School of Electrical, Computer and Telecommunication Engineering
ECTE477 - Programming Autonomous Systems

Laboratory Notes

Lab 10: Line Following

1 Aim

This tenth laboratory aims to teach the student about line following. It will cover the following areas:

- Combining subscriber messages with message filters.
- Image processing to detect where the line is in the image.
- Steering the robot based on the line's position.

These notes have been adapted from the ROS Wiki Tutorials and the Robotis Turtlebot3 e-manual which can be found at the following URLs:

- `http://wiki.ros.org/ROS/Tutorials`
- `http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/`

2 Background

Today we will begin on the third demonstration which is designed around autonomous driving. The end goal will be to perform lane following in addition to other tasks, however, today we are starting with *line* following. You will be keeping track of the position of a yellow line and using that to steer the robot. You will first be doing this in a Gazebo world and then you will test the same code on the Turtlebots.

3 Procedure

3.1 Setting up the line following node

To begin with, make sure you have the `line_world` package on your computer in the `catkin_ws/src` directory. If you do not have it, you should download it from `https://github.com/UOW-SECTE-477/line_world.git`.

In *your package*, create the following script called `line_follower.py` and make sure it is executable with the `chmod +x` command:

```python
#!/usr/bin/env python

class line_follower:
    def __init__(self, name):
        self.name = name

if __name__ == '__main__':
    rospy.init_node('line_follower', anonymous=True)
    rospy.loginfo("[line_follower] Starting Line Follower Module")
    lf = line_follower("line_follower")
    try:
        rospy.spin()
    except KeyboardInterrupt:
        rospy.loginfo("[line_follower] Shutting Down Line Follower Module")
```

You should also create a launch file `line_follower.launch`:

```xml
<launch>
    <node pkg="<your_package>" type="line_follower.py" name="line_follower" output="screen" />
</launch>
```

At the top of your script include the following libraries:

```python
import rospy, message_filters, cv2
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist
from cv_bridge import CvBridge, CvBridgeError
```

3.2 Synchronising messages with message filters

You will be performing image processing requiring both colour and depth images from the Realsense camera. For the last demonstration, you created two separate callbacks and potentially spent different amounts of time in each meaning you couldn't be sure if the current depth image perfectly corresponded to the given colour image. At slow speeds this wasn't a problem, however, at faster speeds the delays between the two images could have caused blatantly incorrect depth estimations.

Today, we will be using `message_filters` to synchronise messages in a single callback based on timestamps. To create a combined subscriber use the following:

```python
colour_image_topic = "/camera/rgb/image_raw"
depth_image_topic = "/camera/depth/image_raw"
colour_sub = message_filters.Subscriber(colour_image_topic, Image)
depth_sub = message_filters.Subscriber(depth_image_topic, Image)
self.camera_sync = message_filters.TimeSynchronizer([colour_sub, depth_sub], 10)
self.camera_sync.registerCallback(self.callback_camera)
```

In the above code, we are separating out the topic names into variables. The reason for this is that later on you will be running this code on the Turtlebots where the two topic names will differ slightly, so we want to place them in an easy location.

The next two lines involve creating the individual subscribers. Consider the following:

```python
rospy.Subscriber(depth_image_topic, Image, self.callback_depth)
message_filters.Subscriber(depth_image_topic, Image)
```

In the former, we are using `rospy` and are specifying the callback. In the latter, we are using `message_filters` and do not specify the callback. This is because we will be specifying a joint callback.

On the next line, we create a `TimeSynchronizer` which combines the subscribers. The first argument takes a list of subscribers to be included and the second argument is a queue size. The messages of each subscriber are synchronised based on their timestamps and because of delay between the arrival of messages a queue is used to hold them while waiting for their corresponding ones from other topics. If there is too much delay then the callback will never be triggered.

If we are worried about a large delay we can either increase the size of the queue, or we can use an alternative synchroniser that allows for slight differences in timestamps. This is called the `ApproximateTimeSynchronizer` and would be used as follows:

```
self.camera_sync = message_filters.ApproximateTimeSynchronizer([colour_sub, depth_sub], queue_size=10,
    slop=0.1)
```

This takes a queue size like the previous, but also has a parameter `slop` which is the maximum allowable discrepancy between timestamps.

The final line registers the callback to the synchroniser. You should create this callback now:

```
def __init__(self, name):
    ...
    self.bridge = CvBridge()
...
def callback_camera(self, colour_msg, depth_msg):
    rospy.loginfo("[%s] callback_camera()", self.name)
    colour_image = self.bridge.imgmsg_to_cv2(colour_msg,desired_encoding='bgr8')
    depth_image = self.bridge.imgmsg_to_cv2(depth_msg, desired_encoding="passthrough")
```

This is very similar to previous callbacks, however, now we have extra arguments for each topic. These arguments should appear in the same order that they were passed into the `ApproximateTimeSynchronizer`.

To ensure that this is working correctly, you can add the following to your callback:

```
self.image = colour_image
```

and the following to the end of your `__init__` function:

```
self.image = None
while not rospy.is_shutdown():
    if self.image != None:
        cv2.imshow("window", self.image)
        cv2.waitKey(30)
```

You can test the node using:

```
roscore
roslaunch line_world line_world_1.launch
roslaunch <your_package> line_follower.launch
```

You should be able to see the robots view of the world shown in Figure 1.

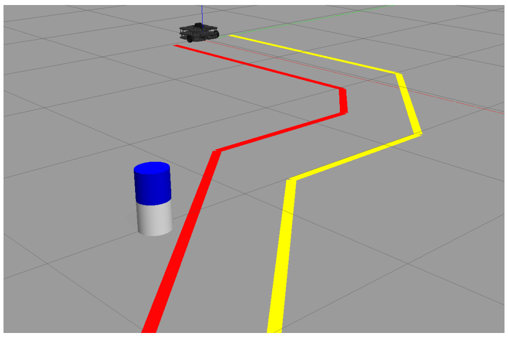

Figure 1: The *Line World* Gazebo world.

3.3 Finding the line

In the callback, we want to find the target object based on its HSV values. This is done using the range-based masking with erosion and dilation:

```
hsv = cv2.cvtColor(colour_image, cv2.COLOR_BGR2HSV)
lower_yellow = (25, 200, 100)
upper_yellow = (35, 255, 255)
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
mask = cv2.erode(mask, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)
```

We are separating out the colour tuples because when we move from Gazebo to the Turtlebot these values will change. To see the progress of this masking comment out the previous `self.image` assignment and add the following after the masking operation:

```
self.image = mask
```

Now when you run the node, you should see a black image with a white line remaining where the yellow line was, as in Figure 2a.

The part of the yellow line farther away from the robot is not particularly useful for line following. We want to move towards the line by aiming at the nearest part. A simple heuristic to only focus on the near part is to black out the upper region of the image. In the following code, we set the top three-quarters of the image to black, then we keep a 20 pixel band of the image. We are assuming that the part of the line that we want to drive towards will fall within this 20 pixel band.

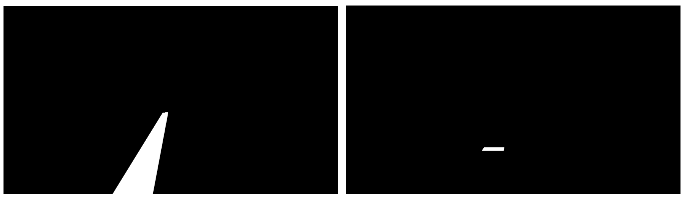
(a) The full detected line. (b) The trimmed line segment.

Figure 2: The full line is shown on the left, with the trimmed region shown on the right.

```
h, w, d = colour_image.shape
search_top = 3*h/4
search_bot = 3*h/4 + 20
mask[0:search_top, 0:w] = 0
mask[search_bot:h, 0:w] = 0
```

To interpret this code, remember that in image coordinates the top-left of the image is $(0, 0)$, therefore `0:3*h/4` is the top three-quarters of the image and `0:w` is the entire width. This code should now find a small blob relating to the near part of the line, as shown in Figure 2b. You can test it out by moving the `self.image = mask` line to after the above code.

3.4 Steering the robot

For the beacon detection, we found the centroid by placing a bounding box around the largest contour. This week, we will do this using the `cv2.moments()` function. This takes the weighted average of the white pixels in the masked image to find the centroid of irregular shapes. The code to find the centroid of our line blob is:

```
M = cv2.moments(mask)
if M['m00'] > 0:
    cx = int(M['m10']/M['m00'])
    cy = int(M['m01']/M['m00'])
```

We can plot this centroid on the colour image by adding the following inside the if-statement:

```
cv2.circle(colour_image, (cx, cy), 20, (0,0,255), -1)
self.image = colour_image
```

The only value that we care about is `cx`. This tells us where the line is horizontally. We can use this in a simple proportional-controller that rotates the robot with the goal of keeping the centroid in the centre of the image. Remember that the robot can be controlled using `Twist` messages sent to `/cmd_vel` with `linear.x` controlling forward movement and `angular.z` controlling rotation. You can implement proportional-control by adding the following to the if-statement:

```
err = cx - w/2
twist = Twist()
twist.linear.x = 0.2    # Constant forward movement
twist.angular.z = -float(err) / 1000
```

Finally, to publish this you want to create a publisher in the `__init__` function and publish the `Twist` inside the if-statement:

```
self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
...
self.cmd_vel_pub.publish(twist)
```

You should now be able to test the complete line following with:

```
roscore
roslaunch line_world line_world_1.launch
roslaunch <your_package> line_follower.launch
```

3.5 Running on the Turtlebots

You can also run your line following code on the physical Turtlebots. *Ask the tutor to make sure one is currently available.*

To connect to the Turtlebot, you will be copying your code to the Robot Server and controlling it from there. When connecting to the Robot Server, the username is `student` and the password is also `student`. You can connect using `ssh robotserver`, then once you are connected you can connect to the Turtlebot using `ssh tb01`. You cannot connect directly to the robot from your virtual machine, so you will have to connect to the server first. In this section, when referring to a command run on your virtual machine it will be annotated with **(VM)**, the Robot Server will be annotated with **(RS)**, and the Turtlebot will be annotated with **(TB)**. You should not actually type those annotations into the terminal.

Before you move your code to the server, you will need to change a few values. The image and depth topics are different, and you will also need to slightly widen the HSV range. Edit the corresponding lines of code:

```
colour_image_topic = "/camera/color/image_raw"
depth_image_topic = "/camera/aligned_depth_to_color/image_raw"
...
lower_yellow = (18, 100, 100)
upper_yellow = (35, 255, 255)
```

Now, you can copy your code to the server using:

```
(VM) scp -r /home/secte/catkin_ws/src/<your_package>/ robotserver:~/catkin_ws/src/<your_package>
```

Finally, you can run your program on the Turtlebot using the following in *three* separate terminal tabs:

```
# Terminal 1
(VM) ssh robotserver
(RS) roscore
# Terminal 2
(VM) ssh robotserver
(RS) ssh tb01
(TB) roslaunch turtlebot3_bringup turtlebot3_robot.launch
# Terminal 3
(VM) ssh robotserver
(RS) export DISPLAY=:0
(RS) roslaunch <your_package> line_follower.launch
```

You should now see the robot follow the yellow line. Your image output window will appear on the Robot Server screen.

4 Summary

In today's lab you have explored combining messages into synchronised callbacks using `message_filters`, and you have implemented a simple line follower that you tested on the Turtlebot. This will form the basis of the Demonstration 3 task, where you will be performing lane following.