Lab Procedure for Python
Obstacle Detection

Setup

1. It is recommended that you review Lab 4 – Application Guide before starting this lab.
2. Launch Quanser Interactive Labs, scroll to the "QBot Platform" menu item and then select the "Warehouse" world.
3. Run observer.py first to initiate receiving data feeds from the QBot Platform.
4. On a separate terminal, run line_following.py on the QBot Platform. When the script is run successfully, the QBot and the Environment should be spawn in QLab and User LEDs on the virtual QBot will turn blue. Verify that the QBot is spawn as shown in Figure 1, press the "U" key and stop the script.

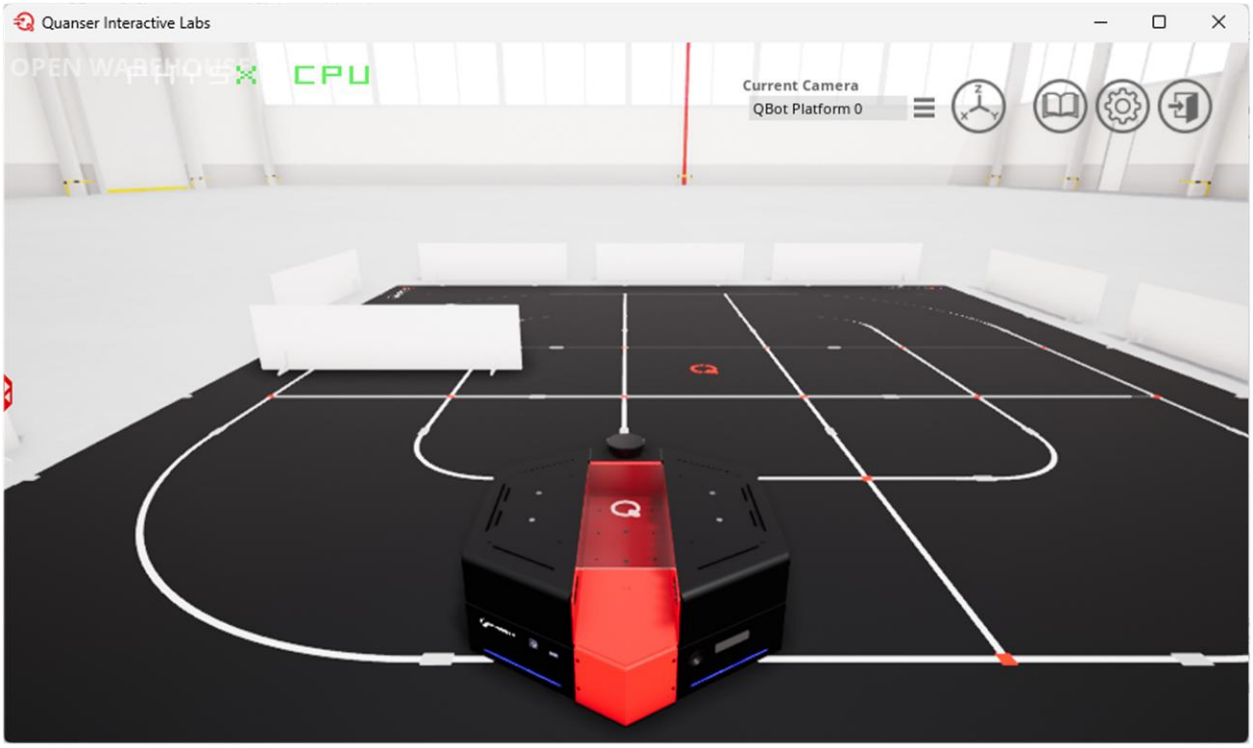

Figure 1. Successful set up of the Quanser Interactive Lab Workspace

LiDAR Data Localization

1. In this lab you will be focusing on processing the LiDAR scan data to enable obstacle detection of the QBot. Open obstacle_detection.py. In Section A, right click over **QBotPlatformLidar()** and select "go to definition". This class make used of the Quanser general **Lidar** object. Right click over **Lidar** and select "go to definition" and take a look at what data is stored in this object. In this lab, only *distances* and *angles* will be used.

2. Go back to the main code, in Section A, right click over **QBPRanging()** and select "go to definition". In this object, the reference frame of the LiDAR data is transferred from the center of the sensor to the center of the QBot. Obstacle detection is also handled by this object, which we will go through later in this lab.

3. Notice in **adjust_and_subsample()** under **QBPRanging()**, heading angles from the LiDAR sensor are multiplied by -1 and added to pi/2. This is to correct the headings such that the angle value of 0 corresponds to the front of the QBot, as shown in Figure 2. In addition, the angles are now increasing counterclockwise, consistent with the positive convention of our reference frame. Furthermore, this function downsamples the distance and angle data, which is beneficial in the obstacle detection application as it allows for faster computation when high resolution is not necessary.

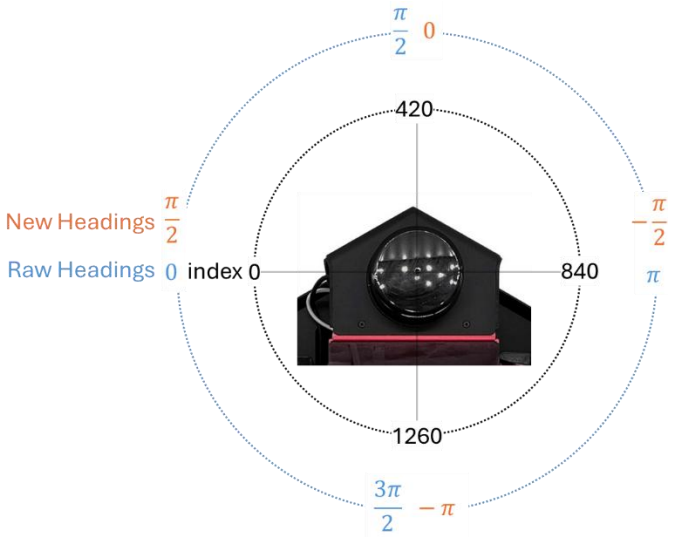

Figure 2. LiDAR measurement correction

4. The **correct_lidar()** function under **QBPRanging()** adjust the center of LiDAR measurements to the center of the QBot. Use the downsampled range and angle data and the position of LiDAR to complete this function.

5. Use **correct_lidar()** and the output of **adjust_and_subsample()** to finish Section E in obstacle_detecion.py.

Dynamic Monitor Region

1. Go to **detect_obstacle()** function under **QBPRanging()**. This function uses the LiDAR measurement to determine when the QBot should stop to avoid collision with an obstacle. In this lab, you will implement a simple obstacle detection by completing this function.

2. Notice that in addition to the LiDAR measurements, QBot body speed commands are also used as input to the function. The additional inputs allow our obstacle detection algorithm to dynamically change monitor region and safety threshold, as shown in Figure 3.

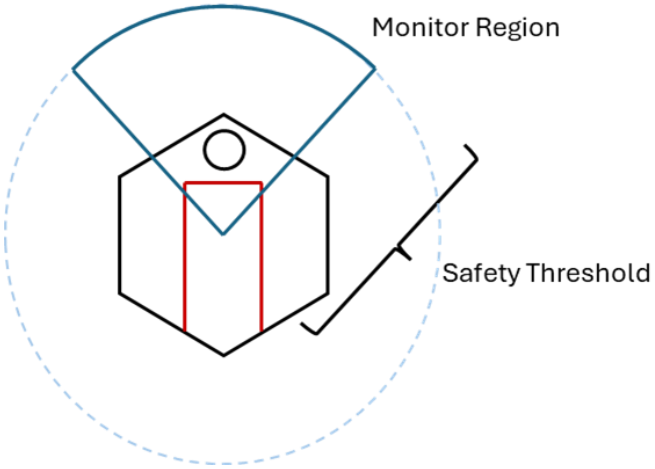

Figure 3. Dynamic Scanning

3. In Section 1 of **detect_obstacle()**, The variable *startingIndex* corresponds to the westmost angles of the monitor region. First, normalize *turnSpd* using the turn speed limit (3.564). Then, derive an equation for *startingIndex* in terms of normalized *turnSpd* and *turnSpeedGain* and complete section 1.

4. Similarly, normalized *forSpd* using the forward speed limit (0.35), then derive an equation for *safetyThreshold* in terms of normalized *forSpd* and *forSpeedGain* and complete section 2.

5. Use **detect_obstacle()** to finish Section F in obstacle_detection.py.

6. Save the change to **QBPRanging()** and go back to obstacle_detection.py. Verify that Section D – Image processing is already completed for you. However, feel free to change the gains in the PD controller or copy over your own functions from the previous lab.

7. Run observer.py first to initialize receiving data. Then, on a separate terminal, run obstacle_detection.py.

8. When observer.py is running, observe the two figures that are opened. They should display the LiDAR scan in the monitor region, as well as a uniform arc representing the safety threshold.

9. Without arming the QBot, press the movement keys. Notice that the monitor region and safety threshold may not be changing with the body speed commands. This is most likely due to *forSpeedGain* and *turnSpeedGain* being too small to produce any noticeable responses. Press the right "U" key to stop the script.

10. Change the two gains in Section F and re-run the scripts. Observe how the monitor region and safety threshold dynamically change in response to the varying speed commands. Iterate through this process as much as you need until you are satisfied with the monitor region responses. Press the "U" key to stop the code.

Obstacle Detection

1. Go to section F in obstacle_detection.py. Notice that *minThreshold* has already been define for you. This variable represents the radius of the bounding circle of QBot.

2. Go to **detect_obstacle()** function under **QBPRanging()**. In section 3, the total number of points that lays between *minThreshold* and *safetyThreshold* will be computed and compared to *obstacleNumPoints*. When total number of obstacle point exceeds *obstacleNumPoints*, *obstacleFlag* should be set to true. Complete section 3 and save your changes.

3. Use **detect_obstacle()** to complete section F.

4. Run observer.py first to initialize receiving data. Then, on a separate terminal, run obstacle_detection.py.

5. Without arming the QBot, gradually move an obstacle towards the LiDAR sensor to trigger *obstacleFlag*. When an obstacle is detected, the User LED will turn magenta.

6. How sensitive is your obstacle detection algorithm to obstacle? Does it trigger too early or too late?

7. Tune *obstacleNumPoints* until you are satisfied with the obstacle detection response.

8. While arming the QBot, drive it to a line on the mat, and press the "7" key to start line following. Observe the obstacle detection response to different scenarios.

    a. The obstacle is directly on the line, as shown in figure 4a.

    b. The obstacle is on the side of the line such that the QBot would narrowly drive pass the obstacle, as shown in figure 4b.

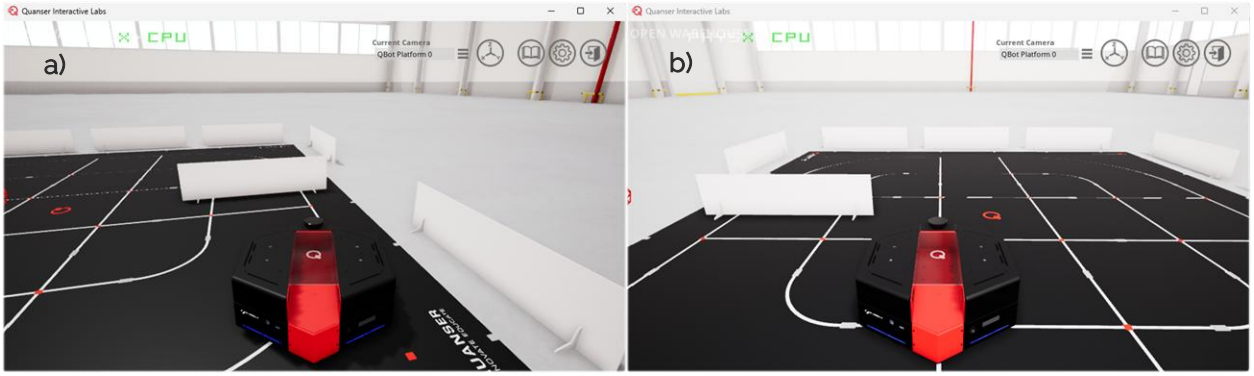

Figure 4. a) Obstacle directly on path; b) obstacle on the side of the path

9. Does the QBot stop in time to avoid collision? Does the QBot stop when it could have moved pass the obstacle?

10. Can you move the QBot closer to obstacle after *obstacleFlag* is triggered?

11. Tune *forSpeedGain*, *turnSpeedGain*, and *obstacleNumPoints* until the QBot can response correctly to the two obstacle configurations in step 7.

12. Stop the code when complete by pressing the "U" key. Ensure that you save a copy of your completed files for review later. Close Quanser Interactive Labs.