Lab Session 3 – Simulated Turtlebot

This lab session will focus on learning the basics of taking in data from the camera, processing it, and then making decisions based on it. Just like we did in the last lab session, we will run our code first in the simulator and thereafter on the real turtlebot.

Set up the singularity environment

Using steps described in lab 1, create a singularity container and then a few Ubuntu terminals to use for later.

Making sure your git clone is up-to-date

Before you do anything else, make sure your git clone of lab3 is up-to-date:

```
cd $HOME/catkin_ws/src/lab3

git pull
```

Starting Turtlebot Simulator with a different World file

In this lab, we will load the simulated Turtlebot into a different environment. You will find the rgb.world file under $HOME/catkin_ws/src/lab3/src/rgb.world. Now in an Ubuntu terminal, execute:

```
export TURTLEBOT_GAZEBO_WORLD_FILE=$HOME/catkin_ws/src/lab3/src/rgb.world

roslaunch turtlebot_gazebo turtlebot_world.launch
```

The first command above tells the simulator to use the new rgb.world file. The second command starts the simulator, as in the previous lab session. In the simulator environment, you should now see three spheres of red, green and blue colour. You can grab and move them around.

Now let's focus on getting your python node capable of reading the camera data and then getting it to display to a screen. Open the file Skeleton_Code_First_Step.py.

OpenCV

To process images and make decisions based on them we will be using a library called OpenCV (Computer Vision). Specifically we will be using OpenCV2 so in any python scripts that you want to handle images you need to import cv2:

```
import cv2
```

Since we are handling ROS we also need the use of a library called cv_bridge, which can translate ROS images into images readable by OpenCV. (More info abut cv_bridge is here: http://wiki.ros.org/cv_bridge/Tutorials)

Importing necessary modules

We always start by importing the necessary python modules and classes.

```
import cv2

import numpy as np
```

```
import rospy

from sensor_msgs.msg import Image

from cv_bridge import CvBridge, CvBridgeError
```

Subscribing to the image topic

In order to receive and process image data from the cameras we must create a subscriber to the topic that our camera outputs to. The RGB image from the 3D sensor is on the topic: camera/rgb/image_raw. Create a subscriber for this topic in the constructor (__init__) of the colourIdentifier class, and specify the callback function of the colourIdentifier class as the callback of the image topic.

Converting between openCV and ROS image formats

The image from the camera arrives in the ROS type Image. To manipulate it in OpenCV, we must convert the image from the ROS format of Image into an OpenCV image. Luckily OpenCV has buit in functions to do this for us. Call the function imgmsg_to_cv2(data, "bgr8") on a CvBridge object you have created in your class. It's always a wise idea to wrap calls to this method in an exception handler in case anything is wrong with the camera feed.

The only thing left to do is to output the camera feed to the screen.

Displaying image on a new window

We declare that we want to have a named window called 'camera feed' then we show it.

```
cv2.namedWindow('Camera_Feed')

cv2.imshow('Camera_Feed', <image name>)
```

Try to run your script and see if it works.

Manipulating the camera feed to produce new images

Next we're going to look at manipulating our image to isolate all parts of a certain colour. Isolating and detecting colours from each other is a very useful tool for robot computer vision.

You will have noticed that when we converted the ROS image into an OpenCV image that "bgr8" was given as an argument. This is the original colourspace of the camera feed and thus what colourspace the converted image should be. This means our OpenCV image is in the bgr8 colourspace. However we will be working off of HSV images. OpenCV provides methods for converting one colourspace to another. To convert colours we make a simple method call providing the source image and which colour conversion algortihm should be used, this will have been mentioned in Lectures. This produces a HSV image stored in a variable you assign it to.

Once we have converted the image, we need to decide what we want to filter out. In the case of the code provided we want to filter out anything that isn't green. So we define upper and lower bounds for the value of green in the image:

```
hsv_green_lower = np.array([60 - sensitivity, 100, 100])

hsv_green_upper = np.array([60 + sensitivity, 255, 255])
```

Then we want to create a mask image filtering out anything that isn't within those two colours. The inRange method will examine each pixel and turn anything outside of the range to black and anything inside the range to white, refer to the lecture notes for use of this method.

This will produce a black and white image stored in a variable. However just producing a black and white image isn't good information for someone outside of the code. They won't know that you're isolating green. So let's try and produce an output that a human would find useful.

The only way to add images to each other, requires them to be in rgb colourspace. However we have a way around this even in hsv. We simply bitwise_and the original image with itself and also supply a mask to be overlayed onto the image, this method will also be in the lecture notes.

Now when you use imshow to show the image created and stored in the variable you assigned it to you will see green objects are still visible while other objects are blacked out.

Exercises

1. Try completing the skeleton code to filter out all colours apart from one in an image to start with. (Skeleton_Code_First_Step.py)

2. Try adding a second and third colour to ones you want to isolate from the initial image and produce an image similar to the output in the previous example. Tip:(You can combine masks via the bitwise_or method) (Skeleton_Code_Second_Step.py)

3. As discussed in lectures it is not enough to simply isolate colours from the original image, we need to be able to detect that an object is there, judge it's shape, distance and area, in order to make informed decisions concerning it. Complete the skeleton code to send messages based on what colours are present in the image view.. Follow the comments and using methods that were talked about in lectures attempt to create the node. (Skeleton_Code_Third_Step.py)

4. Now that you can send messages based on a colour being present, try creating one final node that can instruct the robot to follow a particular colour, stopping when it catches sight of another colour. (Skeleton_Code_Final.py)

Lab Session 3 – Real Turtlebot

Having completed the simulation phase, we will now perform the same exercise on the real Turtlebot.

Initializing your group's git repo

If you do not remember how to do this, please read the section "Initializing your group's git repo" from the lab1 worksheet, and initialize your group's git repo for lab3.

Setup Turtlebot laptop for your group

If you do not remember how to do this, please read the section "Setup Turtlebot laptop for your group" from the lab1 worksheet.

Run the following launch files to bring up the required nodes and topics for this lab session:

- roslaunch turtlebot_bringup minimal.launch
- roslaunch astra_launch astra.launch

Now try running your script from the simulation lab, this time on the real Turtlebot, to repeat the Exercises. Remember that the image data from a real camera can be much noisier than the image data from a simulated camera.

Do not forget to push any change in your code to git! Otherwise you will lose them when you logout of the Turtlebot laptop!