Final Project

Python Applications for Robotics

July 23, 2021

Instructors:
Z. Kootbally

Students:
Gokul Hari

Nicholas Novak

Abijitha Nadagouda

Group:
group number 3

Semester:
Summer 2021

Course code:
ENPM809E

Contents

1 Introduction

The idea of the project is to leverage the technique of Visual Servoing, which guides two robots to traverse the given waypoints in a configured gazebo world. Visual Servoing is a technique that uses feedback information from the vision sensor to regulate and track a robot's motion. In this project, We have used two turtlebot3 waffle models. These models are equipped with multiple onboard sensors and a depth camera capable of generating RGB images. Two robots are assigned different roles i.e., a leader, and a follower. A leader robot has an attached Aruco marker tag and is used by the follower to determine waypoints.

Two robots have their defined purposes. Reading the YAML file containing the pose of the next desired location is the task of the leader robot. To travel to one of those locations, the leader uses the $move_base$ function. Besides, the follower robot follows the leader robot and is responsible for detecting the unique Aruco marker attached to the leader robot to compute the location of the tag in the map. Once the tag is detected, the follower uses the $move_base$ function to travel to the determined location. In the event that the Aruco tag is not picked up by the follower robot, it automatically rotates 360° in order to find the tag. As a result, the follower robot will follow the leader robot and be within 0.5m distance from the leader robot.

In addition to navigation, the leader robot must update the parameters that store the next location points. Whenever the robot is unable to find the tag after rotation, it will use this data.

2 Approach

2.1 Leader Node:

First, we have to generate a map of the world to successfully navigate. We used Gmapping to accomplish this. Two files are generated after saving the map: one in a ".yaml" format, the other in a ".pgm" format, where a file with the extension ".pgm" contains occupancy data of the map, and a file with the extension ".yaml" contain metadata.

Teleoperation was performed and ROS service was invoked to save the robot locations in a YAML file. Following, the leader robot must be tagged with an ArUco marker tag after both follower and leader robots are generated in the world, which can be read using the 2D camera. Marker attachment to the leader robot begins with initializing the service clients through node *"spawn_marker_on_robot"* with *"rospy.ServiceProxy"*. Following that, the robot and the marker will be given model names. For our example, we assigned *"leader"* to the robot's name, and *"leader_marker_front"*, and *"leader_marker_back"* to the marker. We calculated the position and orientation of the marker pose based on the current pose of the robot, and using the function *"quaternion_from_euler*, orientation is calculated.

Specifically, the project entails making a leader go to a specific room, and getting the follower to follow. A YAML file is used to store the Leader robot's locations. Each entry in the YAML file is read and stored in room_keys list of dictionary, where key is a room name and each room consists of position and orientation frame coordinates.The position data is read from this list and is published on the leader move_base. Following that, we will provide a transform between the Aruco marker and camera frame in order to allow the follower robot to follow the Aruco marker. Waypoints must be filled in the map frame in order to instruct the follower robot where to go.

Algorithm 1 Leader

```
procedure READ_WAYPOINTS(waypoints_path)
    procedure READ_YAML(waypoints_path)
        $waypoints\_path \leftarrow loadwaypointspath$
        try
        $waypoints \leftarrow readwaypointsfromthefile$
        except
        $stdout \leftarrow exception$
        return waypoints
    end procedure
    $pose \leftarrow extractpositioncoordinates$
    $frame\_orientation \leftarrow extractframecoordinates$
    $pose.extend(frame\_orientations)$
    $output \leftarrow positioncoordinates$
    return output
end procedure

procedure LEADER_CLIENT(coordinates)
    $client \leftarrow actionlib.simpleactionclient(move\_base, MoveBaseAtion)$
    client.wait_for_server()
    $goal \leftarrow MoveBaseGoal()$
    $goal.target\_position \leftarrow fromcooardinates$
    $client.send\_goal(goal)$
    $wait \leftarrow client.wait_for_result()$

    if $not\ wait$ then
    $stdout \leftarrow Actionservernotavailable!$
    $stdout\_shutdown \leftarrow Actionservernotavailable!$
    else
    $returnclient.get\_result()$
end procedure

procedure MAIN
    $rospy.init\_node() \leftarrow initializealeadernode$
    $locationsreadwaypointsfromwaypoints\_path$
    try

    for $location, coordinates.$in $locations$ do
        if coordinates is not None then
        $result \leftarrow leader\_client(coordinates)$
        if result then
        $stdout \leftarrow locationreached$
        except
        $stdout \leftarrow exception$

```

refer 2.2 - example to use references

2.2 Fiducial Broadcaster Node:

The fiducial broadcaster will create a subscriber that will listen to the $/map$ and $fiducial\_transforms$ topics. $fiducial\_transforms$ recieves data from the $aruco\_detect$ node, which reads the follower's camera data.

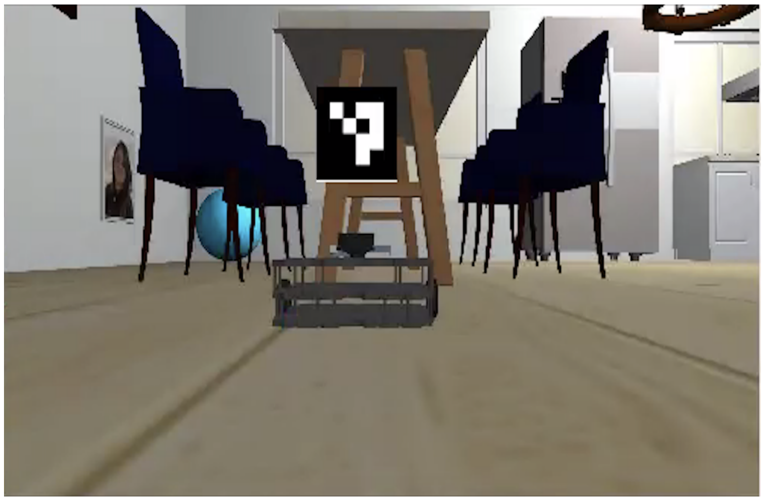


Figure 1: The aruco marker as seen from the follower's camera

Then, the transformation between the map and aruco marker is obtained by determining the transformations from **map to robot**, **robot to camera**, and finally from **camera to marker**. To obtain the camera to marker transformation, the fiducial transforms that are recieved contain the relative pose of the marker to the follower camera frame. To utilize these transforms, a tf broadcaster is used to connect these transforms to the tf tree. Then, a tf transform can be used to get the pose of the marker with respect to the map frame.

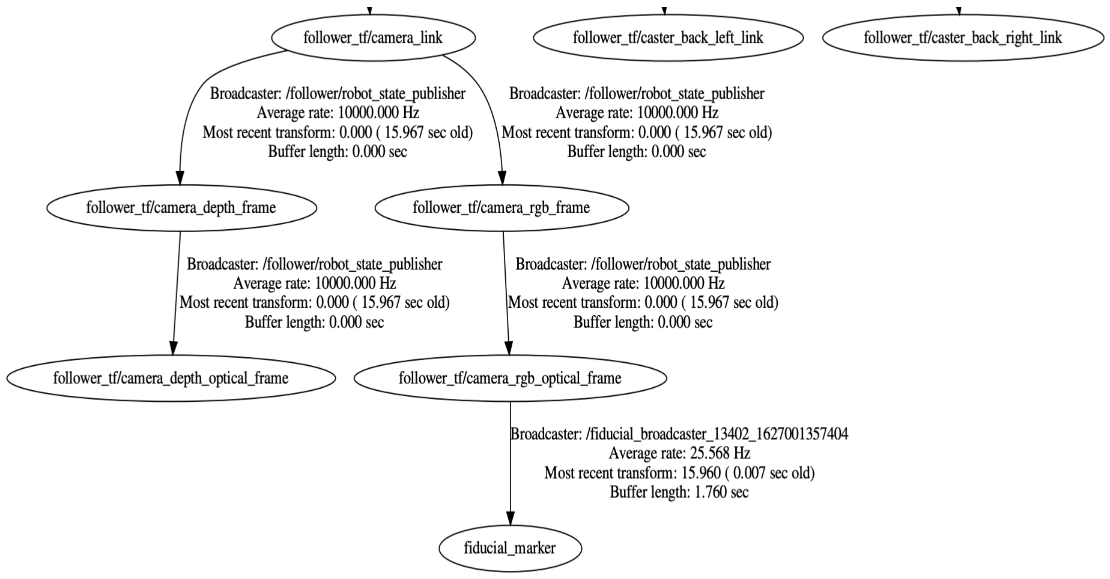

follower_tf/camera_link
follower_tf/caster_back_left_link
follower_tf/caster_back_right_link
Broadcaster: /follower/robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (15.967 sec old)
Buffer length: 0.000 sec
Broadcaster: /follower/robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (15.967 sec old)
Buffer length: 0.000 sec
follower_tf/camera_depth_frame
follower_tf/camera_rgb_frame
Broadcaster: /follower/robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (15.967 sec old)
Buffer length: 0.000 sec
Broadcaster: /follower/robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (15.967 sec old)
Buffer length: 0.000 sec
follower_tf/camera_depth_optical_frame
follower_tf/camera_rgb_optical_frame
Broadcaster: /fiducial_broadcaster_13402_1627001357404
Average rate: 25.568 Hz
Most recent transform: 15.960 (0.007 sec old)
Buffer length: 1.760 sec
fiducial_marker

Figure 2: The fiducial marker is successfully added to the tf tree

Algorithm 2 Broadcaster

```
procedure CALLBACK($msg$)
    $broadcaster \leftarrow Setup\ a\ broadcaster$
    $fiducial\_frame \leftarrow Transforms\ object\ for\ broadcasting$
    Set timestamp, parent, and child frame for $fiducial\_frame$
    for $m$ in $msg.transforms$ do
        Read the message pose:
        $trans \leftarrow translation$
        $rot \leftarrow rotation$
        Read pose values and store them in $fiducial\_frame$.
        Broadcast this object.
    end for
end procedure
procedure LISTEN_LOOKUP_PRINT
    $listener \leftarrow Setup\ a\ listener$
    Wait for three seconds.
    Set sleep rate.
    while rospy is running do
        From listener:
        $(trans, rot) \leftarrow transform\ from\ camera\ to\ map$
        Notify user if nothing can be listened to.
    end while
end procedure
procedure IF( $\_\_name\_\_ == \_\_main\_\_$)
    $fiducial\_broadcaster \leftarrow create\ new\ node$
    Create Subscriber to $/fiducial\_transforms$
    Include callback in Subscriber with input $FiducialTransformArray$
end procedure
    =0
```

refer 2.3 - example to use references

2.3 Follower Node:

The task of the follower is to detect the fiducial marker attached to the leader bot and follow it as closely as possible. There are three actions to be performed by the follower bot and this is programmed through the follower node.

- Detect the aruco marker and execute the $move_base$ action client to move towards the aruco marker
- If the aruco marker is not detected the follower has to rotate in-place 360 degrees to locate the marker
- If the follower is unable of detecting the marker in the previous steps, it has to read the leader's current goal published in the parameter server and set this as a new goal, while also actively searching for the fiducial marker during it's journey towards the goal.

First, we need to *listen* to the marker's pose data broadcasted by the broadcaster node that was discussed in 2.2. We built a *find_fiducial* procedure that uses a transforms-*listener* from the *tf* package, to obtain the pose of the detected fiducial marker in this node. This pose is set as the goal location of our bot. This goal location is passed to the move_base action client which performs the path planning to the goal. By default, the action client would block the program's control flow until a goal state is reached and this behaviour is unacceptable as the location of the leader bot is dynamic. Hence, we need to pass a preset duration as an argument to the action client's function using *rospy.Duration().* This provides the flexibility for the program flow to step over, while the follower is reaching to the next goal. As long as the fiducial marker's pose is being listened by the listener, the program loop continues as it cancels the previous goal and update the new goal position of our moving fiducial marker. The goal is updated to the action client until the distance between follower's position and the goal position, which is done by the *follower_at_goal* procedure. This mode will be denoted as *detect&follow*

Next, when the fiducial marker is out of sight of the follower's camera, we are hit with a *LookupException.* This exception is handled to search for the marker by rotating in-place, which is achieved using the *rotate* procedure. We start a counter that publishes a preset constant angular velocity along z-axis to */follower/cmd_vel* and the loop continues back to the previous *try* condition. This mode is denoted as *rotate&search.* If the marker is found in the forthcoming iterations, the counter is reset and the *detect&follow* mode takes place. If the marker is not found until the counter exceeds a maximum count level, we need to break out of the *rotatesearch* mode to enter the *explore* mode.

In the *explore* mode, we will retrieve the leader's goal position that was set by the leader node discussed in 2.1. The follower node passes this goal to the movebase client in loop and also checks for the *find_fiducial* procedure to produce a *LookupExecution.* The program breaks out of it's *explore* mode if there is no *LookupException*, thus the control returns to the *detect&follow* mode.

The overall process is given in the Algorithm 3

Algorithm 3 Follower

```
procedure FIND_FIDUCIAL($listener, margin$)
    $T \leftarrow lookupTransform$(map,fiducial_marker)
    $R_{(w,x,y,z)} \leftarrow lookupTransform$(map,leader_tf/base_link)
    $goal \leftarrow (T.x - margin, T.y - margin, T.z, R_{(w,x,y,z)})$
    return $goal$
end procedure

procedure FOLLOWER_AT_GOAL($listener, goal, margin$)
    $T, R \leftarrow lookupTransform$(map,follower_tf_base_link)
    if $abs(goal.position - T) < margin$ then
    return $true$
    else return $false$
end procedure

procedure ROTATE($publisher, twist\_msg$)
    $twist\_msg.z \leftarrow -1.5$
    $publisher.publish(twist\_msg)$
    $sleep\ for\ 5h$
    return $goal$
end procedure

procedure EXPLORE($listener$)
    $goal \leftarrow leader\ goal\ from\ param\ server.$
    while $true$
    $try$
     $if\ a\ LookupException\ for\ FIND\_FIDUCIAL\ occurs$, break
    $except$
     $wait\_state \leftarrow movebase.send(goal)$
     if $!wait\_state$ then $cancel\ goal$; continue
     elsebreak
end procedure

procedure MAIN
    $listener \leftarrow tf.transforms\ listener$
    $margin \leftarrow margin\ to\ maintain\ from\ goal$
    $publisher \leftarrow Publisher\ to\ pass\ Twist\ msg\ to\ topic\ follower/cmd\_vel$
    try
     $goal \leftarrow FIND\_FIDUCIAL(listener, margin)$
     if $FOLLOWER\_AT\_GOAL(listener, goal, margin)$ then
       $stop\ rotating$; continue
     $reset\ counter$
     $wait\_state \leftarrow movebase.send(goal)$
     if $!wait\_state$ then $cancel\ goal$; continue
     elsebreak
    except
     if $counter < 10$ then
      $ROTATE(publisher, msg)$; continue
     else $stop\ rotating$; $reset\ counter$
      $EXPLORE(listener)$
end procedure
```

2.4 Flowchart:

This flowchart shows the decisions the actions the follower takes when running.

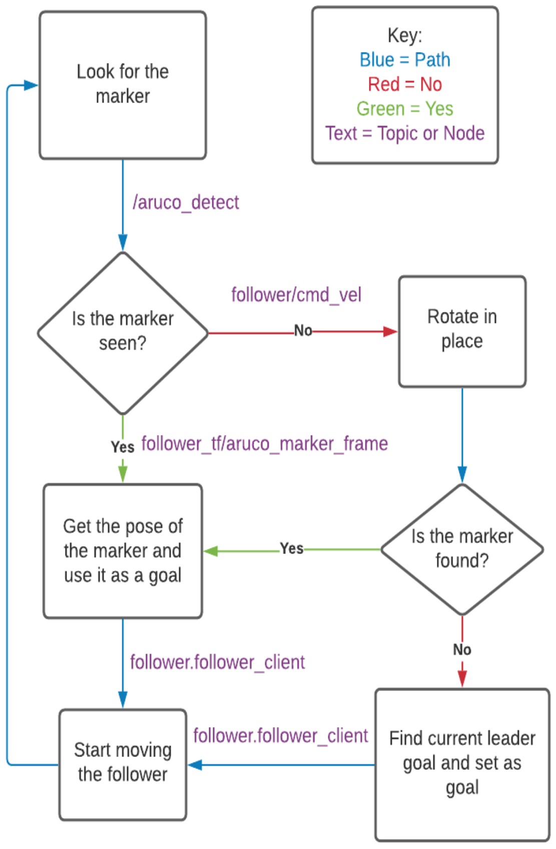


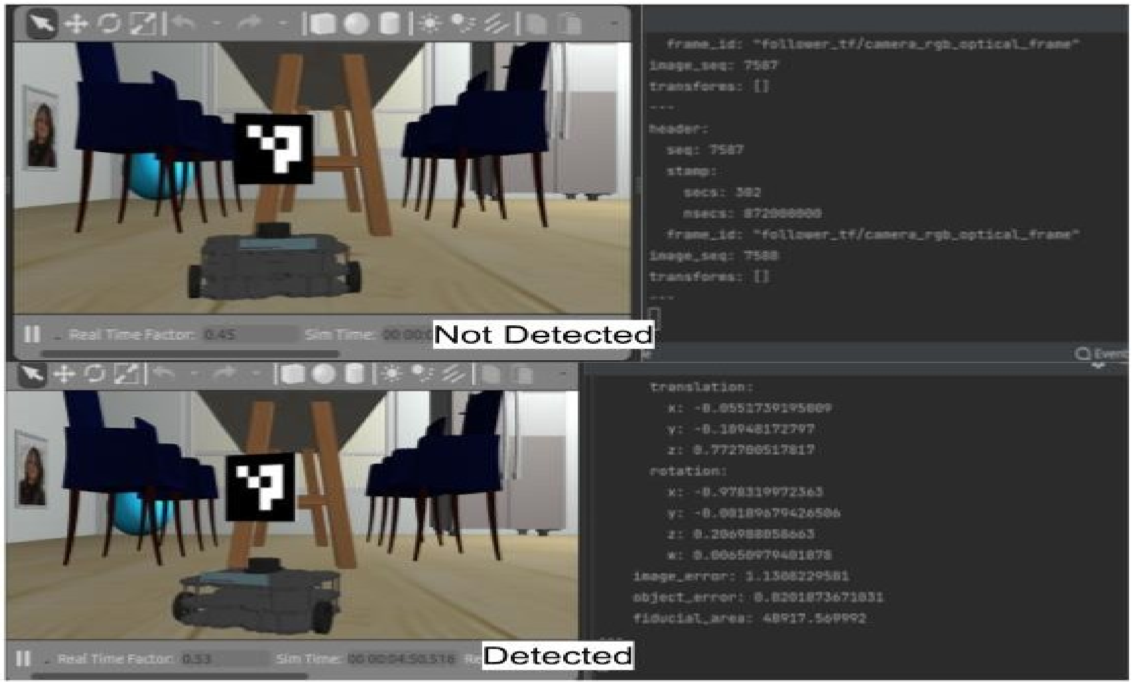
Figure 3: Flowchart of follower decisions

3 Challenges

- - ***Marker not detected*** At first we expected the marker to be detected by the aruco detect package whenever, the marker is in the field of view of the follower's camera. However, we realised a behaviour of the arcuo detect packages, that it fails to detect the marker when there is a black coloured background material present behind one or more edges/corners of the marker. Figure 4 illustrates such a scenario where we can find the transforms object in the fiducial transforms message is empty. This affected the performance at scenarios, but also gave us an idea on the shortcomings of the aruco detect package and had us aware of these issues while building the algorithm.

- - ***Follower runs movebase at short distances.*** The goal position is chosen to be few units away from the fiducial marker position, and when the follower reaches the goal, it again obtains the marker's pose using the *find_fiducial* method and the action client attempts to reach the same goal's neighboring points which is minutely displaced from current goal. This results in strange behavior of the action client and often results in collision with the leader bot. To solve this issue, we introduced the *follower_at_goal* method that skips the execution of the movebase client when the distance between the follower's pose (obtained through the listener) is close to the current goal location

- - ***Orientation issue*** . When the leader reads the goal location in the yaml file, the given ori-



```
  frame_id: "follower_tf/camera_rgb_optical_frame"
image_seq: 7587
transforms: []
---
header:
  seq: 7587
  stamp:
    secs: 382
    nsecs: 872000000
  frame_id: "follower_tf/camera_rgb_optical_frame"
image_seq: 7588
transforms: []
---
```

```
    translation:
      x: -0.0551739195809
      y: -0.10948172797
      z: 0.772700517817
    rotation:
      x: -0.978319972363
      y: -0.00189679426506
      z: 0.206988058663
      w: 0.00650979481078
  image_error: 1.1308229581
  object_error: 0.0201873671031
  fiducial_area: 48917.569992
```

Figure 4: Issue with aruco detect

entation when directly passed to the leader bot's action client, it does not move the bot at all. We noticed that only the *x,y* position and the *z, w* quaternion orientation.

When the follower is provided with all the DOFs with complete pose of the fiducial marker, the follower does not even move. Hence, we provided only the *x,y* position and the *z, w* quaternion orientation to the follower. However, the orientation of the fiducial marker, which is static, is not exactly the orientation of the leader, which is dynamic. Considering the orientation of the fiducial marker as our goal orientation, often results in situations where the follower's camera-side turn away from the leader with the fiducial. To avoid this issue, we directly listen to the leader's orientation using *lookupTransform* and along with the fiducial marker's position, and pass this as the goal pose to our follower as the solution.

- - ***Speed of the robot*** The leader and the follower, when moving at the default set speeds, the likelihood of follower losing track of the leader is high. Hence, we edited the param file to reduce the default set speed of the leader. We wanted to set separate speeds for the leader and the follower. This could've been achieved by separately publishing angular velocity to *leader/cmd_vel* and *follower/cmd_vel* whenever movebase client is called. However, we could not implement and test it within the deadline.

- - ***Follower marker lag issue*** When the follower reaches it's current goal, and at that instant, if the marker is not seen in the frame as the leader moves, it is supposed to enter the *rotate&search* mode instantly. However, this does not happen immediately. The follower still stays in the *detect&follow* mode for quite sometime, since the fiducial broadcaster still continues to broadcast transforms information, which in turn does not raise a *LookupException* in the follower node. This lag is considerably high, which results in scenarios where the leader would've moved a far away before the bot enters the *rotate&search* mode when the LookupException is finally raised. This created majority of the issues we faced with the follower losing track of the leader.

- - ***Stop condition failure*** When the follower completes the entire process of reaching the final location of the leader, it is expected to end the program. To perform this, we appended a "end" key with None value, as the final goal of the leader in the parameter server. When the follower reaches all the goals, and it will eventually reach the *explore* mode where the the final leader location is set to None and when the follower's routine is set to None, it must quit the program. Even after we have included the logic in the follower node our follower continues to stay in the program as it is without moving. We could not solve this issue due to time constraints.

4 Project Contribution

- Gokul Hari: I worked on the follower node mainly, and certain parts of the broadcaster node. I used the 2D Nav goal feature in Rviz to provide new goals to the leader and observe the response of the follower and how the three different follower modes of operation take place. I studied what different frames in the tf tree for the turtlebot signify and how I need to compute the relative pose between two frames in ROS. I also learnt and implemented listeners and publishers and utilised the action client for the overall process. In terms of the report, I handled the explanation of the follower node's approach and the algorithm. I also observed most of the challenges encountered and debugged during the project and explained them in the challenges section.
- Nicholas Novak: I built the leader and waypoint reader files and debugged some issues with reading the coordinates properly. I also created the documentation for all files, created the follower flowchart, and wrote the approach for the Fiducial Broadcaster and Resources section in the report.
- Abijitha:I have worked on the leader approach and documented it thoroughly with pseudocode. I extensively documented the project objectives, approach, and the step-by-step procedure undertaken to complete the goals of the project in the introduction, leader, and the approach section of the final report. Also, I have tested the project for underlying issues.

5 Resources

  - map_server package
  - move_base package
  - aruco_detect package
  - actionlib
  - fiducials
  - amcl package
  - tf_broadcaster
  - navigation
  - http://wiki.ros.org/tf/Tutorials

6 Course Feedback

- Gokul Hari: Pursuing robotics, though has been my goal since 8th grade, ROS was something I never really found time/compulsion to learn it myself due to the nature of deep learning research and projects that I often found myself into. This course came at the right time when

I really wanted to take a summer course to set my overall course schedule correctly. Though I had known python before, I learnt lot of new conventions and coding tips in this course. Though this course is a life-saver for many beginner programmers out there, personally I expected more of ROS and less of python, but that was okay. Professor Zeid's patient approach towards solving doubt and the hands-on nature of this course was a great experience.

- Abijitha Nadagouda: It has been a tremendous experience learning through a hands-on approach to working with ROS. This was my first Robotics class, initially, I was a little worried it might be hard since it's relatively a new course for my background. But the structure of the curriculum, assignments, and the final project gave me a practical understanding of the applications and approach towards robotics programming. Furthermore, the detailed practical python classes prepared me for my professional career. Thank you professor for your efforts in making this a detailed-oriented and hands-on course.

- Nicholas Novak: As my first course in Robotics at Maryland, this course was extremely eye-opening. The introduction to ROS was very useful as it gives me a foundation to build the rest of my ROS experience on. Additionally, I know that I can always fall back on python to create ROS projects before shifting to another language. The only thing that I would change is to have more time in this course. It was very enjoyable and I loved all of the topics. I only wish I could spend more time in each of them. I am aware that this is a result of the class being taught in the shorter summer semester, so it just means that I have to find different areas to apply these newly-learned skills. Thank you for a wonderful introduction to The University of Maryland, Dr. Kootbally. I look forward to seeing you on campus in the coming year!