Turtlebot Control & Sensor Data

ECE 4900/5900 Lecture Slides

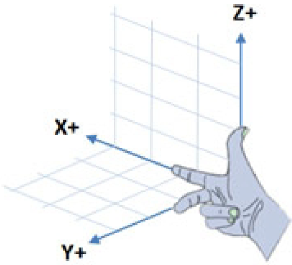
Fall 2019

Instructor: Wing-Yue Geoffrey Louie

Summary and Quick Links

These slides contain the following concepts:

ROS Coordinate Systems and Units



- ROS uses a right-hand convention for the orientation of coordinate axes

- For mobile robots this means:
  - The x-axis is forwards
  - The y-axis is left
  - The z-axis is upwards

ROS Coordinate Systems and Units

Below are the standard units of measure in ROS:

Quantity	**Unit**
length	meter
mass	kilogram
time	second
angle	radian
frequency	hertz
force	newton
linear velocity	m/s
angular velocity	rad/s

Launch Files

- Launch files are scripts to automate the running of a ROS system.
- They can be used to spawn any number of nodes, while also configuring parameters and topic names.
- All the vivid details can be found on the ROS wiki:
  - Command line usage
    http://wiki.ros.org/roslaunch/Commandline%20Tools
  - File syntax
    http://wiki.ros.org/roslaunch/XML

Launch Files

- Launch files follow XML syntax:

```
<?xml version="1.0"?>

<launch>
  <node pkg="package_name" type="node_type" name="node_name"
                                         output="screen" />
</launch>
```

- This launch file runs a single node using the **node** tag:
  - The compiled node name is **node_type**.
  - The node is compiled in the package **package_name**.
  - Its run-time name is changed to **node_name**.
  - The optional **output="screen"** allows any console output to be displayed in the terminal.

Launch Files

- While the **name** property must be set, it can be the same name as the node type.

- However, when launching multiple instances of the same node, their run-time names have to be different:

```
<launch>
  <node pkg="package_name" type="node_type" name="inst_1" />
  <node pkg="package_name" type="node_type" name="inst_2" />
</launch>
```

Launch Files

- Some of the common XML tags used in ROS launch files are:
  - **<node>** – Launches a particular node.
  - **<include>** – Used to include other launch files.
  - **<param>** – Sets a particular ROS parameter to a specific value.
  - **<rosparam>** – Used to load a set of parameters specified in a YAML file.
  - **<arg>** – Used to specify variable arguments to the launch file.

Timers

- Timers are used to execute code at periodic intervals.
- Node handles manage the timer. The user just specifies the desired period.
- Timers are usually instantiated in the main function of a node:

```
rospy.Timer(rospy.Duration(0.5), timer_callback)
```

- In this case, the timer is:
  - Set to trigger at a frequency of 2 Hz (period = 0.5 sec).
  - Set to call the **timer_callback** function when triggered.

Timers

- Timer callbacks are called with a "TimerEvent" structure argument:

```
def timer_callback(event):
    # Code for timer goes here
    print 'Timer called at ' + str(event.current_real)
```

- The TimerEvent contains four ROS time stamp values:
  - **current_real** – The system time as of when the callback function was called.
  - **last_real** – The system time at the *last* time the function was called.
  - **current_expected** – The scheduled time when the callback function was supposed to be called.
  - **last_expected** – When the last time this function was supposed to be called.

Simulated Robots in Gazebo

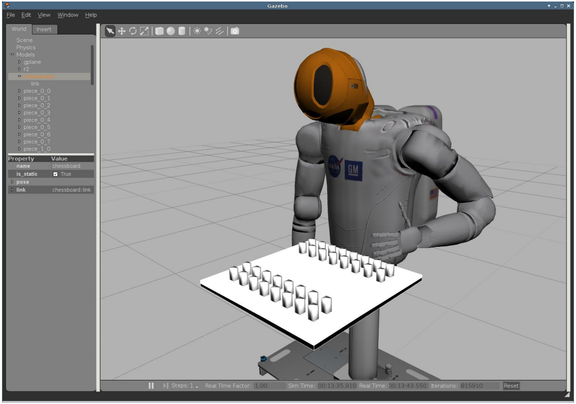
- Gazebo can be used to create applications for a robot without depending physically on an actual robot.

- Simulations have the advantage of:
  - **Reducing Cost** - No risk of damage when testing new apps, no need to purchase real physical hardware, no maintenance
  - **Reducing Time** - No need to charge a battery, run simulations in parallel (Great for ML!), no need to take time repairing hardware
  - **Experiments** - You can use any environment, robot, and sensors. Repeatable experiments for environment, hardware, and inputs

Simulated Robots in Gazebo

Simulations are limited because the:

- Cannot perfectly simulate the complexity of real-world environments, hardware failures, and noise
- Cannot currently simulate human-robot interactions
- Make numerous assumptions

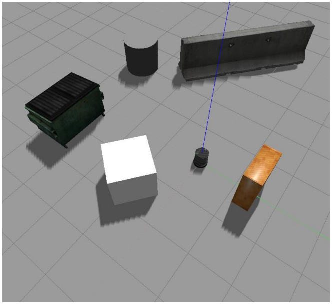


Running Turtlebot in Gazebo

There are numerous simulated robots available to be used in Gazebo which can be controlled through ROS. A list can be found here: ROS/Gazeo Robots

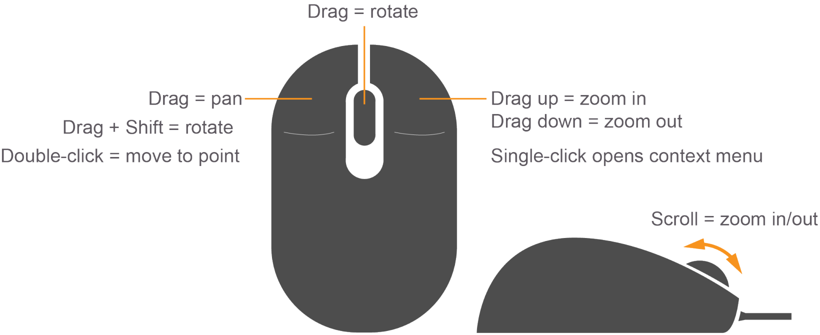
- If you have followed the class computer setup instructions we have already installed the Turtlebot simulation on your computer. Run the following in your terminal:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```



Running Turtlebot in Gazebo

Below is a graphic explaining how to view the gazebo simulation using your mouse:



Controlling Turtlebot

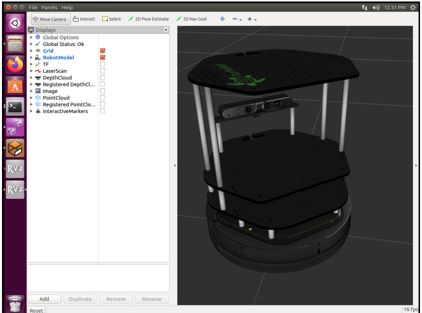
There are many ways to control the Turtlebot in ROS but we will discuss two as examples:

- Teleoperation using RVIZ interactive markers or a keyboard
- Programmatically through the rostopic **/cmd_vel_mux/input/teleop** which uses a **twist** message type

Controlling Turtlebot: Teleoperation

- RVIZ is a visualization environment in ROS to understand what the robot is "seeing, thinking, and doing".
- It allows you to visualize and log sensor information for debugging and development.
- To run RVIZ type the following in a terminal with Gazebo running:

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```



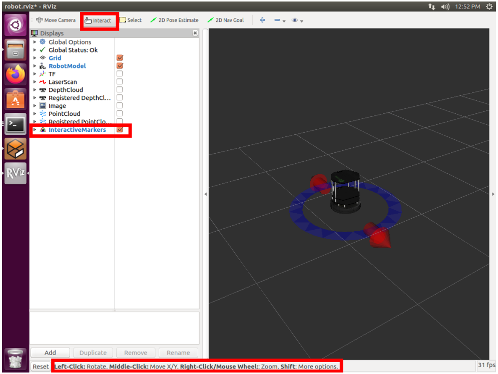
Controlling Turtlebot: Teleoperation

Now that RVIZ and Gazebo are running for Turtlebot you can control the robot using interactive markers by running the following:

```
$ roslaunch turtlebot_interactive_markers
interactive_markers.launch
```

Controlling Turtlebot: Teleoperation

- Make sure that the "InteractiveMarkers" checkbox in RVIZ is checked and you have clicked on the "Interact" button.
- Once those have been completed the blue (circle) and red (arrow) interactive markers will show up in RVIZ.
- Dragging the arrow will make the robot move back and forward and spinning the circle will make it rotate.



Controlling Turtlebot: Teleoperation

You can also control Turtlebot using a keyboard by running the following node:

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

- Instructions for use will pop up on the terminal for you to refer to

Message Headers

- Many messages published on ROS topics contain a *header* (**std_msgs.msg.Header**)
- A std_msgs.msg.Header is a structure that contains the following information:
  - **seq** — An unsigned 32-bit integer indicating the sequence number of the given message. This value is automatically incremented every time a message is published on the topic.
  - **stamp** — A ROS time stamp that is typically used to specify when the data contained in the message was generated.
  - **frame_id** — A string that indicates which reference frame the message's data is represented in (more on this later).

Message Headers

- Many ROS messages also have *stamped* versions, which means there is a **std_msgs.msg.Header** attached to the original message type.

student@ros-vm: ~

```
student@ros-vm:~$ rosmsg show geometry_msgs/Point
float64 x
float64 y
float64 z

student@ros-vm:~$
```

student@ros-vm: ~

```
student@ros-vm:~$ rosmsg show geometry_msgs/PointStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point point
  float64 x
  float64 y
  float64 z

student@ros-vm:~$
```

Controlling Turtlebot: ROS Twist Message

The following slides will:

- ▷ Define a ROS Twist message
- ▷ Describe how to create a Python class to encapsulate a ROS node
- ▷ Create a package to interact with the Turtlebot simulation
- ▷ Create a program where the robot plays red light/green light to demonstrate sending control commands to the turtlebot

Controlling Turtlebot: ROS Twist Message

- The Turtlebot movements can be controlled by publishing a ROS **Twist** message to the **/cmd_vel_mux/input/teleop** topic

```
geoff@ubuntu:~/hri-fall2019$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Controlling Turtlebot: ROS Twist Message

We've already discussed how to generate **package.xml** and **CMakeLists.txt** from scratch. There is a convenience function that automatically generates a ROS package for you.

- First go into your workspace **src** folder.

```
$ cd ~/course_year/ros/src
```

- Now use the **catkin_create_pkg** command:

```
$ catkin_create_pkg hri_bot rospy geometry_msgs sensor_msgs
```

- The general syntax for the **catkin_create_pkg** command is:

```
$ catkin_create_pkg {pkg_name} {depend_1} .... {depend_n}
```

Controlling Turtlebot: ROS Twist Message

- Since catkin_create_pkg only creates a **src** folder for C++ code you will still need to make a scripts folder and create the python file

```
geoff@ubuntu:~/hri-fall2019/ros/src/hri_bot$ mkdir scripts
geoff@ubuntu:~/hri-fall2019/ros/src/hri_bot$ cd scripts
geoff@ubuntu:~/hri-fall2019/ros/src/hri_bot/scripts$ touch simple_control.py
geoff@ubuntu:~/hri-fall2019/ros/src/hri_bot/scripts$ ls
simple_control.py
geoff@ubuntu:~/hri-fall2019/ros/src/hri_bot/scripts$
```

Controlling Turtlebot: ROS Twist Message

First we will setup our Python script with the necessary ROS specific libraries and messages

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

import rospy
from geometry_msgs.msg import Twist
```

This code snippet imports the:

- The ROS Python Libraries
- The necessary **Twist** message structure from the **geometry_msgs** packages

Controlling Turtlebot: ROS Twist Message

```python
class RedGreenLightBot:
  def __init__(self):
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist,
                                       queue_size=1)
    self.green_light = True
    self.red_light_twist = Twist()
    self.green_light_twist = Twist()
    self.green_light_twist.linear.x = 0.5
    rospy.init_node('red_light_green_light')
    rospy.Timer(rospy.Duration(3),self.light_change_callback)
    rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
    rospy.spin()

  def light_change_callback(self, event):
    self.green_light = not self.green_light

  def pub_drive_command(self, event):
    if self.green_light:
      self.cmd_vel_pub.publish(self.green_light_twist)
    else:
      self.cmd_vel_pub.publish(self.red_light_twist)
```

Controlling Turtlebot: ROS Twist Message

Create a Python class with a constructor

```
class RedGreenLightBot:
  def __init__(self):
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist,
                                       queue_size=1)
    self.green_light = True
    self.red_light_twist = Twist()
    self.green_light_twist = Twist()
    self.green_light_twist.linear.x = 0.5
    rospy.init_node('red_light_green_light')
    rospy.Timer(rospy.Duration(3),self.light_change_callback)
    rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
    rospy.spin()
```

The constructor:

- Initializes a publisher which publishes a Twist command

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

Controlling Turtlebot: ROS Twist Message

- Initializes Twist messages which command the robot to go a different velocities depending on whether it is a "green" (0.5 m/s forward) or "red" (0.0 m/s) light

```
self.green_light = True
self.red_light_twist = Twist()
self.green_light_twist = Twist()
self.green_light_twist.linear.x = 0.5
```

- Initializes the ROS node

```
rospy.init_node('red_light_green_light')
```

Controlling Turtlebot: ROS Twist Message

- Initializes two Timers which change the "light" every 3 seconds and publish a velocity command every 0.1 seconds

```
rospy.Timer(rospy.Duration(3),self.light_change_callback)
rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
```

- Keep the node running and processing callbacks from the timers

```
rospy.spin()
```

Controlling Turtlebot: ROS Twist Message

- Define the light_change_callback which inverts the state of the light every time the callback is called:

```
def light_change_callback(self, event):
  self.green_light = not self.green_light
```

- Define the pub_drive_command callback to send commands depending on the current state of the light. We need to continually publish a stream of velocity command messages because most mobile base drivers will time out and stop the robot if they do not receive at least several messages per second.

```
def pub_drive_command(self, event):
  if self.green_light:
    self.cmd_vel_pub.publish(self.green_light_twist)
  else:
    self.cmd_vel_pub.publish(self.red_light_twist)
```

Controlling Turtlebot: ROS Twist Message

Finally, we can simply initialize the class in the function and catch for exceptions when the node has been closed:

```
if __name__ == "__main__":
  try:
    RedGreenLightBot()
  except rospy.ROSInterruptException:
    pass
```

Controlling Turtlebot: ROS Twist Message

We can now fire up and test our robot control program (Don't forget to chmod +x your python script!):

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

```
$ rosrun hri_bot simple_control.py
  cmd_vel:=cmd_vel_mux/input/teleop
```

The second terminal command runs the ros node simple_control.py but also remaps the nodes "cmd_vel" topic to the "cmd_vel_mux/input/teleop" topic the Turtlebot software stack is subscribed to.

Sensor Data on the TurtleBot

The Turtlebot has numerous sensors on it including a 2D camera, bumper/cliff sensors, Depth camera, pseudo LIDAR, and encoders. A good way to find out what sensors are available is to use:

```
rostopic list
```

You can then find out more info about the topic using:

```
rostopic info {topic_name}
```

If there is a custom message you can then find out more info by:

```
rosmsg info {message_type}
```

Sensor Data on the TurtleBot: Bumper Sensors

The bumper sensors are located at the front left, right, and center of the robot. To access sensory information you need to subscribe to the topic "/mobile_base/events/bumper"

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rostopic info /mobile_base/events/bum
per
Type: kobuki_msgs/BumperEvent

Publishers:
 * /gazebo (http://turtlebot-Lenovo-FLEX-6-11IGM:42137/)

Subscribers: None
```

The topic publishes a custom msg with the event information described below

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rosmsg info kobuki_msgs/BumperEvent
uint8 LEFT=0
uint8 CENTER=1
uint8 RIGHT=2
uint8 RELEASED=0
uint8 PRESSED=1
uint8 bumper
uint8 state
```

Sensor Data on the TurtleBot: Cliff Sensors

The cliff sensors are located at the front left, right, and center of the robot. To access sensory information you need to subscribe to the topic "/mobile_base/events/cliff"

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rostopic info /mobile_base/events/cli
ff
Type: kobuki_msgs/CliffEvent

Publishers:
 * /gazebo (http://turtlebot-Lenovo-FLEX-6-11IGM:42137/)

Subscribers: None
```

The topic publishes a custom msg with the event information described below

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rosmsg info kobuki_msgs/CliffEvent
uint8 LEFT=0
uint8 CENTER=1
uint8 RIGHT=2
uint8 FLOOR=0
uint8 CLIFF=1
uint8 sensor
uint8 state
uint16 bottom
```

Sensor Data on the TurtleBot: Encoders Sensors

The Turtlebot has a set of encoders on its wheels which enables dead reckoning of the position of the robot and sensing the velocity of the robot. To access this information you can subscribe to the "/odom" topic.

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://turtlebot-Lenovo-FLEX-6-11IGM:42137/)

Subscribers: None
```

Sensor Data on the TurtleBot: Encoders Sensors

The topic publishes an Odometry message from the nav_msgs package

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rosmsg info nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Sensor Data on the TurtleBot: Encoders Sensors

- The **pose** value of the message provides the estimated position and orientation of the robot in 3D space. The origin is where the Turtlebot software stack started in 3D space.
- The **Twist** message then provides the current linear and angular velocities of the robot.

Sensor Data on the TurtleBot: Pseudo Laser Sensor

Although the Turtlebot does not have a laser range finder, pseudo laser sensory information is generated from the Astra/Kinect/Xtion depth sensor mounted on the robot using the depthimage_to_laserscan package. The pseudo laser information is published on the **scan** topic.

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rostopic info /scan
Type: sensor_msgs/LaserScan

Publishers:
 * /laserscan_nodelet_manager (http://turtlebot-Lenovo-FLEX-6-11IGM:34347/)

Subscribers: None
```

Sensor Data on the TurtleBot: Pseudo Laser Sensor

The topic publishes a LaserScan message from the sensor_msgs package

```
turtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rosmsg info sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Sensor Data on the TurtleBot: Pseudo Laser Sensor

The value we are most interested here is the **float32[] ranges** array. From this array we can estimate the bearing for a particular range estimate using the following equation:

```
bearing = msg.angle_min + i * msg.angle_max/len(msg.ranges)
```

To retrieve the range of the closest object we can simply use:

```
closest_range = min(msg.ranges)
```

To retrieve the range of an obstacle directly in front of the robot we can then use:

```
range_ahead = msg.ranges[len(msg.ranges)/2]
```

Wander Bot (Laser Example)

The following slides will:

- Create a program which reads LIDAR data
- Actuate a robot to wander around without running into any objects
- Create a launch file which runs existing launch files and nodes

Wander Bot (Laser Example)

As usual we will first setup our Python script file with the necessary libraries and messages it wil be using

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
```

Here we are using the:

- The **Twist** message from geometry_msgs
- The **LaserScan** message from sensor_msgs

Wander Bot (Laser Example)

We will then a create a class which encapsulates a ROS node to perform the following functions:

- Prevent the robot from getting close to objects (<0.8m)
- Change the state of the robot to turning or going forward every 5 seconds

```
    pass

class WanderBot():
  def __init__(self):
    #Class constructor code here

  def scan_callback(self,msg):
    #Code for when a laser scan is received

  def change_state_callback(self, event):
    #Code to change the state of the robot every 5 seconds
```

Wander Bot (Laser Example)

The WanderBot class constructor

```python
def __init__(self):
  rospy.init_node('wander_bot')
  rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
  rospy.Timer(rospy.Duration(5),self.change_state_callback)

  self.scan_sub = rospy.Subscriber('scan', LaserScan,
                                    self.scan_callback)
  self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist,
                                      queue_size=1)
  self.twist_msg = Twist()
  self.g_range_ahead = -1
  self.driving_forward = True
  rospy.spin()
```

Wander Bot (Laser Example)

We first initialize the "wander_bot" ROS node

```
rospy.init_node('wander_bot')
```

We will then create our **Timers** for changing the state of the robot's wander behavior every 5 seconds and publish drive commands based on the current state of the robot every 0.1 seconds

```
rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
rospy.Timer(rospy.Duration(5),self.change_state_callback)
```

Wander Bot (Laser Example)

The WanderBot class constructor

```
def __init__(self):
  rospy.init_node('wander_bot')
  rospy.Timer(rospy.Duration(0.1),self.pub_drive_command)
  rospy.Timer(rospy.Duration(5),self.change_state_callback)

  self.scan_sub = rospy.Subscriber('scan', LaserScan,
                                   self.scan_callback)
  self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist,
                                     queue_size=1)
  self.twist_msg = Twist()
  self.g_range_ahead = -1
  self.driving_forward = True
  rospy.spin()
```

Wander Bot (Laser Example)

Initialize a subscriber to obtain LIDAR data from the **scan** topic and a publisher to output **cmd_vel** to the robot.

```
self.scan_sub = rospy.Subscriber('scan', LaserScan,
                                 self.scan_callback)
self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist,
                                   queue_size=1)
```

Initialize variables which tracks the twist messages to be published to the robot, closest object to the robot based on laser data, and the current driving state of the robot.

```
self.twist_msg = Twist()
self.g_range_ahead = -1
self.driving_forward = True
```

Wander Bot (Laser Example)

We now define a function that does the following when LaserData message is received by the subscriber:

- Determines the distance of the closest object in a field of view of the Pseudo Laser Sensor
- Changes the behavior of the robot to spin when the closest object is less than 80cm away.

```
def scan_callback(self,msg):
  self.g_range_ahead = min(msg.ranges)
  if self.g_range_ahead < 0.8:
    self.driving_forward = False
```

Wander Bot (Laser Example)

This function inverts the behavior of the robot to either spinning on the spot or moving forward when the Timer processes a callback every 5 seconds.

```python
def change_state_callback(self, event):
  self.driving_forward = not self.driving_forward
```

Wander Bot (Laser Example)

This function then publishes the drive command to going forward at 1 m/s or spinning at 1 rad/s depending on the desired behavior of the robot at the initialized Timer rate (10hz)

```
def pub_drive_command(self,event):
  if self.driving_forward:
    self.twist_msg.linear.x = 1
    self.twist_msg.angular.z = 0
  else:
    self.twist_msg.linear.x = 0
    self.twist_msg.angular.z = 1
    self.cmd_vel_pub.publish(self.twist_msg)
```

Wander Bot (Laser Example)

Finally, we initialize the class to run the node in the main function

```
if __name__ == "__main__":
  try:
    WanderBot()
  except rospy.ROSInterruptException:
    pass
```

We can then run the node with:

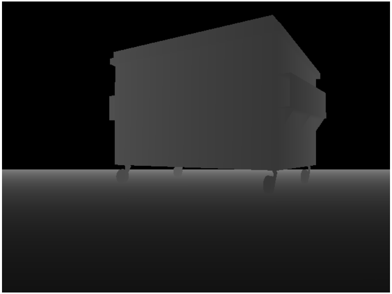
```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

```
rosrun hri_bot wander_bot.py
```

Sensor Data on the TurtleBot: Image Data

The Turtlebot also publishes 2D RGB Camera Data and Depth Camera Data which you can process using OpenCV. This sensory information is published as sensor_msgs/Image messages on the topics:

- RGB Image: /camera/rgb/image_raw
- Depth Image: /camera/depth/image_raw



Sensor Data on the TurtleBot: Image Data

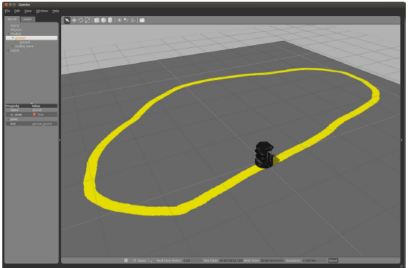
```
^Cturtlebot@turtlebot-Lenovo-FLEX-6-11IGM:~$ rosmsg info sensor_msgs/Image
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

- **height** - Height dimension of the image
- **width** - Width dimension of the image
- **encoding** - Encoding of pixels – channel meaning, ordering, size
- **is_bigendian** - Expresses order in which data is stores (LSB vs. MSB)
- **step** - Full row length in bytes
- **data** - Actual matrix data, size is (step * rows)

Follow Bot (OpenCV Example)

The following slides will:

- Create a program which reads image sensor data
- Converts the ROS **Image** message to a OpenCV **Mat** structure
- Process the image using some OpenCV functions
- Control a robot to follow a yellow line



Follow Bot (OpenCV Example)

First, we will need to include the dependencies to work with images. These dependencies include:

- sensor_msgs
- cv_bridge
- rospy
- std_msgs

In your ROS workspace **src** folder type:

```
$ catkin_create_package opencv_bot sensor_msgs cv_bridge
rospy std_msgs
```

Also create your scripts folder in the package and the node file **line_follower_bot.py**

Follow Bot (OpenCV Example)

The cv_bridge package does all the heavy lifting for converting ROS Image messages to OpenCV processable formats (i.e. **Mat**). You simply need to import the **CvBridge** object into your python file and use the **imgmsg_to_cv2** function. The generic usage of this is shown below:

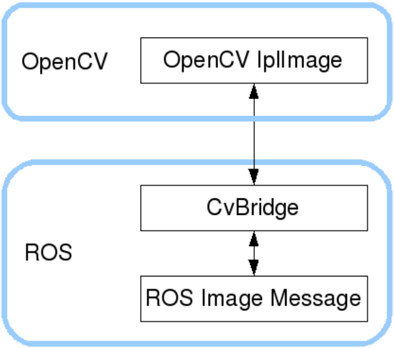
```
from cv_bridge import CvBridge, CvBridgeError

bridge = CvBridge()
bridge.cv2_to_imgmsg(mat_format_image, "bgr8")
```

You can also do the reverse (Mat->sensor_msgs/Image) with:

```
bridge.imgmsg_to_cv2(ros_image_msg, "bgr8")
```

Follow Bot (OpenCV Example)

- Once you have the image in the OpenCV **Mat** format you can play with all the capabilities of OpenCV
- For more OpenCV specific documentation you can refer to these two links:
  - OpenCV Python Tutorials
  - OpenCV Documentation



Follow Bot (OpenCV Example)

We will now import all the python packages we require for our node:

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

- **Numpy** - Package that efficiently deals with multidimensional data
- **cv2** - Package for computer vision
- **cv_bridge** - Package to convert ROS messages to OpenCV Mat structures
- **Image** - The message type used in ROS for images

Follow Bot (OpenCV Example)

Below is the basic class structure we will be using for our line follower robot:

```
class FollowBot:
  def __init__(self):
    # Sets up the ROS node, subscriber for images, publisher for
    # cmd_vels, and CvBridge

  def image_callback(self, msg):
    # Takes a ROS Image message, processes it in OpenCV to find
    # the yellow line, and controls the robot motion
```

Follow Bot (OpenCV Example)

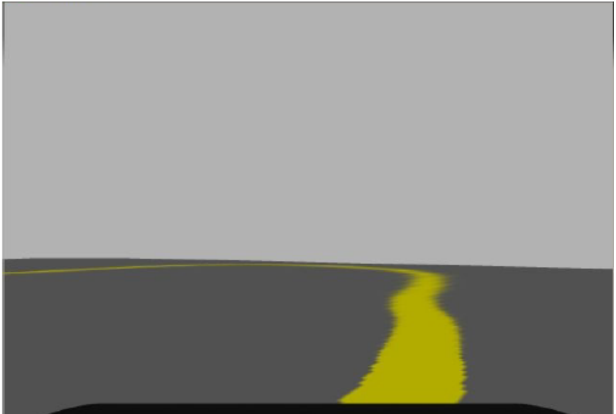
```python
def __init__(self):
  rospy.init_node('follower')
  self.bridge = cv_bridge.CvBridge()
  self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                     Image, self.image_callback)
  self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input/teleop',
                                      Twist, queue_size=1)
  self.twist = Twist()
  rospy.spin()
```

- Init the rospy node
- Init a subscriber to process the images published by the Turtlebot 2D camera
- Init a publisher to send command velocities to control the Turtlebot
- Init a Twist message that we will later populate with the robot motions
- Keep the program running and callbacks to be processed

Follow Bot (OpenCV Example)

For the image_callback function, we need to first change the ROS **Image** message to a OpenCv **Mat**

```
def image_callback(self, msg):
  image = self.bridge.imgmsg_to_cv2(msg,desired_encoding='bgr8')
```



Follow Bot (OpenCV Example)

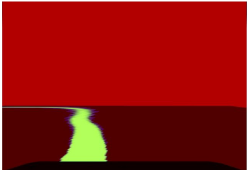
We then extract the yellow line from the image:

- We convert the color space from RGB to HSV to reduce challenges with illumination variance

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

- A simple threshold can then be applied to the 3 color channels to extract hues near yellow

```
lower_yellow = numpy.array([ 10, 10, 10])
upper_yellow = numpy.array([255, 255, 250])
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)
```



Follow Bot (OpenCV Example)

We then remove all pixels that are not in a 20 row portion of the image corresponding to ∼1m in front of the robot:

```
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
```

Follow Bot (OpenCV Example)

We then calculate the centroid of the blob for the binary image:

```
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)
```

The **cv2.moments** function is used to obtain the "moments" for the binary image.

$$M_{0,0} = \sum_{0}^{w} \sum_{0}^{h} f(x, y) \tag{1}$$

$$sum_x = \sum \sum x f(x, y) \tag{2}$$

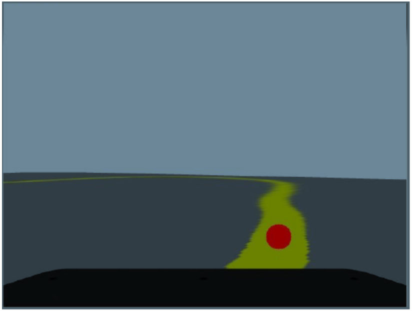
$$sum_y = \sum \sum x f(x, y) \tag{3}$$

Follow Bot (OpenCV Example)

$$M_{1,0} = sum_x / M_{0,0} \tag{4}$$

$$M_{0,1} = sum_y / M_{0,0} \tag{5}$$

Intuitively, these equations represent the centroid of a blob as the average of the x position and y position of all the pixels that are white in the image.



Follow Bot (OpenCV Example)

We then implement a basic proportional controller to control the movement of the robot

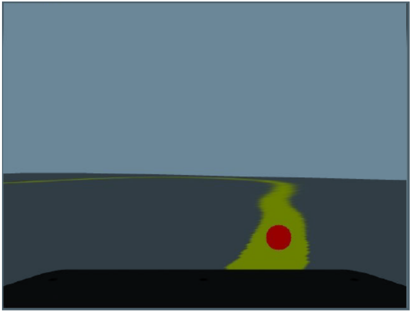
```
err = cx - w/2
self.twist.linear.x = 0.2
self.twist.angular.z = -float(err) / 100
self.cmd_vel_pub.publish(self.twist)
```

Namely, the control velocity command is always going 0.2 m/s forward and the angular velocity or rate at which the robot turns is directly proportional to how far the yellow line is from the center of the image.

Follow Bot (OpenCV Example)

We also visualize the image processing done by the node to obtain the centroid of the line

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



Follow Bot (OpenCV Example)

As usual we instantiate the class in the main function:

```
if __name__ == "__main__":
  try:
    follower_bot = FollowerBot()
  except rospy.ROSInterruptException:
    pass
```

We can then test our program by first bringing up a simulation of a course provided in the class repo:

```
roslaunch line_course course.launch
```

Then run the node we made:

```
roslaunch opencv_bot line_follower_bot.py
```