BACHELOR IN COMPUTER VISION AND ROBOTICS
ROBOTICS ENGINEERING II
PROJECT REPORT
"TURTLEBOT DRIVING FROM ANDROID AND MATLAB"

PRESENTED BY:

YAMID ESPINEL

PANKAJ BAGGA

UNIVERSITE DE BOURGOGNE
LE CREUSOT, FRANCE
2016

Table of contents

PROJECT REPORT
"TURTLEBOT DRIVING FROM ANDROID AND MATLAB"

INTRODUCTION

The ROS development and control libraries are nowadays being ported to almost every existing platform, which lets a system powered with it to be monitored and controlled from numerous devices and places such as mobile devices and remote websites. However, these libraries generally come with almost null documentation, making their reutilization to be a very tedious and lengthy process which normally translates into the impossibility of applying them.

Driving and monitoring a ROS-based system from a mobile device is more accessible and convenient than from a conventional workstation, making it worth to implement a full-featured Android application from which the user can perform the same operations that could be done from a workstation, such as:

- Teleoperation driving by using joystick or keyboard
- Color camera streaming
- Linear camera readings
- Map streaming
- Path planning with pose and goal setting.

In order for the app to have these functionalities, it must be able connect to the ROS environment of the device under control, receive the data sent by it and display the processed information to the user, and capture the different gestures to send them through the corresponding topics to the robot, making it to displace to the desired position.

The ROS team has developed a series of libraries called "RosJava", which lets developers to create ROS-based applications in Java environments such as Android. These libraries include image rendering, map viewer, custom node configuration, network configuration, among others, all of which are very poor documented. Consequently, the main objective of this project is to take advantage of some of these useful libraries to build an application which can show the global position of a Turtlebot in a mapped environment, set an initial pose and move it to a desired location while avoiding the obstacles in its way, show the streaming of the Kinect's camera, and manually drive the robot by using either a joystick or a "keyboard", all while explaining in detail the process followed to make the application.

INSTALLATION OF THE ROSJAVA ENVIRONMENT AND THE ANDROID DEVELOPMENT PLATFORM

The RosJava set of libraries is installed via console and using the deb packages available from Ubuntu's repositories. We follow the procedure indicated in http://wiki.ros.org/android/Tutorials/hydro/Installation%20-%20Ros%20Development%20Environment, which will download all the sources for the Rosjava core libraries.

Following the installation of the Rosjava libraries, we proceed to download the Android Studio software, taking care about the version to install. The Rosjava libraries for the ROS Hydro environment depend on the Android Gradle Plugin v0.12.2, which can only be used with Android Studio v0.8.9 (according to http://tools.android.com/tech-docs/new-build-system/version-compatibility). The installation procedure is done by following the instructions in http://wiki.ros.org/android/Android%20Studio/Download.

CREATION OF THE APPLICATION

INTERFACE DEFINITION:

After creating the new "android_turtlebot" project inside the *android_core* environment (installed in the previous section), the interface is designed and implemented. As long as the application must be able to perform teleoperation with keyboard and joystick, along with camera viewing, mapping and goal posing, the following items should be added:

- Buttons to make the robot move forward, backward and turn right or left.
- Joystick widget that lets the robot to move front/back and turn at the same time, and at variable speed.
- Camera viewer to show the image broadcasted by the Kinect's RGB Camera.
- Map widget in charge of showing the environment map, the position of the robot, and marking initial and goal poses.
- Selection widgets to tell the robot if the pose being marked in the map widget is an initial one or a goal one.

Then we proceed to add the necessary libraries into the interface. It's arranged in such way that it's easy for the user to manipulate the controls and to have a clear view of what's happening with the robot. The interface is organized as follows:

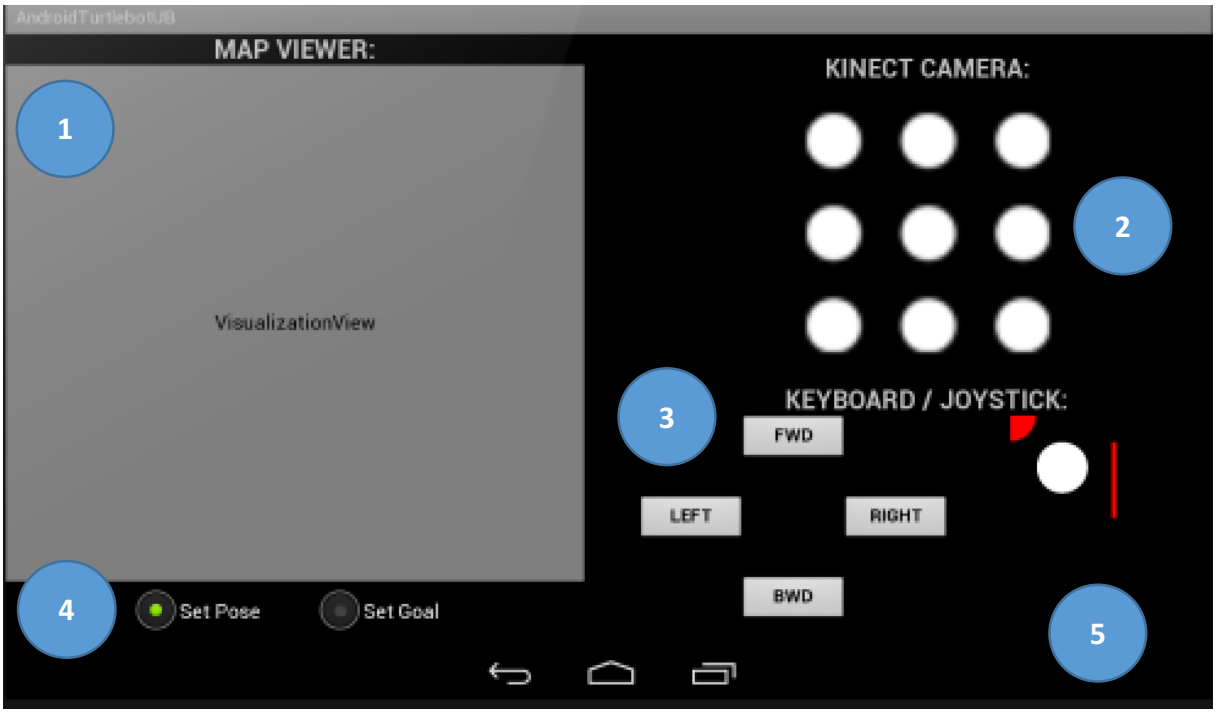


Figure: Interface of the application

In the main interface we can find the following components:

1) Mapping widget: It's a control of type *org.ros.android.view.visualization.VisualizationView,* which is included in the Rosjava libraries. Here it will show the map loaded in the robot, the robot's position, and also it will capture the gestures made by the user and translate them into actual pose orientations and locations.
2) Camera widget: Depends on the class *org.ros.android.view.RosImageView* and it's in charge of showing the images sent by the Kinect's RGB camera.
3) Keyboard teleoperation: These make the robot to move forward/backward and to rotate left/right at a constant velocity.
4) Set Pose or Goal selectors: They let the application know if the pose being currently set is an initial one or a final goal.
5) Joystick control: It's an object from the class *com.zerokol.views.JoystickView* (an independent library not included in Rosjava). It moves according to the position of the finger and returns the corresponding values of angle (in degrees), power (in percentage) and direction.

CODING:

A .java file for the main interface is created, under the name *MainActivity.java*. It contains all the procedures to make the widgets inside the window to actually control the robot, by means of publishing the commands generated by them into the corresponding topics.

Libraries to be used:

We start by adding the libraries derived from the Android environment. Line by line we explain the purpose of each library:

```
import android.os.Bundle; //General operating system dependency
import  android.view.MotionEvent;  //Library  used  for  capturing  the  touch
                                    movements on the teleoperation buttons
import  android.view.View;  //Library  used  to  extract  and  use  the  widgets'
                            properties
import  android.widget.Button;  //Button  library  used  to  register  the  touch
                                events for the keyboard teleoperation
```

Now come the ROS libraries required to communicate the application with the robot:

```
import org.ros.android.BitmapFromCompressedImage; //Used to read the image
                                                    data  sent  by  the  robot
                                                    and convert it to bitmap
import org.ros.android.RosActivity; //It's the type of activity that every
                                    Android  ROS-based  application  should
                                    use for its interfaces
import org.ros.node.NodeConfiguration; //Library required for setting node
                                        parameters before publishing them
import  org.ros.node.NodeMainExecutor;  //Library  in  charge  of  running  the
                                        desired   nodes   according   the
                                        configurations      made      with
                                        org.ros.node.NodeConfiguration
import org.ros.concurrent.CancellableLoop; //It lets to create a loop inside
                                            a  node  declaration  that  will
                                            stop if the node shuts down
import  org.ros.namespace.GraphName;  //Used  to  give  the  node  an  identifier
                                    name
import  org.ros.node.AbstractNodeMain;  //Brings  the  general  structure  to
                                        declare a ROS node in Java
import      org.ros.node.ConnectedNode;      //Comes      along      with
                                               org.ros.node.AbstractNodeMa
                                               in. Used for defining among
                                               other the name of the topic
                                               where  the  node  will  be
                                               publishing and the type of
                                               the generated message
import org.ros.node.topic.Publisher; //Serves as the method that will receive
                                    a message and publish it
import  org.ros.android.view.visualization.VisualizationView;  //Library  to
                                    access the mapping widget properties
import         org.ros.android.view.visualization.layer.CameraControlLayer;
                                    //Brings  the  possibility  of  adding
                                    viewing  gestures  to  the  currently
                                    shown map like rotation, translation
                                    and zooming
```

```
import org.ros.android.view.visualization.layer.LaserScanLayer; //Makes the
                                        map widget to show the Lidar readings
import         org.ros.android.view.visualization.layer.OccupancyGridLayer;
                                        //Provides the layer where the map is
                                        shown
import org.ros.android.view.visualization.layer.PathLayer; //Adds a layer in
                                        the map widget which shows the
                                        current path planned by the robot
import        org.ros.android.view.visualization.layer.PoseSubscriberLayer;
                                        //Retrieves the current goal pose for
                                        the robot
import  org.ros.android.view.visualization.layer.RobotLayer;  //Shows  the
robot's position and orientation inside the map widget.
import  org.ros.android.view.RosImageView;    //Library  that  let's  the
                                        application to display the images
                                        sent by a camera from the robot
import  geometry_msgs.Twist;  //Provides  the  ability  to  create  twist-type
                                    messages
```

The libraries header continues with the inclusion of those required for performing control operations according to the gestures captured by the joystick:

```
import com.zerokol.views.JoystickView;
import com.zerokol.views.JoystickView.OnJoystickMoveListener;
```

Node definition for manual teleoperation:

The first and most essential thing we should understand is how to declare a ROS node in Java by using the Rosjava libraries. That's why we begin with the implementation of the node that will send the velocities chosen by the user to manually drive the robot.

Declaring a node consists basically on redefining the abstract class *AbstractNodeMain.* The *Talker* node declaration along with the variables to be used by it is done as follows:

```
class Talker extends AbstractNodeMain {

    public double xVel,yVel,zVel; //Variables that contain linear velocities
    public   double   xAng,yAng,zAng;   //Variables   that   contain   angular
                                        velocities
    public  boolean  message_enable=false;  //Flag  to  enable/disable  the
                                        publishing  of  twist  messages  and
                                        avoid  interference  with  the  path
                                        following routine
    private Twist velCommand; //Message of type twist that will contain the
                                    velocities to be published
```

Now, we can define the node identifier with which the robot will be able to recognize the precedence of the published information, like:

```
    @Override
    public GraphName getDefaultNodeName() {
        return   GraphName.of("rosjava_android_turtlebot/talker");   //Node
identifier
    }
```

In this *Talker* class (derived from the abstract class *AbstractNodeMain*) there exists the *onStart()* function that will be invoked every time the class is initialized. Practically, here is where we define all the node's behavior, being composed at the same time by:

1) The *setup()* function, that will initialize the variables declared previously.
2) The *loop()* function, which will run indefinitely and will contain the procedures to keep publishing messages into a specific topic.

Taking the previous structure into account, we override the *onStart()* function in the *Talker* class as follows:

```
@Override
    public void onStart(final ConnectedNode connectedNode) {
        final Publisher<geometry_msgs.Twist> pub =
                connectedNode.newPublisher("cmd_vel",
                 geometry_msgs.Twist._TYPE); //Publisher definition with its
                 topic name and the type of message
        // This CancellableLoop will be canceled automatically when the node
           shuts down.

        connectedNode.executeCancellableLoop(new CancellableLoop() {

            @Override
            //Initialise all the variables:
            protected void setup() {
                xVel = 0;
                yVel = 0;
                zVel = 0;
                xAng = 0;
                yAng = 0;
                zAng = 0;
            }

            @Override
            //Main execution loop:
            protected void loop() throws InterruptedException {

                if(message_enable) {
                    velCommand  =  pub.newMessage();  //Create  new  twist
                                                        message
                    velCommand.getLinear().setX(xVel);     //Set     linear
                                                             velocities
                    velCommand.getLinear().setY(yVel);
                    velCommand.getLinear().setZ(zVel);
                    velCommand.getAngular().setX(xAng);    //Set    angular
                                                             velocities
```

```
                    velCommand.getAngular().setY(yAng);
                    velCommand.getAngular().setZ(zAng);
                    pub.publish(velCommand); //Publish the twist message
                    Thread.sleep(100);  //Do a delay of 100ms to let the
                                        message to be captured by the robot
                }

            }
        });
    }
```

Implementation of the main interface class:

Once we have defined the node in charge of sending the twist messages for the manual teleoperation, we proceed with the declaration of the main activity class and all the procedures that will define the behavior of the elements inside the window. As long as we are implementing an application that will communicate remotely with a ROS-powered device, we should declare the main interface as a redefinition of the class *RosActivity* and not of the common Android's *Activity* class. We implement this and also we declare the elements that conform the main interface, like:

```
public class MainActivity extends RosActivity {


    private Talker talker; // Define new twist publisher
    private JoystickView joystick; // Joystick control widget
    private VisualizationView visualizationView; //Map viewer widget
    private RosImageView<sensor_msgs.CompressedImage> image; //Camera widget
    private Button btnFwd; //Forward button
    private Button btnBwd; //Backward button
    private Button btnRight; //Right button
    private Button btnLeft; //Left button
    private MapPosePublisherLayer mapPosePublisherLayer; //Object to be used
                                        to capture the gestures from the
                                        map viewer and publish the
                                        corresponding goals to the robot

    public MainActivity() {
        // The RosActivity constructor configures the notification title and
        // ticker messages.
        super("Android Turtlebot", "Android Turtlebot");
    }
```

The next step is to implement the initialization procedures for the elements inside the window. Within the *onCreate()* function, derived from the abstract class *RosActivity*, it's possible to instantiate the widgets and link the different methods regarding the interactions between the user and widgets. Thus, the function along with the interface elements are declared as:

```
@SuppressWarnings("unchecked")
    @Override
    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // referring as others views
        joystick = (JoystickView) findViewById(R.id.joystickView);

        btnFwd = (Button)findViewById(R.id.buttonFWD);
        btnBwd = (Button)findViewById(R.id.buttonBWD);
        btnRight = (Button)findViewById(R.id.buttonRIGHT);
        btnLeft = (Button)findViewById(R.id.buttonLEFT);
```

After having instantiated all the widgets, we proceed to implement the methods that will capture and process the gestures made by the user over these elements. In first place, the interactions with the teleoperation buttons are defined in such way that, when the user presses one of them, it will change the corresponding twist velocity and then, when it's released, it will set that velocity again to zero.

For example, for the forward button we have:

```
btnFwd.setOnTouchListener(new View.OnTouchListener() {
            @Override
            public boolean onTouch(View v, MotionEvent event) {
                if(event.getAction() == MotionEvent.ACTION_DOWN) {

                    talker.xVel = 0.3;
                    talker.zAng = 0;
                    talker.message_enable=true;
                } else if (event.getAction() == MotionEvent.ACTION_UP) {
                    talker.xVel = 0;
                    talker.message_enable=false;

                }
                return true;
            }
        });
```

From the previous procedure we can see how when the forward button is pressed, the speed in the X axis is set to a positive value and the publisher is enabled, making the robot to move to the front; and then, when it's released, this velocity is set to zero and the publisher is disabled, letting other operations from the application to be performed without interference.

Once we have defined all the interactions for the teleoperation buttons, we do the same for the joystick control. For doing this, we must recall the way it provides information according to the positon of the small ball inside the general circle:

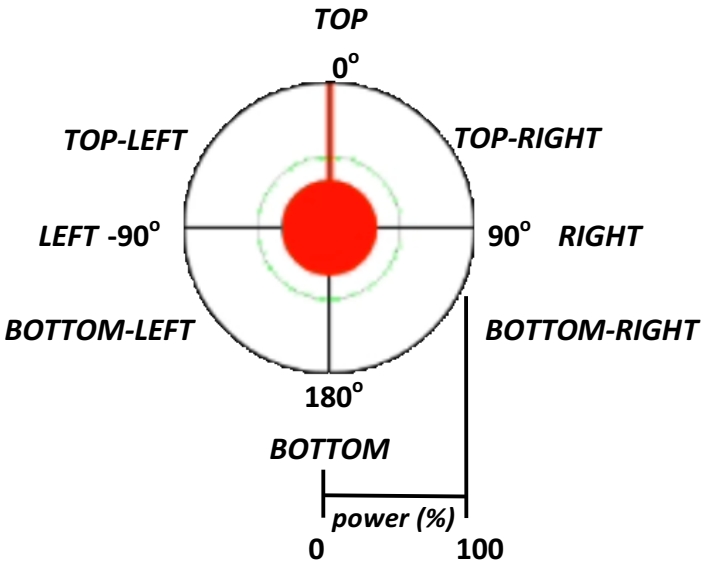


Figure: Joystick control and the three types of data it provides: Angle, Power and Direction

For our teleoperation control, we only need two informations from the joystick; the Angle and the Power. The procedure we implement to convert this data into actual linear angular speeds to make the robot move in the desired way, is:

```
joystick.setOnJoystickMoveListener(new OnJoystickMoveListener() {
            @Override
            public void onValueChanged(int angle, int power, int direction)
            {

                //Convert joystick values into actual robot velocities:

                if(angle==0 && power==0) {
                    talker.message_enable=false;
                }else {
                    talker.message_enable=true;
                    if (angle > 90 && angle < 180) {
                        //If it's inside the bottom-right quadrant:
                        talker.xVel = -(power / 100.0) / 4;
                        talker.zAng = (angle / 180.0 - 1) * 1.8;
                    } else if (angle < -90 && angle > -180) {
                        //If it's inside the bottom-left quadrant:
                        talker.xVel = -(power / 100.0) / 4;
                        talker.zAng = (angle / 180.0 + 1) * 1.8;
                    } else if (angle >= -90 && angle <= 90) {
                        //If it's inside the two top quadrants:
                        talker.xVel = (power / 100.0) / 4;
                        talker.zAng = -(angle / 90.0) / 1.2;
                    }
                }
            }
        }, JoystickView.DEFAULT_LOOP_INTERVAL);
```

In the previous method, the divisions performed in the conversion operations are selected through several experiments in such way that the robot could have enough responsiveness to user's movements and also for it to move at a safe velocity.

We can now begin to configure the widgets, by doing:

```
visualizationView = (VisualizationView) findViewById(R.id.visualization);
visualizationView.getCamera().setFrame("map"); //Set camera image as the map
                                               received from the robot
image             =             (RosImageView<sensor_msgs.CompressedImage>)
findViewById(R.id.image);
image.setTopicName("camera/rgb/image_color/compressed"); //Subscribe camera
                                               widget to the corresponding topic
image.setMessageType(sensor_msgs.CompressedImage._TYPE);     //Treat     the
                                               incoming data as compressed image
image.setMessageToBitmapCallable(new BitmapFromCompressedImage());
```

The next function to reimplement is called *init().* Inside it, we must finish configuring the widgets and then proceed to execute the nodes according to those parameters.

To be able to configure the map widget, first we must see the way it interacts with the user's gestures and how it sends the information to the robot. In the following picture, we can observe the information that it's able to display and how to interact with it to perform the operations we want:

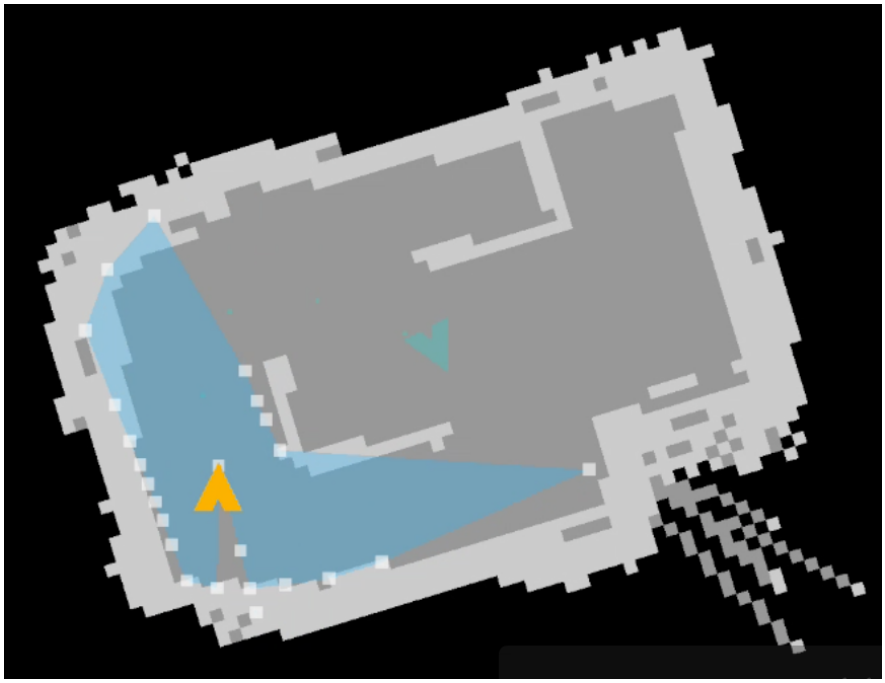


Figure: Map widget showing the robot, the environment map, the Lidar readings, the planned path and the final goal

Inside the previous figure, we can differentiate all the data shown in the widget. In gray colors we see the map loaded in the robot, the yellow arrow representing the actual position of the robot, the blue blue area around the robot showing the Lidar readings, the blue arrow that represents the final goal, and the small blue dots that go from the robot to the goal representing the path planned by the system.

The way the widget interacts with the user to receive commands can be divided in the two following gestures:

- Traslation, rotation and zooming: These ways of modifying the position of the map can be handled by using the common finger gestures used when handling a photo in any mobile application.
- Pose setting: To set an initial pose or a goal, the user must press the widget in the desired location for two seconds and then drag with the finger to rotate the resulting arrow to the desired orientation.

With the above behaviors in mind, we can now configure the corresponding layers of information for the map widget by telling to it which topics should subscribe or publish to effectively communicate with the robot:

```
@Override
protected void init(NodeMainExecutor nodeMainExecutor) {

    talker = new Talker();

    visualizationView.addLayer(new CameraControlLayer(this,
    nodeMainExecutor
            .getScheduledExecutorService()));
    visualizationView.addLayer(new OccupancyGridLayer("map"));
            //Subscribe to the map topic
    visualizationView.addLayer(new
    PathLayer("move_base/TrajectoryPlannerROS/global_plan"));
            //Retrieve the paths planned by the robot
    visualizationView.addLayer(new LaserScanLayer("scan")); //Retrieve
              the Lidar readings
    visualizationView.addLayer(new
    PoseSubscriberLayer("move_base_simple/goal")); //Subscribe to the
              current position of the robot
    mapPosePublisherLayer = new MapPosePublisherLayer(this);
    visualizationView.addLayer(mapPosePublisherLayer); //Create a new
              goal pose publisher
    visualizationView.addLayer(new
    InitialPoseSubscriberLayer("initialpose")); //Create a new initial
              pose publisher
    visualizationView.addLayer(new RobotLayer("base_link")); //Retrieve
              robot status
```

When all the widgets and node configurations have been made, we have to tell the system what is the IP address of the robot and also the local IP address, so that the application can send information to the robot and vice-versa. Finally, we can execute the nodes as follows:

```
NodeConfiguration nodeConfiguration = null;
try {
    nodeConfiguration = NodeConfiguration.
                        newPublic("192.168.0.101",
                        new URI("http://192.168.0.100:11311/"));
                        //Set local and remote IPs
} catch (URISyntaxException e) {
    e.printStackTrace();
}

nodeMainExecutor.execute(talker, nodeConfiguration);

nodeMainExecutor.execute(visualizationView,
    nodeConfiguration.setNodeName("android/map_view"));

nodeMainExecutor.execute(image,
    nodeConfiguration.setNodeName("android/camera_view"));
```

Launch file for robot:

Now that we have the complete Android application, we know which ROS services to start in the robot and provide to the mobile device all the requested information. For this, we can use and join three of the launch files used during the ROS course:

- Turtlebot_lidar: This launch file provides all the basic services and nodes for performing teleoperation tasks, and also publishes the data read by the built-in Lidar.
- Map_launch: Brings up the necessary services to load and publish an environment map, and for accepting initial and goal poses to perform automatic path planning and following.
- Openni: It activates and streams the images captured by the Kinect's RGB camera.

All the commands from these three launch files are put in a new one under the name "android_turtlebot.launch", which is located in the folder /turtlebot_le2i/. This will let the user to run all the services in one single command, instead of doing it separately.

RESULTS WITH THE ANDROID APPLICATION

Driving of the robot by means of the Android application is shown in https://www.youtube.com/watch?v=S-ED2SAQYCo. Here, all the operations described above are performed with a parallel view between the running application and the moving robot.

In general, the robot's response to the application's orders are pretty accurate, showing that a good usage of the Rosjava libraries can lead to replace many of the functions performed by a computer-based application like Rviz, adding portability to the system and thus saving time and money.

The full source code of the project is available through Github, at https://github.com/espinely/android_turtlebot.

BENEFITS OF USING ROS WITH MATLAB

Robotics System Toolbox™ provides algorithms and hardware connectivity for developing autonomous mobile robotics applications. Toolbox algorithms include map representation, path planning, and path following for differential drive robots. You can design and prototype motor control, computer vision, and state machine applications in MATLAB® or Simulink® and integrate them with core algorithms in Robotics System Toolbox.

The system toolbox provides an interface between MATLAB and Simulink and the Robot Operating System (ROS) that enables you to test and verify applications on ROS-enabled robots and robot simulators such as Gazebo. It supports C++ code generation, enabling you to generate a ROS node from a Simulink model and deploy it to a ROS network.

Robotics System Toolbox includes examples showing how to work with virtual robots in Gazebo and actual ROS-enabled robots.

Key Features

- Path planning, path following, and map representation algorithms
- Functions for converting between different rotation and translation representations.
- Bidirectional communication with live ROS-enabled robots
- Interface to Gazebo and other ROS-enabled simulators
- Data import from rosbag log files
- ROS node generation from Simulink models (with Embedded Coder)

GET STARTED WITH ROS

INTRODUCTION

Robot Operating System (ROS) is a communication interface that enables different parts of a robot system to discover, send, and receive data. MATLAB support for ROS is a library of functions that allows you to exchange data with ROS-enabled physical robots, or robot simulators such as Gazebo.

This example introduces how to:

* Set up ROS within MATLAB
* Get information about capabilities in a ROS network
* Get information about ROS messages

INITIALIZE ROS NETWORK

* Use rosinit to initialize ROS. By default, rosinit creates a ROS master in MATLAB and starts a "global node" that is connected to the master. The "global node" is automatically used by other ROS functions.

rosinit

* Use rosnode list to see all nodes in the ROS network. Note that the only available node is the global node created by rosinit.

rosnode list

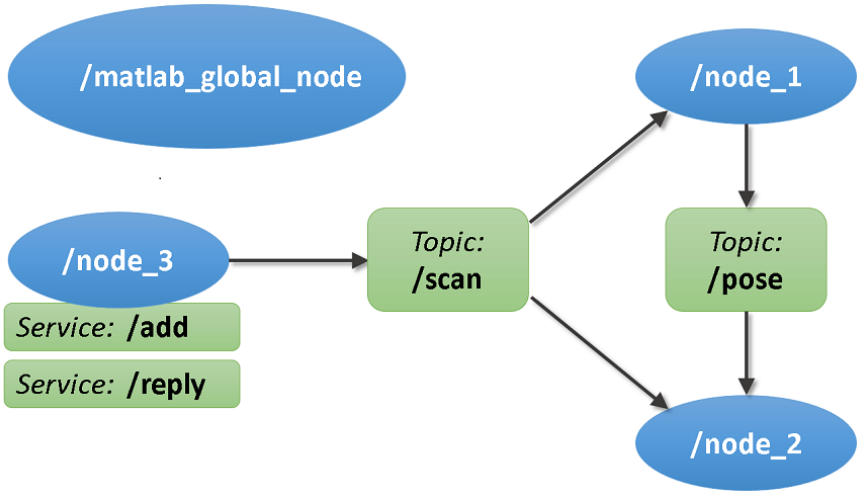
* Use exampleHelperROSCreateSampleNetwork to populate the ROS Network with three additional nodes and sample publishers and subscribers.

exampleHelperROSCreateSampleNetwork

* Use rosnode list again, and observe that there are three new nodes (node_1, node_2 and node_3).

rosnode list

A visual representation of the current state of the ROS network is shown below. Use it as a reference when you explore this sample network in the remainder of the example.
The MATLAB global node is disconnected since it currently does not have any publishers, subscribers or services.



a. Topics

- Use rostopic list to see available topics in the ROS network. Observe that there are three active topics: /pose, /rosout, and /scan. rosout is a default logging topic that is always present in the ROS network. The other two topics were created as part of the sample network.

- Use rostopic info to get specific information about a specific topic. The command below shows that /node_1 publishes (sends messages to) the /pose topic, and /node_2 subscribes (receives messages from) that topic (see Exchange Data with ROS Publishers and Subscribers for more information).

- Use rosnode info to get information about a specific node. The command below show that node_1 publishes to /pose and /rosout topics, and subscribes to the /scan topic.

b. Services

ROS Services provide a mechanism for "procedure calls" across the ROS network. A *service client* sends a request message to a *service server*, which processes the information in the request and returns with a response message (see Call and Provide ROS Services).

- Use rosservice list to see all available service servers in the ROS network. The command below shows that two services (/add and /reply) are available.

- Use rosservice info to get information about a specific service.

c. Messages

Publishers, subscribers, and services use ROS messages to exchange information. Each ROS message has an associated *message type* that defines the datatypes and layout of information in that message (See Work with Basic ROS Messages).

- Use rostopic type to see the message type used by a topic. The command below shows that the /pose topic uses messages of type geometry_msgs/Twist.

- Use rosmsg list to see the full list of message types available in MATLAB.

d. Shut Down ROS Network

- Use exampleHelperROSShutDownSampleNetwork to remove the sample nodes, publishers, and subscribers from the ROS network. This command is only needed if the sample network was created earlier using exampleHelperROSStartSampleNetwork.

exampleHelperROSShutDownSampleNetwork

- Use rosshutdown to shut down the ROS network in MATLAB. This shuts down the ROS master that was started by rosinit and deletes the global node. It is recommended to use rosshutdown once you are done working with the ROS network.

Rosshutdown

Shutting down global node /matlab_global_node_37458 with Node URI http : //bat6348glnxa64:46181/
Shutting down ROS master on http : //bat6348glnxa64:11311/

CONNECT TO ROS NETWORK

Introduction

A ROS network consists of a single ROS master and multiple ROS nodes. The ROS master facilitates the communication in the ROS network by keeping track of all active ROS entities. Every node needs to register with the ROS master to be able to communicate with the rest of the network. MATLAB® can start the ROS master or the master can be launched outside of MATLAB (for example, on a different computer). All ROS nodes register with the master and declare the network address where they can be reached.

When you work with ROS, you will typically follow these steps:

- Connect to a ROS network. To connect to a ROS network, you can create the ROS master in MATLAB or connect to an existing ROS master. In both cases, MATLAB will also create and register its own ROS node (called the MATLAB "global node") with the master. The rosinit function manages this process.

- Exchange Data. Once connected, MATLAB exchanges data with other ROS nodes through publishers, subscribers, and services.
- Disconnect from the ROS network. Calling the rosshutdown function disconnects MATLAB from the ROS network.

This section shows you how to:

- Create a ROS master in MATLAB
- Connect to an external ROS master

Create a ROS Master in MATLAB

- To create the ROS master in MATLAB, call rosinit without any arguments. This will also create the "global node", which MATLAB will use to communicate with other nodes in the ROS network.

- ROS nodes that are external to MATLAB can now join the ROS network. They can connect to the ROS master in MATLAB by using the hostname or IP address of the MATLAB host computer.

- You can shut down the ROS master and the global node by calling rosshutdown.

Connect to an External ROS Master

You can also use the rosinit command to connect to an external ROS master (for example running on a robot or a virtual machine). You can specify the address of the master in two ways: by an IP address or by a host name of the computer that runs the master.

After each call to rosinit, you have to call rosshutdown before calling rosinit with a different syntax. For brevity, these calls to rosshutdown are omitted in the following sections.

- In this example, use master_host as an example host name and 192.168.1.1 as an example IP address of the external ROS master. Adjust these addresses depending on where the external master resides in your network. Note that the following commands will fail if no master is found at the specified addresses.

rosinit('192.168.1.1')
rosinit('master_host')

- Both calls to rosinit assume that the master will accept network connections on port 11311, with is the standard ROS master port.
- If the master is running on a different port, you can specify it as a second argument. To connect to a ROS master running on host name master_host and port 12000, use the following command:

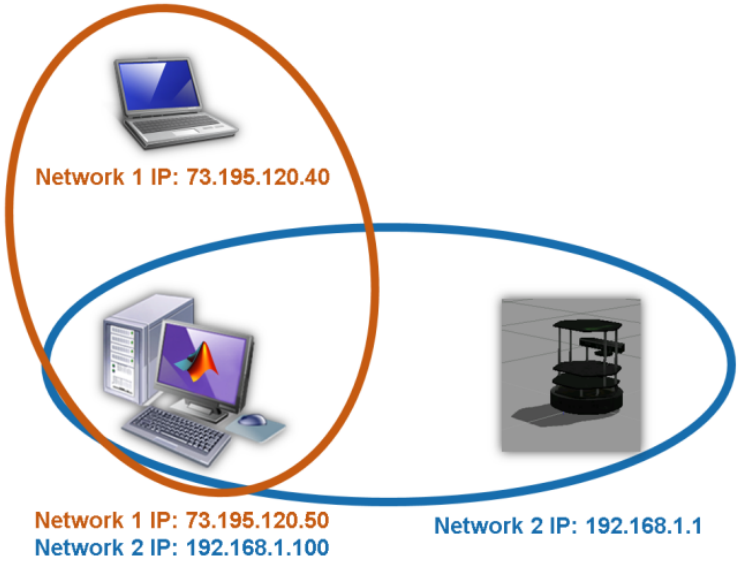
rosinit('master_host', 12000)

- If you know the entire URI (Uniform Resource Identifier) of the master, you can create the global node and connect to this master using the following syntax:

rosinit('http://192.168.1.1:12000')

Node Host Specification

In some cases, your computer may be connected to multiple networks and have multiple IP addresses. See the following illustration as an example.



The computer on the bottom left runs MATLAB and is connected to two different networks. In one subnet, its IP address is 73.195.120.50 and in the other, its IP is 192.168.1.100. This computer wants to connect to the ROS master on the TurtleBot® computer at IP address 192.168.1.1. As part of the registration with the master, the MATLAB global node has to specify the IP address or host name where other ROS nodes can reach it. All the nodes on the TurtleBot will use this address to send data to the global node in MATLAB.

When rosinit is invoked with the master's IP address, it will try to detect the network interface used to contact the master and use that as the IP address for the global node. If this automatic detection fails, you can explicitly specify the IP address or host name by using the NodeHost name-value pair in the rosinit call. All prior methods for calling rosinit are still permissible with the addition of the NodeHost name-value pair.

- For the following commands, assume that you want to advertise your computer's IP address to the ROS network as 192.168.1.100.

rosinit ('192.168.1.1', 'NodeHost', '192.168.1.100')
rosinit('http://192.168.1.1:11311', 'NodeHost', '192.168.1.100')
rosinit('master_host', 'NodeHost', '192.168.1.100')

- Once a node is registered in the ROS network, you can see the address that it advertises by using the command rosnode info NODE. NODE is the name of a node in the ROS network. You can see the names of all registered nodes by calling rosnode list.

ROS Environment Variables

In advanced use cases, you might want to specify the address of a ROS master and your advertised node address through standard ROS environment variables. The calling syntaxes that were explained in the previous sections should be sufficient for the majority of your use cases.

- If no arguments are provided to rosinit, the function will also check the values of standard ROS environment variables. These variables are ROS_MASTER_URI, ROS_HOSTNAME, and ROS_IP. You can see their current values using the getenv command:

getenv('ROS_MASTER_URI')
getenv('ROS_HOSTNAME')
getenv('ROS_IP')

- You can set these variables using the setenv command. After setting the environment variables, call rosinit with no arguments. The address of the ROS master is specified byROS_MASTER_URI and the global node's advertised address is given by ROS_IP or ROS_HOSTNAME. *If you specify additional arguments to* rosinit*, they will override the values in the environment variables.*

setenv('ROS_MASTER_URI','http://192.168.1.1:11311')
setenv('ROS_IP','192.168.1.100')
rosinit

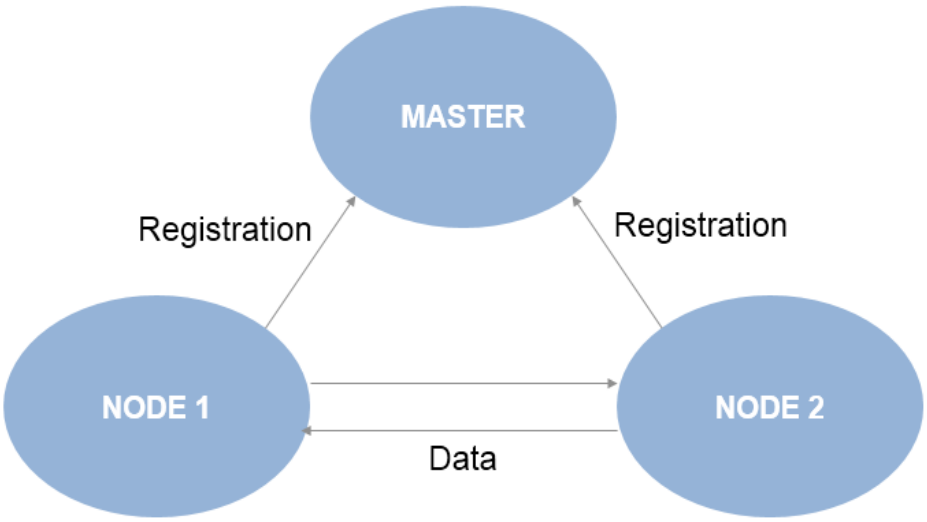
- You do not have to set both ROS_HOSTNAME and ROS_IP. If both are set, ROS_HOSTNAME takes precedence.

Verify Connection

For your ROS connection to work correctly, you must ensure that all nodes can communicate with the master and with each other. The individual nodes must communicate with the master to register subscribers, publishers, and services. They must also be able to communicate with one another to send and receive data.

Because the communication works in this way, it is possible to be able to send data and unable to receive it (or vice versa) if your ROS network is not set up correctly.

Here is a diagram of the communication structure in a ROS network. There is a single ROS master and two different nodes that register themselves with the master. Each node will contact the master to find the advertised address of the other node in the ROS network. Once each node knows the other node's address, a data exchange can be established without involvement of the master.



TEST BASIC FUNCTIONALITY OF TURTLEBOT WITH MATLAB

INTRODUCTION

This example helps you to explore basic autonomy with the TurtleBot. The described behaviour drives the robot forward and changes its direction when there is an obstacle. You will subscribe to the laser scan topic and publish the velocity topic to control the TurtleBot.

HARDWARE SUPPORT PACKAGE FOR TURTLEBOT

This example gives an overview of working with a TurtleBot using its native ROS interface. The Robotics System Toolbox™ Support Package for TurtleBot® based Robots provides a more streamlined interface to TurtleBot. It allows you to:

- Acquire sensor data and send control commands without explicitly calling ROS commands
- Communicate transparently with a simulated robot in Gazebo or with a physical TurtleBot

To install the support package, open Add-Ons > Get Hardware Support Packages on the MATLAB Home tab and select "TurtleBot based Robots". Alternatively, use the roboticsAddons command.

CONNECT TO THE TURTLEBOT

Make sure you have a TurtleBot running either in simulation through Gazebo® or on real hardware. Refer to Get Started with Gazebo and a Simulated TurtleBot or Get Started with a Real TurtleBot for the startup procedure. Any Gazebo world works while running in simulation, however, Gazebo TurtleBot World is the most interesting for the purposes of this example.

- Initialize ROS. Connect to the TurtleBot by replacing the sample IP address (192.168.1.1) with the IP address of the TurtleBot.

```
ipaddress = '192.168.1.1'

    rosinit(ipaddress)
```

- Create a publisher for the robot's velocity and create a message for that topic.

```
    robot = rospublisher('/mobile_base/commands/velocity');
    velmsg = rosmessage(robot);
```

RECEIVE SCAN DATA

Make sure that you start the Kinect camera if you are working with real TurtleBot hardware. That command is: roslaunch turtlebot_bringup 3dsensor.launch. You must execute the command in a terminal on the TurtleBot. The TurtleBot uses the Kinect data to simulate a laser scan that is published on the /scan topic. For the remainder of this example, the term "laser scan" refers to data published on this topic.

- Subscribe to the topic /scan

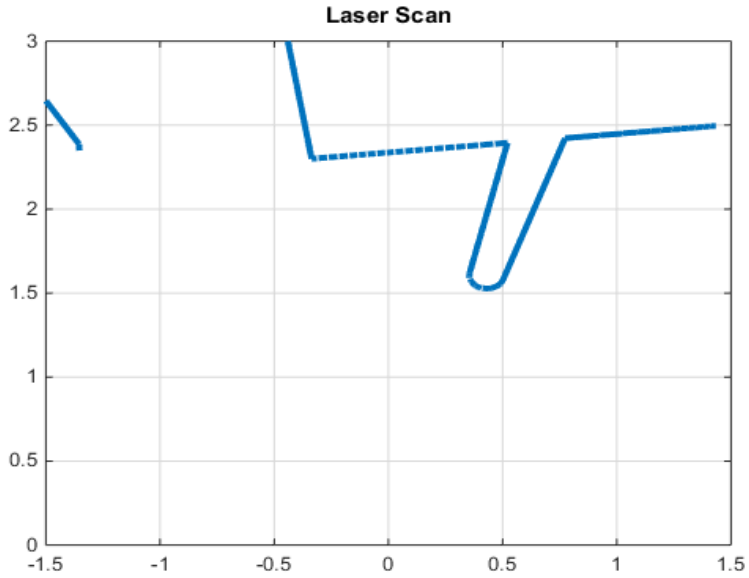
```
laser = rossubscriber('/scan');
```

- Wait for one laser scan message to arrive and then display it.

```
scan = receive(laser,3)
figure
plot(scan);
```

If you see an error, it is possible that the laser scan topic is not receiving any data. If you are running in simulation, try restarting Gazebo. If you are using hardware, make sure the that you started the Kinect camera properly.

- In Gazebo, the scan is similar to this scan:



- Run the following lines of code, which plot a live laser scan feed for twenty seconds. Move an object in front of the TurtleBot and bring it close enough until it no longer shows up in the plot window. The laser scan has a limited range because of hardware limitations of the Kinect camera. The Kinect has a minimum sensing range of 0.8 meters and a maximum range of 4 meters. Any objects outside these limits will not be detected by the sensor.

```
tic;
while toc < 10
   scan = receive(laser,3);
   plot(scan);
end
```

SIMPLE OBSTACLE AVOIDANCE

Based on the distance readings from the laser scan, you can implement a simple obstacle avoidance algorithm. You can use a simple while loop to implement this behavior.

- Set some parameters that will be used in the processing loop. You can modify these values for different behavior.

```
spinVelocity = 0.6;      % Angular velocity (rad/s)
forwardVelocity = 0.1;   % Linear velocity (m/s)
backwardVelocity = -0.02; % Linear velocity (reverse) (m/s)
distanceThreshold = 0.6;  % Distance threshold (m) for turning
```

- Run a loop to move the robot forward and compute the closest obstacles to the robot. When an obstacle is within the limits of the distanceThreshold, the robot turns. This loop stops after 20 seconds of run time. CTRL+C (or Control+C on the Mac) also stops this loop.

```
tic;
while toc < 20
    % Collect information from laser scan
    scan = receive(laser);
    plot(scan);
    data = readCartesian(scan);
    x = data(:,1);
    y = data(:,2);
    % Compute distance of the closest obstacle
    dist = sqrt(x.^2 + y.^2);
    minDist = min(dist);
    % Command robot action
    if minDist < distanceThreshold
        % If close to obstacle, back up slightly and spin
        velmsg.Angular.Z = spinVelocity;
        velmsg.Linear.X = backwardVelocity;
    else
        % Continue on forward path
        velmsg.Linear.X = forwardVelocity;
        velmsg.Angular.Z = 0;
    end
    send(robot,velmsg);
end
```

DISCONNECT FROM THE ROBOT

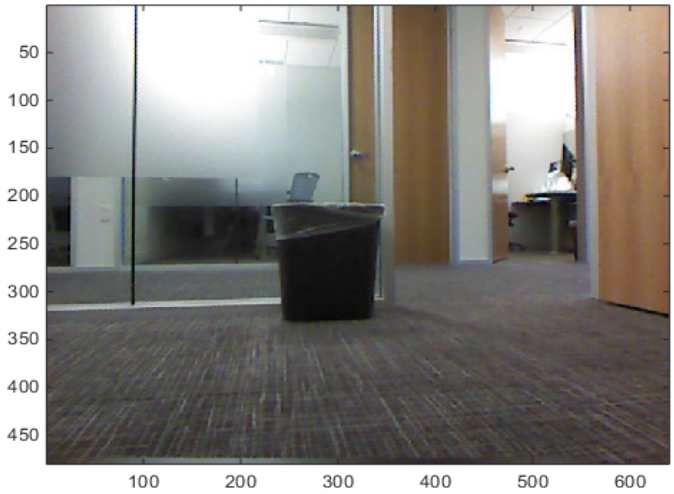
- It is good practice to clear the workspace of publishers, subscribers, and other ROS related objects when you are finished with them.

- It is recommended to use rosshutdown once you are done working with the ROS network. Shut down the global node and disconnect from the TurtleBot.

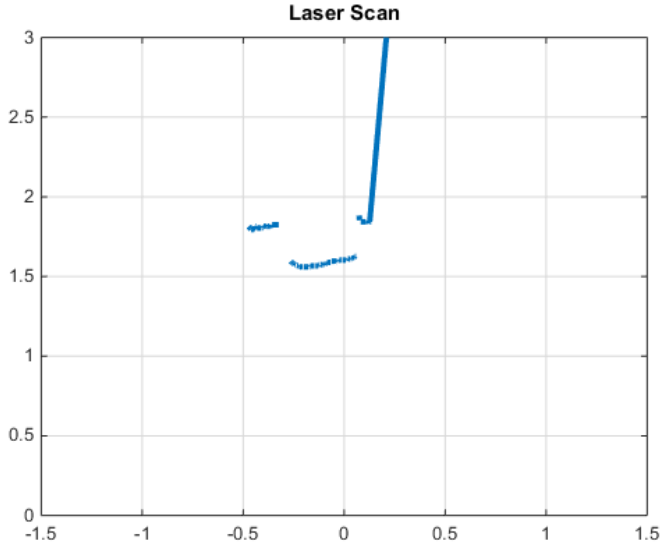
Rosshutdown

MORE INFORMATION

The laser scan has a minimum range at which it no longer sees objects in its way. That minimum is somewhere around 0.5 meters from the Kinect camera.
The laser scan cannot detect glass walls. Following is an image from the Kinect camera:

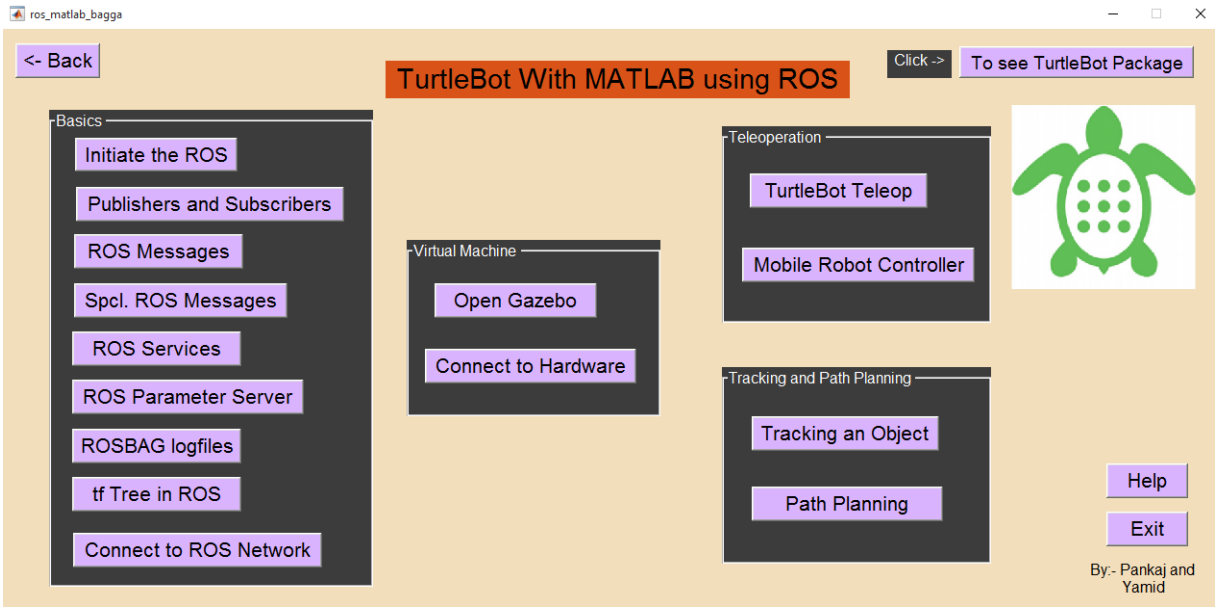


Here is the corresponding laser scan:

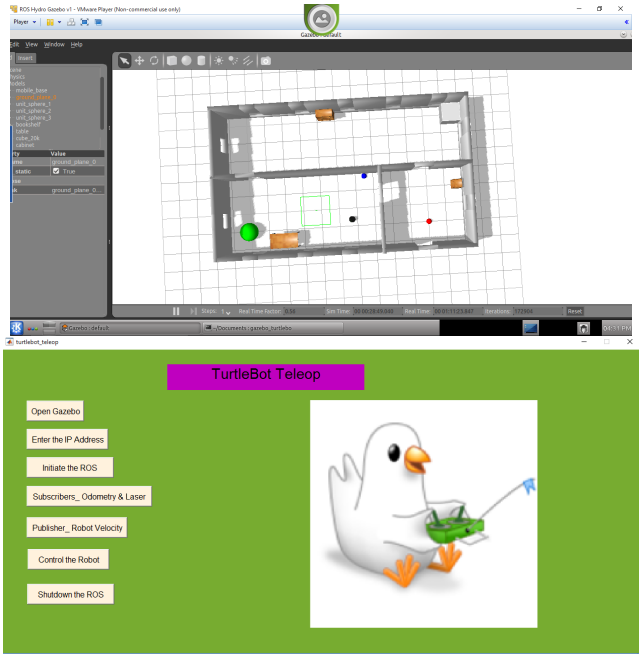


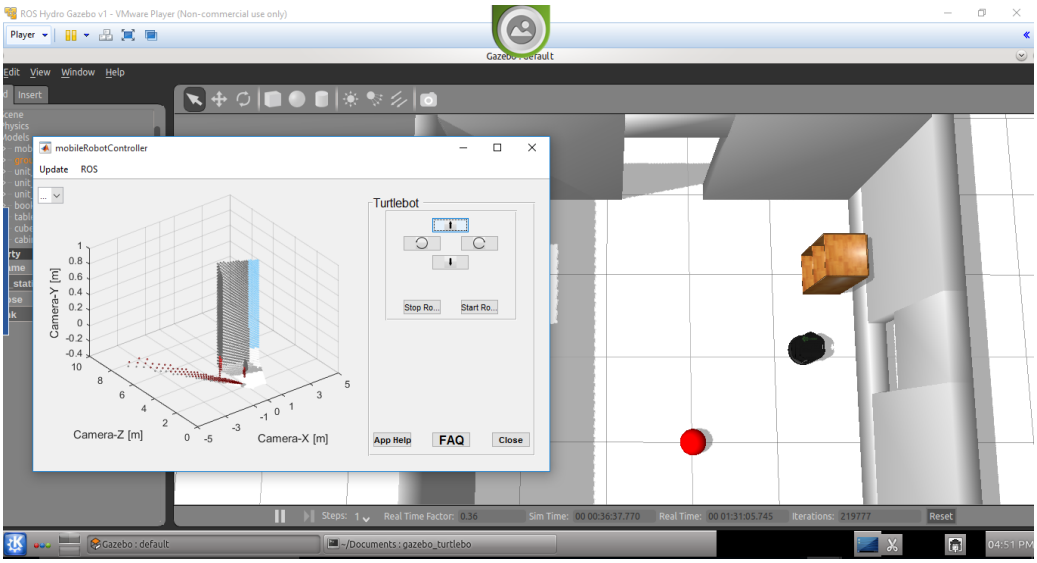
The trash can is visible, but you cannot see the glass wall. When you use the TurtleBot in areas with windows or walls that the TurtleBot might not be able to detect, be aware of the limitations of the laser scan.

RESULTS WITH THE MATLAB APPLICATION

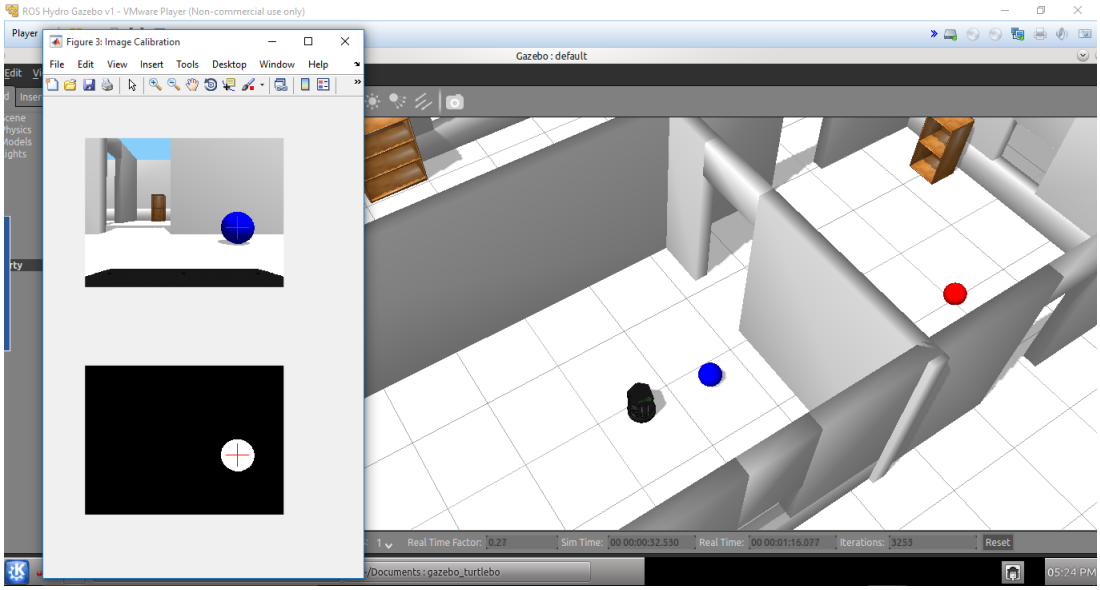


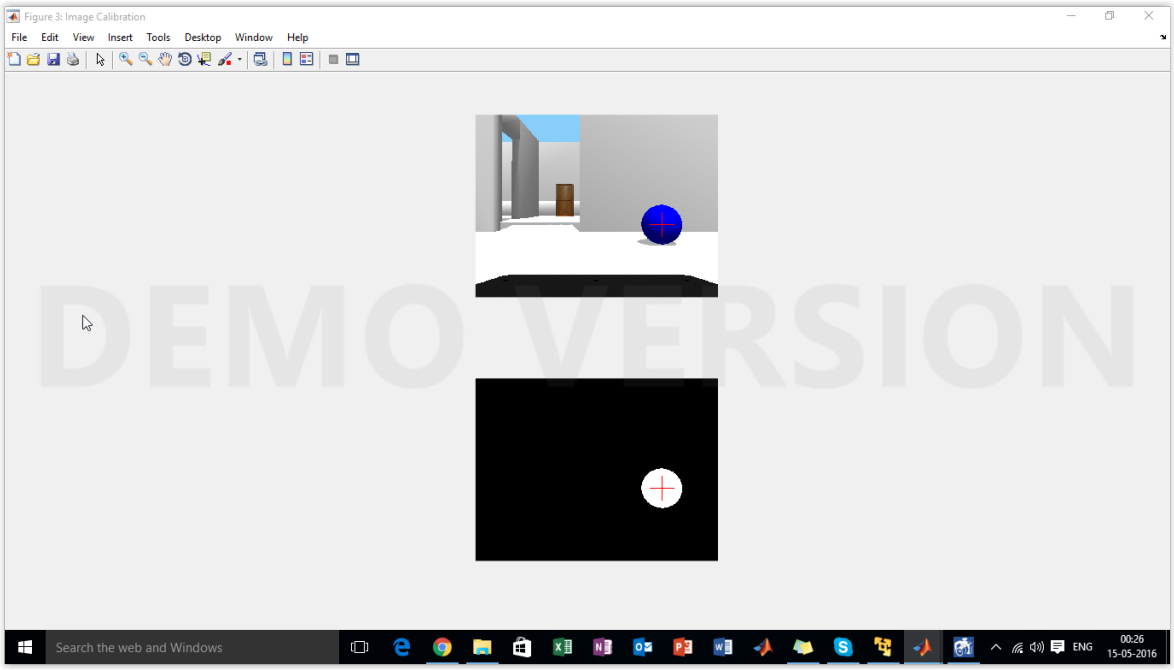
(i) Teleop operation



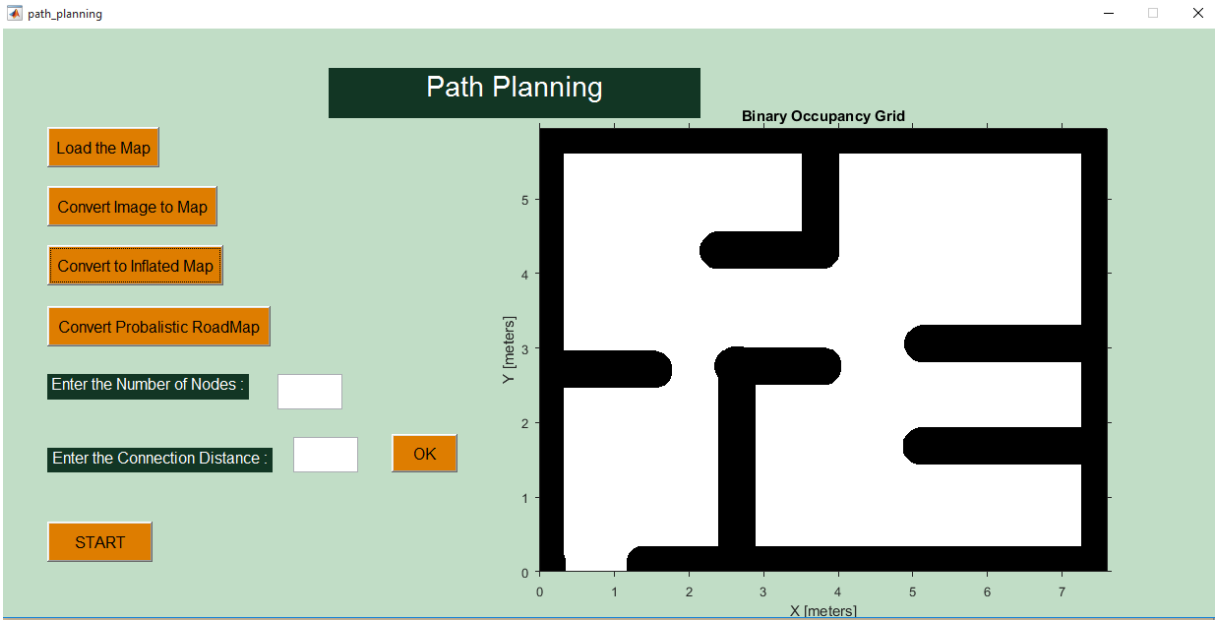


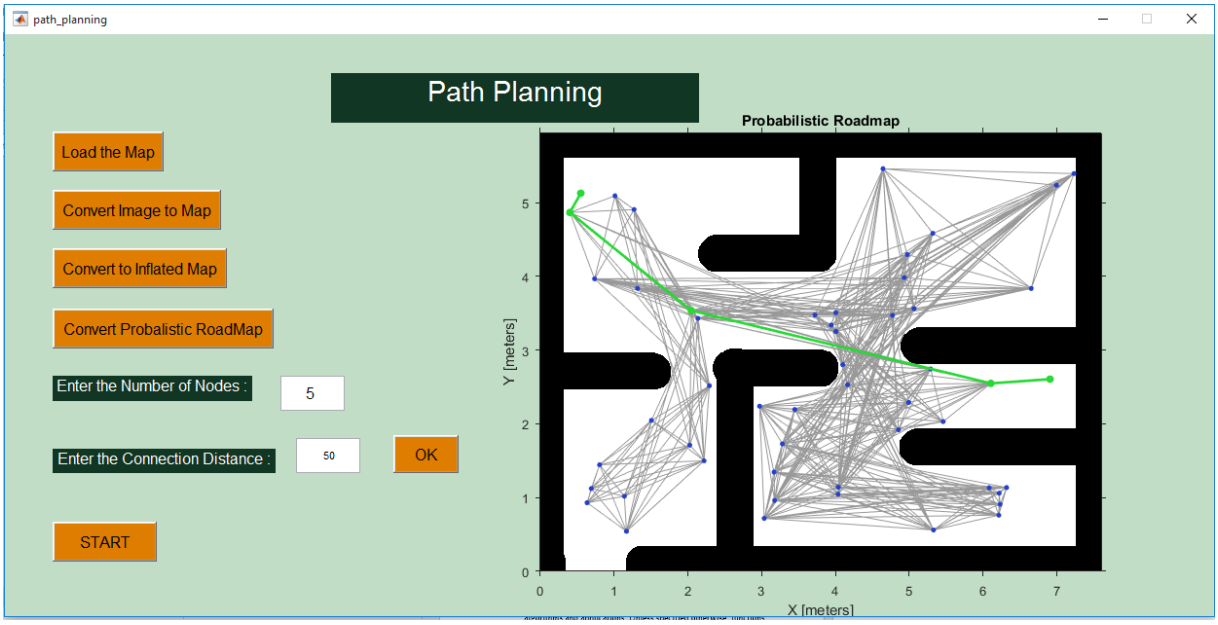
(ii) Track an Object





(iii) Path Planning





The complete source code can be found at https://github.com/pnkjbagga/ROS-PROJECT.

COMPILATION OF SOURCE CODE AND VIDEO LINKS

- Android app source code: https://github.com/espinely/android_turtlebot
- Android app demonstration video: https://www.youtube.com/watch?v=S-ED2SAQYCo
- ROS with Matlab source code: https://github.com/pnkjbagga/ROS-PROJECT