AE DELIVER

Spring 2022 Intelligent Mobile Robotics (CS-424-01, CS-524-01)

1st Aleyna Ceylan
Information Systems Engineering
Computer Science
Vestal, United States
aceylan1@binghamton.edu

2nd Erkut Cetiner
Information Systems Engineering)
Computer Science
Binghamton, United States
ecetine2@binghamton.edu

***Abstract*—The aim of this project is to develop an on-demand autonomous delivery service using a navigation robot. The service will be coded in Gazebo ROS and Python, and will be designed to autonomously deliver food, grocery, and parcels from initial location and then dispatch for the delivery location and auto navigation channels agent to the drop-off location.**

I. Introduction

A. AEDelivers Navigation Robot Overview

II. Intelligent Agent Deployment

In order to create a delivery agent in ROS, we first need to create a new package. We can do this with the following command: catkin_create_pkg delivery_agent rospy std_msgs message_generation actionlib actionlib_msgs. This will create a new directory called delivery_agent, which contains a few files and directories. The most important file is the package.xml file, which contains information about the package, including its dependencies.

We also need to create a few Python files. The first one is the delivery agent itself. We can call it delivery_agent.py and put it in the src/delivery_agent directory. The contents of this file should look like this:

```
import rospy from std_msgs.msg
import String from delivery_agent.msg
import Task, Result

def delivery_agent():
  rospy.init_node('delivery_agent')
  pub = rospy.Publisher(
                '/delivery_result',
                  Result, queue_size=10)

  while not rospy.is_shutdown():
    task = rospy.wait_for_message('/task',
                                  Task)
    result = Result()
    result.status = "success"
    pub.publish(result)

if __name__ == '__main__':
  try:
    delivery_agent()
  except rospy.ROSInterruptException:
    pass
```

This file defines a ROS node called delivery_agent. This node subscribes to the / task topic, which is where it will receive information about tasks that need to be completed. It then completes the task and publishes the result to the /delivery_result topic. The second file we need is a message definition file. This goes in the msg/ directory and is called task.msg. The contents of this file should look like this: string id string description — string status. This file defines a message type called Task, which has two fields: an id and a description. The id field is used to uniquely identify the task, and the description field is used to provide a human-readable description of the task. The message also has a third field, called status, which is used to indicate the result of the task. The third and final file we need is a launch file. This goes in the launch/ directory and is called delivery_agent.launch. The contents of this file should look like this: ¡launch¿ ¡!– start delivery agent node –¿ ¡node pkg="delivery_agent" type="delivery_agent.py" name="delivery_agent"/¿ ¡/launch¿ This file simply launches the delivery_agent node defined in the previous file.

Once all of these files have been created, it is possible to build and run the package with the following commands: cd /catkin_ws catkin_make source devel/ setup.bash roslaunch delivery_agent delivery_agent.launch Now it is possible to see that the delivery_agent node is running and the results of its tasks being published to the / delivery_result topic.

III. Creating the Environment

When creating a delivery robot's action space in ROS, there are a few key things to keep in mind. First, the action space should be large enough to accommodate the robot's movements. Second, it should be designed in a way that allows the robot to avoid obstacles and reach its destination safely.

One way to create a delivery robot's action space is to use the Navigation Stack. The Navigation Stack is a set of tools that allow robots to move around autonomously while avoiding obstacles. It can be used to create an action space for a delivery

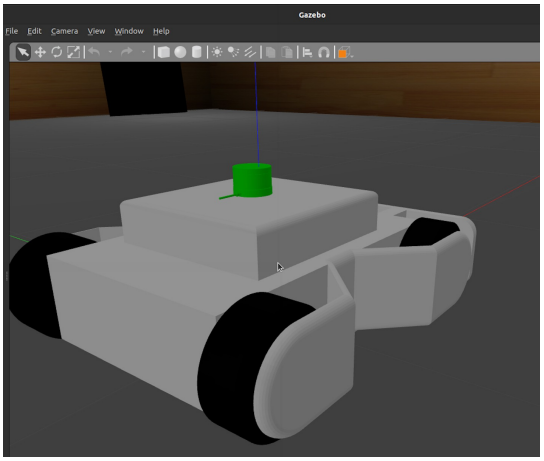

Fig. 1. Agent

robot by setting up a map of the area and providing the robot with instructions on how to navigate from one point to another.

Another way to create a delivery robot's action space is to use the MoveIt! Motion Planning Framework. The MoveIt! Framework is a set of tools that allow robots to plan and execute complex motions. It can be used to create an action space for a delivery robot by providing the robot with a set of waypoints to follow and setting up collision avoidance rules.

A. How We Accomplished To Create Our Action Space

- Simulation environment designed in the Gazebo.
- Models added to the model, model names inserted in model.stf file.
- The relevant mesh is imported and saved to be ready for application in the Gazebo environment.
- We downloaded almost 1000 models from Github and then selected some suitable models.
- Then we put these models in home/.gazebo/models folder. Gazebo detected them and allowed us to bring in the view.
- Then we created a world file where we called these selected models.

```python
import rospy
import actionlib
import tf
import roslib
import threading
from std_msgs.msg import String
from Move_Base_msgs.msg import MoveBaseAction, MoveBaseGoal

def callback(msg):
    global detection
    detection = msg.data


class Patrolz:

    def __init__(self):
        # Get a move_base action client
        self.client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
        self.client.wait_for_server()

    def set_goal_to_point(self, point):

        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "odom"
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose.position.x = point[0]
        goal.target_pose.pose.position.y = point[1]
        quaternion = tf.transformations.quaternion_from_euler(0.0, 0.0, point[2])
        goal.target_pose.pose.orientation.x = quaternion[0]
        goal.target_pose.pose.orientation.y = quaternion[1]
        goal.target_pose.pose.orientation.z = quaternion[2]
        goal.target_pose.pose.orientation.w = quaternion[3]

        self.client.send_goal(goal)
        wait = self.client.wait_for_result()
        if not wait:
            rospy.logerr("Action server not available!")
            rospy.signal_shutdown("Action server not available!")
        else:
            return self.client.get_result()


if __name__ == '__main__':
    rospy.init_node('patrolling')
    try:
        p = Patrolz()

        p.set_goal_to_point([-0.6942512966802, 8.2123017685, 1.593])
        p.set_goal_to_point([-0.744286853385, 8.2123017685, 1.593])
        p.set_goal_to_point([-0.744286853385, 10.0, 1.593])

    except rospy.ROSInterruptException:
        rospy.logerr("Something went wrong when sending the goal")
```

Fig. 2. reached.py

B. Possible Difficulties

IV. Reached.py Error Handling

One potential issue with this code is that it doesn't check for any errors when sending goals to move_base. So if the move_base server is down or not responding, this code will just keep trying to send goals forever without giving any error messages. To fix this, you could add some error checking to the set_goal_to_point function.

Another potential issue is that this code doesn't cancel any goals that have been sent to move_base before trying to send new ones. So if the robot is currently moving to a goal and the code tries to send it a new goal, the robot will just keep moving towards the first goal. To fix this, you could add a line of code to cancel any existing goals before sending new ones.

A. Navigation

```python
marker_size  = 50 #- [cm]
stopp = 0
add = -20
yaw = 3.14
n = 0
m = 0
a = 0
b = 0
c = 0

def callback(msg):
    global x, y, z
    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y
    z = msg.pose.pose.orientation.z

if get_pos == 0:
    odo_sub = rospy.Subscriber('/odom', Odometry, callback)
    get_pos = 1

# Get a move_base action client
rospy.init_node('patrolling')
client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
rospy.loginfo('Connecting to move_base...')
client.wait_for_server()
rospy.loginfo('Connected to move_base.')



def set_goal_to_point(point):

    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "odom"
    goal.target_pose.header.stamp = rospy.Time.now()
    goal.target_pose.pose.position.x = point[0]
    goal.target_pose.pose.position.y = point[1]
    quaternion = tf.transformations.quaternion_from_euler(0.0, 0.0, point[2])
    goal.target_pose.pose.orientation.x = quaternion[0]
    goal.target_pose.pose.orientation.y = quaternion[1]
    goal.target_pose.pose.orientation.z = quaternion[2]
    goal.target_pose.pose.orientation.w = quaternion[3]

    client.send_goal(goal)
    wait = client.wait_for_result()
    if not wait:
        rospy.logerr("Action server not available!")
        rospy.signal_shutdown("Action server not available!")
    else:
        return client.get_result()



def main():
    point = (-10, 10, 0.000)
    set_goal_to_point(point)
    print"1st patient reached"
```

Fig. 3. navigate.py

The code above is a simple Python script that sends a goal to the move_base action server. The goal is to move to a specific point in the world (specified by the x, y, and z coordinates). The quaternion parameter specifies the orientation of the robot at that goal point.

The script first subscribes to the odom topic to get the current position of the robot. It then uses this information to create a MoveBaseGoal object. This object specifies the desired goal pose of the robot. Finally, it sends this goal to the move_base action server and waits for a result. If successful, it prints out a message indicating that the goal was reached. Otherwise, it shuts down with an error message.

This code can be easily modified to send goals to different points in the world or to change the orientation of the robot at the goal point. For example, you could modify the quaternion

parameter to rotate the robot by 90 degrees before moving to the goal point. You could also add additional points to the list of points that the robot should visit. By doing this, you could create a simple patrol script that causes the robot to move between multiple points in the world.

B. *place.py*

```python
#!/usr/bin/env python
import rospy
import numpy as np
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import*


def carry_box():


    state = arm("robot::base_footprint","")
    state_msg = ModelState()
    state_msg.model_name = 'beer'
    state_msg.pose.position.x = state.link_state.pose.position.x
    state_msg.pose.position.y = state.link_state.pose.position.y
    state_msg.pose.position.z = 0.3
    grip_state = grip(state_msg)

def main():
    carry_box()


rospy.init_node('carry_box')
grip = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
rospy.wait_for_service('/gazebo/set_model_state')
arm = rospy.ServiceProxy('/gazebo/get_link_state', GetLinkState)
rospy.wait_for_service('/gazebo/get_link_state')
while not rospy.is_shutdown():
    main()
```

Fig. 4. place.py

The code above imports the necessary packages, defines the carry_box() function, and contains a main() function that calls carry_box().

The carry_box() function uses the GetLinkState service to get the current state of the robot's arm link. It then uses the SetModelState service to set the pose of a beer box model so that it is grippable by the robot's arm. The main() function simply calls carry_box() repeatedly until the ROS node is shut down. This code would allow a robot to pick up a beer box and carry it around.

C. *Detection.py & OpenCV*

The code below uses the OpenCV library to detect faces and eyes in real time. First, the required trained XML classifiers (for face and eye detection) are loaded. Then, frames from a camera are captured and converted to grayscale. The face_cascade classifier is used to detect faces in the grayscale image, and the eyes are detected using the eye_cascade classifier. Finally, rectangles are drawn around the detected faces and eyes, and the resulting image is displayed in a window.

A few things to note are the parameters passed to the detectMultiScale function (in both the face and eye detection). The first parameter is the grayscale image to be used for detection. The second parameter is the scale factor, which specifies how much the image size should be reduced at each image scale. A lower scale factor means that more faces/eyes will be detected (since smaller objects are more easily detected at a lower scale). The final two parameters are the minimum and maximum neighbors, which specify how many neighbors each candidate rectangle should have to be

```python
import cv2

# load the required trained XML classifiers
# https://github.com/Itseez/opencv/blob/master/
# data/haarcascades/haarcascade_frontalface_default.xml
# Trained XML classifiers describes some features of some
# object we want to detect a cascade function is trained
# from a lot of positive(faces) and negative(non-faces)
# images.
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# https://github.com/Itseez/opencv/blob/master
# /data/haarcascades/haarcascade_eye.xml
# Trained XML file for detecting eyes
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

# capture frames from a camera
cap = cv2.VideoCapture(0)

# loop runs if capturing has been initialized.
while 1:

    # reads frames from a camera
    ret, img = cap.read()

    # convert to gray scale of each frames
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detects faces of different sizes in the input image
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)

    for (x,y,w,h) in faces:
        # To draw a rectangle in a face
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,0),2)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img[y:y+h, x:x+w]

        # Detects eyes of different sizes in the input image
        eyes = eye_cascade.detectMultiScale(roi_gray)

        #To draw a rectangle in eyes
        for (ex,ey,ew,eh) in eyes:
            cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,127,255),2)

    # Display an image in a window
    cv2.imshow('img',img)

    # Wait for Esc key to stop


# Close the window
cap.release()

# De-allocate any associated memory usage
cv2.destroyAllWindows()
```

Fig. 5. detection.py

considered a face/eye (higher values mean that more neighbors are required).

In general, the face detection works well, but it is not perfect. Sometimes faces are not detected, or multiple faces are detected when there is only one person in the frame. The eye detection is also not perfect, and sometimes eyes are not detected or false positives are detected (e.g., an eyebrow may be detected as an eye). However, overall this code provides a good starting point for implementing face and eye detection in real time. OpenCV is a powerful tool for performing image analysis tasks such as detecting faces.

V. Robot Control Panel via Interface

A. *About the PyQt5*

Qt is a cross-platform software for designing graphical user interfaces and cross-platform applications that run on various software and hardware platforms, as well as embedded devices, with little or no changes to the underlying codebase while maintaining native capabilities and speed. [1] An interface (interface) is a boundary software and hardware unit that allows two systems to interact with or for a user to interact with a system. [2] Exchange can be between software, computer hardware, peripherals, people, and combinations. Some computer hardware devices, such as a touchscreen, can send and receive data via the interface. In contrast, others, such as a mouse or microphone, may only provide an interface to send data to a particular system. [2]

Many tools are used to design GUIs depending on the programming language used. For example, we have examined tools such as Tkinter, wxPython, and PyQt that we preferred since we use Python in our project. We preferred the PyQt tool to create an agent control interface due to its structure and because Ros comes with a Full Desktop installation.

PyQt is a set of Python bindings that allow us to develop a Qt application framework developed by OT Company. Qt is a graphical user interface development toolkit that supports multiple platforms. Moreover, it is a structure designed and developed with C++. Thanks to this PyQt layer, we can connect Qt to Python. As a result, PyQt acts as an intermediate binding layer here. Intermediate layers are responsible for enabling a library written in any language to be used in another language. Therefore, PyQt has a structure that combines all the advantages of Python and Qt.

Other features that make us prefer the Qt platform:

- It is an object-oriented application.
- Provides an easy-to-use user interface.
- It can work with many platforms such as Linux, Windows, macOS, and Android.
- It contains many modules and classes.
- Includes Qt Designer Tool.

B. Creating an Interface and Adding Tools to the Interface

To develop the control panel, first, an interface window was created. We used Visual Studio Code as IDE to create an interface window with PyQt5 and implement other applications. Then we imported the sys modules and the QWidget modules. During the development phase, other modules can be imported within the scope of the features expected from the panel. However, these two modules are sufficient to create a simple user interface window.

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

Fig. 6. Main Function

An application object, QApplication, is then created. A window is created with QWidget. After these operations, additional operations are performed to develop and edit the panel, and then the operations required to display the window are applied. Here, using the exec_() method, displaying the window continuously is performed by providing the main loop.

C. Qt Designer Tool

The design and development process of the interface panel was carried out through QtDesigner. After the panel's appearance was completed, it was saved as a file with a "ui" extension. Then this file was deployed as a python file with the pyuic5 command over the terminal. Then, by making arrangements on this file, the panel was enabled to work with ROS. Finally, by making the created python node executable on the terminal, the node was made suitable for controlling the robot.

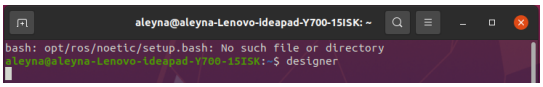

Fig. 7. Terminal Command to start Qt

In order to run the Qt designer, it is sufficient to give the designer command over the terminal. The previously saved interface studies can be opened again, and interface developments can be made from scratch.

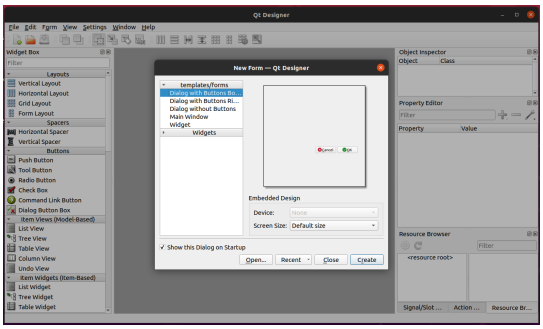

Fig. 8. The Qt Interface / Starting the Design

The first operation that was expected to be carried out via the panel was that the robot could be steered with the control buttons. First, an empty window is created in the main window option.

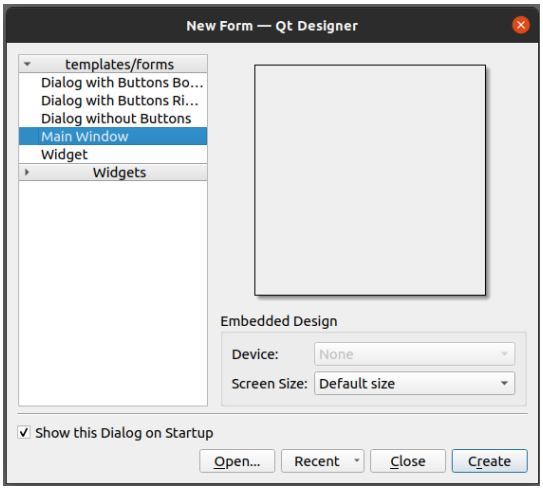

Fig. 9. Opening the Project File

The speed message will be published every time the agent is checked via B-buttons, we also aimed to display this via the panel. This process is basically a publishing process. In addition, with the subscription feature, the speed information of the robot can be entered via the panel. Object name allows

programs to recognize all elements on the panel, including the window. The title of the window can also be changed easily via the application.

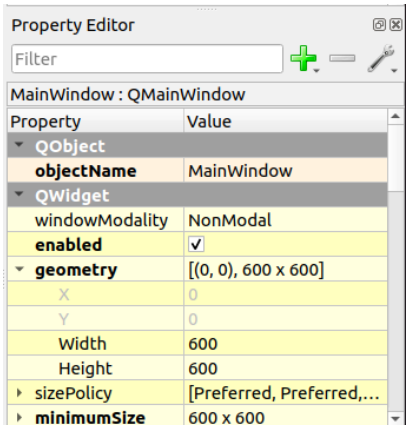

Fig. 10. Object Name

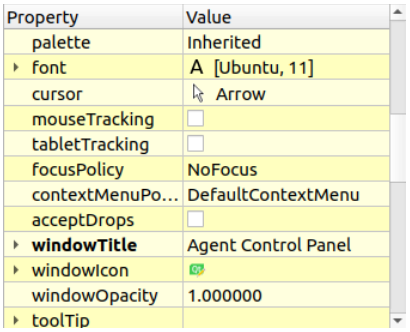

Fig. 11. Defining the title of the Window

The next step is to add the buttons to the window and leave a layout. We used the Grid Layout to do this. Then, by adding labels, we ensured that the processed panel sections were distinguishable from each other.

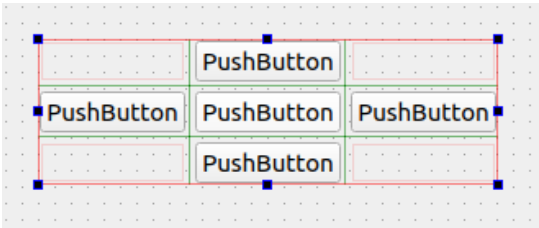

Fig. 12. Organizing Buttons' Layout

In the next step, we created the view we wanted to obtain by using text labels and Line edits using Form Layout so that the linear and angular speeds of the robot could be displayed. Then we performed the same process to show the robot's position in terms of x and y. By enclosing the entire screen in the grid layout, we ensured that the views we created could remain in regular order.

In the first stage, we wanted to see the robot's current speed and position information on the screen, so we gave Line edits only reading permission. One of the operations we wanted to do, but took the comment line because we had a problem with QtWidgets, was to display the image from the agent's camera live through the interface.

D. Arrangements were made for the interface to work with ROS

For the created GUI file, a file named ui has been created in the driver folder of the package. The ui file created with the "pyuic5 -x agent_control.ui -o agent_control.py" command is converted to a python file in this folder. This python file has been moved to the src folder where other python programs are located.

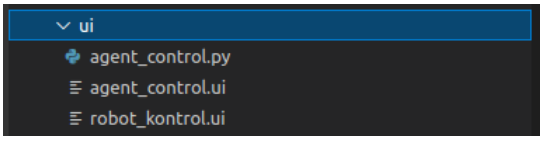

Fig. 13. File Arrangements

Arrangements have been made to perform ROS-related operations on this python file. The first line, "#!/usr/bin/env python3," was added to the code so that these edits work and ROS can recognize this file as a python file.

Then, the modules required for ROS-related operations are imported. Since we are coding with Python, rospy is imported. We used Twist under "geometry.msgs.msg" because we wanted to publish speed information on the panel. We also imported the Odometry under "nav.msgs.msg" to our code to get the robot's location information. Thus, we have added the necessary library and packages.

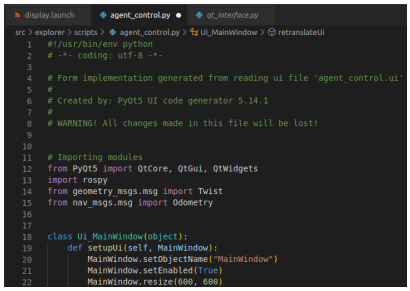

Fig. 14. Importing Modules and Libraries

There are panel adjustments made according to the tags we have determined in the "setupUi" function in the code. After initializing the node in "reTranslateUi," we created a 10 queue sized publisher of Twist type on "cmd_vel ."Then we created a Twist-type speed message. Also, since we need a subscription, we defined a subscriber with "rospy. Subscriber" that uses the Odometry message from the "Odom" topic and enters the function we define as odomCallBack each time a subscription is made.

The odomCallBack function takes two parameters, self, and incoming message. The operation we want to perform in this function is to print the incoming odom data to x and y. We used the ”round” method to make these values observable.

```
def odomCallback(self,vel_message):
    self.line_loc_x.setText(str(round(vel_message.pose.pose.position.x)))
    self.line_loc_y.setText(str(round(vel_message.pose.pose.position.y)))
```

Fig. 15. odomCallBack Function

In the previous stage, we defined the buttons via QtDesigner, but we did not define a function or broadcast a signal to these buttons. In order to perform this operation, we used ”clicked.connect” and functions that define the operations performed when the button is pressed.

```
# Defining Button Actions
self.button_stop.clicked.connect(self.agent_stop)
self.button_forward.clicked.connect(self.agent_go_forward)
self.button_back.clicked.connect(self.agent_go_back)
self.button_left.clicked.connect(self.agent_turn_left)
self.button_right.clicked.connect(self.agent_turn_right)
```

Fig. 16. Enabling Button Actions

Then, we defined the button functions that will allow the robot to be intervened by changing the published linear and angular velocity values. Then we have this speed message published with ”self.pub .”The printing of incoming speed message information was also carried out within the function. Based on the operations we performed in this function, functions were defined for other buttons by making adjustments to the angular and linear velocity values.

```
rospy.init_node("control_panel") # Initializing the node
self.pub = rospy.Publisher("cmd_vel",Twist,queue_size=10) # Creating the publisher
self.vel_message = Twist() # Creating the Velocity Message
rospy.Subscriber("odom",Odometry(),self.odomCallback) # Define a subscriber
```

Fig. 17. Publishing and Subscribing Operations

In order to facilitate robot control, linear velocity values of forward and reverse buttons were defined, and angular velocity values were set to zero as a constant.

```
# Setting Initial Line Edit Values
self.line_acc.setText(str(0.0))
self.line_lineer.setText(str(0.0))
self.line_loc_x.setText(str(0.0))
self.line_loc_y.setText(str(0.0))
```

Fig. 18. Setting Default Line Values to Zero

On the other hand, we made the opposite definition for the buttons used for the right and left directions. In the stop button, it was defined as zero in both values. In addition, we set the initial value to zero for the Line Edits we specified. Finally, we made this node executable with ”chmod +x.”

As a result, we created a panel where we can manually control our robot with buttons and observe its current position and angular and linear velocity values that change due to interference with the buttons.

VI. Conclusion

The delivery robot developed in ROS makes us learned that how to first developing ROS project and how to install some necessary ROS packages. However, the biggest obstacle we encountered during this project is that we don’t have enough time to debug our code . The other thing we want to say is Figure6:camera.py that, after this project, we have a better understanding of robot operating systems and image processing.In addition to that we know how to manipulate detection algorithms and incase autonomous navigation does not respond with the help of control panel we are be able to manually manipulate the agent in 4 dynamic directions and 1 static position which are right, left, up,down and stop.An on-demand delivery robot can be simulated in a simula- tion environment using Gazebo, Python, and ROS. This allows for the testing and development of the on-demand delivery robot without having to worry about potential damage to the robot or its surroundings. Additionally, this setup can be used to study and analyze the behavior of the on-demand delivery robot in different scenarios.

Gazebo is a 3D robotic simulator that allows for modeling and simulation of complex robots and their environments. Python is a programming language that can be used to control Gazebo and ROS. ROS is a framework for communication between different parts of a robotic system, including controllers, sensors, and actuators. By using these three tools, it is possible to create a simulation of an on-demand delivery robot and its environment. This setup can be used to test and debug the on-demand delivery robot before it is deployed in the real world. Additionally, this setup can be used to study the behavior of the on-demand delivery robot in different scenarios. For example, the robot’s movement might be studied in a cluttered environment or it might be tested for its ability to navigate stairs.

Simulating an on-demand delivery robot can be useful for a variety of purposes. It can be used to test and debug the robot before it is deployed in the real world. Additionally, it can be used to study the behavior of the robot in different scenarios. This setup can also be used to develop new algorithms or controllers for the on-demand delivery robot. By using Gazebo, Python, and ROS, it is possible to create a realistic simulation of an on-demand delivery robot and its environment. This simulation can be used to test and debug the on-demand delivery robot before it is deployed in the real world. Additionally, this setup can be used to study the behavior of the on-demand delivery robot in different scenarios.

References

[1] https://en.wikipedia.org/wiki/Qt_(software)
[2] https://en.wikipedia.org/wiki/Interface_(computing)
[3] https://www.steadfast.net/blog/almost-everything-you-need-know-about-raid
[4] https://github.com/osrf/gazebo_models