CMSC 234/334 Mobile Computing Chien, Winter 2017

Assignment #3: Basic Control of Magoo F1/10 Car Platform

***Release**: Jan 17, 2017* ***Due**: January 23 and 24, 2017 (depends on signup)*
Checkoff Location: John Crerar Library - Kathleen A. Zar Room

Overview
The objective for this lab is to teach you how to setup your ROS (Robot Operating System) environment correctly, send commands from your laptop to control the car, and retrieve depth sensor data from the car. We will setup the space at John Crerar Library - Kathleen A. Zar Room for driving the car. Please sign up for a 15 mins checkoff time slot. http://bit.ly/2iNC6qI

This lab has two parts. The first part you can do on your own with your laptop. The second part requires a Magoo car. So, you will need to sign for a time slot to lab to connect to the Magoo system to debug your design. You must sign up for exactly one 1-hour debug time slot. http://bit.ly/2jdurk9

Please make sure you finish the Part 1 before you come to debug lab hours.

Goals for this lab:
- ROS environment setup
- Building ROS packages for remote control of car
- Use for basic motion moving forward, backward, and turning.
- Controlled maneuvers: Drive in a circle with 1m radius at a constant speed.

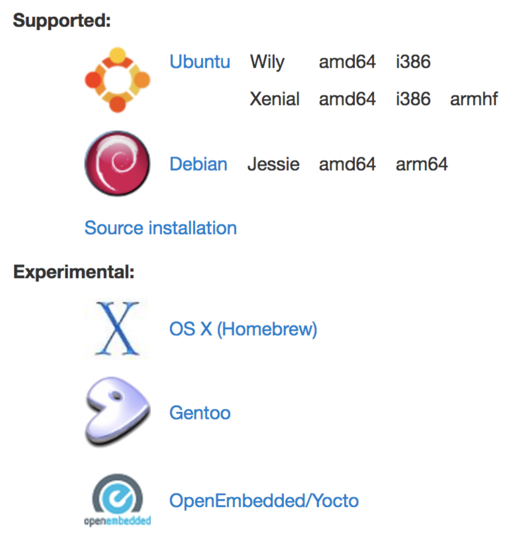
Part 1:
- Installing ROS Indigo on Ubuntu 14.04
- Learning the basic concept of ROS
- Creating a ROS package
- Setup your laptop wireless configuration to control the Magoo Car

Part 2:
- Run your ROS control package on client side to control the Magoo car platform
- Use program to control the car run a circle.
- Implement collision avoidance in your control package

Part 1: ROS Installation

Installing ROS Indigo on Ubuntu 14.04

As ROS officially supports Ubuntu and Debian operating system, we use Ubuntu as the development platform for this course. Highly recommend to use Ubuntu as the development platform. If you don't have a Ubuntu machine, you can install Ubuntu on a VM. If you choose other OS, we cannot provide support for that.



Supported:

Ubuntu	Wily	amd64	i386	
	Xenial	amd64	i386	armhf
Debian	Jessie	amd64	arm64	

Source installation

Experimental:

OS X (Homebrew)

Gentoo

OpenEmbedded/Yocto

Please refer to ROS installation website to see the detailed step-by-step commands.
http://wiki.ros.org/indigo/Installation/Ubuntu

Learning the basic concept of ROS

After ROS is installed on your development environment, we can now run some example programs on your machine to show how ROS works.

First, we need to initiate the ROS framework by the command:

```
$ roscore
```

Roscore is the master node to communicate with all ROS subsystem and provide interconnect among them. It has to be running in the background before you want to launch any of ros packages.

Open another terminal, we use *rosrun* to execute a node, turtlesim_node, from the *turtlesim* package

```
$ rosrun turtlesim turtlesim_node
```

Now, you can see a turtle in the middle of the ocean. To control this turtle, you need to run another node to capture the keyboard strokes and then send the data to the *turtlesim_node*. Open another terminal to run the node, *turtle_teleop_key*, to control the turtle.

```
$ rosrun turtlesim turtle_teleop_key
```

Staying on the *turtle_teleop_key* terminal, and you can use the arrow keys to control the turtle moving forward, backward, and turning.

To see the all ROS node in this system, we can use the following command

```
$ rosnode list
```

We can see there are 3 nodes. /rosout is running by default to capture logs, and the other two are we just started. The *turtlesim_node* and the *turtle_teleop_key* node are communicating with each other over a ROS Topic. *turtle_teleop_key* is publishing the key strokes on a topic, while *turtlesim_node* subscribes to the same topic to receive the key strokes. To see the all ROS topic published in this system, we run the command

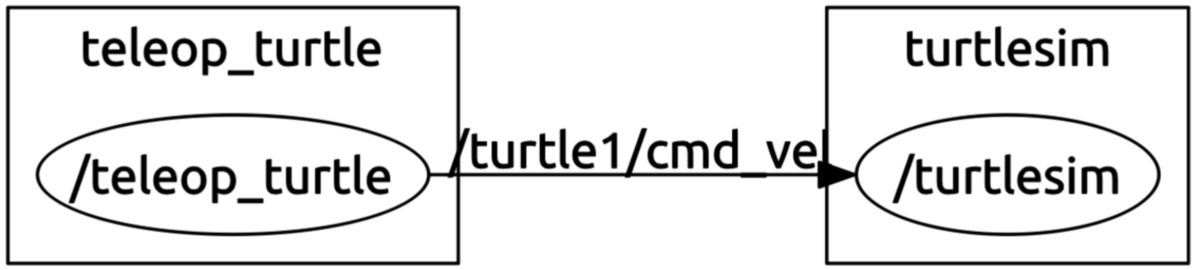
```
$ rostopic list
```

We can see the raw data that is published on the topic by running

```
$ rostopic echo /turtle1/cmd_vel
```

Let's use rqt_graph which shows the nodes and topics currently running.

```
$ rosrun rqt_graph rqt_graph
```



Creating a ROS package

Catkin is the tool we use to initialize the workspace and create ROS package. You can see more detail in the ROS website: http://wiki.ros.org/ROS/Tutorials/CreatingPackage

First, let's create a catkin workspace:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Catkin is a ROS tool for compiling the different subsystems and modules used in ROS. Perform a catkin_make from the root of the workspace.

```
$ cd ~/catkin_ws/
$ catkin_make
```

If you look in your current directory you should now have a 'build' and 'devel' folder. Inside the 'devel' folder you can see that there are now several setup.*sh files. Sourcing any of these files will overlay this workspace on top of your environment. To add the workspace to your ROS environment you need to source the generated setup file.

```
$ source devel/setup.bash
```

Now we can create a ROS package in src folder.

```
$ cd src
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

The beginner_tutorials is the name of the package, and the std_msgs, rospy, and roscpp are the dependencies we are defining. If the environment is sourced, we can use roscd to navigate to the package folder.

```
$ roscd beginner_tutorials
```

Put talker.py and listener.py into src folder under beginner_tutorials package.

Talker.py:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Listerner.py:

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

You can get more detail about talker.py and listener.py from this ROS tutorial webpage:
http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

Before running the node, remember to set them to executable file

```
$ chmod +x talker.py listener.py
```

Bulid the nodes:

```
$ cd ~/catkin_ws
$ catkin_make
```

Start roscore

```
$ roscore
```

In a new terminal, run listener node:

```
$ source ~/catkin_ws/devel/setup.bash
$ rosrun beginner_tutorials listener.py
```

In another terminal, run talker node:

```
$ source ~/catkin_ws/devel/setup.bash
$ rosrun beginner_tutorials talker.py
```

Now you are successful to create your own ROS package and running the nodes from the package.

Wireless network setup:

Follow the instruction below to setup your laptop wireless connection before you come to the lab debugging and checkoff.

Setup the wireless network to connect the car system

To connect to the car system, please add a new wireless connection and connect to SSID: *f1tenthcar* or *f1tenthcar2*.

Go to IPv4 setting, manually set IP address to 192.168.1.X, Netmask: 255.255.255.0, and Gateway: 192.168.1.X. (X is not 1 or 20)

Here, we use IP address 192.168.1.234 as an example.
Edit ~/.bashrc and add the lines:

```
Export ROS_MASTER_URI=http://192.168.1.1:11311
Export ROS_IP=192.168.1.234
```

Run command:

```
$ source ~/.bashrc
```

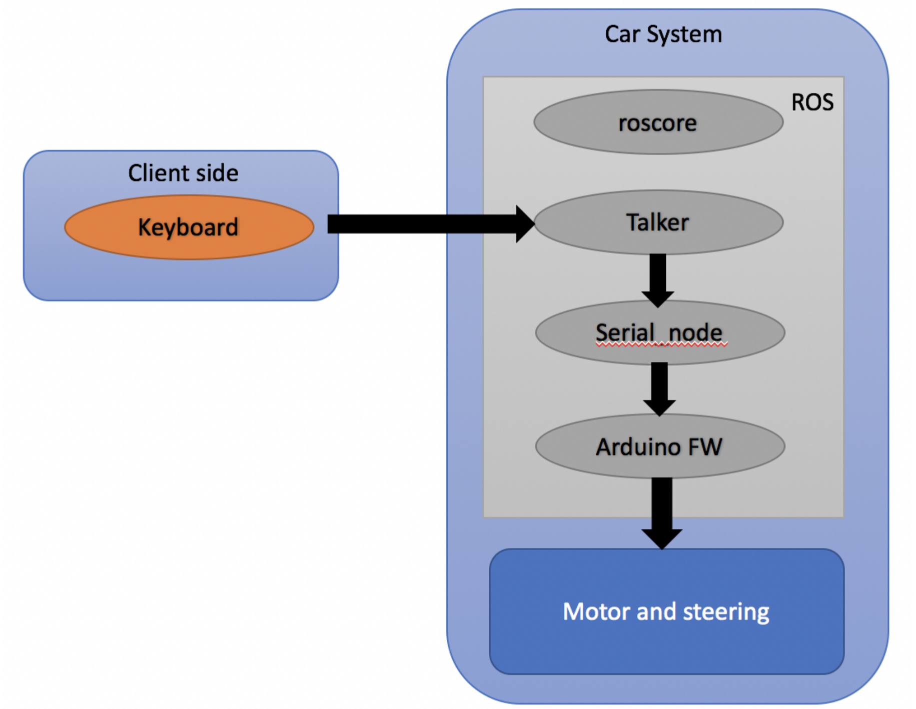
Since the roscore node will be running on the car system which has the IP address 192.168.1.1, the ROS_MASTER_URI indicates the master node IP, and ROS_IP is your laptop IP to communicate with the ROS framework.

Part 2: Magoo Maneuvers

Please make sure you finish the Part 1 before you start working on part 2 and come to debug lab hours.

Implementation:

Here's the diagram showing all the nodes related to the car control system.



Those nodes on the car system are ready to go. In this lab, all you need to do is to create your own package containing keyboard control node and collision avoidance node, and execute them on your laptop to publish and subscribe the corresponding topics to trigger the car moving and complete a collision avoidance feature for your self-driving prototype. To prevent the car crash, we have implemented a collision avoidance with distance 1m, and also set the car moving step by step.

Please download the sample code: http://bit.ly/2iqmRoU
In the sample code, we have two folders, car_system and client.

car_system: To help you fully understand how the control signals actually work, the source code of talker.py and arduino fw are released in the car_system folder.

client: The templates of keyboard.py and avoidance.py can be found in the client folder. You need to modify the keyboard.py and avoidance.py, and build your own ROS package containing the both nodes.

To see the image data from realsense camera, *RViz* is a common tool to display visualization. On your laptop, use the following commands to check if the realsense camera topics are available

```
$ rostopic list
```

Use the following command to open RViz.

```
$ rviz
```

In RViz, choose Add -> By topic -> select /camera/rgb/image_color, and then you should be able to see a image with RGB data. Selecting topic /camera/depth/image_raw to see the depth data.

You can refer to http://wiki.ros.org/rviz/UserGuide to get more information.

Requirement:

1. Execute keyboard.py to control the car:
   Arrow up: speed up / moving forward
   Arrow down: slow down / moving backward
   Arrow left: steering turn left
   Arrow right: steering turn right
   Key 's' : stops the car and center the steering
   Key 'c' : drives the car in a circle with 1m radius

2. Execute avoidance.py The car will move forward at a slow constant speed, and stop when it approaches an object within 2 meters. However, remember that the sensor has range 0.5-5m, so it farsighted and can't "see" objects closer than about 0.75m. Once the obstacle is removed, the car should keep moving forward.