***Introduction to Robotic Systems Course***

**LAB 6B**

**Self-driving Robot**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc30074975)

[1.1 Lab Overview 1](#_Toc30074976)

[2 Requirements 1](#_Toc30074977)

[3 Robot Operation Specification 1](#_Toc30074978)

[4 Task: Modify OpenCR Firmware 1](#_Toc30074979)

[4.1 Edit turtlebot3\_core\_config.h file 2](#_Toc30074980)

[4.2 Edit turtlebot3\_core 2](#_Toc30074981)

[5 Task: Create a New ROS Package 3](#_Toc30074982)

[5.1 Edit Package.xml file 3](#_Toc30074983)

[5.2 Edit CMakeLists.txt file 4](#_Toc30074984)

[6 Task: Program the Robot 5](#_Toc30074985)

[6.1 Create self\_driving.py file 5](#_Toc30074986)

[6.2 Create setup.py file 7](#_Toc30074987)

[6.3 Build the Self-driving Package 7](#_Toc30074988)

[7 Task: Test the Program on the Robot 7](#_Toc30074989)

[8 Task: Test Self-driving Robot 8](#_Toc30074990)

[9 Task: Visualize the Graph 8](#_Toc30074991)

# Introduction

## Lab Overview

In this lab, the robot will be programmed to do line following when it detects a moving object in front of it. The IR and LDS sensors will track the black line and detect the moving object, respectively. The sensor outputs will be communicated to the self-driving control program using the ROS Computation Graph concepts. The control program will adjust the robot’s velocity to maintain a set distance from the followed object.

# Requirements

The following hardware and software are required to complete this lab:

* **Hardware:**
	+ TurtleBot 3 Burger (Robot).
	+ Four TCRT5000 IR reflector sensors.
* **Software:** Arduino IDE.

# Robot Operation Specification

The desired characteristics of the programmed robot is defined thus:

1. The robot stops moving when it fails to detect an object in front of it within a defined distance.
2. When an object is detected within the close range (e.g., 0.2-0.5 m), the robot will start following it.
3. A safety distance (e.g., 0.3 m) should be kept between the robot and the object.
4. The robot should adjust its velocity automatically, i.e., speed up to keep up with the object and decelerate or even reverse when too close to the object.
5. A speed limit should apply to the robot.
6. The robot should follow the line when moving.

# Task: Modify OpenCR Firmware

TurtleBot3 has a pre-installed firmware with an initial setup for the user to operate the TurtleBot3 using ROS, like the SLAM and Navigation operations in previous labs. In this lab, we will modify this software to suit our needs.

* Open Arduino IDE.
* Go to **File->Example->turtlebot3->turtlebor3\_burger->turtlebot3\_core**.

This will open the **turtlebot3\_core** program and its header files in three tabs. Now, we need to add some code to these files to publish the IR sensor data.

## Edit turtlebot3\_core\_config.h file

* Click on the **turtlebot3\_core\_config.h** tab.
* First, include the Byte header by entering the following at the top region of the file where other headers are included.

#include <std\_msgs/Byte.h>

* Next, define the macros by entering the following at the top region of the file.
1. #define IR\_PUBLISH\_FREQUENCY 20 //hz
2. // define IR sensor pins
3. #define sensor\_LL A2
4. #define sensor\_ML A0
5. #define sensor\_MR A1
6. #define sensor\_RR A3
* Next, scroll down to the “function prototype” section to declare the function that reads the IR data.

void publishIRsensorReading(void);

* Next, scroll down to the “Publisher” section. Here, several topics are published including those from sensors, inertial measurement unit, odometry, battery information, etc. At the end of the section, add the following code to create a publisher **pub\_ir** that publishes the “IR” message.
1. //IR sensor
2. std\_msgs::Byte IR\_msg;
3. ros::Publisher pub\_ir("IR", &IR\_msg);

## Edit turtlebot3\_core

* On the Arduino window, click the **turtlebot3\_core** tab.
* In the include region, add the following line of code.

#include <std\_msgs/Byte.h>

* In the “**Setup()”** function, “advertise” the created publisher using the following line.

nh.advertise(pub\_ir);

* In the **loop()** function, call the **publishIRsensorReading** periodically according to the **IR\_PUBLISH\_FREQUENCY**. Add the following lines.
1. if ((t - tTime[6]) >= (1000 / IR\_PUBLISH\_FREQUENCY))
2. {
3. publishIRsensorReading();
4. tTime[6] = t;
5. }
* Finally, define the content of **publishIRsensorReading()** function as follows:
1. void publishIRsensorReading(void)
2. {
3. IR\_msg.data = (analogRead(sensor\_LL) > 700)<<3|(analogRead(sensor\_ML) > 700)<<2|(analogRead(sensor\_MR) > 700)<<1|(analogRead(sensor\_RR) > 700)<<0;
4. pub\_ir.publish(&IR\_msg);
5. }
* After modifying the firmware:
	+ Click **File** -> **Save** and give the sketch a name.
	+ Compile and upload it to the board.

# Task: Create a New ROS Package

* Open a terminal on the Remote PC. Change directory to **~/catkin\_ws/src** and enter the following command to create a new package called “self\_driving,” which depends on various other packages such as std\_msgs rospy, geometry\_msgs, and turtlebot3\_msgs.

$ catkin\_create\_pkg self\_driving message\_generation std\_msgs rospy geometry\_msgs turtlebot3\_msgs

* Open the created directory at **~/catkin\_ws/src/self\_driving** and open the “src” folder, which should contain these two files: **package.xml** and **CMakeLists.txt**. Next, you will edit these files.

## Edit Package.xml file

* Open **package.xml**. This is the ROS configuration file that contains information about the package such as package name, author, license, and dependent packages. For now, let’s keep the file as shown below. Enter your email for maintainer email element.

<?xml version="1.0"?>

<package format="2">

 <name>self\_driving</name>

 <version>0.0.0</version>

 <description>The self\_driving package</description>

 <maintainer email="yours@todo.todo">arm</maintainer>

 <license>TODO</license>

 <buildtool\_depend>catkin</buildtool\_depend>

 <build\_depend>geometry\_msgs</build\_depend>

 <build\_depend>message\_generation</build\_depend>

 <build\_depend>message\_runtime</build\_depend>

 <build\_depend>rospy</build\_depend>

 <build\_depend>std\_msgs</build\_depend>

 <build\_depend>turtlebot3\_msgs</build\_depend>

 <build\_export\_depend>geometry\_msgs</build\_export\_depend>

 <build\_export\_depend>message\_generation</build\_export\_depend>

 <build\_export\_depend>message\_runtime</build\_export\_depend>

 <build\_export\_depend>rospy</build\_export\_depend>

 <build\_export\_depend>std\_msgs</build\_export\_depend>

 <build\_export\_depend>turtlebot3\_msgs</build\_export\_depend>

 <exec\_depend>geometry\_msgs</exec\_depend>

 <exec\_depend>message\_generation</exec\_depend>

 <exec\_depend>message\_runtime</exec\_depend>

 <exec\_depend>rospy</exec\_depend>

 <exec\_depend>std\_msgs</exec\_depend>

 <exec\_depend>turtlebot3\_msgs</exec\_depend>

</package>

## Edit CMakeLists.txt file

* Open the **CMakeLists.txt** file. This is the build configuration file required by the CMake build system. Edit it as the follows:

cmake\_minimum\_required(VERSION 2.8.3)

project(self\_driving)

## Find catkin macros and libraries

find\_package(catkin REQUIRED COMPONENTS

 geometry\_msgs

 message\_generation

 rospy

 std\_msgs

 turtlebot3\_msgs

)

catkin\_python\_setup()

## Generate added messages and services with any dependencies listed here

 generate\_messages(

 DEPENDENCIES

 geometry\_msgs

 std\_msgs

 turtlebot3\_msgs

 )

## catkin specific configuration ##

catkin\_package(

 CATKIN\_DEPENDS geometry\_msgs message\_generation rospy std\_msgs turtlebot3\_msgs message\_runtime

)

## Build ##

## Specify additional locations of header files

include\_directories(

 include

 ${catkin\_INCLUDE\_DIRS}

)

## Install ##

# all install targets should use catkin DESTINATION variables

catkin\_install\_python(PROGRAMS

 src/self\_driving.py

 DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

# Task: Program the Robot

## Create self\_driving.py file

Next, create the source file that contains the node of the self-driving robot. Data from the IR sensor and the LDS sensor will be fed to the node. It computes the velocity and direction of steer of the robot from the data received.

Open the “src” folder inside the package directory and create a file **self\_driving.py**. Copy the following Python code to the file.

#!/usr/bin/env python

import rospy

import std\_msgs.msg

from sensor\_msgs.msg import LaserScan

from geometry\_msgs.msg import Twist

class SelfDriving():

 def \_\_init\_\_(self):

 self.kp=0.4

 self.ki=0.1

 self.kd=0.07

 self.nowTime=rospy.get\_time()

 self.lastTime=self.nowTime

 self.error=0

 self.lastErr=0

 self.errsum=0

 self.dTime=0

 self.dErr=0

 self.IR=15

 self.speed\_limit=-0.1

 self.maximum\_target=0.5

 self.LIDAR\_ERR = 0.02

 self.safety\_distance= 0.3

 self.\_cmd\_pub = rospy.Publisher('cmd\_vel', Twist, queue\_size=1)

 self.IR\_sub = rospy.Subscriber('IR', std\_msgs.msg.Byte, self.get\_IR, queue\_size=1)

 self.twist = Twist()

 self.motoring()

 def get\_scan(self):

 scan = rospy.wait\_for\_message("scan",LaserScan)

 self.scan\_filter=[]

 self.scan\_filter.append(1000000)

 for i in range(360):

 if i>=150 and i < 210:

 if scan.ranges[i] >= self.LIDAR\_ERR:

 self.scan\_filter.append(scan.ranges[i])

 self.detect\_target()

 def detect\_target(self):

 if 0.1<=min(self.scan\_filter)<=self.maximum\_target:

 self.target\_found=True

 else:

 self.target\_found=False

 def get\_IR(self, IRsensor):

 self.IR=IRsensor.data

 def PID\_control(self):

 if self.target\_found:

 self.nowTime=rospy.get\_time()

 self.dTime=self.nowTime-self.lastTime

 self.target\_distance=min(self.scan\_filter)

 self.error=self.target\_distance-self.safety\_distance

 self.errsum+=self.error\*self.dTime

 self.dErr=(self.error-self.lastErr)/self.dTime

 self.motor\_lin\_vel=-(self.kp\*self.error+self.ki\*self.errsum+self.kd\*self.dErr)

 self.lastTime=self.nowTime

 self.lastErr=self.error

 else:

 self.motor\_lin\_vel=0

 self.error=0

 self.errsum=0

 self.lasterror=0

 self.dErr=0

 def motoring(self):

 while not rospy.is\_shutdown():

 self.get\_scan()

 if min(self.scan\_filter) < 0.1 or min(self.scan\_filter)==1000000 :

 self.twist.linear.x = 0.0

 self.twist.angular.z = 0.0

 self.\_cmd\_pub.publish(self.twist)

 rospy.loginfo('Stop!')

 else:

 self.PID\_control()

 self.twist.linear.x=max(self.motor\_lin\_vel,self.speed\_limit)

 if (self.IR<=7):

 self.twist.angular.z = -0.25\*(bin(self.IR).count("1"))

 elif (self.IR<15):

 self.twist.angular.z = 0.25\*(bin(self.IR).count("1"))

 else :

 self.twist.linear.x = 0.0

 self.twist.angular.z = 0.0

 rospy.loginfo('dTime : %f', self.dTime)

 rospy.loginfo('distance of the obstacle : %f', min(self.scan\_filter))

 rospy.loginfo('error : %f', self.error)

 rospy.loginfo('errsum of the obstacle : %f', self.errsum)

 rospy.loginfo('dErr : %f', self.dErr)

 self.\_cmd\_pub.publish(self.twist)

def main():

 rospy.init\_node('self\_driving')

 try:

 selfdriving = SelfDriving()

 except rospy.ROSInterruptException:

 pass

if \_\_name\_\_ == "\_\_main\_\_":

 main()

## Create setup.py file

Before building the package, create a **setup.py** file and locate this file in the same directory as the **package.xml** and **CMakeLists.txt** files.

* Copy the following into the file.

## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD

from distutils.core import setup

from catkin\_pkg.python\_setup import generate\_distutils\_setup

# fetch values from package.xml

setup\_args = generate\_distutils\_setup(

 packages=['self\_driving'],

 package\_dir={'': 'src'},

)

setup(\*\*setup\_args)

* Save and close the file.

## Build the Self-driving Package

* Open a terminal and enter the following command to build the package that should not yield any errors.

$ cd ~/catkin\_ws && catkin\_make

* Double-check this using **rospack list | grep self\_driving** command. You will see your package name and the path to the package directory.



* To make the file executable, go to the package folder and run the following command.

$ chmod u+x self\_driving

# Task: Test the Program on the Robot

After uploading the firmware edited in Arduino, reconnect the OpenCR1.0 board to the TurtleBot PC. Make sure both the Remote PC and the Pi are in the same network and connected to the Internet. Then, follow the steps below to test the self-driving capabilities of the robot.

* On remote PC, open a terminal and run

$ roscore

* Open a second terminal and ssh into the Turtlebot PC.
* On the Turtlebot PC, run the **Bringup** program.

$ roslaunch turtlebot3\_bring turtlebot3\_robot.launch

Now, the Turtlebot is ready for us to run the package and node we just created.

* On the Remote PC, run the following command to launch the ”self\_driving” node in the “self\_driving” package.

$ rosrun self\_driving self\_driving.py

# **Task: Test Self-driving Robot**

* + Place the robot on a track (black line).
	+ Place an object in front of it to let it follow.
	+ The robot should adjust its speed as it follows the object. It’s recommended to tune the PID control parameters so that the robot accelerates or decelerates smoothly.
	+ Place the object very close to the LDS sensor to trigger emergency stop.

# Task: Visualize the Graph

You can use “rqt” to visualize the node information of the ROS system as shown in Figure 1 and run the following command:

$ rqt



*Figure 1: Node diagram of the ROS Graph.*