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

- ROS Perception, "*vision_opencv*" Available: vision_opencv GitHub Repository.

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):

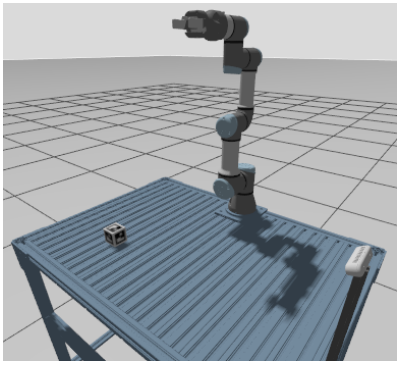
```
All packages for Lab 6 found. Docker setup is correct.
```

1.4 Required ROS 2 Packages

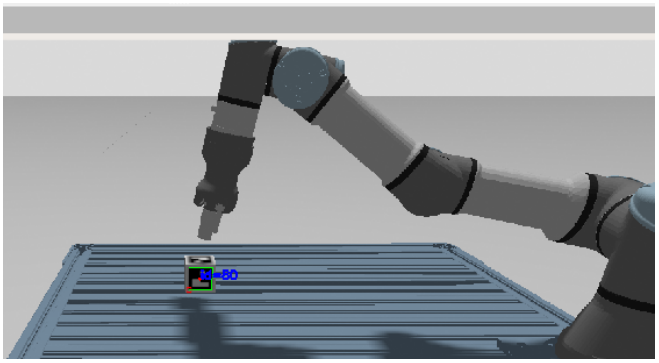
To support the programs your will develop over the course of this lab procedure, we have provided the following ancillary ROS 2 packages (they are located within the `src` folder):

- `aruco_interfaces`: Custom interfaces for pose estimation of AruCo markers
- `ur3e_hande_gz`: Gazebo world, models, and robot spawn configuration
- `ur3e_hande_description`: Unified URDF/XACRO and meshes for UR3e + Hand-E system
- `robotiq_hande_description`: Base description package for the Robotiq Hand-E gripper.

Unless otherwise instructed, these packages are not to be modified during the lab procedure.



(a) Gazebo simulation environment.



(b) Robot with tool frame near detected ArUco tag as visualized in RViZ.

Figure 1: An ArUco tagged cube visualized in Gazebo and recognized using ROS 2 tools and OpenCV. In the final exercise, you'll program the robot to move its end-effector to the tag's estimated pose (Figure 1b).

1.5 ROS 2 Perception Message Interfaces

ROS 2 nodes exchange images using the `sensor_msgs/Image` message interface. While other formats like `LaserScan` and `PointCloud2` exist, this lab focuses on images. The `Image` message supports color, depth, and infrared data, which must be parsed carefully due to their binary structure.

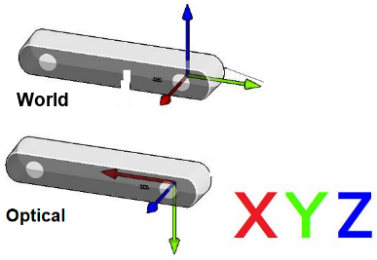


Figure 2: World-centric and (camera) optical frame conventions.

1.6 Optical vs. World-Centric Frame Conventions in ROS 2

Camera frames often use a different axis convention than standard (X, Y, Z) Cartesian frames, typically ZYX. While you'll mostly work with camera poses expressed in an inertial frame, it is useful to inspect transformations when needed. ROS 2 provides tools for this via the `tf2` library.

2 Basic Image Manipulation & Visual Feature Extraction

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

Before performing any image processing, let's first learn how to acquire and parse image data in ROS 2. As discussed in Section 1.5, ROS 2 represents images using the `sensor_msgs.msg.Image` message type. This message encodes the pixel data along with metadata such as width, height, and encoding type. Let's examine its contents.

Assuming packages in your workspace have already been built using `--symlink-install`, begin by loading the simulation environment we'll be working with, using the command:

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

This should start up Gazebo, with the Gazebo bridge configured, and load up an RViz instance configured to display color and depth image streams (with respective panel titles `RGB Image` and `Depth Image`) of a simulated 1280×720 RealSense D435i camera. In a new terminal, inspect what camera-specific topics are available 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
```

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.

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 following link.)

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 raw image is stored in the `data` field of length equal to the product of the `step*height`,

2.2 Interoperability with OpenCV

While OpenCV provides powerful image manipulation tools, ROS 2 images are stored as flattened byte arrays. To bridge this gap, we use the `cv_bridge` package, which converts between ROS 2 image messages and OpenCV image objects. Properly translating images is essential for accurate

processing and visualization. We will rely heavily on `cv_bridge` throughout the lab; see the `cv_bridge` Python API for reference.

> **Sim-to-Real Concerns in Perception**
> The same ROS 2 message interfaces (`sensor_msgs/Image`, `sensor_msgs/CameraInfo`) are used for simulated and real cameras. Simulation simplifies real-world challenges, like lighting changes, sensor noise, motion blur, and lens distortions. Thus, robust perception in reality may require additional handling.

2.3 Extracting Features from Images

In robotic manipulation, perception serves to answer questions that *enable* action:

- Where is the object the robot can interact with?
- How should the robot move relative to it?
- What region of the image corresponds to a feasible grasp or approach direction?

Exercise 1. ROS 2 Simple Image Annotator (20 pts.)

In this exercise, you will complete code to read and manipulate images streamed from a simulated RGB-D camera sensor using the ROS 2 `cv_bridge` package and the Open Computer Vision (OpenCV) Library .

Deliverables:

To begin, open the `image_annotator.py` file in the `scripts` folder with the following template code:

```
...
```

Modify the sections within `color_cb` and `depth_cb` callback functions that are marked with the multi-line comment '**your code here**' (lines **67**, **110**, and **111** in the provided template code from GitHub) to return the actual numerical values of `center_box_pixel`, `center_box_depth`, and `max_box_depth`. Run the node after making your modifications using

```
ros2 run lab_aruco_pose simple_image_subscriber_node.py
```

If your logic is sound, your final output should look like the image depicted in Figure 3 (there are helper functions that provide the image annotation, so those will appear if the numbers are computed correctly and you do not need to annotate the image yourself). The center and maximum depth values should be respectively close to 0.97 m and 1.05 m for the given region, and the center box pixel coordinates should be close to `[589.0 298.75]`.

The terminal should print:

```
Max Depth: 1.05 m, Center Depth: 0.96 m
```

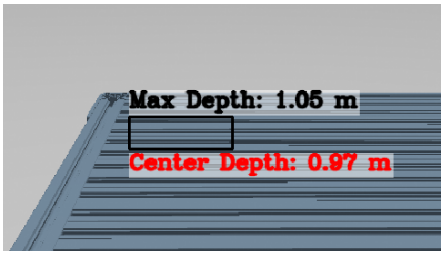


Figure 3: Target output for Exercise 1.

Nothing out of the ordinary yet — as expected, the center pixel appears closer than the pixel at the corners. You'll come to appreciate this exercise once we get to the pose estimation bit. For now, let's move to ArUco tag detection.

> **Question 2.** Switch the encodings for the color and depth OpenCV images. What happens to your subscriber logic? Is `max_box_depth`, if it exists, consistent with the original value you obtained?

3 Writing a Package to Recognize an ArUco Tag using OpenCV

In this second sub-unit, you will learn about a special class of fiducial markers called ArUco tags, understand their uses, and develop a ROS 2 package to recognize an ArUco tag using tools already introduced in Exercise 1.

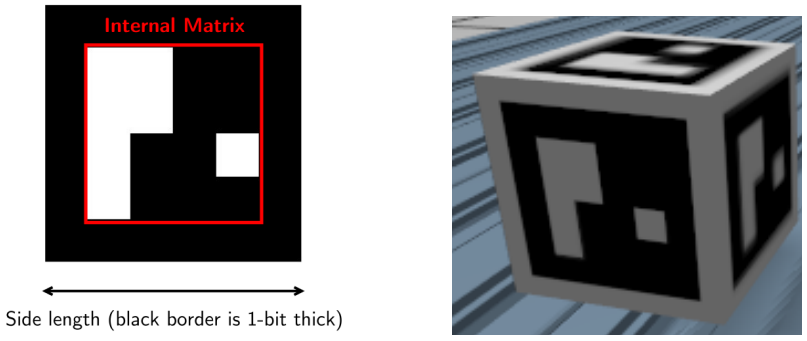
3.1 Fiducial Markers, Explained

Fiducial markers (like April tags or ArUco markers) are grayscale images that encode unique patterns that vision systems today can easily interpret to identify important objects, locations, or other artifacts. Due to the relatively computationally light recognition algorithms available for fiducial marker recognition, these markers have now become a de facto tool for location identification, pose tracking, object tagging, and other robot vision applications.

Each ArUco marker is uniquely identified by an inner binary matrix, which determines its integer identifier (ID), its size (length of each tag side, typically in mm), and a dictionary, which defines the size of the tag's internal matrix. For instance, an AruCo marker with a dictionary of 4x4 and a size of 100 mm will have a binary internal matrix with 16 elements encoding 16 bits. Some helpful resources you can optionally consult include OpenCV's ArUco detection tutorial or one of the many online ArUco tag generators (see here). 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):

```
SUPPORTED_DICTS = {"DICT_4X4_250": cv2.aruco.DICT_4X4_250,
"DICT_4X4_100": cv2.aruco.DICT_4X4_100}
# OpenCV defines only the values in SUPPORTED_DICTS; casting OpenCV
# Dictionary() objects to values in a custom dict is often necessary
```

To successfully detect a tag using OpenCV, a tag's ID must lie within the predefined dictionary. For the rest of this lab, we will use a simulated cube tagged with a pre-generated 4x4 80 mm ArUco



(a) ID 80 ArUco tag.

(b) ID-80 ArUco cube in Gazebo.

Figure 4: The tagged object used in this lab.

marker, depicted in Figure 4. Each face of the cube is tagged with the same marker to avoid dealing with viewpoint issues and ensure that the tag is always visible to the robot's camera.

3.2 ROS 2 ArUco Message Interfaces

ROS 2 provides custom interfaces for working with ArUco tags via the `aruco_markers_msgs` package, namely: `aruco_markers_msgs/msg/Marker` and `aruco_markers_msgs/msg/MarkerArray`. Issue an `interface show` directive with one first message interface to see its contents (truncated for brevity here).

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

We now have the background we need to write an ArUco detection ROS 2 node. Let's spawn an ArUco tagged cube in the simulator to get started.

3.3 Spawn an ArUco-Tagged Cube in Gazebo Harmonic

To spawn objects in a running simulation, a convenient approach is to use the Gazebo spawn service assuming your models are available and stored in a path included within Gazebo's model resources environment variable, `GZ_SIM_RESOURCE_PATH`. In a sourced terminal, run the following (note: individual commands are separated by blank lines):

```
export GZ_SIM_RESOURCE_PATH=$HOME/lab6_ws/src/ur3e_hande_gz_sim/\
ur3e_hande_gz/models:$GZ_SIM_RESOURCE_PATH

gz service -s /world/default/create --reqtype gz.msgs.EntityFactory \
--reptype gz.msgs.Boolean --req "name: 'aruco_box', \
sdf_filename: 'aruco_box', \
pose: {position: {x: 0.02, y: 0.54, z: 0.83}}"
```

You should see the cube show up in Gazebo and RViz2, and the terminal output should read

```
data: true
```

signifying that the cube was spawned correctly.

Exercise 2. AruCo Marker Detector (10 pts.)

This exercise will utilize the

Deliverables:

Image Subscriber & AruCo Detection:

a. Create a class variable `annotated_image` in the `__init__` method, and set it to `None`.

b. Create a class subscriber that reads color images of message type `sensor_msgs.msg.Image.msg` streamed on the bridged Gazebo camera topic `/rgbd_camera/image`. Assign it's callback to a class method we will create in (c).

c. Create a class method `image_cb` that waits on an `Image` message and implements the detection routine using OpenCV. Start with the followin template code, and implement the detection logic using this example. The subscriber should also annotate the input image to highlight the detected AruCo tag

c. Create a publisher that should publish the annotated image

Detection Publisher:

d. Use the following template code: Open the `detect_aruco.py` Python script with the following template code (truncated for conciseness here):

Test the ArUco Detection Node:

Build (with the `--symlink-install` directive) and source your workspace. If there are no build errors, you should be able to start your detection node by running:

```
ros2 run lab_aruco_pose detect_aruco.py
```

This will bring up an OpenCV window (titled Aruco Cube) showing the camera's view and a green box around the detected cube, with the cube's ID in blue color. In the shell session where the node is running, you should see the following command, if a cube with the specified ID has been picked up by the camera:

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

e. Try playing around with different spawn locations for the cube by using the simulator GUI (alternatively, you can delete the cube and rerun the spawn command for each new spawn location in Gazebo, but the first method is simpler). Beyond what `x` value (in meters) does the detector fail to return a result? You can leave the detector node running while you vary the spawn locations in Gazebo Sim.

Question 3. ...

3.4 Visualize a Detection-Annotated Image Stream in RViZ2

Right now, besides the detector node, no other components of the ROS graph are aware that a detection exists. We can confirm this by viewing the image provided on the camera's topic (`/rgbd_camera/image`) on RViZ2. Running the simulator command should have already started up RViZ2 with the image panel. Modify the detection node in the highlighted section to publish the annotated image to a new topic named `/aruco_cube_image`. After the modification, run the node, verify that your publisher logic works by confirming that the new topic is available and changing the topic of the RGB Image panel in RViZ2. You should now see exactly the same OpenCV view within the RViz2 panel.

Question 4. Modify the `desired_encoding` argument of the `imgmsg_to_cv2` CvBridge class method within your detection logic by choosing any **five** supported encodings from the list provided at the message interface source code (see here). What encoding types yield or do not yield a detection? Freeze the cube's position to `x: 0.24, y: 0.542925, z: 0.83` as you vary the image encodings. You will have to rerun the node to see changes.

4 Creating an ArUco-Tag Based Object Pose Action Server

Great! We can now tell if an ID-80 tagged object is within a reasonable fraction of the camera's field of view. However, we'd like to do more than just assert that such an object exists. In particular, for robotic manipulation, our robot will also need to know *where* the cube is, or the cube's **pose**, which is more information than we can provide now. Without this information, our robot can't move its end-effector to the cube never mind picking it up. Let's change that.

4.1 ROS 2 Pose Message Interfaces

Poses are a spatial (three-dimensional) geometric construct that encode both the spatial position ($p = [x, y, z]^\top \in \mathbb{R}^3$) and the spatial orientation ($R \in \mathrm{SO}(3)$) of the coordinate frame attached to a rigid body. If we think of rotations as represented by Euler angles, i.e., ϕ (roll), θ (pitch), and ψ (yaw), then each pose, can be compactly written as a vector, $\xi = [x, y, z, \phi, \theta, \psi]^T \in \mathbb{R}^6$. In ROS 2, the message interface types for relaying pose information are the `geometry_msgs/Pose` and `geometry_msgs/PoseStamped` message interfaces. To get a sense of the content of each message type, issue an `interface show` directive to the above message interfaces. This should show the following output (truncated for conciseness) from where we see that the 'stamped' message just appends time information to the pose (via a `std_msgs/Header` message interface) .

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

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 (monocular, with a single aperture and optical axis perpendicular to the image plane), the coordinates of each observed pixel (u, v) in an image, $I(u, v)$, each correspond to a 3D point

```
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 5: ROS 2 `Pose` and `PoseStamped` message definitions.

X_o, Y_o, Z_o in the camera's optical frame related by the perspective transformation matrix, as in

$$\begin{aligned} X_o &= \frac{(u - c_x) \cdot Z}{f_x} \\ Y_o &= \frac{(v - c_y) \cdot Z}{f_y} \\ Z_o &\equiv Z = D(u, v), \end{aligned} \tag{1}$$

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

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

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
```

In the output that's printed, the `k` field contains the parameters we need.

Question 5. What are the values of c_x, c_y, f_x, and f_y?

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:

$$\begin{bmatrix} R^W_{\text{cube}} & t^W_{\text{cube}} \\ 0_3 & 1 \end{bmatrix} = \begin{bmatrix} R^W_{\text{cam}} & t^W_{\text{cam}} \\ 0_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} R^{\text{cam}}_{\text{cube}} & t^{\text{cam}}_{\text{cube}} \\ 0_3 & 1 \end{bmatrix} \tag{3}$$

Question 6. Try reversing the order of the transform lookup and the rotation application. What happens to your pose estimate? What property of matrices convinces you that the pose estimate must necessarily be different if the order of transformation is reversed?

Exercise 3. AruCo Marker Pose Server (20 pts.)

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. Let's create a custom action interface for our pose action server. Within your workspace's `lab_aruco_pose` package, create an action interface named `GetArucoPose.action` file within the `action` folder. Add the following interface definition to the file:

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

Modify your `package.xml` and `CMakeLists.txt` files to register your custom action interface, build with `--symlink-install`, and source your workspace. Next, create a Python file named `estimate_aruco_pose.py` at your `lab_aruco/scripts` directory, and add the following code to it. If you've built the package right, your IDE's linter will correctly resolve the action import line.

```
...
```

Deliverables:

`GetArucoPose` Action Server:

5 Closing the Perception-to-Action Loop

Great, our robot can now tell that an ArUco-tagged object is in its observable workspace; it also knows where the object is relative to its base frame, bringing us to a point where we can close the perception–action loop at a high level, i.e., use a *static* pose estimate from the marker to plan and execute a motion. Since there are no obstacles in our robot's workspace, we can make do with straightforward trajectory generation and control. This is the subject of the third and final exercise.

5.1 Deriving a Simple ImageFeedback Controller

Question 7. ...

Exercise 4. AruCo Arm Commander (15 pts.)

The ament_python package, aruco_arm_commander in the src folder contains a Python file move_to_aruco.py with skeleton code for a ROS 2 action client that we will finish developing. Complete this exercise by performing the following steps:

Deliverables:

GetArucoPose Action Client:

a. Inside the Node class constructor method, create an action client with action type GetArucoPose and action name get_aruco_pose.

b. Create a class method send_aruco_pose_goal that should handle the sending of an action Goal to the GetArucoPose action server to retrieve the pose of the ArUco tag in its workspace.

GetArucoPose Action Client:

c.

d. Use the following code template:

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
Exercise 1.	20
Exercise 2.	10
Exercise 3.	20
Exercise 4.	15
Total	100