QR Tracking using ClearPath Jackal

Atharva Y. Risbud
Computer Science
North Carolina State University
Raleigh, NC, USA
atharva.risbud@gmail.com

***Abstract*—The ClearPath Jackal UGV can be programmed to detect a QR code attached to a moving object and subsequently track and follow the object while maintaining a safe distance. The system can be further enhanced to retrace the path of the moving object while avoiding any obstacles that may be present. We want to implement this approach with a warehouse setting in mind to help with material transport.**

This is a pilot project to show that using the ROS 2 framework, we can pass specific commands to the robot based on which the robot will take action. The robot is also able to read QR codes and process information encoded in the QR code. We have used the gazebo virtual environment to set up the system in place.

***Index Terms*—ROS, ClearPath Jackal, QR Codes, SLAM, Computer Vision**

I. Introduction

Warehouses have historically been labor-intensive environments with potential for safety hazards. Before the advent of modern computer vision systems and advancement in control systems, robots were restricted to predictable and accurately controlled environments. But these are historical limitations. Nowadays, Warehouse robots are still one of the most used application robots since they carry out more involved tasks. In recent years, robots have emerged as a promising solution to improve the efficiency and safety of warehouse operations. The demand for robots in warehouse spaces is on the rise. Notably, factories are one of the few settings where a fully autonomous system can be deployed without stringent government regulations.

In the context of warehouse robotics, a promising approach involves using follower robots that track a lead robot. To implement this approach, a simple camera-equipped robot can be programmed to follow a QR code attached to the lead robot, maintaining a safe distance and stopping appropriately. This could be a potential product for material transportation within the warehouse. In future work, the follower robots could also be made to track and follow human workers in the warehouse, increasing their flexibility and usefulness.

Using a ClearPath Jackal UGV to design a ”follow bot” is this project's scope. We are able to set up a viral environment with a custom-designing ”world,” which has QR code embeddings. By reading these embedded QR codes, the ”follow bot” is able to detect which object will be tracked and followed.

The remaining part of this report is split into Section III, which discuses the details of the software packages and how they are integrated to perform the experiment described in Section V. Section IV talks about the challenges during implementation and the leanings from those. Section VI concludes the report, and section VII talks about the future scope

II. Related Work

The historic technique of visual servoing described by Hutchinson et al [1] uses visual feedback to control the end effector and make it a closed-loop control system. The general framework of visual servoing involves using cameras mounted on robots to provide visual feedback and control their path in real-time. This paper laid the groundwork for current standard approaches to visual servoing used in robotics today. Though evaluating camera data is computationally expensive, recent advances in CPU and GPU technology have made real-time implementation of computer vision much more feasible.

The paper by Atali et al. describes a study in which a new path-planning algorithm is evaluated using QR codes to guide a mobile industrial robot [2]. A matrix of QR codes is created to represent the work area and guide the robot on the path to the target point. The authors used the Kobuki robot platform and ROS for development. The robot reads the location information on the QR codes and compares it with the path information sent to it before it starts moving. In the algorithm proposed by the authors, they additionally use wireless communication to send information.

The work by ZHang et al. discusses localization and navigation for indoor environments in detail [3]. Their setup is a robot pointing a camera to the ceiling and reading landmark QR Codes at high speeds. The authors have used a laser range finder to map the area to avoid collision with unknown objects. Algorithms like Dijkstra's and Dynamic Window Approach, which are based on ROS packages, are used for path planning for the system. The publication also discusses approaches to QR detection using the ZBar library and Open Source Computer Vision (OpenCV). This work is a complete system that takes advantage of QR codes for localization and successfully shows path planning, at least for indoor environments.

Previous research, such as the work by Purnama et al., has also explored the implementation of a ”follow bot” that can track and follow humans [5]. In their study, they discussed the use of template matching with HOG (Histogram of Gradients) for detecting human figures and also included tracking the human's distance from the robot and detecting its pose.

This suggests that utilizing QR codes for position localization or object tracking is a well-established approach, and

we are confident that in the current technology landscape, this would be the most effective method to implement our proposed robot application. The literature review has provided valuable insight into how similar problems have been addressed in the past.

III. Methodology

A. Jackal

The Jackal is an Unmanned Ground Vehicle (Jackal) created by ClearPath Robotics, renowned for its power and versatility. Equipped with a potent x86 architecture CISC CPU (as opposed to RISC or ARM), it also supports the addition of the Jetson AGX Xavier for increased GPU capabilities. With external dimensions of 508 x 430 x 250 mm and a weight of approximately 17 kg, the Jackal boasts a 20 kg payload capacity, making it a great choice for exploration and warehouse tasks. Its battery has the capacity to support up to 4 hours of use, and it can achieve a maximum speed of 20 m/s. Additionally, the Jackal has the flexibility to accommodate various accessories, including single or stereo cameras, LiDAR sensors, bumper sensors, GPS, wireless communication, IMUs, and even robotic arms. A range of packages is available, such as the Jetson Package, the Explorer Package, and the Navigator Package. The Jackal is widely used in diverse applications, such as navigation, tracking, VSLAM, 3D imaging, machine learning, and human-robot interaction.

Fig. 1. Real World Image of a ClearPath Jackal UGV

The Jackal is designed to support the ROS framework, which enables seamless integration with various platforms. It is compatible with both ROS1 and ROS2. Another advantage of the Jackal is its integration with Gazebo, which simplifies software development and makes it more cost-effective. The current version of the Jackal Gazebo Integration supports ROS1 (Melodic, Noetic) as well as ROS2 (Foxy). However, the ROS2 integration with Gazebo is still in development, and not all packages and plugins available with ROS1 have been fully ported to ROS2, which poses a challenge.

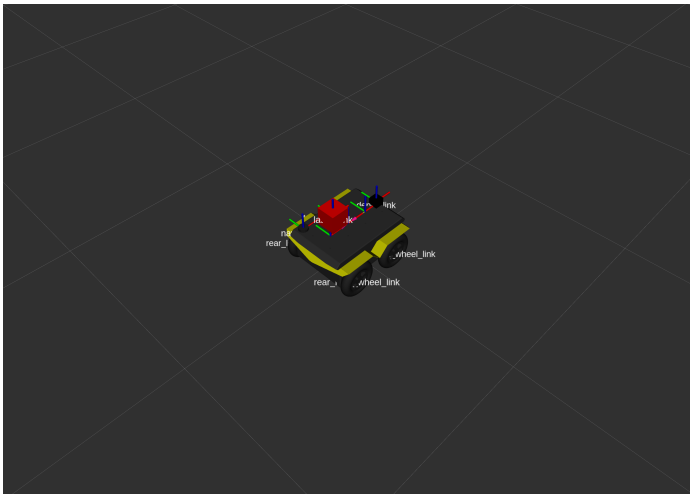

Fig. 2. RViz model of Jackal

B. Gazebo

A virtual environment gives controlled and repeatable settings without the need for robot assembly or hardware malfunctions. Since the emphasis of the project is integration across and not the real world applicable, we made the choice of the virtual world. Another factor was the number of people working on the project and the duration of the project, in which it was not possible to do real-world testing of the project and account for the uncertainties and risks. Moreover, virtual environments offer the flexibility to modify various parameters, and this helped when experimenting with new directions for the project.

Gazebo is a popular open-source robot simulation tool used for testing and prototyping in a virtual environment. It was first developed as part of a DARPA project. Gazebo 11, the latest version, was used for this project, which is supported in Linux.

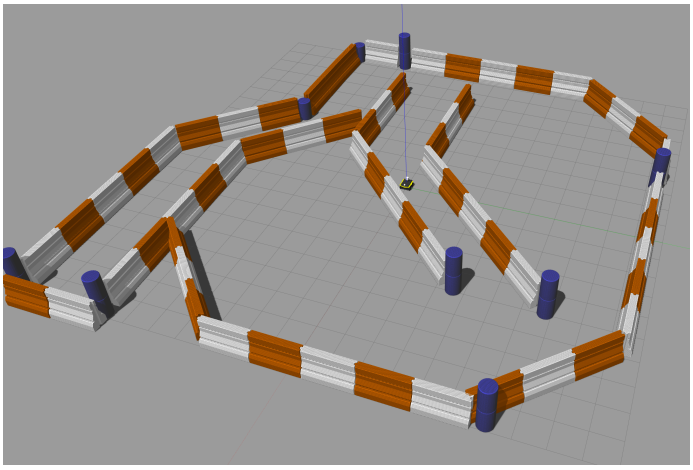

Fig. 3. Default Race World for Jackal Gazebo

Alternative robot simulation tools include V-REP, Webots, and MORSE, but Gazebo's robustness and features make it one

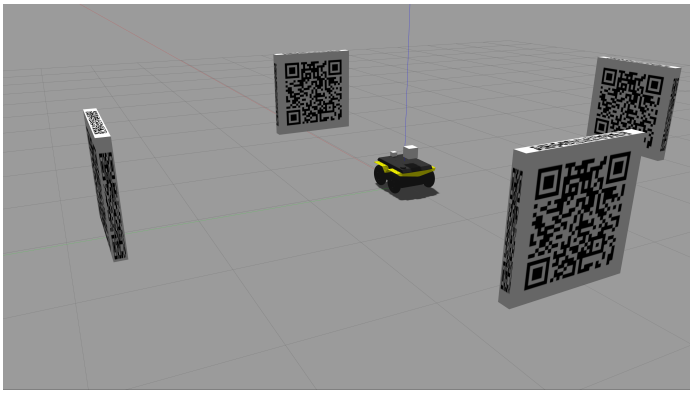

Fig. 4. Custom World With QR Codes

of the most preferred options. One of the reasons for Gazebo's popularity is its integration with the Robot Operating System (ROS), allowing for the jackal to be directly imported into the environment. Gazebo's simulation environment also provides flexibility in simulating various other robot models. The choice the jackal thought depended on the physical availability of the bot, it could prove an insightful experiment to test the performance of existing algorithms against robots with similar specs to the gazebo virtual environment.

C. QR codes

Quick Response (QR) codes were initially developed to track vehicles during manufacturing and can be written and read by various devices, storing more information than traditional barcodes. QR codes are scannable by mobile devices and can provide an efficient means of storing and transmitting data, improving communication and coordination among different components in a warehouse environment. Fiducial markers, such as AprilTags, are also used for localization and tracking in robotics and computer vision. These markers feature unique patterns that can be easily recognized by cameras for accurate pose estimation, even under challenging illumination conditions. While AprilTags have faster computing times, they do not store as much information as QR codes. Additionally, the way of having embedded programming lines in QR codes to prompt the robot to take the next action makes them more versatile. These lines of embeddings could increase in complexity. Due to these reasons, QR codes were selected for this project.

In the paper by Sneha et al. the authors have utilized QR codes in their project to store additional information related to the path that the robot follows, rather than embedding lines of code within the QR code [3].

Fig. 5. QR Code as seen from the simulated camera

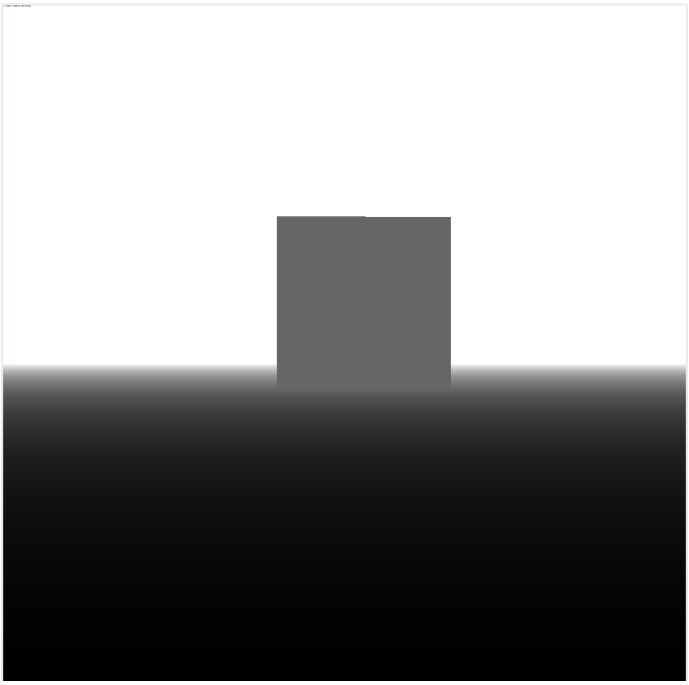

Fig. 6. Depth map of the QR Code

D. Integration

The first step of the integration process for this project involved installing ROS2 Foxy on Ubuntu, specifically version 20.04 (Focal), due to its compatibility with ROS Foxy. The installation process for both ROS Foxy and ROS Noetic was completed by following their respective documentation. With ROS successfully set up, the next step was to install packages for Jackal Simulation, which were initially installed natively to Ubuntu. However, there were a few challenges encountered during this process, which have been documented in the challenges section. One minor inconvenience was the incompatibility between Anaconda and Gazebo, necessitating the deactivation of the Anaconda Environment to launch Gazebo. Once this was completed, the race track world was

successfully simulated in Gazebo.

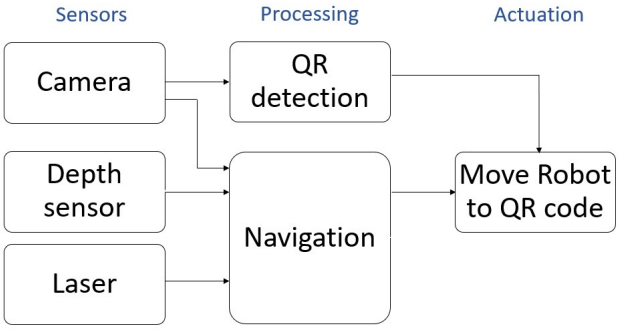

Fig. 7. Flow Chart of the system

The subsequent step involved generating and scanning QR codes. Initially, this was done using Python and OpenCV, along with a webcam for scanning. To generate QR codes, we used the ”qrcode” library, which produced a 2D QR code in PNG format. For the first detection of QR codes, we used a webcam along with OpenCV in Python. This allowed us to obtain the bounding box for the QR code and its corresponding text.

The stock simulated Jackal only included the base, so we needed to add a camera (with a depth camera) and a LiDAR for our project. During this process, we discovered that a separate workspace needed to be created for the project, and all of the Jackal simulation-related files had to be downloaded and built. This was necessary because it’s not possible to edit the base installation of either ROS or Jackal Gazebo. The appropriate folder structure was created for the workspace, and all of the necessary Jackal-related files were cloned. We then vetted the dependencies using rosdep and built the project using colcon build.

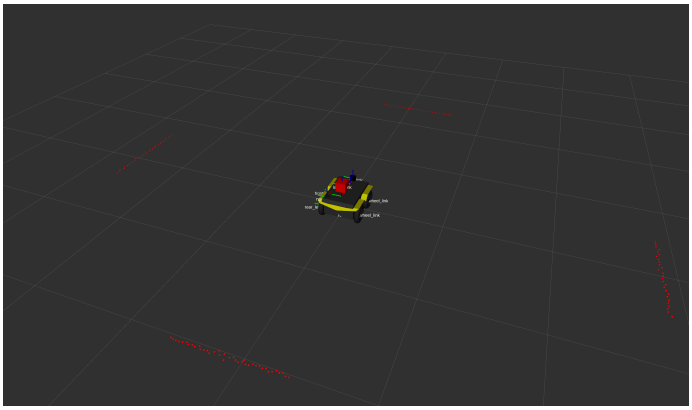

Fig. 8. Rviz Model of Jackal with Lidar Showing position of QR Code Walls

Next, a camera was added to the setup. To attach any accessory to the Jackal, two files needed to be created: a .urdf.xacro file that specifies where and how the attachment

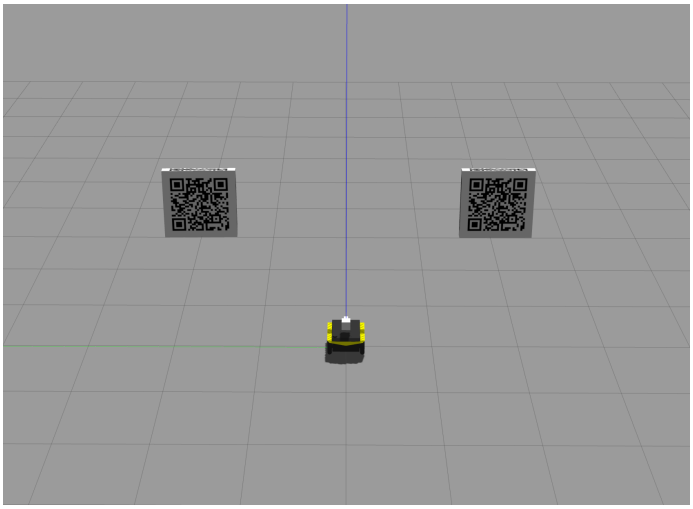

Fig. 9. Experimental Setup

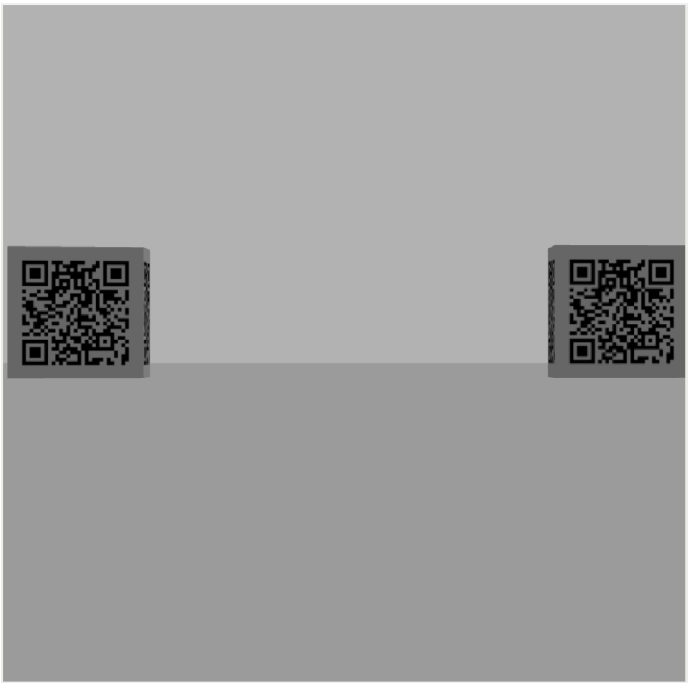

Fig. 10. Experimental Setup as seen from the simulated camera

should be placed (including the links and joints and how and where they connect), and a .gazebo file that detailed the sensor we wanted to add, along with its intrinsic and extrinsic properties.

In the .xacro file, a small box was created to house the camera. ROS2 provides plugins that help Gazebo simulate specific accessories, which are typically in the .so file format and contain the word ”libgazebo”. In ROS Foxy, several of these plugins have been combined for greater efficiency, such as the stereo camera and depth camera being combined in the camera module itself. The type of sensor specified in the .gazebo file determines the behavior of the accessory. For the

camera setup, the camera plugin was used with the sensor type set to ”depth”, which enabled the camera to publish color and depth output as topics with names such as /depth/.. or /color/...

Subsequently, the lidar sensor was installed. In contrast to ROS1, setting it up was more complicated since the Jackal Gazebo did not include several LiDAR accessories. After thorough research, it was discovered that the ray sensor plugin had replaced all laser sensors. A box was then created in the .xacro file, and the behavior of the box was specified in the .gazebo file. The sensor type was set to ”ray,” and its minimum and maximum angles were adjusted to capture a 360° angle. The laser sensor then publishes to the ”laserscan” topic in ROS.

To integrate the QR Detector with the Jackal in Gazebo, an individual created a Python file in the src folder in the jackal_gazebo package, which is where the simulation and its launch files are located. They set up a publisher and subscriber, where the subscriber would subscribe to the camera topic, and the publisher would publish the data inferred from the QR Code. To accomplish this, they utilized the OpenCV and cv_bridge libraries. The QR Code was detected from the images published by the camera color topic, and the detectAndDecodeMulti function from OpenCV was used. This primarily returned whether a QR Code exists in the frame, the vertices of the QRCode, and the information that was decoded from the QR Code.

The approach envisioned was to set up a waypoint to the QR Code and then navigate to it using Nav2. To achieve this, the following process was planned:

- Obtain the pixel coordinates of the edges of the detected QR Code.
- Determine the approximate center pixel coordinates.
- Use the depth camera to retrieve the depth at that point.
- Convert the pixel-level coordinates to world coordinates.
- This would result in a waypoint that can be provided to Nav2 for navigation.

IV. Challenges

A. Setting up ROS and Jackal Gazebo

Setting up ROS and Jackal Gazebo involved installing compatible versions of Ubuntu, ROS Noetic, and ROS Foxy. Initially, Ubuntu 22.04 (Jammy) was used, but it was found that the ROS version compatible with Jackal Gazebo could not be installed on this distro. Further investigation led to the understanding that Ubuntu 20.04 was compatible with ROS Noetic and Foxy, as well as Jackal Gazebo. Therefore, a separate OS partition was created for Ubuntu 20.04, and ROS Noetic and Foxy were installed natively. This was necessary because a simulator needed to be run on the system, and a native installation allows the OS to take full advantage of the available resources such as CPU cores, GPU, and RAM. In hindsight, it would have been useful to research the compatibility requirements of ROS and install a compatible version of Ubuntu to avoid such issues.

B. Lack of documentation for ROS2 Foxy with Jackal

The documentation for simulating Jackal in Gazebo is limited for ROS2 Foxy. While the documentation for ROS1 distros like Noetic and Melodic is extensive enough to cover adding LiDAR and navigation, it is not the case for ROS2. The Jackal website provides basic instructions for launching the race track world with a Jackal in it, but documentation for adding plugins for ROS2 is limited. Similarly, the Gazebo website has detailed instructions for adding plugins for ROS1 distros, but the plugins and their features are not readily available for ROS2. After extensive research, it was discovered that a lot of plugins had been combined for ROS2, and the documentation for their use was missing. A porting issues and resolution page was found on Github, but more comprehensive documentation would have been helpful.

C. Adding attachments to the Jackal in Gazebo

The process of adding attachments to Jackal in Gazebo was not very clear. Helpful and detailed tutorials were not available. After consultation with our Professor, Dr. Ore, the procedure to add attachments became clear. Overall we feel that this process was not very intuitive.

D. Integrating a new ros publisher/subscriber node with python

From overall resources on the web, it was not very clear how to add additional Python files to a package. It was discovered that the files needed to be added to the src folder of the ROS package. The way to run these files was also not very clear. At first, we tried to modify the CMake file (from a resource found on the web), but that did not solve the issue. We decided that we would run these files manually using the python3 command from the terminal.

E. Mapping

For our approach, we depended on Nav2 to navigate the robot in the world and to get to the goal. It was understood that we would first need to create a map of the world we want the jackal to move. Again here, several techniques seem to be available, including gmapping, octomap, and slam-toolbox. For ROS2, we found that the slam toolbox was the best way to map an environment. However, from the online resources that we found, we were unable to use the tool that actually visualizes the area mapped. This tool works using RViz. We are still trying to figure out how to make the mapping work so that we can use Nav2 to give it the waypoint to the QRCode that we want.

F. Trouble with Anaconda and Gazebo

For some odd reason, we found that the jackal gazebo launch behaves erratically when launched through an anaconda environment. Hence to launch the gazebo, we had to deactivate all anaconda environments. However, everything else works perfectly well with Anaconda.

V. Experiments and Results

The experimental setup for this project includes a Jackal in a Gazebo world. The Jackal has a camera (with a depth camera) and a LiDAR sensor on it. The entire project is done on a Ryzen CPU laptop with an Nvidia GPU. The operating system used is Ubuntu 20.04. The Gazebo World has a wall with two QR Codes. We would pass on a String that will indicate which QR Code the robot needs to follow. The robot will then mark a waypoint in the Gazebo world and travel to the location using Nav2.

VI. Conclusion

This project proves to be a pilot experiment to demonstrate the ability of a "follow bot" to detect the position of the QR code, read and decode the code, and also identify if that code belongs to an object to be followed. This project was successful in integrating a virtual world with a robot camera that can process camera input and react to that information. This is a good fundamental project to reinforce the learning taught during the course and also how to handle implementation in ROS 2 and Gazebo environment. These foundation skills were further developed as a result of this exercise. Overall, this project serves as a valuable starting point for exploring the capabilities and limitations of autonomous robot systems in complex environments.

VII. Future Work

The current scope of the project only assumes that we go to a particular QR Code that is situated on a wall. In the future, we would like to implement an actual autonomous robot in Gazebo and have the robot follow it. This currently went out of scope since adding another autonomous robot is a difficult and long process. Furthermore, we could also include implementing more advanced algorithms for object detection and tracking, such as reinforcement learning-based approaches, to improve the robot's autonomy and robustness.

Additionally, there could be a system where robots can adjust the camera to scan a specified region to detect the presence of a QR code and identify if the QR is relevant to the task. The QR code could also have additional information embedding which is relevant to warehouse operations. Finally, this project could also be deployed on a physical Jackal to test its performance in real-world scenarios. For this, more safety measures would need to be additionally integrated into the system.

References

[1] S. Hutchinson, G. D. Hager, and P. I. Corke, "A tutorial on visual servo control," IEEE Transactions on Robotics and Automation, vol. 12, no. 5. Institute of Electrical and Electronics Engineers (IEEE), pp. 651–670, 1996. doi: 10.1109/70.538972.

[2] G. Atali, Z. Garip, S. S. Ozkan, and D. Karayel, "Path Planning of Mobile Robots Based on QR Code," Academic Perspective Procedia, vol. 1, no. 1. Academic Perspective, pp. 31–38, Nov. 09, 2018. doi: 10.33793/acperpro.01.01.9.

[3] A. Sneha, V. Teja, T. Mishra, and K. Satya Chitra, "QR Code based Indoor Navigation system for Attender Robot," EAI Endorsed Transactions on Internet of Things, vol. 6, no. 21. European Alliance for Innovation n.o., p. 165519, Aug. 17, 2020. doi: 10.4108/eai.13-7-2018.165519.

[4] H. Zhang, C. Zhang, W. Yang, and C.-Y. Chen, "Localization and navigation using QR code for mobile robot in indoor environment," 2015 IEEE International Conference on Robotics and Biomimetics (ROBIO). IEEE, Dec. 2015. doi: 10.1109/robio.2015.7419715.

[5] I. K. E. Purnama, M. A. Pradana, and Muhtadin, "Implementation of Object Following Method on Robot Service," 2018 International Conference on Computer Engineering, Network and Intelligent Multimedia (CENIM). IEEE, Nov. 2018. doi: 10.1109/cenim.2018.8710819.

APPENDIX

Code Artifacts

A. xacro and gazebo files

```
<!-- Depth Camera xacro -->
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/
    wiki/xacro">

   <xacro:property name="depth_link" value
      ="0.05" /> <!-- Size of square '
      camera' box -->

   <joint name="depth_joint" type="fixed">
   <axis xyz="0 1 0" />
   <origin xyz="0.2 0 0.2" rpy="0 0 0"/>
   <parent link="base_link"/>
   <child link="depth_link"/>
   </joint>

   <!-- Camera -->
   <link name="depth_link">
      <collision>
         <origin xyz="0 0 0" rpy="0 0 0"/>
         <geometry>
      <box size="${depth_link} ${
         depth_link} ${depth_link}"/>
         </geometry>
      </collision>

      <visual>
         <origin xyz="0 0 0" rpy="0 0 0"/>
         <geometry>
      <box size="${depth_link} ${
         depth_link} ${depth_link}"/>
         </geometry>
         <material name="red"/>
      </visual>

      <inertial>
         <mass value="1e-5" />
         <origin xyz="0 0 0" rpy="0 0 0"/>
         <inertia ixx="1e-6" ixy="0" ixz="
            0" iyy="1e-6" iyz="0" izz="1e
            -6" />
      </inertial>
   </link>
</robot>
```

```
<!-- Depth Camera gazebo -->
<?xml version="1.0"?>
<robot>
   <gazebo reference="depth_link">
   <sensor name="depth1" type="depth">
      <always_on>true</always_on>
      <update_rate>30</update_rate>
      <pose>0 0 0 0 0 0</pose>
      <camera name="realsense_depth_camera
         ">
         <horizontal_fov>1.46608</
            horizontal_fov>
         <image>
            <width>800</width>
            <height>800</height>
            <format>B8G8R8</format>
         </image>
         <clip>
            <near>0.05</near>
            <far>5</far>
         </clip>
      </camera>
      <plugin name="
         intel_realsense_d430_depth_driver
         " filename="libgazebo_ros_camera
         .so">
         <ros>
            <namespace>d430</namespace>
            <remapping>depth/image_raw:=
               color/image_raw</remapping
               >
            <remapping>depth/depth/
               image_raw:=depth/
               image_rect_raw</remapping>
            <remapping>depth/camera_info:=
               camera_info</remapping>
            <remapping>depth/depth/
               camera_info:=depth/
               camera_info</remapping>
            <remapping>depth/points:=depth
               /points</remapping>
         </ros>
         <camera_name>depth</camera_name>
         <frame_name>depth_link_optical</
            frame_name>
         <hack_baseline>0.07</
            hack_baseline>
         <min_depth>0.05</min_depth>
         <max_depth>20</max_depth>
      </plugin>
   </sensor>
</gazebo>
</robot>
```

```
<!-- Laser xacro -->
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/
    wiki/xacro">
   <joint name="laser_joint" type="fixed">
      <axis xyz="0␣1␣0" />
      <origin xyz="-0.1␣0␣0.3" rpy="0␣0␣0"
          />
      <parent link="base_link"/>
      <child link="laser_link"/>
   </joint>

   <!-- Hokuyo Laser -->
   <link name="laser_link">
      <collision>
      <origin xyz="0␣0␣0" rpy="0␣0␣0"/>
      <geometry>
      <box size="0.1␣0.1␣0.1"/>
      </geometry>
      </collision>

      <visual>
      <origin xyz="0␣0␣0" rpy="0␣0␣0"/>
      <geometry>
         <box size="0.1␣0.1␣0.1"/>
      </geometry>
      </visual>

      <inertial>
      <mass value="1e-5" />
      <origin xyz="0␣0␣0" rpy="0␣0␣0"/>
      <inertia ixx="1e-6" ixy="0" ixz="0"
          iyy="1e-6" iyz="0" izz="1e-6" />
      </inertial>
   </link>
</robot>
```

```
<!-- Laser gazebo -->
<?xml version="1.0"?>
<robot>
  <gazebo reference="laser_link">
    <sensor type="ray" name="laser_link">
      <pose>0 0 0 0 0 0</pose>
      <visualize>false</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>500</samples>
            <resolution>1</resolution>
            <min_angle>-3.14</min_angle>
            <max_angle>3.14</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.10</min>
          <max>30.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="
          gazebo_ros_ray_controller"
          filename="libgazebo_ros_ray_sensor
          .so">
      <ros>
          <remapping>~/out:=scan</remapping>
        </ros>
        <topicName>laser/scan</topicName>
        <frameName>laser_link</frameName>
        <output_type>sensor_msgs/LaserScan</
            output_type>
      </plugin>
    </sensor>
  </gazebo>
</robot>
```

B. python file

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image,
    CameraInfo
import cv2 # OpenCV library
from cv_bridge import CvBridge
from std_msgs.msg import String
import numpy as np

class ImagePublisher(Node):
```

```
def __init__(self):
  super().__init__('image_publisher')
  self.image = None

  self.counter = 0

  # Create the publisher. This publisher
     will publish an Image
  # to the video_frames topic. The queue
     size is 10 messages.
  self.publisher_ = self.create_publisher
     (String, 'qr_string', 10)
  self.publisher_2 = self.
     create_publisher(String, "
     rect_points", 10)

  # publish a message every 0.1 seconds
  timer_period = 0.1 # seconds


  # # Used to convert between ROS and
     OpenCV images
  self.br = CvBridge()
  self.subscription2 = self.
     create_subscription(Image, 'd430/
     depth/image_rect_raw', self.
     depth_callback, 1)
  self.subscription3 = self.
     create_subscription(CameraInfo, '
     d430/camera_info', self.
     camInfo_callback, 1)
  self.subscription = self.
     create_subscription(Image, 'd430/
     color/image_raw', self.
     image_callback, 1)
  self.subscription4 = self.
     create_subscription(String, '/
     toFollowString', self.
     getFollowString_callback, 1)
  # self.midpoint = []
  self.depthMap = None
  self.K = None
  self.R = None
  self.t = None
  self.toFollowString = None

def image_callback(self, msg):
  myMidpoint = 0
  myQrDetector = cv2.QRCodeDetector()

  try:
    cv_image = self.br.imgmsg_to_cv2(msg,
        "bgr8")
  except Exception as e:
    self.get_logger().error('Error␣
       converting␣ROS␣Image␣to␣OpenCV:␣{}
       '.format(e))
    return
  ret_qr, decoded_info, points, _ =
     myQrDetector.detectAndDecodeMulti(
     cv_image)
  if ret_qr:
    for s, p in zip(decoded_info, points)
       :
      if s == self.toFollowString:
        try:
          myS = String()
          myS.data = s
          self.publisher_.publish(myS)
          a = p[0]
          b = p[1]
          c = p[2]
          d = p[3]
          intersectionPar = [int((a[0] + b
             [0] + c[0] + d[0])/4), int((a
             [1] + b[1] + c[1] + d[1])/4)]
          print(intersectionPar)
          print("Second␣Print␣of␣depth␣map:
             ␣", self.depthMap)
          myDepthVal = self.depthMap[int(
             intersectionPar[0]), int(
             intersectionPar[1])]
          print("Print␣of␣depth␣val:␣",
             myDepthVal)
          #convert to camera co_ordinate
             system
          cameraSys = np.dot(np.linalg.inv(
             self.K), np.array([
             intersectionPar[0],
             intersectionPar[1], 1]))
          print("Camera␣Sys␣print:␣",
             cameraSys)
          #convert to world co_ordinate
             system
          myCamDepth = myDepthVal *
             cameraSys
          print("Camera␣sys␣with␣depth:␣",
             myCamDepth)
          print("R␣Transpose:␣", self.R.T)
          print("myCamDepth:␣", myCamDepth)
          print("value␣of␣T:␣", self.t)
          print("Difference", myCamDepth -
             self.t)
          myWorldCoOrds = np.dot(self.R.T,
             myCamDepth - self.t) #np.dot(
             self.R.T, myCamDepth) - np.
             dot(self.R.T, self.t) #
          print("World␣Coordinates:␣",
             myWorldCoOrds)
          print(type(myWorldCoOrds))
```

```
        # self.pu
        print(s)
        # print(type(s))
        color = (0, 255, 0)
      except Exception as e:
        self.get_logger().error('Error␣
            getting␣waypoint:␣{}'.format(
            e))
    else:
      color = (0, 0, 255)

 def camInfo_callback(self, msg):
  print("Camera␣Info␣Message:␣", msg)
  self.K = np.array(msg.k).reshape(3, 3)
  self.R = np.array(msg.r).reshape(3, 3)
  P = np.reshape(msg.p, (3, 4))
  Tx = -P[0][3] / P[0][0]
  Ty = -P[1][3] / P[1][1]
  Tz = -P[2][3] / P[2][0]
  T1 = [Tx, Ty, Tz]
  print("T1:␣", T1)
  K_inv = np.linalg.inv(self.K)
  T = K_inv.dot(P[:, 3])
  print("Value␣of␣T:␣", T)
  self.t = T

 def depth_callback(self, msg):
  try:
     self.depthMap = self.br.
         imgmsg_to_cv2(msg, "passthrough"
         )
     print("Depth␣Map␣Type", type(self.
         depthMap))
     print("Depth␣Map␣Shape", self.
         depthMap.shape)
  except Exception as e:
     self.get_logger().error('Error␣
         converting␣ROS␣Image␣to␣DepthMap
         :␣{}'.format(e))
     return

 def getFollowString_callback(self, msg):
  self.toFollowString = msg.data

 def timer_callback(self):
  depthofPoint = self.depthMap[self.
      midpoint[0], self.midpoint[1]]
  myWaypoint = (self.midpoint[0], self.
      midpoint[1], depthofPoint)


def main(args=None):

 # Initialize the rclpy library
 rclpy.init(args=args)

 # Create the node
 image_publisher = ImagePublisher()

 # Spin the node so the callback function
     is called.
 rclpy.spin(image_publisher)

 # Destroy the node explicitly
 # (optional - otherwise it will be done
     automatically
 # when the garbage collector destroys
     the node object)
 image_publisher.destroy_node()

 # Shutdown the ROS client library for
     Python
 rclpy.shutdown()

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