UE 2-4 Robotics Engineering 2

Technical Report

Submitted by **Meldrick REIMMER and Selma BOUDISSA**

Contents

Introduction **3**
Turtlebot 2 4
Hardware & Software Specifications 4

Objectives **5**

1 Part 1: Motion control **7**
1.1 Goal 1: Basic Motion of Mobile Base 7
1.2 Goal 2: Advanced Motion of Mobile Base 8
1.3 Goal 3: Navigating a Square using Twist+Odometry 8
1.4 Goal 4: Navigation with Path Planning 9

2 Part 2: Planar Lasar Range Finder **10**
2.1 Goal 1: ROS Interfacing 10
2.2 Goal 2: Integration on Turtlebot2 10

3 Part 3: Navigation & Localization **11**
3.1 Goal 1: Map Building 11
3.2 Goal 2: Navigation & Localization 13
3.2.1 Design: 13
3.2.2 Implementation: 13
3.2.3 Demonstration: 13

4 Part 4: Robotic Arm **14**
4.1 PhantomX Pincher Robot Arm 14
4.2 Application 14

5 Errors encountered **15**

6 Conclusion **16**

REFERENCES **16**

VIDEOS **17**

APPENDIX **18**
6.1 APPENDIX 5 30

Acknowledgement

We are grateful to the teachers of the Robotics Engineering, Mr Ralph Seulin, Mr Raphael Duverne, Mr Jiang Cansen and Mrs Mojdeh Rastagoo for their kind gesture and warm reception they gave us during this final semester. Things were more easier and fun with their help. We will also like to thank our colleagues for their help,sharing of knowledge and time during our needs of help. Special thanks to Antoine Merlet for the much help he gave us during the final minute of our project. A great semester we have had a brighter future we see together. Good things they say comes to no man who waits and put his hands in between his legs. We can say we have fought towards the end and achieved marvelous results. We dedicate this report to our Parents and our cherished loved ones.

Introduction

A semester has come to an end and what a great one it has been. This semester we were introduce to a new world of technology. A world of robotics, an environment where the belief is that the task humans can do a Metallic with artificial intelligence can achieve that task, which is the ROBOT. The new world would have them in every aspect of life mostly places where there is high demand of human factor. They have started coming into existence, as they are being used in many manufacturing companies and even in some households. They are costive but gives high productivity to what they are meant do. For instance, in a car manufacturing companies they are mainly use to assembly the parts of the car. Research made says they perform an approximately 80% of job done in very company that they are found.

Turtlebot2

This semester we had the opportunity to work with one of them which is the Kobuki Trutlebot2. The Kobuki Turtlebot is a low-cost mobile research base designed for education and research on state of art robotics. With continuous operation in mind, Kobuki provides power supplies for an external computer as well as additional sensors and actuators. Its highly accurate odometry, amended by our factory calibrated gyroscope, enables precise navigation. TurtleBot was created by Willow Garage by Melonee Wise and Tully Foote in November 2010.

Hardware & Software Specifications

The turtlebot2 is the updated versiosn of the turtlebot . Few specifications are as follows:

Functional:

- Maximum translational velocity: 70 cm/s
- Maximum rotational velocity: 180 deg/s (>110 deg/s gyro performance will degrade)
- Payload: 5 kg (hard floor), 4 kg (carpet)
- Cliff: will not drive off a cliff with a depth greater than 5cm
- Threshold Climbing: climbs thresholds of 12 mm or lower
- Rug Climbing: climbs rugs of 12 mm or lower
- Expected Operating Time: 3/7 hours (small/large battery)
- Expected Charging Time: 1.5/2.6 hours (small/large battery)
- Docking: within a 2mx5m area in front of the docking station

Hardware:

- PC Connection: USB or via RX/TX pins on the parallel port
- Motor Overload Detection: disables power on detecting high current (>3A)
- Odometry: 52 ticks/enc rev, 2578.33 ticks/wheel rev, 11.7 ticks/mm
- Gyro: factory calibrated, 1 axis (110 deg/s)
- Bumpers: left, center, right
- Cliff sensors: left, center, right
- Wheel drop sensor: left, right
- Power connectors: 5V/1A, 12V/1.5A, 12V/5A
- Expansion pins: 3.3V/1A, 5V/1A, 4 x analog in, 4 x digital in, 4 x digital out
- Audio : several programmable beep sequences
- Programmable LED: 2 x two-coloured LED
- State LED: 1 x two coloured LED [Green - high, Orange - low, Green & Blinking - charging]
- Buttons: 3 x touch buttons
- Battery: Lithium-Ion, 14.8V, 2200 mAh (4S1P - small), 4400 mAh (4S2P - large)
- Firmware upgradeable: via usb
- Sensor Data Rate: 50Hz
- Recharging Adapter: Input: 100-240V AC, 50/60Hz, 1.5A max; Output: 19V DC, 3.16A
- Netbook recharging connector (only enabled when robot is recharging): 19V/2.1A DC
- Docking IR Receiver: left, centre, right

Software:

- C++ drivers for Linux
- ROS driver
- Gazebo simulation

Objectives

Our goal for this semester is for us to implement the Turtlebot2 in performing certain tasks. We are to give report based on what we do with it concerning the procedures used in performing these various tasks.

Firstly, we are to perform a motion control which consist of the Twist, Odometry and Move_base. There are four goals in this section, which includes:

1. We are to make a Basic Motion of Mobile Base which will involve a mobile base control with ow-level programming.
2. Advanced Motion of Mobile Base which will involve a mobile base control with higher-level programming.
3. Navigating a square using Twist and Odometry
4. Navigation with Path planning (Move_Base)

Secondly, we are to implement the Planar Laser RangeFinder on the Turtlebot2. We are to perform a Ros interfacing to be able to use it in the ros environment and Integrate it on the Turtlebot2.

Thirdly, we are to perform Navigation and localization. This part contains two goals, which includes:

1. Map Building, which involves gmapping and TeleOperaton.
2. Designing, Implementing and Demonstrating the manner in which Navigation and localization is done.

Lastly, we are to implement the PhantomX Pincher Robotic Arm on the Turtlebot2.

1. Assemble the parts
2. Perform a test and debugging on a Robotic Arm.
3. Perform a Ros Interfacing to be able for us to use it in the Ros environment.
4. Perform an application with the Robotic Arm.

Chapter 1

Part 1: Motion control

In this part, the main idea is to understand that to accomplish a task given to the TurtleBot which is our robot, there are many ways in which we can achieve that task. In ROS environment, we can illustrate different ways in which the same task assign to the TurtleBot can be done.

1.1 Goal 1: Basic Motion of Mobile Base

For the first manipulation we use the timed Twist command to move robot foward a certain distance. The test have been realized on a real robot which is the Turtlebot 2 We display the combined odometry data.The script timed_out_and_back.py is publishing Twist messages on the /cmd_vel_mux topic. Basic example of how ROS allow us to abstract away the lower-level of motion control. The script used for this manipulation is on appendix 1 at the end of this final report.

Here after the figure of the result executing the script timed_out_and_back.py.

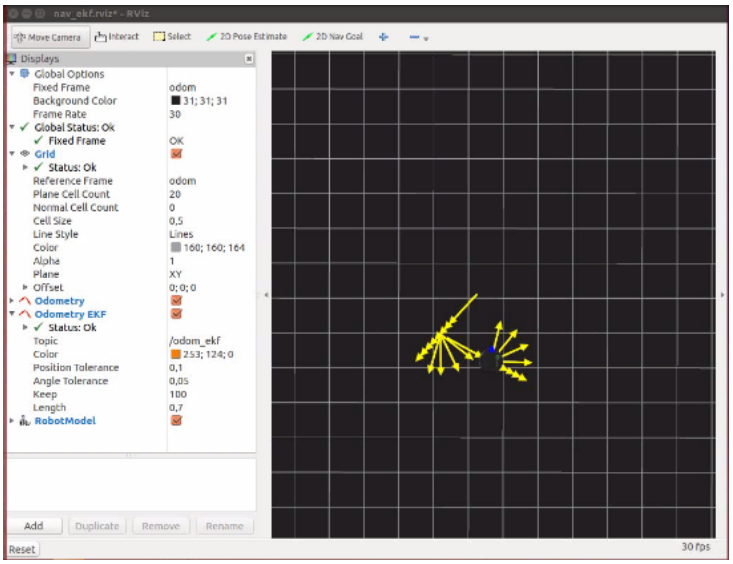

Figure 1.1: Publishing twist message

Explanation: The robot's base controller node use odometry and PID controller to turn motion request into real-world velocities. Refer to Twist for the video of this manipulation.

1.2 Goal 2: Advanced Motion of Mobile Base

For this second manipulation we will use Odometry. You can find the script that had been used called: odom_out_and_back.py in appendix 2.

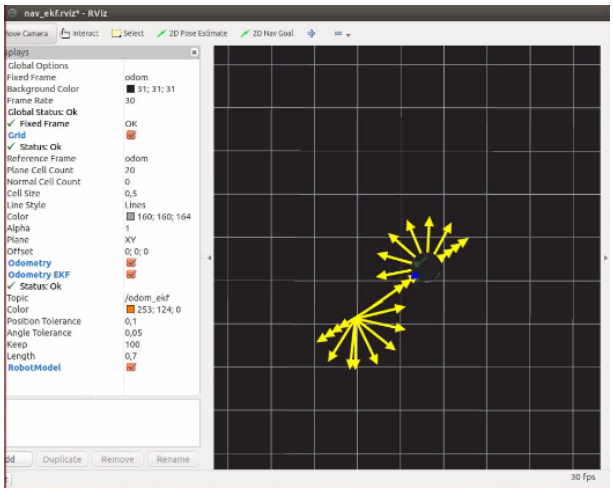

Figure 1.2: Using Odometry

Explanation: Compare to the figure above using the Twist command, the result here is much better. The pacakage turtlebot_le2i, that you can found in the reference number, (put the reference nb and the github link) is where is stored the calibrations parameters. Refer to Odom for the video of this manipulation.

1.3 Goal 3: Navigating a Square using Twist+Odometry

For this goal we use the script nav_square.py in appendix 3 :

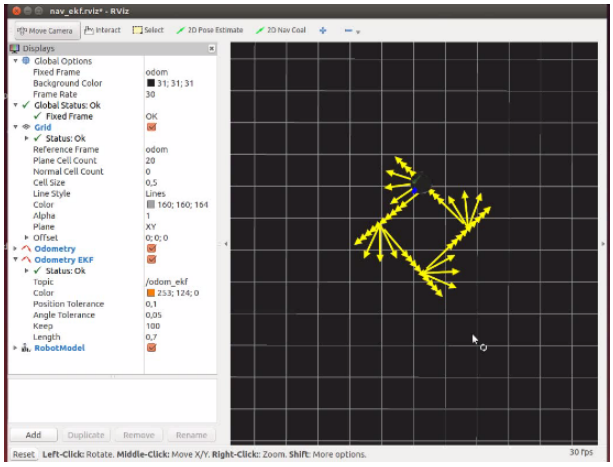

Figure 1.3: Navigating a square

Explanation: In the figure above you can see that the robot it performing a square sequence. Refer to Square for the video of this manipulation

1.4 Goal 4: Navigation with Path Planning

The move_base is provided by ROS to do the same manipulation as shows in goal 3 in a nicer way. The move_base package implements a ROS action for reaching a given navigation goal. The move_base package incorporates the base_local_planner that combines odometry data with both global and local cost maps, when selecting a path for the robot to follow.

To specify a navigation goal using move_base, we provide a target pose which is the position and orientatio of the robot with respect to a particular frame or reference. The move_base package uses the MoveBaseAction message type for specifying the goal.

The test of the goal have be done on the turtlebot using the script move_base_square.py in appendix 4.

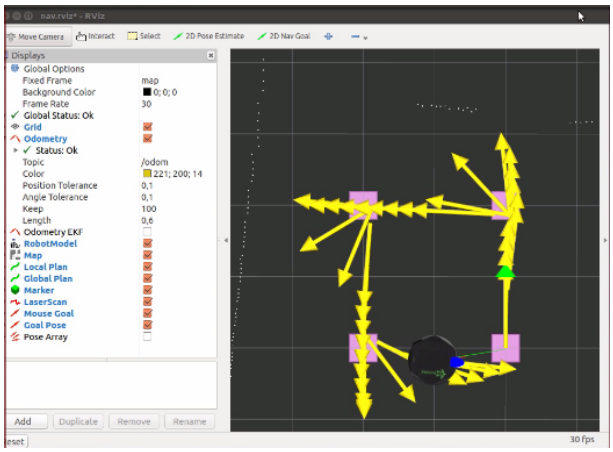

Figure 1.4: Move_base

Refer to Move_Base for the video of this manipulation.

Chapter 2

Part 2: Planar Lasar Range Finder

This have been already implemented on our Turtlebot2 we will just given some explanation about our understanding about it.

RPLIDAR is a low cost lidar sensorsuitabke fir indoor robotic Simultaneous Localization and Mapping (SLAM) applications.

RPLIDAR specification:

- Scan field:360 degrees
- Rotating frequency: 5.5 Hz/ 10Hz
- Ranger distance: 8 meters

2.1 Goal 1: ROS Interfacing

For this part we download on github the RPLIDAR package. This package provides basic device handling for 2D Laser Scanner RPLIDAR A1/A2. In the following link Wiki rplidar you will found more explanation about the rplidar rospackage and details about ROS nodes, device setting...

2.2 Goal 2: Integration on Turtlebot2

Our explanation refere to the ROS Vol 2 chapter 4.6 The laser was already implemented on the Turtlebot2.

Take a picture using RVIS $ roslaunch rbx2_description box_robot_base_with_laser.launch $ rosrun rviz rviz - d ‘rospack find rbx2_description‘/urdf.rviz

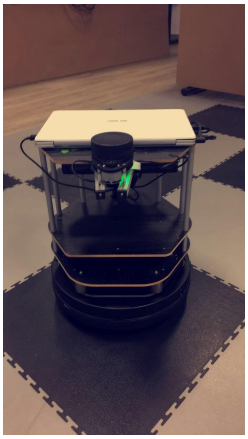

Figure 2.1: RPLIDAR on Turtlebot2

Chapter 3

Part 3: Navigation & Localization

Now that we have the knowledge and the way how to control a drive robot, we are goind to introduce the Simultaneous Localization and Mapping (SLAM), one of the more powerful features in ROS.

In this chapter we will use the two essential ROS package that able the navigation which is:

1. gmapping
2. amcl

3.1 Goal 1: Map Building

In this section we used the gmapping and the Teleoperation. This manipulation have been realized on a real map of the robot's environnement.

Just know that under ROS a map is represented by a bitmap image representing an occupancy grid where we draw our map.

The ROS gmapping package contains the slam_gmapping node that does the work of combining the data from laser scans and odometry into occupancy map.

First step we collect and record the scan data. Start your ROS environnement by the command roscore. (Start the minimal launch file in the workstation or Turtlebot2 be careful if you use to settle on your workstation to ssh before using this command:

- ssh turltebot@192.168.0.100

Then start the minimal launch by using:

- $ roslaunch turtlebot_le2i remap_rplidar_minimal.launch

The turtlebot_le2i package store the calibration parameters of our Turtlebot2.

Next step launch the gmapping file by using

- $ roslaunch rbx1_nav gmapping_demo.launch

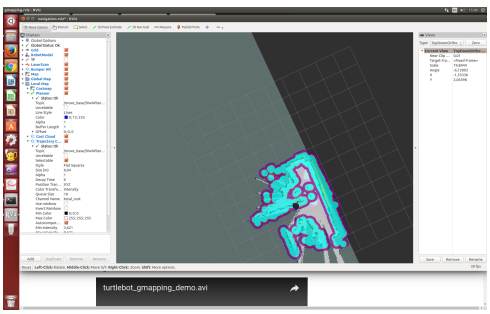

Figure 3.1: Gmapping and Teleop

- $ rosrun rviz rviz -d ‘rospack find rbx1_nav‘/gmapping.rviz

Last command use a teleop node, we uses the joystick for our case to be able to drive around the desired map the Turtlebot2.

- $ roslaunch rbx1_nav joystick_teleop.launch

Be careful to configure properly your joystick before using the command above refereing to this link Joystick configuration where you will find how to configure the joystick.

Final step, start the recording process by recording the data to a bag files using the command below:

- $ roscd rbx1_nav/bag_files

and start to record with:

- $ rosbag record - 0 my_scan_data /scan/tf

The data we need to record is the scan data and the tf transformation. After that we’ve execute those command we was able to drive the robots around the map that we wanted to create . Being careful to drive slowly.

The collecting and recording data proccess is done, to save the map use:

- $ roscd rbx1_nav/map
- $ rosrun map_server map_server - f my_map2 (which is the name of our map)

Those command created two new files under rbx1_nav/maps directory which is my_map2.pgm a picture of the map and my_map2.yaml that describes the dimensions of the map. This two files will be used for the map naviagtion.

In the picture below the result of our map. Refer to Gmapping + Teleop for the video of this manipulation

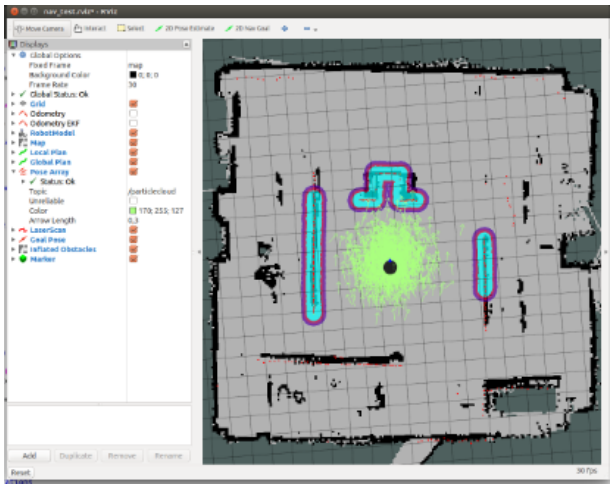

Figure 3.2: Navigating a square

3.2 Goal 2: Navigation & Localization

We navigate with our robot using the map that we've created previously.

3.2.1 Design:

The amcl package is used by ROS to localize the robot within an existing map using the file my_map2.yaml, my_map2 is the name we choose for our map.

3.2.2 Implementation:

To start the navigation of the Turtlebot2 regarding the created map use the command below:

- $ roslaunch turtlebot_le2i remap_rplidar_minimal.launch
- $ roslaunch rbx1_nav tb_demo_amcl.launch map:=my_map2.yaml
- $ rosrun rviz rviz -d 'rospack find rbx1_nav'/nav_test.rviz

3.2.3 Demonstration:

When the amcl start up, we find to different way to set the initial pose.

First method: using RVIZ tools Using the 2D Pose Estimate RViz button click on it and choose the point in the map where you your robot go, and use the button 2D Nav Goal for the navigation of the robot. On the terminal when you use the 2D Nav Goal you will have the coordinate of the postion and orientation of the robot.

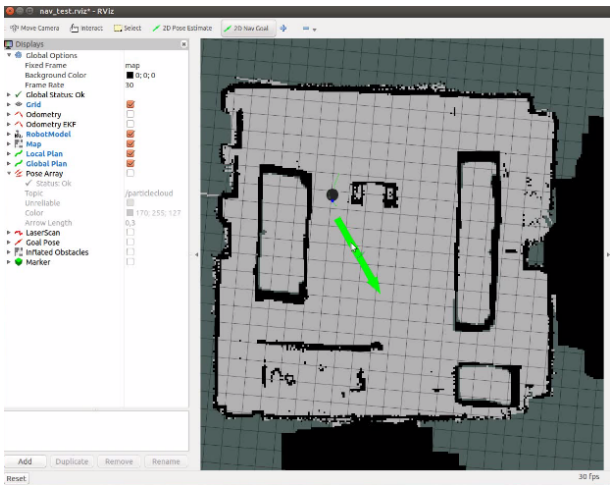

Figure 3.3: Navigating using the RVIz tools

Refer to Navigation using RVIz for the video of this manipulation

Second method: Autonomous navigation This method is the approach we choose to make the robot move autonomously without using any tools in RVIz. In appendix 5 the code that we used for that method: nav_test.py

Explanation: You can observe new locations, we put in comment the locations by default and we add our new locations that we bring using the tools in RVIz to recover the coordinates of each point our robot have to go. Also we put in comment the line to use the command in RVIz to set the initial pose. Our robot is able to move autonomously regarding the coordinate that we add in the nav_test.py.

Chapter 4

Part 4: Robotic Arm

4.1 PhantomX Pincher Robot Arm

The PhantomX Pincher is a robotic arm designed by Trossen robotics. The phantomX Pincher programmable robotic arm is a robotic kit that allows you to perform basic tasks like grabbing, moving and flipping of small objects in your environment. It is a 5 Degrees of freedom robotic arm consisting of 5 joints which allows rotating, lifting or moving of objects. It is equipped with 5 Dynamixel AX-12A servomotors which enables each motor respond to specific command assign to it. This hardware kit comes with everything needed to physically assemble and mount the arm as a standalone unit or as an addition to your TurtleBot robot platform.

The PhantomX Pincher is also designed to complement the TurtleBot and TurtleBot 2 robotic kits, enabling you to add a new possibility for interaction with your environment to your programmable robot. It is less expensive compared to other types of robotics arm. For more details please refer to our previous report Technical Survey Report

4.2 Application

In the youtube video is the application that we was able to do with our Robot Arm. We check our Robot using Pincher Test. Here the video of our robot control by the command arbotix_gui.

Figure 4.1: PhantomX Robot Pincher Arm

Chapter 5

Errors encountered

1. Error about the configuration not established properly

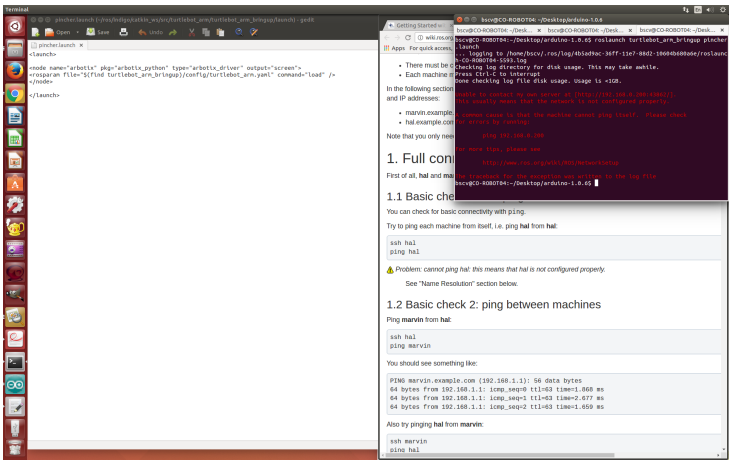

Figure 5.1: Configuration error

2. Error about the package building

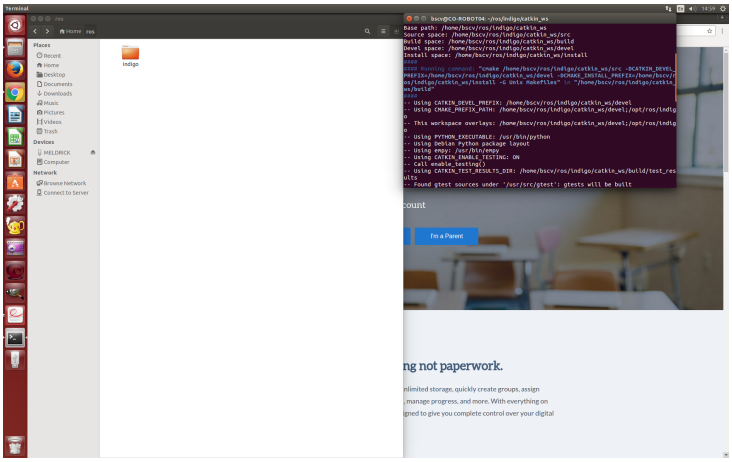

Figure 5.2: Building package error

Chapter 6

Conclusion

We will conclude by saying that we have been able to achieve our goals for this report. We were not able to give more applications with the Robotic arm but was able to work more on the turtlebot. We will not just stop here but continue in finding applications to which we can achieve with the robotic arm. We made great achievements with the trutlebot2 and enjoyed the friendship it gave us. We faced many difficulties but with time and dedication we were able to solve each that came our way.

REFERENCES

1. **ROS by Example Volume 1, R. Patrick Goebel**
2. **ROS by Example Volume 2, R. Patrick Goebel**
3. **Configuration of the Joystick**
   http://wiki.ros.org/joy/Tutorials/ConfiguringALinuxJoystick
4. **RPLIDAR**
   http://wiki.ros.org/rplidar

VIDEOS

Link of our videos on Youtube

- Part 1: Motion control, Basic Motion of mobile base.
  https://www.youtube.com/watch?v=aXN8vrgKaVU
- Part 1: Motion control, Advance Motion of mobile base.
  https://www.youtube.com/watch?v=bmieAum3SIY&feature=youtu.be
- Part 1: Motion control, Navigating a Square.
  https://www.youtube.com/watch?v=nubviqXMenI
- Part 1: Motion control, Navigation Path Planning.
  https://www.youtube.com/watch?v=XnruzcV3kQw&feature=youtu.be
- Part 3: Navigation & Localization, Map Building
  https://www.youtube.com/watch?v=GcMuJutT3g0
- Part 3: Navigation & Localization, Navigation with first method
  https://www.youtube.com/watch?v=IE1HGP_NTLA
- Part 4: Robot Arm, Application using the arbotix_gui command
  https://www.youtube.com/watch?v=U1I5eNihs9w
- Part 4: Robot Arm, Application of the Pincher Test
  https://www.youtube.com/watch?v=z6gMaZt2cHk

APPENDIX

APPENDIX 1

Script: timed_out_and_back.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from math import pi

class OutAndBack():
    def __init__(self):
        # Give the node a name
        rospy.init_node('out_and_back', anonymous=False)

        # Set rospy to execute a shutdown function when exiting
        rospy.on_shutdown(self.shutdown)

        # Publisher to control the robot's speed
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=1)

        # How fast will we update the robot's movement?
        rate = 50

        # Set the equivalent ROS rate variable
        r = rospy.Rate(rate)

        # Set the forward linear speed to 0.2 meters per second
        linear_speed = 0.2

        # Set the travel distance to 1.0 meters
        goal_distance = 1.0

        # How long should it take us to get there?
        linear_duration = goal_distance / linear_speed

        # Set the rotation speed to 1.0 radians per second
        angular_speed = 1.0

        # Set the rotation angle to Pi radians (180 degrees)
        goal_angle = pi

        # How long should it take to rotate?
        angular_duration = goal_angle / angular_speed

        # Loop through the two legs of the trip
        for i in range(2):
            # Initialize the movement command
            move_cmd = Twist()

            # Set the forward speed
            move_cmd.linear.x = linear_speed
```

```
            # Move forward for a time to go the desired distance
            ticks = int(linear_duration * rate)

            for t in range(ticks):
                self.cmd_vel.publish(move_cmd)
                r.sleep()

            # Stop the robot before the rotation
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

            # Now rotate left roughly 180 degrees

            # Set the angular speed
            move_cmd.angular.z = angular_speed

            # Rotate for a time to go 180 degrees
            ticks = int(goal_angle * rate)

            for t in range(ticks):
                self.cmd_vel.publish(move_cmd)
                r.sleep()

            # Stop the robot before the next leg
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

        # Stop the robot
        self.cmd_vel.publish(Twist())

    def shutdown(self):
        # Always stop the robot when shutting down the node.
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        OutAndBack()
    except:
        rospy.loginfo("Out-and-Back node terminated.")
```

sanstitre6.py

APPENDIX 2

Script: odom_out_and_back.py

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist, Point, Quaternion
import tf
from rbx1_nav.transform_utils import quat_to_angle, normalize_angle
from math import radians, copysign, sqrt, pow, pi

class OutAndBack():
    def __init__(self):
        # Give the node a name
        rospy.init_node('out_and_back', anonymous=False)

        # Set rospy to execute a shutdown function when exiting
        rospy.on_shutdown(self.shutdown)

        # Publisher to control the robot's speed
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)

        # How fast will we update the robot's movement?
        rate = 20

        # Set the equivalent ROS rate variable
        r = rospy.Rate(rate)

        # Set the forward linear speed to 0.15 meters per second
        linear_speed = 0.15

        # Set the travel distance in meters
        goal_distance = 1.0

        # Set the rotation speed in radians per second
        angular_speed = 0.5

        # Set the angular tolerance in degrees converted to radians
        angular_tolerance = radians(1.0)

        # Set the rotation angle to Pi radians (180 degrees)
        goal_angle = pi

        # Initialize the tf listener
        self.tf_listener = tf.TransformListener()

        # Give tf some time to fill its buffer
        rospy.sleep(2)

        # Set the odom frame
        self.odom_frame = '/odom'

        # Find out if the robot uses /base_link or /base_footprint
        try:
            self.tf_listener.waitForTransform(self.odom_frame, '/base_footprint',
    rospy.Time(), rospy.Duration(1.0))
            self.base_frame = '/base_footprint'
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            try:
                self.tf_listener.waitForTransform(self.odom_frame, '/base_link',
    rospy.Time(), rospy.Duration(1.0))
                self.base_frame = '/base_link'
            except (tf.Exception, tf.ConnectivityException, tf.LookupException):
                rospy.loginfo("Cannot find transform between /odom and /base_link or
```

```
    /base_footprint")
                rospy.signal_shutdown("tf Exception")

        # Initialize the position variable as a Point type
        position = Point()

        # Loop once for each leg of the trip
        for i in range(2):
            # Initialize the movement command
            move_cmd = Twist()

            # Set the movement command to forward motion
            move_cmd.linear.x = linear_speed

            # Get the starting position values
            (position, rotation) = self.get_odom()

            x_start = position.x
            y_start = position.y

            # Keep track of the distance traveled
            distance = 0

            # Enter the loop to move along a side
            while distance < goal_distance and not rospy.is_shutdown():
                # Publish the Twist message and sleep 1 cycle
                self.cmd_vel.publish(move_cmd)

                r.sleep()

                # Get the current position
                (position, rotation) = self.get_odom()

                # Compute the Euclidean distance from the start
                distance = sqrt(pow((position.x - x_start), 2) +
                                pow((position.y - y_start), 2))

            # Stop the robot before the rotation
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

            # Set the movement command to a rotation
            move_cmd.angular.z = angular_speed

            # Track the last angle measured
            last_angle = rotation

            # Track how far we have turned
            turn_angle = 0

            while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not
    rospy.is_shutdown():
                # Publish the Twist message and sleep 1 cycle
                self.cmd_vel.publish(move_cmd)
                r.sleep()

                # Get the current rotation
                (position, rotation) = self.get_odom()

                # Compute the amount of rotation since the last loop
                delta_angle = normalize_angle(rotation - last_angle)
```

```
                # Add to the running total
                turn_angle += delta_angle
                last_angle = rotation

            # Stop the robot before the next leg
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1)

        # Stop the robot for good
        self.cmd_vel.publish(Twist())

    def get_odom(self):
        # Get the current transform between the odom and base frames
        try:
            (trans, rot)  = self.tf_listener.lookupTransform(self.odom_frame, self.
    base_frame, rospy.Time(0))
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            rospy.loginfo("TF Exception")
            return

        return (Point(*trans), quat_to_angle(Quaternion(*rot)))

    def shutdown(self):
        # Always stop the robot when shutting down the node.
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        OutAndBack()
    except:
        rospy.loginfo("Out-and-Back node terminated.")
```

odom.py

APPENDIX 3

Script: nav_square.py

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

import rospy
from geometry_msgs.msg import Twist, Point, Quaternion
import tf
from rbx1_nav.transform_utils import quat_to_angle, normalize_angle
from math import radians, copysign, sqrt, pow, pi

class NavSquare():
    def __init__(self):
        # Give the node a name
        rospy.init_node('nav_square', anonymous=False)

        # Set rospy to execute a shutdown function when terminating the script
        rospy.on_shutdown(self.shutdown)

        # How fast will we check the odometry values?
        rate = 20

        # Set the equivalent ROS rate variable
        r = rospy.Rate(rate)

        # Set the parameters for the target square
        goal_distance = rospy.get_param("~goal_distance", 1.0)       # meters
        goal_angle = radians(rospy.get_param("~goal_angle", 90))     # degrees
    converted to radians
        linear_speed = rospy.get_param("~linear_speed", 0.2)         # meters per
    second
        angular_speed = rospy.get_param("~angular_speed", 0.7)       # radians per
    second
        angular_tolerance = radians(rospy.get_param("~angular_tolerance", 2)) #
    degrees to radians

        # Publisher to control the robot's speed
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)

        # The base frame is base_footprint for the TurtleBot but base_link for Pi
    Robot
        self.base_frame = rospy.get_param('~base_frame', '/base_link')

        # The odom frame is usually just /odom
        self.odom_frame = rospy.get_param('~odom_frame', '/odom')

        # Initialize the tf listener
        self.tf_listener = tf.TransformListener()

        # Give tf some time to fill its buffer
        rospy.sleep(2)

        # Set the odom frame
        self.odom_frame = '/odom'

        # Find out if the robot uses /base_link or /base_footprint
        try:
            self.tf_listener.waitForTransform(self.odom_frame, '/base_footprint',
    rospy.Time(), rospy.Duration(1.0))
            self.base_frame = '/base_footprint'
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            try:
                self.tf_listener.waitForTransform(self.odom_frame, '/base_link',
```

```
    rospy.Time(), rospy.Duration(1.0))
                self.base_frame = '/base_link'
            except (tf.Exception, tf.ConnectivityException, tf.LookupException):
                rospy.loginfo("Cannot find transform between /odom and /base_link or
     /base_footprint")
                rospy.signal_shutdown("tf Exception")

        # Initialize the position variable as a Point type
        position = Point()

        # Cycle through the four sides of the square
        for i in range(4):
            # Initialize the movement command
            move_cmd = Twist()

            # Set the movement command to forward motion
            move_cmd.linear.x = linear_speed

            # Get the starting position values
            (position, rotation) = self.get_odom()

            x_start = position.x
            y_start = position.y

            # Keep track of the distance traveled
            distance = 0

            # Enter the loop to move along a side
            while distance < goal_distance and not rospy.is_shutdown():
                # Publish the Twist message and sleep 1 cycle
                self.cmd_vel.publish(move_cmd)

                r.sleep()

                # Get the current position
                (position, rotation) = self.get_odom()

                # Compute the Euclidean distance from the start
                distance = sqrt(pow((position.x - x_start), 2) +
                                pow((position.y - y_start), 2))

            # Stop the robot before rotating
            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1.0)

            # Set the movement command to a rotation
            move_cmd.angular.z = angular_speed

            # Track the last angle measured
            last_angle = rotation

            # Track how far we have turned
            turn_angle = 0

            # Begin the rotation
            while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not
    rospy.is_shutdown():
                # Publish the Twist message and sleep 1 cycle
                self.cmd_vel.publish(move_cmd)

                r.sleep()
```

```
                # Get the current rotation
                (position, rotation) = self.get_odom()

                # Compute the amount of rotation since the last lopp
                delta_angle = normalize_angle(rotation - last_angle)

                turn_angle += delta_angle
                last_angle = rotation

            move_cmd = Twist()
            self.cmd_vel.publish(move_cmd)
            rospy.sleep(1.0)

        # Stop the robot when we are done
        self.cmd_vel.publish(Twist())

    def get_odom(self):
        # Get the current transform between the odom and base frames
        try:
            (trans, rot)  = self.tf_listener.lookupTransform(self.odom_frame, self.
    base_frame, rospy.Time(0))
        except (tf.Exception, tf.ConnectivityException, tf.LookupException):
            rospy.loginfo("TF Exception")
            return

        return (Point(*trans), quat_to_angle(Quaternion(*rot)))

    def shutdown(self):
        # Always stop the robot when shutting down the node
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        NavSquare()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation terminated.")
```

square.py

APPENDIX 4

Script: move_base_square.py

```
#!/usr/bin/env python
import rospy
import actionlib
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, Point, Quaternion, Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from tf.transformations import quaternion_from_euler
from visualization_msgs.msg import Marker
from math import radians, pi

class MoveBaseSquare():
    def __init__(self):
        rospy.init_node('nav_test', anonymous=False)

        rospy.on_shutdown(self.shutdown)

        # How big is the square we want the robot to navigate?
        square_size = rospy.get_param("~square_size", 1.0) # meters

        # Create a list to hold the target quaternions (orientations)
        quaternions = list()

        # First define the corner orientations as Euler angles
        euler_angles = (pi/2, pi, 3*pi/2, 0)

        # Then convert the angles to quaternions
        for angle in euler_angles:
            q_angle = quaternion_from_euler(0, 0, angle, axes='sxyz')
            q = Quaternion(*q_angle)
            quaternions.append(q)

        # Create a list to hold the waypoint poses
        waypoints = list()

        # Append each of the four waypoints to the list.  Each waypoint
        # is a pose consisting of a position and orientation in the map frame.
        waypoints.append(Pose(Point(square_size, 0.0, 0.0), quaternions[0]))
        waypoints.append(Pose(Point(square_size, square_size, 0.0), quaternions[1]))
        waypoints.append(Pose(Point(0.0, square_size, 0.0), quaternions[2]))
        waypoints.append(Pose(Point(0.0, 0.0, 0.0), quaternions[3]))

        # Initialize the visualization markers for RViz
        self.init_markers()

        # Set a visualization marker at each waypoint
        for waypoint in waypoints:
            p = Point()
            p = waypoint.position
            self.markers.points.append(p)

        # Publisher to manually control the robot (e.g. to stop it, queue_size=5)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=5)

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

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

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

```
        rospy.loginfo("Connected to move base server")
        rospy.loginfo("Starting navigation test")

        # Initialize a counter to track waypoints
        i = 0

        # Cycle through the four waypoints
        while i < 4 and not rospy.is_shutdown():
            # Update the marker display
            self.marker_pub.publish(self.markers)

            # Intialize the waypoint goal
            goal = MoveBaseGoal()

            # Use the map frame to define goal poses
            goal.target_pose.header.frame_id = 'map'

            # Set the time stamp to "now"
            goal.target_pose.header.stamp = rospy.Time.now()

            # Set the goal pose to the i-th waypoint
            goal.target_pose.pose = waypoints[i]

            # Start the robot moving toward the goal
            self.move(goal)

            i += 1

    def move(self, goal):
            # Send the goal pose to the MoveBaseAction server
            self.move_base.send_goal(goal)

            # Allow 1 minute to get there
            finished_within_time = self.move_base.wait_for_result(rospy.Duration(60)
)

            # If we don't get there in time, abort the goal
            if not finished_within_time:
                self.move_base.cancel_goal()
                rospy.loginfo("Timed out achieving goal")
            else:
                # We made it!
                state = self.move_base.get_state()
                if state == GoalStatus.SUCCEEDED:
                    rospy.loginfo("Goal succeeded!")

    def init_markers(self):
        # Set up our waypoint markers
        marker_scale = 0.2
        marker_lifetime = 0 # 0 is forever
        marker_ns = 'waypoints'
        marker_id = 0
        marker_color = {'r': 1.0, 'g': 0.7, 'b': 1.0, 'a': 1.0}

        # Define a marker publisher.
        self.marker_pub = rospy.Publisher('waypoint_markers', Marker, queue_size=5)

        # Initialize the marker points list.
        self.markers = Marker()
        self.markers.ns = marker_ns
        self.markers.id = marker_id
        self.markers.type = Marker.CUBE_LIST
```

```
        self.markers.action = Marker.ADD
        self.markers.lifetime = rospy.Duration(marker_lifetime)
        self.markers.scale.x = marker_scale
        self.markers.scale.y = marker_scale
        self.markers.color.r = marker_color['r']
        self.markers.color.g = marker_color['g']
        self.markers.color.b = marker_color['b']
        self.markers.color.a = marker_color['a']

        self.markers.header.frame_id = 'odom'
        self.markers.header.stamp = rospy.Time.now()
        self.markers.points = list()

    def shutdown(self):
        rospy.loginfo("Stopping the robot...")
        # Cancel any active goals
        self.move_base.cancel_goal()
        rospy.sleep(2)
        # Stop the robot
        self.cmd_vel_pub.publish(Twist())
        rospy.sleep(1)

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

movebasesquare.py

6.1 APPENDIX 5

Script: nav_test.py

```
#!/usr/bin/env python
import rospy
import actionlib
from actionlib_msgs.msg import *
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped, Point, Quaternion,
    Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from random import sample
from math import pow, sqrt

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

        rospy.on_shutdown(self.shutdown)

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

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

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

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

        #coordinate for our approach of the autonomous navigation
        locations['pos1'] = Pose(Point(-1.415, 0.834, 0.000), Quaternion(0.000,
    0.000, -0.263, 0.965))
        locations['pos2'] = Pose(Point(-0.803, 2.365, 0.000), Quaternion(0.000,
    0.000, 0.974, 0.226))
        locations['pos3'] = Pose(Point(-3.388, 3.632, 0.000), Quaternion(0.000,
    0.000, 0.868, -0.497))
        locations['pos4'] = Pose(Point(-4.625, 1.055, 0.000), Quaternion(0.000,
    0.000, -0.223, 0.975))
        locations['pos5'] = Pose(Point(-2.301, 0.123, 0.000), Quaternion(0.000,
    0.000, -0.149, 0.989))
        locations['pos6'] = Pose(Point(-1.173, 1.053, 0.000), Quaternion(0.000,
    0.000, 0.985, 0.172))
        #locations['pos7'] = Pose(Point(-2.557, 1.505, 0.000), Quaternion(0.000,
    0.000, 0.976, 0.218))

        # coordinate by default
        #locations['hall_foyer'] = Pose(Point(0.643, 4.720, 0.000), Quaternion
    (0.000, 0.000, 0.223, 0.975))
        #locations['hall_kitchen'] = Pose(Point(-1.994, 4.382, 0.000), Quaternion
    (0.000, 0.000, -0.670, 0.743))
        #locations['hall_bedroom'] = Pose(Point(-3.719, 4.401, 0.000), Quaternion
    (0.000, 0.000, 0.733, 0.680))
        #locations['living_room_1'] = Pose(Point(0.720, 2.229, 0.000), Quaternion
    (0.000, 0.000, 0.786, 0.618))
```

```
    #locations['living_room_2'] = Pose(Point(1.471, 1.007, 0.000), Quaternion
(0.000, 0.000, 0.480, 0.877))
    #locations['dining_room_1'] = Pose(Point(-0.861, -0.019, 0.000), Quaternion
(0.000, 0.000, 0.892, -0.451))

    # Publisher to manually control the robot (e.g. to stop it, queue_size=5)
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=5)

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

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

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

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

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

    # Variables to keep track of success rate, running time,
    # and distance traveled
    n_locations = len(locations)
    n_goals = 0
    n_successes = 0
    i = n_locations
    distance_traveled = 0
    start_time = rospy.Time.now()
    running_time = 0
    location = ""
    last_location = ""

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

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

    rospy.loginfo("Starting navigation test")

    # Begin the main loop and run through a sequence of locations
    while not rospy.is_shutdown():
        # If we've gone through the current sequence,
        # start with a new random sequence
        if i == n_locations:
            i = 0
            sequence = sample(locations, n_locations)
            # Skip over first location if it is the same as
            # the last location
            if sequence[0] == last_location:
                i = 1

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

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

            # Store the last location for distance calculations
            last_location = location

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

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

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

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

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

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

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

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

```
    ")
            rospy.sleep(self.rest_time)

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

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

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

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

navtest.py