ENEE 476: Robotics Project Laboratory
Visual Robot Perception in ROS 2
Lab 6

Learning Objectives

The purpose of this lab is to provide a hands-on introduction to 2D ROS 2 visual perception message interfaces, camera data manipulation using such interfaces, elementary image processing techniques, object pose estimation via fiducial markers (e.g., ArUco tags), and vision-enabled control. After completing this lab, students should be able to:

- Acquire, parse, and display ROS 2 images via ROS 2 topics
- Create ROS 2 nodes for publishing and subscribing different image modalities
- Create ROS 2 nodes for recognizing fiducial markers using OpenCV
- Serve and read the estimated pose of a marker-tagged object via ROS 2 actions
- Move a robot arm's tool within the vicinity of the ArUco tag's pose via ROS 2 actions.

Visual robot perception is often touted as the bedrock of modern robotics and refers to the process through which a robot acquires, interprets, and makes sense of its environment using cameras and other image sensors. Such scene understanding might involve capturing images, depth data, or point clouds and processing them to extract task-specific features such as the type, geometry, and pose of one or more objects in the robot's workspace. Thus, this lab seeks to impart fundamental skills for developing performant robot perception programs that enable intelligent robotic manipulation.

Related Reading

- M. Spong et. al., "*Robot Modeling and Control*", John Wiley & Sons, Inc., 2006.
- N. Atanasov, "*ECE276A: Sensing & Estimation in Robotics Lecture 10: Projective Geometry, Camera Model*." Lecture Notes.
- ROS Perception, "*vision_opencv*" Available: vision_opencv GitHub Repository.
- ros2_control Development Team, "*joint_trajectory_controller*" Available: Webpage.

Lab Instructions

Follow the lab procedure closely. Some code blocks have been populated for you and some you will have to fill in yourself. Pay close attention to the lab Questions (highlighted with a box). Answer these as you proceed through the lab and include your answers in your lab writeup. There are several lab Exercises at the end of the lab manual. Complete the exercises during the lab and include your solutions in your lab writeup.

List of Questions

List of Exercises

How to Read This Manual

Throughout the manual, boxes beginning with a **Question #** title contain questions to be answered as you work through the lab:

Question 0. A question for you.

Boxes with a symbol (like the one below) are extra theoretical details provided for your enlightenment or just general information about the lab. We recommend that you read them during the lab, but they may be skipped during the lab and revisited at a later time (e.g., while working on the lab report):

Some extra context and info.

Terminal commands and code blocks will be set in gray-colored boxes (usually for Python syntax)

```
import math # some important comment you should totally read
```

while terminal outputs will be set in yellow-colored boxes

```
some terminal output
```

Files and directories will be set in magenta color with a teletype font, e.g., `~/some/dir/to/a/file.txt`, code variable names and in-text Python commands or snippets will be set in blue teletype font, e.g., `a = [0, 0, 0]`, while Python type hints (and occasionally XML tags) will be set in green teletype font, e.g., `save(path: str)` signifies that the argument to the `save` function is a string. Lastly, figure captions go below figures, table captions appear above tables, and cyan text highlights a link to an external online reference.

1 Lab Procedure

1.1 Prerequisites

This lab assumes some familiarity with Linux commands, ROS 2 programming conventions, topics, message interfaces, and other ROS 2 graph components, including nodes, services, and actions. We also assume that students are familiar with rigid-body motion concepts and feedback control.

1.2 Getting this Lab's Code

To begin, in a new terminal session, navigate to the `~/Labs` directory and then clone the lab repo:

```
cd ~/Labs
git clone https://github.com/ENEE467-F2025/lab-6.git
cd lab-6/docker
```

Verify that the lab-6 image is built on your system

```
docker image ls
```

Make sure that you see the `lab-6` image in the output:

```
lab-6-image   latest    <image_id>     <N> days ago   <image_size>GB
```

Enable X11 forwarding by running the following command:

```
xhost +local:root
```

Then start the Docker container:

```
userid=$(id -u) groupid=$(id -g) docker compose -f lab-6-compose.yml run \
--rm lab-6-docker
```

Once inside the container, you should be greeted with the following prompt indicating that the container is running (where `<user_name>` is the active user account on the lab computer):

```
(lab-6) robot@<user_name>:~$
```

1.3 Test Your Setup

From within the container, build and source your workspace:

```
cd ~/ros2_ws/
colcon build --symlink-install
source install/setup.bash
```

Then test your build by running the following command from an active container:

```
cd ~/ros2_ws/src && python3 test_docker.py
```

This should print the following output to the terminal (stop and contact your TA if it doesn't):

```
All packages for Lab 6 found. Docker setup is correct.
```

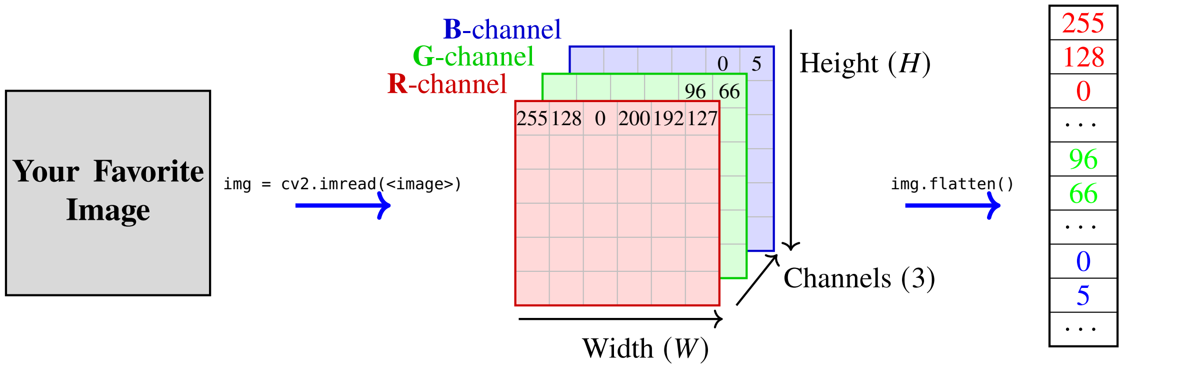
2 Basic Image Manipulation & Visual Feature Extraction

2.1 Digital Images: Structure and Representation

An image is a visual snapshot captured by a camera, represented digitally as discrete *pixels* arranged in 2D (or sometimes 3D), where each pixel holds a numeric value. Common 2D image types include:

- **Binary:** one bit per pixel, values 0 (black) or 1 (white)
- **Grayscale:** K-bit values (`uint<K>`) from 0 to $2^K - 1$
- **Color:** multiple channels per pixel, e.g., RGB (3 channels) or CMYK (4 channels)

The *size* of a 2D image is width × height. An RGB image is three stacked $H \times W$ arrays (one per channel), so each pixel stores three 8-bit (`uint8`) values $[0, 255]$, forming an $H \times W \times 3$ array.



(a) Pixel image **(b)** 3-channel matrix **(c)** Flattened image vector

Figure 1: Conversion of an RGB image (a) to a 3-channel array (b) and then to a row-major flattened vector (c). Blue arrows: OpenCV operations; black arrows: array indexing and channel order.

2.2 Acquiring & Parsing ROS 2 Image Messages from a (Simulated) Camera

Before processing images, we must first acquire them in ROS 2. Image data is exchanged via the `sensor_msgs/Image` message, which stores pixel values and metadata (e.g., width, height, encoding) for color, depth, or infrared streams. To launch the simulated camera environment:

```
ros2 launch ur3e_hande_gz view_gz.launch.py
```

This starts Gazebo and RViz with color and depth feeds from a simulated 720 × 1280 RealSense D435i camera. List available camera topics with:

```
ros2 topic list | grep camera
```

You should see a list of the following topics (we'll be making use of all but the last topic):

```
/rgbd_camera/camera_info # camera parameters
/rgbd_camera/depth_image # depth stream
/rgbd_camera/image       # color/RGB stream
/rgbd_camera/points      # point clouds (lab 8?)
```

In a new pane, probe the color image stream for information using `ros2 topic info`. Verify that messages of type `sensor_msgs/msg/Image` are streamed on this topic. Then answer Question 1.

Question 1.

a. What are the fields of the `sensor_msgs/msg/Image` message type? (**Hint**: Run `ros2 interface show <message_type>`.)

b. What frame convention (*optical* or *world*; see Figure 2) does the `frame_id` field of the `Image` message type in (a) above assume? (**Hint**: see the comments for `frame_id` in the interface definition from (a).)

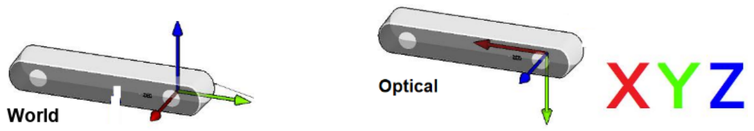


Figure 2: World and camera (optical) frame conventions: Camera frames often use an axis convention (typically ZYX) different from standard Cartesian frames.

After probing the topic, in a new `tmux` pane, introspect the color image stream (make sure the Gazebo simulation is running unless you will not record any terminal outputs!):

```
ros2 topic echo /rgbd_camera/image --once
```

The terminal prints a single `Image` message whose raw image is stored in the `data` field as a byte list. Pixels are arranged in *row-major order* (row by row, top to bottom) with channels in sequence (R $\rightarrow$ G $\rightarrow$ B). Each pixel occupies s bytes depending on the `encoding` (e.g., `mono8`, `rgb8`, `bgr8`). Let H denote the height (rows) and W the width (columns); then the `data` length is $H \times$ `step`, where `step` (bytes per row) equals $W \times s$.

2.2.1 Accessing Image Data from ROS 2 Messages

Let `offset` denote the *byte offset*, i.e., the number of bytes to skip in the `data` array to reach the first byte of a given pixel. Then

$$\texttt{offset} = \begin{cases} i \times s, & \text{for pixel index } i \text{ (0-based)} \\ (m \times \texttt{step}) + (n \times s), & \text{for pixel at row } m,\ \text{column } n \end{cases} \tag{1}$$

The pixel's channel values can then be read as:

```
data[offset : offset + s]  # upper bound excluded
```

2.2.2 Minimal Image Subscriber

We can implement a simple ROS 2 subscriber to listen for `Image` messages. To make it useful to the overall lab, we'll implement the subscriber in a single ROS 2 node called `SimpleImageAnnotator`, which we will gradually add features to, to evolve it into a fully functional image annotator. The `ament_python` package, `simple_image_annotator`, in the `src` folder contains a ROS 2 node (`image_annotator.py`) with the following template code (the version below is abridged):

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
```

```python
from sensor_msgs.msg import Image   # Class constructor for ROS2 Image message

class SimpleImageAnnotator(Node):
    def __init__(self):
        super().__init__('image_annotator')
        self.create_subscription(
            Image,
            '/rgbd_camera/image',
            self.color_image_callback,
            10)

    def color_image_callback(self, msg: Image):
        # Truncated for brevity
```

For now, the file contains a ROS 2 node implementing a simple subscriber that listens for messages on the *color* image topic and prints a few fields from the `Image` message. Terminate all running ROS 2 processes, build the package, and source your workspace:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select simple_image_annotator
source install/setup.bash
```

Then in separate `tmux` panes, start the simulator bringup using this command, and run the node:

```
ros2 run simple_image_annotator image_annotator --ros-args -p px_count:=10
```

The node prints metadata and the raw bytes of the middle `<px_count>` pixels (10 here) from the bottom row of the first received image, then exits.

```
Encoding: rgb8
Height x Width: 720 x 1280
Step (bytes per row): 3840
Data length: 2764800 bytes
Bytes/pixel (s): 3
Sampled pixels: [[115, 138, 157], [115, 138, 157], ...
Number of sampled pixels: 10
```

Run the node a few times. As you will notice, even for a simulated camera, each pixel appears to have slightly different intensity (± 1). This is due to lighting, shading, or depth-based effects, a problem more manifest in real settings.

2.2.3 Visualize the Retrieved Image Data

We can visualize sampled pixels to sanity-check our subscriber. The `utils` folder in the `/ros2_ws/src/simple_image_annotator/simple_image_annotator` directory contains a script, `plot_pixels.py`, for this purpose. First, terminate the `image_annotator` node. Next, copy the last array printed by your node (after `Sampled pixels:`) into the variable `pixels_to_plot`. Finally, change to the `/ros2_ws/src` folder and run the script:

```
python3 simple_image_annotator/simple_image_annotator/utils/plot_pixels.py
```

A plot will appear showing the middle 10 pixels of the bottom row from the color image stream, corresponding to a region on the table (hence appearing blue).

Question 2. Try running the image_annotator node for a few values of px_count. What happens when this argument is too high, say 1290? Over what range of values is px_count valid? (**Hint**: Recall that our camera's resolution is 720 (H) × 1280 (W).)

2.3 Image Manipulation in ROS 2 with OpenCV and CvBridge

Obviously, we want to do more than extract a few pixels. However, manually handling raw bytes is tedious for large images or varied encodings. OpenCV offers powerful tools, but ROS 2 images are flattened byte arrays. cv_bridge bridges the gap, converting between ROS 2 Image messages and OpenCV arrays without loss (for standard encodings like bgr8, rgb8, mono8). Let's examine a few use-cases.

2.3.1 Converting and Displaying Images

CvBridge provides the following convenient conversion functions.

- imgmsg_to_cv2: sensor_msgs.msg.Image → cv2.Mat (OpenCV image / NumPy array)
- cv2_to_imgmsg: cv2.Mat → sensor_msgs.msg.Image (ROS 2 image message)

Confirm that cv_bridge and opencv_python are installed in your container instance by running:

```
pip show cv-bridge opencv-python
```

You should see output in your terminal indicating that both packages are installed. Next, create an instance of CvBridge named bridge in your node class constructor (__init__), and add a class variable named full_color_image to hold the cv2 image we will generate:

```
self.bridge = CvBridge() # object for OpenCV-ROS2 translation
self.full_color_image = None  # class variable to hold the full color image
```

To read and display a ROS 2 image using OpenCV, we combine the imgmsg_to_cv2 method from our class's CvBridge() object and the imshow method from cv2. Verify this by adding the following lines to your color_image_callback class method after (and outside) the if block:

```
# convert and display the current image
try:
    full_color_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
    self.full_color_image = full_color_image
except CvBridgeError as e:
    self.get_logger().error(f"CvBridge Error: {e}")
    return
if not self.px_specified:
    cv2.imshow("Color Image", self.full_color_image)
    cv2.waitKey(3)
```

Then start the node (ensure the simulator is running):

```
ros2 run simple_image_annotator image_annotator
```

You should see an OpenCV window with the same image streamed by the simulated camera. Note that the encoding here is BGR (bgr8), unlike the RGB (rgb8) encoding used by the ROS 2 image message. OpenCV stores color images as BGR by default, but CvBridge handles the conversion correctly when bgr8 is specified as the desired_encoding. Press Ctrl + C to terminate the OpenCV window, and then answer Question 3.

Question 3. Run the simple_annotator node for the desired_encoding argument in the imgmsg_to_cv2 function set to (i). rgb8 (ii). mono8, and (iii). 16UC1. You will need to terminate the node between each change, but the simulator can remain running. In your report, briefly summarize your observations in each of the three cases above. No figures are required.

2.3.2 Feature Localization & Annotation in Images: Bounding Boxes & Centroids

Now that we can parse images with OpenCV, let's see how to *extract*, *localize*, and *mark* regions of interest (bounding boxes) and key geometric quantities (centroids) in pixel coordinates. Bounding boxes isolate relevant regions, while centroids provide a reference pixel (u, v) linking 2D image coordinates to 3D geometry (as in Exercise 2). Let $\mathcal{B} = \{(u_i, v_i)\}_{i=1}^{4}$ denote a bounding box. Its centroid is the arithmetic mean of the corners, i.e., $(u_c, v_c) = {}^1\!/_4 \sum_{i=1}^{4}(u_i, v_i)$.

We can implement this easily with NumPy using numpy.mean(<box>, axis=0).astype(int), where <box> is a NumPy array containing the pixels corresponding to $\mathcal{B}'s$ corners, which averages the corner coordinates along each image dimension to yield the centroid. Add the following objects to the class constructor of your image_annotator node:

```
self.latest_color_box = None          # store image[roi]
self.color_box_center_px: int = None  # store the center pixel

# Define ROI by bottom-left corner (u0, v0), width w, and height h
u0, v0 = 640, 334
w, h = 41, 36

self.roi = np.array([
            [u0, v0],         # bottom-left
            [u0, v0 - h],     # top-left
            [u0 + w, v0],     # bottom-right
            [u0 + w, v0 - h]  # top-right
]) # B in the manual
```

Then in your color_image_callback function, populate the variables you just created by adding the following code block after the self.full_color_image assignment line within the try block:

```
if self.full_color_image is not None:
    self.latest_color_box = crop_box(
                            self.full_color_image,
                            self.roi)
    ######## TODO #################
    self.color_box_center_px = None # <-- MODIFY
    ######## TODO #################

    # LEAVE THIS FUNCTION CALL AS IS
    if not self.px_specified:
        simple_annot(
        self.full_color_image,
        self.roi,
        self.color_box_center_px)
```

We provide two helper functions: `crop_box` extracts the ROI *image*, and `simple_annot` draws a green rectangle with a red centroid. Replace the `TODO` line with an `np.mean` call (see previous page), then run the annotator node. You should see the result in Figure 3 if your modification is correct. Note that the box and centroid annotations that appear in Figure 3 have been hardcoded to the specific coordinates of one face of the ArUco cube only for illustrative purposes. In the next unit, we will demonstrate how to *automatically* detect an ArUco tag within an image stream.

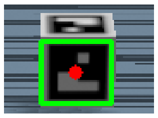


Figure 3: Visualizing the ROI and its centroid in the color image.

Great, our annotator node is almost complete. Before adding depth utilities and a publisher in Exercise 1, let's briefly note how depth images differ from color images.

2.4 Depth Images

A depth image assigns each pixel (u, v) a value $D(u, v)$ representing its distance from the camera along the optical Z-axis (see Camera Models). It is typically a matrix $D \in \mathbb{R}^{H \times W}$, where each element $D(u, v) = Z \in \mathbb{R}$ denotes depth in meters. Assuming depth and color frames share resolution (as in our case), their images will be of equal size. The RViz window already displays the depth image stream from the simulated camera. Echo a single `Image` message published on the depth topic (`/rgbd_camera/depth_image`) using:

```
ros2 topic echo /rgbd_camera/depth_image --once
```

You should see the following field names (truncated for brevity):

```
height: 720    width: 1280    encoding: 32FC1    step: 5120    data: ...
```

Observe that the field names are exactly the same as those of the color image; however, a few differences in some key values signal the type of image: an `encoding` of `32FC1` (`float32`) points to depth values in meters (the C1 indicates a single channel/scalar value per pixel). Each pixel encodes depth as a 32-bit float, so the image uses **4 bytes per pixel**.

Question 4. Given the depth image metadata from the message output above, explain how the `encoding` and the `step` field together confirm the number of bytes per pixel. (**Hint**: Recall that `step` $= W \times s$, where s is the number of bytes per pixel and W is the image width.)

Exercise 1. Simple Image Annotator (30 pts.)

In this exercise, we will extend the minimal subscriber we've been developing to:

a. Subscribe to the depth image stream
b. Determine the center and maximum depth values corresponding to the region of interest
c. Annotate the color stream with the depth values in b using OpenCV

Deliverables:

To complete this exercise, complete the SimpleImageAnnotator node class in image_annotator.py by performing the following steps:

Objects for Depth Image Processing:

a. In your node's __init__, create and initialize these class variables to None:
   • latest_depth_box • max_box_depth • center_box_depth • full_depth_image

Depth Image Subscriber:

b. Create a subscription in your node's __init__method with an Image message type, a topic name of /rgbd_camera/depth_image, and a QoS of 10. Set its callback to the class method depth_cb (see (c)).

c. Complete the depth_cb class method that processes Image messages via its argument msg to compute the quantities in (a). The file already contains a code template.

   Fill in the MODIFY-marked lines within the TODO-block as follows:

   – Set max_box_depth to the maximum depth within the bounding box in meters, computed only over valid (strictly positive) depth values stored in valid_depths.
   – Set center_box_depth to the depth at the center pixel, i.e., the value in self.full_depth_image at the coordinates cx, cy. Remember that row corresponds to y and column to x.

Depth Annotation:

d. Finally, annotate the color image using the computed variables from (c) by adding the following code block **after** the simple_annot function call in the color_image_callback (Note: this block is ready to use as is.):

```
## TODO: Ex 1d: Add depth annotation to the image
if (self.annot_depth
and self.max_box_depth is not None
and self.center_box_depth is not None
and self.roi is not None
and not self.px_specified):
    img_annotation_helper(
    max_box_depth=self.max_box_depth,
    center_box_depth=self.center_box_depth,
    roi=self.roi,
    full_color_image=self.full_color_image)
#################################################
```

Test Your Node:

Assuming packages were built with --symlink-install, start the simulator if it isn't already running, and run the node using:

```
ros2 run simple_image_annotator image_annotator --ros-args -p \
annot_depth:=True
```

If your logic is sound, your final output should look like the image depicted in Figure 4 (there are helper functions that provide the image annotation already in the code, so those will appear if the numbers are computed correctly and you do not need to annotate the image yourself). The terminal should also print:

```
Center Depth: 0.83 m, Max Depth: 0.86 m, Center Pixel: (316, 660)
```

Center and maximum depth values may vary due to sensor noise, but should be near 0.823 m and 0.853 m; the center pixel should be approximately $(316, 660)$ ($\pm$1–5 units).

Files to Submit

To receive full credit for this exercise, make sure you turn in the following files:

- Your completed `image_annotator.py` script implementing the methods above
- Screenshot of the `tmux` terminal displaying the center and maximum depth values
- OpenCV window showing the annotated image (as in Figure 4)

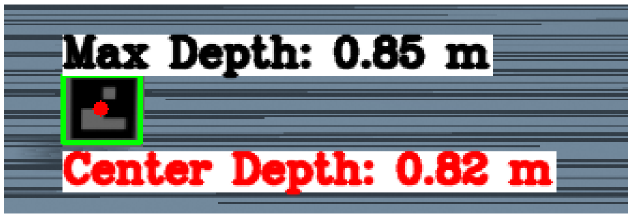


Figure 4: Target output for Exercise 1.

Nothing unusual: the center pixel is slightly closer than the corners. This exercise sets the stage for pose estimation. Next: ArUco tag detection.

3 Object Detection via ArUco Tags in OpenCV

In this second sub-unit, you will learn how to use a special class of fiducial markers, called ArUco tags, to facilitate object detection, by developing a ROS 2 package that can recognize these markers.

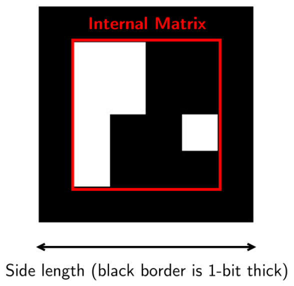
3.1 Fiducial Markers, Explained

Fiducial markers like ArUco tags are grayscale images encoding unique patterns that robot perception systems can quickly recognize for object identification, pose tracking, or location marking. Each marker is defined by a binary matrix that determines its ID, its physical size (side length in mm), and its dictionary (matrix dimensions). For example, a 100 mm ArUco marker from a 4×4 dictionary has 16 bits encoding its ID. For reference, see OpenCV's ArUco tutorial or use online generators (see: Online ArUco Markers Generator).

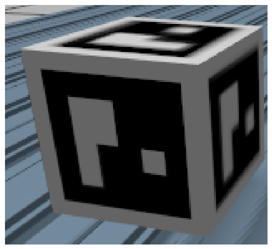
In OpenCV, some of the supported ArUco dictionaries include the following (note: the last integer in each dictionary's name corresponds to the maximum marker ID within that dictionary):

```
"DICT_4X4_250": cv2.aruco.DICT_4X4_250
"DICT_4X4_100": cv2.aruco.DICT_4X4_100 # ... and more
```

For a full list of parameters, see the ROS ArUco Pose documentation. To detect a tag with OpenCV,



(a) ID 80 ArUco tag.



(b) ID-80 ArUco cube in Gazebo.

Figure 5: The tagged object used in this unit.

its ID must be in the predefined dictionary. In this lab, we use a simulated cube with a 4x4 80 mm ArUco marker on *each* face (Figure 5) to ensure that the tag is always visible to the camera.

3.2 ROS 2 ArUco Message Interfaces

ROS 2 provides custom interfaces for ArUco tags via the `aruco_markers_msgs` package, including:

- `aruco_markers_msgs/msg/Marker`
- `aruco_markers_msgs/msg/MarkerArray`

Issue a `ros2 interface show` directive with the first message interface to see its contents:

```
std_msgs/Header header
uint32 id
geometry_msgs/PoseStamped pose
float64 pixel_x
float64 pixel_y
```

Observe that the `Marker` message references the tag's ID (`id`), the tag's center pixel coordinates ((`pixel_x`, `pixel_y`)), and the tag's Cartesian pose (`pose`). We now have the background we need to write an ArUco detection ROS 2 node.

3.3 Publishing ArUco Marker Detections

3.3.1 Test the ArUco Detector

The `ament_python` package, `aruco_pose_estimation` within the `src` folder contains `detect_aruco.py`, a Python file that implements a ROS 2 node for ArUco tag detection. Inspect it to see its contents. The file is mostly complete; we will mainly enhance it by adding a publisher to provide detections to ROS 2 and visualize these detections in RViz. Terminate all running ROS 2 processes, build the package, and source your workspace:

```
cd ~/ros2_ws/
colcon build --symlink-install --packages-select aruco_pose_estimation
source install/setup.bash
```

Then start the simulator using the command from here. Once the Gazebo and RViz windows load, run the detector in a new `tmux` pane using:

```
ros2 run aruco_pose_estimation detect_aruco
```

You should now see an OpenCV window titled `Detected ArUco Cube`, showing the camera's view with a green bounding box around the detected cube and text annotations containing the tag ID (`id=80`) rendered in blue on one or more cube faces. In the terminal running your node, you should see messages confirming detection of the cube with the specified ID:

```
[aruco_cube_detector]: Detected cube with ID: 80
```

Proceed to the next section only after confirming that your ArUco tag detector is working correctly.

3.3.2 Implementing an ArUco Detection Publisher

Our tag detection result (or image) still lives in OpenCV land, so we need a way to bring it into ROS 2. While it might seem that a single publisher would suffice, we actually use *two* publishers to keep the node modular and lightweight: one publishes the visual output (the annotated image) for tools like RViz, and the other publishes the geometric information required by downstream nodes that don't need the full image data.

Inside the `ArucoCubeDetector` node constructor (`__init__` method) within `detect_aruco.py`, create the visual-only publisher to broadcast the annotated image using the following code block:

```
self.aruco_img_pub = self.create_publisher(
    Image,
    '/aruco_cube_image',
    10)
```

Then create another publisher named `aruco_marker_pub` in the `__init__`method for publishing ArUco marker information. This publisher should publish messages of type `Marker` on the topic `/aruco_cube_marker`, with a QoS profile of 10. Next, in the node's image callback (`image_cb`), convert the OpenCV image (`frame`) to a ROS 2 `Image` message, and publish the output image:

```
out_msg = self.bridge.cv2_to_imgmsg(frame, encoding='bgr8') # frame is the
self.aruco_img_pub.publish(out_msg)                         # OpenCV detection
```

Finally, publish the ArUco `Marker` message, contained in the class variable, `latest_aruco_det`, using the publisher you just created (i.e., `aruco_marker_pub`).

3.3.3 Test Your ArUco Detection Publisher

Assuming you built your package with the `--symlink-install` directive at the start of this unit, run the simulator using the following command (note the `use_aruco` parameter):

```
ros2 launch ur3e_hande_gz view_gz.launch.py use_aruco:=true
```

This will start the simulator with an RViz instance containing a custom Image panel (with the title `Detected ArUco Cube` configured to display `Image` messages from our publisher's detection topic, **and also start the `detect_aruco` node**. So you will not need to run the AruCo detection node in this case. If there are no errors in your logic, you should now see exactly the same OpenCV view containing annotations of the detected tag within the custom RViz panel (see: Fig. 6).

We can confirm this by echoing a single message from the detections topic. With both the simulator and the ArUco detection node running (ensure `use_aruco:=true` was used to start the simulator),

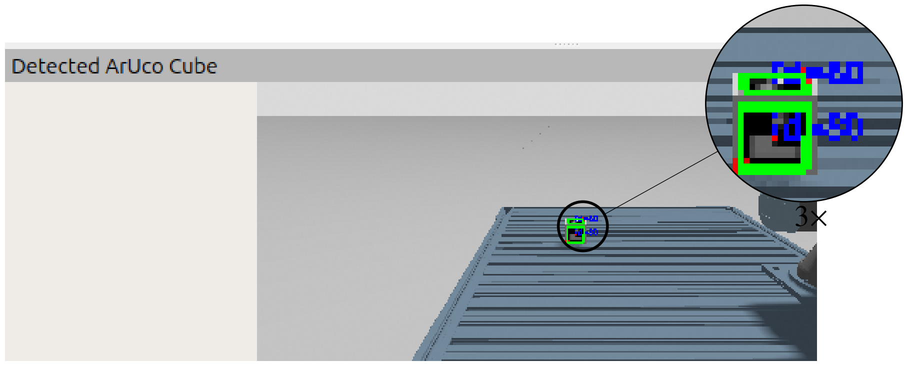


Figure 6: Detected ArUco cube in RViz. Zoomed inset highlights annotations.

in a new `tmux` pane, check that our new topics are available:

```
ros2 topic list | grep aruco_cube
```

You should see a list containing the two topics defined by the publishers we just created: i.e., `/aruco_cube_image` and `/aruco_cube_marker`. Run the following command to print a single `Marker` message from the ArUco marker topic:

```
ros2 topic echo /aruco_cube_marker --once
```

You should see the following (abridged) output :

```
id: 80
pose:
  frame_id: ''
  pose:
    position: {x: 0.0, y: 0.0, z: 0.0}
    orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}
pixel_x: 661.25
pixel_y: 316.0
```

Notice that the `id` and pixel coordinates (`pixel_x`, `pixel_y`) are accurate (compare the pixel coordinates here with the center pixel from Exercise 1). However, the `pose` field is empty (a generic Cartesian pose at the origin) because the detector provides only *2D* information, not the 3D pose needed for manipulation. We will address this in Section 4. For now, run the following command to print a single `Image` message from the ArUco image topic, then answer Question 5:

```
ros2 topic echo /aruco_cube_image --once
```

Question 5.

a. What is the value of the `frame_id` field in the ArUco marker message?

b. What is the encoding of the published ArUco image? Briefly explain in your report why it is or isn't different from the expected `rgb8` encoding.

4 Building an ArUco-Based Object Pose Action Server

By now, we can detect an ID-80 tagged object in the camera's field of view and pass both visual and geometric features between OpenCV and ROS 2. However, as seen previously, the marker message lacks the object's *pose*, which is essential for manipulation. To move the robot's end-effector or pick up the cube, we must know its **pose**. Let's address this next.

4.1 ROS 2 Pose Message Interfaces

Poses encode the 3D position ($p = [x, y, z]^\top \in \mathbb{R}^3$) and orientation ($R \in \mathrm{SO}(3)$) of a rigid body's frame. Using Euler angles ϕ (roll), θ (pitch), and ψ (yaw), a pose can be compactly written as $\xi = [x, y, z, \phi, \theta, \psi]^\top \in \mathbb{R}^6$. In ROS 2, pose information is transmitted via `geometry_msgs/Pose` and `geometry_msgs/PoseStamped`, where the latter simply adds a timestamp of type `std_msgs/Header`. Use `ros2 interface show` to inspect their contents.

```
Point position # x y z
Quaternion orientation # x y z w
```

(a) `geometry_msgs/Pose`

```
std_msgs/Header header
    # truncated for brevity
Pose pose
    Point position # x y z
    Quaternion orientation # x y z w
```

(b) `geometry_msgs/PoseStamped`

Figure 7: ROS 2 `Pose` and `PoseStamped` message definitions.

4.2 Deriving a Pixel's Pose Using Camera Parameters & Depth Information

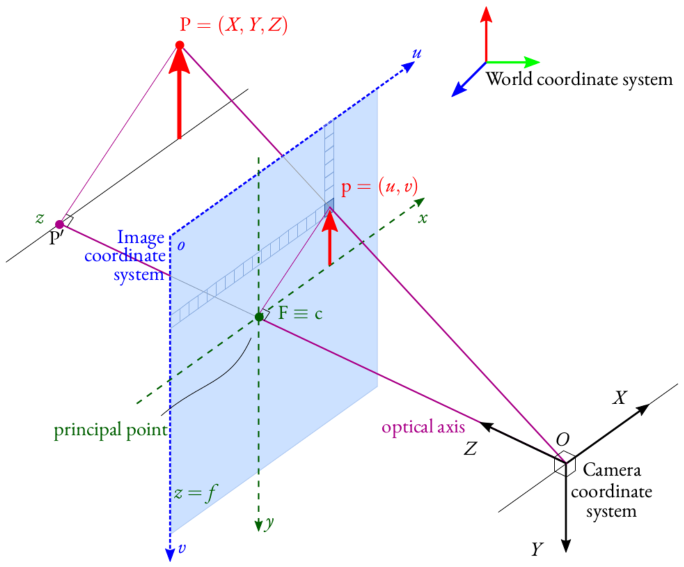


Figure 8: Pinhole camera model. O is the optical (or camera) frame. Source.

4.2.1 Retrieving Camera Parameters in ROS 2

To determine the pose of our cube in the world frame, we need a few equations that relate the observed pixels by the camera sensor to an actual 3D position in the world. For a pinhole camera model (see Figure 8), the coordinates of each observed pixel (u, v) in an image, $I(u, v)$, each correspond to a 3D point X_o, Y_o, Z_o in the camera's optical frame related by parameters of the perspective transformation matrix, as in

$$\begin{aligned} X_o &= \frac{(u - c_x) \cdot Z}{f_x} \\ Y_o &= \frac{(v - c_y) \cdot Z}{f_y} \\ Z_o &\equiv Z = D(u, v), \end{aligned} \tag{2}$$

where (c_x, c_y) is the camera's principal point, f_x, f_y are the components of its focal length along the optical abscissa and ordinate axes, and $D(u, v)$ is the depth value corresponding to the specified pixel coordinates, in mm or m, depending on the convention adopted by the camera manufacturer. Together, the parameters c_x, c_y, f_x, and f_y make up the camera's intrinsic matrix

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

In ROS 2, one can obtain K by simply reading messages of type `sensor_msgs/CameraInfo` published on a dedicated topic (usually namespaced by `camera_info`) by a node provided by the camera's ROS 2 driver or Gazebo plugin.

In a new `tmux` pane, start the simulator bringup, if it isn't already running, then echo data streamed on the camera info topic once:

```
ros2 topic echo /rgbd_camera/camera_info --once
```

In the output that's printed, the `k` field is arranged in row-major order and contains the camera parameters we need. Using the value for this field, answer Question 6.

Question 6. What are the values of c_x, c_y, f_x, and f_y? (**Hint**: see MathWorks's documentation on row-major array layouts)

4.2.2 Computing the Tag's Pose in the World Frame

Given the position in the camera's optical frame, we can obtain the corresponding world-frame position (t^W_{cube}) using the camera's extrinsics, i.e., the camera-to-world frame transformation:

$$\overbrace{\begin{bmatrix} R^W_{\text{cube}} & t^W_{\text{cube}} \\ 0_3 & 1 \end{bmatrix}}^{T^W_{\text{cube}}} = \overbrace{\begin{bmatrix} R^W_{\text{cam}} & t^W_{\text{cam}} \\ 0_3 & 1 \end{bmatrix}}^{T^W_{\text{cam}}} \cdot \overbrace{\begin{bmatrix} R^{\text{cam}}_{\text{cube}} & t^{\text{cam}}_{\text{cube}} \\ 0_3 & 1 \end{bmatrix}}^{T^{\text{cam}}_{\text{cube}}}, \tag{4}$$

where R^W_{cam} and t^W_{cam} are the rotation and translation of the camera in the world frame, and $R^{\text{cam}}_{\text{cube}}$ and $t^{\text{cam}}_{\text{cube}}$ are the rotation and translation of the cube relative to the camera.

Thanks to the `tf2_ros` package, we do not need to compute these transformations by hand and can simply perform a lookup via the terminal or within any ROS 2 node. To verify the camera-to-world transformation in ROS 2, we can use the `tf2_echo` utility to inspect the relative pose between frames. The syntax is:

```
ros2 run tf2_ros tf2_echo <source_frame> <target_frame>
```

This command returns the transform needed to move points (or get data) from the target frame (`<target_frame>`) **into** the source frame (`<source_frame>`). Note that this differs from the `lookupTransform` API approach (see here) and can be confusing at first. Let's see an example.

Start the simulator if it isn't already running (use the base launch command without parameters). Then in a new `tmux` pane, run the following command to determine the transform needed to get poses expressed in the (depth) camera's optical frame `camera_depth_frame` into the robot's reference (or world frame), `base_link` (a complete kinematic tree with all frame names is available within `tf_tree.pdf` in the `src/tf` folder):

```
ros2 run tf2_ros tf2_echo base_link camera_depth_frame
```

Once the first matrices appear, you can stop `tf2_echo`. This output gives the translation and rotation of `camera_depth_frame` in `base_link`, i.e., T^{W}_{cam} in Equation (4); reversing the arguments gives the inverse. Use it for Question 7.

Question 7. From the `tf2_echo` output, construct the homogeneous transform T^{W}_{cam}.

4.3 Writing the ArUco Pose Action Server

We will now create an action server that broadcasts (on request by a client ROS 2 node) the 6D pose information about a cube with a specific ID, if such a cube exists or reports failure and rejects the request otherwise. As you learned from the ROS 2 refresher lab, the `ActionServer` python constructor in ROS 2 takes as arguments the action interface specifying message types for the action goal or request, the server's response or result, and an optional feedback message.

The `ament_cmake` package, `aruco_interfaces`, in the `/ros2_ws/src` folder defines a custom action interface `GetArucoPose` that we'll need for our pose action server. Build the `aruco_interfaces` package, and source your workspace.

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select aruco_interfaces
source install/setup.bash
```

Then check that the `GetArucoPose` action interface definition exists using:

```
ros2 interface show aruco_interfaces/action/GetArucoPose
```

Verify that the following output is printed to the terminal (note the field names):

```
# Request
int32 marker_id
---
# Result
geometry_msgs/PoseStamped aruco_pose
---
# Feedback
bool marker_found # from std_msgs/Bool
```

Exercise 2. ArUco Marker Pose Server (30 pts.)

In this exercise, we will implement a ROS 2 `ActionServer` that listens for detections provided by our AruCo detector (`detect_aruco`) node and advertises the pose of the detected tag (in the camera's optical frame) by:

- Obtaining camera information from the `/rgbd_camera/camera_info` topic exposed by Gazebo via the bridge
- Determining the depth value corresponding to the marker's pixel coordinates
- Back-projecting the marker pixel and camera parameter and depth value to obtain the optical-frame pose of the ArUco tag.

Deliverables:

The `aruco_pose_estimation` `ament_python` package contains a Python file (`estimate_aruco_pose.py`) that we will finish developing to implement our action server. To complete this exercise, perform the following steps:

Camera Intrinsics Subscriber:

a. In your node's constructor (`__init__` method), create a subscription to the topic `/rgbd_camera/camera_info` for messages of type `CameraInfo`. Set the callback to the class method `camera_info_cb` to obtain the camera intrinsic matrix, and use a QoS of 10.

b. Create a class method `camera_info_cb` that processes the `CameraInfo` message stored in its argument (`cam_info`). Use the following template (the `TODO`-highlighted bits are yours to fill in):

```
def camera_info_cb(self, cam_info: CameraInfo):
    # Convert flat K vector to 3x3 matrix
    k = cam_info.k

    # TODO: Ex. 2b
    self.K = None # use Equation (3) and np.array;
                  # be sure to cast to float
                  # using dtype:=np.float32

    if not self._printed_camera_info and self.K is not None:
        self._printed_camera_info = True
        self.get_logger().info(
            "Camera intrinsic matrix set:\n"
            f"  fx={self.K[0,0]},\n"
            f"  fy={self.K[1,1]}"
        )
```

Complete this callback converting the flattened 9-element vector `k` to a 3×3 NumPy array, `self.K`. Use your answers from Question 6 and Equation (3).

ArUco Pose Action Server:

c. Create an `ActionServer` in your class constructor named `aruco_pose_server` in your node's `__init__` method. The server should be registered to your node, with an action type of `GetArucoPose`, string action name `/get_aruco_pose`, and execute callback, `get_aruco_pose_cb`.

d. Modify the class method, `get_aruco_pose_cb`, to back-project the marker pixel (u, v) using depth z and intrinsic parameters f_x, f_y, c_x, c_y to obtain (X, Y, Z) in the camera frame using Equation (2). The variables are named exactly as they appear in the formula except for the intrinsic parameters: c_x (`cx`), c_y (`cy`), f_x (`fx`), f_y (`fy`), and the depth corresponding to the ArUco centroid pixel, Z (`z`).

Build and Test Your Server:

After making your modifications, terminate all running ROS 2 processes, build the `aruco_pose_estimation` package, and source your workspace:

```
cd ~/ros2_ws
colcon build --symlink-install --packages-select aruco_pose_estimation
source install/setup.bash
```

Then create two new panes. In one pane, start the simulator bringup, setting the `use_aruco` parameter to `true`:

```
ros2 launch ur3e_hande_gz view_gz.launch.py use_aruco:=true
```

This will start the simulator and ArUco detector node. Once detections appear on RViz, run your action server in the second `tmux` pane:

```
ros2 run aruco_pose_estimation estimate_aruco_pose
```

You should see the following output if your implementation is correct:

```
Camera intrinsic matrix set: fx=642.3121337890625, fy=642.3121337890625
```

Leave the server running, then verify that the server is available:

```
ros2 action list  | grep aruco
```

You should see `/get_aruco_pose` in the returned output, the server is active, and you can proceed to send an action goal (in a new and sourced `tmux` pane) using an action client that we have provided for you as part of the `aruco_pose_estimation` package. The visualizer does the transformation in Equation (4), so you do not need to transform the pose results from the server:

```
ros2 run aruco_pose_estimation aruco_pose_visualizer
```

After running the client, you should see the following output in the terminal if your implementation is correct. Note that due to internal simulation initialization values and sensor

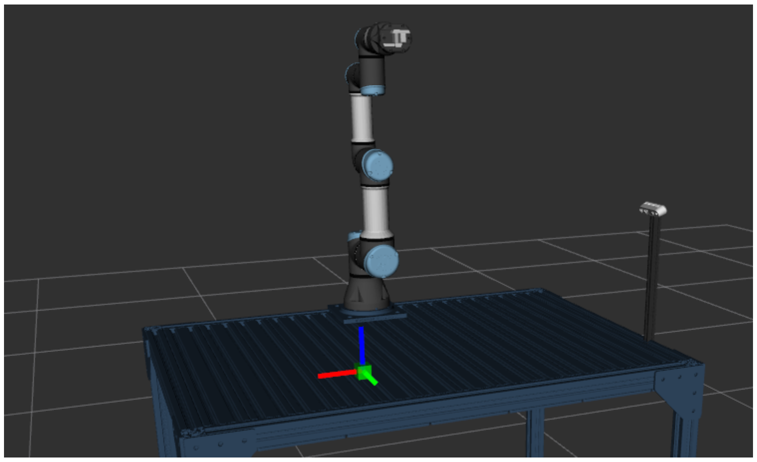


Figure 9: Detected ArUco cube visualized in RViz as a green marker at the server's computed pose. Axes indicate a default orientation.

noise (yes, even in simulation, sensors aren't perfect), your results may vary slightly from these:

```
Sent goal for ArUco ID 80
Goal accepted.
Transformed pose to base_link frame (0.16767978630730596,
0.5733807031046259, 0.023700766788989636).
Published marker for ArUco ID 80
```

and the `estimate_aruco_pose` server should assert pose estimation success, printing only the cube's *camera-frame* position (the orientation is fixed since we aren't moving the cube):

```
Estimated 3D position of marker ID 80: X=0.030929476090103516,
 ↪  Y=-0.06957668455306647, Z=0.8182250261306763
```

Monitor the RViz window. Once the client obtains the pose from the server, it will publish a `visualization_msgs.msg.Marker` messages of type `CUBE` (green cube similar in size to the ArUco cube in Gazebo) and type `LINE_LIST` (RGB axes) to highlight the computed world-frame pose (see: Figure 9).

Files to Submit

To receive full credit for this exercise, make sure you turn in the following files:

- Your completed `estimate_aruco_pose.py` script implementing the methods above
- A screenshot of your `tmux` terminal displaying the output printed by the `aruco_pose_visualizer` client
- A screenshot of your `tmux` terminal displaying the estimated pose of the detected ArUco tag as printed by the `estimate_aruco_pose` server
- RViz window showing the published pose marker (cube + RGB axes) in the robot's workspace (see: Figure 9).

5 Closing the Perception-to-Action Loop

Great! Our robot can now detect an ArUco-tagged object within its observable workspace and determine its pose relative to the robot's base frame. This brings us to the stage where we can close the perception–action loop at a high level: using a *static* pose estimate from the marker to plan and execute a motion. Since there are no obstacles in the workspace, straightforward trajectory generation and control suffice. The `ros2_control` library provides a ready-made action interface for this purpose. One such interface is the `FollowJointTrajectory` action from the `control_msgs` ROS 2 package. Let's sample it.

5.1 Send a Pre-Computed Joint Goal via the Command Line

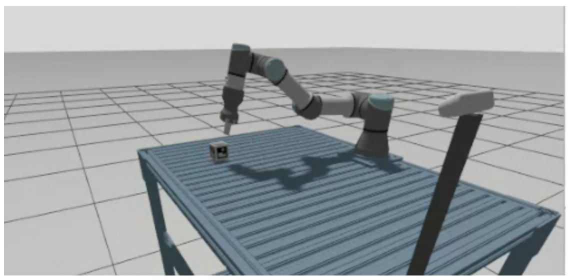
Before diving in, recall that in general, reaching a target pose requires computing joint values via inverse kinematics (IK). Although the joint goal was computed via inverse kinematics, we will not perform the IK calculation here; instead, we use the pre-computed goal directly. With the simulator active, we can send an action goal using the following syntax:

```
ros2 action send_goal [--feedback] <action_name> <action_type> "<goal_yaml>"
```

We've set up a controller interface that scales control commands to the robot, exposed via the action topic `/scaled_joint_trajectory_controller/follow_joint_trajectory`. Use the command below to send a simple joint goal through this interface:

```
ros2 action send_goal \
/scaled_joint_trajectory_controller/follow_joint_trajectory \
control_msgs/action/FollowJointTrajectory "{trajectory:
  {joint_names: [
    'shoulder_pan_joint',
    'shoulder_lift_joint',
    'elbow_joint',
    'wrist_1_joint',
    'wrist_2_joint',
    'wrist_3_joint'],
  points: [
    {positions: [
    -1.8390587011920374,
    -3.1046506367125453,
    0.6568387190448206,
    -2.020653387109274,
    1.6111266613006592,
    -0.007072750722066701
    ], time_from_start: {sec: 3}}
   ]}}"
```

You should see the robot move to a pose near the cube. In an ideal setup, the pose would be fed to an action client that computes the corresponding joint goal via inverse kinematics; however, we will defer this routine to a later lab.


(a) Robot at joint goal in Gazebo.

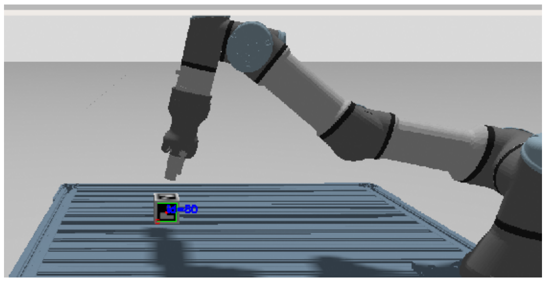

(b) RViz view.

Figure 10: Expected result after running the control action goal CLI command.

6 Conclusion and Lab Report Instructions

After completing the lab, summarize the procedure and results into a well written lab report. Include your solutions to all of the question boxes in the lab report. Refer to the List of Questions to ensure you don't miss any. Include your full solution to the exercises in your lab report. If you wrote any Python programs during the exercises, include these files as a separate attachment, appending your team's number and last names (in alphabetical order) to the file, e.g., **Jane Doe**, **Alice Smith**, and **Richard Roe** in **Team 90** submitting their modified `main.py` would submit the file `main_Team90_Doe_Roe_Smith.py` and the report `Lab_<LabNumber>_Report_Team90_Doe_Roe_Smith.pdf`, where `<LabNumber>` is the number for that week's lab, e.g., **0**, **1**, **2**, etc. Submit your lab report along with any code to ELMS under the assignment for that week's lab. You must complete and submit your lab report by **Friday, 11:59 PM** of the week following the lab session. See the **Files** section on ELMS for a lab report template named `Lab_Report_Template.pdf`. The grading rubric is as follows:

Component	Points
Lab Report and Formatting	10
Question 1.	5
Question 2.	5
Question 3.	5
Question 4.	5
Question 5.	5
Question 6.	2.5
Question 7.	2.5
Exercise 1.	30
Exercise 2.	30
Total	100