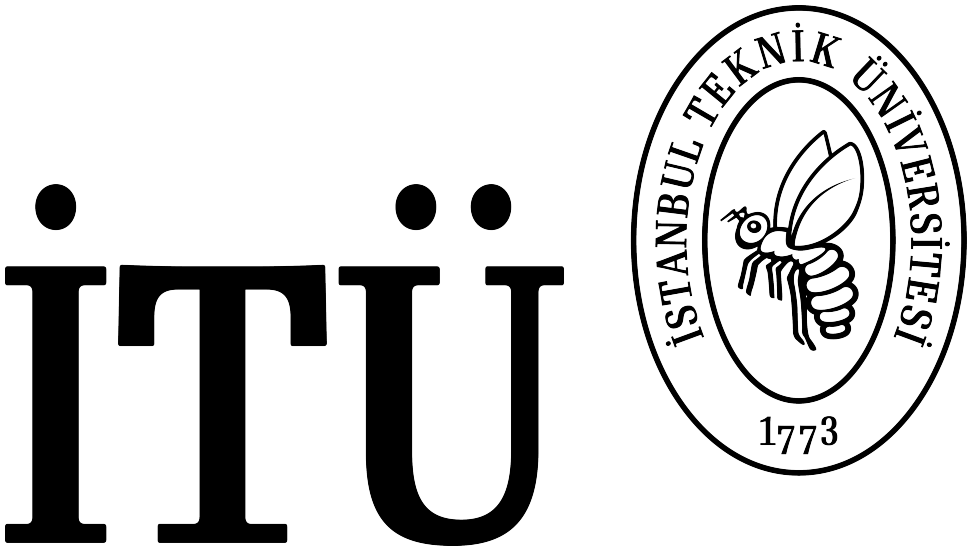

Mobile Robot Applications with Robot Operating System (ROS)

Bachelor's Thesis

Emre AY - 040100675

June 1, 2015

Advisor: Prof. Dr. Hakan TEMELTAŞ

Department of Control and Automation Engineering

Faculty of Electrical and Electronics Engineering,

Istanbul Technical University

Preface

I would like to thank my sister and parents for their motivation, patience and endless supports of all kinds. For his inspiration, guidance and helps together with giving me these opportunities I would like to thank my advisor Prof. Dr. Hakan TEMELTAŞ. For sharing all that time with me and teaching me countless things, I want to indicate my appreciation to Onur ŞENCAN and Osman ERVAN at ITU Robotics Laboratory. I want to thank Serkan TÜRKELİ for his guidance, motivation and most importantly his friendship. Lastly, I would like to express my gratitude for all my professors, lecturers, teachers and assistants at Istanbul Technical University.

Emre AY, May 2015

Contents

Preface i

Contents iii

List of Figures v

Abstract vii

Özet ix

1 Introduction 1
- 1.1 Mobile Robotics . . . 1
- 1.2 Goal . . . 2
- 1.3 Organization . . . 3

2 Autonomous Guided Vehicles & ITU-AGVs 5
- 2.1 Spesifications of ITU-AGVs . . . 6
- 2.2 Sensors and Low Level Processing Layer (LLPL) . . . 6
  - 2.2.1 Light Detection and Ranging (LIDAR) Sensors . . . 6
  - 2.2.2 Inertial Measurement Unit (IMU) . . . 7
  - 2.2.3 Infrared Distance Sensors . . . 7
  - 2.2.4 Microsoft Kinect Sensor . . . 7
  - 2.2.5 Motor Encoders . . . 8
  - 2.2.6 Low Level Processing Layer (LLPL) . . . 8
- 2.3 Kinematics . . . 9

3 Robot Operating System (ROS) 13
- 3.1 Fundamentals of ROS . . . 13
- 3.2 A Brief Review of ROS . . . 13

4 Mobile Robot Application Development 15

4.1 Embedded Program Development for LLPL 16
4.1.1 Communication with EPOS Drivers 16
4.1.2 Communication with HLPL 18
4.1.3 Testing the LLPL 19
4.2 ROS Programming for HLPL 20
4.2.1 ROS Installation & System Initialize 20
4.2.2 Teleoperation Application 20
4.2.2.1 Simulation 20
4.2.2.2 Teleoperation of ITU-AGVs 24
4.2.3 Sensor Integration 25
4.2.3.1 Encoder Reading 25
4.2.3.2 IMU Reading 26
4.2.3.3 LIDAR Reading 26
4.2.3.4 Kinect Reading 26
4.2.4 Odometry Estimation 27
4.2.5 Offline Map Building 28

5 Conclusion & Future Work **31**

A Codes & Scripts **33**

Bibliography **47**

List of Figures

1.1 Front view of Mars Curiosity rover, Courtesy NASA/JPL-Caltech [5] 2

2.1 One of the ITU-AGV robots 5
2.2 Power chart of ITU-AGVs 7
2.3 Microsoft Kinect and Sick LMS200 sensors on ITU-AGVs 8
2.4 Xsens MTi sensor on ITU-AGVs 9
2.5 Frames of a mobile robot in 2D 10

4.1 Signal chart of ITU-AGVs 16
4.2 Configuration flow chart for profile velocity mode [3] 17
4.3 State chart flow diagram of LLPL in Simulink 18
4.4 Testing LLPL with a written Python script 19
4.5 Urdf tree 22
4.6 Urdf model of ITU-AGVs 23
4.7 3D model ITU-AGVs opened in Rviz 24
4.8 Teleoperation of ITU-AGV model in Rviz 24
4.9 Node graph (rqt-graph) of teleoperation 25
4.10 Simultanoues stream of LIDAR data on Rviz and the image of real environment 26
4.11 Yaw angle values is retrieved from IMU while the robot is rotating 27
4.12 RGB and point cloud data stream from Kinect on Rviz 27
4.13 Node graph (rqt graph) while the robot pose ekf is publishing fused odometry data 28
4.14 An offline map of ITU Control and Automation Eng. Department corridor is built with the collected data 29

5.1 Husky robot, ITU-AGVs and AR Drone quadcopter at ITU Robotics Laboratory 32

Abstract

Robot Operating System (ROS) has gained a rapid spread that cannot be overlooked in the robotics programming in less than a decade. This success of ROS is of course based on its functionalities and features. In ITU Robotics Laboratory, the advantages of ROS are noticed and many systems are migrated to ROS. There are two mobile robots named ITU-AGVs in the laboratory that were built before ROS and it is needed to maintain these robots and develop base applications on the ROS framework that would be used for complex applications in the future projects. This project contains the development of embedded software that will be communicable for ROS and creating ROS applications.

Özet

Robot İşletim Sistemi (Robot Operating System, ROS), on yıldan daha kısa bir süre içerisinde robotik programlamada göz ardı edilemeyecek bir gelişim göstermiştir. ROS'un bu başarısının altında sunduğu çeşitli özellikler ve işlevselliği bulunmaktadır. İTÜ Robotik Laboratuvarı'nda da ROS'un bu avantajları fark edilerek, birçok sistemde ROS'a geçiş yapılmıştır. Laboratuvarda İTÜ-AGV ismi verilen iki adet mobil robot önceki yıllarda yapılmıştır. Bu robotlara ileride karmaşık projelere temel oluşturacak önemli uygulamaların ROS ortamında yazılması ihtiyacı bulunmaktadır. Bu proje kapsamında, ROS ile haberleşmeye uygun gömülü yazılımın ve ROS üzerinde çeşitli uygulamaların geliştirilmesi yer almaktadır.

Chapter 1

Introduction

1.1 Mobile Robotics

Mobile robotics has shown significant developments in the last years resulting in growing areas of applications from Mars exploration (Figure 1.1) to house cleaning. Within this growth, mobile robots varied in the ways of their movement, in their target environments and applications. Hence, they have spread into more branches which include quite new research areas. A rough classification of mobile robots can be made as follows;

Type of structure

- Legged robots
- Tracks
- Wheeled robots

Operating environment

- Indoor robots
- Space robots
- Land robots (Unmanned Ground Vehicles - UGVs)
- Underwater robots (Autonomous Underwater Vehicles - AUVs)
- Aerial robots (Unmanned Aerial Vehicles - UAVs)
- Polar robots

Application area

- Service robots
- Cleaning robots

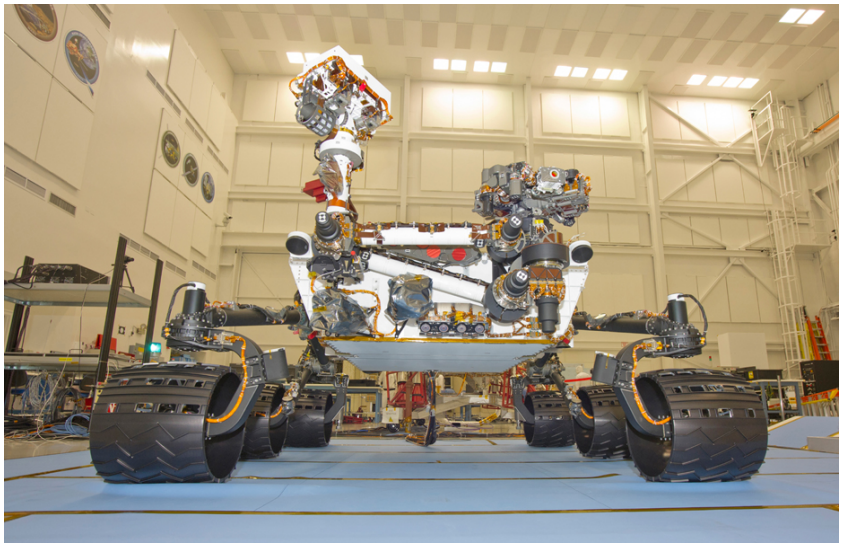

Figure 1.1: Front view of Mars Curiosity rover, Courtesy NASA/JPL-Caltech [5]

- Field robots
- Social robots
- Material handling robots

The classification can be varied with new application areas and technologies. Although the specific applications are developed depending on the type and target task of the mobile robots, they share several cases to be handled such as sensor reading, tele-operation, to estimate the position and orientation, navigation and so on. These cases form the basics of a mobile robot application development.

1.2 Goal

In this project, it is aimed to develop base applications using Robot Operating System (ROS) framework for the mobile robots of the type Autonomous Guided Vehicle (AGV) at Istanbul Technical University Robotics Laboratory named ITU-AGVs. The goal covers developing embedded software to be able to communicate with the microcontroller and developing on ROS framework for base tasks including simulation, sensor integration and reading, tele-operation, estimation of position and orientation, data collection and offline map building. It is aimed to provide these operations so that, they can be used as a basis –which ITU-AGVs lack– for developing specific applications.

1.3 Organization

It is necessary to provide a background information in order to emphasize the project work. Hence in Chapter 2, information regarding Autonomous Guided Vehicles together with the details of ITU-AGVs are given and the kinematic model is derived. In Chapter 3, basics of Robot Operating System are provided with a brief review. Afterwards, all the work regarding the application development process is told in detailed in Chapter 4 and its subsections. Then, the overall outcomes, conclusions and possible future work are given in Chapter 5.

Chapter 2

Autonomous Guided Vehicles & ITU-AGVs

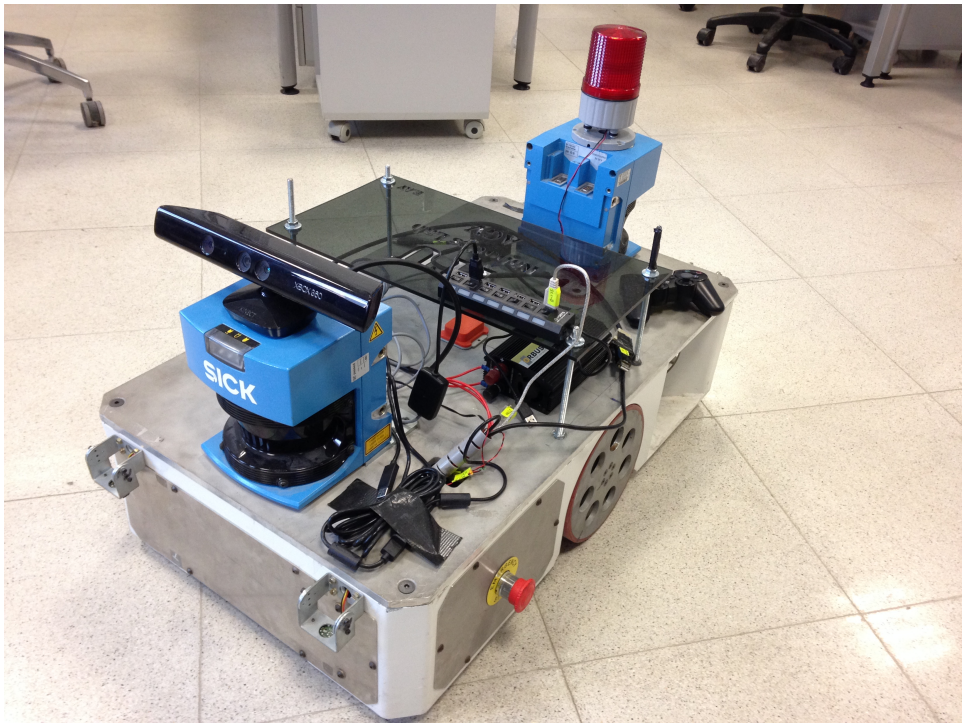

Figure 2.1: One of the ITU-AGV robots

Autonomous Guided Vehicles (AGVs) or Automated Guided Vehicles are mobile robots that use lines or wires installed on the floor, cameras or laser sensors in order to navigate. They are industrial robots and their usual task is to carry objects or products in indoor or outdoor environments. Their market might be considered as the biggest one in mobile robotics [1]. The

present-day AGVs are mostly use laser sensors instead of floor wires or lines.

There are two identical AGVs that have built at Istanbul Technical University Robotics Laboratory named ITU-AGVs. Although they are designed as AGVs, they do not necessarily have to be used for warehouse automation or object carrying, but also they can be used as multi-purpose wheeled mobile robots.

2.1 Spesifications of ITU-AGVs

ITU-AGV robots are differential-driven, bidirectional mobile robots with two driving wheels and two caster wheels (Figure 2.1). They have two 250 Watt Maxon EC54 brushless DC motors with a ratio of 1:100 reduction gear-boxes. The motors are driven by using Maxon EPOS 70/10 drivers. The robots are powered with two serially connected batteries with 12 V output voltage and 26 Ah charge, each. To supply motors and all other hardware, there are 5 V, 12 V and 24 V voltage regulators in order to acquire necessary voltage levels.

ITU-AGVs have 49 cm width, 82 cm length and 22 cm height. The driving wheels have 10 cm radius. They weight approximately 70 kg without the additional sensors and their payload is approximately 100 kg for each.

It is possible to install various sensors on the robot. To make it a multi-purpose robot that can be used in different future projects, two laser range finder sensors, a Microsoft Kinect sensor, an inertial measurement unit (IMU) and four analog distance sensors are mounted on ITU-AGVs. The microcontrollers provided for ITU-AGVs were Texas Instruments TMS320F28335 with Spectrum Digital eZdsp F28335 board. The power chart of ITU-AGVs can be seen in Figure 2.2.

2.2 Sensors and Low Level Processing Layer (LLPL)

2.2.1 Light Detection and Ranging (LIDAR) Sensors

There are two light detection and ranging (LIDAR) sensors mounted on ITU-AGVs. These are SICK Laser Measurement System LMS200 (Figure 2.3) and they are indoor sensors with 180 degrees scanning field and 10 meters range. Their working principle is based on time of flight measurement of reflected infrared light beam emitted by the sensor. In order to have a radial range, a rotating mirror deflects the emitted infrared light to the environment. The

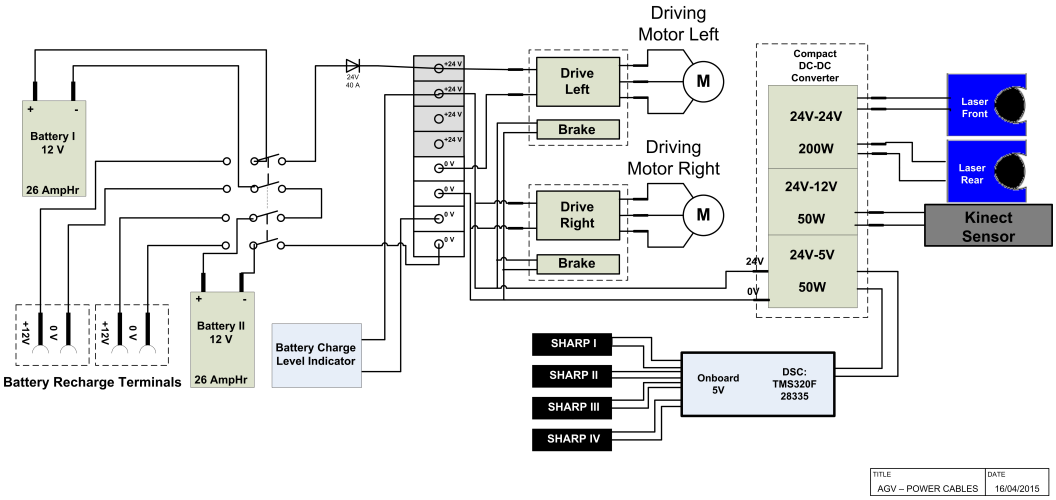

Figure 2.2: Power chart of ITU-AGVs

sensor outputs the distance values in the range for 180 degrees at 9600 baud rate [12].

2.2.2 Inertial Measurement Unit (IMU)

Using an internal measurement unit is useful for position and orientation estimation. Hence, a 3 DOF Xsens MTi Attitude and Heading Reference System (AHRS) is mounted on ITU-AGVs (Figure 2.4). Xsens MTi is an IMU that has magnetometers, accelerometers and gyroscopes and it outputs orientation, acceleration, rate of turn and earth magnetic data in three dimensions [14]. It has small dimensions ($58 \times 58 \times 22$ mm) and low weight (50 g). The sensor is mounted along the center of the robot.

2.2.3 Infrared Distance Sensors

Four infrared distance sensors are mounted on front and rear of ITU-AGVs in order to understand if there is a hole or a stair while the robots are moving in the environment. The infrared distance sensors are analog Sharp sensors with 30 cm ranges.

2.2.4 Microsoft Kinect Sensor

Microsoft Kinect sensor has an RGB camera with 1280×960 resolution, an infrared emitter and infrared depth sensor to get the depth information by measuring the distance of objects from the reflected infrared beams that have

emitted by the sensor, a microphone array, an accelerometer and a tilt motor [4]. It is mounted on the front of ITU-AGVs (Figure 2.3) and its RGBD output can be used for many applications.

Figure 2.3: Microsoft Kinect and Sick LMS200 sensors on ITU-AGVs

2.2.5 Motor Encoders

Motor encoders are also vital for estimation of position and orientation. The motors have three channels, 500 counts per turn HEDL 9140 encoders.

2.2.6 Low Level Processing Layer (LLPL)

As in many robotic systems, there are two processing layers in ITU-AGVs. Low Level Processing Layer (LLPL) is responsible of getting commands from High Level Processing Layer (HLPL), communicating with motor drivers and sending necessary signals to drive the motors, requesting the encoder values and send them to HLPL. In addition to these flow, the analog distance sensors are also connected to LLPL. The microcontrollers used at LLPL on ITU-AGVs are Texas Instruments TMS320F28335 with Spectrum Digital eZdsp F28335 board.

The eZdsp F28335 is a stand-alone board with TMS320F28335 Digital Signal Controller. It works at 150 Mhz operating speed and it has 32-bit floating

Figure 2.4: Xsens MTi sensor on ITU-AGVs

point unit, 68 KB RAM, 512 KB Flash memory, 256 KB off-chip SRAM memory, 12-bit Analog to Digital Converter (ADC), 30 MHz input clock, RS232 and CAN connectors, USB JTAG controller and multiple General Purpose Input Output (GPIO) pins [13].

2.3 Kinematics

It is necessary to construct the kinematic model of ITU-AGVs since it will be used in application development. Consider r as the radius of the wheels, R as the radius of rotation, L as the length between the wheels, $\omega(t)$ and $V(t)$ as the angular and linear velocities of the robot, $V_L(t)$ and $\omega_L(t)$ as the linear and angular velocities of the left wheel, $V_R(t)$ and $\omega_L(t)$ as the linear and angular velocities of the right wheel, θ is the angle between x axis of the frames, x_0 and y_0 as the coordinate axis of the world frame and x_1 and y_1 as the coordinate axis of the robot frame as in Figure 2.5.

At any time instant t, the linear velocities at left and right wheels can be calculated from the product of their angular velocities and radius;

$$V_L(t) = \omega_L(t) \cdot r \tag{2.1}$$

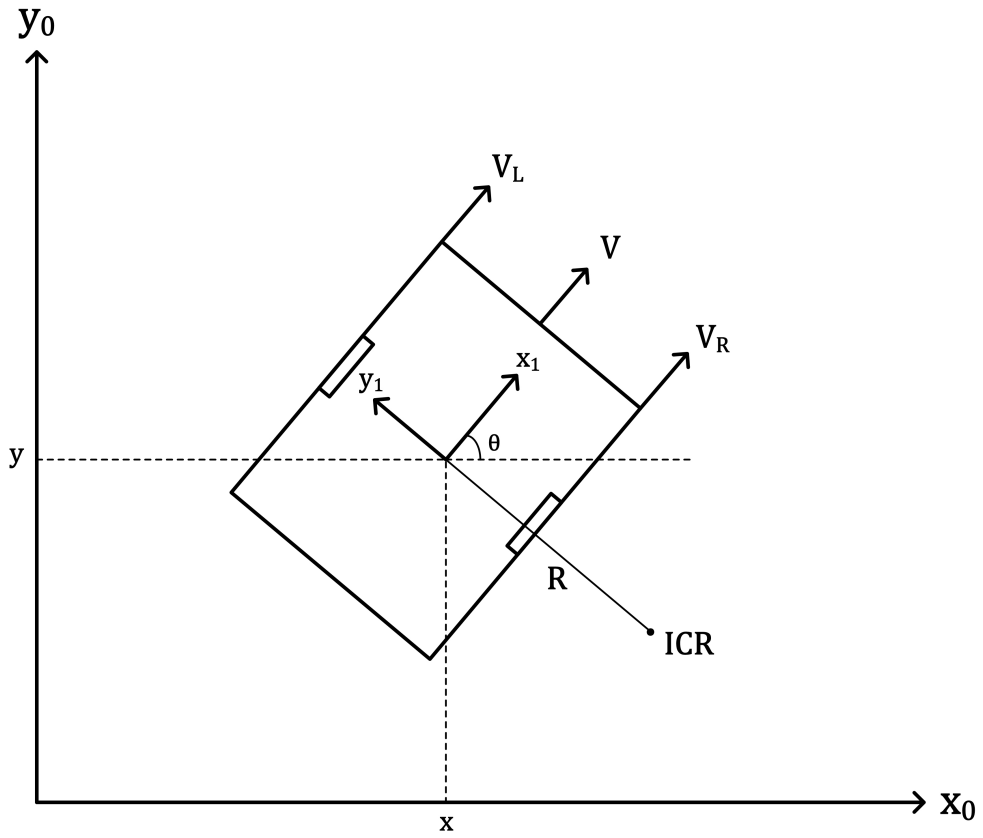

Figure 2.5: Frames of a mobile robot in 2D

$$V_R(t) = \omega_R(t) \cdot r \tag{2.2}$$

The linear velocities also can be written from the angular velocity of the robot;

$$V_L(t) = \omega(t) \cdot (R - \frac{L}{2}) \tag{2.3}$$

$$V_R(t) = \omega(t) \cdot (R + \frac{L}{2}) \tag{2.4}$$

So, combining and solving these equations yields the angular velocity of the robot;

$$\omega(t) = \frac{V_R(t) - V_L(t)}{L} \tag{2.5}$$

The linear velocity of the robot is simply;

$$V(t) = \frac{V_R(t) + V_L(t)}{2} \tag{2.6}$$

The kinematic model in the world frame can be constructed as;

$$\begin{bmatrix} v_{x_0}(t) \\ v_{y_0}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} cos\theta & 0 \\ sin\theta & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} V(t) \\ \omega(t) \end{bmatrix} \tag{2.7}$$

and the kinematic model in the robot frame can be constructed as;

$$\begin{bmatrix} v_{x_1}(t) \\ v_{y_1}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ 0 & 0 \\ -\frac{R}{L} & \frac{R}{L} \end{bmatrix} \cdot \begin{bmatrix} \omega_L(t) \\ \omega_R(t) \end{bmatrix} \tag{2.8}$$

where $v_{x_0}(t), v_{y_0}(t), v_{x_1}(t)$ and $v_{y_1}(t)$ are the velocities at frame axis x_0, y_0, x_1 and y_1.

Chapter 3

Robot Operating System (ROS)

3.1 Fundamentals of ROS

Robot Operating System (ROS) framework has shown an increasing popularity at robotics applications since it was first released at 2009 by Willow Garage. Even though it has "operating system" in its name, ROS is not an actual operating system. It might be classified as a *framework* or a *middleware* that serves various useful tools.

The hardware on the present-day robots differs broadly. To prevent writing codes again for same or similar tasks on robots with different hardware, or in others words to avoid reinventing the steel, by providing an environment is the basic logic behind the ROS.

ROS is designed to be peer-to-peer, tool based, multi-lingual, thin, free and open-source [6]. Every ROS application consists of computational units or programs named *nodes* and communications and relationships of them. The *nodes* can communicate over *topics* by passing certain data named *messages*.

ROS is based on four fundamentals; passing messages by publishing or subscribing to topics, passing messages using services, recording messages and playing-back when necessary and having a dynamically reconfigurable distributed parameter server [7]. Using these core features, it is possible to perform many simple and complex tasks.

3.2 A Brief Review of ROS

ROS project officially supports Ubuntu and pre-compiled ROS distributions are supplied officially. However, since it is an open-source project, its source files are available and it is possible to compile them on similar platforms.

Hence there are experimental pre-compiled repositories available such as the one for the Raspbian operating system of Raspberry Pi development boards [9].

ROS file system is based on unit software organizations called *packages* and organization of related packages called *metapackages* [10]. There used be *stack* organization but they have replaced with *metapackages* in the newer ROS distributions. Every package can contain libraries, executables, source codes, launch files, scripts and so on.

ROS uses a build system called *catkin* which combines CMake build system with Python codes. A build system is the system that constitutes target files which might be executables, libraries, header files from the source code [8]. ROS changed its build system to *catkin* on and after the Groovy distribution.

As mentioned in the previous section, one of the design goals of ROS was to be multi-lingual. ROS supports C++ and Python. So the nodes can be written in both C++ and Python and a node that has written in C++ can communicate with the one that has written in Python and vice versa.

ROS does not only provide features to make the nodes communicate in certain methods, but it also provides several tools to make diagnostics and debug. For example, there are tools to see the nodes and topics as a graph, to plot messages according to time or to find out the message passing frequency on a topic. Also there are simulation and visualization environments working with ROS such as Rviz and Gazebo.

ROS framework forms an environment for reuse the codes. There are many message type definitions for common messages and also it is easy to create custom messages. Many sensors and hardware have their ROS libraries and drivers which makes easier and faster to install various hardware. Finally, ROS has an involved and wide community. All these features make ROS preferable and usable.

Chapter 4

Mobile Robot Application Development

It is needed to have a software basis for ITU-AGVs that will be used to develop specific applications for various tasks in the future possible projects, theses and works. ITU-AGVs have built before ROS was developed. At the time when ITU-AGVs built, the software systems that used were different and custom so the software of the robots was written concerning them which became out-of-date now.

As mentioned previously in the Introduction chapter, the goal is to construct a set of applications for the basic problems and needs using up-to-date tools. It is desired to write the embedded code for LLPL so the robots can be communicate with ROS and to develop ROS applications for tele-operation, sensor integration and reading, odometry estimation, data collection and offline map building. With realization of this basis, ITU-AGVs can be used as multi-purpose indoor land vehicle kits available using rapidly for educational purposes, theses, autonomous system design and algorithm development at ITU Robotics Laboratory.

The processing work is divided with a hierarchy. Low Level Processing Layer (LLPL) is responsible for getting commands, communicating motor drivers to drive the motors as desired in the given commands, requesting encoder values and sending them to High Level Processing Layer. High Level Processing Layer is responsible for complex calculations and is the part where the ROS runs. The signal chart of ITU-AGVs can be seen in Figure 4.1. The project is started with LLPL work and then HLPL after.

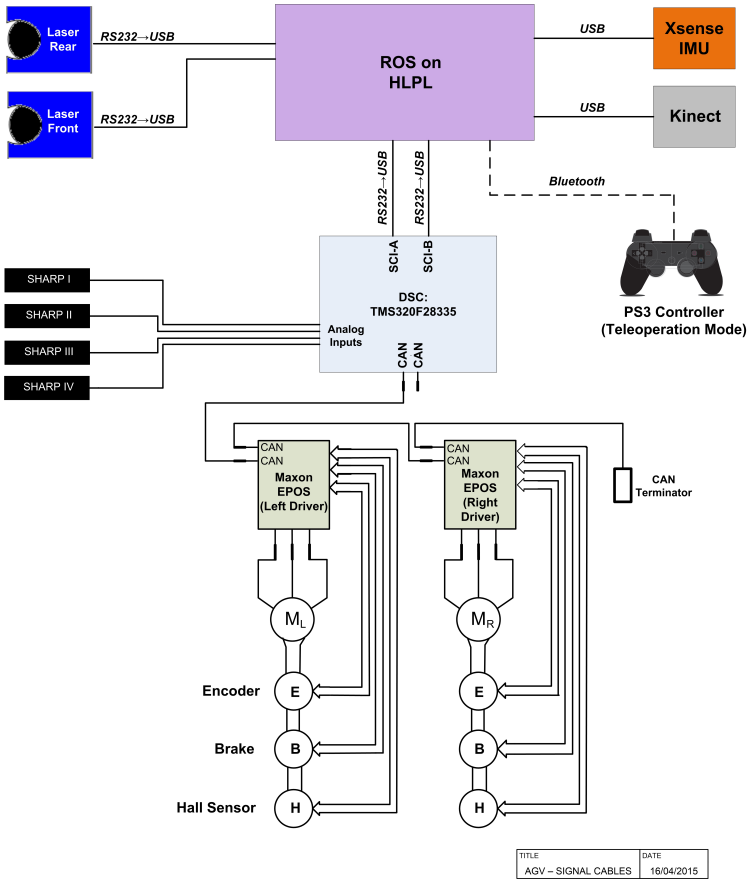

Figure 4.1: Signal chart of ITU-AGVs

4.1 Embedded Program Development for LLPL

The embedded software for TMS320F28335 microcontrollers at LLPL is developed using Simulink Embedded Target Coder in MATLAB r2012b and then the make files are uploaded to the microcontroller using TI Code Composer Studio v4.

4.1.1 Communication with EPOS Drivers

The Maxon EPOS 70/10 motor drivers are designed to be used with CANOpen protocol. In CAN communication several hardware are connected as slaves to a master using a CAN Bus. The communication and configuration occurs with using array of variables called objects. Object dictionary includes all object addresses with 16-bit index and 8-bit sub index.

EPOS drivers have their configurable controllers and there are several driving modes such as position mode, velocity mode, profile velocity mode and so on. It is desired to send velocity commands to the LLPL and to settle the motors on the desired velocity references. So the operation mode would be selected as the profile velocity mode.

According to the EPOS 70/10 Manual [3], the motors are controlled with given profile velocity and acceleration limits and selection of motion profile type. Motion profile type can be selected as linear or sinusoidal. It is desired to obtain a sinusoidal profile. In the manual, all configuration and communication object values and their places in the work flow are provided.

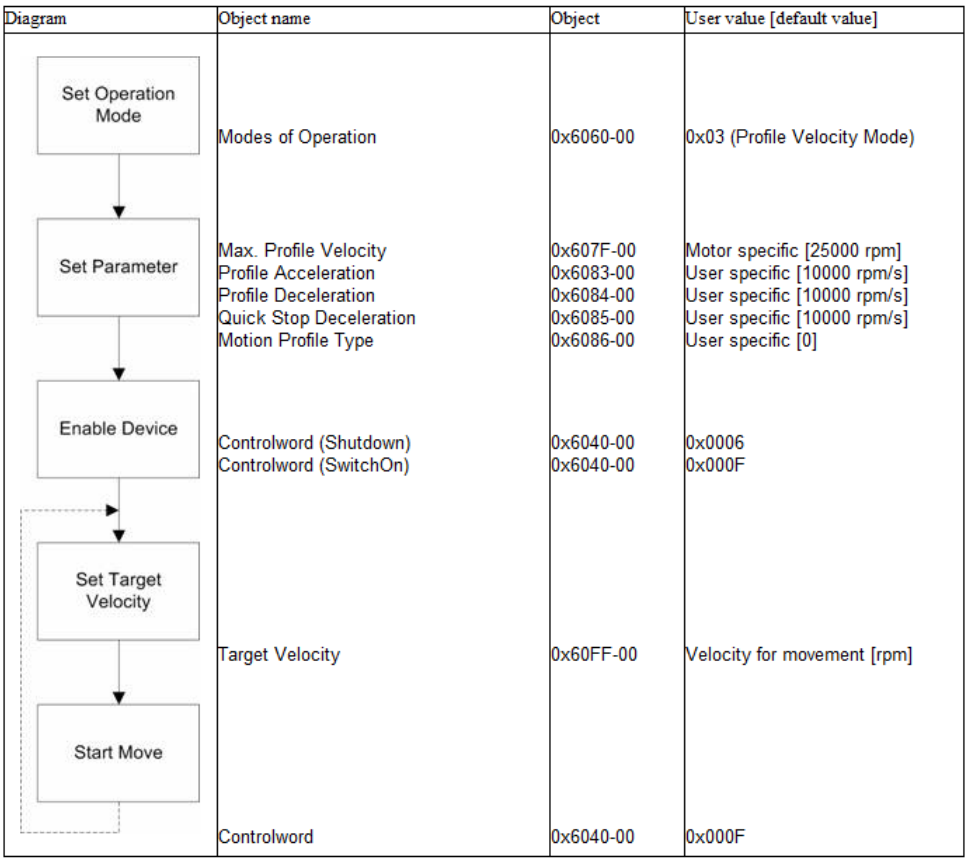

Figure 4.2: Configuration flow chart for profile velocity mode [3]

In Simulink, a state chart diagram is created in order to make the program flow as a state machine. In the program, according to the EPOS 70/10 Manual the necessary configurations are being made. First, all CANOpen nodes are being reset and all slaves are being set as operational. Then according to the chart in Figure 4.2, the operation mode is being selected as profile

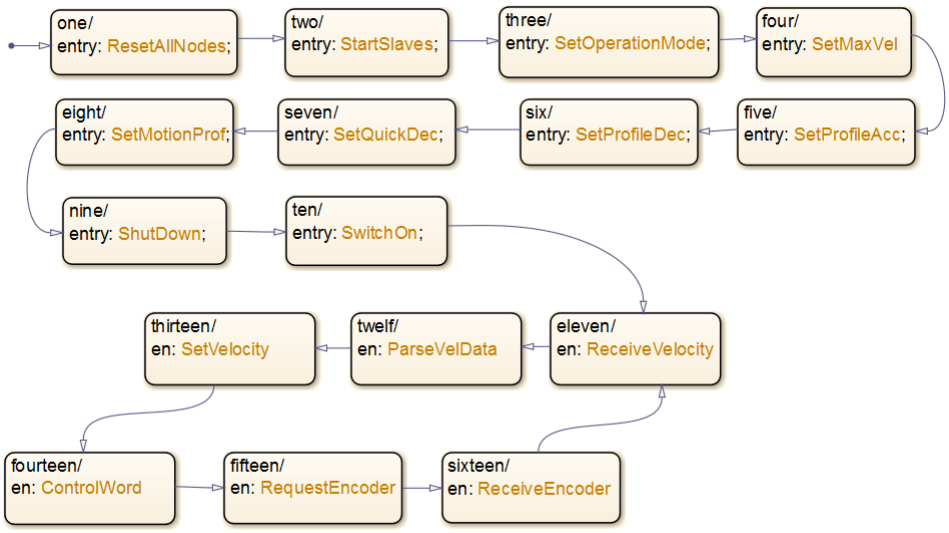

Figure 4.3: State chart flow diagram of LLPL in Simulink

velocity mode, the values for maximum profile velocity, profile acceleration, profile deceleration, quick stop deceleration and motion profile type are being configured over their objects and a necessary reset is being done. After these configurations, the program enters a loop. In the loop, the commands are being taken from HLPL over SCI-A (Serial Communication Interface - A) serially. The commands are being parsed and left and right motor commands are separated and set as target velocity values. The control word is being sent and encoder values are being requested. After the encoder values are received, they are being sent to HLPL over SCI-B and the loop begins again. The flow of the state chart diagram in Simulink can be seen in Figure 4.3.

4.1.2 Communication with HLPL

The velocity commands are designed to be in one sixteenth of desired rpm values at the motor shaft before gear-box. LLPL takes commands in 16-bit integers. The command word will be sent starting with "#" character and ending with "!" character. The first 16-bits after starting character will be the left motor command and the second 16-bits until the end character will be the right motor command. Commands are 16-bits and the first byte of each command is for direction and the second byte is for one sixteenth of the rpm value desired at the motor shaft before the gear-box. Before sending the values to the drivers, this value is multiplied by. It is important to remind the gear-box on the motor since it reduces the rpm with a ratio of 1:100.

For example, if it is wanted to drive the wheels at 40 rpm, a basic calculation can be made. The rpm value at the shaft of the motor before the gear-box would be $40 \times 100 = 4000$. This is the target rpm value, so the second byte of the command must have the value of $4000 \div 16 = 250$.

The direction is set such that, if the value of the direction byte is less than or equal to 127 it is counted as positive direction and the otherwise is negative direction. So the necessary word needed to be send to LLPL in order to drive wheels at 40 rpm in the positive direction should be [#0 250 0 250!]. The parsing is made in then made in LLPL.

EPOS 70/10 can provide various calculations with encoder values and it can give position and velocity. The encoder values are being sent in 16-bits to HLPL. Both SCI-A and SCI-B serial communications are set at 115200 baud.

4.1.3 Testing the LLPL

In order to test the embedded software and the serial communication a simple test script is written with Python. In the script the direction and desired wheel rpm values are requested from the user for left and right wheels and the command word is calculated and sent over serial port to the ITU-AGVs (Figure 4.4). After using this test script, it is concluded that the LLPL is functionally working and the project can be moved on to HLPL.

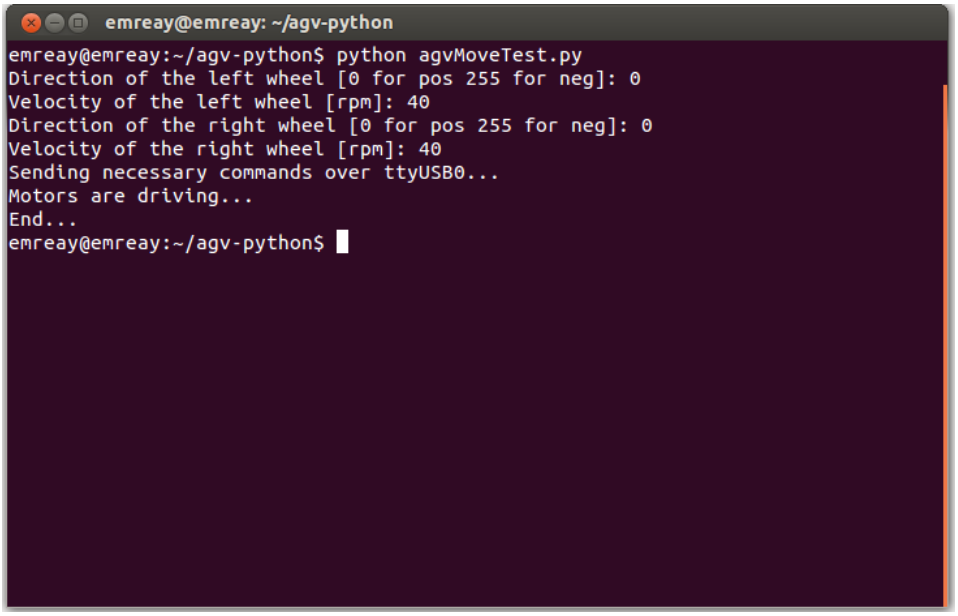

Figure 4.4: Testing LLPL with a written Python script

4.2 ROS Programming for HLPL

4.2.1 ROS Installation & System Initialize

Pre-compiled repositories for ROS distributions are officially supported and supplied for Ubuntu. The HLPL would be ROS running on Ubuntu 12.04 LTS on a notebook computer. ROS Groovy distribution is supported on Ubuntu 12.04 LTS so its complete packages are installed according to the directives at ROS Wiki Website.

After installation a workspace is needed to be created. Workspace is the container folder where all the packages and their relative files are stored. Catkin build system has a command to easily create a workspace for ROS.

There are several packages needed to be installed that are not included in the ROS core packages. These are usually specific work or hardware packages. Since the sensors to be installed are decided, their relative software driver packages for ROS are installed including ps3joy for Play Station 3 joystick, sicktoolbox wrapper for laser sensors and xsens driver for IMU.

To contain the applications that are going to be written, a ros package is needed to be created. Catkin also provides an easy package creation with a command and its several arguments. Packages usually depend on other packages in order to use their functionalities. So a package named *agv* is created with dependencies to *roscpp* and *rospy* packages. This package together with all the written ROS codes in this project is available at the link given in Appendix A.

4.2.2 Teleoperation Application

Teleoperation is a vital functionality for manual data collection or to move the robot to the application areas. It is needed to be able to move ITU-AGVs manually as desired in order to collect laser data or images for use in algorithm development on Simultaneous Localization and Mapping (SLAM), loop closure or similar applications.

4.2.2.1 Simulation

Before directly passing to work on ITU-AGVs, the teleoperation is desired to be applied on a simulation environment of ROS. To achieve this, a visual model has to be created. Rviz visualization environment supports an xml based format for basic 3D robot representation called Unified Robot Description Format (*.urdf). It is a simple parser with a simple syntax. So basically,

a box with to wheels in the dimensions and placement of the ITU-AGVs can be created by a urdf file as follows;

Urdf code for ITU-AGVs

```
<robot name="AGV">
    <link name="base_link">
        <visual>
            <geometry>
                <box size="0.82 .49 .2"/>
            </geometry>
            <origin rpy="0 0 0" xyz="0 0 0.01"/>
            <material name="gray">
                <color rgba="0 0 0 0.6"/>
            </material>
        </visual>
    </link>

    <link name="wheel_left">
        <visual>
            <geometry>
                <cylinder length="0.05" radius
                   ="0.1"/>
            </geometry>
            <origin rpy="1.57079633 0 0" xyz="0 -0.22
                0.01"/>
            <material name="black">
                <color rgba="1 0 0 1"/>
            </material>
        </visual>
    </link>

    <link name="wheel_right">
        <visual>
            <geometry>
                <cylinder length="0.05" radius
                   ="0.1"/>
            </geometry>
            <origin rpy="1.57079633 0 0" xyz="0 0.22
               0.01"/>
            <material name="black"/>
        </visual>
    </link>

    <joint name="base_to_wheel_left" type="continuous
```

```
      ">
       <parent link="base_link"/>
       <child link="wheel_left"/>
       <origin xyz="0 -0.22 0.01"/>
       <axis xyz="1 0 0"/>
  </joint>

  <joint name="base_to_wheel_right" type="
     continuous">
       <parent link="base_link"/>
       <child link="wheel_right"/>
       <origin xyz="0 0.22 0.01"/>
       <axis xyz="1 0 0"/>
  </joint>
</robot>
```

So basically the links and joints are geometrically defined with their positions, dimensions together with their relationship with each other. So their hierarchy tree can be understood by the system and it is possible to visually see the tree as in Figure 4.5. When the created urdf file is opened in Rviz, a simple box and two wheels with the given parameters can be seen as in Figure 4.6. This visualization is a simple and quick solution. But it is possible to integrate more realistic 3D models. Rviz supports Collada file format (*.dae) which is an open xml schema, so the 3D models created in various CAD software such as Google SketchUp can be converted to Collada file and implemented to Rviz [2]. So a 3D model of ITU-AGVs are created using Google SketchUp and it is converted to the Collada file (Figure 4.7).

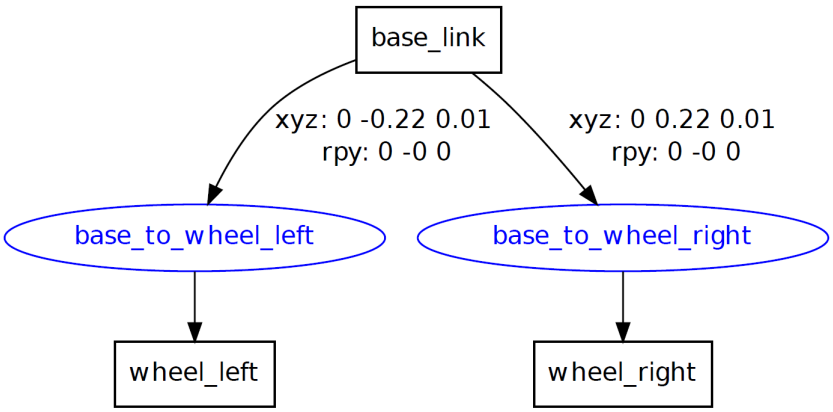

Figure 4.5: Urdf tree

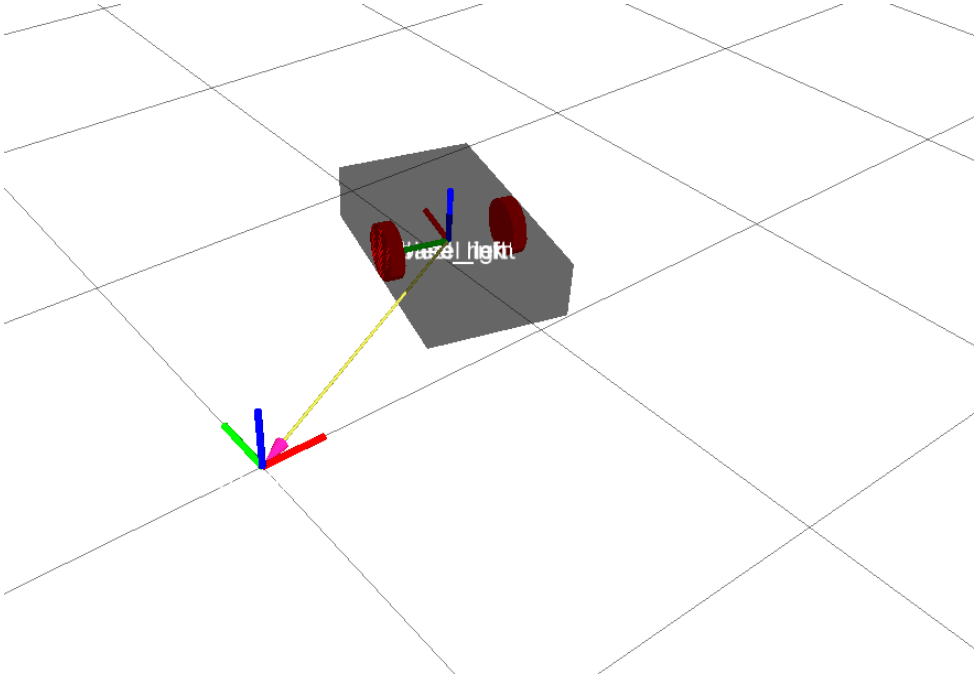

Figure 4.6: Urdf model of ITU-AGVs

It is wanted to realize the tele-operation using the Play Station 3 joystick. It is desired to control with both analog buttons and digital buttons. In the analog mode, the vertical values of left and right analog buttons will be the angular velocity references (in rpm) of left and right wheels. In digital mode, cross buttons will drive the motors on constant angular velocities. The up and down direction buttons will move the robot forwards and backwards, left and right buttons will drive the motors in the opposite directions resulting in a clockwise and a counter clockwise rotation around the central point. Lastly it is desired to only one mode at a time, hence while R2 button on the joystick is pushed digital mode would be active and otherwise analog mode would be active.

Using the installed ps3joy package, communication with PS3 joystick over Bluetooth is handled and the button values are published to ROS environment. A node that subscribed to the joystick topic is created. This node gets the joystick button values, and passes the values of the necessary buttons as the relevant joint's velocity with multiplying it with a scalar, then publishes all the joint states. Another node is written so that it is subscribed to the topic that joint states are published. When the joint states data is received this node passes the velocity data to the parameter server in the callback function. Then in the main loop, it calculates the odometry of the robot and publishes the odometry information.

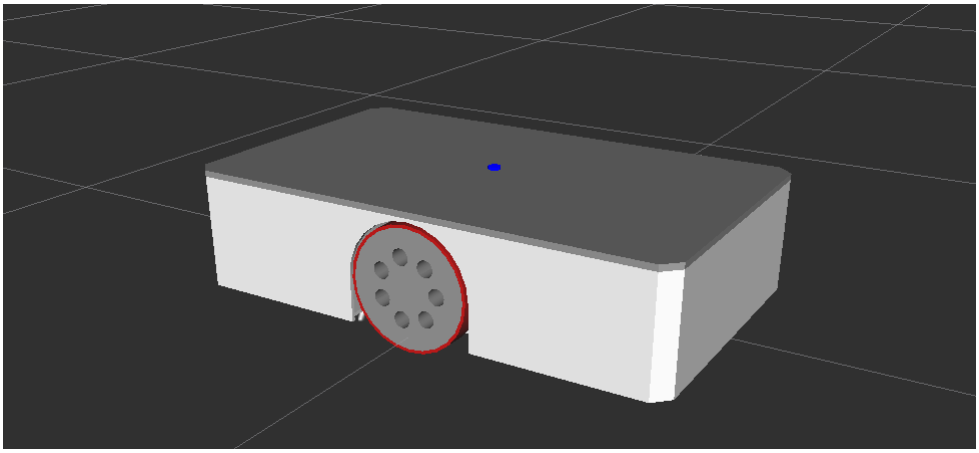

Figure 4.7: 3D model ITU-AGVs opened in Rviz

After building the nodes and launching them the 3D model of ITU-AGVs have successfully moved with using the PS3 joystick (Figure 4.8)

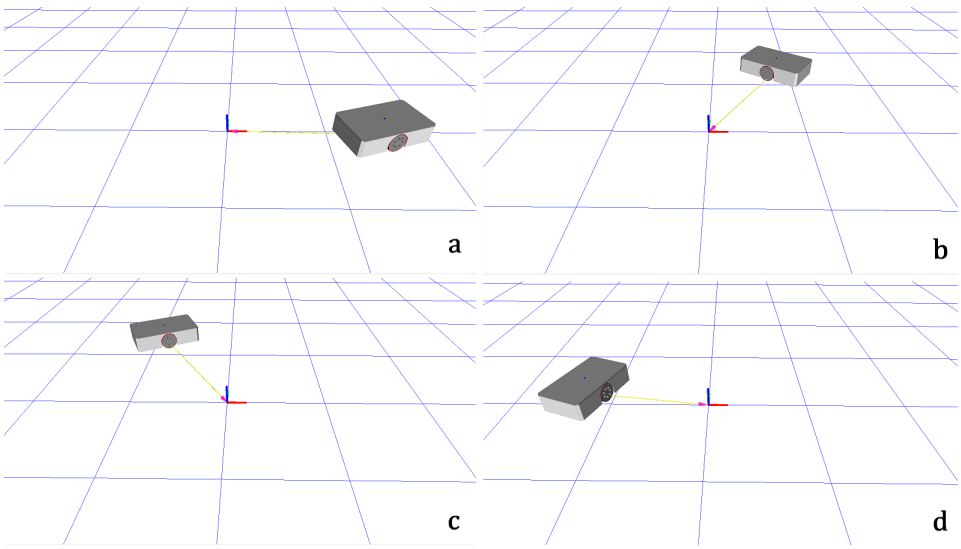

Figure 4.8: Teleoperation of ITU-AGV model in Rviz

4.2.2.2 Teleoperation of ITU-AGVs

Since the teleoperation is successfully made on the simulation, it is convenient to realize the teleoperation of ITU-AGVs. A similar but modified approach is made. A node is created so that it would subscribe to the joystick topic and every time the joystick data is received it takes the needed button

values. If the R2 button is pressed, it configures several variables depending on the values of digital cross buttons. If R2 button is not pressed, it configures the same variables depending on the analog button values. In the main loop, the node publishes an array of the variables which are configured in the callback function. This array is in the form that has been specified in Section 4.1.2. In order to send the commands to ITU-AGVs, another node is subscribed to the topic in which the array is published and it sends the array to LLPL of ITU-AGVs over a serial COM port. After the nodes are built, teleoperation of ITU-AGVs is successfully achieved. Node graph is shown for the teleoperation in Figure 4.9. It is possible to see the node graph in ROS in order to examine the relationships of nodes over topics and to see which nodes and topics are active. This is a powerful feature for diagnostics.

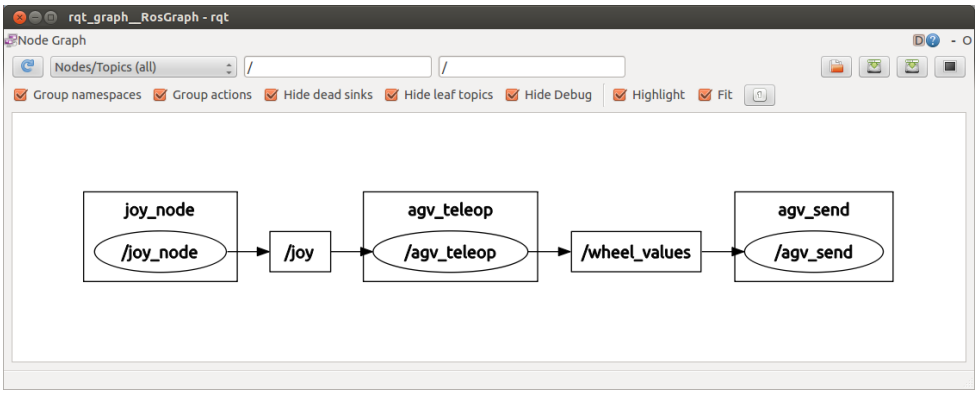

Figure 4.9: Node graph (rqt-graph) of teleoperation

4.2.3 Sensor Integration

Software packages of various sensors were installed during the system initialize. Since the teleoperation is applicable, reading stable data from the sensors on ROS is the next goal.

4.2.3.1 Encoder Reading

The encoder values are sending over SCI-B as they have configured to do in Section 4.1.2. In the HLPL, they have to be read. To read the encoder values, a Python node is written. In this node, the COM port assigned to SCI-B (ttyUSB1) is continuously listened. Since the sending format is the same (start character as "#", end character "!"), the node converts the unsigned 16-

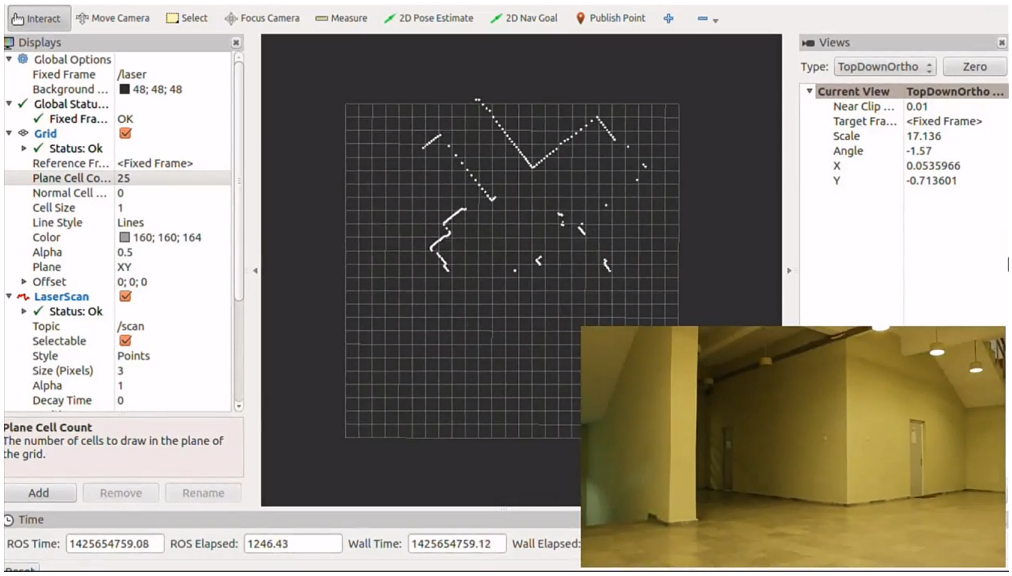

Figure 4.10: Simultanoues stream of LIDAR data on Rviz and the image of real environment

bit integer values to signed 16-bit values and publishes the converted values on a topic.

4.2.3.2 IMU Reading

The related ROS package for Xsens MTi IMU was installed before. Using this package a launch file is created and the node that publishes IMU data is successfully initialized. In order to calibrate the data and make certain settings the software provided by Xsens is used. As shown in Figure 4.11, the robot is rotated around its center for approximately 90 ° clockwise and counter clockwise and the yaw angle is plotted from the IMU data. It can be also seen that the robot settles smoothly in sinusoidal profile as the motor drivers are configured to do in Section 4.1.1.

4.2.3.3 LIDAR Reading

The related package publishes laser scan data on ROS environment. A launch file is created and the data scanned by the LIDAR is simultaneously published on Rviz as in the Figure 4.10.

4.2.3.4 Kinect Reading

OpenNI driver packages for Kinect are installed. After a launch file is created, both RGB and point cloud data are streamed to Rviz (Figure 4.12).

Figure 4.11: Yaw angle values is retrieved from IMU while the robot is rotating

4.2.4 Odometry Estimation

Odometry estimation is a vital step since it is not possible to robot to autonomously move, navigate and plan without the information of its position and orientation with respect to the environment. The basic odometry calculation is made using the left and right wheel velocities obtained from the motor encoders. However, this calculation alone might give inexact or wrong odometry information due to errors in the calculation of velocity or slippage of the robot wheels.

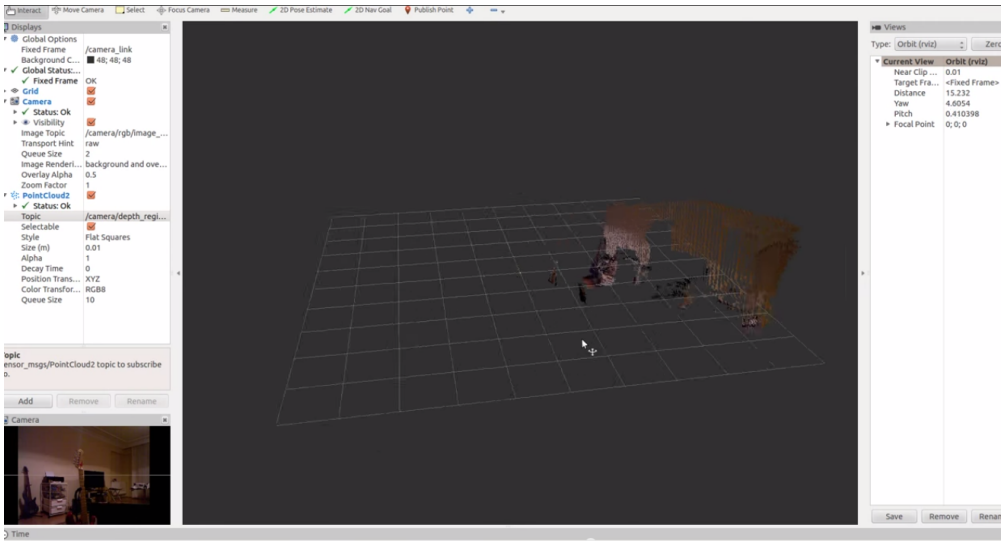

Figure 4.12: RGB and point cloud data stream from Kinect on Rviz

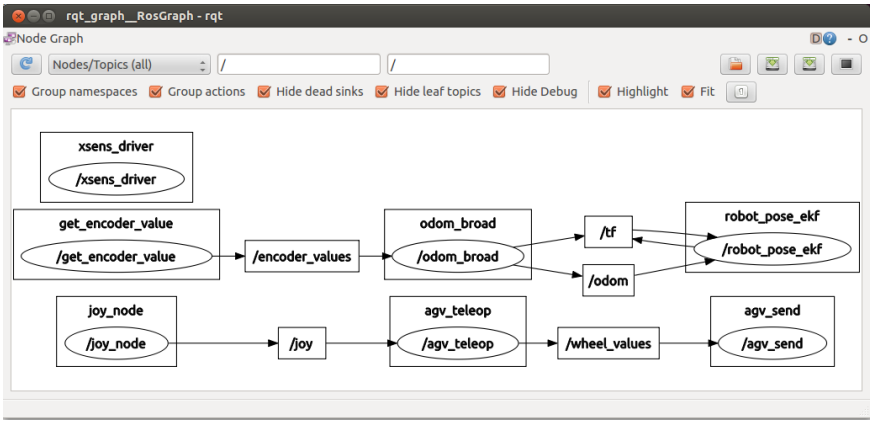

Figure 4.13: Node graph (rqt graph) while the robot pose ekf is publishing fused odometry data

To get better and more reliable odometry information, the present-day robotics systems use IMU data or vision along with the encoders. The process called data fusion is applied in these cases to integrate various data. There are various advanced methods for multi-sensor data fusion which are beyond the scope of this project. However, there is a ROS package that provides data fusion for IMU and encoder data to estimate the pose of a robot using Extended Kalman Filter (EKF) named robot pose ekf [11].

Extended Kalman Filter in this case estimates an optimal value for odometry from the data of IMU and encoder and with a covariance matrix that tells how much accurate the data are. Using robot pose ekf package, the fused odometry information can be obtained. The necessary launch file is created so the nodes that publish IMU data and encoder values are started and the fused odometry information is published on a topic. The node graph can be seen in Figure 4.13.

4.2.5 Offline Map Building

It is desired to build maps using collected data from the indoor environment. ROS can record and replay messages with makes it suitable for data collection. Since the odometry of ITU-AGVs can be estimated, sensors are installed, working and ready to publish data on ROS environment it is now possible to collect the necessary data while moving the ITU-AGVs to the desired areas.

A launch file for activating the nodes for IMU and encoder reading, odometry calculations and laser reading is created. ROS records data to bag files and it is very simple to record desired or all topics that are active at the time of record. In this case the laser data and fused odometry information

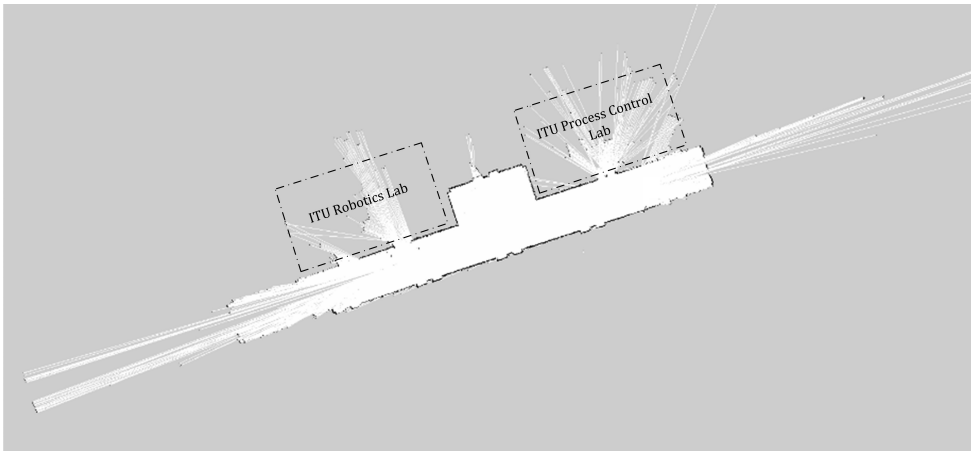

Figure 4.14: An offline map of ITU Control and Automation Eng. Department corridor is built with the collected data

is needed to build a map. One of the ITU-AGVs is moved using Play Station 3 joystick while the selected topics are recorded inside ITU Electric and Electronics Faculty. After the data collection is done, the recorded data is replayed and using another package called gmapping an offline map of Control and Automation Department corridor is built as shown in Figure 4.14.

Chapter 5

Conclusion & Future Work

During the project, the necessary maintenance is made for ITU-AGVs, new embedded software is developed and it is designed to be ready for communication with ROS. A ROS package is created for ITU-AGVs which contain all the source codes, scripts, launch files, urdf files and custom messages. A 3D model is built for use in visual simulations and representation, the created model is implemented in ROS environment. It is possible in the future to use this model for simulation of navigation, SLAM, multi-robot system and various trials that would be subject of other theses, works or learning projects at ITU Robotics Laboratory.

ITU-AGVs are equipped with numerous sensors, and all the sensors are ready to be used, their sensor packages are installed and tested, launch files for different sensor combinations are created.

Teleoperation application is developed and tested. ITU-AGVs can simply be moved using a joystick. This would help to transport the robots to the desired application areas as well as data collecting. Thesis students who only focuses on certain data processing on a moving robot, i.e. RGBD or laser data, does not need to deal with the background process. They can directly do the teleoperation.

The vital step of acquiring odometry information is done. This information is directly provided for future studies that strictly depend on it. Lastly an offline map is built by recording selected data and combining them. This is the first part into mapping and it should be continued to autonomously online mapping.

The necessary basis is formed as desired with both embedded and computer software. This solved the problem of ITU-AGVs for being out-of-date and made a contribution to ITU Robotics Laboratory.

Based upon this work, many improvements can be initiated in the future.

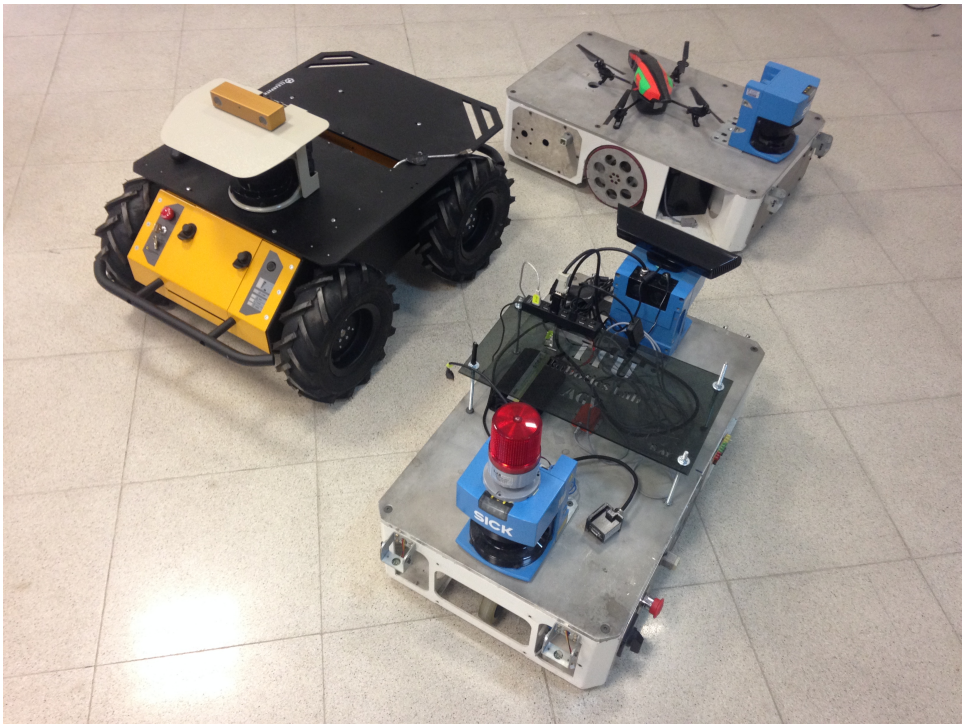

Figure 5.1: Husky robot, ITU-AGVs and AR Drone quadcopter at ITU Robotics Laboratory

The created ROS package is a prototype since the process of learning proceeded in parallel with the project. The package can be separated into specific packages, and a metapackage for them can be created. The codes can be review in order to use in newer distributions of ROS.

It is possible to develop autonomous applications based on this package. Navigation, SLAM, loop closure applications or swarm systems can be realized as well as heterogeneous robot groups. In ITU Robotics Laboratory, a group of robots are strategically gathered (Figure 5.1). A new outdoor robot kit Husky is bought and it is planned to work on with heterogeneous groups of Husky robot, AR Drone quadcopter and ITU-AGVs in outdoor or indoor, depending on the groups. ROS plays a significant role in the future projects and the package created in this project is the base step for this target.

Appendix A

Codes & Scripts

The written codes of the indicated nodes are given in the appendix in order to avoid the interruption of the flow. Only selected codes and scripts are given in the appendix. All the written codes and scripts are available at;

https://github.com/emreay-/agv

Odometry Broadcaster Node

```
#include<string>
#include<ros/ros.h>
#include<sensor_msgs/JointState.h>
#include<sensor_msgs/Joy.h>
#include<tf/transform_broadcaster.h>
#include<boost/bind.hpp>
#include<boost/ref.hpp>

//physical dimensions
const float wheelRadius = 0.1;
const float length = 0.8;

//callback function for subscribed joint states
void jsCallback(ros::NodeHandle &node_handle, const
   sensor_msgs::JointState::ConstPtr& jointState)
{
        double wL, wR;
        wL = jointState->velocity[1];
        wR = jointState->velocity[2];
        node_handle.setParam("wLeft",wL);
        node_handle.setParam("wRight",wR);
}

//broadcastTF function definition
```

```
void broadcastTF(tf::TransformBroadcaster *
   tf_broadcaster,
                std::string device_frame,
                double x, double y, double z,
                double theta)
{
        // broadcast Transform from vehicle to device
        geometry_msgs::TransformStamped odom_trans;
        odom_trans.header.stamp = ros::Time::now();
        odom_trans.header.frame_id = "odom";
        odom_trans.child_frame_id = device_frame;
        odom_trans.transform.translation.x = x;
        odom_trans.transform.translation.y = y;
        odom_trans.transform.translation.z = z;
        odom_trans.transform.rotation = tf::
           createQuaternionMsgFromYaw(theta);
        //broadcasting the transform
        tf_broadcaster->sendTransform(odom_trans);
}

//main function
int main(int argc, char** argv)
{
        double x = 0.0;
        double y = 0.0;
        double th = 0.0;
        double wLeft, wRight, vLeft, vRight, vx, wth,
            dt, delta_x, delta_y, delta_th;

        ros::init(argc, argv, "odom_transform"); //
           node initialize
        ros::NodeHandle nh_; //node handle object

        tf::TransformBroadcaster broadcaster; //tf
           broadcaster object
        ros::Subscriber js_sub_; //subscriber object
        //subscribe to joint_state_pub topic
        js_sub_ = nh_.subscribe<sensor_msgs::
           JointState>("joint_state_pub",
        100, boost::bind(jsCallback,boost::ref(nh_),
           _1));
        //time variables
        ros::Time current_time, prev_time;
        prev_time = ros::Time::now();
```

```
ros::Rate loop_rate(30); //set loop rate to
   30 Hz
while (ros::ok())
{
        ros::spinOnce(); //check new messages
        current_time = ros::Time::now(); //
           set current time

        nh_.getParam("wLeft",wLeft); //get
           left wheel angular velocity value
           from parameter server
        nh_.getParam("wRight",wRight); //get
           right wheel angular velocity value
            from parameter server

        vLeft = wheelRadius*wLeft; //linear
           velocity of left wheel
        vRight = wheelRadius*wRight; //linear
            velocity of rigth wheel
        vx = (vRight+vLeft)/2; //linear
           velocity in ROBOT frame
        wth = (vRight-vLeft)/length; //
           angular velocity of the ROBOT
        dt = (current_time - prev_time).toSec
           (); //time differential
        delta_x = vx*cos(th)*dt; //change in
           x axis in world frame
        delta_y = vx*sin(th)*dt; //change in
           y axis in world frame
        delta_th = wth*dt; //change in the
           orientation

        x += delta_x;
        y += delta_y;
        th += delta_th;
        //broadcast transforms
        broadcastTF(&broadcaster,"base_link",
           x,y,0,th);
        broadcastTF(&broadcaster,"wheel_left"
           ,x,y,0,th);
        broadcastTF(&broadcaster,"wheel_right
           ",x,y,0,th);
        //set previous time
        prev_time = current_time;
```

```
                //rest until loop rate is done
                loop_rate.sleep();
        }//end of while
        return 0;
}//end of main
```

Joint State Publihser Node

```
#include<string>
#include<ros/ros.h>
#include<sensor_msgs/JointState.h>
#include<sensor_msgs/Joy.h>
#include<tf/transform_broadcaster.h>

        //definition of the class TeleopAgv
        class TeleopAgv
        {
        public:
        TeleopAgv(); //constructor
        ~TeleopAgv(); //deconstructor

        private:
        //callback function for joystick
        void joyCallback(const sensor_msgs::Joy::
           ConstPtr& joy);
        //node handle object
        ros::NodeHandle nh_;

        int linearL_, linearR_;
        double l_scale_L, l_scale_R;
        ros::Publisher js_pub_; //publisher object
        ros::Subscriber joy_sub_; //subscriber object
        }; //end of class definition

        //class constructor definition
        TeleopAgv::TeleopAgv():
          linearL_(1),
          linearR_(2)
        {
                ROS_INFO("Calling␣the␣constructor␣of␣
                   class␣TeleopAgv");
                //getting necessary parameters from
                   parameter server
```

```
        nh_.getParam("axis_linear_L",
           linearL_);
        nh_.getParam("axis_linear_R",
           linearR_);
        nh_.getParam("scale_linear_L",
           l_scale_L);
        nh_.getParam("scale_linear_R",
           l_scale_R);
        //advertising to the topic
           joint_state_pub
        js_pub_ = nh_.advertise<sensor_msgs::
           JointState>("joint_state_pub", 1);
        //subscribing to the topic joy
        joy_sub_ = nh_.subscribe<sensor_msgs
           ::Joy>("joy",1000, &TeleopAgv::
           joyCallback, this);
}

//class deconstructor definition
TeleopAgv::~TeleopAgv()
{
        ROS_INFO("Calling␣the␣destructor␣of␣
           class␣TeleopAgv");
}

//joystick callback function definition
void TeleopAgv::joyCallback(const sensor_msgs
   ::Joy::ConstPtr& joy)
{
        sensor_msgs::JointState joint_state;
           //joint state object
        joint_state.header.stamp = ros::Time
           ::now(); //set header time stamp
        //resize the number of names that
           would be given to the joint states
        joint_state.name.resize(3);
        //resize the number of (angular)
           velocity values of the joint
           states
        joint_state.velocity.resize(3);
        //set the name of the first joint
        joint_state.name[0] ="base";
        //set the name of the second joint as
            left
```

```
                joint_state.name[1] ="left_wheel";
                //set the name of the third joint as
                   right
                joint_state.name[2] ="right_wheel";
                //set the velocity of left wheel from
                    the left analog joystick value
                joint_state.velocity[1] = l_scale_L*
                   joy->axes[linearL_];
                //set the velocity of left wheel from
                    the left analog joystick value
                joint_state.velocity[2] = l_scale_R*
                   joy->axes[linearR_];
                //publish joint states
                js_pub_.publish(joint_state);
        }

        //main function
        int main(int argc, char** argv)
        {
                //initialize the node
                ros::init(argc, argv, "jsPublisher");
                //construct a TeleopAgv object
                TeleopAgv teleop_agv;
                //spin
                ros::spin();
        }//end of main
```

Teleoperation Node

```
#include<string>
#include<ros/ros.h>
#include<sensor_msgs/Joy.h>
#include<tf/transform_broadcaster.h>
#include<agv/uint8Array.h>

        //definition of the class TeleopAgv
        class TeleopAgv
        {
        public:
        TeleopAgv(); //constructor
        ~TeleopAgv(); //deconstructor
        ros::Publisher wheel_pub_; //publisher object
        agv::uint8Array wheelArr;
```

```
private:
//callback function for joystick
void joyCallback(const sensor_msgs::Joy::
   ConstPtr& joy);
//node handle object
ros::NodeHandle nh_;
int analogL_, analogR_, right2_;
int select_button, cross_up, cross_right,
   cross_down, cross_left;
double l_scale_L, l_scale_R;
ros::Subscriber joy_sub_; //subscriber object
}; //end of class definition

//class constructor definition
TeleopAgv::TeleopAgv():
  analogL_(1),
  analogR_(2)
{
        ROS_INFO("Calling␣the␣constructor␣of␣
           class␣TeleopAgv");
        //getting necessary parameters from
           parameter server
        nh_.getParam("analog_left_upwards",
           analogL_);
        nh_.getParam("analog_right_upwards",
           analogR_);
        nh_.getParam("right2_", right2_);
        nh_.getParam("cross_up", cross_up);
        nh_.getParam("cross_right",
           cross_right);
        nh_.getParam("cross_down", cross_down
           );
        nh_.getParam("cross_left", cross_left
           );
        nh_.getParam("scale_linear_L",
           l_scale_L);
        nh_.getParam("scale_linear_R",
           l_scale_R);
        //advertising to the topic
           joint_state_pub
        wheel_pub_ = nh_.advertise<agv::
           uint8Array>("wheel_values", 1);
        //subscribing to the topic joy
```

```
        joy_sub_ = nh_.subscribe<sensor_msgs
            ::Joy>("joy",10, &TeleopAgv::
            joyCallback, this);
        wheelArr.data[0] = 35;
        wheelArr.data[1] = 0;
        wheelArr.data[2] = 0;
        wheelArr.data[3] = 0;
        wheelArr.data[4] = 0;
        wheelArr.data[5] = 0;
        wheelArr.data[6] = 0;
        wheelArr.data[7] = 0;
        wheelArr.data[8] = 0;
        wheelArr.data[9] = 33;
}

//class deconstructor definition
TeleopAgv::~TeleopAgv()
{
        ROS_INFO("Calling␣the␣destructor␣of␣
            class␣TeleopAgv");
}

//joystick callback function definition
void TeleopAgv::joyCallback(const sensor_msgs
    ::Joy::ConstPtr& joy)
{
        double left_wheel_raw,
            right_wheel_raw;
        short left_wheel_dir, right_wheel_dir
            , left_wheel_vel, right_wheel_vel;
        short crossUpState, crossRightState,
            crossDownState, crossLeftState,
            r2State;
        r2State = joy->buttons[right2_];

        if(r2State == 0)
        {
                left_wheel_raw = l_scale_L*
                    joy->axes[analogL_];
                right_wheel_raw = l_scale_R*
                    joy->axes[analogR_];

                if(left_wheel_raw<0)
                {
```

```
                        left_wheel_dir = 250;
                        left_wheel_vel = -28
                           * left_wheel_raw;
                }
                else
                {
                        left_wheel_dir = 0;
                        left_wheel_vel = 28 *
                            left_wheel_raw;
                }
                if(right_wheel_raw<0)
                {
                        right_wheel_dir =
                           250;
                        right_wheel_vel = -28
                            * right_wheel_raw
                           ;
                }
                else
                {
                        right_wheel_dir = 0;
                        right_wheel_vel = 28
                           * right_wheel_raw;
                }
        }

        else if(r2State == 1)
        {
                crossUpState = joy->buttons[
                   cross_up];
                crossRightState = joy->
                   buttons[cross_right];
                crossDownState = joy->buttons
                   [cross_down];
                crossLeftState = joy->buttons
                   [cross_left];

                if(crossUpState == 1 && (
                   crossRightState +
                   crossDownState +
                   crossLeftState) == 0)
                {
                        left_wheel_dir = 0;
                        right_wheel_dir = 0;
```

```
                left_wheel_vel = 250;
                right_wheel_vel =
                   250;
        }

        else if(crossRightState == 1
           && (crossDownState +
           crossLeftState +
           crossUpState) == 0)
        {
                left_wheel_dir = 0;
                right_wheel_dir =
                   250;
                left_wheel_vel = 125;
                right_wheel_vel =
                   125;
        }

        else if(crossDownState == 1
           && (crossLeftState +
           crossUpState +
           crossRightState) == 0)
        {
                left_wheel_dir = 250;
                right_wheel_dir =
                   250;
                left_wheel_vel = 250;
                right_wheel_vel =
                   250;
        }

        else if(crossLeftState == 1
           && (crossUpState +
           crossRightState +
           crossDownState) == 0)
        {
                left_wheel_dir = 250;
                right_wheel_dir = 0;
                left_wheel_vel = 125;
                right_wheel_vel =
                   125;
        }

        else
```

```
                        {
                                left_wheel_vel = 0;
                                right_wheel_vel = 0;
                        }
                }

                wheelArr.data[0] = 35;
                wheelArr.data[1] = left_wheel_dir;
                wheelArr.data[2] = 0;
                wheelArr.data[3] = left_wheel_vel;
                wheelArr.data[4] = 0;
                wheelArr.data[5] = right_wheel_dir;
                wheelArr.data[6] = 0;
                wheelArr.data[7] = right_wheel_vel;
                wheelArr.data[8] = 0;
                wheelArr.data[9] = 33;
        }

        //main function
        int main(int argc, char** argv)
        {
                //initialize the node
                ros::init(argc, argv, "agvTeleop");
                //construct a TeleopAgv object
                TeleopAgv teleop_agv;
                ros::Rate loop_rate(5);
                while(ros::ok())
                {
                        ros::spinOnce();
                        teleop_agv.wheel_pub_.publish
                           (teleop_agv.wheelArr);
                        loop_rate.sleep();
                }
        }//end of main
```

Node for sending the commands to LLPL

```
#!/usr/bin/env python
import rospy
from agv.msg import uint8Array
import struct
import serial

def callback(data):
```

```
    rospy.loginfo(data.data)
    port.write(data.data)

def agvSend():
    rospy.init_node('agvSendTest', anonymous=True)
    rospy.Subscriber("wheel_values", uint8Array,
       callback)
    # spin() simply keeps python from exiting until
       this node is stopped
    rospy.spin()

if __name__ == '__main__':
    port = serial.Serial("/dev/ttyUSB0", baudrate =
       115200, bytesize = 8, stopbits = 1, timeout =
       2)
    port.open()
    agvSend()
```

Launch file for Teleoperation

```
<launch>
        <!-- JOY NODE -->
        <node respawn="true" pkg="joy"
                type="joy_node" name="joy_node" >
                <param name="dev" type="string" value
                   ="/dev/input/js0" />
                <param name="deadzone" value="0.12"
                   />
        </node>

        <!-- PS3 JOYSTICK PARAMETERS -->
        <param name="analog_left_upwards" value="1"
           />
        <param name="analog_right_upwards" value="3"
           />
        <param name="right2_" value="9" />
        <param name="cross_up" value="4" />
        <param name="cross_right" value="5" />
        <param name="cross_down" value="6" />
        <param name="cross_left" value="7" />

        <!-- SCALAR PARAMETERS -->
        <param name="scale_linear_L" value="9" />
        <param name="scale_linear_R" value="9" />
```

```
        <!-- TF BROADCASTER NODE -->
        <node pkg="agv" type="agvTeleopnew"  name="
           agv_teleop" ></node>

        <node respawn="true" pkg="agv" type="agvSend.
           py" name="agv_send"></node>
</launch>
```

Node for getting encoder values from LLPL

```
#!/usr/bin/env python
import rospy
from agv.msg import int16Array
import struct
import serial
from std_msgs.msg import Char

def agvGet():
    pub = rospy.Publisher('encoder_values',
       int16Array)
    rospy.init_node('agvGet', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        values = port.read(20)
        for x in range(0,12):
                if values[x] == "#" and values[x+5]
                   == "!":
                        left_enc = (ord(values[x+1])
                           +(ord(values[x+2])*256))
                        if left_enc > 32767:
                                left_enc = left_enc -
                                    65536
                        right_enc = (ord(values[x+3])
                           +(ord(values[x+4])*256))
                        if right_enc > 32767:
                                right_enc = right_enc
                                    - 65536
                        msg = (35, left_enc/100,
                           right_enc/100, 33)
                        rospy.loginfo(msg)
                        pub.publish(msg)
        rate.sleep()

if __name__ == '__main__':
```

```
    port = serial.Serial("/dev/ttyUSB2", baudrate =
       115200, bytesize = 8, stopbits = 1, timeout =
       2)
    port.open()
    agvGet()
```

Bibliography

[1] Alonzo Kelly. *Mobile Robotics, Mathematics, Models and Methods*, page 3. Cambridge University Press, 2013.

[2] Aaron Martinez and Enrique Fernandez. *Learning ROS for Robotics Programming*, chapter 5, pages 156–158. Pack Publishing Ltd., 2013.

[3] Maxon Motor. *EPOS 70/10 Online Manual*, 2007.

[4] Microsoft Corporation. Specifications for the kinect. https://msdn.microsoft.com/en-us/library/jj131033.aspx. Accessed: 2015-05-25.

[5] NASA/JPL-Caltech. Mars rover curiosity, front view. http://www.jpl.nasa.gov/spaceimages/details.php?id=PIA14254, 2011. Accessed: 2015-03-15.

[6] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *International Conference on Robotics and Automation*, 2009. Open-Source Software Workshop.

[7] ROS Website. Core components. http://www.ros.org/core-components/. Accessed: 2015-05-25.

[8] ROS Wiki Website. Catkin conceptual overview. http://wiki.ros.org/catkin/conceptual_overview. Accessed: 2015-05-24.

[9] ROS Wiki Website. Installing on raspberry pi/raspbian. http://wiki.ros.org/groovy/Installation/Raspbian. Accessed: 2015-05-23.

[10] ROS Wiki Website. Navigating the ros filesystem. http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem. Accessed: 2015-05-24.

[11] ROS Wiki Website. Robot pose ekf. http://wiki.ros.org/robot_pose_ekf. Accessed: 2015-05-22.

[12] SICK AG. *LMS200/211/221/291 Laser Measurement Systems Technical Description*.

[13] Spectrum Digital, Inc. *eZdsp^{TM} F28335 Technical Reference*, November 2007.

[14] Xsens Technologies B.V. *MTi Miniature Attitude and Heading Reference System Leaflet*.