ROS Noetic Specification and using ROS to make a hypothetical robot

Fatema Mirza

Intelligent System and Robotics

Student ID: 77264327

LEEDS BECKETT UNIVERSITY

ROS Noetic Specification and using ROS to make a hypothetical robot

by

Fatema Mirza

Student ID: 77264327

Project duration: February 1, 2021 – May 10, 2021

Thesis committee: Dr. Abiodun Yusuf, Leeds Beckett University, Supervisor
Prof. David Love, Leeds Beckett University

This report is confidential and cannot be made public until May 10, 2021.

An electronic version of this report will be available at https://github.com/fatimamirza94/.

Abstract

This report will provide an explanation on how ROS Noetic works and the different components involved with ROS Noetic. Furthermore, a hypothetical robot named roomba_robot will be developed with full documentation on the source code as well as how to install and run it.
The documented source code is available at: https://github.com/fatimamirza94/roomba_robot.git

Fatema Mirza
Leeds, May 2021

Contents

List of Figures

1

Delving into the ROS Environment

This report will look into the ROS environment and using the knowledge gained in through that, will create a hypothetical roomba robot that is capable of sensing dirt, travelling to a position defined by the user as well as travel back to the charging station. For highly valued customers or users, the roomba robot will also allow viewing of pictures taken during its snapshot while traversing.

1.1. Installing ROS Noetic

ROS Noetic was used as the operating system. This was installed on a virtual machine of the developers computer. To install ROS-Noetic on the virtual machine the following commands were used.

Setting up the sources list

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"> /etc/apt/sources.list.d/ros-latest.list'

Setting up the keys

sudo apt-key adv –keyserver 'hkp://keyserver.ubuntu.com:80' –recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C65

File installation using sudo

sudo apt update
sudo apt install ros-noetic-desktop-full

Running the installation using a bash script

source /opt/ros/noetic/setup.bash
echo "source /opt/ros/noetic/setup.bash" » /.bashrc source /.bashrc

Installing dependencies

sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential
sudo apt installForis python3-rosdep
sudo rosdep init
rosdep update

Checking if ros has been installed correctly

To ensure whether ROS has been installed correctly, run roscore in the terminal to initiate the ROS Master which is the backbone responsible for for communication between nodes where only one instance of it can run at a time. The command will return the following:

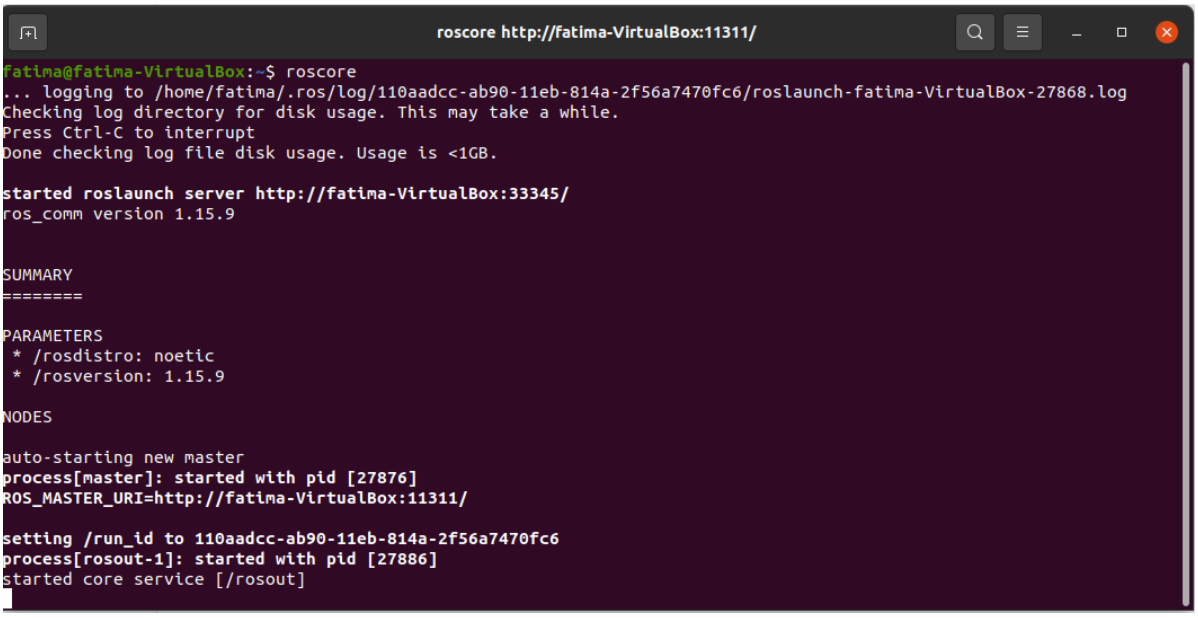


Figure 1.1: Roscore command

To download any additional ros-noetic packages

To download any additional ros-noetic packages, use the following command:
sudo apt install ros-noetic-usb-cam

1.1.1. Catkin Workspace

Catkin workspace is a directory to ensure smooth build, running and installation of the ROS projects. Run the following commands in the terminal at the level of home directory to create the catkin workspace.
mkdir -p /catkin_ws/src
cd /catkin_ws/
mkdir src
catkin_make

The command catkin_make will create the CMakeLists in the src directory. Additional 'build' and 'devel' folder will be created and inside the 'devel' folder it can be seen that there are now several setup.*sh files (ROS.org, 2021c).

1.2. Creating ROS Package

To create a ROS package, enter the terminal at the level of the src folder of the catkin_ws. Enter the following command in the terminal with dependencies to create the package roomba_robot .

catkin_create_pkg roomba_robot roscpp rospy std_msgs

A node is an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes to publish or subscribe to a Topic or provide or use a Service (ROS.org, 2021f). To ensure a proper structure of the ROS project, the nodes will be placed in a scripts directory (created by the developer) in the roomba_robot folder.

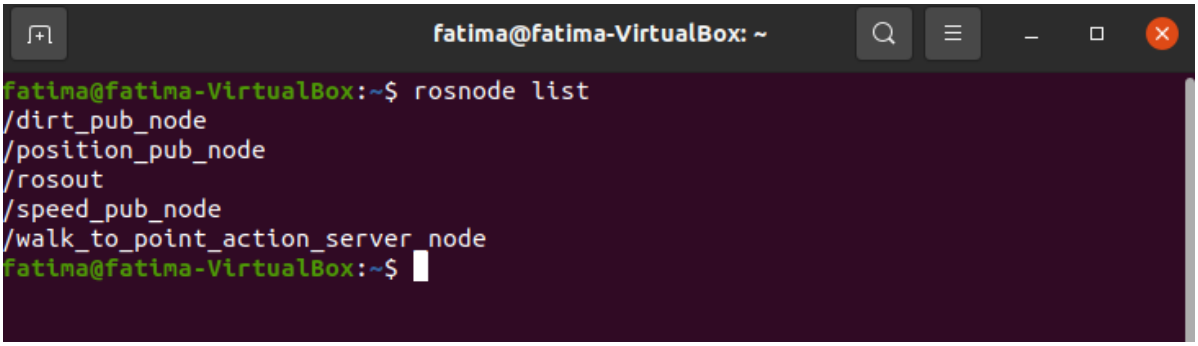


Figure 1.2: List of Current Nodes in the ROS project for roomba_robot

1.3. Publishers and Subscribers

As mentioned in the previous section, nodes can be publishers or subscribers. Publishers send messages with a specific topic and subscribers listen for that particular topic in the stream. For the subscriber to listen to the topic, it first has to subscribe to the topic using the call back functions.

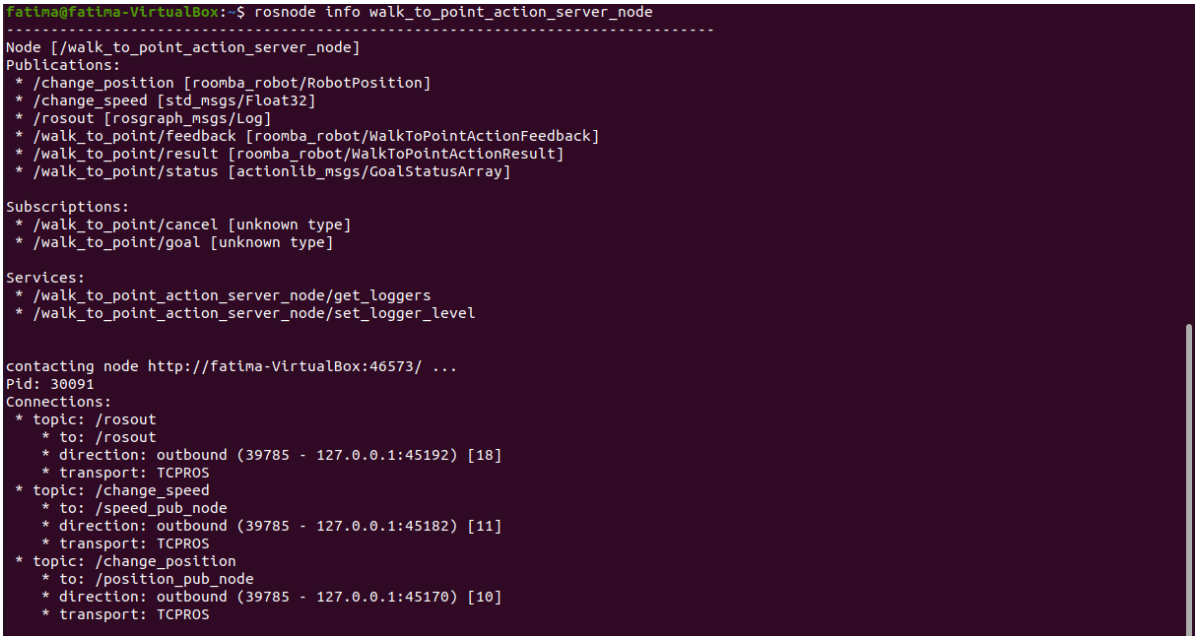


Figure 1.3: A node publishing and subscribing to events

1.4. ROS Parameter Server

A parameter server is a shared, multi-variate dictionary that is accessible via network APIs used by nodes to store and retrieve parameters at runtime. This means that paramater lists can be changed during runtime, which is useful for testing and observing the impact of those changes real time.

```
fatima@fatima-VirtualBox:~$ rosparam list
/initial_post_x
/initial_post_y
/max_cordinate
/min_cordinate
/robot_max_speed
/robot_name
/rosdistro
/roslaunch/uris/host_fatima_virtualbox__41297
/rosversion
/run_id
```

Figure 1.4: List of ROS parameters

ROS parameters can be changed and retrieved using the get set method. An example is shown below where the value was initially 0, and is now set to 2.

```
fatima@fatima-VirtualBox:~$ rosparam get /initial_post_x
0
fatima@fatima-VirtualBox:~$ rosparam set /initial_post_x 2
fatima@fatima-VirtualBox:~$ rosparam get /initial_post_x
2
```

Figure 1.5: List of ROS parameters

Alternatively, parameter server variables can be saved in a yaml file to access and change the variables from that file using the following command:
rosparam dump param.yaml

1.5. ROS Launch Files

roslaunch - which already includes the roscore - is a tool for launching multiple ROS nodes locally and remotely via SSH and set parameters on the Parameter Server (ROS.org, 2021d). roslaunch limits the number of terminals needed for the codes - as it can be done in one go rather than requiring to open a new terminal for each node. For roslaunch to work successfully, the package must be sourced through the command typed in the terminal:
source devel/setup.bash

```
fatima@fatima-VirtualBox:~/catkin_ws$ source devel/setup.bash
fatima@fatima-VirtualBox:~/catkin_ws$ roslaunch roomba_robot roomba_robot.launch
... logging to /home/fatima/.ros/log/fb0f5070-abff-11eb-814a-2f56a7470fc6/roslaunch-fatima-VirtualBox-30063.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://fatima-VirtualBox:41297/

SUMMARY
========

PARAMETERS
 * /initial_post_x: 0.0
 * /initial_post_y: 0.0
 * /max_cordinate: 100.0
 * /min_cordinate: -100.0
 * /robot_max_speed: 0.5
 * /robot_name: roomba_robot
 * /rosdistro: noetic
 * /rosversion: 1.15.9

NODES
  /
    dirt_pub_node (roomba_robot/dirt_pub_node.py)
    position_pub_node (roomba_robot/position_pub_node.py)
    speed_pub_node (roomba_robot/speed_pub_node.py)
    walk_to_point_action_server_node (roomba_robot/walk_to_point_action_server.py)

auto-starting new master
process[master]: started with pid [30071]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to fb0f5070-abff-11eb-814a-2f56a7470fc6
process[rosout-1]: started with pid [30081]
started core service [/rosout]
process[dirt_pub_node-2]: started with pid [30088]
process[speed_pub_node-3]: started with pid [30089]
```

Figure 1.6: Using roslaunch

1.5.1. ROS Bag files

A bag is a file format in ROS for storing ROS message data which can subscribe to one or more ROS topics, and store the serialized message data in a file as it is received and playback at a later time (ROS.org, 2021b).

```
fatima@fatima-VirtualBox:~$ rosbag record -a -o test.bag
[ INFO] [1620042004.871548753]: Subscribing to /rosout_agg
[ INFO] [1620042004.879317053]: Subscribing to /rosout
[ INFO] [1620042004.879483686]: Recording to 'test_2021-05-03-13-40-04.bag'.
[ WARN] [1620042004.879567097]: Less than 5 x 1G of space free on disk with 'test_2021-
05-03-13-40-04.bag.active'.
[ INFO] [1620042004.886500974]: Subscribing to /dirt
[ INFO] [1620042004.894942250]: Subscribing to /position
[ INFO] [1620042004.905506203]: Subscribing to /change_speed
[ INFO] [1620042004.912621644]: Subscribing to /change_position
[ INFO] [1620042004.919841065]: Subscribing to /walk_to_point/status
[ INFO] [1620042004.922893978]: Subscribing to /walk_to_point/result
[ INFO] [1620042004.925496217]: Subscribing to /walk_to_point/feedback
[ INFO] [1620042004.928089893]: Subscribing to /speed
```

Figure 1.7: Using rosbag to record information

```
fatima@fatima-VirtualBox:~$ rosbag play test.bag
[ INFO] [1620042543.228833356]: Opening test.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1614798199.919378   Duration: 10.081344 / 61.000444
 [RUNNING]  Bag Time: 1614798199.922469   Duration: 10.084436 / 61.000444
 [RUNNING]  Bag Time: 1614798200.023503   Duration: 10.185469 / 61.000444
 [RUNNING]  Bag Time: 1614798200.120236   Duration: 10.282203 / 61.000444
 [RUNNING]  Bag Time: 1614798200.124647   Duration: 10.286614 / 61.000444
 [RUNNING]  Bag Time: 1614798200.224796   Duration: 10.386763 / 61.000444
 [RUNNING]  Bag Time: 1614798200.322380   Duration: 10.484347 / 61.000444
 [RUNNING]  Bag Time: 1614798200.324592   Duration: 10.486558 / 61.000444
 [RUNNING]  Bag Time: 1614798200.424961   Duration: 10.586927 / 61.000444
 [RUNNING]  Bag Time: 1614798200.522283   Duration: 10.684250 / 61.000444
 [RUNNING]  Bag Time: 1614798200.525929   Duration: 10.687895 / 61.000444
 [RUNNING]  Bag Time: 1614798200.626778   Duration: 10.788745 / 61.000444
 [RUNNING]  Bag Time: 1614798200.719633   Duration: 10.881600 / 61.000444
```

Figure 1.8: Using rosbag to playback recorded data

1.6. ROS Services

The publish / subscribe model is useful communication model, but it is not suitable for RPC request / reply interactions. Request / reply is done via a Service, which is defined by a pair of custom messages built as per the requirement of the project: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Services are defined using srv files, which are compiled into source code by a ROS client library (ROS.org, 2021e). The .srv file will contain a request, and the corresponding response. In the case of this roomba, turn_degrees is a angle requested for the file name, whereas the image is the response message. A directory named srv is created and the following dependencies are added in the package.xml file.
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>

A .srv file is created and the service and dependencies are added in the CMakeList file.

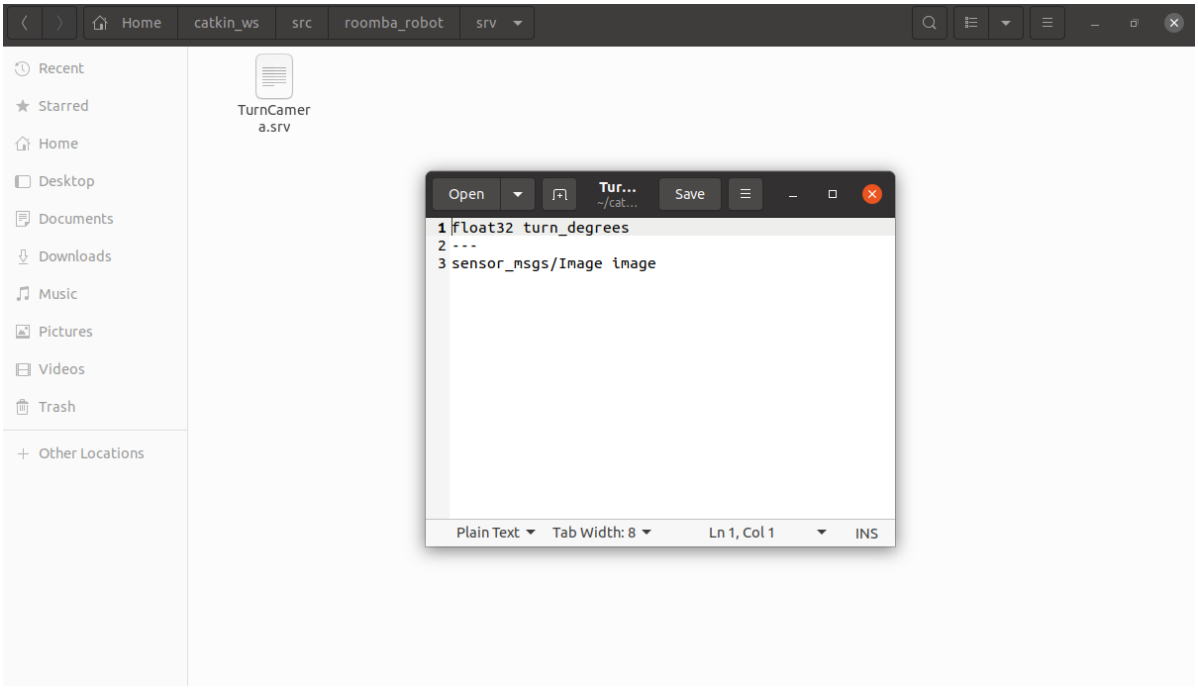


Figure 1.9: Creating an srv file

```
63 ## Generate services in the 'srv' folder
64  add_service_files(
65    FILES
66    TurnCamera.srv
67  )
68
69 ## Generate actions in the 'action' folder
70  add_action_files(
71    FILES
72    WalkToPoint.action
73  )
74
75 ## Generate added messages and services with any dependencies listed here
76  generate_messages(
77    DEPENDENCIES
78    std_msgs
79    actionlib_msgs
80    sensor_msgs
81    geometry_msgs
82  )
```

Figure 1.10: CMakeList File to reflect the changes for services

```
fatima@fatima-VirtualBox:~/catkin_ws$ rosservice list
/control_node/get_loggers
/control_node/set_logger_level
/dirt_pub_node/get_loggers
/dirt_pub_node/set_logger_level
/position_pub_node/get_loggers
/position_pub_node/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
/speed_pub_node/get_loggers
/speed_pub_node/set_logger_level
/turn_camera
/turn_camera_service_node/get_loggers
/turn_camera_service_node/set_logger_level
/walk_to_point_action_server_node/get_loggers
/walk_to_point_action_server_node/set_logger_level
```

Figure 1.11: Checking that the rosservice has been included in the project

1.7. ROS Action

ROS Action refers to a standardized platform for interfacing with preemptable tasks such as moving the roomba to a particular place or returning to base ROS.org, 2021a. From the original documentation, the following image is copied to best describe the process of ROS Action.

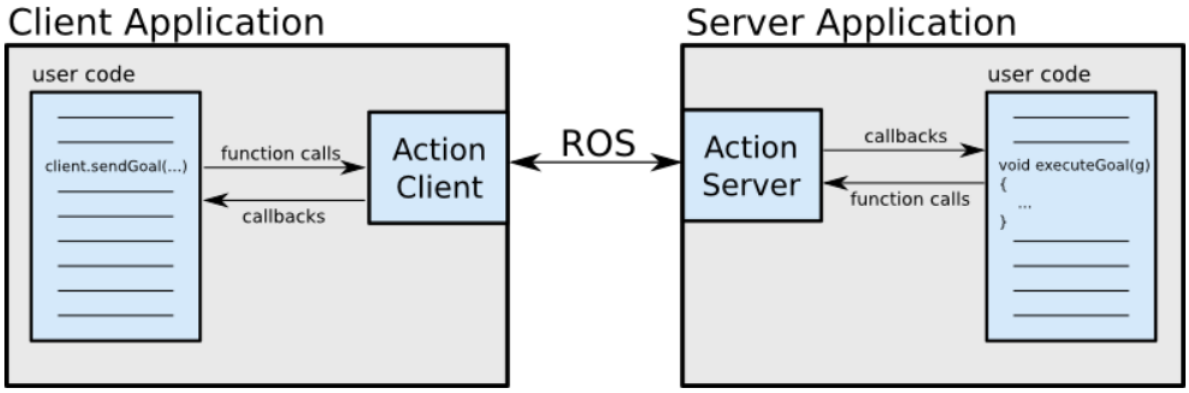


Figure 1.12: Action server and client at work

For the client and server to communicate, custom messages are defined - action specification. This defines the Goal, Feedback, and Result messages with which clients and servers communicate in the .action file:

Goal Goal aims to complete tasks using actions, the goal is a message that is sent to an Action-Server by an ActionClient and in the case, it is the destination to reach by the roomba.

Feedback Feedback returns the incremental progress of the goal to the ActionClient- that is how much distance has been covered till now to reach the destination.

Result A result is generated from the ActionServer to inform the ActionClient that the goal has been completed - that is how long the roomba took to traverse from the previous current position to the specified final destination.

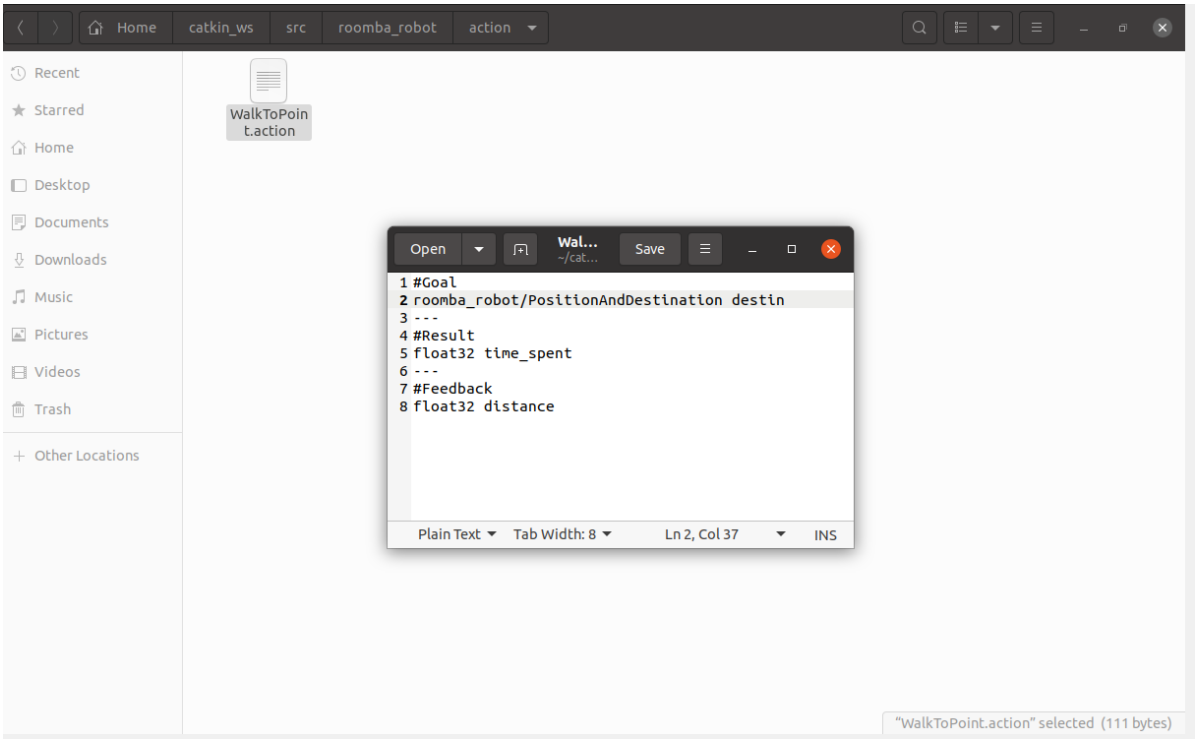


Figure 1.13: Action file

To initiate the messages of action, add the following dependencies in the package.xml similar to what was done for the service files. file.
<build_depend>actionlib</build_depend>
<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>

Generate the action file dependencies in the CMakeLists.

```
63 ## Generate services in the 'srv' folder
64  add_service_files(
65    FILES
66    TurnCamera.srv
67  )
68
69 ## Generate actions in the 'action' folder
70  add_action_files(
71    FILES
72    WalkToPoint.action
73  )
74
75 ## Generate added messages and services with any dependencies listed here
76  generate_messages(
77    DEPENDENCIES
78    std_msgs
79    actionlib_msgs
80    sensor_msgs
81    geometry_msgs
82  )
```

Figure 1.14: CMakeList File to reflect the changes for actions

Ensure that the service has been correctly defined, check whether the goal, feedback and results have been added to the rostopic.

```
fatima@fatima-VirtualBox:~$ rostopic list
/change_position
/change_speed
/dirt
/position
/rosout
/rosout_agg
/speed
/walk_to_point/cancel
/walk_to_point/feedback
/walk_to_point/goal
/walk_to_point/result
/walk_to_point/status
```

Figure 1.15: Checking to ensure that services has been added successfully

2

Roomba_Robot Overview

The previous chapter introduced all the terminologies related to ROS programming and provided a description for them. Utilizing those core ideologies a new robot named Roomba_Robot has been developed which has been briefly touched upon in the previous chapter.

2.1. Roomba_Robot Description & Functionalities

Roomba_Robot is a robot which has a hypothetical dirt sensor that can detect the amount of dirt at the current position and inform the user about the dirt amount as well as where it is situation in a 2D plane at the moment. It is assumed that if the sensor has detected dirt it picks it up. Furthermore, the user can direct the robot to walk to a particular position (specified in a 2D coordinates, as for now the roomba can only traverse on the ground and has no concept of height) and also direct it to walk up to the charging base. Furthermore, an additional feature is present for special customers - that is those users can view the pictures taken by the roomba (it is assumed that the roomba has a hypothetical pictures using a hypothetical camera and stores it in the images folder). It has to be noted for user discrepancy that the camera is not really an actual camera and is just assumed to be present and has not been coded. However, the portion where the user can view the pictures stored in the roomba has been coded.

To simplify the functionalities the roomba can perform:

1. Detect dirt amount and report to the user.
2. Detect current position in a 2D plane and report to the user.
3. Traverse to position in a 2D plane and report to the user to the distance and time taken to reach there.
4. Return back to charging station and report to the user to the distance and time taken to reach there.
5. Can retrieve images stored in the directory based on the file specified by the user which is a secret functionality.

2.2. Roomba_Robot Nodes, Services & Action

The nodes are the files which are in the scripts folder.

1. control_node - a subscriber node that controls using user input from the terminal. It subscribes to dirt, position and speed to display to the user the dirt amount, the current position and the distance and time needed to traverse a requested location.
2. dirt_pub_node - only a publisher node that publishes the amount of dirt randomly.
3. position_pub_node - simultaneously a publisher and subscriber node that stores and deals with the position data. This node publishes the topic position and subscribes to the topic of change_position.

4. speed_pub_node - simultaneously a publisher and subscriber node that stores and deals with the speed data. This node publishes the topic speed and subscribes to the topic of change_speed.
5. turn_camera_client - this is a service node that assists the user in asking the roomba to retrieve which image by asking for the image name in the format of angles. This node sends this information to the server to request this image to be shown.
6. turn_camera_service - this is a service node that accepts the requests send forward by the client for viewing a particular image and responds by displaying the image on the terminal screen.
7. walk_to_point_action_server_node - is an action server nodes that assists the roomba is walking to a point specified by the user as well as returning to charging station. it operates by publishing change_position, change_speed, walk_to_point/feedback, walk_to_point/result and walk_to_point/status and subscribing to walk_to_point/cancel and walk_to_point/goal.

2.3. Roomba_Robot Topics

1. change_position
2. change_speed
3. dirt
4. position
5. speed
6. walk_to_point/cancel
7. walk_to_point/feedback
8. walk_to_point/goal
9. walk_to_point/result
10. walk_to_point/status

2.4. Roomba_Robot Installation & Running

1. Place the package roomba_robot in the src folder of catkin_ws.
2. Make the files inside the scripts folder of the roomba_robot executable by running the following command in the terminal: chmod +x *.py so that they can be launched by roslaunch.
3. Check if the install_dependencies.sh file exists in the root folder of the package roomba_robot. If it does not exist, it has to installed using the following commands typed in the terminal:
   chmod +x *.sh
   ./install_dependencies.sh

   This will cause the virtual machine to restart - added as a note so that the user does not panic and think the program has crashed.
4. Source the project by using the following command at catkin_ws level:
   source devel/setup.bash
5. Build dependencies for the package at the same catkin_ws level by using following command:
   catkin_make

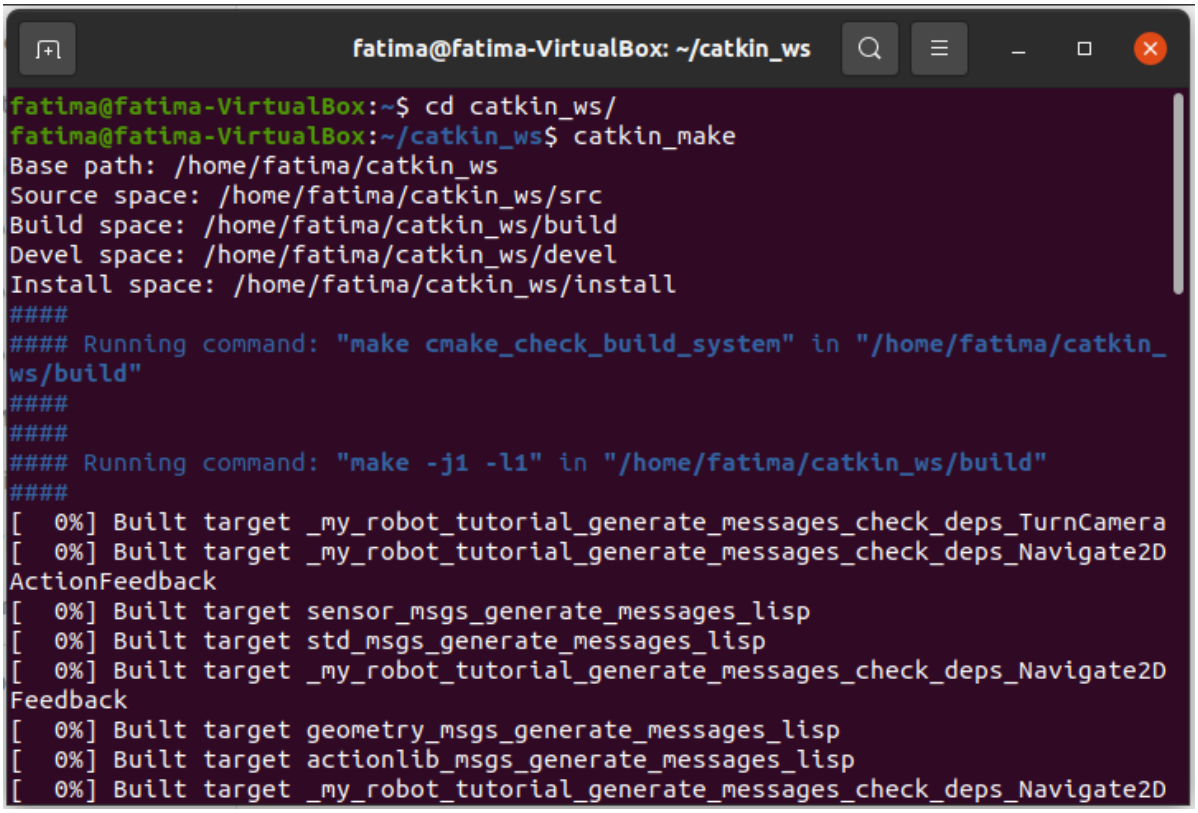


Figure 2.1: Successful catkin_make

6. Once catkin_make has been successfully executed, at the same level catkin_ws source the project again using the following command:
   source devel/setup.bash

7. Launch roomba_robot using the launch file named roomba_robot.launch by using the following command:
   roslaunch roomba_robot roomba_robot.launch

```
fatima@fatima-VirtualBox:~/catkin_ws$ roslaunch roomba_robot roomba_robot.laun
ch
... logging to /home/fatima/.ros/log/9ab9ce08-ac27-11eb-814a-2f56a7470fc6/rosl
aunch-fatima-VirtualBox-33645.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://fatima-VirtualBox:40167/

SUMMARY
========

PARAMETERS
 * /initial_post_x: 0.0
 * /initial_post_y: 0.0
 * /max_cordinate: 100.0
 * /min_cordinate: -100.0
 * /robot_max_speed: 0.5
 * /robot_name: roomba_robot
 * /rosdistro: noetic
 * /rosversion: 1.15.9
```

Figure 2.2: Successful roslaunch

8. Once the project has successfully launched, open a new terminal at the level catkin_ws and source the project again using the following command:
   source devel/setup.bash

9. To control the roomba_robot and view the menu of options, type in the following command: rosrun roomba_robot control_node.py
   The user inputs have all been validated to ensure that wrong inputs are not accepted and the user will be prompted for the correct inputs.

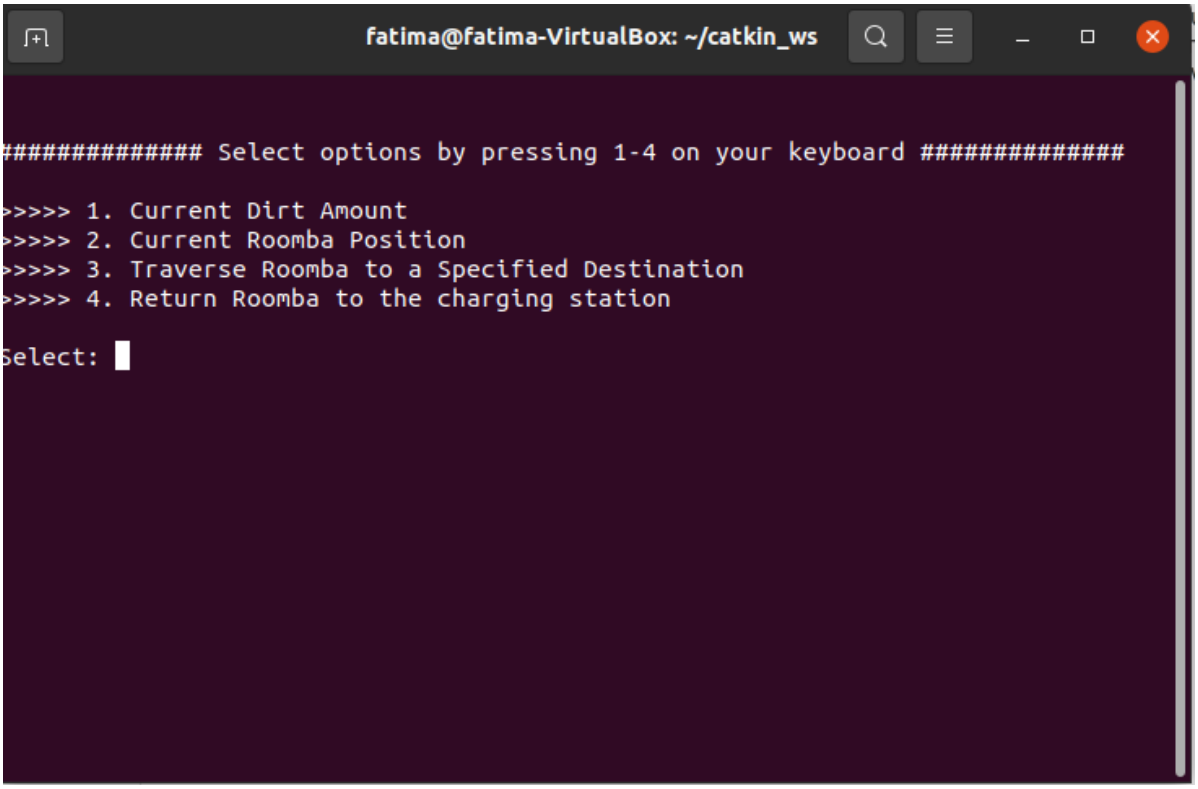


Figure 2.3: Successful rosrun

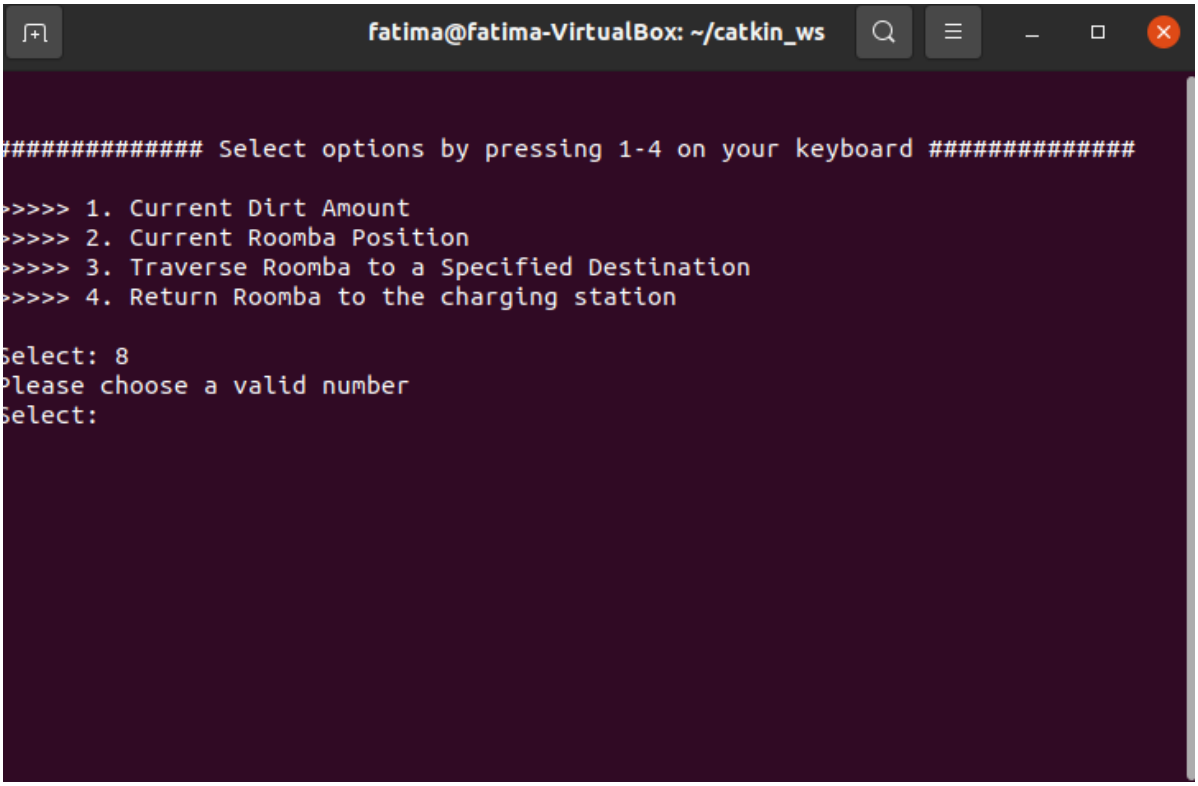


Figure 2.4: Validating input example

10. User can now play with the roomba as the user prefers.
11. To access the secret service for retrieving images stored in the roomba_robot, do the following:
    (a) Open a new terminal at the level catkin_ws and source the project again using the following command:
    source devel/setup.bash
    (b) Run the Server service using the following command:
    rosrun roomba_robot turn_camera_service.py

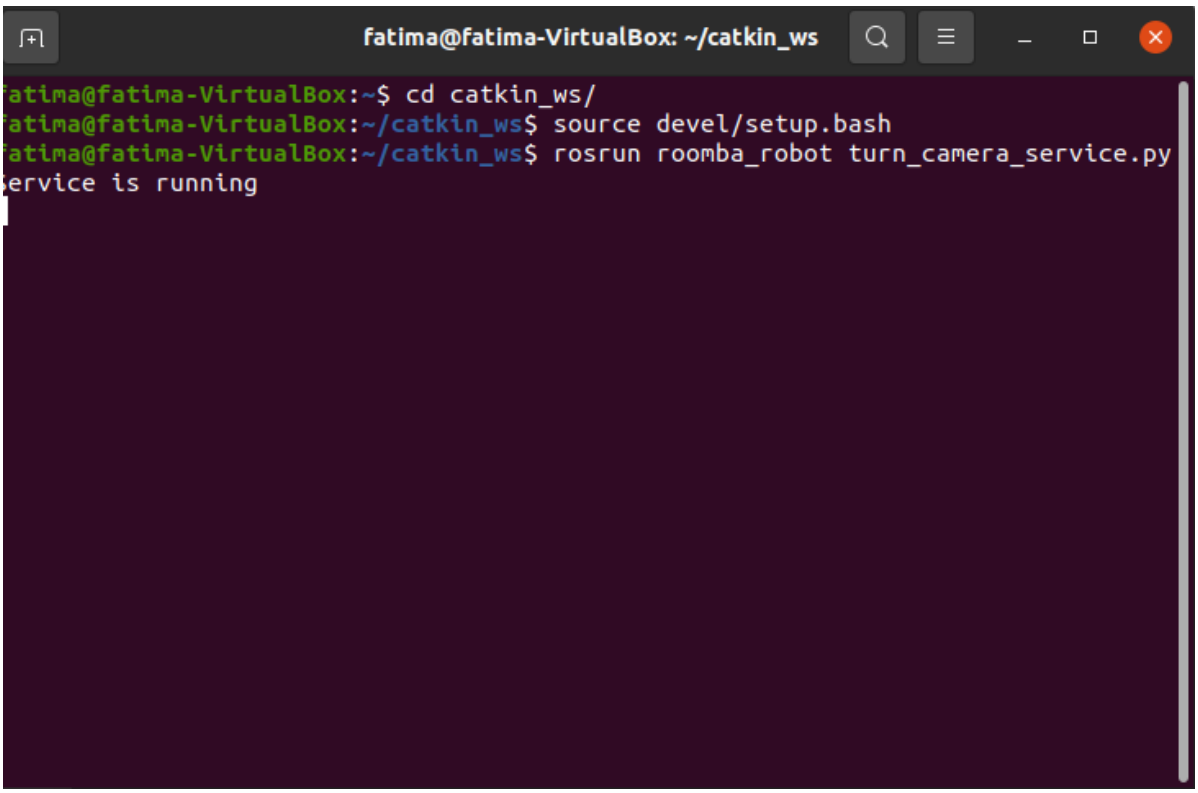


Figure 2.5: Successful service server running

    (c) Open a new terminal at the level catkin_ws and source the project again using the following command:
    source devel/setup.bash
    (d) Run the Client service using the following command:
    rosrun roomba_robot turn_camera_client.py

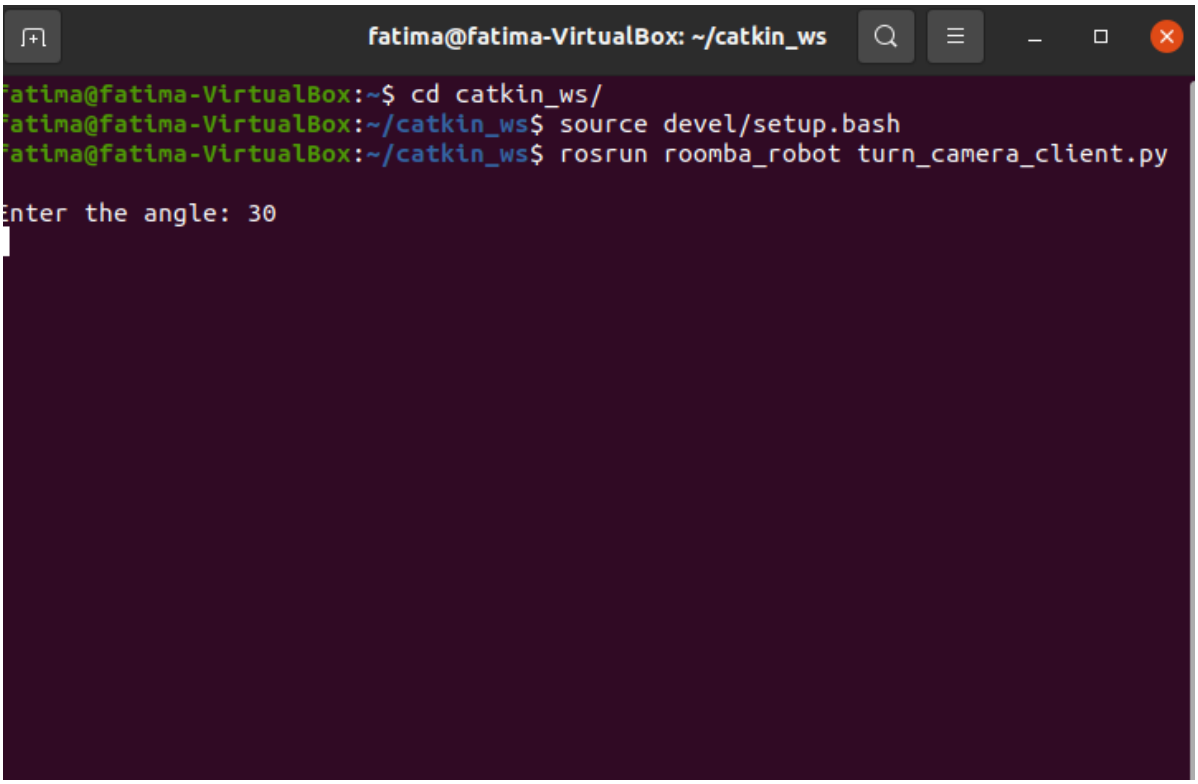


Figure 2.6: Successful service client running

3

Full Code Specification

The codes have been arranged in alphabetical orders as they appear in the folders. The codes have been fully commented, documented and explained with reasoning for the logic behind it and is available at - https://github.com/fatimamirza94/roomba_robot.git.

Note: Check out the repository for a clearer explanation as the screenshots of the code might not be clear.

3.1. Action files

Action files are placed in the action folder and there is only 1 action file named: WalkToPoint.action

WalkToPoint.action

```
1  #Goal
2  roomba_robot/PositionAndDestination destin
3  ---
4  #Result
5  float32 time_spent
6  ---
7  #Feedback
8  float32 distance
9
```

Figure 3.1: WalkToPoint.action code

3.2. CMakeLists file

Everything that has been added to the new file have already being discussed in chapter 1.

3.3. Launch file

Launch files are placed in the launch folder and there is only 1 launch file named: roomba_robot.launch

File Edit View Selection Find Packages Help

roomba_robot.launch

```
<launch>

  <!-- Setting Global parameters -->
  <param name="/robot_name" type="string" value="roomba_robot" />
  <param name="/robot_max_speed" type="double" value="0.5" />
  <param name="/initial_post_x" type="double" value="0" />
  <param name="/initial_post_y" type="double" value="0" />
  <param name="/max_cordinate" type="double" value="100" />
  <param name="/min_cordinate" type="double" value="-100" />

  <!-- Initializing nodes in the right order, otherwise there will be errors-->
  <node name="dirt_pub_node" pkg="roomba_robot" type="dirt_pub_node.py"/>
  <node name="speed_pub_node" pkg="roomba_robot" type="speed_pub_node.py"/>
  <node name="position_pub_node" pkg="roomba_robot" type="position_pub_node.py"/>
  <node name="walk_to_point_action_server_node" pkg="roomba_robot" type="walk_to_point_action_server.py"/>

</launch>
```

Figure 3.2: roomba_robot.launch code

3.4. Message files

Message files are placed in the msg folder and there are 2 custom message files named: PositionAndDestination.msg and RobotPosition.msg

PositionAndDestination.msg RobotPosition.msg

```
#defines floating point values for current and final position
float32 cur_x
float32 cur_y
float32 dest_x
float32 dest_y
```

Figure 3.3: PositionAndDestination.msg code

PositionAndDestination.msg | RobotPosition.msg

```
1 #defines floating point for the position variables
2 float32 pos_x
3 float32 pos_y
4
```

Figure 3.4: RobotPosition.msg code

3.5. package.xml file

Everything that has been added to the new file have already being discussed in chapter 1.

3.6. Nodes

The node files are placed in the script files. There are many files in this and contain the most important aspects of the codes. Hence, each node's code will be displayed in the subsections.

3.6.1. control_node.py

control_node.py

```
7  import actionlib
8
9  #"Float32" msg type is imported from standard messages
10 from std_msgs.msg import Float32
11
12
13 #msg types are needed to enable for Action client and
14 #subsequent Subscriber to position topic
15 #They are imported from the messages that have been created before by the author
16 from roomba_robot.msg import RobotPosition
17 from roomba_robot.msg import PositionAndDestination
18
19 #msg types required to communicate with Action Server are imported from msgs folder
20 from roomba_robot.msg import WalkToPointAction
21 from roomba_robot.msg import WalkToPointResult
22 from roomba_robot.msg import WalkToPointFeedback
23 from roomba_robot.msg import WalkToPointGoal
24
25 #importing built-in packages and modules
26 import os
27 import math
28
29
30 #creating menu for selection of task for roomba
31 menu = """
32 ############## Select options by pressing 1-4 on your keyboard ##############
33
34 >>>>> 1. Current Dirt Amount
35 >>>>> 2. Current Roomba Position
36 >>>>> 3. Traverse Roomba to a Specified Destination
37 >>>>> 4. Return Roomba to the charging station
38 """
39
40
41 #Initializing Global variables with default and rosparam server values
42 CUR_DIRT = 0
43 CUR_ROB_POS = [rospy.get_param('initial_post_x'), rospy.get_param('initial_post_y')]
44 ROB_BASE_POINT = [rospy.get_param('initial_post_x'), rospy.get_param('initial_post_y')]
45 CUR_ROB_SPEED = 0
```

scripts/control_node.py 17:52

Figure 3.5: control_node.py code-1

control_node.py

```
45  CUR_ROB_SPEED = 0
46  ROB_MAX_SPEED = rospy.get_param("robot_max_speed")
47
48  #getting range values for the roomba
49  MAX_CORDINATE = rospy.get_param("max_cordinate")
50  MIN_CORDINATE = rospy.get_param("min_cordinate")
51  # -----------------------------------------------------
52
53
54  #funtion to set up the environment and initialize values
55  def setup():
56      #setting up the menu as global
57      global menu
58      #clear previous things on the screen
59      clear()
60
61
62
63  #main function of Control Node
64  def main():
65
66
67      #main loop of the control_node
68      while True:
69          #printing menu variable to the screen to show options to the user
70          print(menu)
71
72          #Validity check for input
73          check = True
74          while check:
75              try:
76                  #asking the user for input
77                  choice = int(input("Select: "))
78                  #cheking if the input in this list, otherwise AssertionError will be raised
79                  assert choice in [1, 2, 3, 4, 5]
80                  check = False
81              except:
82                  print("Please choose a valid number")
83
84          #if choice is 1, let the user know about the amount of dirt in the position
```

scripts/control_node.py 81:9

Figure 3.6: control_node.py code-2

control_node.py

```
83
84          #if choice is 1, let the user know about the amount of dirt in the position
85          if choice == 1:
86              print("Current DIRT: {} Amount".format(CUR_DIRT))
87
88
89          #if choice is 2, let the user know about the current position of robot
90          elif choice == 2:
91              print("Current Position: X = {}  Y = {}".format(CUR_ROB_POS[0], CUR_ROB_POS[1]))
92
93
94          #if choice is 3, action client is used to go to the desired position
95          elif choice == 3:
96              #from positionanddestination get the goal
97              goal = PositionAndDestination()
98              print("Cordinations of Destination")
99
100
101             #to ensure that the value is within the range
102             x = check_input(" X is equal to  ")
103             y = check_input(" Y is qual to  ")
104
105             #setting goal msg data
106             goal.dest_x = x
107             goal.dest_y = y
108             goal.cur_x = CUR_ROB_POS[0]
109             goal.cur_y = CUR_ROB_POS[1]
110
111             #calling the walk_to_point to create an Action Client
112             walk_to_point(goal)
113
114         #if choice is 5, use action client to go to the charging station of robot
115         else:
116             #creating the goal message
117             goal = PositionAndDestination()
118             #setting goal msg data
119             goal.dest_x = 0
120             goal.dest_y = 0
121             goal.cur_x = CUR_ROB_POS[0]
122             goal.cur_y = CUR_ROB_POS[1]
```

scripts/control_node.py 119:2

Figure 3.7: control_node.py code-3

control_node.py

```
            goal.dest_y = 0
            goal.cur_x = CUR_ROB_POS[0]
            goal.cur_y = CUR_ROB_POS[1]

            #calling the walk_to_point to create an Action Client
            walk_to_point(goal)


#Walk To Point Action client function
def walk_to_point(message):
    #defining global variables
    global ROB_MAX_SPEED
    global CUR_ROB_POS

    #equation to determing is the robot is in the final destination
    if message.dest_x == message.cur_x and message.dest_y == message.cur_y:
        print("Roomba is in the final destination")
    #otherwise create an action client to walk to the final destination
    else:
        #creating action client; action name is 'walk_to_point'
        #msg type is WalkToPointAction
        walk_client = actionlib.SimpleActionClient('walk_to_point', WalkToPointAction)
        #waiting for Action server to respond
        walk_client.wait_for_server()
        #create a goal message for walking
        destination = WalkToPointGoal(message)

        #Let the user know about distance and time to reach the final destination
        distance = round(math.dist([message.cur_x, message.cur_y], [message.dest_x, message.dest_y]), 1)
        time_to_reach = int(distance/ROB_MAX_SPEED)
        print("Distance: {} meters and the Time Tken: {} s".format(distance, time_to_reach))

        #sending the goal to the Action server, feedback callback function is process_feedback
        #Feedback callback function creates a thread for itself
        walk_client.send_goal(destination, feedback_cb = process_feedback)
        #wait for the Action server to finish the task to get the result
        walk_client.wait_for_result()

        rospy.sleep(0.7)
```

scripts/control_node.py 156:32

Figure 3.8: control_node.py code-4

control_node.py

```
159
160         #clear the terminal screen
161         clear()
162
163         #printing the current position of roomba
164         print("Current Cordinations position: X = {}  Y = {}".format(CUR_ROB_POS[0], CUR_ROB_POS[1]))
165
166
167 #action client feedback callback funtion
168 def process_feedback(feedback):
169     #accessing gloabl variable
170     global CUR_ROB_SPEED
171     #Let user know abot the distance to reach final destination
172     print("Current Distance: {} m moving at Speed: {} m/s".format(round(feedback.distance, 1), CUR_ROB_SPEED))
173
174
175
176
177
178 #callback function for the subscriber of '/dirt' topic
179 def dirt_sub(val):
180     #defining the global variable
181     global CUR_DIRT
182     #updating the global variable value to a the float number using built in round function
183     CUR_DIRT = round(val.data, 1)
184
185
186
187
188
189 #callback function for the subscriber of '/position' topic
190 def rob_pos_sub(val):
191     #defining the global variable
192     global CUR_ROB_POS
193     #updating the global variable value to a the float number using built in round function
194     CUR_ROB_POS[0] = round(val.pos_x)
195     CUR_ROB_POS[1] = round(val.pos_y)
196
197
198 #callback function for the subscriber of '/speed' topic
```

scripts/control_node.py 195:35

Figure 3.9: control_node.py code-5

control_node.py

```
197
198 #callback function for the subscriber of '/speed' topic
199 def rob_speed_sub(val):
200     #defining the global variable
201     global CUR_ROB_SPEED
202     #updating the global variable value to a the float number using built in round function
203     CUR_ROB_SPEED = round(val.data, 1)
204
205
206 #clearing the user terminal  function
207 def clear():
208     # for mac and linux (os.name is 'posix')
209     if os.name == 'posix':
210         _ = os.system('clear')
211         print()
212     else:
213     # for windows platfrom
214         _ = os.system('cls')
215         print()
216
217
218 #auto-validate user input values
219 def check_input(asked):
220     #defining the global variable
221     #the range of roomba operation
222     global MAX_CORDINATE
223     global MIN_CORDINATE
224     #Flag variable to keep track of whether the user input is valid
225     flag = True
226
227     while flag:
228         try:
229             #User input for valid position
230             number = float(input(asked))
231             #checking the input value is within the acceptable rance
232             assert MIN_CORDINATE <= number and number <= MAX_CORDINATE, "Select from {} to {}"
233             #if there is no AssertionError,the loop is stopped
234             flag = False
235
236         #if AssertionError occurs  raise an excetion and print an error has occured
```

scripts/control_node.py 233:18

Figure 3.10: control_node.py code-6

control_node.py

```
        #if AssertionError occurs, raise an excetion and print an error has occured
        except AssertionError as msg:
            print("Error")

        #if other exception arises, ask user to reenter the input
        except:
            print("Enter a valid number")

    #return the valid user input
    return number
if __name__ == "__main__":
    try:
        #initializing a control node
        rospy.init_node("control_node")

        #creating a subscriber for the topic "dirt"
        #received dirt as data type of Float32 from the callback func named dirt_sub
        rospy.Subscriber('dirt', Float32, dirt_sub)
        #creating a subscriber for the topic "position"
        #received position as data type of RobotPosition from the callback func named rob_pos_sub
        rospy.Subscriber('position', RobotPosition, rob_pos_sub)
        #creating a subscriber for the topic "speed"
        #received spped as data type of Float32 from the callback func named rob_speed_sub
        rospy.Subscriber('speed', Float32, rob_speed_sub)


        #launching setup function to set up the environment
        setup()
        #main function of Control Node - this is from where all the execution begins
        main()

        #to continiously run this control_node
        rospy.spin()

    #catch only ROS related exceptions and print them out
    except rospy.ROSInterruptException as e:
        print(e)

```

scripts/control_node.py* 245:15

Figure 3.11: control_node.py code-7

3.6.2. dirt_pub_node.py

dirt_pub_node.py

```python
#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy
#"Float32" msg type is imported from standard messages
from std_msgs.msg import Float32
#random module is imported to generate random values for dirt amount
import random
#publisher function for dirt
def pub_dirt_val():

    #initializing a dirt publisher node
    rospy.init_node('dirt_pub_node')

    #creating a publisher class object to publish dirt values
    # pusblish topic 'dirt' as datatype of Float32 whereas 'queue_size' defines outgoing message queue used for asynchronous publishing of the dirt publisher node
    dirt_pub = rospy.Publisher('dirt', Float32, queue_size=10)

    #setting rate of frequency as 5 times per second
    rate = rospy.Rate(5)

    #While ros is still running
    while not rospy.is_shutdown():
        #generate a random value between 20 and 25 for dirt amount
        val = random.random()*5 + 20
        #publish the value obtained of the dirt
        dirt_pub.publish(val)
        #sleeping 0.2 seconds since Rate is 5 Hz
        rate.sleep()

if __name__ == "__main__":
    try:
        #call the dirt publisher function
        pub_dirt_val()
    #catch only ROS related exceptions and print them out
    except rospy.ROSInterruptException as e:
        print(e)

```

scripts/dirt_pub_node.py* 1:1 LF UTF-8 Python GitHub Git (0)

Figure 3.12: dirt_pub_node.py code

3.6.3. position_pub_node.py

position_pub_node.py

```python
#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy

#import RobotPosition to access pos_x and pos_y for Roomba
from roomba_robot.msg import RobotPosition

#Define and initialize global variables
POSITION = RobotPosition()
#Access and assign initial values from rosparam server
POSITION.pos_x = rospy.get_param('initial_post_x')
POSITION.pos_y = rospy.get_param('initial_post_y')


#callback function for the subscriber of '/change_position' topic
def update_position(data):
    #defining the global variable
    global POSITION
    #updating the global variable value with current position value to always keep the last position
    POSITION = data


#publisher function
def pub_position_val():
    #defining the global variable
    global POSITION

    #creating a publisher class object to publish position values
    #publish topic 'position' as msgtype of RobotPosition where'queue size' is defined as outgoing message queue used for asynchronous publishing
    position_pub = rospy.Publisher('position', RobotPosition, queue_size=10)

    #setting rate of frequency at 10 times per second
    rate = rospy.Rate(10)
```

Figure 3.13: position_pub_node.py code-1

```python
    #while ros is still running
    while not rospy.is_shutdown():
        #publishing the position whose msg type is of the type RobotPosition
        position_pub.publish(POSITION)
        rate.sleep()


if __name__ == "__main__":
    try:
        #initializing this position publisher node
        rospy.init_node('position_pub_node')

        #creating a subscriber for the topic "change_position"
        #received change position as data type of RobotPosition from callback func of update_position
        rospy.Subscriber("change_position", RobotPosition, update_position)

        #call publisher for position
        pub_position_val()

        #to continiously keep this node running
        rospy.spin()

    #catch only ROS related exceptions and print them out
    except rospy.ROSInterruptException as e:
        print(e)
```

scripts/position_pub_node.py 61:1 LF UTF-8

Figure 3.14: position_pub_node.py code-2

3.6.4. speed_pub_node.py

position_pub_node.py speed_pub_node.py

```python
#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy

#"Float32" msg type is imported from standard messages
from std_msgs.msg import Float32

# intializing the speed to start off with 0
SPEED = 0


#callback function for the subscriber of '/change_speed' topic
def update_speed(val):
    #defining the global variable
    global SPEED
    #updating the speed to reflect the current speed value with the datatype of float32 as set by round function
    SPEED = round(val.data, 1)


#publisher function
def pub_speed_val():
    #defining the global variable
    global SPEED

    #creating a publisher class object to publish speed values
    #publish the topic 'speed' as datatype of Float32 where the 'queue_size' is defined outgoing message queue used for asynchronous publishing
    speed_pub = rospy.Publisher('speed', Float32, queue_size=10)

    #setting the frequency of 10 times per second
    rate = rospy.Rate(10)

    #while the ros is still running
    while not rospy.is_shutdown():
        #publishing the speed as datatype of Float32
        speed_pub.publish(SPEED)
        rate.sleep()
```

Figure 3.15: speed_pub_node.py code-1

```python
if __name__ == "__main__":
    try:
        #initializing this speed_publisher node
        rospy.init_node('speed_pub_node')

        #creating a subscriber for the topic "change_speed"
        #received the topic change_speed as data type of Float32 where  the callback func is defined as update_speed
        rospy.Subscriber("change_speed", Float32, update_speed)

        #call publisher for speed
        pub_speed_val()

        #to continiously run this node
        rospy.spin()

    #catch only ROS related exceptions and print them out
    except rospy.ROSInterruptException as e:
        print(e)
```

scripts/speed_pub_node.py 58:1 LF UTF-8 Python GitHub Git (0)

Figure 3.16: speed_pub_node.py code-2

3.6.5. turn_camera_client.py

turn_camera_client.py

```python
#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy

#import TurnCamera and TurnCameraResponse message types from the srv folder
from roomba_robot.srv import TurnCamera, TurnCameraResponse

#import opencv modules and packages for python for displaying the images
import cv2

# import CvBridge module which converts between ROS Image messages and OpenCV images.
from cv_bridge import CvBridge

#import built-in modules and packages
import os
import numpy as np


#define the function for angle to turn image at
def config_request(angle):

#wait for the serice to connect
    rospy.wait_for_service('/turn_camera')

    try:
        #Service definitions container for the request and response type forn turning the camera at a particular angle
        service_proxy = rospy.ServiceProxy('/turn_camera', TurnCamera)
        resp_msg = service_proxy(angle)


        #send response of the image as message
        image_msg = resp_msg.image

        #convert a ROS image message into an cv
        #Since the default value of "passthrough" is given, the destination image encoding will be the same as the image message encoding
        image = CvBridge().imgmsg_to_cv2(image_msg, desired_encoding='passthrough')
```

Figure 3.17: turn_camera_client.py code-1

turn_camera_client.py

```python
        image = CvBridge().imgmsg_to_cv2(image_msg, desired_encoding='passthrough')


        #After successfully converted images to OpenCV format, see a HighGui window with the desired image will be displayed.
        cv2.imshow('Turn Camera Image', image)

        #handling keyboard interrupt
        cv2.waitKey(0)
        cv2.destroyAllWindows()

#catch only ROS Service related exceptions and print them out
    except rospy.ServiceException as e:
        print("Service Reuqest Failed")
        print(e)


if __name__ == "__main__":
    try:
    #initialize this camera client
        rospy.init_node("turn_camera_client_node")

    #prompt user for angle input
        user_input = input("\nEnter the angle: ")

    #continue trying to process angle requirest until user presses q for quitting
        while user_input != 'q':
            try:
            #unless error occurs continue asking user for angles which will be in the float32 datatype
                config_request(float(user_input))
                user_input = input("\nEnter the angle: ")

#if any error while trying to process the request, an exception will be thrown
            except Exception as e:
                print("Error trying to process request")
                print(e)

#catch only ROS related exceptions and pass
    except rospy.ROSInterruptException:
        pass
```

scripts/turn_camera_client.py 66:46 LF UTF-8 Python GitHub Git (0)

Figure 3.18: turn_camera_client.py code-2

3.6.6. turn_camera_service.py

turn_camera_client.py turn_camera_service.py

```
#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy

#import TurnCamera and TurnCameraResponse message types from the srv folder
from roomba_robot.srv import TurnCamera, TurnCameraResponse


#import opencv modules and packages for python for displaying the images
import cv2

# import CvBridge module which converts between ROS Image messages and OpenCV images.
from cv_bridge import CvBridge

#import built-in modules and packages
import os


#class that defines functions that handles image loading, image conversion to messages and subsequently sending those messages to the client
class TurnCameraClass:

    def __init__(self):
    #angles available for calling in the images folder
    #angles here refer to the names of the images stored i nthe folder images
        self.available_angles = [-30, -15, 0, 15, 30]


    #A Service is created using a rospy.Service instance with a callback (send_image) to invoke when new requests are received for the rosservice turn_camera. E
    #each inbound request is handled in its own thread, so services must be thread-safe.
        self.ros_service = rospy.Service('turn_camera', TurnCamera, self.send_image)


#function to send image to the client and display
    def send_image(self, req):

    #retrieve the requested image with the angle as defined by the user
        image = self.get_image(req.turn_degrees)

scripts/turn_camera_service.py* 31:79 LF UTF-8 Python GitHub Git (0)
```

Figure 3.19: turn_camera_service.py code-1

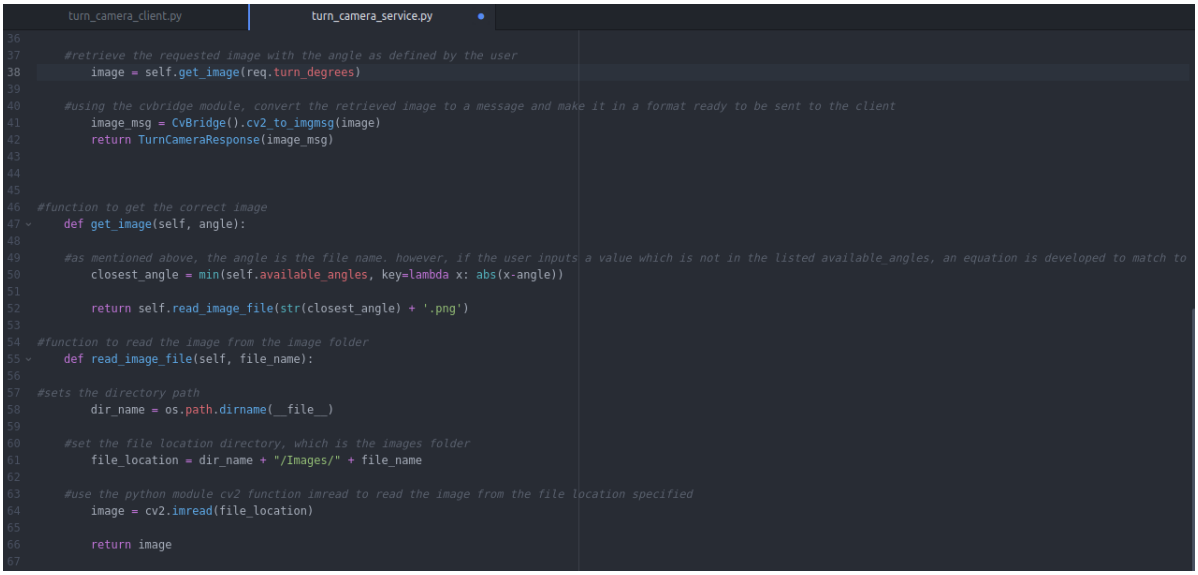


Figure 3.20: turn_camera_service.py code-2

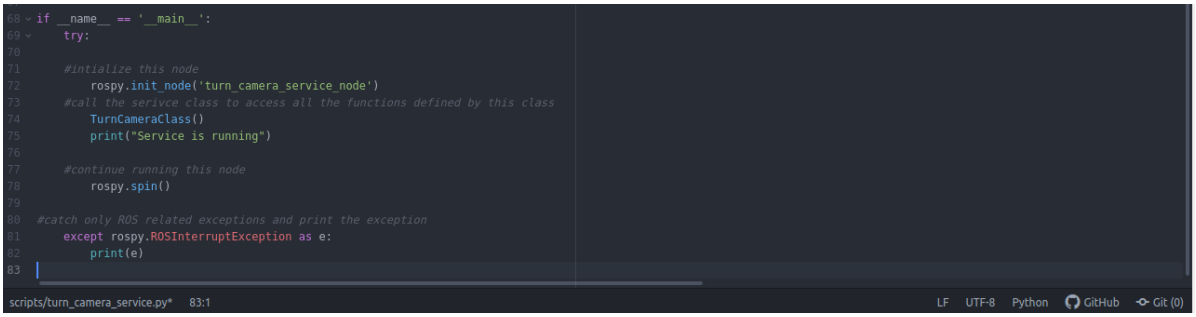
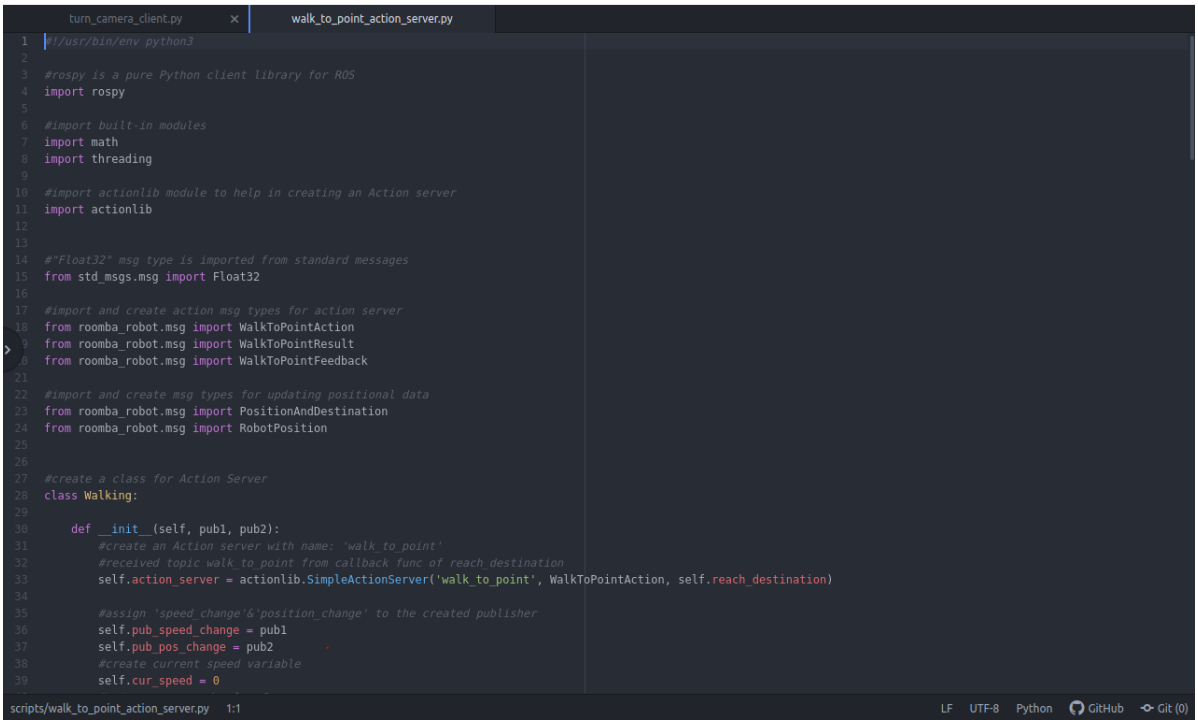


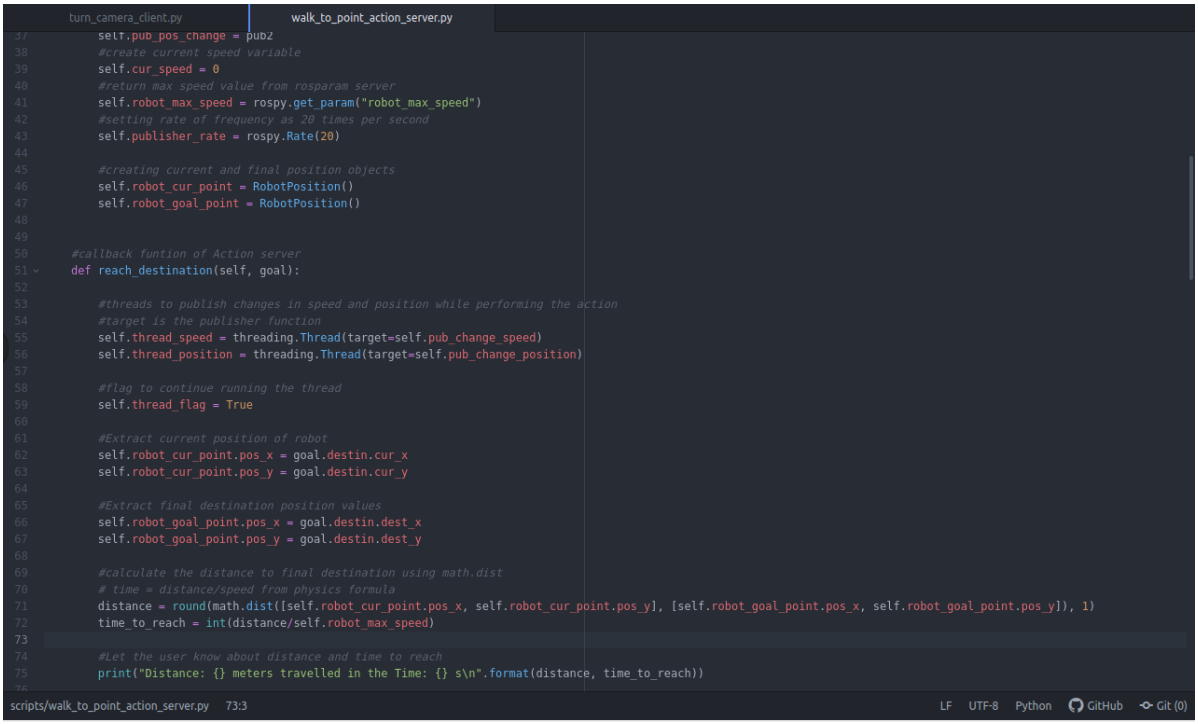
Figure 3.21: turn_camera_service.py code-3

3.6.7. walk_to_point_action_server_node.py



```
turn_camera_client.py   walk_to_point_action_server.py

#!/usr/bin/env python3

#rospy is a pure Python client library for ROS
import rospy

#import built-in modules
import math
import threading

#import actionlib module to help in creating an Action server
import actionlib


#"Float32" msg type is imported from standard messages
from std_msgs.msg import Float32

#import and create action msg types for action server
from roomba_robot.msg import WalkToPointAction
from roomba_robot.msg import WalkToPointResult
from roomba_robot.msg import WalkToPointFeedback

#import and create msg types for updating positional data
from roomba_robot.msg import PositionAndDestination
from roomba_robot.msg import RobotPosition


#create a class for Action Server
class Walking:

    def __init__(self, pub1, pub2):
        #create an Action server with name: 'walk_to_point'
        #received topic walk_to_point from callback func of reach_destination
        self.action_server = actionlib.SimpleActionServer('walk_to_point', WalkToPointAction, self.reach_destination)

        #assign 'speed_change'&'position_change' to the created publisher
        self.pub_speed_change = pub1
        self.pub_pos_change = pub2
        #create current speed variable
        self.cur_speed = 0

scripts/walk_to_point_action_server.py  1:1   LF  UTF-8  Python  GitHub  Git (0)
```

Figure 3.22: walk_to_point_action_server_node.py code-1



```
turn_camera_client.py   walk_to_point_action_server.py

        self.pub_pos_change = pub2
        #create current speed variable
        self.cur_speed = 0
        #return max speed value from rosparam server
        self.robot_max_speed = rospy.get_param("robot_max_speed")
        #setting rate of frequency as 20 times per second
        self.publisher_rate = rospy.Rate(20)

        #creating current and final position objects
        self.robot_cur_point = RobotPosition()
        self.robot_goal_point = RobotPosition()


    #callback funtion of Action server
    def reach_destination(self, goal):

        #threads to publish changes in speed and position while performing the action
        #target is the publisher function
        self.thread_speed = threading.Thread(target=self.pub_change_speed)
        self.thread_position = threading.Thread(target=self.pub_change_position)

        #flag to continue running the thread
        self.thread_flag = True

        #Extract current position of robot
        self.robot_cur_point.pos_x = goal.destin.cur_x
        self.robot_cur_point.pos_y = goal.destin.cur_y

        #Extract final destination position values
        self.robot_goal_point.pos_x = goal.destin.dest_x
        self.robot_goal_point.pos_y = goal.destin.dest_y

        #calculate the distance to final destination using math.dist
        # time = distance/speed from physics formula
        distance = round(math.dist([self.robot_cur_point.pos_x, self.robot_cur_point.pos_y], [self.robot_goal_point.pos_x, self.robot_goal_point.pos_y]), 1)
        time_to_reach = int(distance/self.robot_max_speed)

        #Let the user know about distance and time to reach
        print("Distance: {} meters travelled in the Time: {} s\n".format(distance, time_to_reach))

scripts/walk_to_point_action_server.py  73:3   LF  UTF-8  Python  GitHub  Git (0)
```

Figure 3.23: walk_to_point_action_server_node.py code-2

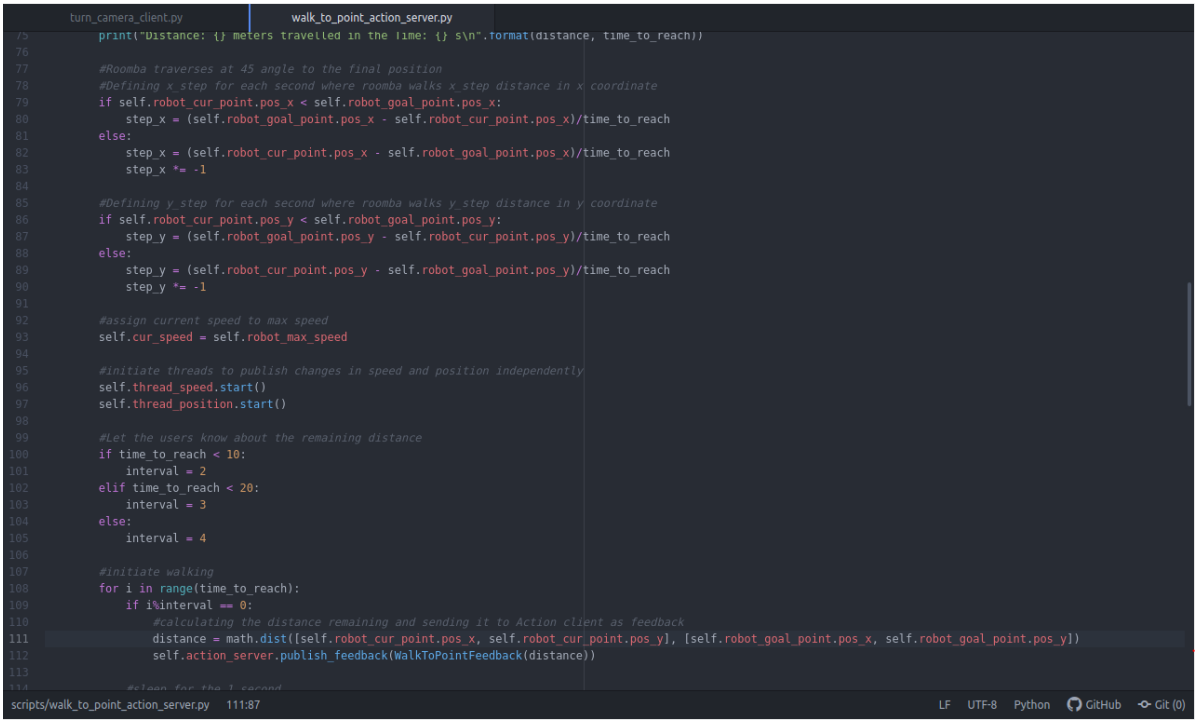


Figure 3.24: walk_to_point_action_server_node.py code-3

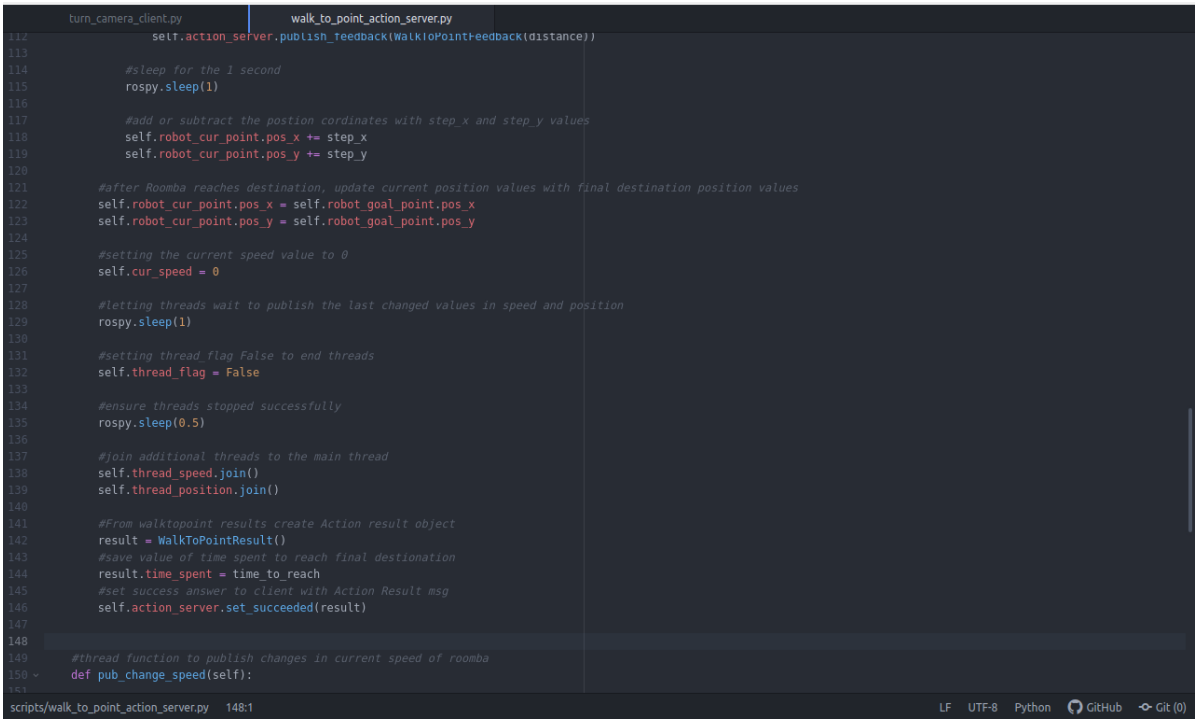


Figure 3.25: walk_to_point_action_server_node.py code-4

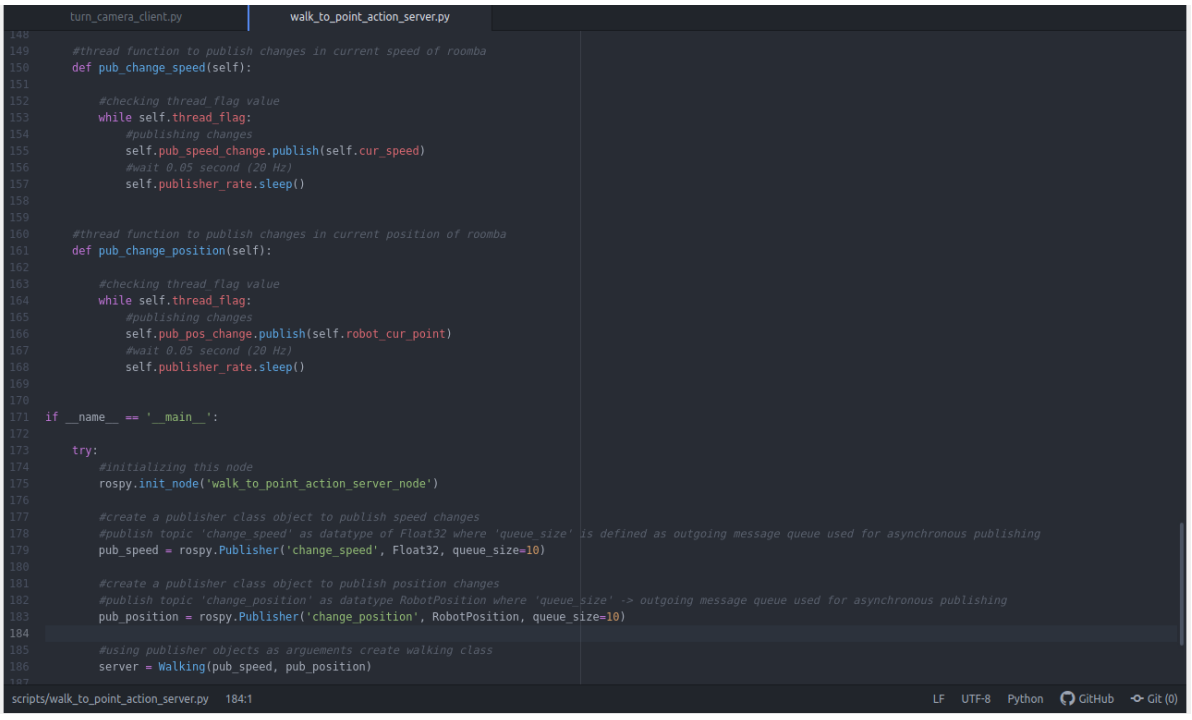


Figure 3.26: walk_to_point_action_server_node.py code-5

```
        #using publisher objects as arguements create walking class
        server = Walking(pub_speed, pub_position)

        #to continiously run this node
        rospy.spin()

    #catch only ROS related exceptions and print them out
    except rospy.ROSInterruptException as e:
        print(e)

```

scripts/walk_to_point_action_server.py 194:1 LF UTF-8 Python GitHub Git (0)

Figure 3.27: walk_to_point_action_server_node.py code-6

3.7. srv files

srv files are placed in the srv folder and there is only 1 srv file named: TurnCamera.srv

TurnCamera.srv

```
1  #request
2  float32 turn_degrees
3  ---
4  #response
5  sensor_msgs/Image image
6
```

Figure 3.28: TurnCamera.srv code

Bibliography

ROS.org. (2021a). *Actionlib*. http://wiki.ros.org/actionlib
ROS.org. (2021b). *Bags*. http://wiki.ros.org/Bags
ROS.org. (2021c). *Creating a workspace for catkin*. http://wiki.ros.org/catkin/Tutorials/create_a_workspace
ROS.org. (2021d). *Roslaunch*. http://wiki.ros.org/roslaunch
ROS.org. (2021e). *Services*. http://wiki.ros.org/Services
ROS.org. (2021f). *Understanding ros nodes*. http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes