_distance.py _final.py _avoidance.py

TECHNICAL UNIVERSITY OF CLUJ NAPOCA
APPLIED INFORMATICS AND AUTOMATION

TurtleBot3 Path Control Project

Student name: *Tudor Ada Maria, Maliț a Alin Ciprian*

Course: *Robot Control Systems* – Professor: *Dr. Ing. Anastasios Natsakis*
Due date: *Jan 8th, 2024*

Contents

1. Introduction

TurtleBot3 is a versatile and compact robot platform designed for education, research, and hobbyist purposes. Developed by ROBOTIS, the TurtleBot3 series offers an affordable and accessible way to explore the field of robotics. This project aims to utilize the TurtleBot3 Burger as a foundation for experimentation, learning, and the implementation of various robotic applications.

The TurtleBot3 Burger is a compact, modular, and versatile robotics platform developed by ROBOTIS. Equipped with sensors like LiDAR, IMU, and a camera, it runs on the Robot Operating System (ROS), offering an accessible way to learn and experiment with robotics. Its expandable design and compatibility make it ideal for educational purposes and research, enabling users to explore various applications in autonomous navigation and manipulation. In contrast, the Waffle is larger, more versatile, and better suited for advanced projects needing more customization and capabilities in larger environments.

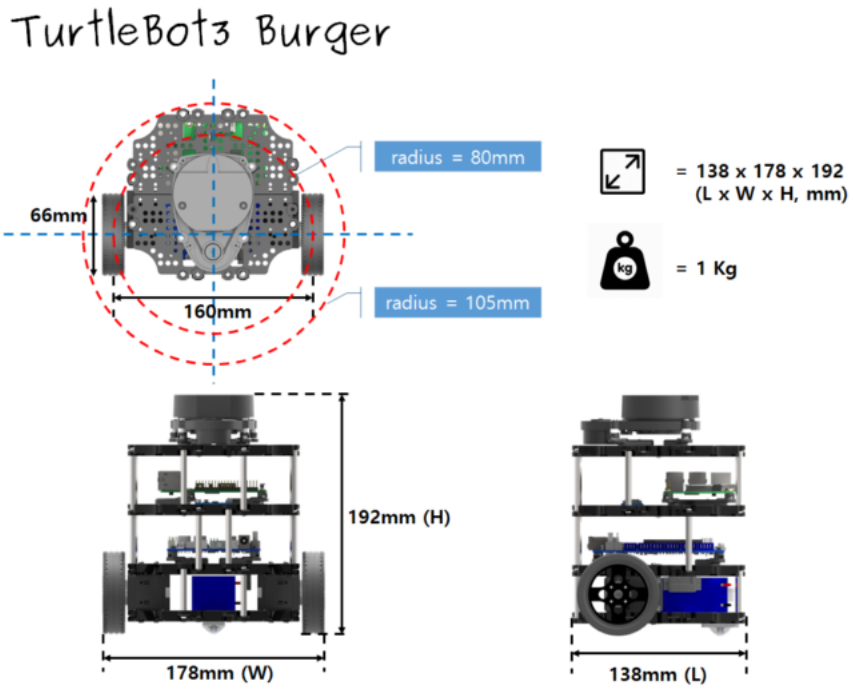

Figure 1: TurtleBot3 Burger

The following project aims to create an algorithm for the TurtleBot3 model that would allow the definition of areas for patrolling, the traversal to patrol areas. Simultaneously, the robot should detect nearby obstacles by laser scanning and take on predefined avoidance routes, after which it would resume the patrolling algorithm. The first task of user input, route calculation and traversal is completed by Tudor Ada-Maria, and second task, the obstacle avoidance algorithm is completed by Malita Alin Ciprian. The two tasks will communicate continuously and work harmoniously in order to achieve the project's goal.

2. Packages

For this application we used:

1. ROS Noetic
2. Gazebo 11
3. Turtlebot3 dependecy packages

(check for installation guides Section 6, References)

Figure 2: House Map

As part of the TurtleBot3 library, "turtlebot3_house.launch" specifies a launch file within the turtlebot3_gazebo package. Launch files in ROS contain configurations and instructions for starting specific nodes and setting up the environment. In this case, turtlebot3_house.launch launches a Gazebo simulation environment with a house-like structure for the TurtleBot3 robot to navigate and operate within. This environemnt is pictured above in Figure 2.

In order to be able to run the python scripts that will be presented below, we need to create two additional files in a ros workspace in our project folder:

2.1. CMakeLists.txt. The primary purpose of the CMakeLists.txt file is to define how the code within a package should be built. It contains directives that specify the

project's dependencies, compilation flags, linked libraries, and custom build instructions. Within the context of ROS and Gazebo applications, the CMakeLists.txt file configures the build process for ROS nodes, plugins, or other components, ensuring that they compile correctly and can be integrated into the broader ROS ecosystem.

2.2. package.xml. The package.xml file in ROS serves as a manifest file, detailing metadata about the package itself. It contains essential information such as the package name, version, maintainer contact details, dependencies on other ROS packages, and a description of the package's purpose. This file is crucial for ROS to understand the package's structure and dependencies. In the context of running Python scripts for a ROS Noetic and Gazebo 11 application, the package.xml file helps define the necessary ROS dependencies and any Python-specific dependencies that the package relies on, ensuring seamless integration and proper execution within the ROS ecosystem.

3. Patrolling Algorithm

Tasks:

1. There will be areas define by the user.
2. The robot needs to move to and patrol these areas
3. The robot has a maximum allocated time to complete a movement. If it does not reach it, it will move to the next goal.
4. The whole process needs to stop after being given a user command.

3.1. Area Definition. First we will establish the logic behind how an area is defined. In the figure below we can see how we allocated the points based on a given length and a given central point. These, together with an area name, are the inputs the user needs to provide in order to define an area.

In the code underneath this paragraph we can see the definition of the function define_area which takes as parameters the object, an area name, an area center and an area length. A dictionary entry will be created to store the center and the length where they key word is the area name. We can also see a define_boundary function that returns the calculated points according to the image above. Outside the defined class we have two other functions that simply take the user input in order to define an area and choose one to patrol.

```
1
2  ## ----------------- PATROLER CLASS
3  class TurtleBotPatroler:
4
5  # ... other code here
6
7    ## ----------------- AREA to patrol DEFINITION
8      def define_area(self, area_name, center,length):
9          self.areas[area_name] = [center, length]
10
11     ## ----------------- BOUNDARY points of area DEFINITION
12     def define_boundary(self, area_name):
```

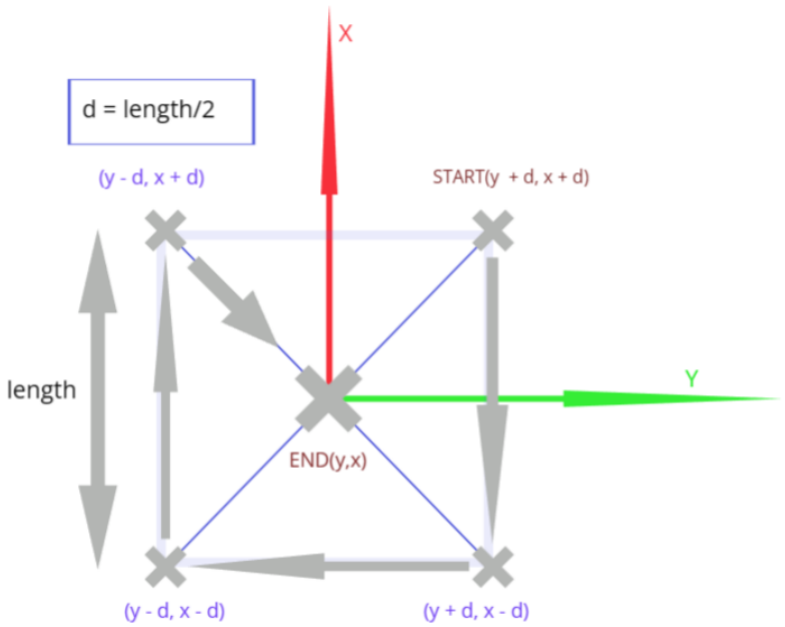

Figure 3: Patrol Points

```
        if area_name not in self.areas:
            print(f"Area '{area_name}' not found.")
            return

        boundary_points = []
        center = self.areas[area_name][0]  # Accessing the first two elements (index 0 and 1) for the center (x, y)
        length = self.areas[area_name][1]  # Accessing the third element (index 2) for the length/distance value

        boundary_points.append([center[0]+length/2,center[1]-length/2])
        boundary_points.append([center[0]-length/2,center[1]-length/2])
        boundary_points.append([center[0]-length/2,center[1]+length/2])
        boundary_points.append([center[0]+length/2,center[1]+length/2])
        boundary_points.append([center[0],center[1]])

        return boundary_points

## ---------------- END PATROLER CLASS

  ##---------- USER INPUT AREA
  def define_area_console_input(patroller):
    area_name = input("Enter area name: ")
    center_x = float(input("Enter center X coordinate: "))
    center_y = float(input("Enter center Y coordinate: "))
    radius = float(input("Enter radius: "))
    patroller.define_area(area_name, (center_x, center_y), radius)

##---------- USER INPUT AREA
def what_area_to_patrol(patroller):
    area_name = input("What area should I patrol?")
```

```
42     patroller.patrol_area(area_name)
```

Listing 1: Area_definition.py

In the previous code we can see that a function patrol_area is called. In the next, we will present it by separating it into four sections: Initialization, Aligning, Traversing, Patrolling.

3.2. Movement.

1. **Initialization**. For the code to work we first need to import some key functions and to do some initialization for the robot. Aside from some imports like atan2, quaterion_multiply, etc., that are used in order to do some calculations, it is important to take note of *from geometry_msgs.msg import Twist* and *from nav_msgs.msg import Odometry*. Twist allows us to send velocity commands to the robot, while Odometry is essential in order to be aware of the robot's position. Inside the class we will create a Subscribers and Publishers to allow us to communicate with the robot and we will establish a reading frequency of 10Hz. A function that we will frequently use is stop_robot that sends a twist message to set all velocities to zero, in other words, the function stops the robot.

```
1  ## ----------------- IMPORTS
2  from socket import timeout
3  import time
4  import signal
5  import rospy
6  from geometry_msgs.msg import Twist
7  from nav_msgs.msg import Odometry
8  from sensor_msgs.msg import LaserScan
9  from math import atan2, pi, sqrt, sin, cos
10 from tf.transformations import euler_from_quaternion
11 from tf.transformations import quaternion_multiply,
   ↪ quaternion_conjugate, quaternion_inverse
12 import numpy as np
13
14 ## ----------------- PATROLER CLASS
15 class TurtleBotPatroler:
16
17 ## ----------------- INITIALIZATION
18     def __init__(self):
19         rospy.init_node('turtlebot_controller', anonymous=True)
20
21         self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist
   ↪ , queue_size=20)
22
23         self.pose_subscriber = rospy.Subscriber('/odom', Odometry,
   ↪ self.update_pose)
24         self.scan_subscriber = rospy.Subscriber('/scan', LaserScan,
   ↪  self.update_scan)
25
26         self.pose = None
27         self.scan_data = None
28
29         self.rate = rospy.Rate(10) # Creates a Rate object with a
   ↪ frequency of 10 Hz
30         self.areas = {}   # Initialize self.areas as an empty
   ↪ dictionary
```

```
31
32  ## ----------------- UPDATING POSITION
33      def update_pose(self, data):
34          self.pose = data.pose.pose
35
36  # .... other code
37
38  ## ----------------- STOPPING THE ROBOT
39      def stop_robot(self):
40
41          twist = Twist()
42          twist.linear.x = 0.0  # linear speed
43          twist.angular.z = 0.0
44          twist.linear.y = 0.0
45          twist.linear.z = 0.0
46          twist.angular.x = 0.0
47          twist.angular.y = 0.0
48          self.velocity_publisher.publish(twist)
49
50          print(f"Robot stopped")
51
52  # .... other code
```

Listing 2: Init.py

2. **Aligning**. First we will calculate the goal angle from the goal position and then proceed to continuously calculate the angle difference between the current angle and the goal angle, while changing the angular velocity of the robot. When reaching an acceptable error margin, the robot will stop as it considers itself sufficiently aligned.

```
1  ## ----------------- PATROLER CLASS
2  class TurtleBotPatroler:
3
4  # ..other code
5
6
7      ##---------------- CALCULATING GOAL ANGLE
8      def goal_angle_is(self,goal_x,goal_y):
9
10         if self.pose is None:
11             return None
12
13         goal_angle = atan2(goal_y - self.pose.position.y, goal_x -
       ↪ self.pose.position.x)
14
15         print(f"Calculated Goal Angle: {goal_angle}")
16         return goal_angle
17
18     ## ----------------- CALCULATING ANGLE DIFFERENCE BETWEEN
       ↪ desired position and current one
19     def get_angle_difference(self,goal_angle):
20
21         if self.pose is None:
22             return None
23
24         current_quaternion = [ self.pose.orientation.x, self.pose.
       ↪ orientation.y, self.pose.orientation.z, self.pose.orientation
```

```
    ↪ .w ]
25
26          # Convert the angle to quaternion
27          goal_quaternion = [ 0.0, 0.0, sin(goal_angle / 2), cos(
    ↪ goal_angle / 2) ]
28
29          # Convert both quaternions to unit quaternions
30          current_quaternion = current_quaternion / np.linalg.norm(
    ↪ current_quaternion)
31          goal_quaternion = goal_quaternion / np.linalg.norm(
    ↪ goal_quaternion)
32
33          # Calculate the quaternion difference
34          quaternion_diff = quaternion_multiply(goal_quaternion,
    ↪ quaternion_inverse(current_quaternion))
35
36          # Convert the quaternion difference to Euler angles
37          _, _, angle_difference = euler_from_quaternion(
    ↪ quaternion_diff)
38
39          print(f"Calculated Angle Difference: {angle_difference}")
40          return angle_difference
41
42      ##---------- ALIGNING with goal angle
43      def aligning(self,goal_angle,twist,angle_difference,start_time)
    ↪ :
44
45          # Stopping the robot
46          self.stop_robot()
47
48          # allowed error
49          a_error=0.0001
50
51          #speed
52          speed=0.09
53
54          while abs(angle_difference) > a_error:
55
56              angle_difference=self.get_angle_difference(goal_angle)
    ↪ # recalculating angle_difference
57
58              if angle_difference > 0:
59                  angular_speed = speed
60              else:
61                  angular_speed= (-1)*speed
62
63              if angle_difference < a_error and angle_difference > 0:
    ↪  angular_speed = 0.0
64              if angle_difference > (-1)*a_error and angle_difference
    ↪  < 0: angular_speed = 0.0
65
66              twist.angular.z = angular_speed # modifies the twist
    ↪ object
67              self.velocity_publisher.publish(twist) # publishes
    ↪ through twist cmd to robot
68              self.rate.sleep() # ensures that a 10 Hz frequency is
    ↪ maintained
69
```

```
            # if rotating longer than a minute, exit loop
            to=self.timeout(start_time)
            if(to==1):
                print(f"Goal Angle: {goal_angle}")
                self.stop_robot()
                break

        print(f"Goal Angle: {goal_angle}")

        #stop robot
        self.stop_robot()
```

Listing 3: Movement_Angle.py

3. **Traversing**. After the robot aligns itself it will move on its x-axis (the linear velocity on x will be changed, do not confuse it with the base frame of Gazebo) until the calculated distance to the goal position will reach an predefined error margin.

```
## ----------------- PATROLER CLASS
class TurtleBotPatroler:

# ..other code

    ## ----------------- CALCULATING POSITION BETWEEN TWO POINTS
    def get_distance(self, goal_x, goal_y):

        if self.pose is None:
            return None

        print(f"current x is: {self.pose.position.x}")
        print(f"current y is: {self.pose.position.y}")
        x = self.pose.position.x - goal_x
        y = self.pose.position.y - goal_y

        return sqrt(x*x + y*y)

#.. other code
##------ TRAVERSE DISTANCE
    def traversing(self,goal_x,goal_y,twist,distance,start_time):

        # stopping the robot
        self.stop_robot()

        # allowed error
        t_error=0.3 # 30 cm

        # speed
        speed=0.1

        while distance > t_error:

            # linear speed gets modified
            twist.linear.x = speed
            self.velocity_publisher.publish(twist)

            # recalculating distance
            distance = self.get_distance(goal_x, goal_y)

```

```
            # ensuring 10Hz
            self.rate.sleep()

            # if too far go to next goal
            if(distance > 7):
                print(f"Goal x: {goal_x}, Goal y: {goal_y}")
                self.stop_robot()
                print(f"Distance is to great, skip goal")
                break

            # if moving longer than a set time, exit loop
            to=self.timeout(start_time)
            if(to==1):
                print(f"Goal x: {goal_x}, Goal y: {goal_y}")
                self.stop_robot()
                break

        print(f"Goal x: {goal_x}, Goal y: {goal_y}")

        # Stopping the robot
        self.stop_robot()
```

Listing 4: Movement_distance.py

4. **Patrolling**. Finally, the previous parts are combined in order to iterate through the calculated goal points and move towards them.

```
## ----------------- PATROLER CLASS
class TurtleBotPatroler:

# ..other code

## ----------------- MOVING TO GOAL
    def move_to_goal(self, goal_x, goal_y):


        # declaring twist to send msg
        twist = Twist()

        if self.pose is None:
            print(f"Pose is not updated yet.")
            return

        # starting measuring time
        start_time=time.time()

        # Calculating Goal angle
        goal_angle=self.goal_angle_is(goal_x,goal_y)

        # aligning with desired position through rotation (angular)

        angle_difference=self.get_angle_difference(goal_angle)
        self.aligning(goal_angle,twist,angle_difference,start_time)

        # traversing distance (linear)

        distance = self.get_distance(goal_x, goal_y)
        self.traversing(goal_x,goal_y,twist,distance,start_time)

```

```
## ----------------- PATROL IMPLEMENTATION
    def patrol_area(self, area_name):
        if area_name not in self.areas:
            print(f"Area '{area_name}' not found.")
            return

        boundary_points=self.define_boundary(area_name)
        for point in boundary_points:
            print(f"Patrolling near: {point}")
            self.move_to_goal(point[0], point[1])
            self.rate.sleep()
```

Listing 5: Movement_final.py

3.3. Timeout. In the Movement.Patrolling section we created a variable that takes the current time and passes it to both the aligning and the traversing functions that will have calls to the timeout function as a fail-safe. The timeout function takes the passed start_time and calculates the difference between it and the current time. If this difference, measured in seconds, is larger than the pre-established accepted time constraint, the robot stops and moves towards the new goal.

```
## ----------------- TIMEOUT check
    def timeout(self,start_time):

        max_time=120 # 120s
        current_time = time.time()
        elapsed_time = current_time - start_time

        if elapsed_time > max_time:
            print(f"Skipping the goal due to timeout.")
            return 1

        return 0
```

Listing 6: Timeout.py

3.4. Full Stop. For the final part we create a signal handler, that will listen for the Ctrl+C command that stops the process and will give the robot one final command to stop.

```
##--------- SIGINT HANDLER
def sigint_handler_wrapper(patroller):
    def sigint_handler(signum, frame):
        print("Ctrl + C detected. Stopping the robot.")
        if patroller:
            patroller.stop_robot()
        rospy.signal_shutdown('Ctrl + C detected')
    return sigint_handler



## ----------------- MAIN
def main():
    try:
        patroller = TurtleBotPatroler()
        rospy.sleep(1)
```

```
17
18
19      #... other code
20
21          # Register the signal handler for Ctrl + C
22          signal.signal(signal.SIGINT, sigint_handler_wrapper(patroller))
23
24          #... other code
25
26          rospy.spin()
27
28      except rospy.ROSInterruptException:
29          pass
```

Listing 7: FullStop.py

3.5. Limitations.

1. The user is limited in what they can offer as input. They can only provide the center of the desired area (which implies that they need to know its coordinates) and the length that will define the square-based movement pattern. They have no say in how the pattern is and if they make a mistake when entering the input, there is no fail-safe, they will simply be given an error and be forced to run the script again.

2. There are multiple limitations also resulting from calculation errors. The robot is pretty slow in order to not miss its error margins. In addition, the user cannot provide too many decimals when giving the center coordinates of an area or provide centers too far away from each other as any of these actions will increase the chance of the robot never entering is acceptable error margin, resulting in the robot only being stopped by either timeout or a too great pre-defined distance error constraint.

3. The last known major limitation is to be solved by the Obstacle Avoidance Tasks from Section 4, as the code so far has no such algorithm implemented and will get stuck in obstacles until it is forced to move to the next goal.

4. Obstacle Avoidance Algorithm

Tasks:

1. Use the LiDAR to determine distance from obstacles
2. Differentiate between obstacle types and declare when an obstacle is detected and its characteristics
3. Employ appropriate avoidance strategy based on obstacle, then return to patrolling

The project was created with the consideration that the robot would have zero prior knowledge of its surroundings and thus the trajectory calculated to its goals has no way to account for the possible obstacles in the way of the robot. Therefore, algorithms will be utilised to detect any obstacles in the robot's way before any collision could occur, as well as to detour around the detected obstacles and arrive to the destination. Below are described the detection and avoidance steps part of this task.

4.1. Detection.

1. **Scanning**. The code in in Movement.Initialization section showcases the subscriber for the *LaserScan imported from sensor_msgs.msg* topic, to which we assign the callback function *update_scan* with self attribute. The data recevied by the callback is then transfered to the *detect_obstacle(self,data)* method from the avoidance script.

```
1
2  class TurtleBotPatroler:
3      ## ----------------- INITIALIZATION
4      def __init__(self):
5          rospy.init_node('turtlebot_controller', anonymous=True)
6
7          self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist
       ↪ , queue_size=20)
8
9          self.pose_subscriber = rospy.Subscriber('/odom', Odometry,
       ↪ self.update_pose)
10         self.scan_subscriber = rospy.Subscriber('/scan', LaserScan,
       ↪  self.update_scan)
11
12         self.pose = None
13         self.scan_data = None
14
15         self.rate = rospy.Rate(10) # Creates a Rate object with a
       ↪ frequency of 10 Hz
16         self.areas = {}   # Initialize self.areas as an empty
       ↪ dictionary
17
18   ## ----------------- UPDATING SCANNER
19     def update_scan(self, data):
20         self.scan_data = data.ranges
21         self.obstacle_detected = detect_obstacle(self,data)
```

Listing 8: ModifiedScan.py

2. **Obstacle Detection**. The algorithm for detection begins with the definition of the two thresholds *thr1 and thr2*. The *data.ranges* variable holds the information for the distance between the burger robot and the obstacles around it, being an array with 360 values. These values attributed for each degree around the robot, starting from right in front of the robot and their order being counter-clockwise, as shown in Figure 4. The variable *thr1* is for detection of obstacles right in front of the robot, while *thr2* is for the left and right obstacles, found at the angles of 15 and 345 respectively. Both thresholds are given the value of 0.6 meters for this application. Two if conditions are used, the first checks whether the robot has registered in its data any object. If only this first if condition is met, the robot will consider that it detected a small obstacle, returning '1'. This obstacle will be added to the **obstacle** dictionary of the controller object by calling the *add_obstacle* method, which formats the data so it can be attributed to the controller. If no obstacles are close enough to trigger the first 'if' condition, the function returns '0'. The results of the *detect_obstacle* function are stored in the controller's variable *obstacle_detected*, initialized with '0'. Obstacles are considered as large if they block the robot from all 3 angles simultaneously, as shown in the second 'if' condition, which will return '2' on a positive check.

```
#!/usr/bin/env python
import rospy # Python library for ROS
from sensor_msgs.msg import LaserScan # LaserScan type message is defined in sensor_msgs
from geometry_msgs.msg import Twist #
from Area_definition_and_movement import *

## ---------------- DETECTING THE PRESENCE AND POSITION OF THE OBSTACLE
def detect_obstacle(self,data):
        thr1 = 0.6 # Treshold for Front collision
        thr2 = 0.6 # Treshold for Left and Right collision
        if (data.ranges[0]<thr1 or data.ranges[15]<thr2 or data.ranges[345]<thr2): # Checks if there are obstacles in front and 15 degrees left and right
            #  print (f"Range data at 0 deg:  {data.ranges[0]}" )
            #  print (f"Range data at 15 deg:  {data.ranges[15]}")
             # print ( f"Range data at 345 deg: {data.ranges[345]}")
              if (data.ranges[0]<thr1 and data.ranges[15]<thr2 and data.ranges[345]<thr2): #Large obstacle case
                print ('---Large Obstacle Detected---')
                self.obstacle['obstacle'] = add_obstacle(data.ranges[0], data.ranges[15], data.ranges[345],2)
                return 2
              print ('---Small Obstacle Detected---')
              self.obstacle['obstacle']  = add_obstacle(data.ranges[0], data.ranges[15], data.ranges[345],1)
              return 1
        return 0 #case of no obstacle detected

## ---------------- ADDING OBSTACLE OBJECT TO CONTROLLER MEMORY
def add_obstacle(front_dist, left_dist, right_dist, size):
    #Obstacle types: { Small=>size=1, Large=>size=2}
    return [front_dist, left_dist, right_dist,size]
```

Listing 9: Obstacle_detection.py

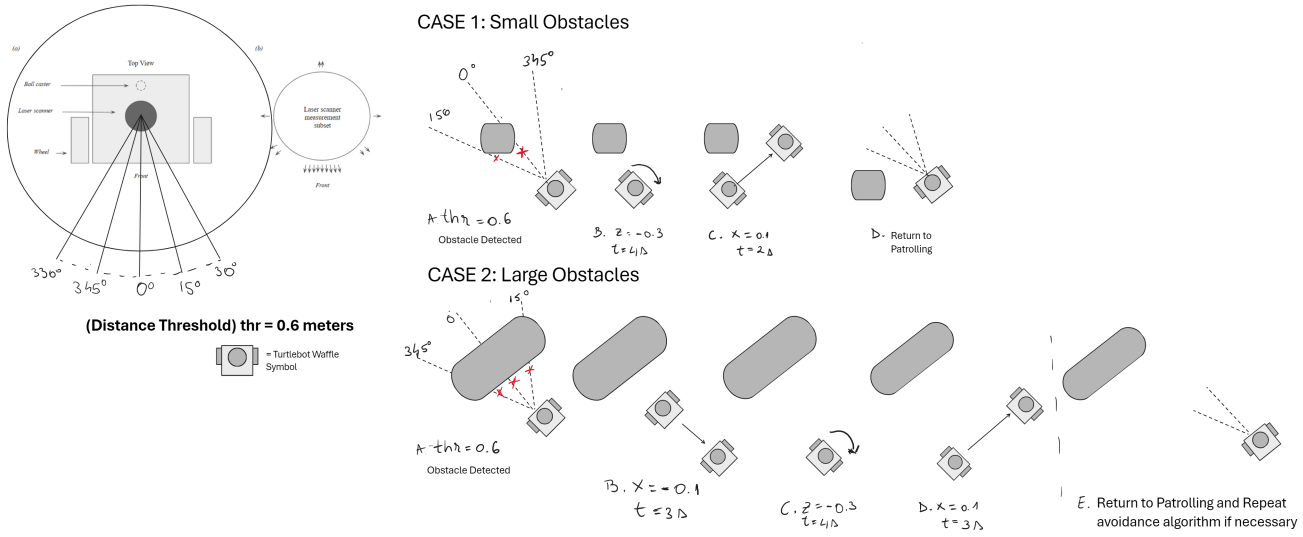

Figure 4: Avoidance Algorithm

4.2. Avoidance.

1. **Traversal**. The *traversing* method is modified to rely on the condition of the *obstacle_detected*, which will allow the robot to either move forward towards its goal, or to enter the obstacle avoidance algorithm by calling the method described in the next section, *avoid_obstacle*. Once the avoidance algorithm is complete, the robot will recall the *move_to_goal* method and reroute to the goal from its new position. The script makes sure to notify the end of the avoidance algorithm before beginning to reroute itself.

```
def traversing(self,goal_x,goal_y,twist,distance,start_time):
      # stopping the robot
      self.stop_robot()
      # allowed error
      t_error=0.3 # 30 cm
      # speed
      speed=0.15
      while distance > t_error:
          if(self.obstacle_detected == 0):
          # linear speed gets modified
           twist.linear.x = speed
           self.velocity_publisher.publish(twist)

           # recalculating distance
           distance = self.get_distance(goal_x, goal_y)

          # ensuring 10Hz
           self.rate.sleep()

          # if too far go to next goal
           if(distance > distance > speed * 0.15 * 240 ):
```

```
                #0.15m/s is the robot's regular speed
                 print(f"Goal x: {goal_x}, Goal y: {goal_y}")
                 self.stop_robot()
                 print(f"Distance is to great, skip goal")
                 break

            # if moving longer than a set time, exit loop
             to=self.timeout(start_time)
             if(to==1):
                 print(f"Goal x: {goal_x}, Goal y: {goal_y}")
                 self.stop_robot()
                 break
            elif(self.obstacle_detected >0):
                avoid_obstacle(self)
                print(f"Reroute to Goal x: {goal_x}, Goal y: {
    ↪ goal_y}")
                self.obstacle_detected = 0 #reset the obstacle
    ↪ detection
                self.stop_robot()
                self.move_to_goal(goal_x,goal_y)


        print(f"Goal x: {goal_x}, Goal y: {goal_y}")
        # Stopping the robot
        self.stop_robot()
```

Listing 10: ModifiedTraversing.py

2. **Obstacle Avoidance**. The *avoid_obstacle* method initializes a threshold variable with the same value of 0.6 meters. As described in Figure 4, the robot has 2 available avoidance strategies based on a simple distinction between what is defined as small or large obstacle (through the variable size of the obstacle object). Avoiding small obstacles involves a small rotation and detour linear movement before returning to the patrolling trajectory. Larger obstacles are avoided by first backing away and then making a wider rotation and linear movement away from the obstacle. In both cases the robot takes into account which direction it should rotate in based on proximity to the obstacle. After accomplishing an avoidance strategy, the robot will stop its movements with a final twist command before calling the *stop_robot* method from within the controller.

```
def avoid_obstacle(self):
    # def avoid_obstacle(self,goal_x,goal_y):  previous version that
    ↪ would take goal into account
    thr1 = 0.6 # Treshold
    twist=Twist()
    angular_speed = 0.3
    linear_speed = 0.1

    if(self.obstacle['obstacle'][3] == 1): # small obstacle algorithm
      if(self.obstacle['obstacle'][1] > self.obstacle['obstacle'][2] ):
        twist.angular.z=angular_speed #rotate to the left if
      else:
       twist.angular.z=-angular_speed

      self.velocity_publisher.publish(twist)
      rospy.sleep(4) # sleep after angular twist
```

```
16      twist.angular.z=0.0
17      twist.linear.x=linear_speed
18      self.velocity_publisher.publish(twist)
19      rospy.sleep(2)  # sleep after short linear movement
20      twist.linear.x=0.0
21      self.velocity_publisher.publish(twist)
22      print ('---Avoiding Small Object---')
23
24    elif(self.obstacle['obstacle'][3] == 2):  # large obstacle algorithm
25
26     twist.linear.x=-linear_speed
27     self.velocity_publisher.publish(twist)
28     rospy.sleep(3)# sleep after moving backwards
29      #Large object evasion
30     if(self.obstacle['obstacle'][1] > thr1 and self.obstacle['obstacle'
   ↪ ][2]  ):
31         twist.angular.z=angular_speed
32     else:
33       twist.angular.z=-angular_speed
34
35     twist.linear.x=0.0
36     self.velocity_publisher.publish(twist)
37     rospy.sleep(4)# sleep after angular twist
38     twist.angular.z=0.0
39     twist.linear.x=linear_speed
40     self.velocity_publisher.publish(twist)
41     rospy.sleep(3)  # sleep after short linear movement
42
43     twist.linear.x=0.0
44     self.velocity_publisher.publish(twist)
45     print ('---Avoiding Large Object---')
```

Listing 11: Obstacle_avoidance.py

4.3. Limitations.

1. The avoidance algorithm comes with its own clear limitations. While it is able to ensure the integrity of the robot and avoid any collisions, it can still meet scenarios where its avoidance algorithms are insufficient. Large detours caused by the tight corners and certain coordinates pairs require more advance rerouting strategies. Overcoming such scenarios could be achieved by using the internal memory of the robot of the environment it has scanned and using this obtained map to create more complex alternative routes.

2. Obstacles are detected under strict rules, but errors may occur during scanning or during the declaration of detected obstacles. Also the differentiation between large and small objects may not be satisfactory for all applications and further distinction parameters can be employed to better adapt to the environment.

3. All testing was done on a uniform plane. All issues with terrain dents and holes are not detected nor addressed by the algorithm. The strategy for large obstacle avoidance also assumes the robot has a clear path behind itself. The robot is therefore limited to flat areas.

5. Conclusions

Taking into account all limitations, the presented project manages to create a rudimentary form of patrolling and objection detection and avoidance with a grade of efficiency that can be greatly improved. The showcased programs enable the robot to navigate dynamically, autonomously avoiding obstacles while patrolling an environment.

Moving forward, enhancements and refinements to the current system could further improve the robot's performance and reliability. Additionally, exploring the incorporation of machine learning techniques for decision-making in complex environments could be a promising avenue for future development.

In summary, our endeavor to equip the TurtleBot3 Burger with patrolling and obstacle avoidance capabilities has been a significant step toward unlocking its potential in practical applications. This project not only advanced our understanding of robotics but also exemplified the adaptability and agility of the TurtleBot3 platform. As robotics continues to evolve, the capabilities demonstrated here pave the way for further innovation and exploration in autonomous navigation and obstacle avoidance for robotic systems.

6. References

1. TurtleBot3 Documentation
2. Quaternions Video Tutorial
3. How to find angle difference in Quaternions?
4. Package Creation Video Tutorial 1
5. Package Creation Video Tutorial 2
6. Packages ROS Tutorial
7. Subscriber and Publisher ROS tutorial