(self):
        rospy.init_node('nav_test', anonymous=True)

        rospy.on_shutdown(self.shutdown)

        # How long in seconds should the robot pause at
           each location?
        self.rest_time = rospy.get_param("~rest_time",
           10)

        # Are we running in the fake simulator?
        self.fake_test = rospy.get_param("~fake_test",
           False)

        # Goal state return values
        goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED',
                       'SUCCEEDED', 'ABORTED', 'REJECTED'
                          ,
                       'PREEMPTING', 'RECALLING', '
                          RECALLED',
                       'LOST']

        # Set up the goal locations. Poses are defined in
            the map frame.
        # An easy way to find the pose coordinates is to
           point-and-click
        # Nav Goals in RViz when running in the simulator
           .
        # Pose coordinates are then displayed in the
           terminal
        # that was used to launch RViz.
        locations = dict()

        locations['corridor'] = Pose(Point(-1.689, 0.006,
            0.000), Quaternion(0.000, 0.000, 0.709,
           0.705))
        locations['kitchen'] = Pose(Point(-0.872, -0.996,
            0.000), Quaternion(0.000, 0.000, -0.705,
           0.709))
        locations['bedroom'] = Pose(Point(-2.383, -3.235,
            0.000), Quaternion(0.000, 0.000, 1.000,
```

```
            -0.003))
        locations['living room'] = Pose(Point(-2.509,
            -2.435, 0.000), Quaternion(0.000, 0.000,
            0.509, 0.861))
        locations['game room'] = Pose(Point(-1.762,
            -0.673, 0.00), Quaternion(0.000, 0.000,
            -0.861, 0.509))
        locations['dining room'] = Pose(Point(-3.287,
            -0.288, 0.000), Quaternion(0.000, 0.000,
            -0.006, 1.000))
    locations['bathroom'] = Pose(Point(-0.964, -3.221,
       0.000), Quaternion(0.000, 0.000, -0.011, 1.000))

        # Publisher to manually control the robot (e.g.
            to stop it)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel',
            Twist)

    self.say_this = rospy.Publisher("/sayThis", String)

        # Subscribe to the move_base action server
        self.move_base = actionlib.SimpleActionClient("
            move_base", MoveBaseAction)

        rospy.loginfo("Waiting for move_base action
            server...")

        # Wait 60 seconds for the action server to become
             available
        self.move_base.wait_for_server(rospy.Duration(60)
            )

        rospy.loginfo("Connected to move base server")

        # A variable to hold the initial pose of the
            robot to be set by
        # the user in RViz
        initial_pose = PoseWithCovarianceStamped()

        # Variables to keep track of success rate,
            running time,
        # and distance traveled
        n_locations = len(locations)
        n_goals = 0
        n_successes = 0
        i = n_locations
```

```
        distance_traveled = 0
        start_time = rospy.Time.now()
        running_time = 0
        location = ""
        last_location = ""

    sequence=locations.keys()


        # Get the initial pose from the user
        #rospy.loginfo("*** Click the 2D Pose Estimate
           button in RViz to set the robot's initial pose
           ...")
        #rospy.wait_for_message('initialpose',
           PoseWithCovarianceStamped)
        self.last_location = Pose()
        #rospy.Subscriber('initialpose',
           PoseWithCovarianceStamped, self.
           update_initial_pose)
        #initial_pose.pose.pose.position.x =
           -3.38111829758
        #initial_pose.pose.pose.position.x =
           -0.250909358263
        #initial_pose.pose.pose.position.x = 0.0

        #initial_pose.pose.pose.orientation.x = 0.0
        #initial_pose.pose.pose.orientation.x = 0.0
        #initial_pose.pose.pose.orientation.x =
           0.0119877873589
        #initial_pose.pose.pose.orientation.x =
           0.999928143895

    #initial_pose.pose.covariance = [0.25, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0,
       0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
       0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
       0.0, 0.0, 0.06853891945200942]

    #self.update_initial_pose(self, initial_pose)

    self.PatrolFlag=False

    rospy.Subscriber('startPatrol', Bool, self.
       update_bool)

    print self.PatrolFlag
```

```
# Make sure we have the initial pose
while initial_pose.header.stamp == "":
    rospy.sleep(1)

rospy.loginfo("Starting navigation test")

# Begin the main loop and run through a sequence of
   locations
while not rospy.is_shutdown():
    if self.PatrolFlag==True:
        # If we've gone through the current sequence,
        # start with a new random sequence
        if i == n_locations:
        i = 0
        # Skip over first location if it is the same
           as
        # the last location


        # Get the next location in the current
           sequence
        location = sequence[i]

        # Keep track of the distance traveled.
        # Use updated initial pose if available.
        if initial_pose.header.stamp == "":
        distance = sqrt(pow(locations[location].
           position.x -
                            locations[last_location].
                               position.x, 2) +
                        pow(locations[location].
                           position.y -
                            locations[last_location].
                               position.y, 2))
        else:
        rospy.loginfo("Updating current pose.")
        distance = sqrt(pow(locations[location].
           position.x -
                            initial_pose.pose.pose.
                               position.x, 2) +
                        pow(locations[location].
                           position.y -
```

```
                        initial_pose.pose.pose.
                           position.y, 2))
initial_pose.header.stamp = ""

# Store the last location for distance
   calculations
last_location = location

# Increment the counters
i += 1
n_goals += 1

# Set up the next goal location
self.goal = MoveBaseGoal()
self.goal.target_pose.pose = locations[
   location]
self.goal.target_pose.header.frame_id = 'map'
self.goal.target_pose.header.stamp = rospy.
   Time.now()

# Let the user know where the robot is going
   next
rospy.loginfo("Going to: " + str(location))

# Start the robot toward the next location
self.move_base.send_goal(self.goal)

# Allow 5 minutes to get there
finished_within_time = self.move_base.
   wait_for_result(rospy.Duration(300))

# Check for success or failure
if not finished_within_time:
self.move_base.cancel_goal()
rospy.loginfo("Timed out achieving goal")
else:
state = self.move_base.get_state()
if state == GoalStatus.SUCCEEDED:
    rospy.loginfo("Goal succeeded!")
        self.say_this.publish(str(location))
    n_successes += 1
    distance_traveled += distance
    rospy.loginfo("State:" + str(state))
else:
  rospy.loginfo("Goal failed with error code:
      " + str(goal_states[state]))
```

```
            # How long have we been running?
            running_time = rospy.Time.now() - start_time
            running_time = running_time.secs / 60.0

            # Print a summary success/failure, distance
               traveled and time elapsed
            rospy.loginfo("Success so far: " + str(
               n_successes) + "/" +
                    str(n_goals) + " = " +
                    str(100 * n_successes/n_goals) + "%
                       ")
            rospy.loginfo("Running time: " + str(trunc(
               running_time, 1)) +
                    " min Distance: " + str(trunc(
                       distance_traveled, 1)) + " m")
            rospy.sleep(self.rest_time)

    def update_initial_pose(self, initial_pose):
        self.initial_pose = initial_pose

    def update_bool(self,PatrolFlag):
    self.PatrolFlag=PatrolFlag.data

    def shutdown(self):
        rospy.loginfo("Stopping the robot...")
        self.move_base.cancel_goal()
        rospy.sleep(2)
        self.cmd_vel_pub.publish(Twist())
        rospy.sleep(1)

def trunc(f, n):
    # Truncates/pads a float f to n decimal places
       without rounding
    slen = len('%.*f' % (n, f))
    return float(str(f)[:slen])

if __name__ == '__main__':
    try:
        NavTest()
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo("AMCL navigation test finished.")
```