University of Burgundy

Robotics Project

ASMA

Automatic Shopping Mall Assistant

Author:
Abinash Pant
Emre Ozan Alkan

Supervisor:
Ralph Seulin
Cansen Jiang
Raphael Duverne

A report submitted in fulfilment of the requirement for Robotics Project

January 2015

"At bottom, robotics is about us. It is the discipline of emulating our lives, of wondering how we work."

ROD GRUPEN

UNIVERSITY OF BURGUNDY

Abstract

MSCV

Robotics Project

ASMA

by Abinash Pant
Emre Ozan Alkan

The aim of this project is to understand the inner working of ROS and to gain an in-depth knowledge of robot navigation, control and vision using ROS. For this purpose we have created ASMA (Automatic Shopping Mall Assistance), that would navigate in a known map and perform tasks that would be encountered in a day to day life of a shopping mall robot. This has been achieved by using the groovy distribution [1] of ROS running on a turtlebot [2]. The robot would perform tasks such as voice control, face recognition, visual servoing, tag detection and manual override while moving on a predefined map. All of this is achieved by using a finite state machine implementation to make it modular, easy to visualize and debug.

In this report we will primarily discuss about our implementation details along with a brief overview of the theory behind it. In the first part of this report we present the techniques used to comprehend an unknown environment and the process involved in creating a map. Later we would discuss the use of this prior knowledge of the map to help the robot navigate. We would then examine the overall structure of our setup along with our finite state implementation also giving a birds eye view of our setup to the reader. Details of each of the tasks performed by the robot follow after that. We would finally conclude this report by analyzing the result, discussing the problem faced along with our conclusion.

Contents

List of Figures

Abbreviations

ASMA	**A**utomatic **S**hopping **M**all **A**ssistant
ROS	**R**obot **O**perating **S**ystem
ViSP	**Vi**sual **S**ervoing **P**latform
OSRF	**O**pen **S**ource **R**obotics **F**oundation
SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
PT	**P**an & **T**ilt

Chapter 1

Introduction

In this report you will find how we created ASMA to fulfill the requirements of robotics course in our master program. In order to achieve this results, we've followed the tutorials and materials given by our supervisors. First we went through the tutorials about Linux, ROS and TurtleBot. This helped us to learn and understand basics of ROS and robotics. Then we followed the highly recommended book to understand ROS, "Ros By Example" by R. Patrick Goebel[3]. This book helped us to start experimenting with the ROS and the "TurtleBot 2". Thanks to this book and countless help of our supervisors, we have achieved many small tasks which later became our building blocks of ASMA. So we have spent our first half of the semester experimenting with ROS and the later half, with our newly gained knowledge of ROS, on our project.

Our plan was to design a helper robot for a shopping mall that would assist people, navigate around the shopping mall and perform some tasks based on real world applications. This gave birth to ASAM : an Automatic Shopping Mall Assistant. We then decided on four basic tasks that ASMA would perform. They were chosen to be manual override, voice control, visual servoing and face detection along with the basic task of navigation and task recognition. We choose these feature as they would be required in a shopping mall scenario while keeping in mind the time and resource constrains.

We would now discuss in briefly how these task would be performed in a real world scenario. Manual override would give the control to the shoppers to navigate the robot to carry their stuff to any location or can be used to move the robot during failure of its automatic navigation module. Voice control would help people interact with the

robot, get information and also make it more approachable. Visual servoing would enable people to make the robot follow them or to make the robot navigate to a specific location. Face detection and recognition would be enable the robot to recall the persons it has met before to personalize the service or even provide special service for a select few. Patrolling would help the robot navigate to different location of the shopping mall for surveillance and assistance. Task recognition would allow the robot to do any predefined task without any manual interference. We think that these tasks are well suited for a shopping mall. Finally using all our knowledge from our previous experiments, simple modules that we had created and invaluable help and suggestions from our supervisors, we were successful in creating ASMA.

In chapters to follow we will discuss how we built and developed ASMA. In chapter 2, we will be discussing how we built the map and achieved autonomous navigation. In chapter 3, we would discuss the details about our state machine implementation that glues everything together. In chapter 4, we giving implementation details of our robot tasks. In chapter 5, we are sharing the problems we encountered during our initial stage and also while building ASMA. In chapter 6, we share our experiences and conclusion regarding the project. In the appendix that follows you can find the details of the commands used and source code that runs on ASMA.

Chapter 2

Map Building and Navigation

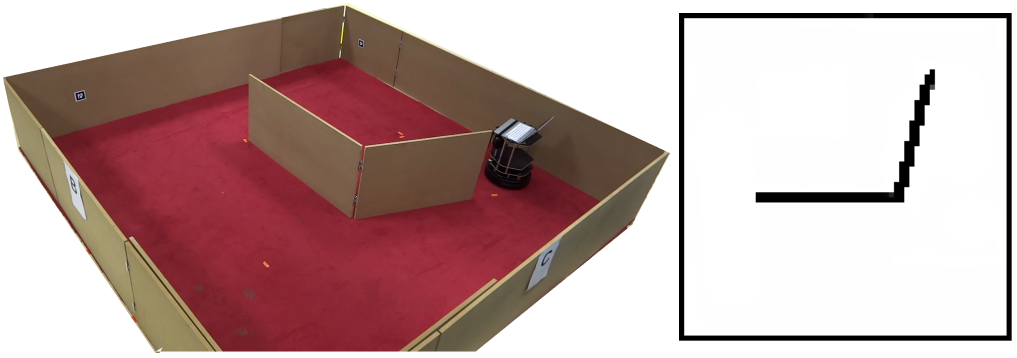
2.1 Map building using SLAM

In this section we discuss our approach of building map by leveraging SLAM. This was done using ROS with the help of the gmapping package which is a third party package by OpenSlam[4]. This provides a depth based SLAM using Rao-Blackwellized particle filter. This is implemented in a node called *slam_gmapping* which takes *sensor_msgs/LaserScan* topic and builds a map using the above mentioned algorithm. As the turtle bot is not equipped with a laser scan, the point cloud data obtained from kinect is transformed into the laser scan data for this purpose. The map generated with this is published in *nav_msgs/OccupancyGrid* topic along with the meta data and entropy in *nav_msgs/MapMetaData* and *std_msgs/Float64* respectively. The constructed map can be saved in yaml format and then later used for other purpose such as Autonomous Navigation.

The map obtained is quite accurate and online. The main difficulty in SLAM using particles with near real-time formulation is to compute the hypotheses for each particle. This hurdle can be removed by carefully reducing the number of particles. But with less number of particles the localization may not be accurate. This problem is solved in gmapping using Rao-Blackwellized particle particle filter. The distribution and resampling techniques described in this algorithm decreases the number of particles without facing any problems that arise due to sub-sampling. This is the reason why gapping is able to deliver very accurate results which are almost real-time.

2.1.1 Implementation

We would now give a basic overview of how we were able to map the arena and save the data as a map file. The details about the commands used for this process can be found in Appendix A. We first start our turtlebot and place it in the arena that needs to be mapped. After our turtle bot is up and running, we then launch the gmapping package in ROS. This task is performed on the turtle bot. This starts up the gmapping package and starts transmitting the occupancy grid. In order to visualize and interact with the map, we take the help of rviz. The robot is then moved manually using joystick or keyboard to cover adequate area of the arena for a successful map creation.After the robot has finished moving through the map, we save the map in a pgm file that holds the occupancy grid and a yaml file that holds the configuration settings. These pairs of file would be used in the next phase of our project where we would be moving around the arena with the help of this data. The map obtained is not perfect so we used an image editor to clean up the map and remove any noise or error that was introduced during the mapping process. Figure 2.1 shows the actual arena and the final map that we have used for this project.



(A) Picture of the arena (B) Map

FIGURE 2.1: Map used

2.2 Autonomous navigation of Turtle bot

The navigation of the turtlebot was achieved using the *turtlebot_navigation* package[5]. This package is developed by OSRF specifically for the use with the turtlebot. This package internally uses the *amcl* package [6] and the *move_base* package [7] in-conjunction with *navigation_velocity_smoother* and *kobuki_safety_controller* packages.

Point to point turtlebot movement was achieved by using the *move_base* package [7]. This package helps in moving the turtle bot to any desired position using the navigation stack. For localization of the robot we use the *amcl* package. The *amcl* package uses adaptive Monte Carlo localization to localize the turtlebot using the data received from the laser scan and the odometery.

The *move_base* along with *amcl* is very accurate and robust but it still required map specific tuning of the parameters to navigate smoothly in our map. These configuration could be changed from the xml files listed below.

- *base_local_planner_params.yaml* :- We set limits on maximum and minimum linear and angular velocity and acceleration
- *costmap_common_params.yaml* :- We bloated the radius of the robot and increased the inflation radius to inflate the obstacles in the map.
- *global_costmap_params.yaml* :- We set a high frequency for updating.
- *local_costmap_params.yaml* :- Similar frequency updating scheme as done for the global costmap was applied here

The parameters were changed in order to to reduce the speed of the robot, make it follow the computed path more strictly and finally to bloat the robot radius and the walls of the map. We will now discuss the motive behind changing the parameters. Restricting the robot's motion was done in order to give ample time for the map to update before robot makes any decision thus avoiding collisions. The walls and the robot model was bloated so that there would be no collision with the walls. The algorithm would be extra cautious near corners (figure 2.1b) or if the robot is near a wall. The values were tuned according to the localization error in *amcl*, thus making sure that even with a large localization error due to *amcl* the robot would still be able to avoid collision. We also made changes to the weight given to the path distance (*pdist_scale*) and the goal distance (*gdist_scale*). Path planned was given more weight so as to make the robot more closely to the optimum calculated path (figure 2.1a). This was done to avoid the local optimization done by *move_base* while the robot was turning or performing pose alignment at the destination. This local optimization would create a curvy path plan that would be slightly different from the optimal path thus might collide with the near

by walls. An overview of how these configuration along with the sensors, controllers and the map server communicate with each other is show in figure 2.2

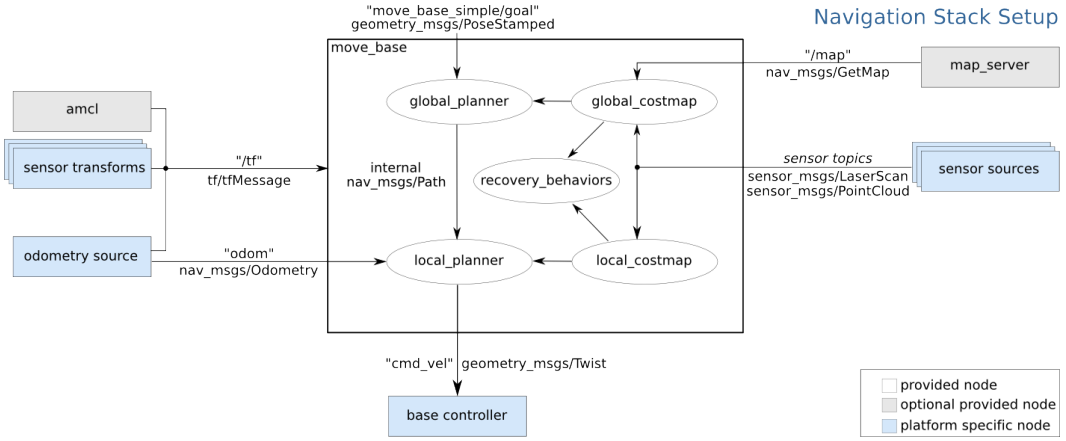
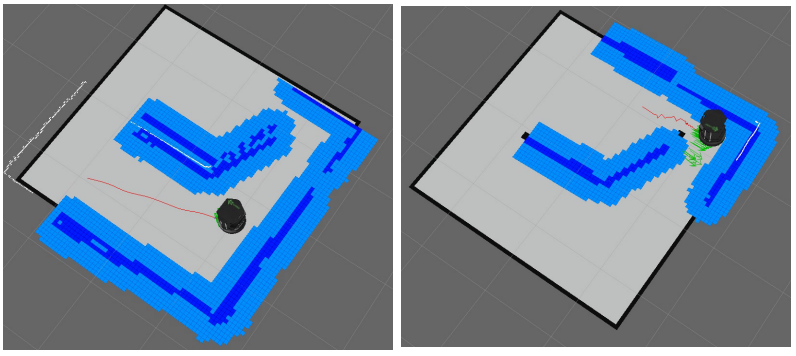


FIGURE 2.2: Overview of Move Base

To avoid obstacles we set the configuration to update the map with the highest frequency possible. This would update the map whenever the pose of the robot would change. Thus any obstacles seen by the laser scan would be detected and taken into consideration while generating a path and also while it is moving . There seems to be a limitation on the range of the sensor as we are using the 3D point information obtained from kinect to create a virtual laser scan. Due to the limitation of kinect the minimum distance that can be measured is only 500-800 mm [8]. This is the reason why it will not detect an object that is very close to the camera. Also to make sure that the error due to localization is not added up we configure the map to gradually purge the old data as the new data comes it. Thus at any point in time the robot is sure about only a section of a map as shown is figure 2.3.



(A) Turtelbot moving to its goal (B) Turtle arriving to a corner

FIGURE 2.3: Move Base in action

2.2.1 Implementation

The moving of the turtlebot programmatically was done by passing the *MoveBaseGoal* messages to the *MoveBaseAction* action client. The *MoveBaseGoal* message is comprised of *PoseStamped* message that includes a header and a desired pose. The header contains information about the message such as the sequence(*seq*), time stamp (*timestamp*) and its frame id (*frame_id*). The *Pose* contains information about the desired location point and the final orientation. The desired position is the given as a 3D coordinate and the orientation by a quaternion coordinate. We move between 4 points in the map for the patrolling scenario. These pose coordinates have already been stored before running the paroling demo. The starting pose of the robot is predefined in the yaml file but can be changed by supplying the initial pose programmatically or by using RVIZ if required. The robot moves along these predetermined points and perform tasks by searching for tags located near these points. This will be discussed in detail in our next chapter.

Chapter 3

State Machine

An autonomous robot is expected to perform multiple tasks depending on various condition and requirements at any given time. Performing a large array of tasks, high a tight coupling becomes quite complex using regular programming methods. Even if they are implemented using traditional programming techniques, it becomes quite hard to debug and know the current situation of the system making it difficult enhance and scale the system. Therefore for managing the system, we have chosen a more structured approach by using finite state machines which made our system more loosely coupled. For this purpose we have used the *smach* package [9] to build our state machine. It was very useful in creating a complex state machine, perform fast prototyping, to handle errors and visualize the process. It provides a simple and easy platform to create a state, define the functionalities of the state, and decide the trigger and output of the state.

To visualize the current interaction between the states we used *smach_viewer* package[10]. This package let us visualize the current state, all possible state transitions, data passed between the states and many more giving a complete freedom to the user to introspect and debug the system. The visualization of our complete state machine is shown in figure 3.1. Here we can see all the state and all the possible transitions. The current state, which is the **DetectTag** state, has been highlighted.

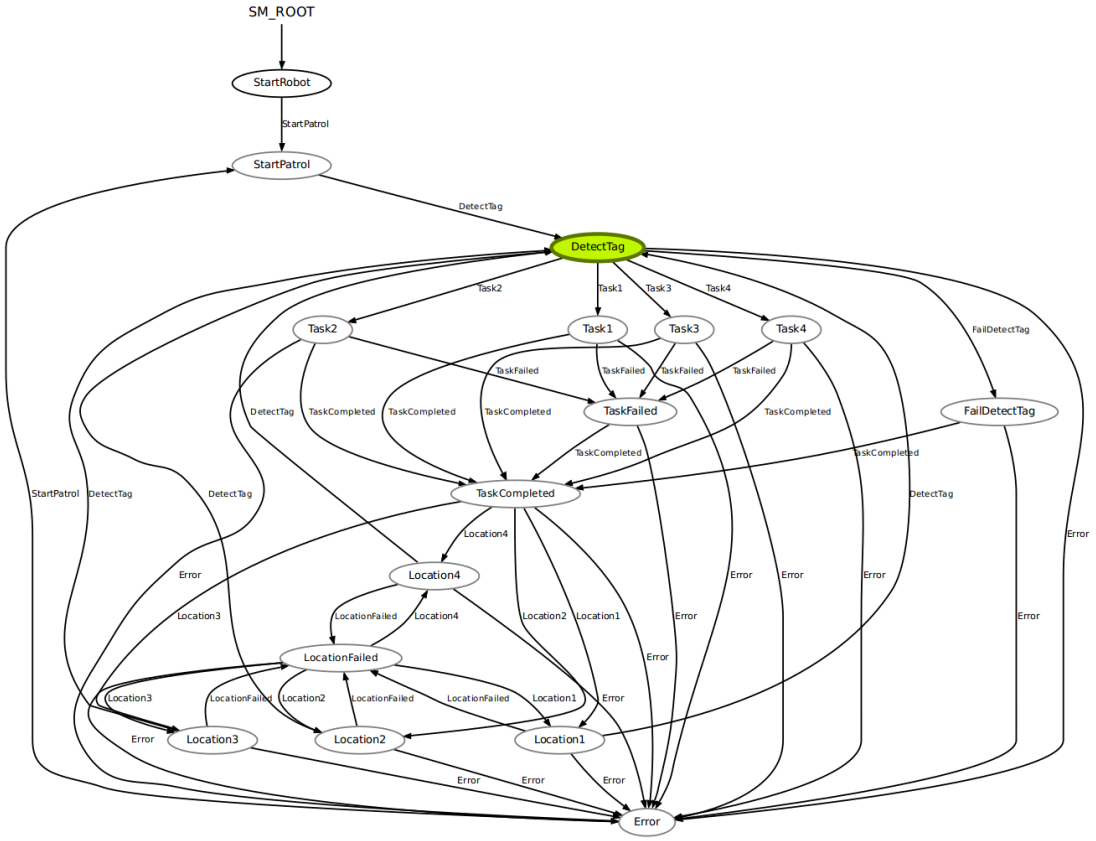


FIGURE 3.1: State Machine visualization of all the states

3.1 State definition and implementation

In this section we would be defining all the states, their purposes and all their outcomes. This along with figure 3.1, it should give an overview of all patrolling scenarios in our system. We have defined a state for every substantial action that a robot takes while patrolling. We decided on this breakup of states as making states more fine grained would add an overhead and also make the overall process seem more complicated and hard to debug. On the other hand clubbing together states would not completely allow us to exploit all the features of a state machine.

Our process first starts with **StartRobot** in which all the required all the initialization required is performed. It is also the state where any initial task, such as an introductory self explanation of the robot, which needs to execute only once is performed. From this state we can only move to the **StartPatrol**. This is a dummy state that acts as the entry point to our patrolling routine. If error occurs during patrolling everything is reseted and it comes back to this state to start everything again.

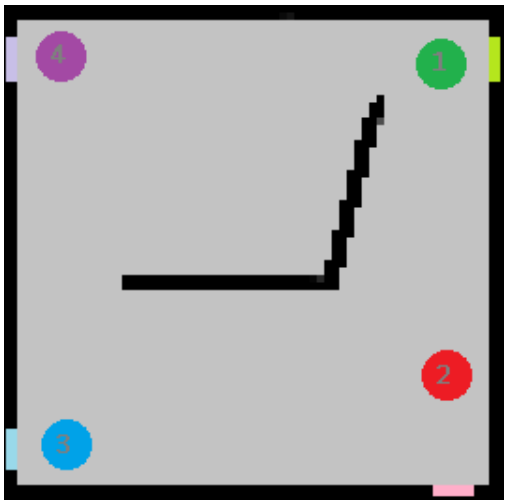


FIGURE 3.2: Map visualization of all the location (circles) and tags (lines)

We can only move to **DetectTag** from the **StartPatrol** state. A simplified view of the map with location represented by circles and tags depicted by lines is shown in figure 3.2. This will help the reader visualize the whole process as we discuss them in this part. In **DetectTag** state we detect tags placed on the walls of the arena to perform specific tasks according to the tag number. After successfully detecting a tag our state machine can move to one of the 4 task states where the relevant task would be performed. Each of these task run only when required and are evoked dynamically so that it doesn't overload the system and also to avoid unnecessary interference between the tasks. If the tag detection fails then it would go to **FailDetectTag** state. In the **FailDetectTag** state it would notify the system and would go the a common state **TaskCompleted** indicating that the task has been completed. All the 4 task states would also go there if it successfully completes the task. If a task fails it would go to **TaskFailed** which also informs the system of the failure and then proceeds forward.

When the state machine has finished performing tasks it arrives to the **TaskCompleted** state. Here according to the current state of the system it would go to one of the 4 location. If the robot reaches the desired location it would move back again to **DetectTag** state where it will look for the tag to perform the next task. If moving to a given location fails, due to obstacle or error during map update, it moves to the next location in the state. This makes sure that the robot is always at a valid location to search for the next tag. At any point if the sate machine encounters an error it moves to the **Error** state where the error is notified and the state machine is reseted to **StartPatrol**.

Chapter 4

Robot Tasks

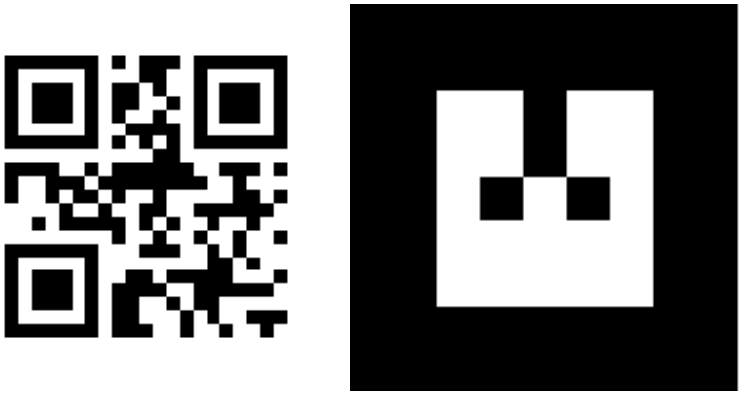
In this chapter we would be discussing the 4 tasks namely, visual servoing, face recognition, voice control and manual override that the robot would be performing while patrolling. We would also discuss the tag detection feature that decides what task has to be performed. Implementation details along with the a brief theory would be presented for all of them.

4.1 Tag Detection

This is one of the most vital functionality for a patrol robot. It helps the to robot perform the right task at the right location and can also be extend to localize the robot more accurately if the pose of the tag is known. To make this task simple the approximate location of tags are predefined i.e the tags would be present near one of the 4 locations. This makes this task relatively challenging by searching for the tag near the vicinity but does not make it tedious and boring by searching every nook and cranny for the tags. We also argue that the same technique with a simple modification of adding more the search location around the map would result in a tag detection algorithm with no predefined points.

We had spent some time deciding the right tag to use. We had initially experimented with QR codes which gave promising results for close range tags. But it had a serious limitation, it had a hard time detecting tags beyond 500 mm[11] as the smallest pixel has to cover at least 2 pixel for a basic detection. The smallest pixel in a QR code is

relatively smaller, thus harder to detect, compared to other more simple tags such as AR tags as we can see in figure 4.1.



(A) QR tag (B) AR tag

FIGURE 4.1: QR and AR tags both representing 0

We therefore used AR tags [12] for tag detection. The time spent on QR codes did not go to waste as we eventually ended up using it for our visual Servoing module. This package is a wrapper around the Alvar library. To set up AR tags we first had to generate tags according to our need. As we have 4 predefined tasks we created 4 AR code ranging from 0-3. For the detection part there are 2 options for this package, one using kinect + RGB data and other using just the RGB data. The kinect data is just used to better determine the pose of the tags in exchange for a larger bandwidth and more computational power. As getting a very accurate pose of the tags was not our prime objective we used the light version that uses only RGB data to detect the tags. Using only RGB data was much faster and lighter compared to it's bulky kinect + RGB counterpart.

The light weight nature of this package allowed us to run it continuously during the patrolling of the turtlebot. To optimize this process further we had to compress the image, the reason for doing so has been discussed in detail in Chapter 5. We subscribed to *visualization_marker* node generated by the package to read the id of the detected tag. The tags are detected en-route to their next location. Even though detection is very accurate we added an extra step just to make sure that tag detection is not missed. The turtle bot after reaching its desired location would turn 360 degree in-situ and search for the tag. This in conjunction with the previously detected tag value would guarantees a robust detection. We also take the extra precaution of rotating 360 degrees as the tags could be any where and not restricted just to the outer boundary of the map.

4.2 Manual Override

Manual override is the most enjoyable task of our robot. It allows people to drive our robot with a joystick as shown in Figure 4.2. In our scenario this would be for fun, during robot's automatic navigation failure and also would help people to make the robot carry their shopping bags. In this task we used the *joy* package [13] and we subscribe to it's *joy_node* to get joystick events. We then published *twist* messages to *cmd_vel* topic to control and drive the robot. However it's not mandatory to use joystick as a control device. As we publish the messages to *cmd_vel* topic, any type of controller could be used thanks to the ROS architecture. During our prototyping stage, we had controlled the robot with a mouse connected to the computer. This could be used as tool used by a remote operator to control the bot. So for this reason we call this task as manual override to imply that we can use any input device to control robot. This task is activated when a specific tag is detected after which the control of the robot is transfered to the user. In order to stop the manual override task, user presses a specific button (in our case back buttons of the joystick), after which the robot continues its patrol route.

FIGURE 4.2: Logitech Joystick

4.3 Voice Control and Feedback

Speech recognition and text to speech plays an important part for a patrol robot. It allows it to iteract with people around it and makes it more approachable. Speech recognition is done using *pocketsphinx* package which internally uses CMU Pocket Sphinx [14] speech recognizer developed by Carnegie Mellon University. It is a lightweight speech

recognition engine which is designed to be used in hand held devices. It may not be very robust compared to its full featured counter part but in our case where we have to recognize only a few set of words it works quite well.

In order to customize the voice recognition module to detect from our own list of keywords, we created a lexical and language model using Sphinx Knowledge Base Tool [15]. We supplied 4 phrases namely **Location One**, **Location Two**, **Location Three** and **Location Four** that defines the 4 location the robot would move to. After compiling these words we get a language dictionary as shown below where every word is split into smaller chunks of sound that make the word.

```
FOUR F AO R
LOCATION L OW K EY SH AH N
ONE W AH N
ONE(2) HH W AH N
THREE TH R IY
TWO T UW
```

Along with this we also have a language model that helps to identify the order in which these pattern would occur i.e. to detect Location "number" together rather than just the number. We dynamically subscribe to the */recognizer/output* topic when the voice recognition starts and wait for the output to match one of the location. After successfully detecting the destination location we simply command the robot to move there using our navigation framework.

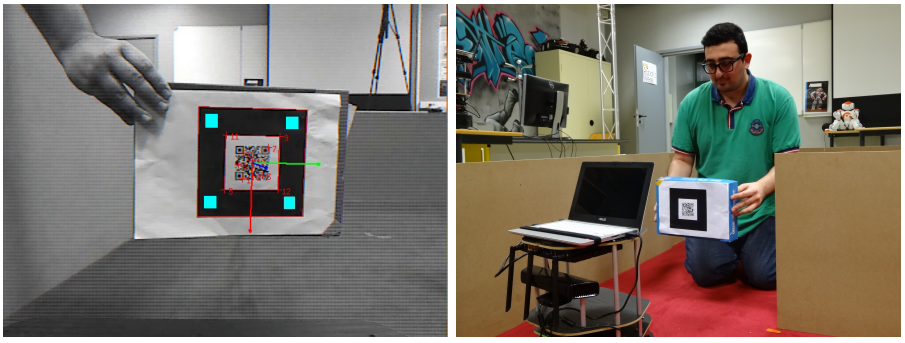
The *sound_play* package [16] was used to provide speech synthesis. We have written a wrapper to play the sound via python by providing a string of name or a file which would then be read out aloud. This functionality was added to make the robot sounds more interactive and also to signal completion of any event or task performed in a human friendly manner.

4.4 Visual Servoing

Visual servoing would help the patrol robot track or follow a person, a moving object or even other robots. This kind of technique would allow for accurate and smooth following.

For this task we try to track an object and move the turtlebot by moving the object. To obtain an accurate pose of the object we use a object of known geometry, which is a QR code in our case, for tracking. The detection of the QR code is done by using *visp_auto_tracker* from the *vision_visp* stack [17]. This gives a very accurate pose of the object(figure 4.3a) allowing for a smooth visual servoing. Modification to the code had to be done to meet our requirement to publish various details such as the data in the QR code for our consumption which were previously not published. The updated code can be found in Appendix B.

The tracker works by first detecting the QR code pattern. After the detection, model based tracker are initialized with this detected location. This allows for the QR Code to be tracked robustly even at a great distance. If at any point the tracker is lost it has to be reinitialized where a new detection is performed. The object being detected has to be described in 3D using VRML. It has already been done for the QR code but would be required for any other object if it is used. This is done to get a more precise pose estimation in 3D as we use this data for visual servoing. It also required the camera to be calibrated so we had done a quick calibration of our kinect RGB camera. After this we configured the library to read the data from the kinect's RGB camera along with the dimension of the QR code we were using. A viewer (*visp_tracker_view*) was linked to view the results. The output of the viewer has been show in figure 4.3a.



(A) Pose estimation of QR tag (B) Robot following the tag

FIGURE 4.3: Visual Servoing

After a successful pose estimation of the QR code we use this data to follow the OR code as it moves. For this implementation we used *demo_pioneer* package [18] as our base. This was implemented for Pioneer robot with camera mounted on a PT-head. A model for the turtlebot was created with relevant iteration matrix and transformation

matrix between the camera frame and the mobile platform. The modified code with the the turtlebot model can be found in Appendix B.

For the turtlebot robot we have 2 velocity which are translation across x and rotation across z axis. These velocity control the robot and are defined as (v_x, w_z). The interaction matrix defined between the robot frame with is origin at the point between the wheels and the object frame is defined as

$$ {}^{e}\mathbf{J}_e = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} $$

The control law $\mathbf{v}$ is then defined as $\mathbf{v} = {}^{e}\mathbf{J}_e \begin{pmatrix} v_x \\ w_z \end{pmatrix}$

This give a simple yet elegant visual servoing scheme for smooth tracking.

4.5 Face Detection

Face detection is an important task to be performed in a shopping mall scenario. It enables the robot to identify a person and perform customized tasks. We decided to use this feature because in our scenario many customers visit the shopping mall frequently. So learning and welcoming customers would be a valuable feature to attract people and also learn about their shopping pattern. This task specifically is quite vast and open to further development. We have left topics such as online face learning, learning customer behaviors and face tracking as a future work.

The kinect RGB camera is placed very low and is not ideal for face detection. For this purpose we have used the camera of the netbook mounted on the turtlebot to perform this task. As this is quite an intensive task we activate the face recognition package only when the specific task is evoked.

To perform face detection we used the *face_recognition* package [19]. As it just a simple wrapper around the OpenCV module for face detection, we just used this package without the reinventing the wheel. In this implementation face recognition is done by offline training, using eigenfaces algorithm for recognition. The eigenfaces implementation in OpenCV detects a face with an average of 70% confidence level, we set this as our threshold to recognize the face. The package is first used to perform offline training of faces by using the stored images of 2 person. After that detection of a face in a live video feed is done using OpenCV's haarcascade frontal face database. After a face has been extrated it is compared against the database of our images to recognize the person. After recognition a message is sent to *face_recognition/result* which we subscribe to, after which we alert the system of the detection. Below is a screenshot of a successful detection during a patrol.

FIGURE 4.4: Face being Detected

Chapter 5

Problems Faced

In this chapter we would discuss a list of problems faced by us and their solution. This would help any one working on a similar project to learn from our mistakes and to avoid or deal with them in a better way.

The first problem we faced was during the navigation of the robot. Initially the map we worked on was more complicated than the one we are currently using. There were few places on the map that the turtlebot had a hard time moving through. This turned out to be the problem due to error in *amcl* localization. As there is always a small error in the localization using *amcl* it has to be taken care of. Therefore the minimum distance in a map should be less than the sum of the robot diameter and twice the localization error(one for each side) due to localization. One can also tweak the parameters as explained in Chapter 2 to make the motion of the robot more predictable. After simplifying the map and changing the parameters we were able to move around the arena with no trouble.

We also faced problem while choosing between the different types of tags such as QR and AR for tag detection. As described in Chapter 3 it is wise to choose the tag according to the need. AR tags have fewer details thus can be used when one need to detect a tag from the large distance. For a more intimate interaction with the robot it is advised to use QR code that can hold more information and along with its error correction feature it can be quite robust.

While integrating the AR tags with our system we faced 2 problems. First problem was the overwhelming data transfer from the turtle bot to the workstation. As both the

depth data and RGB data was passed at regular intervals it had choked the bandwidth preventing a smooth functioning of the system. First we used an alternate method by using only RGB data. Although this was much lighter it was not enough for our requirement. We tried processing the data in the turtle bot itself as the processing was not very intensive and transferring only the data of the tag detected would be much lighter. This is when we had another problem. As this package uses *eigen* library tuned for a 64 bit architecture it does not work in our turtlebot's netbook. Though there are some hack to make it work it does not work as well. Finally the problem was solved by compressing the RGB data and transmitting it over the network. This greatly reduced the bandwidth requirement. It is suggested to the reader to also pay attention to the computer architecture while using any library or packages. Secondly it is always advised to compress the RGB or depth images while transmitting if there is some restriction on the bandwidth.

We had used *rosbuild* for all of the packages and code except the face recognition package. This required *catkin* to build as *rosbuild* was configured to work only on later version of ROS. This create a lot of issues with locating and running our old packages that were build using *rosbuild*. We had to change our *.bashrc* file to search for packages in both the *catkin* directory and the *rosbuild* directory in order for everything to work. Although this hack works we would not recommend using it as there might be conflicts between them. It is highly recommended to use one common build system for everything thing making things simple and easy.

Our last problem was encountered while we were integrating everything together. We had two major problems when we first ran all the packages simultaneously. The first was the overwhelming processing and bandwidth required for the complete thing to work simultaneously. Secondly there were conflicts with the topics that were remapped or nodes (such as *cmd_vel*) with inputs from multiple packages. One way of solving this was to keep track of all packages and all the topics that they were subscribing or publishing to prevent conflicts and analyze every topic on the amount of data transfered so that some compression techniques could be applied. This approach would take a long time and is definitely not scalable. To solve this problem we used *roslaunch* and *subprocess* api in python to run each functionality only when required. They would run as a separate process for each task when evoked and would be closed once there is no need for it. It works very well in our case as we run only one task at a time. This

reduced the bandwidth required , prevented any conflicts and more importantly it made it scalable. Adding a new task or a functionality is now much easier and predictable.

Chapter 6

Conclusion

In conclusion the complete patrolling scenario along with tag detection and 4 individual task was preformed. The robot successful patrols the whole arena and visits 4 specific locations multiple times without drifting. The tag detection is quite robust and detect all the 4 tags that the turtlebot is suppose to detect. Voice detection has some problems detecting words clearly but works after trying for multiple times. The manual override works as designed. We have tested the face recognition module with 2 set of images and they give correct output with high confidence most of the time. The visual servoing also showed promising results with robust detection and tag following. All the tasks and scenarios were executed as planned and no bugs or glitches were noticed while testing. A short video showing our results can be viewed at https://www.youtube.com/watch?v=QVTxOSaessE.

Overall we learned the nitty gritty of ROS especially navigation and vision. We also learned about the importance of using state machines to amalgamate all the task and goals of the robot to make it robust, easy to debug and scalable. We also found benefits of using existing packages rather than reinventing the wheel, in the meantime learning everything on the workings of a package so it can be tuned and modified specifically for our needs.

Appendix A

Map Building & Navigation

A.1 Map Building

Launch gmapping demo.(turtlebot)

```
roslaunch turtlebot_bringup minimal.launch
roslaunch turtlebot_navigation gmapping_demo.launch
```

Open Rviz for visualization. (workstation)

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Using Joystick or keyboard for tele operation.(workstation)

```
rosrun joy joy_node
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Saving the map.(turtlebot)

```
rosrun map_server map_saver -f ~/Maps/map_or
```

A.2 Navigation

Starting AMCL with a map. (turtlebot)

```
roslaunch turtlebot_navigation amcl_demo.launch map_file:=~/Maps/map_or.yaml
```

For visualization and navigation.(workstation)

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Appendix B

Source Code

We have uploaded our code in https://github.com/emreozanalkan/ASMA as a public repository which includes the complete set of files required to run this package. The forked and modified version of **Vision ViSP** (https://github.com/abpant/vision_visp) and **demo_pioneer** (https://github.com/abpant/demo_pioneer) has also been uploaded as a public repository.

To make it easy for our reader we have also added our main python script(patrol.py) that controls everything using state machine has been listed below.

```
#!/usr/bin/python
import welcome
import roslib; roslib.load_manifest('asma')
import rospy
import subprocess
import shlex
import psutil
import smach
import smach_ros
import rospy
import actionlib
import roslaunch
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, Point, Quaternion, Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker, MarkerArray
from math import radians, pi
from visualization_msgs.msg import Marker
from sound_play.msg import SoundRequest
from sound_play.libsoundplay import SoundClient
from sensor_msgs.msg import Joy
from face_recognition.msg import FaceRecognitionActionResult,
    FRClientGoal
```

```
24 from std_msgs.msg import String
25 import os
26 import signal
27
28 currTask = 1 #
29 nextLoc  = 1
30 p = rospy.Publisher('cmd_vel', Twist)
31 #soundhandle = SoundClient()
32 class StartRobot(smach.State):
33     def __init__(self):
34         smach.State.__init__(self, outcomes=['StartPatrol'])
35
36     def execute(self, userdata):
37         rospy.loginfo('Executing state Start Robot')
38         rospy.sleep(1)
39         #TODO: Give self explanation do somethign here like
   speak or move
40         speak('Hello, World! I am ASMA. It stands for Automatic
   Shopping Mall Assitant.')
41         rospy.sleep(4)
42         speak('I will be your guidance through your experience
   in this shopping mall')
43         rospy.sleep(2)
44         speak('I will be patrolling different locations and
   doing tasks for you.')
45         rospy.sleep(2)
46         speak('Please join me in this wonderful experience!')
47         rospy.sleep(2)
48
49         return 'StartPatrol'
50
51 class StartPatrol(smach.State):
52     def __init__(self):
53         smach.State.__init__(self, outcomes=['DetectTag'])
54
55     def execute(self, userdata):
56         rospy.loginfo('Executing state Start Patrol')
57         speak('Now I will start patrolling. Be carefull! Ha ha
   ha!')
58         rospy.sleep(3)
59         #TODO: Implement patrolling here, like moving the next
   location, e.g to the 'nextLoc'
60         return 'DetectTag'
61
62 class DetectTag(smach.State):
63     def __init__(self):
64         smach.State.__init__(self, outcomes=['Task1','Task2','
   Task3','Task4','Error','FailDetectTag'])
65
66     def detectARTag(self):
67         twist = Twist()
68         twist.linear.x = 0
69         twist.linear.y = 0
70         twist.linear.z = 0
71         twist.angular.x = 0
72         twist.angular.y = 0
```

```
73          q_angle = quaternion_from_euler(0, 0, 0.175, axes='sxyz'
    )
74          q = Quaternion(*q_angle)
75          twist.angular.z = 0.8
76          for i in range(115):
77              p.publish(twist)
78              print 'Rotating'
79              rospy.sleep(0.1)
80              print 'Current Task: ' + str(currTask)
81          return currTask
82
83      def execute(self, userdata):
84
85          rospy.loginfo('Executing state Detect Tag')
86          speak('Detecting tags!')
87          rospy.sleep(1)
88
89          val = self.detectARTag()
90
91          if val == 0 :
92              return 'Task1'
93          elif val == 1 :
94              return 'Task2'
95          elif val == 2 :
96              return 'Task3'
97          elif val == 3 :
98              return 'Task4'
99          elif val == -1 :
100             return 'Error'
101         else :
102             return 'FailDetectTag'
103
104
105 class FailDetectTag(smach.State):
106     def __init__(self):
107         smach.State.__init__(self, outcomes=['TaskCompleted','
    Error'])
108     def informFailDetect(self):
109         return
110
111     def execute(self, userdata):
112         rospy.loginfo('Executing '+ type(self).__name__)
113         rospy.sleep(1)
114         self.informFailDetect()
115         return 'TaskCompleted'
116
117 class Error(smach.State):
118     def __init__(self):
119         smach.State.__init__(self, outcomes=['StartPatrol', '
    STOP'])
120
121     def informErrorDetected(self):
122         return 1
123
124     def execute(self, userdata):
125         rospy.loginfo('Executing '+ type(self).__name__)
126         rospy.sleep(1)
```

```
127         errorCode = self.informErrorDetected()
128 
129         if errorCode == 1:
130             return 'StartPatrol'
131         else:
132             return 'STOP'
133 
134 class TaskCompleted(smach.State):
135     def __init__(self):
136         smach.State.__init__(self, outcomes=['Location1','
    Location2','Location3','Location4','Error'])
137     def findNextLocation(self):
138         return nextLoc
139 
140     def execute(self, userdata):
141         rospy.loginfo('Executing '+ type(self).__name__)
142         rospy.sleep(1)
143         val = self.findNextLocation()
144         if val == 0 :
145             return 'Location1'
146         elif val == 1 :
147             return 'Location2'
148         elif val == 2 :
149             return 'Location3'
150         elif val == 3 :
151             return 'Location4'
152         elif val == -1 :
153             return 'Error'
154         else :
155             return 'Error'
156 
157 
158 class TaskFailed(smach.State):
159     def __init__(self):
160         smach.State.__init__(self, outcomes=['TaskCompleted','
    Error'])
161     def informTaskFailed(self):
162         return
163 
164     def execute(self, userdata):
165         rospy.loginfo('Executing '+ type(self).__name__)
166         rospy.sleep(1)
167         self.informTaskFailed()
168         return 'TaskCompleted'
169 
170 
171 class Task2(smach.State):
172     def __init__(self):
173         smach.State.__init__(self, outcomes=['TaskCompleted','
    TaskFailed','Error'])
174         self.np = 0
175     def callbackVoice(self,data):
176         print 'our data:' + data.data
177         if data.data is not None:
178             #speak('Going to ' + data.data )
179             if data.data == "location one":
180                 self.np=0
```

```
181                 self.detected=2
182             elif data.data == "location two":
183                 self.np=1
184                 self.detected=2
185             elif data.data == "location three":
186                 self.np=2
187                 self.detected=2
188             elif data.data == "location four":
189                 self.np=3
190                 self.detected=2
191
192
193     def performTask(self):
194         speak('Executing '+ type(self).__name__)
195         speak('Waiting for voice commands sir!')
196         #
197         self.voice = subprocess.Popen('roslaunch pocketsphinx
    robocup.launch', shell=True, preexec_fn=os.setsid)
198         rospy.sleep(4)
199         self.voiceFeedback =  rospy.Subscriber("/recognizer/
    output", String, self.callbackVoice)
200         #rospy.sleep(30)
201         self.detected=1
202         self.thresh = 0
203
204
205         while(self.detected==1 and self.thresh < 30) :
206             print 'In sound loop'
207             rospy.sleep(1)
208             self.thresh = self.thresh +1
209
210         self.voiceFeedback.unregister()
211         os.killpg(self.voice.pid, signal.SIGTERM)
212         mp.movePoint(self.np)
213
214         return 1
215
216     def execute(self, userdata):
217         rospy.loginfo('Executing '+ type(self).__name__)
218         rospy.sleep(1)
219         val = self.performTask()
220         if val == 1 :
221             return 'TaskCompleted'
222         elif val == 2 :
223             return 'TaskFailed'
224         elif val == 0 :
225             return 'Error'
226
227
228
229 class Task1(smach.State):
230     def __init__(self):
231         smach.State.__init__(self, outcomes=['TaskCompleted','
    TaskFailed','Error'])
232
233         self.doJoy=1
234
```

```
235     def callbackJoy(self,data):
236         print 'Button:' + str(data.buttons[4])
237         if(data.buttons[4] == 1) :
238             print 'data button 4 1'
239             self.doJoy= 2
240             self.joysub.unregister()
241             self.processJoy.stop()
242             print 'Is Alive: ' + str(self.processJoy.is_alive())
243         rospy.loginfo(rospy.get_name()+ str(data.axes))
244         t = Twist();
245         t.linear.x = data.axes[1] / 10;
246         t.angular.z = data.axes[2];
247         p.publish(t)
248 
249     def performTask(self):
250         speak('Executing '+ type(self).__name__)
251         speak('I am giving my control to you. Now you can move
    me around with joystick.')
252 
253         package = 'joy'
254         executable = 'joy_node'
255         joyNode = roslaunch.core.Node(package, executable)
256         launchJoy = roslaunch.scriptapi.ROSLaunch()
257         launchJoy.start()
258         self.processJoy = launchJoy.launch(joyNode)
259 
260         self.joysub =  rospy.Subscriber("joy", Joy, self.
    callbackJoy)
261         self.doJoy= 1
262         while(self.doJoy==1) :
263             rospy.sleep(5)
264         return 1
265 
266 
267     def execute(self, userdata):
268         rospy.loginfo('Executing '+ type(self).__name__)
269         rospy.sleep(1)
270         val = self.performTask()
271         if val == 1 :
272             return 'TaskCompleted'
273         elif val == 2 :
274             return 'TaskFailed'
275         elif val == 0 :
276             return 'Error'
277 
278 class Task3(smach.State):
279     def __init__(self):
280         smach.State.__init__(self, outcomes=['TaskCompleted','
    TaskFailed','Error'])
281     def performTask(self):
282         speak('Executing '+ type(self).__name__)
283         speak('I am in visual servoing mode. I will detect and
    follow the tags.')
284         #TODO: Try catch and then handel error
285         self.procServo = subprocess.Popen('roslaunch
    visp_auto_tracker turtletrack.launch', shell=True, preexec_fn
    =os.setsid)
```

```
286             self.procVisp = subprocess.Popen('roslaunch demo_pioneer
        myservo.launch', shell=True, preexec_fn=os.setsid)
287             #print self.procServo.pid
288             #print self.procVisp.pid
289             #self.pS = psutil.Process(self.procServo.pid)
290             #self.pV = psutil.Process(self.procVisp.pid)
291             rospy.sleep(60)
292             #self.procServo.kill()
293             #self.procVisp.kill()
294             #self.pS.kill()
295             #self.pV.kill()
296             os.killpg(self.procVisp.pid, signal.SIGTERM)
297             os.killpg(self.procServo.pid, signal.SIGTERM)
298             return 1
299
300         def execute(self, userdata):
301             rospy.loginfo('Executing '+ type(self).__name__)
302             rospy.sleep(1)
303             val = self.performTask()
304             if val == 1 :
305                 return 'TaskCompleted'
306             elif val == 2 :
307                 return 'TaskFailed'
308             elif val == 0 :
309                 return 'Error'
310
311 class Task4(smach.State):
312     def __init__(self):
313         smach.State.__init__(self, outcomes=['TaskCompleted','
        TaskFailed','Error'])
314         self.clientMSG = FRClientGoal()
315         self.clientMSG.order_id = 0
316         self.clientMSG.order_argument = "none" # "none"
317         self.detected=1
318         self.thresh = 0
319
320
321     def callbackFeedback(self,data):
322         print 'Names:' + data.result.names[0]
323         if data.result.names[0] is not None:
324             speak("I have found you " + data.result.names[0])
325             self.detected=2
326
327
328     def performTask(self):
329         speak('Executing '+ type(self).__name__)
330         speak('Face recognition activated. Let me see if I can
        recognize you!')
331         self.procServ = subprocess.Popen('rosrun
        face_recognition Fserver _confidence_value:=0.7', shell=True,
        preexec_fn=os.setsid)
332         self.procClient = subprocess.Popen('rosrun
        face_recognition Fclient', shell=True, preexec_fn=os.setsid)
333         #print self.procServ.pid
334         #print self.procClient.pid
335         rospy.sleep(4)
```

```
336         self.ClientGoal = rospy.Publisher('/fr_order',
    FRClientGoal,latch=True)
337         print self.clientMSG
338         self.ClientGoal.publish(self.clientMSG)
339         self.faceFeedback =  rospy.Subscriber("/face_recognition
    /result", FaceRecognitionActionResult, self.callbackFeedback)
340         #rospy.sleep(30)
341         self.detected=1
342         self.thresh = 0
343
344
345         while(self.detected==1 and self.thresh < 30) :
346             rospy.sleep(1)
347             self.thresh = self.thresh +1
348
349
350         self.faceFeedback.unregister()
351         os.killpg(self.procClient.pid, signal.SIGTERM)
352         os.killpg(self.procServ.pid, signal.SIGTERM)
353         return 1
354
355     def execute(self, userdata):
356         rospy.loginfo('Executing '+ type(self).__name__)
357         rospy.sleep(1)
358         val = self.performTask()
359         if val == 1 :
360             return 'TaskCompleted'
361         elif val == 2 :
362             return 'TaskFailed'
363         elif val == 0 :
364             return 'Error'
365
366
367 class LocationFailed(smach.State):
368     def __init__(self):
369         smach.State.__init__(self, outcomes=['Location1','
    Location2','Location3','Location4','Error'])
370     def findNextLocation(self):
371         return nextLoc
372
373     def execute(self, userdata):
374         rospy.loginfo('Executing '+ type(self).__name__)
375         rospy.sleep(1)
376         val = self.findNextLocation()
377         if val == 0 :
378             return 'Location1'
379         elif val == 1 :
380             return 'Location2'
381         elif val == 2 :
382             return 'Location3'
383         elif val == 3 :
384             return 'Location4'
385         elif val == -1 :
386             return 'Error'
387         else :
388             return 'Error'
389
```

```
390 class Location1(smach.State):
391     def __init__(self):
392         smach.State.__init__(self, outcomes=['DetectTag','
    LocationFailed','Error'])
393     def moveToPosition(self):
394     global nextLoc
395     mp.movePoint(nextLoc)
396     nextLoc =1;
397         return 1
398 
399     def execute(self, userdata):
400         rospy.loginfo('Executing '+ type(self).__name__)
401         speak('Moving to '+ type(self).__name__)
402         rospy.sleep(1)
403         val = self.moveToPosition()
404         if val == 1 :
405             return 'DetectTag'
406         elif val == 2 :
407             return 'LocationFailed'
408         elif val == 0 :
409             return 'Error'
410 
411 class Location2(smach.State):
412     def __init__(self):
413         smach.State.__init__(self, outcomes=['DetectTag','
    LocationFailed','Error'])
414     def moveToPosition(self):
415     global nextLoc
416     mp.movePoint(nextLoc)
417     nextLoc =2;
418         return 1
419 
420     def execute(self, userdata):
421         rospy.loginfo('Executing '+ type(self).__name__)
422         speak('Moving to '+ type(self).__name__)
423         rospy.sleep(1)
424         val = self.moveToPosition()
425         if val == 1 :
426             return 'DetectTag'
427         elif val == 2 :
428             return 'LocationFailed'
429         elif val == 0 :
430             return 'Error'
431 class Location3(smach.State):
432     def __init__(self):
433         smach.State.__init__(self, outcomes=['DetectTag','
    LocationFailed','Error'])
434     def moveToPosition(self):
435     global nextLoc
436     mp.movePoint(nextLoc)
437     nextLoc =3;
438         return 1
439 
440     def execute(self, userdata):
441         rospy.loginfo('Executing '+ type(self).__name__)
442         speak('Moving to '+ type(self).__name__)
443         rospy.sleep(1)
```

```
444             val = self.moveToPosition()
445             if val == 1 :
446                 return 'DetectTag'
447             elif val == 2 :
448                 return 'LocationFailed'
449             elif val == 0 :
450                 return 'Error'
451 class Location4(smach.State):
452     def __init__(self):
453         smach.State.__init__(self, outcomes=['DetectTag','
    LocationFailed','Error'])
454     def moveToPosition(self):
455     global nextLoc
456     mp.movePoint(nextLoc)
457     nextLoc =0;
458         return 1
459 
460     def execute(self, userdata):
461         rospy.loginfo('Executing '+ type(self).__name__)
462         speak('Moving to '+ type(self).__name__)
463         rospy.sleep(1)
464         val = self.moveToPosition()
465         if val == 1 :
466             return 'DetectTag'
467         elif val == 2 :
468             return 'LocationFailed'
469         elif val == 0 :
470             return 'Error'
471 
472 class MoveBasePoints():
473 
474     def __init__(self):
475         rospy.on_shutdown(self.shutdown)
476         self.waypoints = list()
477         # How big is the square we want the robot to navigate?
478         square_size = rospy.get_param("~square_size", 1.0) #
    meters
479 
480         # Create a list to hold the target quaternions (
    orientations)
481         quaternions = list()
482 
483         # First define the corner orientations as Euler angles
484         euler_angles = (-1.556,-3.104,1.559,-0.003)
485 
486         # Then convert the angles to quaternions
487         for angle in euler_angles:
488             q_angle = quaternion_from_euler(0, 0, angle, axes='
    sxyz')
489             q = Quaternion(*q_angle)
490             quaternions.append(q)
491 
492         # Create a list to hold the waypoint poses
493 
494 
495         oy=2.7;
496         ox = 1;
```

```
497         # Append each of the four waypoints to the list.  Each
    waypoint
498         # is a pose consisting of a position and orientation in
    the map frame.
499         self.waypoints.append(Pose(Point(1.227-ox, 5.028-oy,
    0.0), quaternions[0]))
500         self.waypoints.append(Pose(Point(1.185-ox, 3.260-oy,
    0.0), quaternions[1]))
501         self.waypoints.append(Pose(Point(-1.245-ox, 2.570-oy,
    0.0), quaternions[2]))
502         self.waypoints.append(Pose(Point(-1.289-ox, 5.040-oy,
    0.0), quaternions[3]))
503
504         # Initialize the visualization markers for RViz
505         self.init_markers()
506
507         # Set a visualization marker at each waypoint
508         for waypoint in self.waypoints:
509             p = Point()
510             p = waypoint.position
511             self.markers.points.append(p)
512
513         # Publisher to manually control the robot (e.g. to stop
    it)
514         self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
515
516         # Subscribe to the move_base action server
517         self.move_base = actionlib.SimpleActionClient("move_base
    ", MoveBaseAction)
518
519         rospy.loginfo("Waiting for move_base action server...")
520
521         # Wait 60 seconds for the action server to become
    available
522         self.move_base.wait_for_server(rospy.Duration(60))
523
524         rospy.loginfo("Connected to move base server")
525         rospy.loginfo("Starting navigation test")
526
527         # Initialize a counter to track waypoints
528         i = 0
529
530     def movePoint(self, i):
531         self.marker_pub.publish(self.markers)
532
533         # Intialize the waypoint goal
534         goal = MoveBaseGoal()
535
536         # Use the map frame to define goal poses
537         goal.target_pose.header.frame_id = 'map'
538
539         # Set the time stamp to "now"
540         goal.target_pose.header.stamp = rospy.Time.now()
541
542         # Set the goal pose to the i-th waypoint
543         goal.target_pose.pose = self.waypoints[i]
544
```

```
545         # Start the robot moving toward the goal
546         self.move(goal)
547
548
549
550     def move(self, goal):
551         # Send the goal pose to the MoveBaseAction server
552         self.move_base.send_goal(goal)
553
554         # Allow 1 minute to get there
555         finished_within_time = self.move_base.wait_for_result(
    rospy.Duration(60))
556
557         # If we don't get there in time, abort the goal
558         if not finished_within_time:
559             self.move_base.cancel_goal()
560             rospy.loginfo("Timed out achieving goal")
561         else:
562             # We made it!
563             state = self.move_base.get_state()
564             if state == GoalStatus.SUCCEEDED:
565                 rospy.loginfo("Goal succeeded!")
566
567     def init_markers(self):
568         # Set up our waypoint markers
569         marker_scale = 0.2
570         marker_lifetime = 0 # 0 is forever
571         marker_ns = 'waypoints'
572         marker_id = 0
573         marker_color = {'r': 1.0, 'g': 0.7, 'b': 1.0, 'a': 1.0}
574
575         # Define a marker publisher.
576         self.marker_pub = rospy.Publisher('waypoint_markers',
    Marker)
577
578         # Initialize the marker points list.
579         self.markers = Marker()
580         self.markers.ns = marker_ns
581         self.markers.id = marker_id
582         self.markers.type = Marker.SPHERE_LIST
583         self.markers.action = Marker.ADD
584         self.markers.lifetime = rospy.Duration(marker_lifetime)
585         self.markers.scale.x = marker_scale
586         self.markers.scale.y = marker_scale
587         self.markers.color.r = marker_color['r']
588         self.markers.color.g = marker_color['g']
589         self.markers.color.b = marker_color['b']
590         self.markers.color.a = marker_color['a']
591
592         self.markers.header.frame_id = 'map'
593         self.markers.header.stamp = rospy.Time.now()
594         self.markers.points = list()
595
596     def shutdown(self):
597         rospy.loginfo("Stopping the robot...")
598         # Cancel any active goals
599         self.move_base.cancel_goal()
```

```
600             rospy.sleep(2)
601             # Stop the robot
602             self.cmd_vel_pub.publish(Twist())
603             rospy.sleep(1)
604
605 # Classes
606 # - StartRobot - outcomes=['StartPatrol']
607 # - StartPatrol - outcomes=['DetectTag']
608 # - DetectTag - outcomes=['Task1', 'Task2', 'Task3', 'Task4', '
    Error', 'FailDetectTag']
609 # - Error - outcomes=['StartPatrol', 'STOP']
610 # - FailDetectTag - outcomes=['TaskCompleted', 'Error']
611 # - TaskCompleted - outcomes=['Location1', 'Location2', '
    Location3', 'Location4', 'Error']
612 # - TaskFailed - outcomes=['TaskCompleted','Error']
613 # - Task1 - outcomes=['TaskCompleted', 'TaskFailed', 'Error']
614 # - Task2 - outcomes=['TaskCompleted', 'TaskFailed', 'Error']
615 # - Task3 - outcomes=['TaskCompleted', 'TaskFailed', 'Error']
616 # - Task4 - outcomes=['TaskCompleted', 'TaskFailed', 'Error']
617 # - LocationFailed - outcomes=['Location1', 'Location2', '
    Location3', 'Location4', 'Error']
618 # - Location1 - outcomes=['DetectTag', 'LocationFailed', 'Error
    ']
619 # - Location2 - outcomes=['DetectTag', 'LocationFailed', 'Error
    ']
620 # - Location3 - outcomes=['DetectTag', 'LocationFailed', 'Error
    ']
621 # - Location4 - outcomes=['DetectTag', 'LocationFailed', 'Error
    ']
622 # - MoveBasePoints
623
624 # main
625 def speak(s):
626     #absolute path for this
627     print 'Word: ' + s
628     #soundhandle.say(s,'voice_kal_diphone')
629     #may be loop with readline so there is a pause at every line
    .
630     rospy.sleep(3)
631
632 def tagDetected(data):
633     #print data.id
634     global currTask
635     if data.id in [0,1,2,3]:
636         currTask = data.id
637         if currTask is not data.id:
638             print 'Tag ID - Current Task Assign: ' + str(data.id
    )
639
640 def main():
641     rospy.init_node('asma')
642     global mp
643     mp = MoveBasePoints()
644     # Create a SMACH state machine
645     sm = smach.StateMachine(outcomes=['STOP'])
646
```

```
647     rospy.Subscriber("visualization_marker", Marker, tagDetected
        )
648
649
650     # Open the container
651     with sm:
652         # Add states to the container
653         smach.StateMachine.add('StartRobot', StartRobot(),
654                                transitions={'StartPatrol':'
        StartPatrol'})
655
656         smach.StateMachine.add('StartPatrol', StartPatrol(),
657                                transitions={'DetectTag':'
        DetectTag'})
658
659         smach.StateMachine.add('DetectTag', DetectTag(),
660                                transitions={'Task1':'Task1',
661                            'Task2':'Task2',
662                            'Task3':'Task3',
663                            'Task4':'Task4',
664                            'Error':'Error',
665                                'FailDetectTag':'FailDetectTag'})
666
667         smach.StateMachine.add('Error', Error(),
668                                transitions={'StartPatrol':'
        StartPatrol',
669                                                'STOP':'STOP'})
670
671         smach.StateMachine.add('FailDetectTag', FailDetectTag(),
672                                transitions={'TaskCompleted':'
        TaskCompleted',
673                            'Error':'Error'})
674
675         smach.StateMachine.add('TaskCompleted', TaskCompleted(),
676                                transitions={'Location1':'
        Location1',
677                            'Location2':'Location2',
678                            'Location3':'Location3',
679                            'Location4':'Location4',
680                            'Error':'Error'})
681
682         smach.StateMachine.add('TaskFailed', TaskFailed(),
683                                transitions={'TaskCompleted':'
        TaskCompleted',
684                            'Error':'Error'})
685
686         smach.StateMachine.add('Task1', Task1(),
687                                transitions={'TaskCompleted':'
        TaskCompleted',
688                            'TaskFailed':'TaskFailed',
689                            'Error':'Error'})
690
691         smach.StateMachine.add('Task2', Task2(),
692                                transitions={'TaskCompleted':'
        TaskCompleted',
693                            'TaskFailed':'TaskFailed',
694                            'Error':'Error'})
```

```
695
696         smach.StateMachine.add('Task3', Task3(),
697                                transitions={'TaskCompleted':'
    TaskCompleted',
698                         'TaskFailed':'TaskFailed',
699                         'Error':'Error'})
700
701         smach.StateMachine.add('Task4', Task4(),
702                                transitions={'TaskCompleted':'
    TaskCompleted',
703                         'TaskFailed':'TaskFailed',
704                         'Error':'Error'})
705
706         smach.StateMachine.add('LocationFailed', LocationFailed
    (),
707                                transitions={'Location1':'
    Location1',
708                         'Location2':'Location2',
709                         'Location3':'Location3',
710                         'Location4':'Location4',
711                         'Error':'Error'})
712
713         smach.StateMachine.add('Location1', Location1(),
714                                transitions={'DetectTag':'
    DetectTag',
715                         'LocationFailed':'LocationFailed',
716                         'Error':'Error'})
717
718         smach.StateMachine.add('Location2', Location2(),
719                                transitions={'DetectTag':'
    DetectTag',
720                         'LocationFailed':'LocationFailed',
721                         'Error':'Error'})
722
723         smach.StateMachine.add('Location3', Location3(),
724                                transitions={'DetectTag':'
    DetectTag',
725                         'LocationFailed':'LocationFailed',
726                         'Error':'Error'})
727
728         smach.StateMachine.add('Location4', Location4(),
729                                transitions={'DetectTag':'
    DetectTag',
730                         'LocationFailed':'LocationFailed',
731                         'Error':'Error'})
732
733
734
735
736         sis = smach_ros.IntrospectionServer('server_name', sm, '
    /SM_ROOT')
737         sis.start()
738
739         #rosrun smach_viewer smach_viewer.py to view the states
740     # Execute SMACH plan
741     outcome = sm.execute()
742     rospy.spin()
```

```
743         sis.stop()
744
745
746 if __name__ == '__main__':
747     main()
748
749 #print 'Welcome'
750 #welcome.wel('welcome.txt')
```

Bibliography

[1] Open Source Robotics Foundation. Ros groovy galapagos, . http://wiki.ros.org/groovy.

[2] Open Source Robotics Foundation. Turtlebot 2, . http://www.turtlebot.com/.

[3] Patrick Goebel. *ROS By Example*. Lulu, 2013.

[4] Brian Gerkey. Slam gmapping, . https://github.com/ros-perception/slam_gmapping.

[5] Tully Foote. Turtlebot navigation. http://wiki.ros.org/turtlebot_navigation.

[6] Brian P. Gerkey. Amcl, . http://wiki.ros.org/amcl.

[7] Eitan Marder-Eppstein. Move base. http://wiki.ros.org/move_base.

[8] Microsoft Robotics. Kinect sensor. http://msdn.microsoft.com/en-us/library/hh438998.aspx.

[9] Jonathan Bohren. Smach, . http://wiki.ros.org/smach.

[10] Jonathan Bohren. Smach viewer, . http://wiki.ros.org/smach_viewer.

[11] Eismann Oreilly. Qr codes scanning distance. https://qrworld.wordpress.com/2011/07/16/qr-codes-scanning-distance/.

[12] Scott Niekum. Ar track alvar. http://wiki.ros.org/ar_track_alvar.

[13] Jonathan Bohren. Joy, . http://wiki.ros.org/joy.

[14] Carnegie Mellon University. Cmu sphinx, . http://cmusphinx.sourceforge.net/.

[15] Carnegie Mellon University. Sphinx knowledge base tool, . http://www.speech.cs.cmu.edu/tools/lmtool-new.html.

[16] Austin Hendrix Blaise Gassend. Sound play. http://wiki.ros.org/sound_play.

[17] Thomas Moulard. Vision visp. http://wiki.ros.org/vision_visp.

[18] Fabien Spindler. Demo pioneeer. http://wiki.ros.org/demo_pioneer.

[19] ROS. Face recognition. http://wiki.ros.org/face_recognition.