UNIVERSITÉ DE BOURGOGNE

MSC COMPUTER VISION

# Robotics Project

*Authors:*
Sandeep MANANDHAR
Mohamed EISSA

*Supervisor:*
Ralph SEULIN
Cansen JIANG
Raphael DUVERNE

January 6, 2016

**Abstract**

In this report we present out work on computer vision and autonomous navigation using TurtleBot 2 with ROS-Hydro. The work done here relies on open-source packages such as: *OpenCV*, *rbx1*, *rbx2*, *rplidar* etc. from which we realize basic building blocks for our algorithms. We have used python language to develop the project.
The turtle bot uses LiDAR for mapping and AMCL approach to localize itself in a map. Kinect one is used for capturing visual and depth information. Computer vision tasks such as AR-tag detection and localization, face detection and recognition has been realized alongside using SMACH framework for concurrency. In addition to this, we present two ways to estimate the initial pose of the robot using visual information obtained from AR-tags.

# Contents

# List of Figures

# Chapter 1

# Introduction

ROS is a powerful framework for robot software development which provides hardware abstraction and low-level control interface between hardware and software. With many sensors and actuators that are compatible with it, many robot applications can be developed. In this project, we develop application for a TurtleBot 2 which can navigate autonomously in a given map accompanied by computer vision.
In this report, we discuss building a map in an indoor environment, localizing the robot with LiDAR, detecting and localizing AR-tags with Kinect as well as face detection and recognition. The task for the robot begins as visiting 4 locations in the map which are predefined in the software. We call these locations as *Primary* targets. While the robot navigates to these locations, the robot discovers many AR-tags around it. These AR-tags are registered with their location and orientation in the map and the robot adds new goal locations accordingly. We call these new locations as *Secondary* targets. Also during the whole operation time, robot detects faces around it. Due to the position of the Kinect sensor near the base, the viewing volume of the robot is near ground level. This might not be practical enough to detect human faces significantly above the ground level, but we have fixed images of faces in the wall of the map near ground level. In addition to this, we present our experiments and results on solving the kidnapped robot problem where we attempt to get the initial position estimate of the robot using visual information.
All of these tasks, i.e. navigation, AR-tag localization and face detection are executed concurrently using SMACH framework. The computer used in the robot itself is low end, so we do the processing job in high-end workstation via wireless connection.


Figure 1.1: TurtleBot 2

# Chapter 2

# Mapping and Navigation

## 2.1 Arena Description

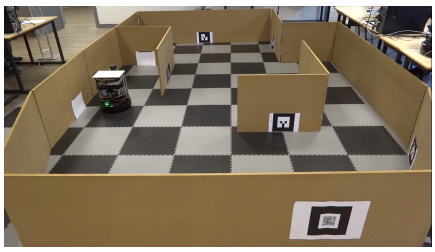

Figure 2.1: Arena: view 1

Figure 2.2: Arena: view 2

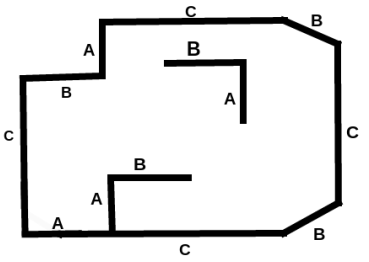


Figure 2.3: Arena scheme: Top view

The arena has walls made of cardboard planks of different lengths. The figure 2.3 describes the scheme of the map with length depicted as **A** for 81*cm*, **B** for 120*cm* and **C** for 244.5*cm* length. The walls are high enough for the LiDAR sensor mounted on top of the TurtleBot to get reading. The floor is concrete ground with carpet with roughness and raggedness usual for an indoor environment. The arena has AR-tags and pictures of faces stuck to the wall at different locations at lower height to obtain good visibility given the Kinect sensor mounted at mid level of the TurtleBot.

## 2.2 Mapping

To build a map, using LiDAR module, we have used *gmapping* package. The map is 2D occupancy grid map created from laser data and pose data of a robot. The *gmapping* package employs Rao-Blackwellised particle filters to learn grid maps. The particles carry individual map of the environment but are sampled to less number making it faster.
The robot needs to move around to get the reading of the whole map. We use joystick to manually move the robot around since we do not have any path planning prepared for now. To see the mapping in real time, we use RViz simulator. It is to be noted that, the origin of the map will be the initial pose of the robot during this phase.
During the mapping phase, robot should move slowly, keeping near distance from walls and taking

slow turn around the corner [put ros by example reference here]. During our experiment, we found that the more the loop closures, the more uncertainty the map has. We define loop closure as the process of moving robot around a region and coming back to the initial point again. One of the reasons for this is the drifting of the robot and accumulative error of odometry. Hence, the boundaries of the arena becomes distorted and thicker. We have swept the whole map in one loop and tried to mitigate the errors due to drifting. As we can see in the figure 2.4, the black borders

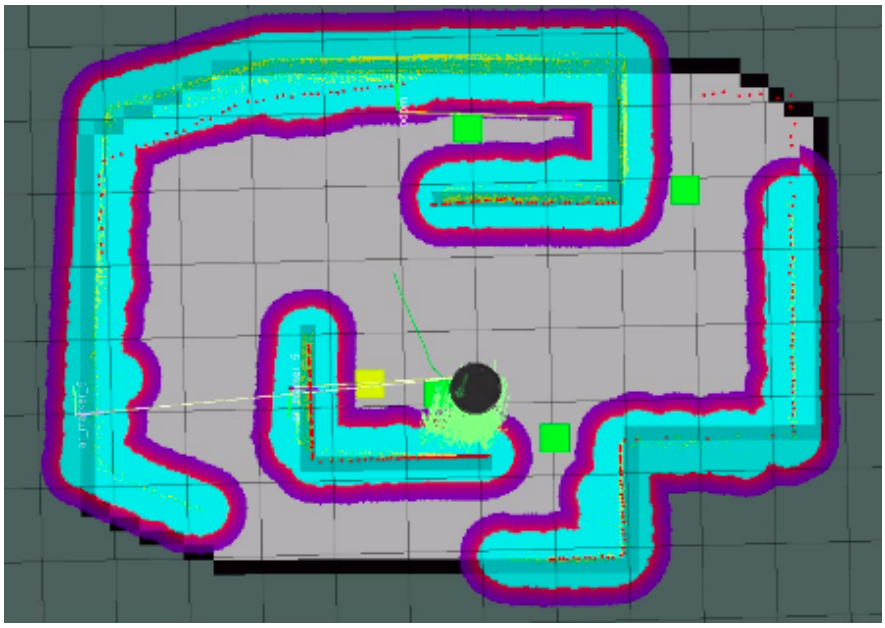

Figure 2.4: RViz display of map and robot

are the boundaries of the map that we created. The red dots near those boundaries are the samples from the LiDAR sensor. The cloud of cyan around the black borders is the uncertainty related to the boundaries.

## 2.3 Autonomous Navigation

Autonomous navigation in ROS is possible using two powerful features, *amcl* and *move_base*. These packages enable the TurtleBot navigating in a map from point *A* to point *B* avoiding the obstruction and localizing itself at the same time.

### 2.3.1 Move base

*move_base* node links a global and a local planner to accomplish its navigation task. It also maintains two costmaps for the planners. It's structure in high-level is shown in figure 2.5 The local planner uses odometry data as well as global planner data to maintain itself while outputting twist messages to *base_controller* to move the robot. The global planner maintains itself using $/map$ information, $/tf$ messages and various sensors. The transformation configuration between various sensors and components like actuators, camera-link, map, etc is stored in $/tf$ messages.

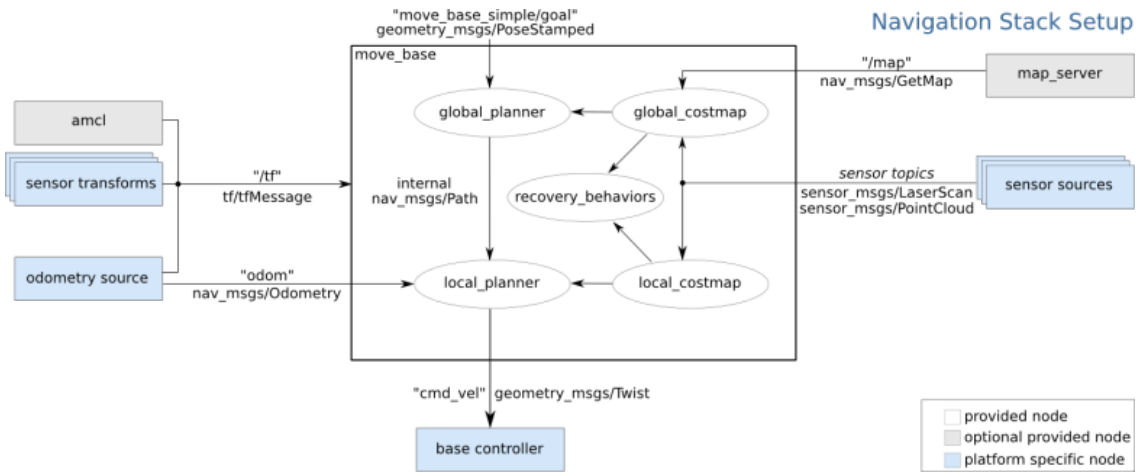


Figure 2.5: Configuration of Navigation stack(taken from ROS wiki)

### 2.3.2 AMCL

ROS implements a probabilistic localization system using adaptive Monte Carlo localization scheme. It transforms incoming scan like from LiDAR, to odometry frame. It estimates a $tf$ of base frame with respect to global frame but it only publishes the $tf$ between global frame and odometry frame which introduces drifting errors.

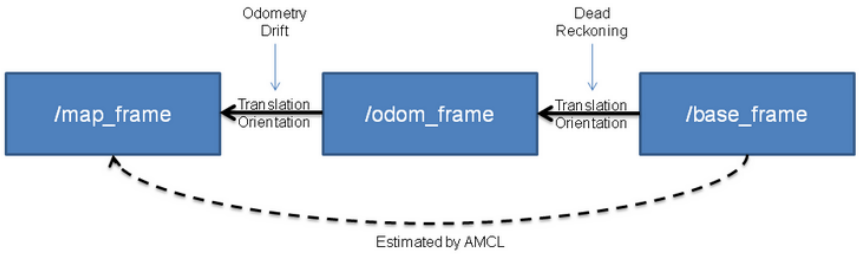


Figure 2.6: AMCL Localization(taken from ROS wiki)

# Chapter 3

# Tasks

## 3.1 Initial Pose Estimation

When a robot is activated in a map, it does not know its pose in the map. The AMCL package enables the robot to localize itself in the map, however, the initial position has to be set by the user. One way to do this, is to use RViz to manually give the initial pose estimate. But still, it is not automatic and depends upon user's knowledge of the map. So we have come up with two techniques that uses visual information acquired by the robot to find its position in the map.

### 3.1.1 Procrustes Analysis based method

We consider that the map is filled with landmarks. Each of these landmarks is known for its position and orientation after a calibration. When a robot is initialized, its local frame which is not necessarily aligned with the global map will register these landmarks in its own coordinate system. In figure 3.1, the map frame has three points in its true pose. These three points are taken as three different AR-tags whose pose can be found visually by the TurtleBot. There positions are known as they have been pre-calibrated. The local frame is rotated and translated giving 3DoFs. This is the frame that robot is initialized with and even though the 3 AR-tags are still visible, they are registered in local frame. The problem here is to find the rotation and translation between global frame and local frame. Since there are 3DoFs, we choose 3 non co-linear points to find the transformation. It is to be noted that, here we treat the robot as a point without considering its heading angle.

Consider the three blue points in map frame subtracted from their centroid $T1$. That brings their mean to origin. The centroid $T1$ essential to find the translation. These zero mean points are arranged in a $3 \times 3$ matrix $A$ considering all for them having $z - coordinate$ of 1. We make matrix $B$ for three corresponding points from local frame(zero mean points). Let $T2$ be their centroid. Till now, we know these points in $A$ from calibration and $B$ from the visual information collected by Kinect. The visual information is encoded in *ar_pose_marker* message type defined from *ar_track_alvar* package.

We then create the covariance matrix between these points.

$$C_{AB} = AB' \tag{3.1}$$

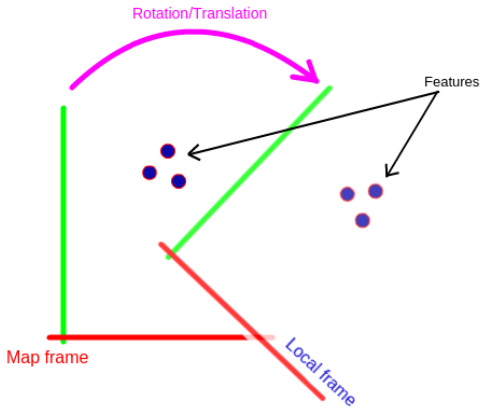


Figure 3.1: Global map in Red
Local map with unknown rotation and translation in blue

Figure 3.2: viewing 3 AR-tags

Then compute the singular value decomposition of $C_{AB}$.

$$USV' = svd(C_{AB}) \tag{3.2}$$

where $U$ and $V$ are orthogonal vectors and $S$ is a diagonal matrix. Then we find a rotation matrix using these set of orthogonal vectors that bring $B$ to $A$. It is given as:

$$R = inv(UV') = VU' \tag{3.3}$$

And the translation is given as:

$$t = -R \times T1 + T2 \tag{3.4}$$

So we have a necessary transformation that brings local frame to global frame. Given that we have the robot's position $P$ in local frame, we can now find the true global position $P_{true}$ on robot as:

$$P_{true} = R * P + t \tag{3.5}$$

But in this method, we assume the robot is a point and no heading angle is considered. Still the heading angle of the robot is not know so there is one more degree of freedom left as unknown. To test the algorithm, we assumed that robot always knows where it should be heading to in order to see the set of given three points. In conclusion, this method only computes position $(x, y, z)$ but leaves out the *yaw* angle of the robot.

### 3.1.2 TFs based method

In this method, instead of relying on three points, we use one AR-tag with known pose to solve the initial pose of the robot. Once again, the AR-tag pose is pre-calibrated so we have the transformation between AR-tag and the original map. The transformation between AR-tag and the robot baselink or cameralink is also known during the run time. This method exploits one of the powerful

features of ROS, i.e TF. By listening to the $tf$ lookup, we find the these transformations whenever the required AR-tag is detected in the scene. As shown in the figure 3.4, the final transformation is found by multiplying these transforms and inverting them. These transformations can be in Euler form or Quaternion form. This transformation can be used to multiply with robot's current pose in local frame and get the true pose in global frame.
One of the issue here is finding the robot's current pose. The *amcl_pose* message is published only when robot is moving. But in this case, where robot is stationary, the message is not useful. To find the robot's pose, we use the transformation from $/odom$ to $/map$. When the transformation is applied to the current frame's robot pose, there may occur a slight value in $z-coordinate$ even though the robot has 3DoFs. It may be the noise caused by the continuously rotating and vibrating LiDAR sensor which is picked up by the IMU sensor.

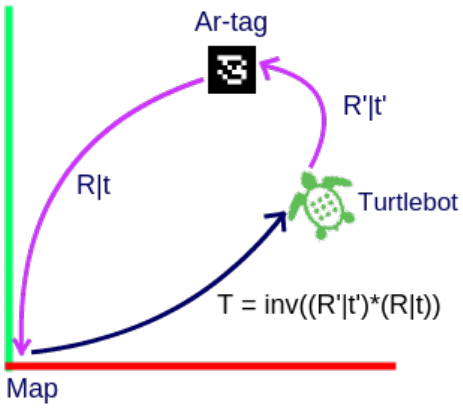


Figure 3.3: Transformation between AR-tag to robot and Map in magenta, map to robot in

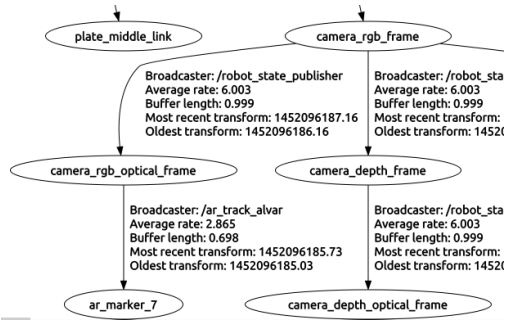


Figure 3.4: AR-tag transformation in TF lookup tree

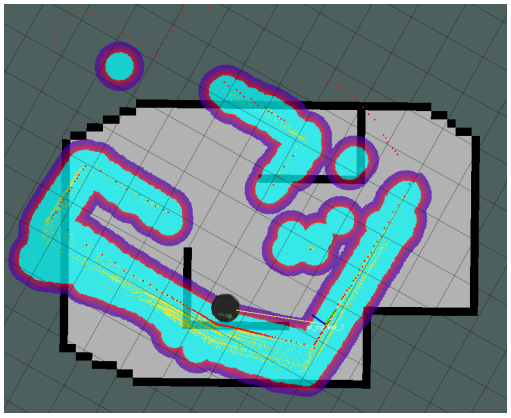

Figure 3.5: ientation of map on initialization

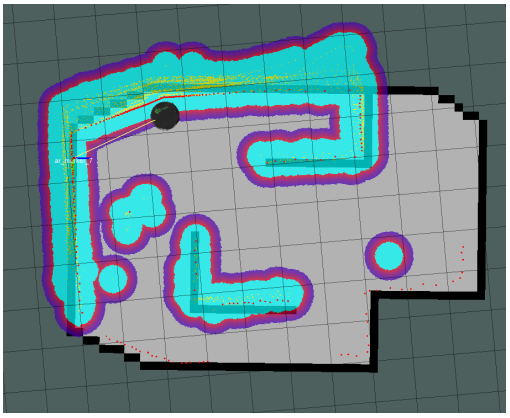

Figure 3.6: Orientation after Auto-pose estimation via TFs

### 3.1.3 Publishing Initial Pose

Once the robot pose is calculated with reference to the global frame, we would like to publish the pose to proper node so that the robot is set to proper location in the map. The pose information should be written to *PoseWithCovarianceStamped* message type and dispatched to *initialpose* node. This message type requires position, orientation in quaternions, the reference frame and co-variance matrix. The co-variance matrix is a $6 \times 6$ matrix that stores the uncertainty in robot pose. The first 3 elements are the position coordinates $(x, y, z)$ and the last 3 despite of quaternions having 4 elements are the yaw, pitch and roll. The reason for encoding covariance matrix not with 4 quaternion values but *RPY* is that it reduces the chances of error propagation from higher dimension.

## 3.2 Navigation

The robot is ordained to visit 4 *Primary* locations in the map. While it commutes to these locations, it registers new *Secondary* locations based on new detected AR-Tag positions. Navigation is implemented using *actionlib* package, which provides a Client-Server model to do tasks in a preemptive manner. Task progress is periodically monitored and can be can be canceled if needed. For a navigation task, ROS provides a *MoveBaseAction* class that can be used to initialize a *SimpleActionClient* and *SimpleActionServer*, and also provide a *MoveBaseGoal* class as the main message type between client and server to state the target goal position. It wraps a Pose class object that defines the 3D location of the target and orientation using a quaternion. Hence, all our primary and secondary positions are declared as a list of the *Pose* class type.

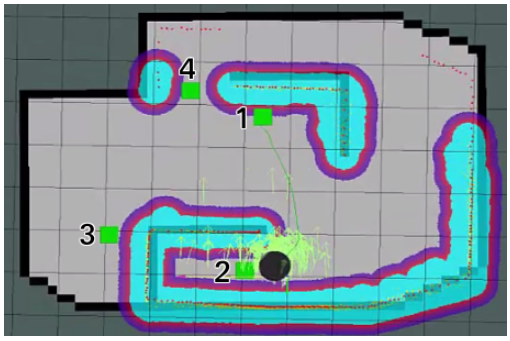


Figure 3.7: 4 primary locations in green tiles and numbered in order of visiting sequence

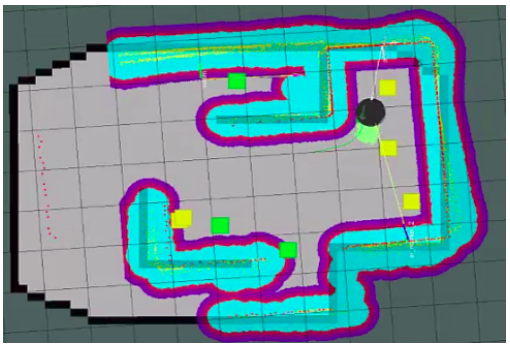

Figure 3.8: Secondary locations in yellow tiles

## 3.3 AR-Tag Localization

An AR-tag, also called augmented reality tag is a bitonal fiduciary marker that can be used in a computer vision system to identify, track and define a plane geometry. The tag, as shown in figure 3.10 is made up of bunch of black and white squares making it easier to detect as it has distinct feature points. Each of the tags has its own identification defined by its geometric feature.

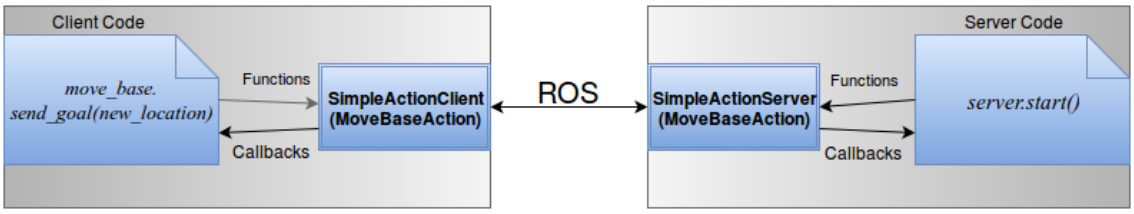


Figure 3.9: *SimpleActionClient* and *SimpleActionServer* for *MoveBaseAction*

The tag is a plane consisting $9 \times 9$ grid of black and white squares. The first 2 units from the borders are black pixels. The remaining 25 units are the internal pattern that will encode the identification of the marker. The black borders help detecting the tag in varying illumination condition. Edge detection approach is used to localise the borders and corner points in projected image. These corner points are then used to compute a homography to sample a $5 \times 5$ grid to decode 25-bit word, and identify the tag and rotation among 4 possible ones.

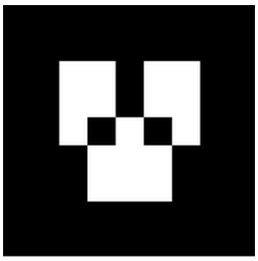

Figure 3.10: AR-tag

We have used $ar\_track\_alvar$ package with Kinect sensor to detect the position and orientation of the tag. When a tag is detected, its position and orientation is calculated. The pose is then published in $/ar\_pose\_marker$. The pose can be in reference of $/camkeralink$, $/baselink$ or the $/map$ if the map server is running. We use $/map$ reference since we need to localize a tag in a map. Then we compute the goal position and orientation of the robot accordingly and set it as a secondary goal.
To do this, first we use the orientation information. If the tag is facing along x-axis, the goal point will be tag's position with some offset in x-direction and orientation of the robot would be against x-axis to make it face the tag. The orientation in quaternion is converted to euler angles to have easy comparison with map's axis orientation. This is not a visual servoing in strict sense since it has no control loop but it can localise itself given the tags oriented along or against x/y-axes.
The AR-tag when once detected is added to the secondary list. Since the kinect sensor sends the video stream of the scene, the AR-tag is being continuously detected. To prevent it from adding multiple goal points of same location, we have to put a sanity check system. It can be done in a manner such that only unique AR-tags are registered for goal points. This would limit us from using same AR-tag in different location. Another way is to check the euclidean distance of the AR-tag from previously found ones, every time it is detected and put threshold on the difference. If the scene is noisy, the distance check might not be efficient. We have decided to use the former method.

## 3.4 Face Detection

Face Detection: Our second vision task is to detect faces using the mounted Kinect camera. We are using OpenCV's built in face detector, a module that uses cascade of classifiers with Haar-like features. Each classifier is trained with a few hundred features of different views of the target object (i.e., a human face). The algorithm is based on Viola-Jones algorithm.
For our face detection module we use training data provided by OpenCV itself, which provides different types of Haar features for multiple targets like human full body, upper body, face front and side view, all features are stored in XML file format. We chose to use 3 classifiers for profile face as well as frontal face with glasses as well.

```
self.cascade1 = cv2.CascadeClassifier('haarcascade_frontalface_alt.xml')
self.cascade2 = cv2.CascadeClassifier('haarcascade_frontalface_alt2.xml')
self.cascade3 = cv2.CascadeClassifier('haarcascade_profileface.xml')
self.haar_params = dict(scaleFactor = 1.3,minNeighbors = 3,
      flags = cv.CV_HAAR_DO_CANNY_PRUNING,
      minSize = (30, 30),maxSize = (150, 150))
```

Kinect camera is used to capture images in real time while the robot is patrolling, ROS package *uvc_camera* was used to stream the images. This package provides drivers for USB Video Class (UVC) cameras which covers most of the consumer web cameras. It publishes two types of main topics; *image_raw* which is a root topic for all kind of images that could be streamed by different types of camera (i.e., RGB, Depth, ... etc), $camera\_info$ which present camera intrinsic parameters.

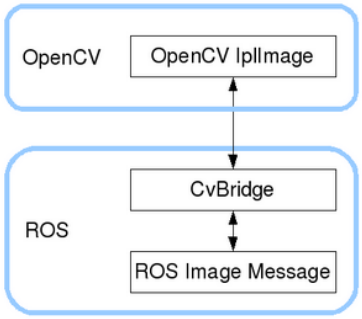


Figure 3.11: A *CvBridge* object works as an adapter between ROS *Image* message type and *OpenCV* Image type.

In our implementation, we subscribe to the topic $/camera/rgb/image\_color$ with the message type of *Image*. A *CvBridge* type object is used as an adapter to convert from ROS *Image* message type to *OpenCV* Image type. Then the *OpenCV* image is converted to grayscale. Histogram Equalization is applied for better range of intensities, before dispatching the output image to each classifier to check for the presence of any faces.

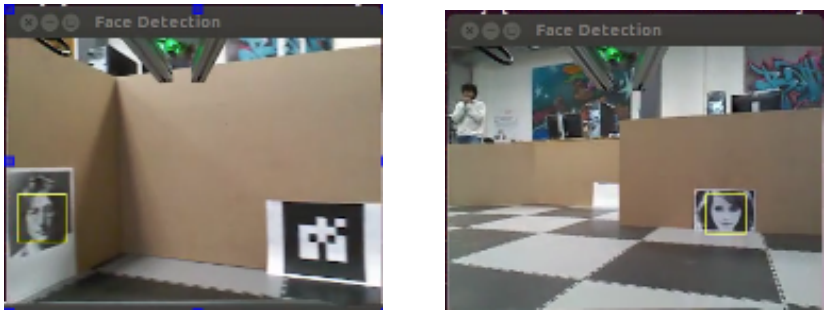


Figure 3.12: Face Detection example using printed image of human faces, detected faces is highlighted in yellow box

## 3.5 Face Recognition

As an addition to the face detection task, we use *face_recognition* ROS package to add the ability of Face Recognition. The package uses the Eigenface method that is based on the Principal Component Analysis (PCA). The algorithm works as follows:

1. Collect a stream of images of size $N$ and crop them so that only eyes and chin are included, but not much else.
2. Convert each image of size $r \times c$ to a column vector of length $r \times c$.
3. Merge all vectors to a one big matrix of size $rc \times N$
4. Calculate the covariance matrix $L$ of size $N \times N$ by multiplying the all vectors matrix by its transpose, hence $L$ will be a symmetric and contains only positive values.
5. Compute the eigenvectors of L, this will produce $N-1$ vector of length $N$
6. Multiply each eigenvector element to the corresponding image, to construct the eigenface, which is a base element in the face space
7. Sort eigenfaces according to the eigenvalues, keep an $M$ eigenface to be used for testing purposes

Now for every new image, if a face exists, it is extracted and transformed to eigen space. Inner product against all previous defined eigenfaces is computed, and if the output is above some threshold, the face is classified as the trained person.

Package *face_recognition* was developed in $C++$ using ROS *actionlib* package, hence we have a ROS node named *FServer* that implements the *SimpleActionServer* interface to provide all face recognition functions which are defined using two arguments:

1. int *order_id*
2. string *order_argument*

They could be combined as follows:

1. $order\_id = 0$ to recognize a face in a video stream only once
2. $order\_id = 1$ to continuously recognize faces in a video stream
3. $order\_id = 2$ and $order\_argument =' person\_name'$ to capture training images for a new person named with the given argument
4. $order\_id = 3$ to train the list of eigenfaces using a list of images that are defined in a configuration text file *train.txt*
5. $order\_id = 4$ to shutdown the server.

The Package *face_recognition* provides another ROS node named *FClient* that implements *SimpleActionClient* interface. However, it was written in $C++$, and since we chose *Python* as our main programming language for this project, we have to define a *SimpleActionClient* interface implementation in *Python*

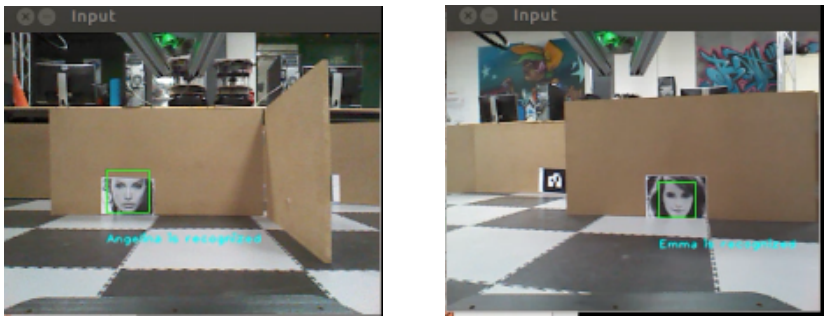


Figure 3.13: Faces Recognized during the runtime

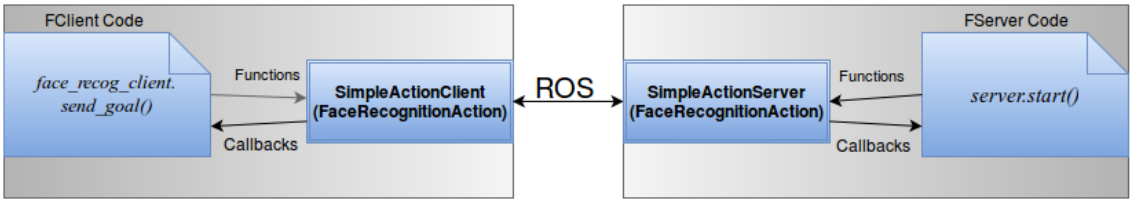


Figure 3.14: *SimpleActionClient* and *SimpleActionServer* for *FaceRecognitionAction*, Client is written in *Python* to be used in our project. Server is written in $C++$ and used directly as a standalone node in our project

# Chapter 4

# Task Execution

In a fully autonomous robot that has a large number of actions to select from based on the task in hand and its current status and conditions, a task organization and modularity is strongly required. ROS provides multiple frameworks to manage complex tasks in a more structured approach. The most two commons frameworks are *SMACH* and *pi_trees*, *SMACH* follows finite state machine model of computation, *pi_trees* adapts the behavior trees paradigms that is common computer games programmers. both frameworks provides a minimum set of key features for task controlling and execution like:

1. Task priorities: Tasks should have a priority attribute that gives the ability for higher priority tasks to preempt lower priority task (e.g., preempt a patrolling task if battery checking status is lower than a give threshold)

2. Pause and Resume: Tasks should have the ability to be paused and resumed (e.g., yield back control for a patrolling task if recharging task is complete)

3. Task Hierarchy: Tasks could be broken into sub-tasks (e.g., recharging could be made up of 3 sub-tasks; navigating to dock station, dock the robot, charge the robot)

4. Conditions: The framework should be able to manipulate data by all mounted sensors, so the robot behavior would differ based on its interaction with published messages from various nodes.

5. Concurrency: Multiple tasks could be executed in parallel, like in our case, the robot is able to detect and recognize faces while navigating between different nodes.

## 4.1 SMACH

For our project we choose SMACH framework for task execution, SMACH provides all previous features with a heavy using of mutlithreading in C++ and Python to provide concurrency in a robust manner. In SMACH, each state is represented as an object instantiation of *smach.State* that must override $execute()$ method to return one or more possible *outcomes*, this method also has an optional parameter *userdata* that can be used to pass data between states. *SMACH* also provides a number of defined *State* subclasses for different applications like

1. *SimpleActionState* class that can be used to represent a ROS *Action* to *SMACH* state.
2. *MonitorState* class that wraps a callback for ROS topic subscription.
3. *ServiceState* class that handles ROS services.
4. *ConditionState* class that checks a condition function a number of times.

*SMACH* also provides a number of predefined states container:

1. *Concurrence* container to have multiple states running in parallel allowing each of them to preempt the other.
2. *Sequence* container that automatically generate sequential transitions between its in states.
3. *Iterator* container that iterate over a number of states until some condition is met.

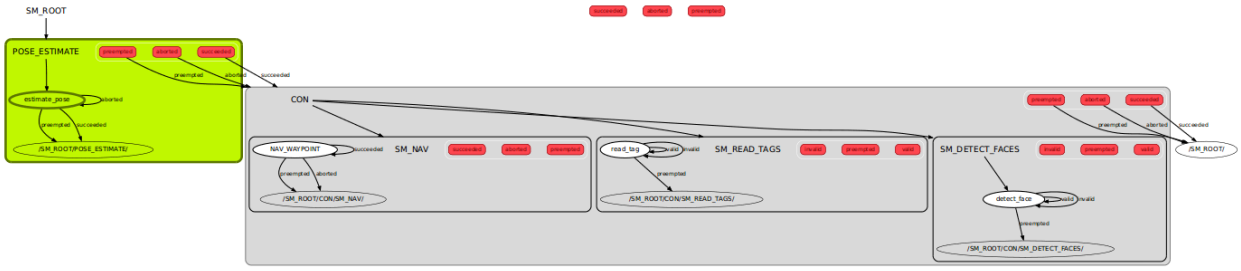


Figure 4.1: Pose Estimate state

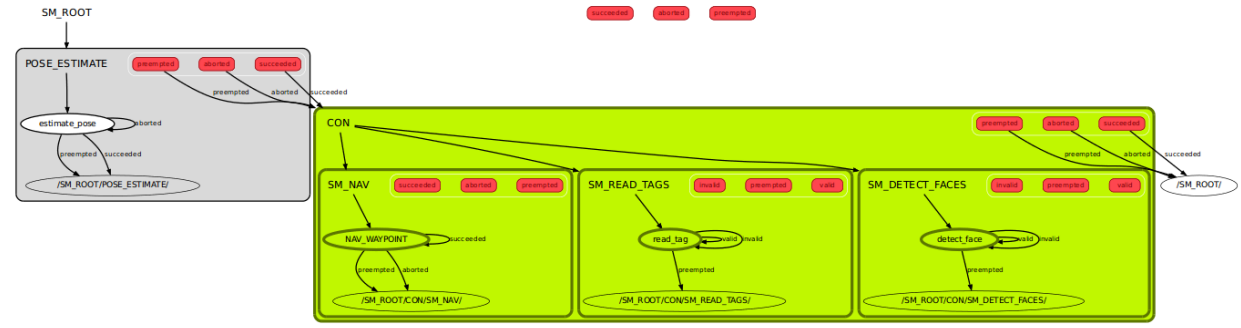


Figure 4.2: Task completion state with 3 concurrent states

Our state machine as described in figure 4.1 and figure 4.2, starts with initial pose estimation state machine *POSE_ESTIMATE*, once it is successful to calibrate the robot position in the map, the concurrence container labeled as *CON* starts executing. It consists of the main state machines of the TurtleBot that runs in parallel; Navigation, AR-tags identification and tracking, and face detection and recognition. We will discuss each as follows:

### 4.1.1 Navigation State Machine

For the navigation task, we have a design option of using *smach_ros.SimpleActionState* class, but since we have a dynamic goal position, we need a new state machine that can add new locations to its list. So, we define a new *smach.State* subclass *NavState* which has access to the list of Poses (primary and secondary) as *userdata*. It picks up the first *Pose* of the list, send it as a *MoveBaseGoal* goal to the *MoveBaseAction* object and remove the location from the list once goal status is succeeded. The list is getting updated by AR-Tag state machine when it finds new AR-tag. The Navigation state machine aborts when the list is empty, preempting all other running states and also aborts the concurrence container.

```
class NavState(State):
  def execute(self, userdata):
    if len(userdata.waypoints) == 0:
      return 'aborted'

    self.goal.target_pose.pose = userdata.waypoints[0]
    self.move_base.send_goal(self.goal)

    state = self.move_base.get_state()
    if state == GoalStatus.SUCCEEDED:
      userdata.waypoints.remove(userdata.waypoints[0])

    return 'succeeded'
```

### 4.1.2 AR-Tag State Machine

For detection and tracking of AR-Tags, a *smach_ros.MonitorState* could be an option. In our case we need to subscribe to topic */ar_pose_marker* with a callback function to update the list of *Pose* objects if it finds a new tag, the list could be passed to the state as *userdata* object, but we found the class leaks a very important feature of having an output, so we have to define a new *SMACH MonitorState* that is able to update a *userdata* object by overriding the existing *smach_ros.MonitorState*.

```
class ReadTagState(MonitorState):
    def __init__(self, topic, msg_type, max_checks=-1):
        smach.State.__init__(self,
          outcomes=['valid', 'invalid', 'preempted'],
          input_keys=['waypoints'], output_keys=['waypoints'])

        self._topic = 'ar_pose_marker'
        self._msg_type = AlvarMarkers
        self.task = 'read_tag'

    def registerAR_cb(self, userdata, msg):
      if new_tag_found:
          userdata.waypoints.append(new_tag_pose)
```

### 4.1.3 Face Detection State Machine

We implement the Face Detection the same way we implement the AR-Tag state machine by overriding *smach_ros.MonitorState*

```
class FaceDetectState(MonitorState):
    def __init__(self, topic, msg_type, max_checks=-1):
        smach.State.__init__(self,
          outcomes=['valid', 'invalid', 'preempted'])

        self._topic = '/camera/rgb/image_color'
        self._msg_type = Image
        self.task = 'Face Detection'

    def execute_cb(self, userdata, msg):
      if new_face_found:
          draw_rectangle(current_frame, face_box)
          imshow(current_frame)
```

### 4.1.4 Face Recognition State Machine

For the Face Recognition, we have to implement *SimpleActionClient* interface to be able to communicate with the *FServer* node from *face_recognition* package. One design option is to use *SimpleActionState* which is provided by *SMACH* API to wrap a ROS *Action* inside a state. However, we preferred to implement our own state class, firstly for more modularity to keep every task in its own defined class, secondly to gave us a chance to dive into *SMACH* source code to understand how does it work under the hood with heavily use of Python threads.

```
class FaceRecognitionState(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded','aborted','preempted'])

        self._action_name = "face_recognition"
        self.face_recog_client = actionlib.SimpleActionClient(
              self._action_name, FaceRecognitionAction)
        self._done_cond = threading.Condition()

    def execute(self, userdata):
        self.goal = FaceRecognitionGoal()
        self._done_cond.acquire()
        self.face_recog_client.send_goal(self.goal)
    # Wait for action to finish
        self._done_cond.wait()
        self._done_cond.release()

        if self._goal_status == GoalStatus.PREEMPTED:
            self.service_preempt()
            return 'preempted'

        return 'succeeded'
```

# Chapter 5

# Utilities

## 5.1 RViz utilities

RViz is the ROS main visualization utility. It gives as a lot of features for monitoring and controlling the robot movement like pose estimation and navigation to a goal pose. It also gives us the ability to add visualization markers for primary position and secondary positions once found.

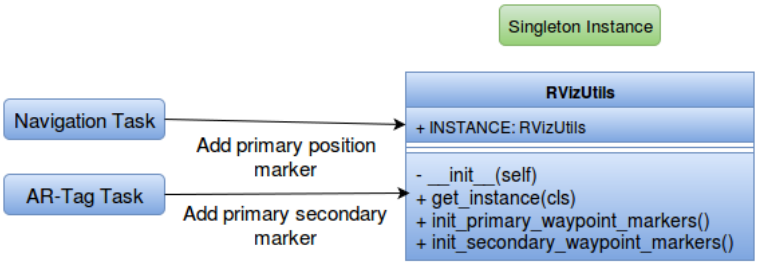


Figure 5.1: *RVizUtils* singleton class instance to be used to multiple clients

We define an *RVizUtils* class to mark both types during robot movement. It is defined following a Singleton design pattern to allow using of one instance per all client requests. It has two main methods; $init\_primary\_waypoint\_markers()$ to be used to add markers for primary positions, $init\_secondary\_waypoint_marker s()$ to be used to add markers for secondary positions, this method is called by AR-Tag task once a new tag is recognized in the map. Both methods publish a message of type *Marker* to topic $/waypoint\_markers$

```
class RVizUtils():
  INSTANCE = None

  @classmethod
  def get_instance(cls):
    if cls.INSTANCE is None:
      cls.INSTANCE = RVizUtils()
    return cls.INSTANCE

  def __init__(self):
    if self.INSTANCE is not None:
      raise ValueError("An instantiation already exists!")

    self.primary_marker_pub = rospy.Publisher('waypoint_markers', Marker)

    self.secondary_marker_pub = rospy.Publisher('waypoint_markers', Marker)

  def init_primary_waypoint_markers(self,primary_waypoints):
    for waypoint in primary_waypoints:
      self.primary_marker_pub.publish(waypoint)

  def init_secondary_waypoint_markers(self,secondary_waypoint):
    self.secondary_marker_pub.publish(secondary_waypoint)
```

## 5.2 Sound Messages utilities

Another way of receiving a feedback from our robot is using sound messages, so we use third party ROS *sound_play* package. It provides a node that convert ROS messages on *robotsound* topic into sound. The packages supports 3 types of sounds messages; build-in sounds, OGG/WAV file types, and speech synthesis using *festival*, a speech synthesis system that offers a full text to speech. We use the system to receive messages from the robot about new AR-Tag detection and for Face detection and recognition tasks. In a general scenario, the robot dispatches many sound messages in short period of time. So it is expected to have an overlap between sound messages which results in consistency when trying to listen to the messages. A proposed solution for this is to use a queue of messages, so the messages are synthesized one by one without overlap. Unfortunately, this solution is not enough since the package has another explicit drawback of not giving a clear way to tell when the sound message stopped playing.

To solve the two issues together, we implement a multi-threaded message queue system as in figure 5.2. The system has 2 main data structure to store the message to decide to play them or not and when to play them.

1. A Key-Value pair data structure (*Dictionary* in Python) is used to store the message with the time stamp of last played instance. This used for the scenario where robot has recognized the

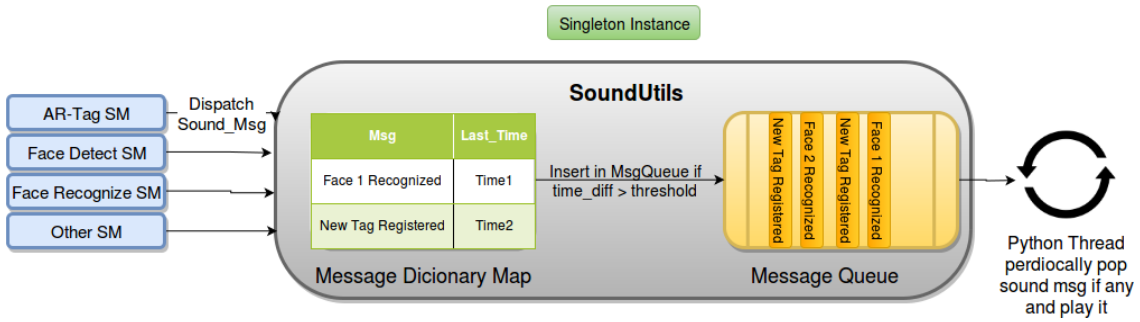


Figure 5.2: *SoundUtils* singleton class instance to be used to multiple clients. It wraps a message queue for sound messages served by standalone thread that keep checking the queue every two seconds

same face for some time. It will not play the message for multiple times when a particular face is recognized. Instead it will check when was this last message played, and according to some threshold it will decide if it will play this message or drop it.

2. If it is decided that the message is going to be played, then it will be inserted to the next data structure; a message queue (*Queue* in Python). A background thread is running every time period to serve the message in the queue front by popping and playing it. For simplicity of our application, we propose that any message duration will not exceed 2 seconds (e.g., "Face is recognized", "New tag is detected"), so we set the playing thread sleeping time to 2 seconds.

```
class SoundUtils():
  INSTANCE = None

  @classmethod
  def get_instance(cls):
    if cls.INSTANCE is None:
      cls.INSTANCE = SoundUtils()
    return cls.INSTANCE

  def __init__(self):
    if self.INSTANCE is not None:
      raise ValueError("An instantiation already exists!")

    self.last_time_said = rospy.Time(0)
    self.time_sleep_in_sec = rospy.Time(2).to_sec()
    self.time_delay_in_sec = rospy.Time(30).to_sec()
    self.message_queue = Queue()
    self.thread =  Thread(target=self.observe_queue)
    self.thread.start()

  def observe_queue(self):
    while True:
    if not self.message_queue.empty():
    self.soundhandle.say(self.message_queue.get(), self.voice, self.volume)
    rospy.sleep(self.time_sleep_in_sec)
    pass

  def say_message(self,message):
    now = rospy.Time().now()
    if message in self.previous_messages:
      last_time_said = self.previous_messages[message]
      diff_in_sec = (now - last_time_said).to_sec()
      if diff_in_sec < self.time_delay_in_sec:
            return

    self.previous_messages[message] = now
    self.message_queue.put(message)
```

# Chapter 6

# Discussion

## 6.1 Challenges

During our project development we faced some challenges that could be listed as below:

1. While we were trying to integrate all tasks together, the TurtleBot used to shutdown many times during task execution probably because of heating issue.

2. For most of ROS main packages, the documentation was quiet clear with the examples. However, this was not the same case for the most of the third party packages, like *SMACH*, *sound_play* and *Face_Recognition*, the official documentation of these packages neither clear enough nor providing enough examples to follow. Thanks to the two volumes of *ROS By Examples* reference and their source code repository, it made this issue less difficult.

3. Package *uvc_camera* that we used in most vision examples following *ROS By Examples* is deprecated, not supported by new ROS versions, and has been replaced by package *libuvc_camera*

4. *smach_ros.MonitorState* had a drawback of disability to update *userdata* input, we fixed this by defining our own *MonitorState* as discussed in section 3.4.

5. Package *sound_play* has two critical drawback as discussed in section 5.2, we fixed this by implementing message queue system for it.

## 6.2 Conclusion and Future Work

We have integrated computer vision tasks and navigation of a robot in an indoor environment using a ROS driven turtlebot. Sophisticated tasks like AR-tag detection and localization, face detection, AMCl works efficiently in concurrence using SMACH implementation. The ease to implement comes from the already available ROS packages as well as higher level understanding of hardware and mechanical design.The project can be extended for full fetched visual servoing using AR-tags.

# Bibliography


[1] Patrick Goebel. ROS By Example Volume 1

[2] Patrick Goebel. ROS By Example Volume 2

[3] ROS By Example Volume 1 Source Code url: https://github.com/pirobot/rbx1

[4] ROS By Example Volume 2 Source Code url: https://github.com/pirobot/rbx2

[5] TurtleBot website url: http://www.turtlebot.com/

[6] ROS Hydro documentation url: http://wiki.ros.org/hydro

[7] *actionlib* url: http://wiki.ros.org/actionlib

[8] *cv_bridge* url: http://wiki.ros.org/cv_bridge

[9] *face_detector* url: http://wiki.ros.org/face_detector

[10] Face Detection using AdaBoost classifier url: http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html

[11] *face_recognition* url: http://wiki.ros.org/face_recognition

[12] Face Recognition using PCA url: https://www.clear.rice.edu/elec301/Projects99/faces/algo.html

[13] *move_base* url: http://wiki.ros.org/move_base

[14] *RViz* url: http://wiki.ros.org/rviz

[15] *SMACH* url: http://wiki.ros.org/smach

[16] *smach* source code url: http://docs.ros.org/fuerte/api/smach/html/python/

[17] *smach_ros* source code url: http://docs.ros.org/fuerte/api/smach_ros/html/python/

[18] How to override *MonitorState* with output keys from ROS Answers url: http://answers.ros.org/question/28100/smach_rosmonitorstate-callback-receives-empty-userdata/

[19] *sound_play* url: http://wiki.ros.org/sound_play

[20] *tf* url: http://wiki.ros.org/tf

[21] *uvc_camera* url: http://wiki.ros.org/uvc_camera