PART III

Perception and Behavior

CHAPTER 12

Follow-bot

The previous several chapters were mostly concerned with getting robots to move around: either moving the robot base for locomotion or moving a robot arm for manipulation. Most of the systems we've built thus far would be considered *open-loop* systems, meaning that they have no feedback loop. That is, these systems do not use sensor data to correct for errors that accumulate over time. In this chapter, we will start working with sensors to create *closed-loop* systems that compute errors and feed them back into the control system, with the goal of reducing errors of various sorts.

Let's start by creating a robot that can follow lines on the ground using a camera. We will do this using OpenCV, a popular open source computer vision library. To build this system, we will need to do the following steps:

- Acquire images from a camera and pass them to OpenCV.
- Filter the images to identify the center of the line we are to follow.
- Steer the robot so that the center of the robot stays on the center of the line.

This will be a closed-loop system: the robot will sense the steering error as it drifts off the line and then steer back toward the center of the line. As we have always been doing in this book, we will develop this entire application in simulation. First, we will show how to subscribe to images in ROS.

Acquiring Images

Images in ROS are sent around the system using the `sensor_msgs/Image` message type. To have images stream into our nodes, we need to subscribe to a topic where they are being published. Each robot will have its own method for doing this, and names may vary. We will explore how to find the topic names using a Turtlebot

simulation. To get started, start three terminals: one for `roscore`, one for the TurtleBot simulation in Gazebo, and one for interactive commands.

Start `roscore` in the first terminal:

```
user@hostname$ roscore
```

In the second terminal, start a Turtlebot simulation:

```
user@hostname$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

Now, in the third terminal, we'll run some interactive shell commands. If this is our first time using this particular robot, we may not know what topics will contain the robot's camera data. So, let's sniff around a bit:

```
user@hostname$ rostopic list
```

This prints out a few dozen topics, some of which sound image-related:

```
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/camera/rgb/image_raw/compressedDepth
/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/camera/rgb/image_raw/compressedDepth/parameter_updates
/camera/rgb/image_raw/theora
/camera/rgb/image_raw/theora/parameter_descriptions
/camera/rgb/image_raw/theora/parameter_updates
```

This is a standard ROS interface for a modern depth camera like the Microsoft Kinect or Asus Xtion Pro. The first three topics start with `camera/depth` and, indeed, they deal with the calibration data and depth-sensor data. We'll get to the depth data later in this chapter, but first, let's deal with the visual images. The visual images streaming from the Turtlebot's camera appear on the `camera/rgb/image_raw` topic. The controller we will write is intended to run directly on the Turtlebot, so we should subscribe directly to the `image_raw` topic. If we were operating over a bandwidth-limited connection, such as a WiFi link, we might want to subscribe to the `image_raw/compressed` topic, which will run each frame through an image-compression library before sending it over the wire. The `theora` topic applies even more compression by creating a compressed video stream, rather than compressing the images one at a time. In typical camera streams, this results in considerable network bandwidth savings, at the expense of compression artifacts, potentially increased processor usage, and latency. In general, compressed video streams make sense when the goal is to

support human teleoperators; however, whenever possible, uncompressed images work best for computer vision algorithms.

Now that we know that the image data is available on the `camera/rgb/image_raw` topic, we can write a minimal `rospy` node that will subscribe to this data, as shown in Example 12-1.

Example 12-1. follower.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image

def image_callback(msg):
  pass

rospy.init_node('follower')
image_sub = rospy.Subscriber('camera/rgb/image_raw', Image, image_callback)
rospy.spin()
```

This program is the minimal code required to subscribe to image messages. But it doesn't really do anything. The image callback doesn't do anything at all:

```
def image_callback(msg):
  pass
```

on the `camera/rgb/image_raw` topic—however—the program does at least subscribe to messages. To verify this, first let's make *follower.py* an executable:

```
user@hostname$ chmod +x follower.py
```

And run it:

```
user@hostname$ ./follower.py
```

Many of the examples in the book change the permissions of a Python source file and then run it as an executable on the command line. This is simply a matter of personal preference. It is equally valid to explicitly invoke the Python interpreter and pass the Python script as an argument:

```
user@hostname$ python follower.py
```

The program will not produce any output. So, how can we know if it really subscribed to the image stream? Let's leave `follower.py` running, start another terminal, and interrogate the system:

```
user@hostname$ rosnode list
```

This will print a list of all currently running nodes. All but one of them are started by the Turtlebot simulation launch file:

```
/bumper2pointcloud
/cmd_vel_mux
/depthimage_to_laserscan
/follower
/gazebo
/laserscan_nodelet_manager
/mobile_base_nodelet_manager
/robot_state_publisher
/rosout
```

We can see that our `follower` node is indeed on the list of running nodes. Now, we can ask `roscore` to give us some details about its connections by typing the following:

```
user@hostname$ rosnode info follower
```

This prints lots of interesting output:

```
Node [/follower]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /camera/rgb/image_raw [sensor_msgs/Image]
 * /clock [rosgraph_msgs/Clock]

Services:
 * /follower/set_logger_level
 * /follower/get_loggers


contacting node http://qbox-home:59300/ ...
Pid: 5896
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /clock
    * to: /gazebo (http://qbox-home:37981/)
    * direction: inbound
    * transport: TCPROS
 * topic: /camera/rgb/image_raw
    * to: /gazebo (http://qbox-home:37981/)
    * direction: inbound
    * transport: TCPROS
```

The first block of that output lists the publications, subscriptions, and services that the node instantiated. Most were autogenerated by `rospy`, but we can see the `camera/rgb/image_raw` subscription that was part of the minimal program of

Example 12-1. The second section is often more interesting. To produce that section, the `rosnode` command-line program contacted the `follower.py` node and received a list of its current connections. The last element in that list shows that the `/camera/rgb/image_raw` subscription is indeed receiving inbound messages from the `/gazebo` node. Often, it is useful to understand how quickly messages are arriving. Fortunately, a simple shell command can estimate this for us:

```
user@hostname$ rostopic hz /camera/rgb/image_raw
```

The `rostopic hz` command will run forever; press Ctrl-C to make it stop. A few seconds of that command will print the output similar to the following:

```
subscribed to [/camera/rgb/image_raw]
average rate: 19.780
    min: 0.040s max: 0.060s std dev: 0.00524s window: 19
average rate: 19.895
    min: 0.040s max: 0.060s std dev: 0.00428s window: 39
average rate: 20.000
    min: 0.040s max: 0.060s std dev: 0.00487s window: 60
average rate: 20.000
    min: 0.040s max: 0.060s std dev: 0.00531s window: 79
average rate: 19.959
    min: 0.040s max: 0.060s std dev: 0.00544s window: 99
average rate: 20.000
    min: 0.040s max: 0.060s std dev: 0.00557s window: 104
```

From this output, we can gather that the `camera/rgb/image_raw` messages are arriving at 20 frames per second. Good!

Now that we know that the program in Example 12-1 is indeed receiving images, we need to do something with them! There are many different ways to proceed, but one of the most popular is to pass the images to the OpenCV library. OpenCV contains efficient, well-tested implementations of many popular computer vision algorithms. To pass data between the ROS and OpenCV image formats, we can use the `cv_bridge` package, which contains functions to convert between ROS `sensor_msgs/Image` messages and the objects used by OpenCV.

Example 12-2 instantiates a `CvBridge` object and uses it to convert the incoming `sensor_msgs/Image` stream to OpenCV messages and display them on the screen using the OpenCV `imshow()` function.

Example 12-2. follower_opencv.py

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import Image
import cv2, cv_bridge

class Follower:
```

```
  def __init__(self):
    self.bridge = cv_bridge.CvBridge()
    cv2.namedWindow("window", 1)
    self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                      Image, self.image_callback)
  def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg,desired_encoding='bgr8')
    cv2.imshow("window", image)
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()
```

As an example, the Turtlebot was moved and rotated within the default simulation world so that it was oriented facing a dumpster, as shown in Figure 12-1.

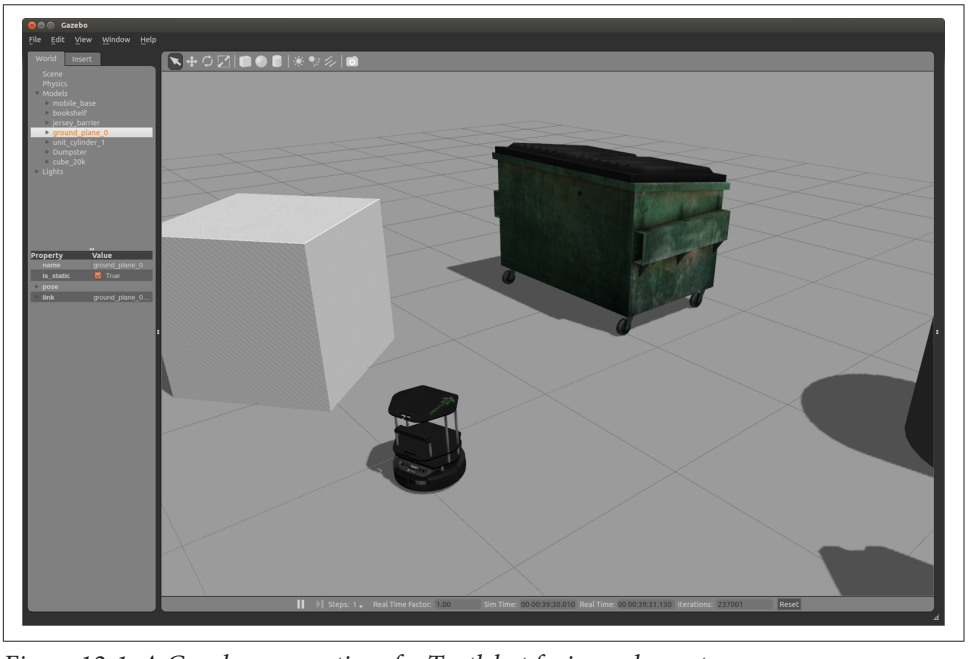

Figure 12-1. A Gazebo perspective of a Turtlebot facing a dumpster

Meanwhile, Gazebo is dutifully generating simulated camera images and streaming them to our program, which is using the OpenCV `imshow()` and `waitKey()` functions to render them to a GUI window (see Figure 12-2).

Figure 12-2. A dumpster, from the TurtleBot's perspective

That's it! We are now streaming simulated camera images through Gazebo, ROS, and OpenCV!

Although it's fun to look at dumpsters, let's look at something else. Let's load a Gazebo world with a nice bright line in it:

```
user@hostname$ roslaunch followbot course.launch
```

That Gazebo world file will start a Turtlebot on a yellow line that we want to follow, as shown in Figure 12-3. Why would we want to follow a line? Because lines are often used to mark routes, whether inside a controlled environment like a warehouse or a factory, or on roadways. Although each country has a particular scheme of colors and stripe patterns, broadly speaking, being able to detect and follow lines is one of the (many) skills required for autonomous driving.

In the next section, we will manipulate the images coming from the Turtlebot's camera to detect the center of the line in the camera frames.

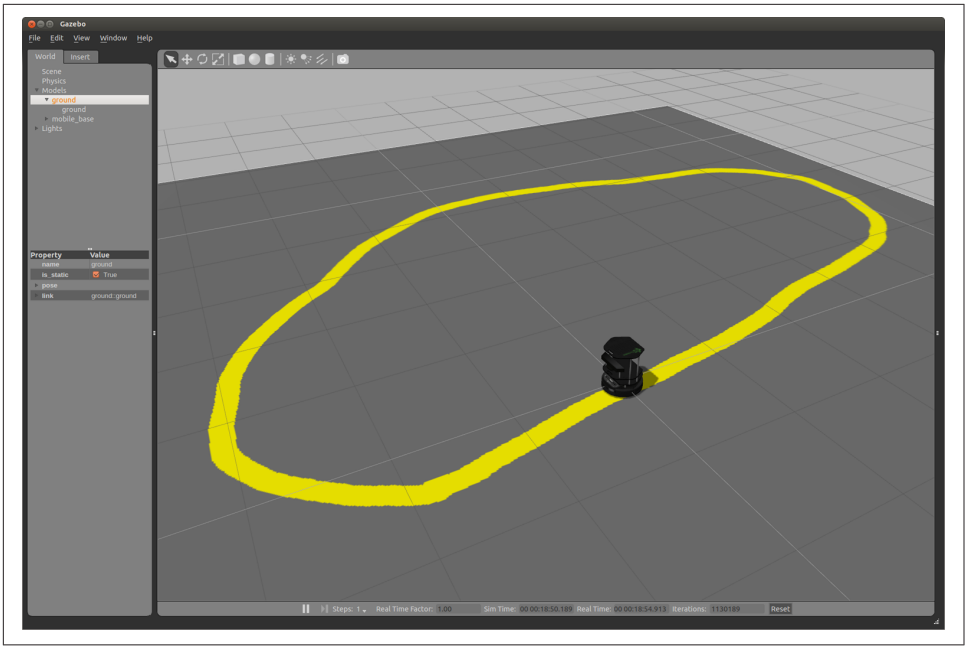

Figure 12-3. A Gazebo screenshot showing a Turtlebot on the course we want to follow

Detecting the Line

In this section, we will use OpenCV in Python to process the images coming through ROS from a simulated Turtlebot in the world shown in Figure 12-3. The goal is to detect the location of the target line in the Turtlebot's camera and follow it around the course. A typical image from the Turtlebot's camera is shown in Figure 12-4.

There are many strategies that can be used to detect and follow lines in various situations. Many PhD dissertations have been granted for this topic, which becomes arbitrarily complex when considering the variability and noise associated with, for example, roadway striping. Fortunately, in our case we are just going to consider an optimally painted, optimally illuminated bright yellow line. Our strategy will be to filter a block of rows of the image by color and drive the robot toward the center of the pixels that pass the color filter. The first step, then, is to filter the image by color. The purpose of this exercise is not just to show how to follow lines, but to demonstrate how to subscribe to an image stream in ROS and push it though the OpenCV library in Python. This general pipeline could then be used in other application problems by tapping into the wide variety of excellent computer vision algorithms implemented in OpenCV.

Figure 12-4. A typical view from the Turtlebot's camera when following a line

The task at hand is to find the yellow line in the Turtlebot's image stream. The most obvious approach is to find the red, green, blue (RGB) value of a yellow image pixel and filter for nearby RGB values. Unfortunately, filtering on RGB values turns out to be a surprisingly poor way to find a particular color in an image, since the raw RGB values are a function of the overall brightness as well as the color of the object. Slightly different lighting conditions would result in the filter failing to perform as intended. Instead, a better technique for filtering by color is to transform RGB images into hue, saturation, value (HSV) images. The HSV image separates the RGB components into hue (color), saturation (color intensity), and value (brightness). Once the image is in this form, we can then apply a threshold for hues near yellow to obtain a *binary image* in which pixels are either true (meaning they pass the filter) or false (they do not pass the filter). The following code snippets and examples images will illustrate this process.

In Example 12-3, we implement this using OpenCV, which makes this task quite easy in Python.

Example 12-3. follower_color_filter.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image

class Follower:
  def __init__(self):
    self.bridge = cv_bridge.CvBridge()
    cv2.namedWindow("window", 1)
    self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                      Image, self.image_callback)
  def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg)
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([ 50,  50, 170])
    upper_yellow = numpy.array([255, 255, 190])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
    masked = cv2.bitwise_and(image, image, mask=mask)
    cv2.imshow("window", mask )
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()
```

As before, the `CvBridge` module converts ROS `sensor_msgs/Image` messages into the OpenCV image format:

```
image = self.bridge.imgmsg_to_cv2(msg)
```

We can then pass the OpenCV image to the `cvtColor()` function to convert between the RGB representation and its equivalent representation in the HSV space:

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

The `cvtColor()` function will produce the HSV image shown in Figure 12-5 when presented with the RGB image shown previously in Figure 12-4.

Then, in the HSV space, we can create lower and upper bounds for the desired hues using `numpy` and then pass those bounds to OpenCV's `inRange()` function to produce a binary image:

```
lower_yellow = numpy.array([ 50,  50, 170])
upper_yellow = numpy.array([255, 255, 190])
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
```

The resulting binary image is shown in Figure 12-6.

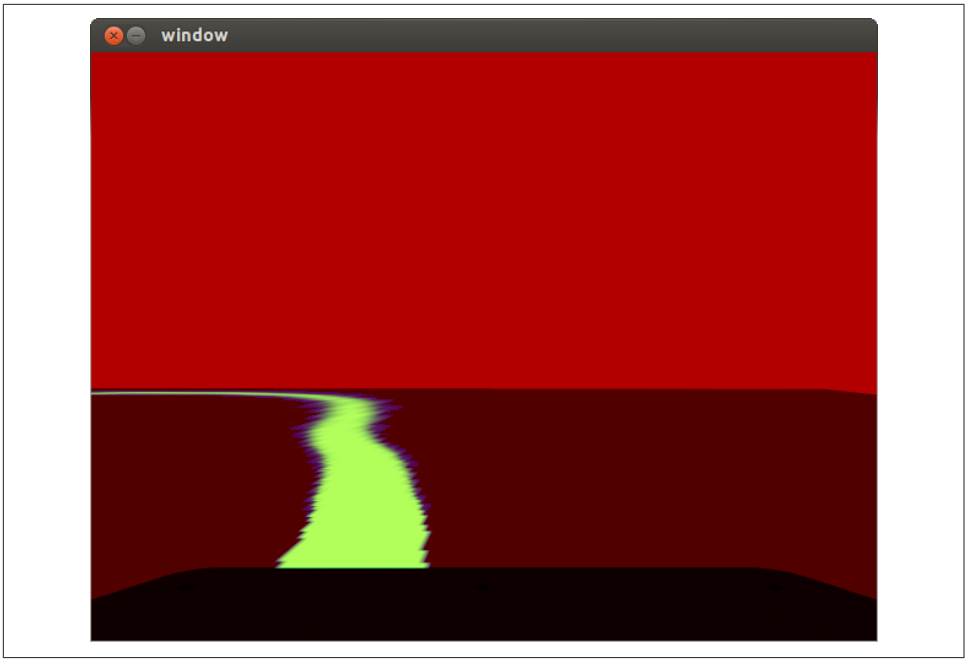

Figure 12-5. The HSV representation of a Turtlebot camera image when following a line

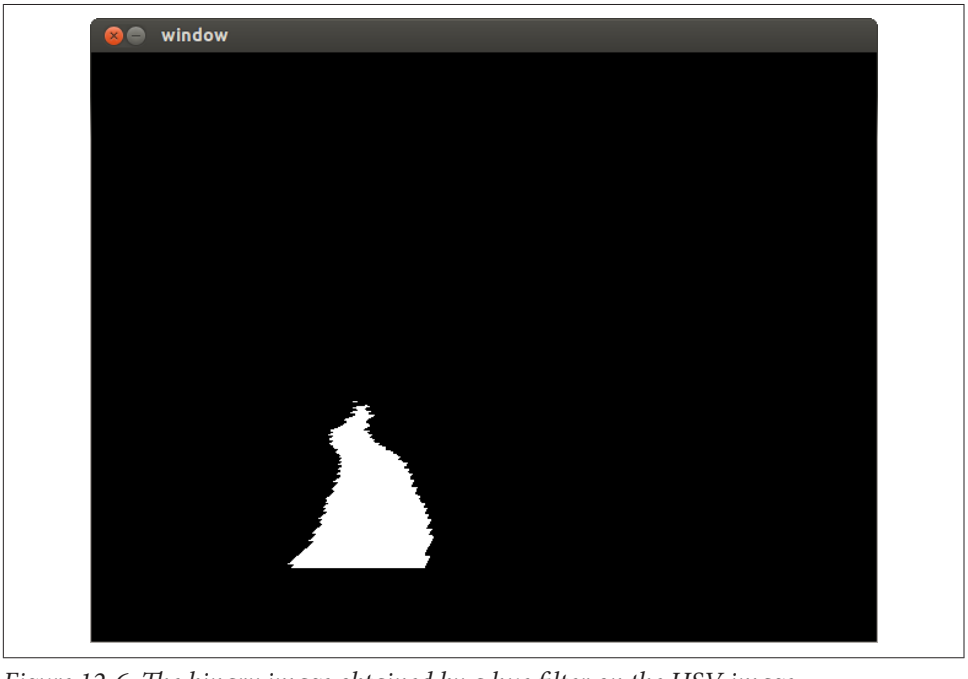

Figure 12-6. The binary image obtained by a hue filter on the HSV image

Obtaining a binary image of the line is a key first step in the image-processing pipeline. However, our goal is to follow the line, not just to take interesting pictures of it! To follow the line, we will use a simple strategy: we will only look at a 20-row portion of the image, starting three-quarters of the way down the image. The rationale behind this approach is that, from a controls perspective, we are really only concerned with the portion of the line that is immediately in front of the robot. With this strategy, what happens to the line five meters in front of the robot is irrelevant; our controller will only be concerned with what is in the field of view of the camera approximately one meter in front of the robot. To debug our implementation, we will first write a program, shown in Example 12-4, that implements this image processing strategy and draws a dot where it thinks the center of the line is within the portion of the image corresponding to roughly one meter in front of the robot.

Example 12-4. follower_line_finder.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image

class Follower:
  def __init__(self):
    self.bridge = cv_bridge.CvBridge()
    cv2.namedWindow("window", 1)
    self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                      Image, self.image_callback)
    self.twist = Twist()
  def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg,desired_encoding='bgr8')
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([ 10,  10,  10])
    upper_yellow = numpy.array([255, 255, 250])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    h, w, d = image.shape
    search_top = 3*h/4
    search_bot = search_top + 20
    mask[0:search_top, 0:w] = 0
    mask[search_bot:h, 0:w] = 0
    M = cv2.moments(mask)
    if M['m00'] > 0:
      cx = int(M['m10']/M['m00'])
      cy = int(M['m01']/M['m00'])
      cv2.circle(image, (cx, cy), 20, (0,0,255), -1)

    cv2.imshow("window", image)
    cv2.waitKey(3)

rospy.init_node('follower')
```

```
follower = Follower()
rospy.spin()
```

To restrict our search to the 20-row portion of the image corresponding to the one-meter distance in front of the Turtlebot, we will use the OpenCV and `numpy` libraries to zero out (i.e., erase any filter hits of) pixels outside the desired region. This code snippet uses the Python *slice notation* to express pixel regions in a compact syntax:

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

Then, we will use the OpenCV `moments()` function to calculate the *centroid*, or arithmetic center, of the blob of the binary image that passes our filter:

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

Finally, to help in debugging, it is often useful to draw calculations and estimates on top of the original camera image. In Example 12-4 we draw a solid red circle on the original RGB image to indicate the algorithm's estimate of the center of the line in the target image portion:

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

This will produce output similar to Figure 12-7.

It is important to note that Example 12-4 is written to handle not just still images, but continual image streams. To better understand the strengths and weaknesses, leave `follower_line_finder.py` up and running, and use the Move and Rotate tools in Gazebo to observe the behavior of `follower_line_finder.py` as the position and bearing change in simulation. Next, we will use the line-centroid estimation as our control input.

Figure 12-7. The original image with the red circle overlay, which shows the algorithm's estimate of the center of the line

Following the Line

In the previous section, we worked up to a line detection algorithm. Now that we have a line detection scheme up and running, we can move on to the task of driving the robot such that the line stays near the center of the camera frame. In Example 12-5, we demonstrate one approach to this problem: a P-controller. The P in this controller's name stands for *proportional* and simply means that a linear scaling of an error drives the control output. In our case, the error signal is the distance between the centerline of the image and the center of the line we are trying to follow.

Example 12-5. follower_p.py

```
#!/usr/bin/env python
import rospy, cv2, cv_bridge, numpy
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist

class Follower:
  def __init__(self):
    self.bridge = cv_bridge.CvBridge()
    cv2.namedWindow("window", 1)
```

```
    self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                      Image, self.image_callback)
    self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input/teleop',
                                       Twist, queue_size=1)
    self.twist = Twist()
  def image_callback(self, msg):
    image = self.bridge.imgmsg_to_cv2(msg,desired_encoding='bgr8')
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lower_yellow = numpy.array([ 10,  10,  10])
    upper_yellow = numpy.array([255, 255, 250])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    h, w, d = 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
    M = cv2.moments(mask)
    if M['m00'] > 0:
      cx = int(M['m10']/M['m00'])
      cy = int(M['m01']/M['m00'])
      cv2.circle(image, (cx, cy), 20, (0,0,255), -1)
      err = cx - w/2
      self.twist.linear.x = 0.2
      self.twist.angular.z = -float(err) / 100
      self.cmd_vel_pub.publish(self.twist)
    cv2.imshow("window", image)
    cv2.waitKey(3)

rospy.init_node('follower')
follower = Follower()
rospy.spin()
```

The P-controller is implemented in the following four lines:

```
err = cx - w/2
self.twist.linear.x = 0.2
self.twist.angular.z = -float(err) / 100
self.cmd_vel_pub.publish(self.twist)
```

The first line calculates the error signal: the distance between the center column of the image and the estimated center of the line. The following two lines calculate the values to be used for the Turtlebot's `cmd_vel` stream and scale it to something physically achievable by a Turtlebot. Finally, the last line publishes the `sensor_msgs/Twist` message to its peer nodes (in this case, is simply the Turtlebot base).

Although the code is surprisingly short, this system is actually doing some reasonable behavior and is able to follow lines in Gazebo.

Summary

In this chapter, we showed how to use OpenCV with ROS in Python. Specifically, we showed how to filter and threshold a ROS image stream by hue, and how to generate an error signal and drive a minimalist feedback controller. The result is a program that will drive a simulated Turtlebot to follow lines around a Gazebo simulation.

Even though line following has many useful applications, such as following road signage or factory floor markings, it is often not quite enough by itself. A common requirement for higher-level robot navigation is to travel between specific points on a map. In the next chapter, we will describe an approach to this problem using the ROS navigation stack and tools for creating and managing state machines.