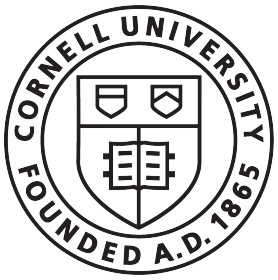

Software Design for the Mollusca Robot

Addy (Jin Hyun) Park
Advisor: Prof. Kirstin Petersen

December 21, 2024

Preface

This document is primarily intended to serve as a documentation for the work I have done for the Mollusca project. The goal is that the person taking over will be able to reference it to understand and build on this work. My main contribution to the project was achieving autonomy for these robots at the lab-scale, which involved building a software framework for the robots.

1 Introduction

The Mollusca project aims to build a collective of snail-like ocean robots intended for microplastic removal from the ocean. The snail robots mimic the Hawaiian apple snails that utilize the undulating motion of their foot to "suck in" floating food particles. To endow the robots with autonomy, I have written a software framework that includes modules for localization (using Lighthouse positioning system), pose estimation using AprilTags, wireless communication, and command handling. The software framework is by no means complete, but my hope that it serves as a good starting point for whoever takes over the project in the future. With that said, this report will be structured into two sections: (1) high-level software architecture and (2) implementation details. In the software design section, I will describe how the software framework is organized without getting into implementation details. The implementation details section will provide the details of the implementation and reasoning behind particular design choices.

2 Software Design

This section details how the software framework is structured.

2.1 Overview

My goal was to build a modular software framework where different modules can 'talk to' one another through topics using a subscriber/publisher model. This offers mainly two advantages:

1. **Modularity**: Modules can easily be replaced or updated without affecting the rest of the system, which simplifies both maintenance and debugging.

2. **Organized data transfer**: By structuring communication into topics, data flows remain organized which will be especially important when scaling to multiple robots.

The on-board computer on the robots is a Raspberry Pi 5 (RPi 5). Because the RPi's have limited computational power, I chose to offload computation-heavy tasks like data visualization to a separate off-board computing unit (i.e. my laptop). Even running light graphics modules like `matplotlib` seems to be slow on the RPi. In the future, I imagine other computationally heavy tasks like motion planning to also be done on an off-board computer and simply send the low-level control signals to the robot.

2.2 Communication Protocol

Communication between the RPi on the robot and the off-board computer is facilitated by an MQTT (Message Queuing Telemetry Transport) broker. The MQTT protocol is a lightweight publish-subscriber messaging protocol. The MQTT broker receives messages from publishing clients and forwards them to the subscribing clients using topics. In this context, the clients are individual modules that need to send or receive messages to and from modules on the other device (e.g. a sensor module on RPi needs to send sensor data to a plotter module on the off-board computer). But before clients are able to send or receive messages on topics, they first need to connect to the MQTT broker. Moreover, since the off-board

computer is more powerful, I chose to run the MQTT broker on the off-board computer. In summary, an MQTT network is made up of three things:

- **Publisher**: A device or application that sends messages to a topic (e.g. Lighthouse module sending position data).
- **Subscriber**: A device or application that receives messages from a topic (e.g. a position controller module receives the robot's position in real-time for feedback control).
- **Broker**: The intermediary that connects publishers and subscribers and manages connections.

2.3 Robot Software

The software framework on the robot has four modules: Lighthouse tracking module, AprilTag pose estimation module, a command handler module, and a communication module. I imagine that list will get longer as the project progresses. Here is a list of modules on the RPi side and what they each do:

- **Lighthouse tracking module**: This module provides localization (i.e. position) for the robot using a Lighthouse positioning system.
- **AprilTag pose estimation module**: This module estimates the poses (i.e. position and orientation) of AprilTags using its forward-facing camera.
- **Command handler module**: This module receives commands from the off-board computer and actuates the robot accordingly.
- **MQTT communication module**: This module serves as the central hub for facilitating data exchange between the RPi and the off-board computer. It aggregates data from other modules on the RPi and publishes it to the appropriate MQTT topics for the off-board computer to receive. Similarly, it subscribes to topics to receive messages from the off-board computer and routes those messages to the relevant modules on the RPi.

2.4 Off-board Computer Software

The off-board computer also has its own set of modules for interacting with the robot. Here is what each of the modules on the off-board computer does:

- **MQTT broker**: As described above, the MQTT broker receives messages from publishing clients and forwards them to the subscribing clients based on topics.
- **MQTT communication module**: Serves the same role as the MQTT communication module on the RPi. It aggregates data from other modules on the off-board computer and publishes it to the appropriate MQTT topics for the RPi to receive. It also subscribes to topics to receive messages from the RPi and routes those messages to the relevant modules on the off-board computer.
- **Lighthouse plotting**: This module receives position data from the Lighthouse tracking module on the RPi and plots them in real time for visualization.

2.5 Overall Architecture

The following is a schematic of how all of the modules interact:

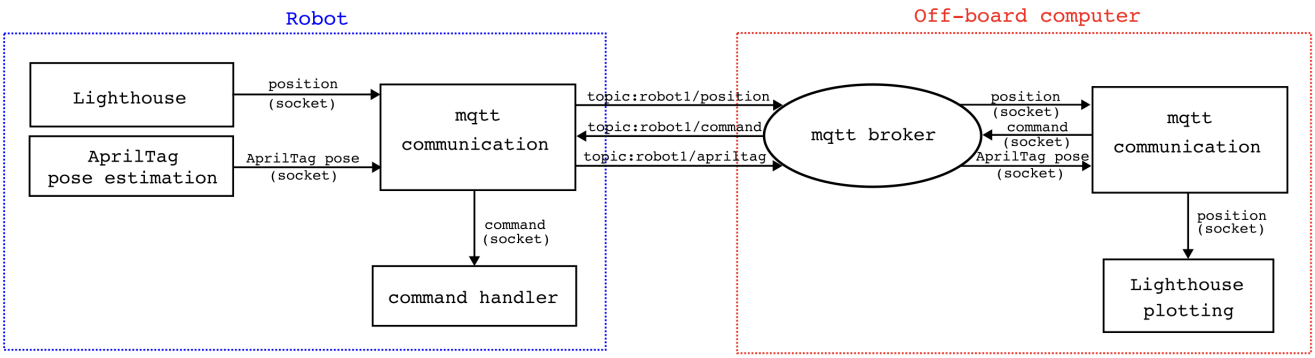

Figure 1: High-level architecture of software framework.

Note that MQTT is only used for communication **across different devices** (by 'device,' I mean the RPi or the off-board computer). Within the same device, different modules communicate using sockets. This means that each pair of modules within a device will need a designated port number to communicate over. A list of designated port numbers can be found in Section A.

Also, in the actual code running on the robot, the command handler code has been written as part of the MQTT communication module. Ideally, I would have separated the command handler into a separate module but did not have the time to do so. If I were to separate it into a separate module, the command handler module would contain two files: (1) `motor_control.py` and (2) `main.py`. I would move all of the code for command handling to a separate `main.py` file in the command handler module like the following:

```python
from IP_receiver import receive_broadcast
import paho.mqtt.client as mqtt
import motor_control

# Command-to-function mapping
command_handler = {
    "move_forward": motor_control.move_forward,
    "move_backward": motor_control.move_backward,
    "turn_left": motor_control.turn_left,
    "turn_right": motor_control.turn_right,
    "stop": motor_control.stop
}

def on_connect(client, userdata, flags, rc):
        """
        Callback function that gets executed upon connecting to MQTT broker.
        """
        print(f"Connected with result code {rc}")
```

```python

def on_message(client, userdata, message):
    """
    Callback function that gets execution upon receving message on subscribed topics.
    """
    command = message.payload.decode()
    print(f"Received command: {command}")

    # Look up and execute the command
    if command in command_handler:
        # For thrusters, check the status flag first
        if command == 'stop' or motor_control.check_motor_status():
            command_handler[command]()
            print(f"Motor command sent: {command}")
        else:
            print("Motor not operable")
    else:
        print(f"Unknown command: {command}")

def main():
    # Robot motor setup
    motor_control.gpio_pin_setup()
    motor_control.initialize_motors()
    print("Motors initialized")

        # Receive ip address of MQTT broker
    mqtt_broker_address = receive_broadcast(port=7000, ack_port=7002) # PORT 7000 is reserved for receving IP address for mqtt
    print(f"Broker address received: {mqtt_broker_address}")

    # Connect to MQTT broker
    mqtt_client = mqtt.Client() # Create client to MQTT broker
    mqtt_client.on_connect = on_connect # Set callback function that will be executed upon successful connection to MQTT broker
    mqtt_client.on_message = on_message # Set callback function that will be executed upon receiving message
    mqtt_client.connect(mqtt_broker_address, 1883, 60) # Connects client to MQTT broker

    # Subscribe to command topic
    mqtt_client.subscribe("robot/control/commands")
    mqtt_client.loop_start() # handle MQTT tasks asynchronously

if __name__ == "__main__":
    main()

```

And the `main.py` script in the MQTT communication module would now only contain code for MQTT communication:

```python
from sensor_data_publisher import SensorDataPublisher
from IP_receiver import receive_broadcast
import paho.mqtt.client as mqtt
import time
import socket
import select

SERVER_IP = '127.0.0.1'

# Each module needs its own port to communicate over
LIGHTHOUSE_PORT = 5006
APRILTAG_PORT = 5007
PORTS = [LIGHTHOUSE_PORT, APRILTAG_PORT]

ROBOT_ID = "robot_1" # ID of this robot

def on_connect(client, userdata, flags, rc):
        """
        Callback function that gets executed upon connecting to MQTT broker.
        """
        print(f"Connected with result code {rc}")

def main():
        # Receive ip address of MQTT broker
    mqtt_broker_address = receive_broadcast(port=7000, ack_port=7002) # PORT 7000 is reserved for receving IP address for mqtt
    print(f"Broker address received: {mqtt_broker_address}")

    # Connect to MQTT broker
    mqtt_client = mqtt.Client() # Create client to MQTT broker
    mqtt_client.on_connect = on_connect # Set callback function that will be executed upon successful connection to MQTT broker
    mqtt_client.on_message = on_message # Set callback function that will be executed upon receiving message
    mqtt_client.connect(mqtt_broker_address, 1883, 60) # Connects client to MQTT broker

    # Create socket object for each port (for communication with other modules)
    sockets = [] # List to store all sockets
    for port in PORTS:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Create a UDP socket
        s.bind((SERVER_IP, port)) # Binds to socket with SERVER_IP and SERVER_PORT
        sockets.append(s) # Append to socket list
```

```
40
41      # Create publisher to publish sensor data
42      sensor_publisher = SensorDataPublisher(mqtt_client, ROBOT_ID) # Createae
        ↪  SensorDataPublisher to publish different sensor data
43
44      mqtt_client.loop_start() # handle MQTT tasks asynchronously
45
46      while True:
47          readable, _, _ = select.select(sockets, [], [])
48
49          # Get message from each module and publish to corresponding topic
50          for sock in readable:
51              message, addr = sock.recvfrom(1024)
52              message = message.decode('utf-8').strip()
53
54              if sock.getsockname()[1] == LIGHTHOUSE_PORT:
55                  pos = message
56                  print(f"Received Lighthouse data: {message}")
57                  sensor_publisher.publish_position_data(pos)
58              if sock.getsockname()[1] == APRILTAG_PORT:
59                  pose = message
60                  print(f"Received AprilTag data: {message}")
61                  sensor_publisher.publish_apriltag_detection_data(pose)
62
63  if __name__ == "__main__":
64      main()
```

In Figure 1, the Lighthouse and AprilTag pose estimation modules are sensor modules – they interface directly with sensors on the robot. These module retrieve sensor readings and relay them to the MQTT communication module, which relays them to the off-board computer. Similarly, the command handler module should interface directly with the actuators on the robot. The commands that the command handler responds to should come from the off-board computer, which is received by the MQTT communication module on the RPi and relayed to the command handler module.

As mentioned at the beginning of this report, I aimed for this framework to be modular. Accordingly, the software framework has been designed such that it can be generalized to any N number of sensor modules:

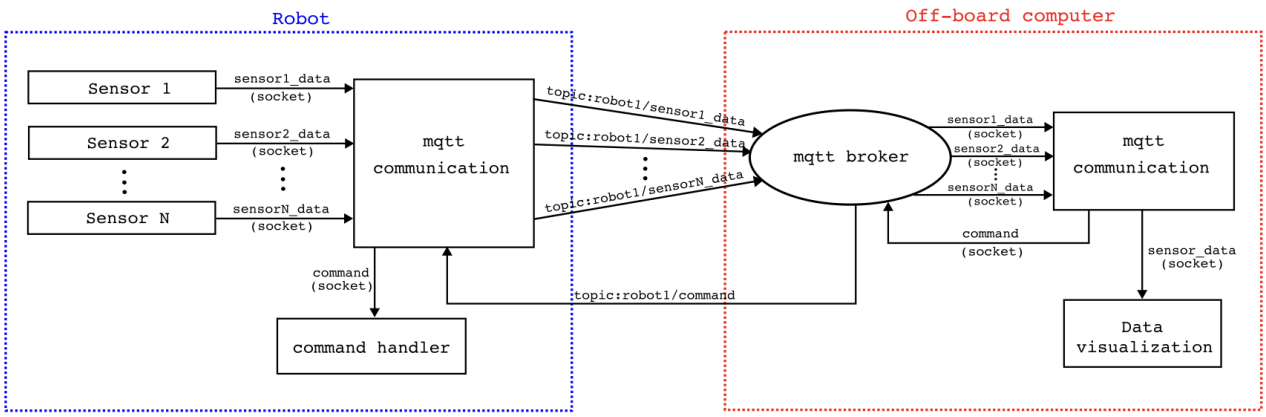

Figure 2: Generalized high-level architecture of software framework.

As you can see above, we can have any number of sensors sending sensor data to the off-board computer through the MQTT communication module. Also, to keep things organized, each sensor will have a designated topic.

To augment the software framework with additional sensor modules, one needs to do the following:

1. Designate a unique port number for the sensor module to communicate with the MQTT communication module.
2. Send the sensor data to the MQTT communication module using the designated port.
3. Add a new publish method to the `SensorDataPublisher` class.
4. Add code to call the publish method in the `main.py` script in the MQTT communication module.

For instance, suppose we have a IMU sensor module and we would like to publish it to a topic called `robot1/imu`. The main script in the IMU sensor module should create a socket and send the data to the MQTT communication module through a designated port number. The main script would look something like the following:

```python
import socket
import time

def get_imu_data():
    """
    Placeholder function that retrieves IMU data.
    """
    ##### Code that interfaces with IMU sensor would go here #####
```

```
    return accel_x, accel_y, accel_z, gyro_x, gyro_y, gyro_z

def main():
    # Define the IP and port of the receiver
    UDP_IP = "127.0.0.1"
    UDP_PORT = 5005

    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    while True:
        # Get IMU data (placeholder)
        accel_x, accel_y, accel_z, gyro_x, gyro_y, gyro_z = get_imu_data()

        # Format the data for sending
        message = f"{accel_x},{accel_y},{accel_z},{gyro_x},{gyro_y},{gyro_z}"

        # Send the data over UDP
        sock.sendto(message.encode(), (UDP_IP, UDP_PORT))
        print(f"Sent: {message}")

        # Sleep for demonstration purposes (adjust as needed)
        time.sleep(1)

if __name__ == "__main__":
    main()
```

In this example script, the designated port number is 5005. '127.0.01' is the loopback address (used by a computer to refer to itself).

To receive this sensor data from the MQTT communication module side, a few lines of code need to be added to the `main.py` script in the MQTT communication module. First, add the port number to the list of port numbers (lines 5 and 6 below):

```
1 ...
2 # Each module needs its own port to communicate over
3 LIGHTHOUSE_PORT = 5006
4 APRILTAG_PORT = 5007
5 IMU_PORT = 5005
6 PORTS = [LIGHTHOUSE_PORT, APRILTAG_PORT, IMU_PORT]
7 ...
```

You do not need to manually create a new socket object to communicate with the newly added IMU sensor module – other parts of this script will automatically do this.

Next, you need to add a new `if` statement in the `while` loop to handle the case where an IMU data is received (lines 18 to 21 below):

```
1  ...
2  while True:
3      readable, _, _ = select.select(sockets, [], [])
4
5      # Get message from each module and publish to corresponding topic
6      for sock in readable:
7          message, addr = sock.recvfrom(1024)
8          message = message.decode('utf-8').strip()
9
10         if sock.getsockname()[1] == LIGHTHOUSE_PORT:
11             pos = message
12             print(f"Received Lighthouse data: {message}")
13             sensor_publisher.publish_position_data(pos)
14         if sock.getsockname()[1] == APRILTAG_PORT:
15             pose = message
16             print(f"Received AprilTag data: {message}")
17             sensor_publisher.publish_apriltag_detection_data(pose)
18         if sock.getsockname()[1] == IMU_PORT:
19             imu_data = message
20             print(f"Received IMU data: {message}")
21             sensor_publisher.publish_imu_data(imu_data)
22 ...
```

This makes sure that, if the message that it received is an IMU reading, it publishes it to the correct topic (the IMU topic) by calling the function `publish_imu_data`.
Lastly, we need to actually define the `publish_imu_data` method inside the `SensorDataPublisher` whose class definition is in `sensor_data_publisher.py` (lines 31 to 37):

```
1  ...
2  class SensorDataPublisher:
3      """
4      This class publishes sensor data (e.g. position, IMU, etc.) to the
5      appropriate topic in the MQTT network.
6
7      This class is meant to be modular to accommodate any type of sensor data.
8      For example, if you wish to send lidar data, define a method publish_lidar_data().
9      The structure of the data is left to the implementer.
10     """
11     def __init__(self, mqtt_client, robot_id):
12         self.mqtt_client = mqtt_client
13         self.robot_id = robot_id
```

```
14
15      def publish_position_data(self, position_data):
16          topic = "robot/sensor_data/position"
17          payload = {
18              "robot_id": self.robot_id,
19              "position": position_data
20          }
21          self.mqtt_client.publish(topic, json.dumps(payload))
22
23      def publish_apriltag_detection_data(self, detection_data):
24          topic = "robot/sensor_data/apriltag"
25          payload = {
26              "robot_id": self.robot_id,
27              "apriltag_data": json.loads(detection_data)
28          }
29          self.mqtt_client.publish(topic, json.dumps(payload))
30
31      def publish_imu_data(self, imu_data):
32          topic = "robot/sensor_data/imu"
33          payload = {
34              "robot_id": self.robot_id,
35              "imu_data": json.loads(imu_data)
36          }
37          self.mqtt_client.publish(topic, json.dumps(payload))
38  ...
```

This framework could even be generalized to work with multiple robots:

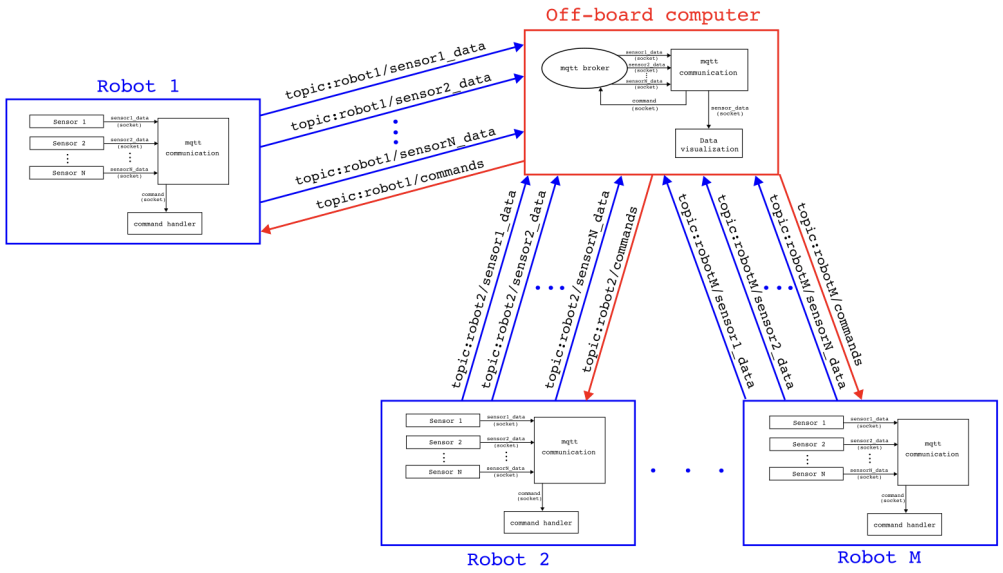

Figure 3: Software framework generalized to M number of robots.

Notice that each robot has a designated namespace which keeps the data streams organized.

3 Implementation Details for Modules on the RPi (Robot) Side

This section delves into the implementation details for each of the modules running on the **robot**.

3.1 Lighthouse Tracking Module

The implementation details for the Lighthouse module can be found in a separate document here.

3.2 AprilTag Pose Estimation Module

The purpose of this module is to detect and estimate the poses of any AprilTags in the image capture by the robot's forward-facing camera. Most of the code came from this repository (only minor changes were made to be compatible with newer versions of dependencies). This module consists of three classes.

3.2.1 Class `Camera`

This class contains methods for initializing the camera, starting the camera's video stream, and capturing images. You can also set the camera parameters such as focal lengths (which should be obtained from calibration) using this class.

3.2.1.1 Method: `__init__`

```
def __init__(self):
                correction_factor = 0.9 # 0.9 seems to work well
                fx, fy, cx, cy = correction_factor * 1467.8745, correction_factor *
                ↪  1447.9533, 263.11, 195.564 # results of calibration
                self.camera_params = (fx, fy, cx, cy)
                # Initialize and start Picamera object
                self.capture = Picamera2()
                self.capture.start()
```

Initializer for the `Camera` class. It creates a `PiCamera2` object (which works with the Arducam cameras as well) and calls the `start` method to start the video stream. `fx`, `fy` are the focal lengths of the camera and `cx`, `cy` are camera principle points. These are results of camera calibration, which can be done by pointing the camera at a printout of a chessboard image and running this script. Because the calibration results were not quite accurate when I performed this calibration, I added a correction factor of 0.9, which gave good results.

3.2.1.2 Method: `read`

```python
def read(self):
    """
    Method for reading/capturing an image from the camera.
    """
    self.img = self.capture.capture_array()
    return self.img
```

This method retrieves an array representing pixel values of an image from the camera.

3.2.1.3 Method: `release`

```python
def release(self):
    """
    Method for stopping the camera.
    """
    self.capture.stop()
```

This method stops the video stream.

3.2.2 Class `TagFinder`

This class contains methods for detecting AprilTags from images captured by the camera and outputting pose of the detected AprilTags. It uses the dt-apriltags library. The main method is the `get_Pose()` method, which does the following:

1. Capture an image using the `Camera` object.
2. Call the `dt_detector.detect()` method (`dt_detector` is a `Detector` object from the dt-apriltag library) to process the captured image and look for any AprilTags.
3. Extract detection results and store into a dictionary.

3.2.2.1 Method: `__init__`

```python
def __init__(self,tag_size):
        self.tag_size = tag_size
        self.camera_obj = camera.Camera()
        self.detector = apriltag.DetectorOptions(families=TAG_FAMILY) # detector object
        ↪  from apriltag library
        self.dt_detector = dt_apriltags.Detector(families=TAG_FAMILY,
                    nthreads=1,
                    quad_decimate=1.0,
                    quad_sigma=0.0,
```

```
                    refine_edges=1,
                    decode_sharpening=0.25,
                    debug=0) # detector from dt_apriltags library
```

Initializer for the `TagFinder` class. As its attributes, it has a `Camera` object and a `Detector` object from the dt_apriltags library. The `Detector` object contains all of the methods for detecting AprilTags in an image and calculating its position and orientation relative to the camera frame. Note that you should change the global variable `TAG_FAMILY` (defined in tag_finder.py) corresponding to the tag family you are using. Also, `tag_size` (which is the physical size of the AprilTags you are using) should be given as an argument when initializing a `TagFinder` object.

3.2.2.2 Method: `capture_Camera`

This method simply captures an image using its `Camera` object.

3.2.2.3 Method: `release_Camera`

This method stops the video stream.

3.2.2.4 Method: `get_Pose`

```
def get_Pose(self):
    """
    Method for getting (X,Y,Z) translations and pitch, yaw, roll rotations of detected
    ↪  apriltag.
    Stores them in self.Poses.
    """
    self.img = self.capture_Camera() # capture image
    self.gray = cv2.cvtColor(self.img, cv2.COLOR_BGR2GRAY)
    self.dt_results = self.dt_detector.detect(self.gray, True,
    ↪  self.camera_obj.camera_params, self.tag_size) # detect apriltag
    self.Yaw, self.Pitch, self.Roll, self.Translation = [], [], [], []
    self.Poses = []
    # iterate over all detected tags
    for result in self.dt_results:
        pose = {} # initialize empty dictionary
        radian, degree = self.get_Euler(result.pose_R)
        X, Y, Z = result.pose_t * 1000 # convert m to mm
        position = (X,Y,Z)

        pose['translation'] = position
        pose['degree'] = degree
        pose['radian'] = radian
        pose['tag_id'] = result.tag_id
```

```
        self.Poses.append(pose)

    return (len(self.dt_results) > 0) # return false if no tag detected
```

This method captures an image from the camera (i.e. retrieves an array containing pixel values seen by the camera) and looks for any AprilTags in the captured image. It stores any detection (a detection is given as a translation vector and a rotation matrix) it makes in its attribute `dt_results`. It then converts the translation vector into mm units and converts the rotation matrix to Euler angles. It stores all of this detection data into a dictionary `Poses` which it keeps as an attribute. It does this for all AprilTags that it detects in the captured image.

3.2.3 Class `Drawing`

This class contains methods for drawing the pose of the detected AprilTag on top of the captured image. A drawing would look something like this:

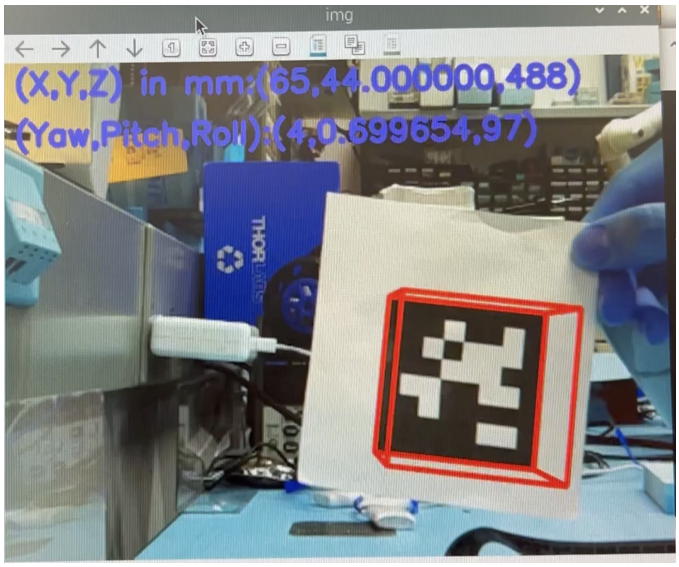

Figure 4: Detection results drawn on top of captured image.

3.2.3.1 Method: `__init__`

Initializer for the `Drawing` class. It only contains one attribute, `tag_obj`, which is a `TagFinder` object (which contains all of the detection results that we want to draw onto the image).

3.2.3.2 Method: `annotate_Image`

Draws text annotation onto the image. The annotation prints out the translation vector and the orientation on the top left corner of the image as seen in Figure 4.

3.2.3.3 Method: `draw_Cube`

Draws a cube on the detected AprilTag (as seen in 4) which is useful for visualizing and sanity checking the estimated orientation of the AprilTag.

3.2.3.4 Method: `draw_Bounding_box`

Because `draw_Cube` draws a box around the AprilTag as part of the cube, this method is slightly redundant.

3.2.4 Script `main.py`

In the main script, there are four global variables defined:

```
...
# Tag information
TAG_SIZE = 0.056

# IP addr & PORT for interfacing with MQTT locally
SERVER_IP = '127.0.0.1'
SERVER_PORT = 5007

# Socket for sending pose data to other modules
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
...
```

The variable `TAG_SIZE` is the physical size of the AprilTag in meters. You should physically measure the dimensions of the AprilTag and store it in this variable. The variable `SERVER_IP` is given the loopback address since this module will only be communicating with modules in the same machine (MQTT communication module). The variable `SERVER_PORT` stores the designated/reserved port number for communication between this module and the MQTT communication module. Lastly, the variable `s` stores the socket object needed to communicate (via UDP) with the MQTT communication module.

There are also four functions defined, which are described below.

3.2.4.1 Function: `estimate_and_draw`

As the name suggests, this function estimates the pose of the AprilTag (using the `TagFinder` class), annotates, and draws a cube around the detected AprilTag (using the `Drawing` class).

3.2.4.2 Function: `only_estimate`

This function only estimates the pose of the AprilTag without annotating and drawing a cube around the detected AprilTag.

3.2.4.3 Function: `send_pose`

This function takes as argument the estimated pose of the AprilTag(s) and sends it to the MQTT communication module using a socket object (which is declared globally in the main script).

3.2.4.4 Function: `main`

This is the main function makes use of the other three functions to detect and send AprilTags to the MQTT communication module.

```python
def main():
    tagfinder_obj = tag_finder.TagFinder(TAG_SIZE)
    draw_obj = drawing.Draw(tagfinder_obj)

    while True:
        # pose = estimate_and_draw(tagfinder_obj, draw_obj)
        pose = only_estimate(tagfinder_obj)
        if pose != 0: # if detected
            send_pose(pose)
            pass
```

3.3 MQTT communication Module

As described previously, the purpose of this module is to serve as the central hub for facilitating data exchange between the RPi and the off-board computer. It aggregates data from sensor modules on the RPi and publishes it to the appropriate MQTT topics for the off-board computer to receive. Similarly, it subscribes to topics to receive messages from the off-board computer and routes those messages to the command handler module on the RPi.

3.3.1 Function: `receive_broadcast`

In order to communicate with the off-board computer, the robot needs to know the IP address of the off-board computer. Although the lab WiFi assigns static IP addresses, I decided to write functions to broadcast and receive IP addresses in case that one needs to work with a network that assigns IP addresses dynamically. When the off-board computer runs its software for communicating with the robot, it will always start by broadcasting its IP address. The `receive_broadcast` function listens for this broadcast and when it receives the IP address of the off-board computer, it sends an ACK (acknowledgment) message back to the off-board computer.

3.3.2 Class: `SensorDataPublisher`

This class contains methods for publishing sensor data to their corresponding topics:

```python
class SensorDataPublisher:
    """
    This class publishes sensor data (e.g. position, IMU, etc.) to the
    appropriate topic in the MQTT network.

    This class is meant to be modular to accomodate any type of sensor data.
    For example, if you wish to send lidar data, define a method publish_lidar_data().
    The strucutre of the data is left to the implementer.
    """
    def __init__(self, mqtt_client, robot_id):
        self.mqtt_client = mqtt_client
```

```
        self.robot_id = robot_id

    def publish_position_data(self, position_data):
        topic = "robot/sensor_data/position"
        payload = {
            "robot_id": self.robot_id,
            "position": position_data
        }
        self.mqtt_client.publish(topic, json.dumps(payload))

    def publish_apriltag_detection_data(self, detection_data):
        topic = "robot/sensor_data/apriltag"
        payload = {
            "robot_id": self.robot_id,
            "apriltag_data": json.loads(detection_data)
        }
        self.mqtt_client.publish(topic, json.dumps(payload))
```

All of the functions follow the same format. The function takes in as argument the data you want to publish. Inside the function, you need to name the topic that you want to publish the data to. Then, the payload should include the robot's ID number (useful for when you want to extend this framework to multiple robots) and the data you want to send (as a json string). Then, you publish the payload to the topic.

3.3.3 Class: `motor_control.py`

This file contains functions that send PWM signals to the robot's motors in order to actuate the robot. This includes functions for moving the robot forward, backward, turning, and stopping.

3.3.4 Script: `main.py`

This is the main script for the MQTT communication module. There are six global variables defined:

```
...
SERVER_IP = '127.0.0.1'

# Each module needs its own port to communicate over
LIGHTHOUSE_PORT = 5006
APRILTAG_PORT = 5007
PORTS = [LIGHTHOUSE_PORT, APRILTAG_PORT]

ROBOT_ID = "robot_1" # ID of this robot

# Command-to-function mapping
command_handler = {
```

```
    "move_forward": motor_control.move_forward,
    "move_backward": motor_control.move_backward,
    "turn_left": motor_control.turn_left,
    "turn_right": motor_control.turn_right,
    "stop": motor_control.stop
}
...
```

`SERVER_IP` is the loopback address for communicating with other modules on the robot. `LIGHTHOUSE_PORT` is the port number over which this module communicates with the Lighthouse module. Similarly, `APRILTAG_PORT` is the port number over which this module communicates with the AprilTag pose estimation module. `PORTS` puts these port numbers into a list that can be looped over. As you add more modules, this list should get longer.

`command_handler` is a dictionary that maps a command message it receives to a function. For example, if the robot receives the `"move_forward"` command from the off-board computer, it calls the `motor_control.move_forward` function.

3.3.4.1 Function: `on_connect`

This is the callback function that is automatically called when MQTT client successfully connects to the MQTT broker.

3.3.4.2 Function: `on_message`

This is the callback function that is automatically called whenever new data arrives on a subscribed topic.

```
def on_message(client, userdata, message):
    """
    Callback function that gets execution upon receving message on subscribed topics.
    """
    command = message.payload.decode()
    print(f"Received command: {command}")

    # Look up and execute the command
    if command in command_handler:
        # For thrusters, check the status flag first
        if command == 'stop' or motor_control.check_motor_status():
            command_handler[command]()
            print(f"Motor command sent: {command}")
        else:
            print("Motor not operable")
    else:
        print(f"Unknown command: {command}")
```

It first decodes the message then looks up the command in the `command_handler` dictionary. If the

command indeed matches a command in the `command_handler`, then it calls the function corresponding to that command.

3.3.4.3 Function: `main`

This is the main function that calls all of the other functions.

```
def main():
    # Robot motor setup
    motor_control.gpio_pin_setup()
    motor_control.initialize_motors()
    print("Motors initialized")

        # Receive ip address of MQTT broker
    mqtt_broker_address = receive_broadcast(port=7000, ack_port=7002) # PORT 7000 is reserved for receving IP address for mqtt
    print(f"Broker address received: {mqtt_broker_address}")

    # Connect to MQTT broker
    mqtt_client = mqtt.Client() # Create client to MQTT broker
    mqtt_client.on_connect = on_connect # Set callback function that will be executed upon successful connection to MQTT broker
    mqtt_client.on_message = on_message # Set callback function that will be executed upon receiving message
    mqtt_client.connect(mqtt_broker_address, 1883, 60) # Connects client to MQTT broker

    # Create socket object for each port (for communication with other modules)
    sockets = [] # List to store all sockets
    for port in PORTS:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Create a UDP socket
        s.bind((SERVER_IP, port)) # Binds to socket with SERVER_IP and SERVER_PORT
        sockets.append(s) # Append to socket list

    # Create publisher to publish sensor data
    sensor_publisher = SensorDataPublisher(mqtt_client, ROBOT_ID) # Create SensorDataPublisher to publish different sensor data

    # Subscribe to command topic
    mqtt_client.subscribe("robot/control/commands")

    mqtt_client.loop_start() # handle MQTT tasks asynchronously

    while True:
        readable, _, _ = select.select(sockets, [], [])

        # Get message from each module and publish to corresponding topic
```

```
36          for sock in readable:
37              message, addr = sock.recvfrom(1024)
38              message = message.decode('utf-8').strip()
39
40              if sock.getsockname()[1] == LIGHTHOUSE_PORT:
41                  pos = message
42                  print(f"Received Lighthouse data: {message}")
43                  sensor_publisher.publish_position_data(pos)
44              if sock.getsockname()[1] == APRILTAG_PORT:
45                  pose = message
46                  print(f"Received AprilTag data: {message}")
47                  sensor_publisher.publish_apriltag_detection_data(pose)
```

First, there is initialization code that initializes the GPIO pin numbers and starts PWM mode for the pins (lines 2-5 above). Then, to connect to the off-board computer, it listens for the IP address broadcasted by the off-board computer (lines 7-9).

Then in lines 11-15, it creates an MQTT client (i.e. a device/application that connects to an MQTT broker), sets two callback functions to `on_connect` and `on_message` (former to be called upon connection and latter to be called whenever new data arrives on a subscribed topic) defined above, and finally connects to the MQTT broker using the IP address it received using `receive_broadcast`.

Next, in lines 17-22, it creates a socket for each of the modules that it needs to communicate with by looping over the list of ports. Then in line 25, it creates a `SensorDataPublisher` object which contains the methods for publishing data to MQTT topics. Line 28 subscribes to the topic where command messages are published to. In the future, if this framework is still used, you could assign each robot with its own namespace (e.g. for robot 3, `"robot3/control/commands"`). `loop_start` in line 30 allows the main program to continue running without blocking on the MQTT network loop. Without this, the main program would block while handling MQTT communication.

Lastly, lines 32-47 handles messages that the MQTT communication module receives from the sensor modules and publishes them to the correct topics using the `SensorDataPublisher` object created in line 25. For example, if this module receives a position data from the Lighthouse module, it makes sure that it gets correctly published to the `"robot/sensor_data/position"` and not the other topics.

As previously mentioned, it would have been ideal to separate the command handling code to a separate module for a more modular design.

4 Implementation Details for Modules on the Off-board Computer Side

This section discusses the implementation details for each of the modules running on the **off-board computer**. There are two modules running on the off-board computer: (1) MQTT communication module

and (2) Lighthouse plotting module.

4.1 MQTT communication Module

The purpose of this module is to serve as the central hub for facilitating data exchange between the off-board computer and the RPi. It aggregates data from modules on the off-board computer and publishes it to the appropriate MQTT topics for the RPi to receive. Similarly, it subscribes to topics to receive messages from the RPi and routes those messages to the appropriate modules on the off-board computer.

4.1.1 Class: `IPBroadcaster`

This class contains methods for broadcasting the IP address of the off-board computer.

4.1.1.1 Function: `__init__`

Initializer for this class.

```
def __init__(self, port, ack_port):
    self.port = port
    self.ack_port = ack_port
    self.broadcast_ip_address = self.get_broadcast_addr()
    self.ip_address = self.get_ip_addr()
    self.ack_received = False
```

There are two ports being used; `port` is the port number over which the IP address will be sent. `ack_port`, on the other hand, is the port number over which the ACK message from the RPi will be received.

There are also two IP addresses being used; `broadcast_ip_address` is the broadcast address of the network that the robot and the off-board computer are connected to. Data should be sent to this address in order to broadcast it to every device connected to this network. `ip_address` is the IP address of the off-board computer.

`ack_received` is a flag that indicates if the ACK message from the RPi has been received or not. An ACK message is a brief communication indicating that the receiver has successfully received and understood the transmitted data.

4.1.1.2 Function: `get_broadcast_addr`

This method fetches the broadcast address of the network.

4.1.1.3 Function: `get_ip_addr`

This method fetches the IP address of the off-board computer.

4.1.1.4 Function: `listen_for_ack`

This method waits until an ACK message is received from the RPi.

```python
def listen_for_ack(self):
    """
    Listen for an acknowledgment (ACK) from the RPi.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', self.ack_port))  # Bind to the ACK port
    sock.settimeout(1)  # Timeout after 1 second if no ACK received

    try:
        data, addr = sock.recvfrom(1024)
        if data.decode('utf-8') == "ACK":
            print(f"ACK received from {addr}")
            self.ack_received = True
    except socket.timeout:
        # No ACK received, continue broadcasting
        pass
    finally:
        sock.close()
```

It creates a socket with port number `ack_port` to receive the ACK message from the RPi. By calling `sock.bind(('', self.ack_port))`, you allow the socket to receive data from any device on the network (indicated by the empty string `''`) and listens for incoming data on the specified `ack_port`. Note that the function call `sock.recvfrom(1024)` is blocking and so the program will wait indefinitely until the ACK message is received. When the ACK message is received, the flag `ack_received` is set to `True`.

4.1.1.5 Function: `broadcast_ip`

This method broadcasts the IP address of the off-board computer to all devices on the network and listens for an ACK message from the RPi (one of the challenges with extending this to multiple robots will be making sure **all** robots received the broadcast).

```python
def broadcast_ip(self):
    """
    Broadcasts the IP address over the network until it's received by an RPi and
    ↪  acknowledged.
    """
    if not self.ip_address:
        print("No IP address found.")
        return

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

    print(f"Broadcasting IP {self.ip_address} on PORT {self.port}, awaiting ACK...")

```

```
    while not self.ack_received:
        try:
            # Broadcast the IP address to the network
            sock.sendto(self.ip_address.encode('utf-8'), (self.broadcast_ip_address,
            ↪  self.port))
            # Listen for ACK
            self.listen_for_ack()
            except Exception as e:
                print(f"Error broadcasting IP: {e}")
                break
    sock.close()
    if self.ack_received:
        print("ACK received, stopping broadcast.")
```

As mentioned above, the function call `listen_for_ack` is blocking. When the ACK message is received, it prints that ACK was received and broadcasting is stopped.

4.1.2 Script: `main.py`

This is the main script for the MQTT communication module. Just like the MQTT communication module on the RPi, it contains a callback function for when the MQTT client connects to the MQTT network and a callback function for when new data arrives on a subscribed topic. In addition, there are functionalities for mapping user key presses to commands. For example, if you press the up arrow key, it publishes the `"move forward"` command to the command topic.

4.1.2.1 Mapping Key Presses to Commands

In order to enable the use of arrow keys to move the robot, the `pynput` library was used. A dictionary was created to map each arrow key to a command message (string):

```
# Command mapping for key presses
command_map = {
    keyboard.Key.up: "move_forward",
    keyboard.Key.down: "move_backward",
    keyboard.Key.left: "turn_left",
    keyboard.Key.right: "turn_right",
    keyboard.Key.space: "stop"
}
```

The following segment of code set up a keyboard listener that waits for key presses and calls the function `on_press` each time a key is pressed. The `listener.join()` call blocks the program until the listener is stopped:

```
# Start listening for keypresses
with keyboard.Listener(on_press=on_press) as listener:
    print("Listening for keypresses. Press ESC to quit.")
    listener.join()
```

Next, the function `on_press` is defined:

```
def on_press(key):
    """
    Function to publish commands upon key press.
    """
    if key in command_map:
        command = command_map[key]
        print(f"Sending command: {command}")
        client.publish("robot/control/commands", command)
    elif key == keyboard.Key.esc:  # Exit on ESC key
        print("Exiting...")
        return False
```

This function gets called every time a key is pressed. If the pressed key matches one of the keys defined in the `command_map` dictionary, then it publishes the command corresponding to that key press to the `robot/control/commands` topic, which the robot is subscribed to.

4.1.2.2 Broadcasting IP Address

The following segment of code uses the `IPBroadcaster` class to broadcast the off-board computer's IP address to all of the RPi's connected to the network.

```
# Broadcast IP address of this device
ipb = IPBroadcaster(port=7000, ack_port=7002) # PORT 7000 is reserved for broadcasting IP
↪  for mqtt
ipb.broadcast_ip()
```

4.1.2.3 Connecting to MQTT Broker

The following segment of code connects this module to the MQTT broker (which is also running on the off-board computer):

```
# Broadcast IP address of this device
ipb = IPBroadcaster(port=7000, ack_port=7002) # PORT 7000 is reserved for broadcasting IP
↪  for mqtt
ipb.broadcast_ip()

# Set up MQTT client
client = mqtt.Client()

```

```
# Set callback function for receiving messages
client.on_message = on_message

# Set callback function for when connected to MQTT broker
client.on_connect = on_connect

# Connect to the MQTT broker running on the central PC (or localhost if broker is on the
↪ PC)
broker_ip = "localhost"  # or use the local IP of the PC if accessed from another machine
client.connect(broker_ip, 1883, 60)
```

Note that the loopback address (`"localhost"`) is used as the MQTT broker address since the broker is running on the same machine.

4.1.2.4 Receiving Data from RPi Modules

In order to receive sensor data from the robot's sensor modules, this module subscribes to the `"robot/sensor_data/position"` and `"robot/sensor_data/apriltag"` topics. Ideally, the MQTT communication module should forward this data to the other modules such as the Lighthouse plotting module. However, due to lack of time, I decided to send the position data straight from the Lighthouse module on the RPi to the Lighthouse plotting module on the off-board computer using sockets. The Lighthouse plotting module will be discussed next.

4.2 Lighthouse Plotting Module

This module receives position data from the Lighthouse module on the robot and plots it in real time using the `matplotlib` library. As mentioned above, it communicates directly with the robot's Lighthouse module rather than communicating through the MQTT communication module. As a result, this module needs to broadcast its IP address to the robot's Lighthouse module and hence it also uses the `IPBroadcaster` class. Below are the functions used to plot the position data in real time. If it works successfully, the plotting should look like this.

4.2.1 Functions: `reconstruct_matrix` and `reconstruct_vector`

All data (vectors, matrices) are sent from the robot as strings. In order to plot this data, it needs to be converted into `numpy` array objects. These two functions do exactly that.

4.2.2 Functions: `plot_global_frame` and `plot_bs_pose`

These functions plot the global reference frame and the base station reference frames.

4.2.3 Functions: `update_point`

This function is called iteratively every time a new position data is received from the robot and updates the plotted point.

5 Other Notes

I have attempted to use one of the RPi's (the one with username 'mollusca2') as a WiFi router that the other RPi's and the off-board computer can communicate through. This has successfully worked many times but doesn't work 100% of time time. If a RPi (one that isn't hosting the network) fails to connect in time to the mollusca2 network, it falls back to connecting to the lab WiFi. A disadvantage in using one of the RPi's as a network access point is that it introduces a single point of failure in the communication network.

Bash script files have been written to run all of the modules at once (on both the RPi and the off-board computer). Moreover, a systemd service has been set up so that the bash script on the RPi's run automatically on bootup using this guide.

A Appendix

The following is a list of ports designated for different modules:

- Port 5006: Used for communication between Lighthouse module on RPi and MQTT module on RPi
- Port 5007: Used for communication between AprilTag module on RPi and MQTT module on RPi
- Port 6000: Used for communication between Lighthouse module on RPi and Lighthouse plotter on off-board computer
- Port 7000: Used for broadcasting IP address from MQTT module on off-board computer to MQTT module on RPi.
- Port 7001: Used for broadcasting IP address from Lighthouse plotter on off-board computer to Lighthouse module on RPi.
- Port 7002: ACK port between MQTT module on RPi and MQTT module on off-board computer.
- Port 7003: ACK port between for Lighthouse module on RPi and Lighthouse plotting module on off-board computer.
- Port 1883: Default, non-encrypted port number used for MQTT connections between the MQTT client and the broker. Multiple clients can simultaneously connect to the same MQTT broker via port 1883, as long as each client has a unique client ID.