UNIVERSITÉ DE BOURGOGNE

BACHELOR IN COMPUTER VISION AND ROBOTICS

PROJECT REPORT-ROBOTICS ENGINEERING 2

GROUP MEMBERS
Romain CORSIN
Flavien DAVID
Fahad KHALID

CO-ORDINATOR
Ralph SEULIN
Raphael DUVERNE
Cansen JIANG

Table of Contents

Introduction

Robot is a programmable mechanical device that can perform tasks and interact with its surroundings without any external/human interaction. The design, manufacturing and application of robots is known to us as the science and technology of robotics.

For the robots to be able to understand and perform tasks a method of communication and message transformation software needs to come into play. There are many platforms like URBI, CARMEN, Microsoft Robotics Studio, Yaarp, ROS and many more, among these platforms very famous one is ROS which we have used to control Turtlebot.

Chapter 1

What is ROS?

 Robot Operating System (ROS) is an open-source collection of software frameworks for robot software development, providing operating system-like functionality on a heterogeneous computer cluster.

 It Enable researchers to rapidly develop new robotic systems without having to “reinvent the wheel” through use of standard tools and interfaces.

 It provides low-level device control, message-passing between processes and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages. Despite the importance of reactivity and low latency in robot control, ROS, itself, is not a Real-time OS, though it is possible to integrate ROS with real-time code.

 A language-independent architecture (C++, python, lisp, java, and more)

The ros interface uses nodes topics and messages to subscribe a message we will look into this by a simple example. A node publishes a message to the topic the topic communicates the message to another node2 which reads the message, hence we can say the topics are simple carrier of the nodes, this s how a message is communicated. To have a clear view of how a node passes a message through the topic we have a graphical representation.

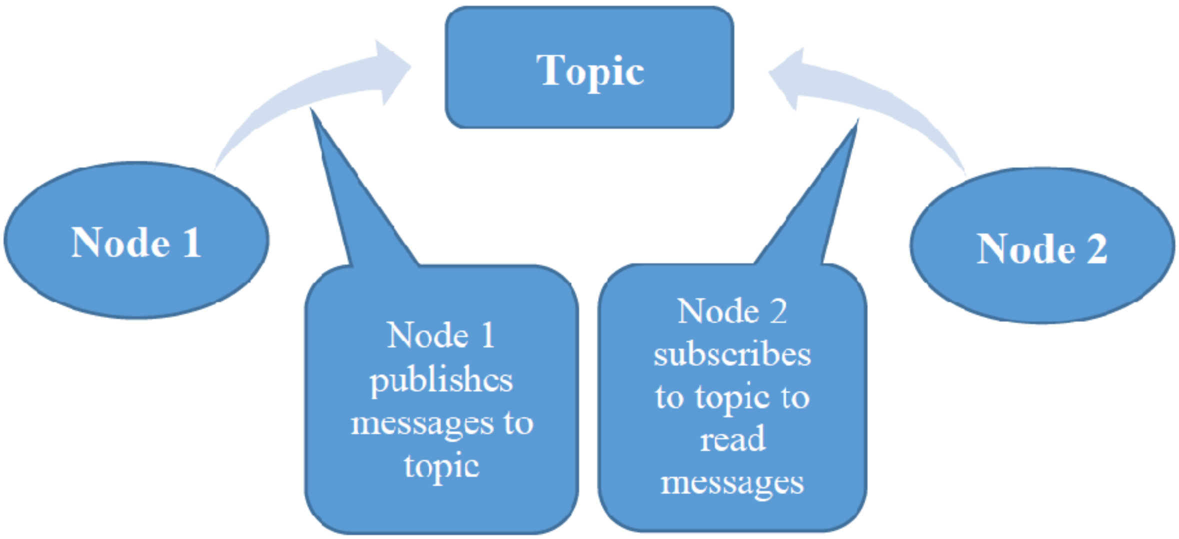


Figure: Nodes and Topics

Turtlebot2;

We used turtlebot2 for this project, the Turtlebot2 is low-cost, personal robot kit with an open source software, and we are able to program the turtlebot2 to be able to move on its own around the environment. The turtlebot2 is extremely user friendly robot which can be given instructions and execute them very easily. The features of the turtlebot2 include avoiding obstacles, Motion control, automatic navigation and localization etc.

Chapter 2

2.1 Motion Control

The most important part to know is motion control. The turtlebot should not be operated to move around until the user is not familiar with motion control. ROS uses right hand convention for orientating the co-ordinate axis with x-axis points forward, y-axis points to the left and z-axis points upward. The metric system is also used by ROS so that linear velocities are always specified in meters per seconds (m/s) and angular velocities are given in radians per second (rad/s). A linear velocity of 0.5 m/s is actually quite fast for an indoor robot so it should be kept below 0.2 m/s maximum and for rotation it should also be adjusted accordingly as required.

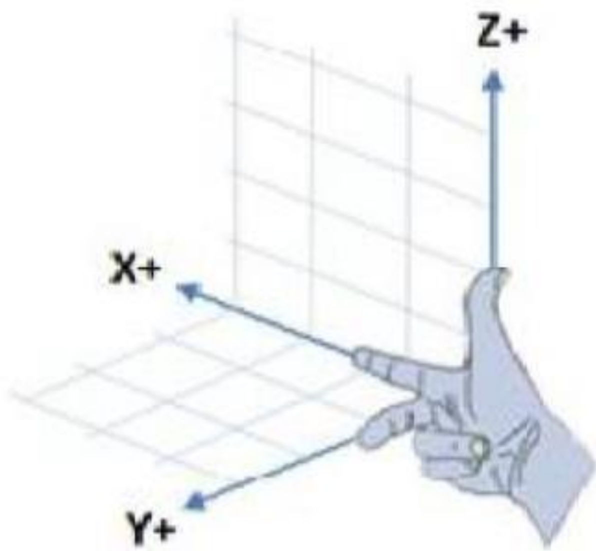


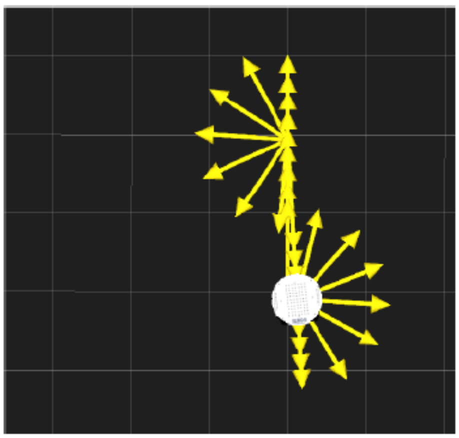
Figure right hand convention.

ROS uses twist messages for publishing motion commands to the topic /cmd_vel which is been subscribed by base controller node to translate those twists messages into motor signals to turn the wheels of robot. ROS twist message contains further two sub-messages such as geometry_msgs/Vector3 linear and geometry_msgs/Vector3 angular with x, y, z axis whose values are of type float. These twist message basically contains information about linear velocity and angular velocity in x, y, and z axis of Turtlebot2. Since turtlebot2 doesn't have Omni-direction wheel. So, we can only use x axis for linear velocity and z for angular velocity because Turtlebot is been run on the ground and to see the components of the twist message we can use a simple command in command window in Linux as rosmsg show geometry_msgs/Twist.

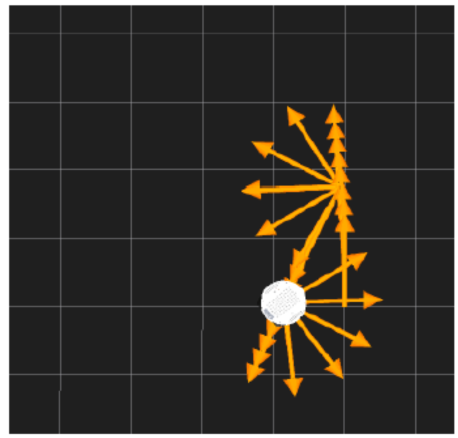
2.2 Twist

ROS uses twist messages for publishing motion commands to the topic /cmd_vel which is been subscribed by base controller node to translate those twists messages into motor signals to turn the wheels of robot. ROS twist message contains further two sub-messages such as geometry_msgs/Vector3 linear and geometry_msgs/Vector3 angular with x, y, z axis whose values are of type float. These twist message basically contains information about linear velocity and angular velocity in x, y, and z axis of Turtlebot2. Since turtlebot2 doesn't have Omni-direction wheel. So, we can only use x axis for linear velocity and z for angular velocity because Turtlebot is been run on the ground and to see the components of the twist message we can use a simple command in command window in Linux as rosmsg show geometry_msgs/Twist.

So to perform a basic task like moving the robot straight from one point to another we will have to publish twist messages in linear direction and if we want any rotation so we can set angular direction but by using twist we cannot reach to exact position where we want our robot to move which can easily be seen in rviz because twist uses time as a reference to cover the given distance but in real time movement the turtlebot2 will not be moving with the same velocity provided by us to move it because the wheels of the robot will be moving on an area that can be slippery or hard which will create disturbance and the robot might reach before or after the time calculated to reach the goal so the robot will not be sure whether it reached to exact position or not with many other factors so it is highly not recommended when we want to move our robot in any environment we should not use twist messages. You can see in the figures below that how robot is moving in RVIZ when it is not running in real time but when moved in real time it doesn't reach to its original position, another way of doing this is to move the turtlebot2 by using odometry which is of type /Odom.



Motion in Rviz



Motion in real time

2.3 Odometery

Odometery makes use of the data acquired from motion sensors of the Turtlebot to detect change in position over time as well as their position relative to a starting location. The robot base controller uses PID control and odometery to change the motion commands into real world velocities and movement. The internal sensors are entirely responsible for the accuracy and reliability of this process, the accuracy of the calibration procedure, and environmental conditions. (For example, some surfaces may allow the wheels to slip slightly which will mess up the mapping between encoder counts and distance travelled.)By typing the below command in command window we will get the odometry of the robot.

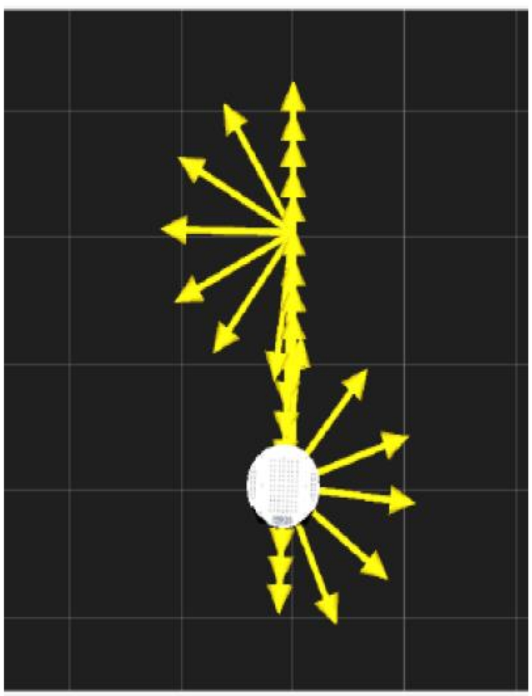
$rosmsg show nav_msgs/Odometry

After writing the above command in the command window we will be able to see an odometry message with a header, string of child frame and two sub-messages for PoseWithCovariance and

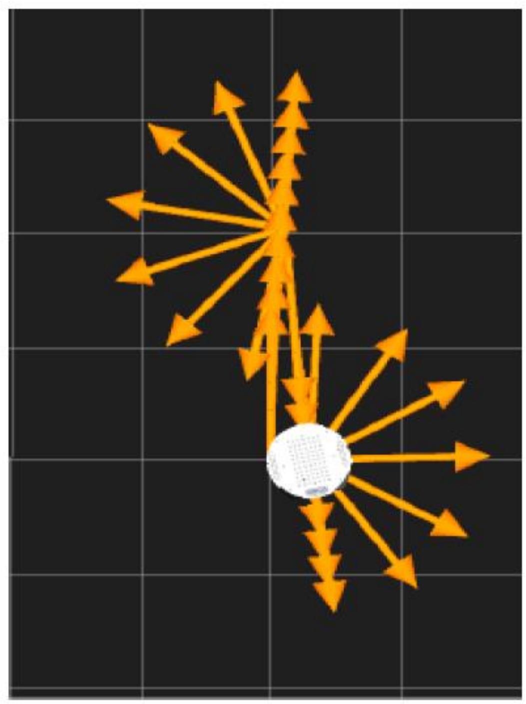
TwistWithCovariance what actually important is PoseWithCovariance because it actually tells you the position and angle of turtlebot2 which eases the process of navigation.

In order to effectively navigate the turtuleboot2 in an environment we just can't rely only on twist for the motion, as the turtleboot2 will not be able to identify the position where it is. A number of examples are already given in ROS rbx1_nav package to understand about twist and odometery.

There are different ways to control Turtlebot2 motion manually, It can be controlled by a joystick or a keyboard and by using either way we can navigate turtlebot2 where we like in the environment what actually we are doing is we are publishing messages from node of joystick to topic which is subscribed node joy2twist which further publish messages to topic /cmd_vel and which is further subscribed by robot driver to drive the robot.



Motion in Rviz



Motion in real-time

2.4 Path planning (move_base)

Now we know about twist and odometry both use different way to control or drive the robot and among both odometery is the better way to go with still there is an issue with it, the environment we use to maneuver the turtleboot2 in consists of an obstacle in the middle, so the turtleboot needs to be navigated in a way that it reaches its destination without any collision with the obstacle. In order to achieve that we have a package named as move_base package which uses point to point movement in order to reach the desired location. We only need to provide the origin co-ordinates and rest is done by the package and this is what path planning is the robot will be aware of its path in reference to origin and will reach its goal position.

Chapter 3

3. Map Building

As stated in the previous chapter the environment or arena being used to navigate the turtleboot2 in consists of an obstacle which the boot needs to be aware of in order to avoid collision with it. We can see the arena but we need to make sure the turtleboot2 should also be aware of the position of the obstacle in the arena, also the boot should be aware of the size and the places where it can freely move with in the arena. Here the concept of map building comes into play. Although the move_base is used to move the robot and avoid obstacle but the turtleboot2 also needs to know the position of the obstacle in order to achieve this we make a map. The map can be built by three basic techniques firstly by measuring the arena and drawing the same map by hand on a paper with black lines as obstacles or boundaries as measured and white area for moving the area , second way is to do the same in paint or any kind of similar software to draw the map but the last and most better way is to use a package which provides tools for building a map through laser scan in the Turtlebot, because map build by robot is more accurate and better as compared to the map been drawn by us .

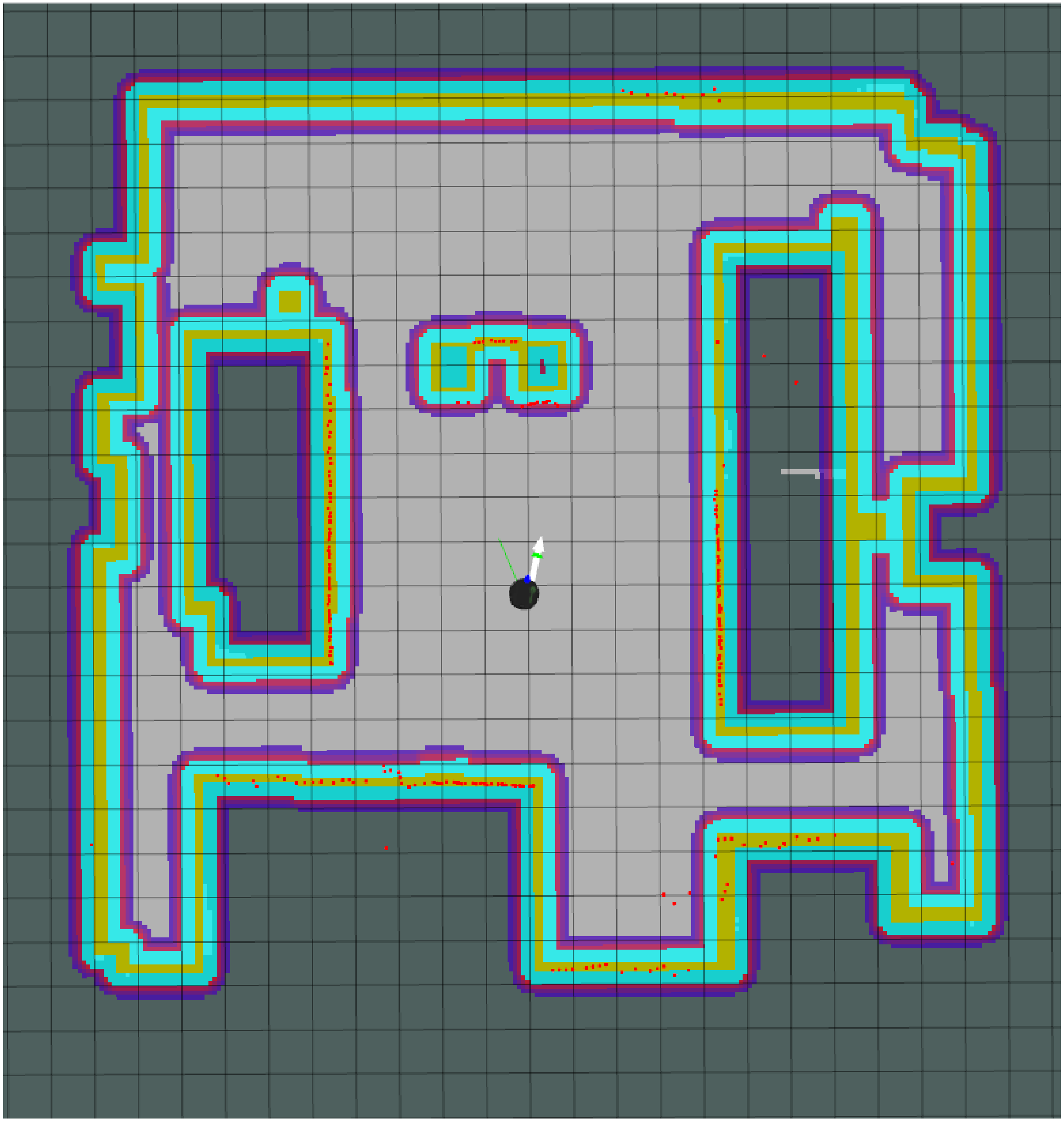
The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping and once the map is been made it should be saved and could be used to perform different task over turtlebot2, There are different commands to do so which are explained one by one as under.

- Put the robot in the arena from where you want to build the map, and turn on the Turtlebot so where ever you put the robot that will be the initial position of the robot where the robot should be put every time in order to navigate in the arena.

- So after turning it on launch the rplidar launch file to turn on the base and lidar of turtlebot2

- Then run *roslaunch turtlebot_le2i rplidar_minimal.launch file and roslaunch rbx1_nav rplidar_gmapping_demo.launch* on the turtlebot2

- On workstation run *roslaunch turtlebot_rviz_launchers view_navigation.launch and roslaunch turtlebot_teleop keyboard_teleop.launch* and then start navigating around the arena slowly with the teleoperation to create the map once you moved around whole arena and map is build save the map by using command *rosrun map_server map_saver -f arena* with arena as the name of the map you can choose whichever you like after doing so the map will be saved in the current directory with the named you specified and when you will open the directory it will have two files arena.pgm the image of the map and arena.yaml that describes the dimensions of the map, **figure** below shows the map we created



Final map

Roslaunch:

roslaunch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

Launch Files:

Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

```
<launch>

  <param name="use_sim_time" value="false" />

  <!-- EDIT THIS LINE TO REFLECT THE NAME OF YOUR OWN MAP FILE
       Can also be overridden on the command line -->
  <arg name="map" default="map_project.yaml" />

  <!-- Run the map server with the desired map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find project)/maps/$(arg map)"/>

  <!-- Start move_base  -->
  <include file="$(find rbx1_nav)/launch/tb_move_base.launch" />

  <!-- Fire up AMCL -->
  <include file="$(find rbx1_nav)/launch/tb_amcl.launch" />

  <node name="visualisation" pkg="rviz" type="rviz" output="screen" args="-d /home/bscv/ros/indigo/catkin_ws/src/rbx1/rbx1_nav/nav_test.rviz"/>


</launch>
```

Chapter 4

4. Navigation and Localization

As the map is build and we are able to navigate in the arena but there arises a problem i.e. the robot doesn't know where it is in the arena let suppose if we put the robot to right most corner but according to robot it is at the same initial position from where it started building the map and according to that it has an obstacle on is sides very near to it but when placed the Turtlebot2 in the middle of arena it doesn't know where it is located in the arena and this is the place where the concept of localization comes into matter.so how we will be able to know that where is our robot we use the concept of localization that is done through amcl.

We used move_base for moving the robot to a goal pose within a given reference frame with gmapping for creating a map from laser scan data and amcl for localization using an existing map.

What is amcl? ROS uses amcl package to localize the robot within an existing map using the current scan data coming from the robot's laser or depth camera.

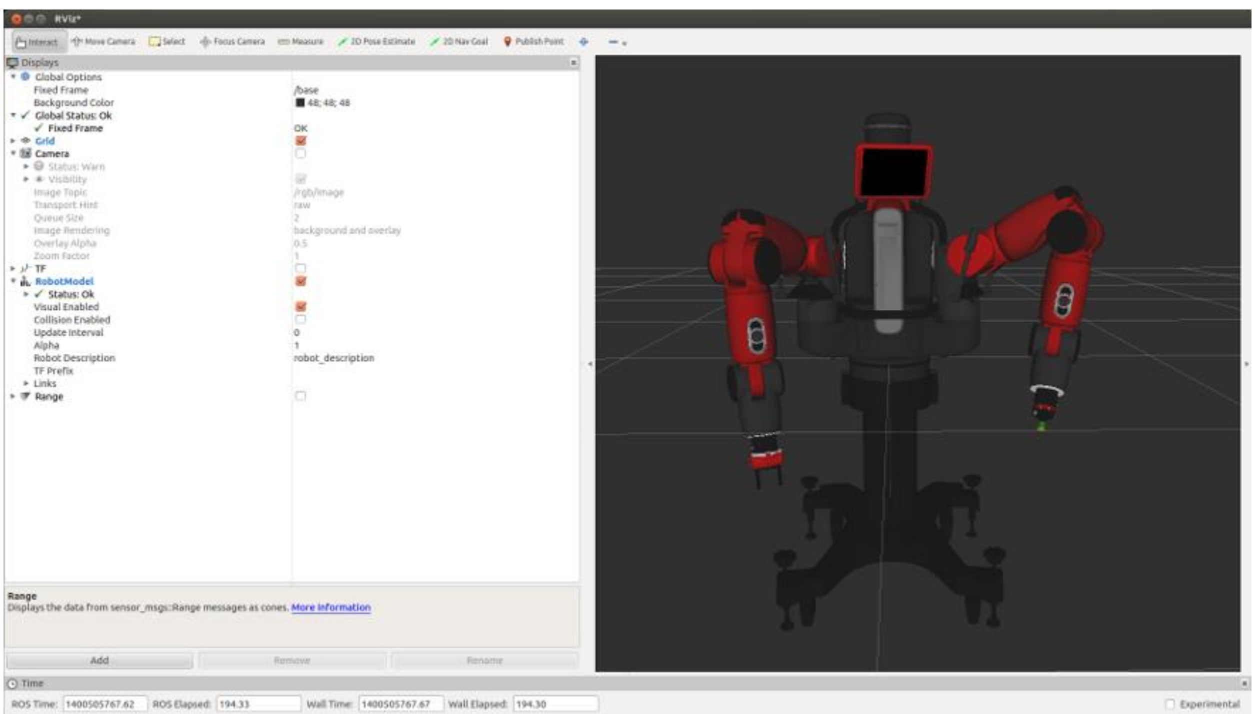
When amcl first starts up, you have to give it the initial pose (position and orientation) of the robot as this is something amcl cannot figure out on its own. Amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates.

To set the initial pose, first click on the 2D Pose Estimate button in Rviz. Then click on the point in the map where you know your robot is located. While holding down the mouse button, a large green arrow should appear. Move the mouse to orient the arrow to match the orientation of your robot, then release the mouse and in this way our initial pose is set according to map and this is the point where our project development comes into matter, we camed up with an idea to develop way that when the amcl start we shouldn't set the initial pose and it should be settle by its own. In the next topic that is 2D–Pose Estimation you will find the way how we been able to set the initial pose of turtlebot2 without using RVIZ.

RVIZ:

rviz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. Using rviz, you can visualize Baxter's current configuration on a virtual model of the robot. You can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

1. RVIZ is perfect for figuring out what went wrong in a vision system. The list on the left has a check box for each item. You can show or hide any visual information instantly.
2. RVIZ provides 3D visualization which you could navigate with just your mouse.
3. The best part of RVIZ is the interactive marker. This is the part where you can be really creative. It makes selecting a certain area in 3D relative easy. You can therefore adjust your vision system manually while it is still running such as select a certain area as your work space and ignoring other region.
4. You can have multiple vision processes showing vision data in the same RVIZ. You simply have to publish the point cloud or shape you want to show using the ROS publishing method. Visualizing is relatively painless once you get used to it.



Our code:

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

import roslib; roslib.load_manifest('rbx1_nav')
import rospy
import actionlib
from actionlib_msgs.msg import *
from geometry_msgs.msg import *
from visualization_msgs import *
from move_base_msgs.msg import MoveBaseActionGoal, MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker
from math import radians, pi
import numpy as np


class MoveBaseWayPoint():
    def __init__(self):
        self.number = 0                    #init the number of position we want to reach
        self.goal = MoveBaseGoal           #init the future goal
        self.WayPointsLists = list()       #create a list which will contain the goals
        self.index = 0                     #init an index
        self.over = 0                      #to know if we need to close the program
        self.MarkersLists = list()         #a list of markers
        self.success = 0                   #this will contain the number of goals reached
        self.getnumber()                   #use the function getNumber
```

```python
def getnumber(self):

    #this function will be used to get the number of goals we want to reach at the end

    rospy.init_node('listener', anonymous=True) #we create our node

    while self.number < 1:      #we don't want a number of goals less than 1 so we
                                #create a loop to obtain a positive integer
        try:
            self.number = int(raw_input('Enter the number of wanted waypoints: '))
                                #we check if the number is an integer

            if self.number < 1: #if the number is less than one
                rospy.loginfo("Not a positive number")  #we ask a positive number
        except ValueError:      #if the input is not an integer
            rospy.loginfo("Not a positive number")      #we ask a positive number

    rospy.loginfo("Now you can enter your differents waypoints")    #we display that we can register our way points
    self.index = 0                      #we reset our index
    while self.index < self.number:     #we want to do it until we got the good number of goals
        rospy.Subscriber('/move_base/goal', MoveBaseActionGoal, self.callback) #we subscribe to the goal node,
        rospy.spin()    #used to let only one instruction pass, this will wait another instruction
```

```python
def callback(self, msg):    #this function will be used each time we create a new nav goal

    if self.over == 0:  #if we hadn't finish the program
        new_move = actionlib.SimpleActionClient("move_base", MoveBaseAction) #"new_move" will be a variable containing a action, here a MoveBaseAction to move the robot
        new_move.wait_for_server()  #we are waiting the robot server
        new_move.cancel_all_goals() #We cancel every goals to avoir a movement of the robot

        self.goal = msg     #goal (a MoveBaseGoal) become the new goal we entered

        self.add_markers(self.goal) #with the goal we add a new markers on this position

        if self.index == 0: #if this is the first move we allow it to not be compare
            self.WaypointsLists = self.WayPointsLists.append(self.goal)
                #We add the new goal to the goals list
            self.index = self.index +1
                #we increment the index

        if self.WayPointsLists[self.index-1] != self.goal:
                #we compare the new goal to the last one to avoid some useless move
            self.WaypointsLists = self.WayPointsLists.append(self.goal)
                #We add the new goal to the goals list
            self.index = self.index +1
                #we increment the index

        if self.index >= self.number and not rospy.is_shutdown():
                #if the index is equal or greater than the number of goals
            rospy.loginfo("Procurement finished")   #we entered all the goals
            self.over = 1
            self.recurrence() #we call the function to reach goals

def recurrence(self):       #this function will be used when every goals will be inputed,
                             this will ask to the robot to reach every goals, one by one in the order of input

    self.index = 0
            #we reset our index
    while self.index <= self.number and not rospy.is_shutdown():
            #if the index is still less than the number of goals
        self.goal = self.WayPointsLists[self.index]
            #our goal become the umpteenth value of the list of goals
        self.index = self.index + 1
            #we increase the index

        self.move(self.goal)
            #we call the move function with the new goal
        if self.index == self.number:
            #if we did all goals
            self.shutdown()
            #we can shutdown the program
```

```python
def move(self, goal):        #this function will make move the robot

        new_move = actionlib.SimpleActionClient("move_base", MoveBaseAction) #"new_move" will be a variable containing a action,
                                                                                  here a MoveBaseAction to move the robot
        new_move.wait_for_server()  #we are waiting the robot server

        my_goal = MoveBaseGoal()    #This variable contain a MoveBaseGoal it will contain position of goals

        #We are creating our goal item by item to avoid a problem because of the conversion between MoveBaseAction and MoveBaseGoal

        my_goal.target_pose.header.frame_id = "/map"; # the frame id is used to determined the topic used
        my_goal.target_pose.pose.position.x = goal.goal.target_pose.pose.position.x #the spacial position X
        my_goal.target_pose.pose.position.y = goal.goal.target_pose.pose.position.y #the spacial position Y
        my_goal.target_pose.pose.orientation.z = goal.goal.target_pose.pose.orientation.z   #the orientation of the robot
        my_goal.target_pose.pose.orientation.w = goal.goal.target_pose.pose.orientation.w


        new_move.send_goal(my_goal)     #we are sending to the robot his goal converted

        rospy.loginfo("sending goal N*" + str(self.success+1)) #we display we are going to the umpteenth point

        new_move.wait_for_result() #we are waiting the robot to reach his destination

        state = new_move.get_state()    #we create a variable like a boolean to know if the robot is arrived
        if state == GoalStatus.SUCCEEDED:   #if we reach the destination
            rospy.loginfo("Goal N*" +str(self.success+1) + " succeeded") #we display that we arrived to the umpteenth goals
            self.suppr_marks(self.success)      #we can suppress the marker from this destination
            self.success = self.success + 1     #we increase the variable containing the number of goals reached



def add_markers(self, pos): #this function will add a marker to the position marked by goal
    marker = Marker()       #we create a new marker, a marker has an architecture pretty similar as the MoveBaseGoal

    marker.header.frame_id = "/map" #the topic

    marker.id = self.index  #the index, it need to be unique
    marker.ns = "Marker"    #some additional name if we need

    marker.action = marker.ADD  #the action permit to create, modify or delete a marker, here we create it

    marker.type = marker.ARROW  #the type of the marker, here an arrow

    marker.lifetime = rospy.Duration(0) #the life time of the marker, a 0 is equal to infity


    marker.pose.position.x = pos.goal.target_pose.pose.position.x   #position of the marker
    marker.pose.position.y = pos.goal.target_pose.pose.position.y
    marker.pose.position.z = 0.0

    marker.pose.orientation.x = 0.0     #orientation of the marker
    marker.pose.orientation.y = 0.0
    marker.pose.orientation.z = pos.goal.target_pose.pose.orientation.z
    marker.pose.orientation.w = pos.goal.target_pose.pose.orientation.w

    marker.header.stamp = rospy.Time.now() #we setup his internal timer to the actual time

    marker.scale.x = 0.75   #the size (in meter) of the marker
    marker.scale.y = 0.08
    marker.scale.z = 0.08

    marker.color.a = 1.0    #the color of the arrow in RGBA (red blue green alpha) , alpha need to be at 1 to not be invisible
    marker.color.b = 1.0
    marker.color.r = 1.0
    marker.color.g = 1.0

    marker.text = ("Goal N" + str(self.index)) #additionnal text

    self.MarkersLists.append(marker) #we add this marker on the list of markers
    self.Marker_Publisher()         #we call the publisher function
```

```python
def Marker_Publisher(self): #this function publish all markers
    self.pub = rospy.Publisher('/waypoint_markers', Marker , queue_size=10) #we create a publisher,
                                                                 here we are publishing to the topic
        # /waypoint_markers, we are publishing a marker.

    ind = 0      #the index

    while ind < len(self.MarkersLists):        #while the index is less than the lenght of the list

        self.pub.publish(self.MarkersLists[ind])    #we publish the umpteenth marker
        ind = ind +1                                #we increment the value of the index



def suppr_marks(self, index):       #this function will suppress our marker

     kill_marker = Marker()

    if len(self.MarkersLists) > 0: #If the length of the list is greater than 0
        self.MarkersLists[index].action = self.MarkersLists[index].DELETE  #we change the action of marker to delete
    #with this he will delete himself on rviz

    self.Marker Publisher() #we call the publisher

    def shutdown(self):
        rospy.loginfo("Stopping the robot...")
        # Cancel any active goals
        new_move = actionlib.SimpleActionClient("move_base", MoveBaseAction)
        new_move.wait_for_server()
        new_move.cancel_all_goals()

        rospy.sleep(2)
        rospy.sleep(1)

        rospy.signal_shutdown("done")
        sys.exit('done')
        # Stop the robot



if __name__ == '__main__':
    try:
        MoveBaseWayPoint()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation test finished.")
        pass
```

Brief overview

Get number;

This function is used to determine the number of goals we need to reach in the end. For this we create a node, we also set a loop and we make sure that in each iteration a single instruction is passed.

Call back;

This function will be used each time we create a new navigational goal. We create a variable containing the movement to be carried out by the robot. MoveBaseGoal will contain every new goal which will be entered.

Recurrence:

this function will be used when every goals will be inputed, this will ask to the robot to reach every goal, one by one in the order of input. We achieve this by setting an index and matching the value of the index with the list of goals to be achieved.

Def move:

This function will make move the robot "new_move" will be a variable containing a action, here a MoveBaseAction to move the robot. We made sure we are creating our goal item by item to avoid a problem because of the conversion between MoveBaseAction and MoveBaseGoal.

Def add_markers:

This function will add a marker to the position marked by goal. We create a new marker, the marker has an architecture pretty similar as the MoveBaseGoal. We make sure the index variable is a new one. Furthermore this function will define all aspects of the marker the type of marker it is, its position and orientation. After defining the marker we add this marker to the publisher list of markers and call the publisher function.

Def Marker_Publisher:

This function publishes all the markers. We create a publisher which is actually publishing all the markers to the topic.

Def suppr_marks:

the main purpose of this function is to kill maker. It works in a way that if the length of the list is greater than 0, we change the action to the makers to delete.

Def shutdown:

as the name suggests it is only to cancle any and all active commads and to stop the turtlebot.

Chapter 5

PhantomX pincher robot arm

The arm is designed by Trossen Robotics, the PhantomX pincher programmable Robotic arm comes with a hardware kit comes with everything needed to physically assemble and mount the arm as a standalone unit. The kit consists of all type of nuts, bolts and ranches to assemble the motors and Arduino board together forming the arm. The arm is able to seize, move and turn over small objects found nearby. . It's a 5DOF (5 degrees of freedom) robotic arm consisting of 5 joints making it capable of lifting, moving and rotational movements.
The Trossen Robotics KIT-RK-PINCHER Robotic Arm can be programmed using an open source ROS library, this robotics kit provides you simply with the hardware platform, and does not come with any software.

The aim was to interface the arm with ROS and achieve certain goals by using the arm to lift small objects. We were supposed to use move it for maneuvering the arm. Unfortunately due to lack of the time the task was not fully achieved.

The task of mounting the arm on the turtlebot was achieved. We did this by cutting out a wooden plank. The base of the arm where its arduino board rests was unscrewed form the body. The arm starting from its very first motor was screwed on to the wooden plank. As shown in the picture below.

Bibliography & Links

http://sdk.rethinkrobotics.com/wiki/File:Rviz.png

https://computervisionblog.wordpress.com/2012/11/18/rviz-a-good-reason-to-implement-a-vision-system-in-ros/

http://wiki.ros.org/rviz

http://answers.ros.org

http://wiki.ros.org/robot_localization

http://wiki.ros.org/gmapping

http://wiki.ros.org/Robots/TurtleBot.

Ros video : https://www.youtube.com/watch?v=5H7vLZGFers

GitHub for codes : https://github.com/RCorsin/BSCV_Robotic2017