SIT310 – Task 6
Navigation Stack II

Overview

This task focuses on using the ROS navigation stack to detect and visualise objects.

Task requirements

a. Ensure you have already completed tasks 1-5 and assessments 1 and 2.
b. To complete this task, you will need to use ROS on the raspberry Pi as we will be using your physical robot
c. If you didn't attend the lecture this week, then either watch the lecture online and/or review the PowerPoint slides on the unit site.
d. Read the task instructions
e. Complete the activities in the task guide
f. Review your progress with your lab tutor or cloud tutor.

Task guide

Last week we created some custom code to control your robot and publish messages using ROS topics about its location. We have also used transformations to visualise the robot using rviz. We have also left turtlesim behind. In weeks 3 and 4 we collected very basic sensor data from the robot and used this to build a simple 'scared robot' demo. You should have also done more complex activities for your assessment 2. This week we will concentrate on creating ROS navigation stack compliant sensor data. We will detect and visualise objects using ROS transforms and visualise them in rviz. We will then build a simple subsumption architecture to use this data to navigate in a more effective way.

1. Sensor data in ROS.

    1.1. ROS supports many types of sensor data, including lasers, fluid pressure, light, satellite, lasers, and temperature. See here for more details: http://wiki.ros.org/sensor_msgs

    1.2. For the purposes of this task, we are most interested in object detection – for e.g. avoiding walls or pets when robot is moving. The two most useful approaches to this are using LaserScan and using PointCloud2. LaserScan supports using a motorised laser to scan an area and produce a detailed map – we don't have a laser though, and these can be expensive. PointCloud2 allows arbitrary coordinates to be described, processed and visualised. For more information about these options, see here: http://wiki.ros.org/navigation/Tutorials/RobotSetup/Sensors

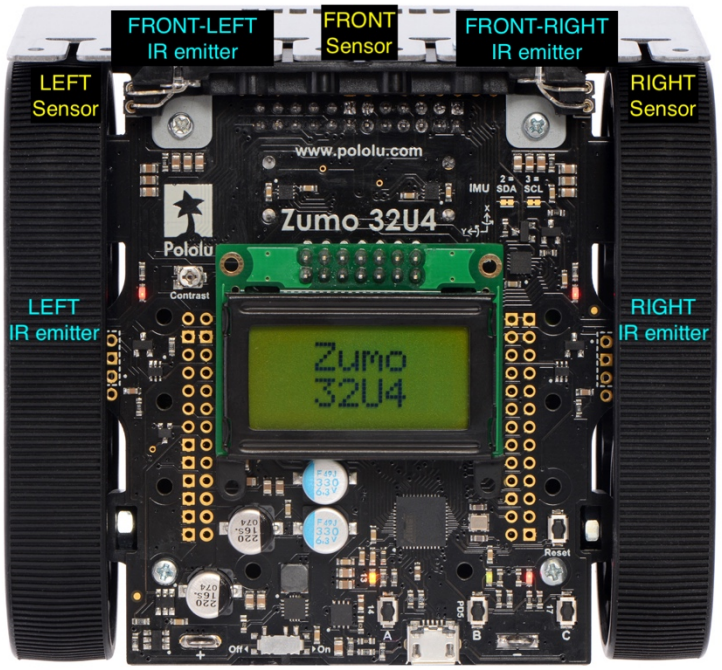
    1.3. A PointCloud2 is a simple data type, apart from information such as timestamps, it is just an array of X, Y and Z coordinates. In the next section we will be using a PointCloud to capture Zumo sensor data.

2. Zumo sensor data

    2.1. The Zumo 32U4 robot has a front powerful sensor array. Have a read of the details here https://www.pololu.com/docs/0J63/3.5

    2.2. For now, we are mainly interested in the Proximity sensors. Have a read of the details here https://www.pololu.com/docs/0J63/3.6

    2.3. The following is a simple diagram showing the sensors on the Zumo 32U4. One of the interesting characteristics of this robot is the separation of the IR emitter from the individual sensors. So, you can use any IR emitter to feed light to any Sensor. This allows you to detect objects around a large amount of the robot.



2.4. According to the Zumo API: "The readings generated by the read() function consist of two numbers for each sensor: the number of brightness levels for the left LEDs that activated the sensor, and the number of brightness levels for the right LEDs that activated the sensor. A higher reading corresponds to more IR light getting reflected to the sensor, which is influenced by the size, reflectivity, proximity, and location of nearby objects. However, the presence of other sources of 38 kHz IR pulses (e.g. from another robot) can also affect the readings".

2.5. The Zumo sensors are designed for object detection rather than distance measurements. To, perform proper distance measurement, you can attach a standard Sharp Ultrasonic sensor like one you have used before. For navigation and object detection purposes, the sensors on the Zumo are perfect.

2.6. The default configuration is as follows. The higher the brightness level, the brighter the IR beam – therefore detecting further away objects. The manufacturer state that the useful values are between 4 and 120. The query functions, e.g. countsLeftWithLeftLeds, will return the number of brightness levels returned.

```
uint16_t defaultBrightnessLevels[] = { 4, 15, 32, 55, 85, 120 };
setBrightnessLevels(defaultBrightnessLevels, 6);
```

2.7. The unit chair tested different levels in a medium light room – and found that something like the following works well to measure close objects – with a high number of 8 meaning something is very close.

```
uint16_t defaultBrightnessLevels[] = { 1,2,3,4,5,6,7,8,9,10};
setBrightnessLevels(defaultBrightnessLevels, 10);
```

2.8. We need to be able to publish sensor data from the Zumo. So, upload the following code to your Zumo using the Arduino IDE.

```
#define USE_USBCON
#include <Wire.h>
#include <Zumo32U4.h>
#include <ros.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/Int8.h>

ros::NodeHandle nh;

Zumo32U4ProximitySensors proxSensors;
bool proxLeftActive;
bool proxFrontActive;
bool proxRightActive;
std_msgs::Int8 prox_msg;
ros::Publisher pub_left("/zumo/prox_left", &prox_msg);
ros::Publisher pub_frontleft("/zumo/prox_frontleft", &prox_msg);
ros::Publisher pub_frontright("/zumo/prox_frontright", &prox_msg);
ros::Publisher pub_right("/zumo/prox_right", &prox_msg);

void ros_handler( const geometry_msgs::Twist& cmd_msg) {
float x = cmd_msg.linear.x;
float y = cmd_msg.linear.y;
int speed = 100;

if(x == 1.0) forward(speed);
if(x == -1.0) backward(speed);
if(y == 1.0) left(speed);
if(y == -1.0) right(speed);
stop();
}

ros::Subscriber<geometry_msgs::Twist> sub("/zumo/cmd_vel", ros_handler);
Zumo32U4Motors motors;

void setup()
```

```
{
  nh.initNode();
  nh.subscribe(sub);
  nh.advertise(pub_left);
  nh.advertise(pub_frontleft);
  nh.advertise(pub_frontright);
  nh.advertise(pub_right);

  proxSensors.initThreeSensors();
  uint16_t defaultBrightnessLevels[] = {1,2,3,4,5,6,7,8,9,10};
  proxSensors.setBrightnessLevels(defaultBrightnessLevels, 10);
}


void publishSensorData()
{
  prox_msg.data = proxSensors.countsLeftWithLeftLeds();
  pub_left.publish( &prox_msg);
  prox_msg.data = proxSensors.countsFrontWithLeftLeds();
  pub_frontleft.publish( &prox_msg);
  prox_msg.data = proxSensors.countsFrontWithRightLeds();
  pub_frontright.publish( &prox_msg);
  prox_msg.data = proxSensors.countsRightWithRightLeds();
  pub_right.publish( &prox_msg);
}


void forward(int time)
{
motors.setLeftSpeed(100);
motors.setRightSpeed(100);
delay(time);
}

void backward(int time)
{
motors.setLeftSpeed(-100);
motors.setRightSpeed(-100);
delay(time);
}

void left(int time)
{
motors.setLeftSpeed(-100);
motors.setRightSpeed(100);
delay(time);
```

```
}

void right(int time)
{
motors.setLeftSpeed(100);
motors.setRightSpeed(-100);
delay(time);
}

void stop()
{
motors.setLeftSpeed(0);
motors.setRightSpeed(0);
}

void loop()
{
static uint16_t lastSampleTime = 0;
  if ((uint16_t)(millis() - lastSampleTime) >= 100)
    {
    lastSampleTime = millis();
    // Send IR pulses and read the proximity sensors.
    proxSensors.read();
    publishSensorData();
    }

nh.spinOnce();
delay(1);
}
```

2.9. You should test this code is working, as follows:

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
$ rostopic echo /zumo/prox_left
$ rostopic echo /zumo/prox_frontleft
$ rostopic echo /zumo/prox_frontright
$ rostopic echo /zumo/prox_right
```

You should get different values depending on how close objects are to the zumo sensors. Make sure you are comfortable with the sensor data in respect to how close objects are.

2.10. Now that we have confirmed that we have sensor data from the zumo reaching ROS using topics we now need to convert this data into a PointCloud2. Setup a new project as follows:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg lab6 tf2 tf2_ros rospy
Check everything is fine.
$ cd ~/catkin_ws
catkin_make
If you added the environment file from the previous lab, then just open a new terminal window (or tab). Else:
$ source ./devel/setup.bash
$ roscd lab6
$ mkdir scripts
```

2.11. Create a new python node as follows:

```
$ cd scripts
$ gedit zumo_tf_sensor.py
```

2.12. The following code collects sensor data and the location of the zumo and creates a CloudPoint2 for detected objects in the front and side of the zumo. Paste it into the file. Make it executable when you have copied it.

```
#!/usr/bin/env python
import rospy
import math
import sys
from time import sleep
import tf

import tf2_ros
import tf2_msgs.msg
import tf_conversions

from tf2_sensor_msgs.tf2_sensor_msgs import do_transform_cloud
from sensor_msgs.msg import PointCloud2
import std_msgs.msg
from std_msgs.msg import Int8
import sensor_msgs.point_cloud2 as pcl2
import geometry_msgs.msg

wall_left, wall_frontleft, wall_frontright, wall_right = 0, 0, 0, 0

transx, transy, transz = 0, 0, 0
rotx, roty, rotz, rotw = 0, 0, 0, 0
```

```
def publish_walls():

        #build the point cloud imagining we are at 0,0,0.
        cloud_points = []
        #if(wall_left>):???
        if(wall_frontleft>0): cloud_points.append([13.0/wall_frontleft, 0.5, 0.5])
        if(wall_frontright>0): cloud_points.append([13.0/wall_frontright, -0.5, 0.5])
        #if(wall_right>):???


        header = std_msgs.msg.Header()
        header.stamp = rospy.Time.now()
        header.frame_id = 'world'
        mypointcloud = pcl2.create_cloud_xyz32(header, cloud_points)

        #We need to build a transformation of our location.
        t = geometry_msgs.msg.TransformStamped()
        t.header.stamp = rospy.Time.now()
        t.header.frame_id = "world"
        t.child_frame_id = "zumo"
        t.transform.translation.x = transx
        t.transform.translation.y = transy
        t.transform.translation.z = transz
        t.transform.rotation.x = rotx
        t.transform.rotation.y = roty
        t.transform.rotation.z = rotz
        t.transform.rotation.w = rotw

        #convert the pointcloud to take into account our location.
        cloud_out = do_transform_cloud(mypointcloud, t)
        pcl_pub.publish(cloud_out)

def handle_zumo_left(wall_msg):
        global wall_left
        wall_left = wall_msg.data
        publish_walls()

def handle_zumo_frontleft(wall_msg):
        global wall_frontleft
        wall_frontleft = wall_msg.data
        publish_walls()

def handle_zumo_frontright(wall_msg):
        global wall_frontright
        wall_frontright = wall_msg.data
        publish_walls()

def handle_zumo_right(wall_msg):
        global wall_right
        wall_right = wall_msg.data
        publish_walls()
```

```
if __name__ == '__main__':
        pcl_pub = rospy.Publisher("/zumo/objectcloud", PointCloud2, queue_size=10)
        rospy.init_node('objectdetect_node')
        listener = tf.TransformListener()
        rospy.sleep(1.)
        rospy.Subscriber('/zumo/prox_left', Int8, handle_zumo_left)
        rospy.Subscriber('/zumo/prox_frontleft', Int8, handle_zumo_frontleft)
        rospy.Subscriber('/zumo/prox_frontright', Int8, handle_zumo_frontright)
        rospy.Subscriber('/zumo/prox_right', Int8, handle_zumo_right)
        rate = rospy.Rate(10.0)
        while not rospy.is_shutdown():
                try:
                        (trans,rot) = listener.lookupTransform('/world', '/zumo', rospy.Time(0))
                except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
                        continue
                transx = trans[0]
                transy = trans[1]
                transz = trans[2]
                rotx = rot[0]
                roty = rot[1]
                rotz = rot[2]
                rotw = rot[3]
        rospy.spin()
```

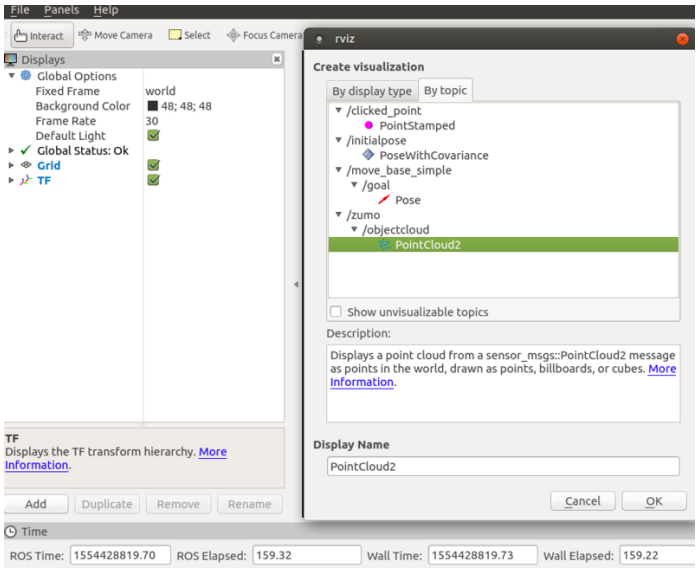
2.13. To test this task, you need to execute the following.

```
$ sudo apt-get install ros-kinetic-tf2-sensor-msgs
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
$ rosrun lab5 zumo_tf_broadcaster.py
$ rosrun lab5 zumo_controller.py
$ rosrun lab6 zumo_tf_sensor.py
```

2.14. You will also need to visualise what is going on, so run rviz as follows

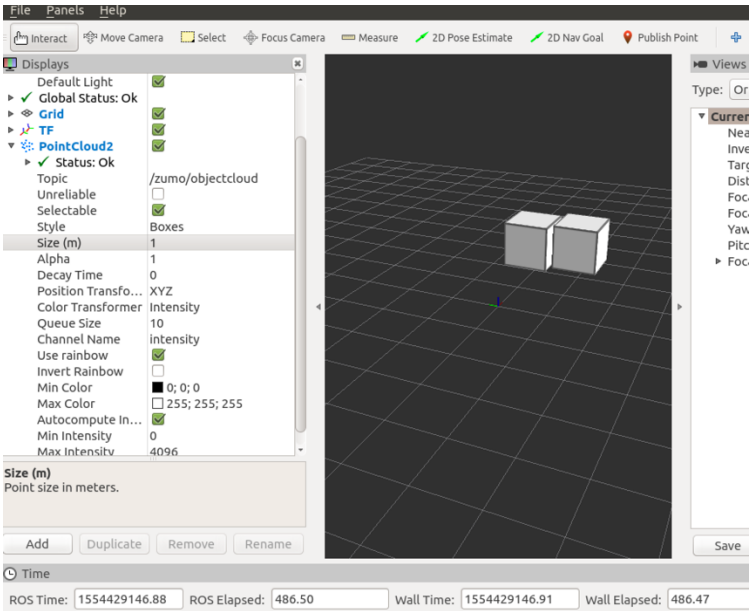
```
$ roscd lab5
$ rviz -d ./zumo.rviz
```

You should test that you are seeing the robot move around on the screen. To visualise the pointcloud data you should click on the + button on the bottom left, and click topic in the window that appears, and select the PointCloud topic – as illustrated in the following.



Then you should expand PointCloud2 in the left tree view. Change the style to Boxes and the size to 1. This will turn our points into larger objects.

If all goes well now, you should now have a live visualization of the robot and some objects. Put something in front of the robot to test it. You should end up with the following.

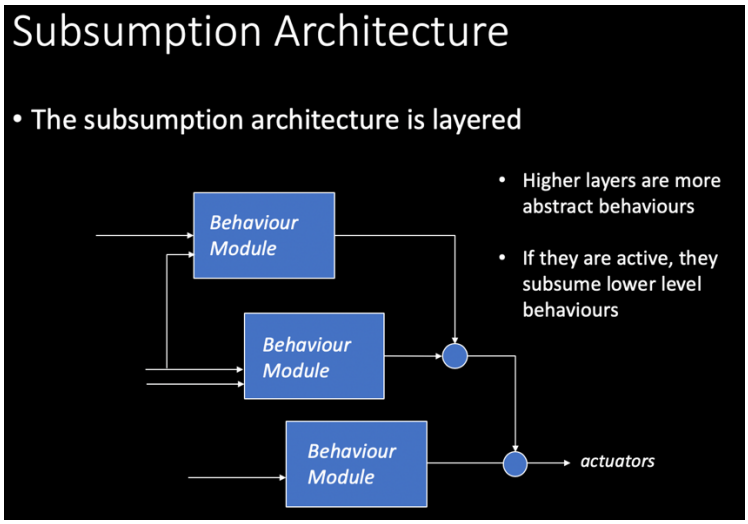


You should fix the code above in publish_walls() in zumo_tf_sensor to display objects to the left and right. Most of the work has been done for you.

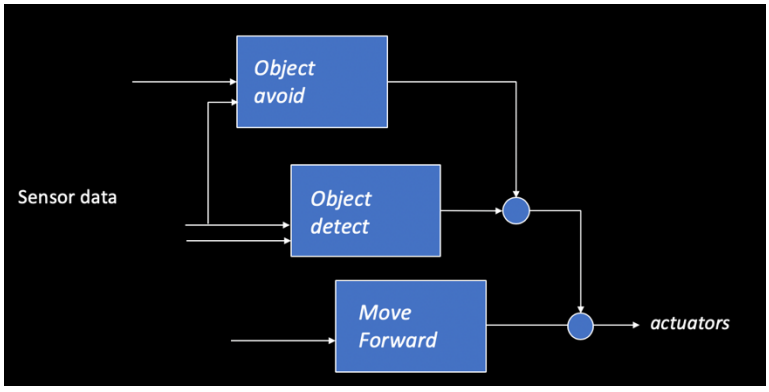
3. Building a subsumption architecture.

3.1. We are now going to build a more advanced version of the scared robot program using a subsumption architecture. This program doesn't run away from objects but attempts to get around them. If you are unsure what a subsumption architecture is, refer to lecture 4.

3.2. Recall from the lecture, the simple subsumption architecture as follows.



We are going to build a variation of this as follows.



This architecture has the following nodes:

- 'zumo_move_forward' node that will instruct the zumo to move forward forever.
- 'zumo_object_detect' node that will stop the zumo when there is an object.
- 'zumo_object_avoid' node that attempt to move around the object.

3.3. We will build the bottom two nodes to start with. You should create each of these in the scripts folder of lab6 and make them executable as follows. The zumo_move_forward just subscribes to the sensor topic to ensure the robot is alive and then just requests it constantly move forward. The zumo_object_detect node will publish to a new topic and switch /zumo/cmd_vel between these two topics.

zumo_move_forward.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int8
from geometry_msgs.msg import Twist
import time
import sys

rospy.init_node('zumo_move_forward', anonymous=True)
pub = rospy.Publisher('/zumo/1/cmd_vel', Twist, queue_size=10)

def move_forward(msg):
    vel_msg = Twist()
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0
    vel_msg.linear.x = 1
    pub.publish(vel_msg)
    time.sleep(0.1)

    #bascially only move if we have sensor data, indicating a live robot
rospy.Subscriber('/zumo/prox_frontleft', Int8, move_forward)
rospy.spin()
```

zumo_object_detect.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int8
from geometry_msgs.msg import Twist
from std_msgs.msg import Int8

import time
import sys
import os

rospy.init_node('zumo_object_detect', anonymous=True)
pub = rospy.Publisher('/zumo/2/cmd_vel', Twist, queue_size=10)
```

```
current_topic = 0
new_topic = 1

def stop():
    vel_msg = Twist()
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0
    vel_msg.linear.x = 0
    pub.publish(vel_msg)

def handle_zumo_sensor(wall_msg):
    global current_topic

    if(wall_msg.data > 6):
        new_topic = 2
        stop() #actually publish a new message..
    else:
        new_topic = 1

    #only change topics if we need to.
    if(current_topic ==1 and new_topic == 2):
        os.system("rosrun topic_tools mux_select mux_cmdvel /zumo/2/cmd_vel")
        current_topic = 2
    if(current_topic ==2 and new_topic == 1):
        os.system("rosrun topic_tools mux_select mux_cmdvel /zumo/1/cmd_vel")
        current_topic = 1

rospy.Subscriber('/zumo/prox_frontleft', Int8, handle_zumo_sensor)
rospy.Subscriber('/zumo/prox_frontright', Int8, handle_zumo_sensor)

#take over
os.system("rosrun topic_tools mux_select mux_cmdvel /zumo/1/cmd_vel")
current_topic = 1
rospy.spin()
```

3.4. To test this, you will need everything running, as follows:

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
$ rosrun lab5 zumo_tf_broadcaster.py
$ rosrun lab6 zumo_tf_sensor.py
$ roscd lab5
$ rviz -d ./zumo.rviz
```

You will also need to run the subsumption nodes as follows.

```
$ rosrun lab6 zumo_move_forward.py
$ rosrun topic_tools mux /zumo/cmd_vel /zumo/1/cmd_vel /zumo/2/cmd_vel mux:=mux_cmdvel
$ rosrun lab6 zumo_object_detect.py
```

You should see the robot move when the 2nd command above is issued as the /zumo/1/cmd_vel is copied to /zumo/cmd_vel. When the object_detect.py script is run, it will take control of the mux'ing, and depending on if anything is in front of the robot, will stop the messages from zumo_move_forward.py program on topic /zumo/1/cmd_vel being copied to /zumo/cmd_vel. It will instead forward its own stop messages to the robot from topic /zumo/2/cmd_vel.

The robot should now drive up to objects, once detected it should drive to the right to try to avoid them. This is a nice foundation for potentially feature-rich robot applications.

4. Assessment

This week's lab assessment is based on the output of this lab as follows. You should start this now. Depending on what you achieve, you should upload a short paragraph of explanation of what you have done, your code in a zip and a short video to the quiz "Lab Assessment 2" on the unit site. Upload your video using DeakinAir – a mobile phone video is fine. Put your zip on your onedrive, grab a link. And insert this link in the text box.

Leave the quiz questions for more advanced functionality blank. Don't worry, there is plenty of time to get great grades in the unit!

- P – Ensure the subsumption architecture is working as described. Get left and right sensors working in rviz, and calibrate all the wall offsets as best as possible (these are the 0.5 values in the Python code). Demonstrate this working on the video you upload.
- C – Implement the node zumo_object_avoid' to have it take over the operation of the robot from the other two nodes and navigate around the object.
- D – have the zumo_object_avoid node decide to try and avoid the front object by either moving towards the left or right depending on if there is an object to the direct left or right – increasing the chances of getting past the object. You should implement this is the same way that zump_object_avoid is implemented, using another mux tool.
- HD – Add an additional node which will back away from an object and try to move around if in a wider curve if 'zumo_object_avoid' fails to move around it.

5. Additional resources

5.1. Learn about linux

- https://www.computerworld.com/article/2598082/linux/linux-linux-command-line-cheat-sheet.html

5.2. Learn about ROS.

- http://wiki.ros.org
- A Gentle Introduction to ROS: https://www.cse.sc.edu/~jokane/agitr/
- http://wiki.ros.org/kinetic/Installation/Ubuntu
- http://wiki.ros.org/ROS/Tutorials
- https://www.pololu.com/docs/0J63/4
- http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup
- http://wiki.ros.org/rosserial_arduino/Tutorials/Hello%20World