Robotics Project Part I

Chalikonda Prabhu Kumar, Jampy Florian, Sai Krishna Pathi

Masters in Computer Vision, University of Burgundy

1 Introduction

Turtlebot from willow garage is a low cost personal robot kit and is equipped with the kinect sensor and open source software. With the kinect equipped, the turtlebot is able to avoid the obstacles by taking the 3d information of objects and is able to localise in 2 dimensional. By localising this information, turtlebot is able to drive without hitting the obstacles. The objectives of this project is to navigate the turtlebot.

2 Overview of Turtlebot

Turtlebot is built by Willow garage, which is programmable with Robot Operating System(ROS). The turtlebot is equipped with kinect and other sensors like odometry and gyro sensors. The company itself provides an open source software development kit(SDK) programmable with C++, ROS, Python. The model of the turtlebot is shown in the picture below.

The aim of this robotic project part1 is to develop certain goals on the turtlebot using Robot Operating System(ROS) . In this, we are working on the TurtleBot using ROS platform.

2.1 Kinect

Kinect is a motion sensing input device by Microsoft for the Xbox 360 video game console and Windows PCs. It enables users to control and interact with the Xbox 360 without the need to touch a game controller, through a natural user interface using gestures and spoken commands. It is one of turtlebot components that incorporate the image processing. The device features an RGB camera, depth sensor and multi-array microphone running proprietary software which provide full-body 3D motion capture, facial recognition and voice recognition capabilities.

2.2 ROS platform

In this project initially we are getting familiar with ROS environment to do certain goals. Robot Operating System (ROS) is a framework for robot software develop-

ment, providing operating system-like functionality of some robots on a heterogeneous computer cluster. It provides libraries and tools to help software developers and users to create robot applications. This TurtleBots can be used in house, for carrying research etc

The main goal of doing this project is to understand the robot motion planning in the environment, TurtleBot Localization and Navigation using ROS.

2.3 Communication betwen the turtlebot and work station

Making communication between TurtleBot and desktop/laptop is important. To run the code from the laptop/desktop this command is necessary inspite of running command on TurtleBot's laptop every time. Command used to run in the TurtleBot's laptop is 'roscore' on the other hand need to run ssh turtlebot@xx.xxx.xx.xx (xx.xxx.xx.xx is IP addrees of TurtleBot's laptop) to connect to the turtlebot's laptop. Some times like when using some packages some commands need to run on TurtleBot's laptop to do this every time on the terminal we have to run ssh turtlebot@xx.xxx.xx.xx on external laptop/desktop which communicate with TurtleBot with network.

To get the interaction with ROS environment and TurtleBot we did some tutorials [5]which helped in understanding the concepts and implementation on real time TurtleBot. There are few tasks which should be accomplished in this project moving turtleBot from one point to another, square and using some path planning packages to achieve optimal path by understanding nodes, topics, Publishing topics and subscribing topics etc. We used Rviz simulator before testing on the real Robot.

The work is summarised as follows:

- Section 3 is about the tasks performed with clear explanation of code, commands used, rqt_graph, and simulator results on the simulator along with the results in real time environment.
- Section 4 explains about the packages used to accomplish the tasks
- The conclusion followed by references are explained in Section 5

3 Tasks

The basic tutorials to understand the ROS environment clearly is cited in [5]. In this reference, the overview of some tutorials are documented.

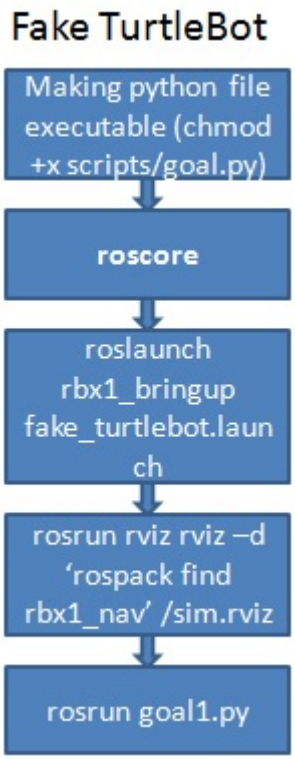
3.1 Task 1- Basic motion of Mobile base

After successfully finishing the tutorials with careful study, our primary task was to make the TurtleBot to move from point A to B, return to start point with low level programming. To accomplish this task the coding was done in python. The main idea is to run the robot with some linear velocity and angular velocity from one

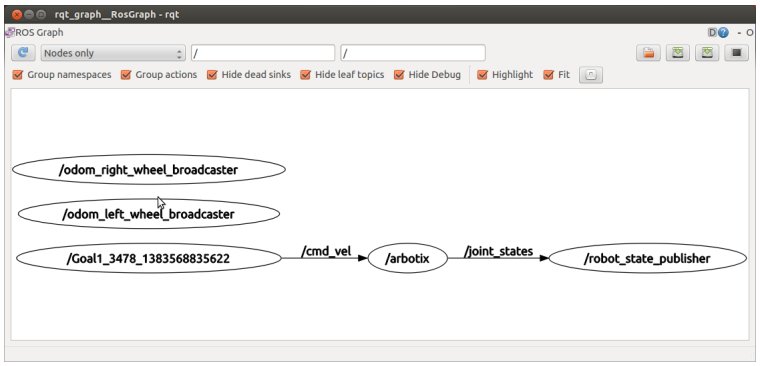
point to another. The start point was consiered as (0, 0) then for few iterations using 'for' loop, robot is moved with some linear velocity. After reaching the point B, the angle was changed using few iterations to make a good rotation and it was moved back to the starting point again for few more iterations.

Testing our code on simulator and real time too gave us good result. There are few things which has to be remembered are making file executable, launching fake/real turtleBot, and running the code that we developed.

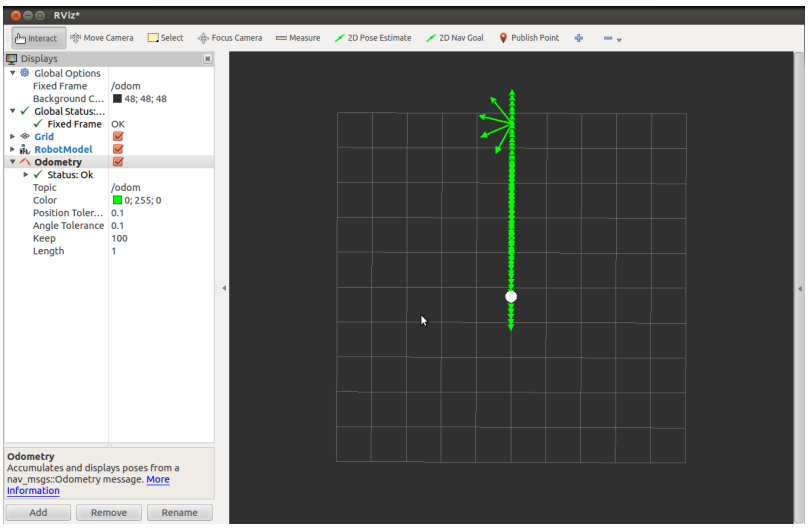
The flow charts below will give a clear picture for step by step implementation on fake and real turtleBot, each command has to run in a new terminal.

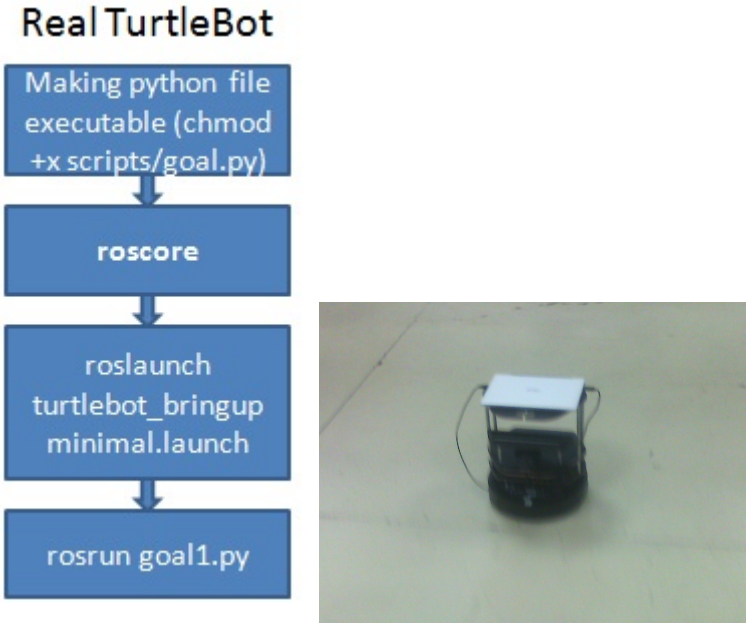


The code is developed from scratch and is presented in appendix section with comments for better understanding. The code developed is explained in appendix. The rqt_graph and rviz simulator results are shown below.



In the above figure, we can see that the turtleBot is moved from start point to target point. In rqt_graph nodes, cmd_velocity is publishing. Here linear and angular velocities are published using the topic Twist. The figure below depicts the results in rviz.

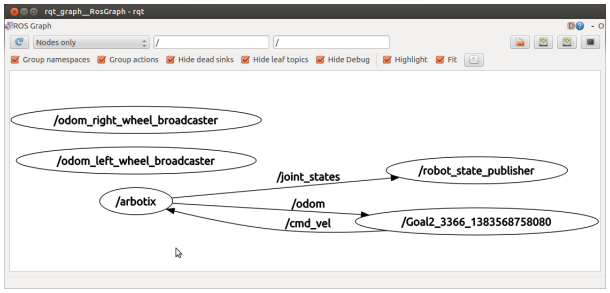




The real time robot gives good results. But, because of some problem with hardware, result was not satisfactory. When the turtlebot was changed, it gave good result as expected.

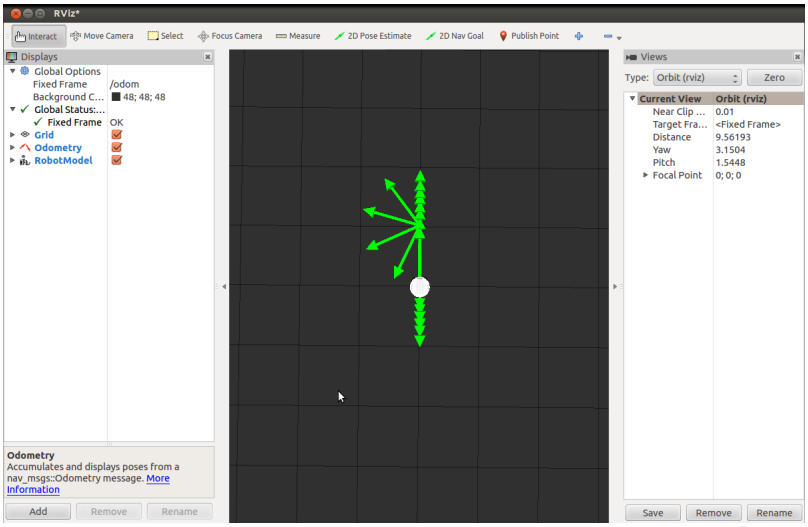
3.2 Task 2- Advanced motion of mobile base

In here the task is to move the robot from A to B and return to A considering the odometery values to accomplish the task. For this, we developed a program to move the robot from the starting point to the target then making an angle correction by taking the internal data of the TurtleBot to move in the same path in order to reach the starting point. Commands that are used in the new terminals are same as task1 changing file name and the python code is developed on simulator and TurtleBot for goal 2 is explained in appendix. The rviz with rqt_graph is shown below.



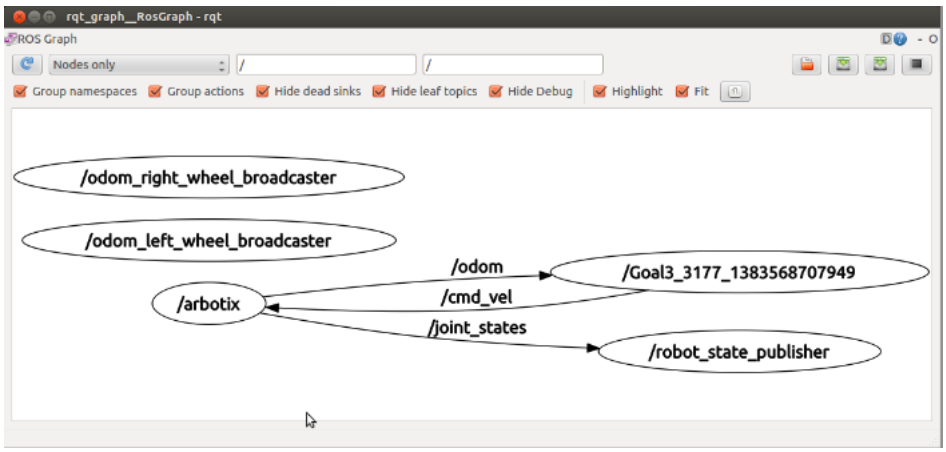
From the above figure, we can see that here we have publishing the cmd_velocity and subscribed to the odometry messages. The odometry is not working exactly

as expected, here we are facing some problems with drift due to the surface . So, here we have taken gyroscope into account. By considering gyro information, it works fine. The parameters like angle precision should be considered while implementing with odom_messages and twist. The precision angle to achieve to move back for the start point at some times problem encountered like losing the angle, where we want to move back, it takes more time to find appropriate angle to reach the start point or it is struck at the point only. The figure shown below is the simulation result by taking gyro information into account which gives better performance in real time and simulator.

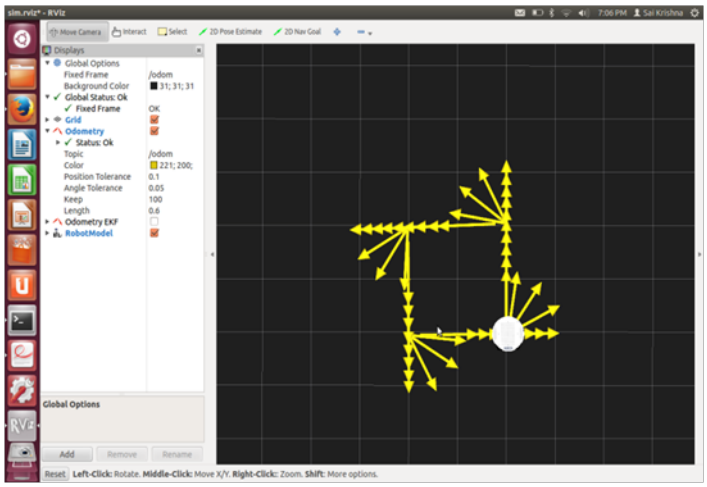


3.3 Task 3- Navigating in a square with twist/odometry

For this task, we have developed a code to move the TurtleBot in square using Twist (Linear and Angular Velocities) and Odometry (TurtleBot Pose and Orientation). The main theme is the code is moving the Turtlebot certain distance (1mt in our case) then making it to turn some angle (90 degrees in our case) and again move forward 1mt and correcting angle likewise till we reach the start point. For this the implementation was using odometry messages but as we did not get good results on the real time TurtleBot. We thought the problem is coming from the drift due to the surface of the floor. So, gyroscope was taken into account and built a code taking the gyro readings. Then it worked well for both simulator and real time robot. Python code with gyro, which gives good results, is explained in appendix. The rqt_graph is shown in the below figure for this code. We can see the publisher and subscriber topics.



The rviz result of goal3.py is shown below

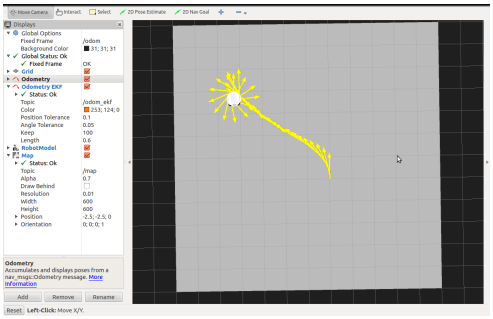


3.4 Task 4- Navigation with Path planning

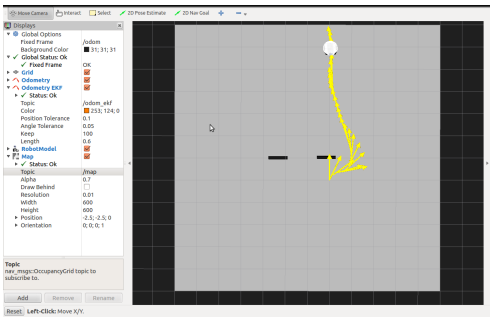
To obtain the navigation with path planning we have installed some packages like move_base which includes local and global path planner, ACML (adaptive Monto Carlo Localization) and EKF (Extended Kalman Filter). The primary goal in this task is to keep the robot pose same as the initial one when it reaches the goal. First, modification is done in move_base.launch file like <node remap "\cmd_vel" to "\mobile_base\commands\velocity" \>

3.4.1 Move_base (Global and Local Planner)

The move_base package [3] provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. The move_base node also maintains two cost maps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks. In this, the main theme is, first it will create a global path to reach goal and the robot follows this path and when it finds an obstacle in the path while moving towards the goal it uses local path planner to avoid obstacle and reach the goal.



(a)

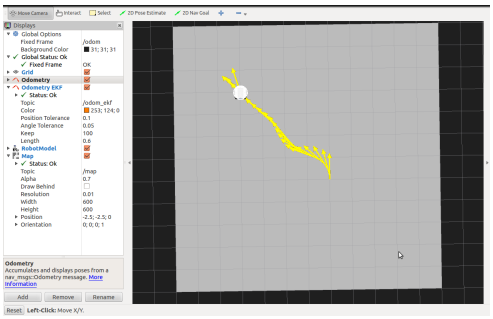


(b)

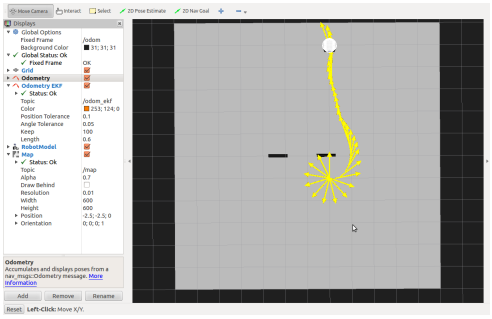
In the above, path planning package with move_base which includes global and local planner figure (a) is without any obstacle in 2D(x=2,y=2): figure (b) with obstacle 1D(x=3,y=0). We can observe that the start pose and target pose are same. In this path planning part to test the planners, we have created fake obstacles in rviz and the results obtained by the path planners were perfect.

3.4.2. AMCL (Adaptive Monto Carlo Localization)

AMCL [2] is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map. In this, implementation is done on an empty map for doing AMCL.



(a)

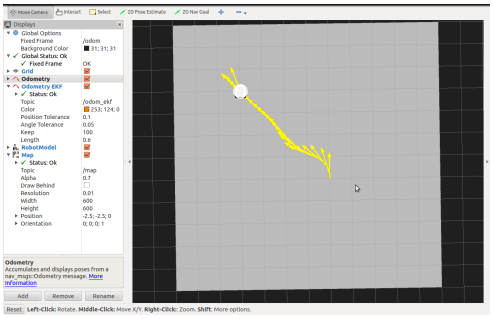


(b)

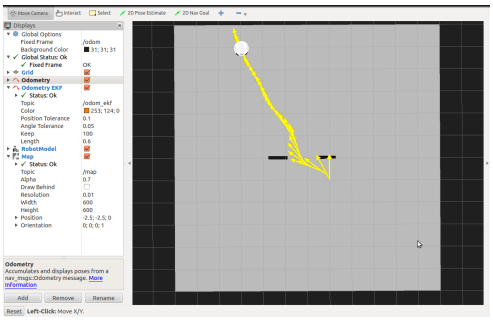
In the above, path planning package with amcl path planner packages figure (a) is without any obstacle in 2D(x=2,y=2): figure (b) with obstacle in 1D(x=3,y=0). We can see start pose and target pose are same. For this, we did the experiments only on the simulator because the path planner uses laser scan to localise itself in the real world with the real robot. We didn't built the map as we weren't suppose to, so we were unable to run on the real TurtleBot.

3.4.3 EKF (Extended Kalman Filter)

The Robot Pose EKF package [4] is used to estimate the 3D pose of a robot, based on (partial) pose measurements coming from different sources. It uses an extended Kalman filter with a 6D model (3D position and 3D orientation) to combine measurements from wheel odometry, IMU sensor and visual odometry. The basic idea is to offer loosely coupled integration with different sensors, where sensor signals are received as ROS messages.



(a)

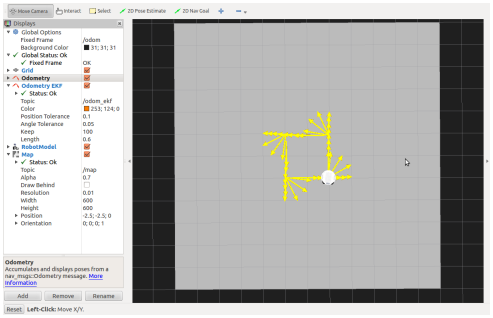


(b)

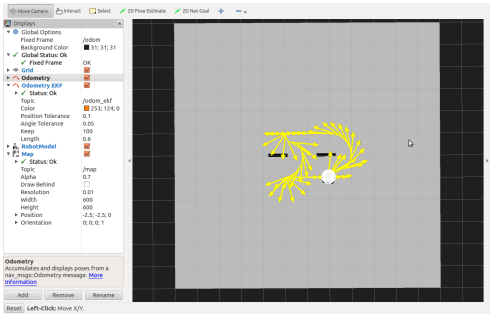
In the above, path planning package with ekf path planner packages figure (a) is without any obstacle in 2D(x=2,y=2): figure (b) with obstacle in 2D(x=3,y=2). We can see start pose and target pose are same. The difference between AMCL & EKF is AMCL is to localize (Laser Scan) the TurtleBot in 2D known map while EKF use 3D pose information coming from different sources. The EKF works by transforming the nonlinear models at each time step into linearized systems.

3.5 Task 5- Navigation of a square with path planning

In this, we are using path planning packages to find the optimal path when we are navigating a TurtleBot in a square. The results of simulator and turtlebot are shown below.



(a)

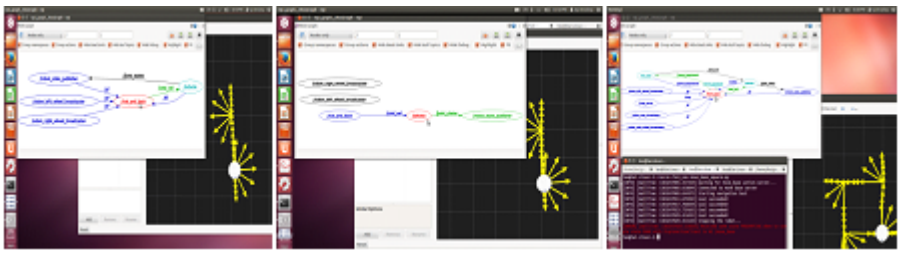


(b)

In the above, path planning package with move_base path planner package which includes global and local planner figure (a) moving in square without any obstacle: figure (b) moving in square with obstacle. Due to time constraint we were unable to work on other path planning packages.

4. Using Packages

In this section, we summarize the packages that are used for making the TurtleBot to move, path planning packages, TurtleBot packages etc. To make sure that packages are installed properly we made some screen shots of images using simulator. To accomplish the first three goals we create a package in the sandbox with some dependencies. "roscreate-pkg part1 roscpp sensor_msgs gmapping". Once packages are created we build it using "rosmake". Then we created new folder named as 'scripts' in part1 package. We developed a code for A to B, A to B to A and square with names goal1.py, goal2.py and goal3.py. Once the python script is done we make it executable using 'chmod +x scripts/goal1.py'. After making the scripts executable we run the commands as showed in the flow charts for Task 1, Task 2 and Task 3. One should make note that every command should run in the new terminal. The below shown images are when we run built-in codes which are provided in the package rbx1.



To accomplish the goal4 and goal5 with path planning packages first we need to install packages[1]. Once packages are installed, should take care of some .yaml file to specify the parameters of the packages, launch files etc then we tested on the simulator. Commands that used to run the path planning packages are shown in section 2, task 4 and task 5 with simulator results. Note: After installing the packages 'sudo apt-get update' command will update all the packages and their dependencies in ros.

5. Conclusion

To get handful experience on ROS with TurtleBot we first implemented the tutorials which were provided in the lab. Once we got the knowledge on ROS commands, then we developed code for moving TurtleBot from A to B. Next we made A to

B and return to A. For this we made some angle correction after reaching B and made the TurtleBot to move back from B to A. In order to make the square, we make TurtleBot to correct the angle by taking odometry messages after moving few meters (1mt in our case) to rotate in appropriate angle and move for certain distance again correct angle to reach the start point by forming square. First we did with odometry messages but there is some problem on real TurtleBot because of surface, drift etc keeping this drawbacks in mind we modified our code taking gyro reading into account then we got succeed in real time TurtleBot to get exact square without any errors. For path planning we tried to implement some packages in fact we succeed in doing with Move_base (global and local planner), AMCL, EKF. Among this, Move_base provided good result for navigating in a square. Path planners are very useful in obstacle avoidance and to reach the goal. The precision, robot pose, and angle rotation are very fine using them. By using more path planners we can explore more things in future and mainly working with Kinect and lasers in path planning will give more precise information.

Bibliography

[1] R.Patrick Gabriel. *ROS by examples volume 1.*

[2] http://wiki.ros.org/amcl.

[3] http://wiki.ros.org/movebase.

[4] http://wiki.ros.org/robotposeekf.

[5] http://wiki.ros.org/ROS/Tutorials. wiki3.

Appendix

Goal 1

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

import rospy
from geometry_msgs.msg import Twist

def Goal1():
    ## Definition
    rospy.init_node('Goal1', anonymous=True)
    pub = rospy.Publisher('/cmd_vel', Twist)
    msg = Twist()
    r = rospy.Rate(10) # 10hz
    ## Move Forward
    for x in range(1, 50):
        msg.linear.x= 1
        msg.angular.z= 0
        pub.publish(msg)    # Publish Twist Topic
        r.sleep()
    ## Rotate
    for x in range(1, 79):
        msg.linear.x= 0
        msg.angular.z= 0.4
        pub.publish(msg)    # Publish Twist Topic
        r.sleep()
    ## Move Forward
    for x in range(1, 50):
        msg.linear.x= 1
        msg.angular.z= 0
        pub.publish(msg)    # Publish Twist Topic
        r.sleep()
    ## Stop Turtlebot
    msg.linear.x= 0
    msg.angular.z= 0
    pub.publish(msg)    # Publish Twist Topic

if __name__ == '__main__':
    try:
        Goal1()
    except rospy.ROSInterruptException: pass
```

Goal 2

```python
def callback(data):
    global Xstart
    global Ystart
    global Anglestart
    global ini
    global Step
    acc = 0.5           # Angle accuracy
    Goal = 1            # Distance from Goal (waypoint)
    Speed = 0.4         # Linear speed
    Rotation = 1        # Angular speed

    msg = Twist()
    # obtaining X Y Z W values from Odom
    X = data.pose.pose.position.x
    Y = data.pose.pose.position.y
    Z = data.pose.pose.orientation.z
    W = data.pose.pose.orientation.w

    # convert Angle in degrees
    Angle = math.degrees(math.asin(Z)) * 2
    # convert Angle format 0-360
    if Angle < 0:
        Angle = 360 + Angle
    elif Angle == 0 and (W == 0):
        Angle = 180
    # initialisation
    if ini == 0:
        Step = 0
        Xstart = data.pose.pose.position.x
        Ystart = data.pose.pose.position.y
        Anglestart = Angle
        ini = 1
    # compute the distance traveled
    dt = math.sqrt(math.pow((math.fabs(X-Xstart)), 2) + math.pow((math.fabs(Y-Ystart)), 2))


    Rangle = Anglestart + 180 # compute desired Angle
    if Rangle >= 360:
        Rangle = Rangle - 360
```

```python
    if dt >= Goal:
        if Rangle > Angle - acc and Rangle < Angle + acc: # stop robot for the good angle
            Xstart = X
            Ystart = Y
            msg.linear.x = 0
            msg.angular.z = 0
            Step = 1
        else: # Rotate to find the good angle
            msg.linear.x = 0
            msg.angular.z = Rotation
            if Step == 1:
                msg.linear.x = 0
                msg.angular.z = 0
    else: # Move forward
        msg.linear.x = Speed
        msg.angular.z = 0
        Anglestart = Angle # Reset Starting angle

    pub.publish(msg) # Publish value


def Goal2():
    global pub
    rospy.init_node('Goal2', anonymous=True)
    rospy.Subscriber("/odom", Odometry, callback)
    pub = rospy.Publisher('/mobile_base/commands/velocity', Twist)
    rospy.spin()

if __name__ == '__main__':
    try:
        ini = 0
        Goal2()
    except rospy.ROSInterruptException: pass
```

Goal 2 gyro

```python
def GetVal(data):   # obtaining angle from Gyroscope
    global Z
    global W
    Z = data.orientation.z
    W = data.orientation.w

def callback(data):
    global Xstart
    global Ystart
    global Anglestart
    global Z
    global W
    global ini
    global Step
    acc = 0.5           # Angle accuracy
    Goal = 1            # Distance from Goal (waypoint)
    Speed = 0.4         # Linear speed
    Rotation = 1        # Angular speed

    msg = Twist()
    # obtaining X Y values from Odom
    X = data.pose.pose.position.x
    Y = data.pose.pose.position.y
    # convert Angle in degrees
    Angle = math.degrees(math.asin(Z)) * 2
    # convert Angle format 0-360
    if Angle < 0:
        Angle = 360 + Angle
    elif Angle == 0 and (W == 0):
        Angle = 180
    # initialisation
    if ini == 0:
        Step = 0
        Xstart = data.pose.pose.position.x
        Ystart = data.pose.pose.position.y
        Anglestart = Angle
        ini = 1

    dt = math.sqrt(math.pow((math.fabs(X-Xstart)), 2) + math.pow((math.fabs(Y-Ystart)), 2))
```

```python
    Rangle = Anglestart + 180 # compute desired Angle
    if Rangle >= 360:
        Rangle = Rangle - 360

    print " dt = %2f" % dt
    print " Angle = %2f" % Angle
    print " Goal = %2f" % Goal
    print " Anglestart = %2f" % Anglestart
    print " Rangle = %2f" % Rangle
    print "_____"

    if dt >= Goal:
        if Rangle > Angle - acc and Rangle < Angle + acc: # stop robot for the good angle
            Xstart = X
            Ystart = Y
            msg.linear.x = 0
            msg.angular.z = 0
            Step = 1
        else: # Rotate to find the good angle
            msg.linear.x = 0
            msg.angular.z = Rotation
            if Step == 1:
                msg.linear.x = 0
                msg.angular.z = 0
    else: # Move forward
        msg.linear.x = Speed
        msg.angular.z = 0
        Anglestart = Angle # Reset Starting angle

    pub.publish(msg) # Publish value


def Goal2():
    global pub
    global Z
    global W
    rospy.init_node('Goal2', anonymous=True)
    rospy.Subscriber("/mobile_base/sensors/imu_data", Imu, GetVal)
    rospy.Subscriber("/odom", Odometry, callback)
    pub = rospy.Publisher('/mobile_base/commands/velocity', Twist)
    rospy.spin()
```

Goal 3

```python
def callback(data):
    global Xstart
    global Ystart
    global Anglestart
    global ini

    acc = 1             # Angle accuracy
    Goal = 1            # Distance from Goal (waypoint)
    Speed = 0.4         # Linear speed
    Rotation = 0.5      # Angular speed

    msg = Twist()
    # obtaining X Y Z W values from Odom
    X = data.pose.pose.position.x
    Y = data.pose.pose.position.y
    Z = data.pose.pose.orientation.z
    W = data.pose.pose.orientation.w

    # convert Angle in degrees
    Angle = math.degrees(math.asin(Z)) * 2
    # convert Angle format 0-360
    if Angle < 0:
        Angle = 360 + Angle
    elif Angle == 0 and (W == 0):
        Angle = 180
    # initialisation
    if ini == 0:
        Xstart = data.pose.pose.position.x
        Ystart = data.pose.pose.position.y
        Anglestart = Angle
        ini = 1

    # compute the distance traveled
    dt = math.sqrt(math.pow((math.fabs(X-Xstart)), 2) + math.pow((math.fabs(Y-Ystart)), 2))

    Rangle = Anglestart + 90 # compute desired Angle
    if Rangle >= 360:
        Rangle = Rangle - 360
```

```
    print " dt = %2f" % dt
    print " Angle = %2f" % Angle
    print " Goal = %2f" % Goal
    print " Anglestart = %2f" % Anglestart
    print " Rangle = %2f" % Rangle
    print "--------------------"


    if dt >= Goal:
        if Rangle > Angle - acc and Rangle < Angle + acc: # stop robot for the good angle
            Xstart = X
            Ystart = Y
            msg.linear.x = 0
            msg.angular.z = 0
        else: # Rotate to find the good angle
            msg.linear.x = 0
            msg.angular.z = Rotation
    else: # Move forward
        msg.linear.x = Speed
        msg.angular.z = 0
        Anglestart = Angle # Reset Starting angle

    pub.publish(msg) # Publish value


def Goal3():
    global pub
    rospy.init_node('Goal3', anonymous=True)
    rospy.Subscriber("/odom", Odometry, callback)
    pub = rospy.Publisher('/mobile_base/commands/velocity', Twist)
    rospy.spin()

if __name__ == '__main__':
    try:
        ini = 0
        Goal3()
    except rospy.ROSInterruptException: pass
```

Goal 3 gyro

```
    # compute the distance traveled
    dt = math.sqrt(math.pow((math.fabs(X-Xstart)), 2) + math.pow((math.fabs(Y-Ystart)), 2))

    print " Anglestart = %2f" % Anglestart
    print " Rangle = %2f" % Rangle
    print "--------------------"
    if dt >= Goal:
        if Rangle > Angle - acc and Rangle < Angle + acc: # stop robot for the good angle
            Xstart = X
            Ystart = Y
            msg.linear.x = 0
            msg.angular.z = 0
            Rotupd = 0
        else: # Rotate to find the good angle
            if Rotupd == 0:
                Rangle = Rangle + 90 # update desired Angle
                if Rangle >= 360:
                    Rangle = Rangle - 360
                Rotupd = 1

            msg.linear.x = 0
            msg.angular.z = Rotation

    else: # Move forward
        msg.linear.x = Speed
        msg.angular.z = 0

    pub.publish(msg) # Publish value

def Goal3():
    global pub
    global Z
    global W
    rospy.init_node('Goal3', anonymous=True)
    rospy.Subscriber("/mobile_base/sensors/imu_data", Imu, GetVal)
    rospy.Subscriber("/odom", Odometry, callback)
    pub = rospy.Publisher('/mobile_base/commands/velocity', Twist)
    rospy.spin()
```

```
def GetVal(data):   # obtaining angle from Gyroscope
    global Z
    global W
    Z = data.orientation.z
    W = data.orientation.w

def callback(data):
    global Xstart
    global Ystart
    global Anglestart
    global Z
    global W
    global ini
    global Rangle
    global Rotupd

    acc = 1             # Angle accuracy
    Goal = 1            # Distance from Goal (waypoint)
    Speed = 0.4         # Linear speed
    Rotation = 0.5      # Angular speed

    msg = Twist()
    # obtaining X Y values from Odom
    X = data.pose.pose.position.x
    Y = data.pose.pose.position.y
    # convert Angle in degrees
    Angle = math.degrees(math.asin(Z)) * 2
    # convert Angle format 0-360
    if Angle < 0:
        Angle = 360 + Angle
    elif Angle == 0 and (W == 0):
        Angle = 180
    # initialisation
    if ini == 0:
        Xstart = data.pose.pose.position.x
        Ystart = data.pose.pose.position.y
        Anglestart = Angle
        Rangle = Anglestart + 90 # compute desired Angle
        Rotupd = 1
        ini = 1
```