

Curs 3

1. Upload the code that you saved previously

For Windows machines, you can install WinSCP (<https://winscp.net/eng/index.php>) to upload/download code on the Raspberry Pi using the visual interface.

For Linux/Mac, you should have the scp command already available.

To upload onto the Raspberry Pi:

```
scp -r saved_workspace_folder pi@192.168.149.1:/home/pi
```

To download from the Raspberry Pi:

```
scp -r pi@192.168.149.1:/home/pi/workspace path_to_location
```

After uploading the code on the Raspberry Pi, check to see if the code runs properly. If copying from a Windows machine, check that the executable permissions are kept for the Python and setup scripts (the ones under the devel folder).

Running

```
catkin_make
```

may solve the issues with the devel folder scripts. If not, add execution permissions manually.

2. Subscribe to a topic

<https://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29>

a. Create a callback

A callback will generally be a regular Python function taking one argument of the same type as the message being received on the topic.

```
def callback(msg):
```

```
    rospy.loginfo("data: " + str(msg))
```

This callback will only print the message on the screen for now. **Note: the function does not need to be named "callback" specifically.**

b. Creating a subscriber

A callback is called whenever a message is received on the subscribed topic.

Creating a subscriber is very similar to creating a publisher:

```
rospy.Subscriber("topic_name", message_type, callback)
```

After we create the subscriber, the node needs to be kept alive to process incoming messages. In this case, we use **rospy.spin()**. This is very similar to a *while True* loop which will stop when rospy begins its shutdown process. We will use the same general structure as for the previous node (init, register shutdown function). The subscriber, so far, should look like this:

```
#!/usr/bin/env python3  
import rospy  
from sensor.msg import Led, RGB
```

```
ROS_NODE_NAME = "my_subscriber"
```

```
def callback(msg):  
    rospy.loginfo("msg: " + str(msg))
```

```
def cleanup():  
    rospy.loginfo("Shutting down...")
```

```
if __name__ == '__main__':  
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO)  
    rospy.on_shutdown(cleanup)  
    rospy.Subscriber(ROS_NODE_NAME + "/my_topic", Led, callback)  
    rospy.spin()
```

3. Modify the existing publisher to send the Led message to two topics

The subscriber that we have created will listen to a new topic (in our case `"/my_subscriber/my_topic"`) but currently, there is no one sending any messages on that topic. We can update the publisher we created previously to send the Led message to our subscriber as well as the original subscriber.

```
#!/usr/bin/env python3  
import rospy  
from sensor.msg import Led, RGB
```

```
ROS_NODE_NAME = "my_led_publisher"  
led_pub = None  
my_pub = None
```

```
def changeColor():  
    global led_pub, my_pub  
    led_pub = rospy.Publisher("/sensor/rgb_led", Led, queue_size=1)  
    my_pub = rospy.Publisher("/my_subscriber/my_topic", Led, queue_size=1)  
    colors = [(255, 0, 0), (255, 100, 0), (255, 255, 0), (0, 255, 0), (0, 0, 255), (75, 0, 130), (148, 0,  
211)]  
  
    size = len(colors)  
    i = 0  
    rate = rospy.Rate(1)  
    led = Led()  
    led.index = 0  
    while not rospy.is_shutdown():  
        led.rgb.r = colors[i][0]  
        led.rgb.g = colors[i][1]  
        led.rgb.b = colors[i][2]  
        led_pub.publish(led)  
        my_pub.publish(led)  
        i = (i + 1) % size  
        rate.sleep()
```

```
def cleanup():
```

```

global led_pub, my_pub
led = Led(0, RGB(0, 0, 0))
if led_pub != None:
    led_pub.publish(led)
if my_pub != None:
    my_pub.publish(led)
rospy.loginfo("Shutting down...")

if __name__ == '__main__':
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO)
    rospy.on_shutdown(cleanup)
    try:
        changeColor()
    except KeyboardInterrupt:
        pass

```

4. Services

<https://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29>

A service functions similarly to a REST service in that it exposes one method that can be called with some arguments and will return a result. The main differences between services and publishers/subscribers are:

- Services operate with messages that have both an input component and an output component. They can be configured to receive parameters and return responses. Publishers and subscribers communicate via topics and a publisher will only send data, without expecting a response, while a subscriber will only consume data.
- From a usage perspective, a service will not be called with a comparable frequency to that of a publisher or subscriber. Hence, the purpose of a service is to execute something that needs to happen just a few times per the lifetime of a node. (As an example, the **/puppy_control/go_home** service should be called at the beginning of an application that intends to use the robot's servos, to ensure that the joints are in a known position as well as at the end of the application, for the same reason)

As a quick example, we can programmatically call the **/puppy_control/go_home** service. For this, we need to identify what type of message we should send to it.

Running *rosservice info /puppy_control/go_home*:

```

pi@raspberrypi:~ $ rosservice info /puppy_control/go_home
Node: /puppy_control
URI: rosrpc://raspberrypi:40869
Type: std_srvs/Empty
Args:

```

We can see that the service uses a message of type **"Empty"**, which means that it takes no arguments and returns no value. Even so, we need to import this message into our code to use it. The service messages are

kept in the subpackage “srv” of a package. (Remember that the publisher/subscriber messages are kept in the “msg” subpackage of a package).

The beginning of our script that calls the **/puppy_control/go_home** will look like this:

```
#!/usr/bin/env python3  
import rospy  
from std_srvs.srv import Empty
```

We also need to create a service proxy, which is similar to a function reference and we can simply call it whenever we want to communicate with the server:

```
srv = rospy.ServiceProxy("/puppy_control/go_home", Empty)
```

The first argument is the service name. The second argument is the message that will be used. (very similar to how we specify a publisher or subscriber). It is recommended that, before calling a service, we wait to make sure that it is active, done with

```
rospy.wait_for_service("/puppy_control/go_home")
```

From here, things will look very similar to what we already know:

```
#!/usr/bin/env python3  
import rospy  
from std_srvs.srv import Empty  
if __name__ == '__main__':  
    srv = rospy.ServiceProxy("/puppy_control/go_home", Empty)  
    rospy.wait_for_service("/puppy_control/go_home")  
    srv() # this is how we call a service proxy with no arguments
```

5. OpenCV (can be practiced at home, using a webcam)

<https://pypi.org/project/opencv-python/>

Keep in mind that Puppy Pi is running **Python 3.7.3** and **OpenCV 4.5.4.60**. If you use other versions on your pc, you may need to make some changes to make the code run.

After installing OpenCV, we can import it in a python script using

```
import cv2
```

A simple program that captures the video feed from the default video device would look like this:

```
import cv2  
  
if __name__ == "__main__":  
    cv2.namedWindow("Camera")  
    vc = cv2.VideoCapture(0)  
  
    rval = True  
    if not vc.isOpened():
```

```
rval = False
```

```
try:
```

```
    while rval:
```

```
        rval, frame = vc.read()
```

```
        cv2.imshow("Camera", frame)
```

```
        cv2.waitKey(1)
```

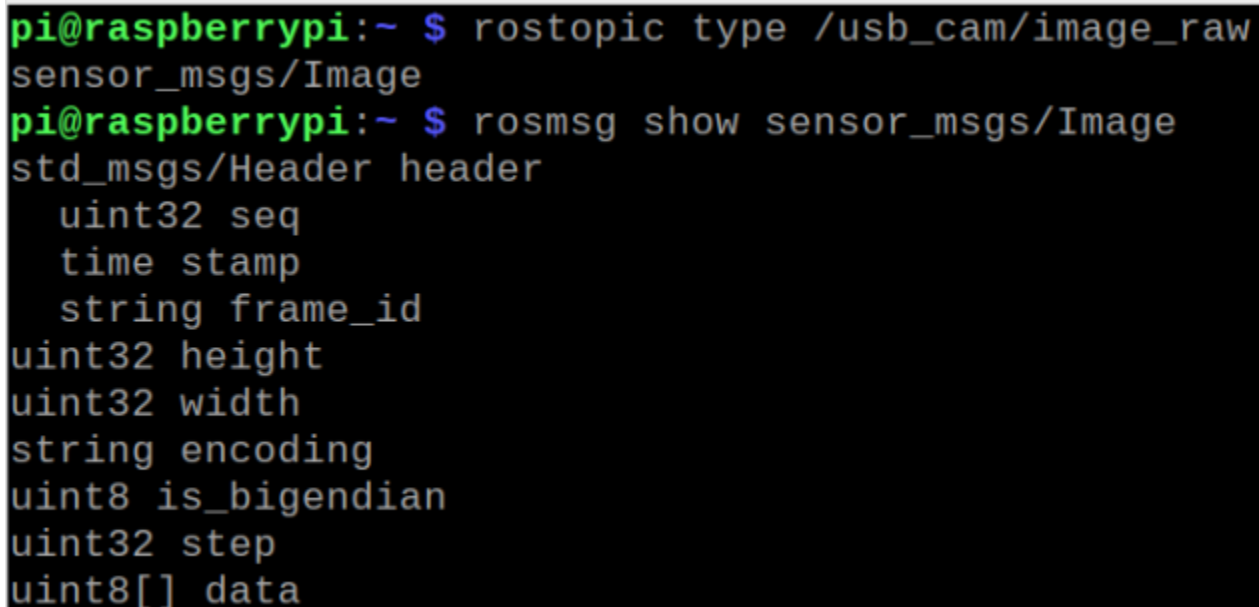
```
    except KeyboardInterrupt:
```

```
        cv2.destroyAllWindows()
```

```
        vc.release()
```

We can transfer this code onto the Raspberry Pi and we can link together data from the camera publisher with the code that displays each frame and test it. We'll start by subscribing to the `/usb_cam/image_raw` topic and log each incoming message to check that everything works correctly.

First, find the message type and the message structure:



```
pi@raspberrypi:~ $ rostopic type /usb_cam/image_raw
sensor_msgs/Image
pi@raspberrypi:~ $ rosmmsg show 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
```

We identify that the necessary import is

```
from std_msgs.msg import Image
```

The other important thing to notice is that the message contains more than the image captured by the camera. Even more, the image data is stored as an array of unsigned bytes, so we have to use OpenCV to convert it into an image before showing it.

To start, let's just print the image width and height. Everything else will be the same as subscribing to a topic.

```
#!/usr/bin/env python3
```

```
import rospy
```

```
import cv2
```

```
from sensor_msgs.msg import Image
```

```
ROS_NODE_NAME = "image_processing_node"
```

```
def img_process(img):
```

```
rospy.loginfo("image width: %s height: %s" % (img.width, img.height))
```

```
def cleanup():
```

```
    rospy.loginfo("Shutting down...")
```

```
if __name__ == "__main__":
```

```
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO)
```

```
    rospy.on_shutdown(cleanup)
```

```
    rospy.Subscriber("/usb_cam/image_raw", Image, img_process)
```

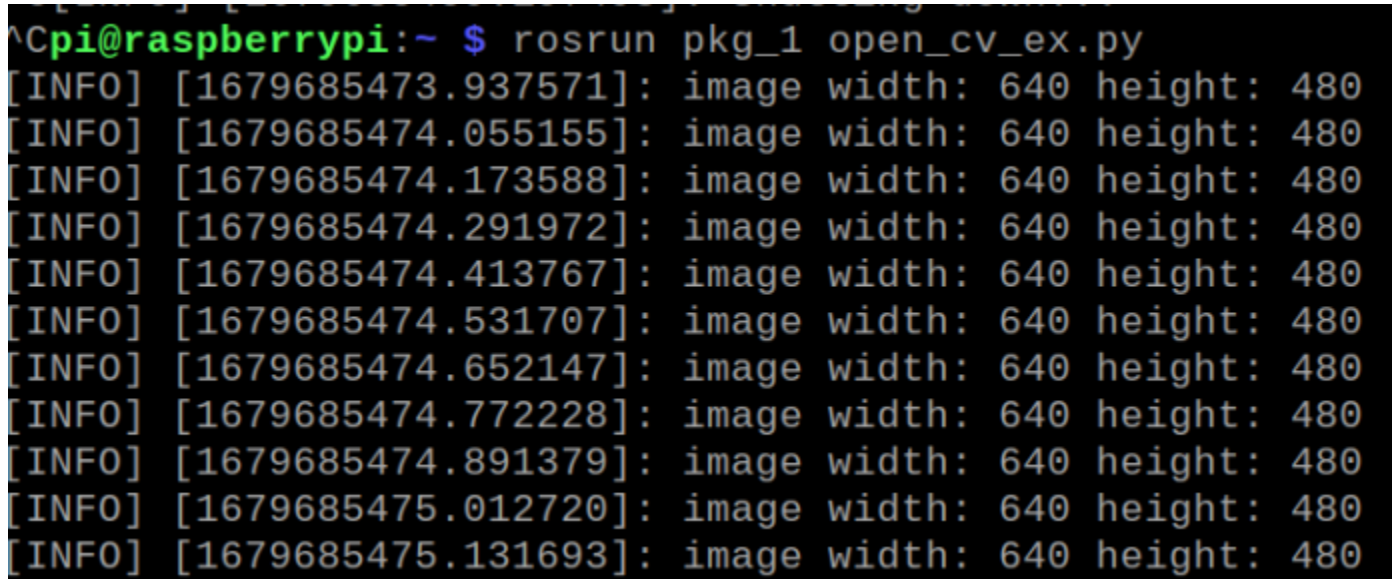
```
    try:
```

```
        rospy.spin()
```

```
    except KeyboardInterrupt:
```

```
        pass
```

Running the code above should result in the node printing the image resolution:

A terminal window on a Raspberry Pi showing the execution of a ROS node. The prompt is 'pi@raspberrypi:~ \$' and the command is 'roslaunch pkg_1 open_cv_ex.py'. The output consists of ten lines of log messages, each starting with '[INFO] [timestamp]: image width: 640 height: 480'. The timestamps are in the format [seconds.nanoseconds].

```
pi@raspberrypi:~ $ roslaunch pkg_1 open_cv_ex.py
[INFO] [1679685473.937571]: image width: 640 height: 480
[INFO] [1679685474.055155]: image width: 640 height: 480
[INFO] [1679685474.173588]: image width: 640 height: 480
[INFO] [1679685474.291972]: image width: 640 height: 480
[INFO] [1679685474.413767]: image width: 640 height: 480
[INFO] [1679685474.531707]: image width: 640 height: 480
[INFO] [1679685474.652147]: image width: 640 height: 480
[INFO] [1679685474.772228]: image width: 640 height: 480
[INFO] [1679685474.891379]: image width: 640 height: 480
[INFO] [1679685475.012720]: image width: 640 height: 480
[INFO] [1679685475.131693]: image width: 640 height: 480
```

To convert an array of bytes into an image we need to reshape the array according to its width, height, and depth. The width and height are already obtained, and the depth should be 3 (the depth is the number of color channels in an image, RGB in this case):

```
frame = np.ndarray(shape=(img.height, img.width, 3), dtype=np.uint8, buffer=img.data)
```

frame will now contain a 3D matrix with height rows, width columns and each element of the matrix is an array of 3 elements (the values for the red, green and blue channels)

OpenCV works with the BGR colorspace (which is essentially a permutation of the more common RGB colorspace) so we have to convert the frame data to BGR:

```
cv2_img = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
```

From this point on, we can treat **cv2_img** just like we treat a regular OpenCV frame captured from a camera.

```

#!/usr/bin/env python3
import rospy
import cv2
import numpy as np
from sensor_msgs.msg import Image

ROS_NODE_NAME = "image_processing_node"

def img_process(img):
    rospy.loginfo("image width: %s height: %s" % (img.width, img.height))
    frame = np.ndarray(shape=(img.height, img.width, 3), dtype=np.uint8, buffer=img.data)
    cv2_img = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
    cv2.imshow("Frame", cv2_img)
    cv2.waitKey(1) #this forces the window opened by OpenCV to remain open

def cleanup():
    rospy.loginfo("Shutting down...")
    cv2.destroyAllWindows("Frame")

if __name__ == "__main__":
    rospy.init_node(ROS_NODE_NAME, log_level=rospy.INFO)
    rospy.on_shutdown(cleanup)
    rospy.Subscriber("/usb_cam/image_raw", Image, img_process)
    try:
        rospy.spin()
    except KeyboardInterrupt:
        pass

```