Autonomous Navigation using Bug2 Algorithm

Submitted by

Ankit Aggarwal	**Meet Jain**	**Srishti Gupta**	**Naman Bhat**
200929036	200929072	200929048	200929030

Under the guidance of

Dr ASHA C S

Associate Professor

Department of Mechatronics

MIT Manipal

in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

MECHATRONICS

DEPARTMENT OF MECHATRONICS

MANIPAL INSTITUTE OF TECHNOLOGY

(A Constituent of Manipal Academy of Higher Education)

MANIPAL - 576104, KARNATAKA, INDIA

June 2022

DEPARTMENT OF MECHATRONICS

Manipal
November 4, 2022

CERTIFICATE

This is to certify that the mini project titled **Autonomous Navigation using Bug2 Algorithm** is a record of work done by **Ankit Aggarwal (200929036)**, **Meet Jain (200929072)**, **Srishti Gupta (200929048)** and **Naman Bhat (200929030)** submitted for Robotics Lab II during the academic year 2022-2023.

Dr ASHA C S
Associate Professor
Dept. of Mechatronics
MIT Manipal

Dr Asha C S
Lab Coordinator
Dept. of Mechatronics
MIT Manipal

ACKNOWLEDGEMENTS

At the outset, the team would like to take this opportunity to thank everyone who assisted, supported, and inspired us throughout the course of the project to make this a success.

We would like to especially thank Dr Asha C S and Dr Jeanne D'Souza for being our mentors during this project,

ABSTRACT

Robotic indoor navigation of robots has been a sought-after topic for the last few decades within the robotic community. An important stimulus for this interest is its potential for a wide range of scenarios, e.g. search-and-rescue, greenhouse observation, industrial inspection. Indoor navigation also comes with a wide range of issues, such as the absence of a reliable GPS-signal and wall interference in long-range communication. An indoor robot should preferably be autonomous and able to navigate based on its on-board sensors and computational capacity. This project aims to approach this problem using the Bug2 navigation algorithm. It aims to make a fully autonomous bot capable of traversing to a given coordinate while avoiding obstacles in the path using the integration of a local planner with a a coordinate system to facilitate the input of goal coordinates.

LIST OF FIGURES

TABLE OF CONTENTS

CHAPTER 1
Introduction

With the continuous development of the advanced manufacturing industry and artificial intelligence, intelligent automatic operation has gradually replaced the traditional cooperative operation mode of mechanical production and manual supervision. The application of robots is gradually infiltrating into all fields of our life and forming a multi-cross and multi-integrated discipline. With the continuous development of advanced algorithms and sensor technology, mobile robots with an autonomous navigation ability are gradually applied in indoor and outdoor locations. A robot capable of autonomous mapping and navigation needs to have the ability of environmental perception, positioning, and path planning.

The Bug's algorithms are simple planners with provable guarantees. When they face an unknown obstacle, they are able to easily produce their own path contouring the object in the 2D surface if a path to the goal exists. The purpose is to generate a collision-free path by using the boundary-following and the motion-to-goal behaviors. In addition, the Bug's family has three assumptions about the mobile robot:

1. The robot is a point
2. The robot has perfect localization
3. The robot's sensors are precise

CHAPTER 2
Literature Review

As discussed in the preceding section, the problem of indoor autonomous navigation is well documented and many possible solutions exist. This section provides a brief overview of relevant examples and their impact, leading up to the case study in the following section.

2.1 Autonomous Navigation Algorithms

The paper chosen for Literature Review is, "**A Comparative Study of Bug Algorithm for Robot Navigation**". The authors of this paper compare different Bug algorithms and check their potential for robotic navigation. Bug is used instead of SLAM as it requires high computational power and can be expensive because of which it can not be used in smaller robotic applications. The authors compare different algorithms such as random-walker, which is largely based on luck, Pledge which can handle mazes with disjoint walls, but it cannot move directly to an exit as it has no knowledge of its position.

One of the reviewed algorithms was the common-sense algorithm. In this the robot moves towards the target whenever it can. The position where the robot hits the obstacle for the first time is called a hit-point, and it has a leave-point as soon as the direction to the target is free. Intuitively, Com could solve many situations; however, it was not reliable in complex environments.

Bug0 is an algorithm used for the same purpose however it doesn't not have memory which reduces its efficiency in real world problems. In this algorithm the robot moves towards the goal till it reaches an obstacle, and when it does it goes around the obstacle till it can continue on the same path to the goal.[1]

Bug1 is an algorithm in which it circumnavigates the obstacle and remembers the closest point to the goal and then it returns to that point. From that point it moves towards the goal. Bug1 is an exhaustive search algorithm and has more predictable performance overall. The Bug1 algorithm generally takes a longer path. Bug1 is better than the previously reviewed algorithms, however it is less intuitive and is not very efficient.

DistBug is a memory efficient algorithm which has been used for real world applications. In this algorithm it remembers only the last hit point. The implementation of this algorithm requires range sensors.

1. **Bug2 Algorithm:** The Bug2 algorithm is used when we have a mobile robot with the following conditions:

(a) Known starting location.

(b) Known goal location.

(c) Inside an unexplored location.

(d) Contains a distance sensor to detect distances to walls and other objects.

(e) Contains an encoder which estimates the distance travelled by robot from start.

The algorithm has 2 main modes of operation:

(a) Go-to-Goal Mode: In this mode the robot moves from the current location to the goal.

(b) Wall-Following Mode: In this mode the robot moves along a wall.

An imaginary line is created between the goal and the start point. The robot begins following the line until it hits a wall. This point is called a hit point. At the hit point the robot changes its path and follows the wall. The robot follows the wall until it is in the line of the start-goal imaginary line. This point is called leave point. When the robot comes to a leave point it begins following the imaginary line again until it hits a wall and the process repeats. Bug2 is a greedy search.

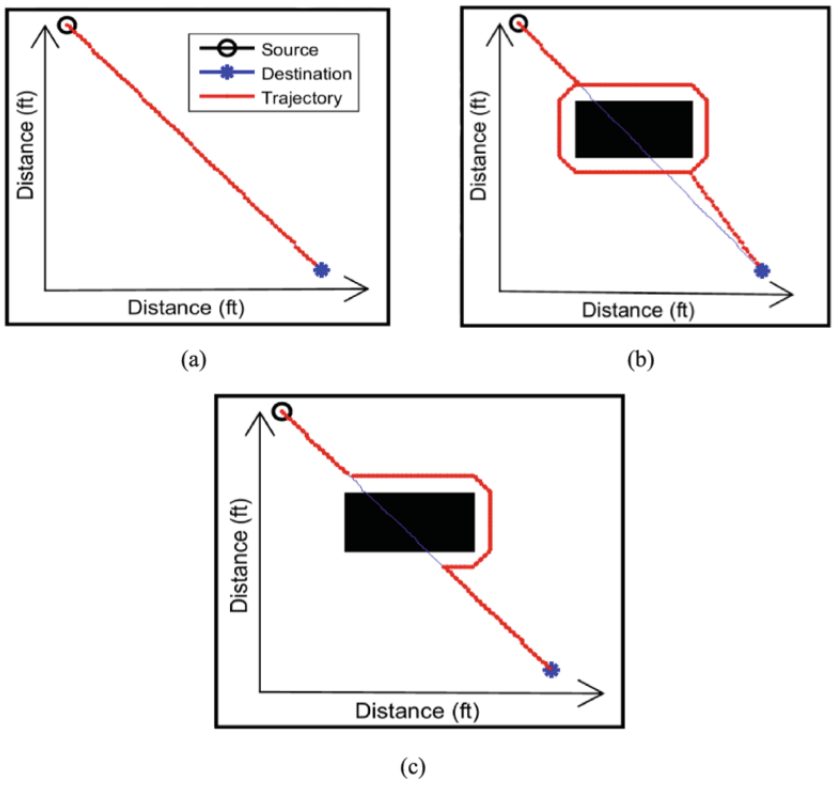

Fig. 2.1: Paths generated using Bug1 and Bug2 algorithms when an obstacle was encountered

2. **Bug1 and Bug0 Algorithms:** Bug 0 and Bug1 were algorithms before Bug2 which were used for the same application. Bug0 is the most basic Bug algorithm in which:

   (a) The obstacle moves towards the goal until an obstacle is met.

   (b) It then follows the boundary until it comes around it.

   Bug0 has no memory. Bug1 is more advanced as compared to Bug0, in which it circumnavigates the obstacle and remembers the closest point to the goal and then it returns to that point. Bug1 is an exhaustive search algorithm and has more predictable performance overall. The Bug1 algorithm generally takes a longer path [2].

2.2 ROS2 Navigation Stack

The ROS2 Navigation Stack is used in navigation applications such as ground delivery, hospitals, offices etc. Nav2 is a collection of software packages that can help in moving a robot from a starting point to a goal location. Some of the functions provided by Nav2 are:

1. Localize robot on a map
2. Plan path from A to B around obstacles
3. Control robot as it follows path
4. Follow sequential waypoints

CHAPTER 3
Methodology

3.1 Implementation of Bug2 Algorithm

Based on the literature review conducted, we decided to settle on using the Bug2 Algorithm to circumvent obstacles and plan optimal paths. [3] The pseudo-code for the algorithm is given as:

1. Calculate a start-goal line. The start-goal line is an imaginary line that connects the starting position to the goal position.
2. While Not at the Goal: Move towards the goal along the start-goal line. If a wall is encountered: Remember the location where the wall was first encountered. This is the "hit point." Follow the wall until you encounter the start-goal line. This point is known as the "leave point." If the leave point is closer to the goal than the hit point, leave the wall, and move towards the goal again. Otherwise, continue following the wall. [4]

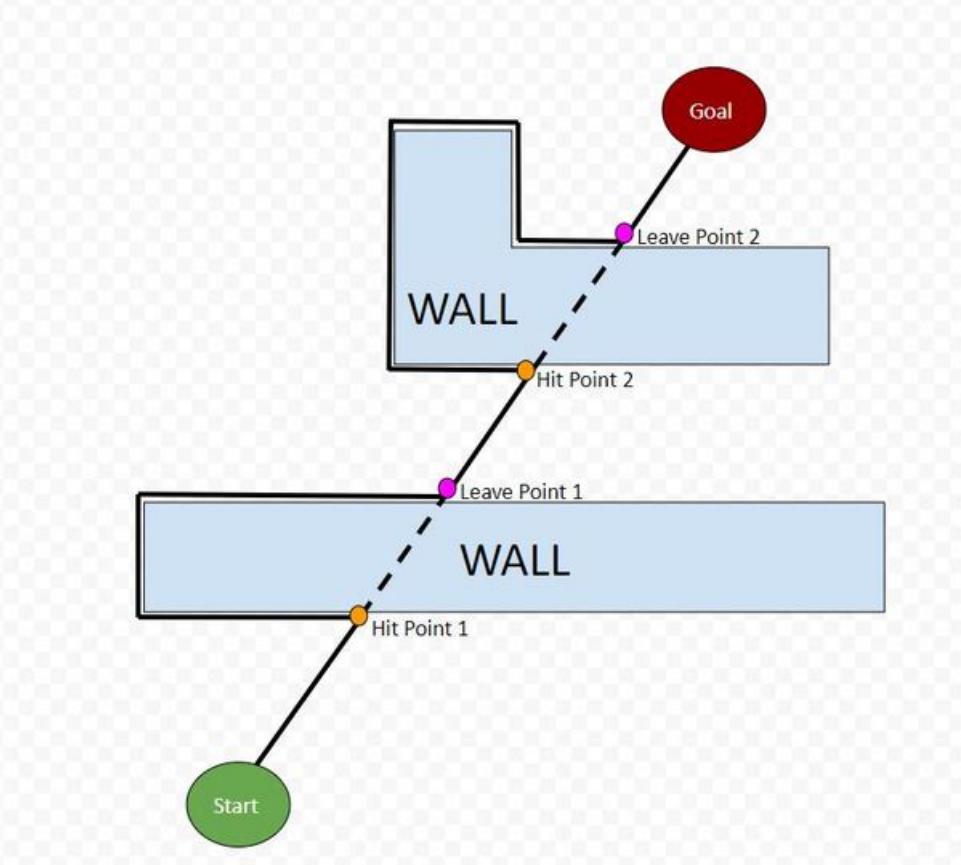

Fig. 3.1: Hit Points and Leave Points

3.2 Python Code for Bug2

```
############## IMPORT LIBRARIES #################
# Python math library
import math

# ROS client library for Python
import rclpy

# Enables pauses in the execution of code
from time import sleep

# Used to create nodes
from rclpy.node import Node

# Enables the use of the string message type
from std_msgs.msg import String

# Twist is linear and angular velocity
from geometry_msgs.msg import Twist

# Handles LaserScan messages to sense distance to obstacles (i.e. walls)
from sensor_msgs.msg import LaserScan

# Handle Pose messages
from geometry_msgs.msg import Pose

# Handle float64 arrays
from std_msgs.msg import Float64MultiArray

# Handles quality of service for LaserScan data
from rclpy.qos import qos_profile_sensor_data

# Scientific computing library
import numpy as np

class PlaceholderController(Node):
    """
    Create a Placeholder Controller class, which is a subclass
    of the Node class for ROS2.
    """

    def __init__(self):
        """
        Class constructor to set up the node
```

```
"""
#### INITIALIZE ROS PUBLISHERS AND SUBSCRIBERS ##
# Initiate the Node class's constructor and give it a name
super().__init__('PlaceholderController')

# Create a subscriber
# This node subscribes to messages of type Float64MultiArray
# over a topic named: /en613/state_est
# The message represents the current estimated state:
#   [x, y, yaw]
# The callback function is called as soon as a message
# is received.
# The maximum number of queued messages is 10.
self.subscription = self.create_subscription(
    Float64MultiArray,
    '/en613/state_est',
    self.state_estimate_callback,
    10)
self.subscription  # prevent unused variable warning

# Create a subscriber
# This node subscribes to messages of type
# sensor_msgs/LaserScan
self.scan_subscriber = self.create_subscription(
    LaserScan,
    '/en613/scan',
    self.scan_callback,
    qos_profile=qos_profile_sensor_data)

# Create a subscriber
# This node subscribes to messages of type geometry_msgs/Pose
# over a topic named: /en613/goal
# The message represents the the goal position.
# The callback function is called as soon as a message
# is received.
# The maximum number of queued messages is 10.
self.subscription_goal_pose = self.create_subscription(
    Pose,
    '/en613/goal',
    self.pose_received,
    10)

# Create a publisher
# This node publishes the desired linear and angular velocity
# of the robot (in the robot chassis coordinate frame) to the
# /en613/cmd_vel topic. Using the diff_drive
```

```
# plugin enables the basic_robot model to read this
# /end613/cmd_vel topic and execute the motion accordingly.
self.publisher_ = self.create_publisher(
    Twist,
    '/en613/cmd_vel',
    10)

# Initialize the LaserScan sensor readings to some large value
# Values are in meters.
self.left_dist = 999999.9 # Left
self.leftfront_dist = 999999.9 # Left-front
self.front_dist = 999999.9 # Front
self.rightfront_dist = 999999.9 # Right-front
self.right_dist = 999999.9 # Right

######### ROBOT CONTROL PARAMETERS #########

# Maximum forward speed of the robot in meters per second
# Any faster than this and the robot risks falling over.
self.forward_speed = 0.035

# Current position and orientation of the robot in the global
# reference frame
self.current_x = 0.0
self.current_y = 0.0
self.current_yaw = 0.0

# By changing the value of self.robot_mode, you can alter what
# the robot will do when the program is launched.
#   "obstacle avoidance mode": Robot will avoid obstacles
#   "go to goal mode": Robot will head to an x,y coordinate
#   "wall following mode": Robot will follow a wall
self.robot_mode = "go to goal mode"

####### OBSTACLE AVOIDANCE MODE PARAMETERS ######

# Obstacle detection distance threshold
self.dist_thresh_obs = 0.25 # in meters

# Maximum left-turning speed
self.turning_speed = 0.25 # rad/s

####### GO TO GOAL MODE PARAMETERS #######
# Finite states for the go to goal mode
#   "adjust heading": Orient towards a goal x, y coordinate
#   "go straight": Go straight towards goal x, y coordinate
```

```
#   "goal achieved": Reached goal x, y coordinate
self.go_to_goal_state = "adjust heading"

# List the goal destinations
# We create a list of the (x,y) coordinate goals
self.goal_x_coordinates = False # [ 0.0, 3.0, 0.0, -1.5, -1.5,  4.5, 0.0]
self.goal_y_coordinates = False # [-4.0, 1.0, 1.5,  1.0, -3.0, -4.0, 0.0]

# Keep track of which goal we're headed towards
self.goal_idx = 0

# Keep track of when we've reached the end of the goal list
self.goal_max_idx =  None # len(self.goal_x_coordinates) - 1

# +/- 2.0 degrees of precision
self.yaw_precision = 2.0 * (math.pi / 180)

# How quickly we need to turn when we need to make a heading
# adjustment (rad/s)
self.turning_speed_yaw_adjustment = 0.0625

# Need to get within +/- 0.2 meter (20 cm) of (x,y) goal
self.dist_precision = 0.2

############# WALL FOLLOWING MODE PARAMETERS ##################
# Finite states for the wall following mode
#   "turn left": Robot turns towards the left
#   "search for wall": Robot tries to locate the wall
#   "follow wall": Robot moves parallel to the wall
self.wall_following_state = "turn left"

# Set turning speeds (to the left) in rad/s
# These values were determined by trial and error.
self.turning_speed_wf_fast = 1.0  # Fast turn
self.turning_speed_wf_slow = 0.125 # Slow turn

# Wall following distance threshold.
# We want to try to keep within this distance from the wall.
self.dist_thresh_wf = 0.45 # in meters

# We don't want to get too close to the wall though.
self.dist_too_close_to_wall = 0.15 # in meters

################## BUG2 PARAMETERS ############################

# Bug2 Algorithm Switch
```

```
# Can turn "ON" or "OFF" depending on if you want to run Bug2
# Motion Planning Algorithm
self.bug2_switch = "ON"

# Start-Goal Line Calculated?
self.start_goal_line_calculated = False

# Start-Goal Line Parameters
self.start_goal_line_slope_m = 0
self.start_goal_line_y_intercept = 0
self.start_goal_line_xstart = 0
self.start_goal_line_xgoal = 0
self.start_goal_line_ystart = 0
self.start_goal_line_ygoal = 0

# Anything less than this distance means we have encountered
# a wall. Value determined through trial and error.
self.dist_thresh_bug2 = 0.15

# Leave point must be within +/- 0.1m of the start-goal line
# in order to go from wall following mode to go to goal mode
self.distance_to_start_goal_line_precision = 0.1

# Used to record the (x,y) coordinate where the robot hit
# a wall.
self.hit_point_x = 0
self.hit_point_y = 0

# Distance between the hit point and the goal in meters
self.distance_to_goal_from_hit_point = 0.0

# Used to record the (x,y) coordinate where the robot left
# a wall.
self.leave_point_x = 0
self.leave_point_y = 0

# Distance between the leave point and the goal in meters
self.distance_to_goal_from_leave_point = 0.0

# The hit point and leave point must be far enough
# apart to change state from wall following to go to goal
# This value helps prevent the robot from getting stuck and
# rotating in endless circles.
# This distance was determined through trial and error.
self.leave_point_to_hit_point_diff = 0.25 # in meters
```

```
def pose_received(self,msg):
    """
    Populate the pose.
    """
    self.goal_x_coordinates = [msg.position.x]
    self.goal_y_coordinates = [msg.position.y]
    self.goal_max_idx = len(self.goal_x_coordinates) - 1

def scan_callback(self, msg):
    """
    This method gets called every time a LaserScan message is
    received on the /en613/scan ROS topic
    """
    # Read the laser scan data that indicates distances
    # to obstacles (e.g. wall) in meters and extract
    # 5 distinct laser readings to work with.
    # Each reading is separated by 45 degrees.
    # Assumes 181 laser readings, separated by 1 degree.
    # (e.g. -90 degrees to 90 degrees....0 to 180 degrees)
    self.left_dist = msg.ranges[180]
    self.leftfront_dist = msg.ranges[135]
    self.front_dist = msg.ranges[90]
    self.rightfront_dist = msg.ranges[45]
    self.right_dist = msg.ranges[0]

    # The total number of laser rays. Used for testing.
    #number_of_laser_rays = str(len(msg.ranges))

    # Print the distance values (in meters) for testing
    #self.get_logger().info('L:%f LF:%f F:%f RF:%f R:%f' % (
    #   self.left_dist,
    #   self.leftfront_dist,
    #   self.front_dist,
    #   self.rightfront_dist,
    #   self.right_dist))

    if self.robot_mode == "obstacle avoidance mode":
        self.avoid_obstacles()

def state_estimate_callback(self, msg):
    """
    Extract the position and orientation data.
    This callback is called each time
    a new message is received on the '/en613/state_est' topic
    """
    # Update the current estimated state in the global reference frame
```

```
        curr_state = msg.data
        self.current_x = curr_state[0]
        self.current_y = curr_state[1]
        self.current_yaw = curr_state[2]

        # Wait until we have received some goal destinations.
        if self.goal_x_coordinates == False and self.goal_y_coordinates == False:
            return

        # Print the pose of the robot
        # Used for testing
        #self.get_logger().info('X:%f Y:%f YAW:%f' % (
        #   self.current_x,
        #   self.current_y,
        #   np.rad2deg(self.current_yaw)))  # Goes from -pi to pi

        # See if the Bug2 algorithm is activated. If yes, call bug2()
        if self.bug2_switch == "ON":
            self.bug2()
        else:

            if self.robot_mode == "go to goal mode":
                self.go_to_goal()
            elif self.robot_mode == "wall following mode":
                self.follow_wall()
            else:
                pass # Do nothing

    def avoid_obstacles(self):
        """
        Wander around the maze and avoid obstacles.
        """
        # Create a Twist message and initialize all the values
        # for the linear and angular velocities
        msg = Twist()
        msg.linear.x = 0.0
        msg.linear.y = 0.0
        msg.linear.z = 0.0
        msg.angular.x = 0.0
        msg.angular.y = 0.0
        msg.angular.z = 0.0

        # Logic for avoiding obstacles (e.g. walls)
        # >d means no obstacle detected by that laser beam
        # <d means an obstacle was detected by that laser beam
        d = self.dist_thresh_obs
```

```
        if   self.leftfront_dist > d and self.front_dist > d and self.rightfront_dist
        > d:
            msg.linear.x = self.forward_speed # Go straight forward
        elif self.leftfront_dist > d and self.front_dist < d and self.rightfront_dist
        > d:
            msg.angular.z = self.turning_speed  # Turn left
        elif self.leftfront_dist > d and self.front_dist > d and self.rightfront_dist
        < d:
            msg.angular.z = self.turning_speed
        elif self.leftfront_dist < d and self.front_dist > d and self.rightfront_dist
        > d:
            msg.angular.z = -self.turning_speed # Turn right
        elif self.leftfront_dist > d and self.front_dist < d and self.rightfront_dist
        < d:
            msg.angular.z = self.turning_speed
        elif self.leftfront_dist < d and self.front_dist < d and self.rightfront_dist
        > d:
            msg.angular.z = -self.turning_speed
        elif self.leftfront_dist < d and self.front_dist < d and self.rightfront_dist
        < d:
            msg.angular.z = self.turning_speed
        elif self.leftfront_dist < d and self.front_dist > d and self.rightfront_dist
        < d:
            msg.linear.x = self.forward_speed
        else:
            pass

        # Send the velocity commands to the robot by publishing
        # to the topic
        self.publisher_.publish(msg)

    def go_to_goal(self):
        """
        This code drives the robot towards to the goal destination
        """
        # Create a geometry_msgs/Twist message
        msg = Twist()
        msg.linear.x = 0.0
        msg.linear.y = 0.0
        msg.linear.z = 0.0
        msg.angular.x = 0.0
        msg.angular.y = 0.0
        msg.angular.z = 0.0

        # If Bug2 algorithm is activated
        if self.bug2_switch == "ON":
```

```
        # If the wall is in the way
        d = self.dist_thresh_bug2
        if (    self.leftfront_dist < d or
            self.front_dist < d or
            self.rightfront_dist < d):

            # Change the mode to wall following mode.
            self.robot_mode = "wall following mode"

            # Record the hit point
            self.hit_point_x = self.current_x
            self.hit_point_y = self.current_y

            # Record the distance to the goal from the
            # hit point
            self.distance_to_goal_from_hit_point = (
                math.sqrt((
                pow(self.goal_x_coordinates[self.goal_idx] -
                self.hit_point_x, 2)) + (
                pow(self.goal_y_coordinates[self.goal_idx] -
                self.hit_point_y, 2))))

            # Make a hard left to begin following wall
            msg.angular.z = self.turning_speed_wf_fast

            # Send command to the robot
            self.publisher_.publish(msg)

            # Exit this function
            return

    # Fix the heading
    if (self.go_to_goal_state == "adjust heading"):

        # Calculate the desired heading based on the current position
        # and the desired position
        desired_yaw = math.atan2(
                self.goal_y_coordinates[self.goal_idx] - self.current_y,
                self.goal_x_coordinates[self.goal_idx] - self.current_x)

        # How far off is the current heading in radians?
        yaw_error = desired_yaw - self.current_yaw

        # Adjust heading if heading is not good enough
        if math.fabs(yaw_error) > self.yaw_precision:
```

```
            if yaw_error > 0:
                # Turn left (counterclockwise)
                msg.angular.z = self.turning_speed_yaw_adjustment
            else:
                # Turn right (clockwise)
                msg.angular.z = -self.turning_speed_yaw_adjustment

            # Command the robot to adjust the heading
            self.publisher_.publish(msg)

        # Change the state if the heading is good enough
        else:
            # Change the state
            self.go_to_goal_state = "go straight"

            # Command the robot to stop turning
            self.publisher_.publish(msg)

    # Go straight
    elif (self.go_to_goal_state == "go straight"):

        position_error = math.sqrt(
                    pow(
                    self.goal_x_coordinates[self.goal_idx] - self.current_x, 2)
                    + pow(
                    self.goal_y_coordinates[self.goal_idx] - self.current_y, 2))


        # If we are still too far away from the goal
        if position_error > self.dist_precision:

            # Move straight ahead
            msg.linear.x = self.forward_speed

            # Command the robot to move
            self.publisher_.publish(msg)

            # Check our heading
            desired_yaw = math.atan2(
                self.goal_y_coordinates[self.goal_idx] - self.current_y,
                self.goal_x_coordinates[self.goal_idx] - self.current_x)

            # How far off is the heading?
            yaw_error = desired_yaw - self.current_yaw
```

```
            # Check the heading and change the state
            if math.fabs(yaw_error) > self.yaw_precision:

                # Change the state
                self.go_to_goal_state = "adjust heading"

        # We reached our goal. Change the state.
        else:
            # Change the state
            self.go_to_goal_state = "goal achieved"

            # Command the robot to stop
            self.publisher_.publish(msg)

    # Goal achieved
    elif (self.go_to_goal_state == "goal achieved"):

        self.get_logger().info('Goal achieved! X:%f Y:%f' % (
            self.goal_x_coordinates[self.goal_idx],
            self.goal_y_coordinates[self.goal_idx]))

        # Get the next goal
        self.goal_idx = self.goal_idx + 1

        # Do we have any more goals left?
        # If we have no more goals left, just stop
        if (self.goal_idx > self.goal_max_idx):
            self.get_logger().info('Congratulations! All goals have
            been achieved.')
            while True:
                pass

        # Let's achieve our next goal
        else:
            # Change the state
            self.go_to_goal_state = "adjust heading"

        # We need to recalculate the start-goal line if Bug2 is running
        self.start_goal_line_calculated = False

    else:
        pass

def follow_wall(self):
    """
    This method causes the robot to follow the boundary of a wall.
```

```
"""
# Create a geometry_msgs/Twist message
msg = Twist()
msg.linear.x = 0.0
msg.linear.y = 0.0
msg.linear.z = 0.0
msg.angular.x = 0.0
msg.angular.y = 0.0
msg.angular.z = 0.0

# Special code if Bug2 algorithm is activated
if self.bug2_switch == "ON":

    # Calculate the point on the start-goal
    # line that is closest to the current position
    x_start_goal_line = self.current_x
    y_start_goal_line = (
        self.start_goal_line_slope_m * (
        x_start_goal_line)) + (
        self.start_goal_line_y_intercept)

    # Calculate the distance between current position
    # and the start-goal line
    distance_to_start_goal_line = math.sqrt(pow(
                x_start_goal_line - self.current_x, 2) + pow(
                y_start_goal_line - self.current_y, 2))

    # If we hit the start-goal line again
    if distance_to_start_goal_line <
    self.distance_to_start_goal_line_precision:

        # Determine if we need to leave the wall and change the mode
        # to 'go to goal'
        # Let this point be the leave point
        self.leave_point_x = self.current_x
        self.leave_point_y = self.current_y

        # Record the distance to the goal from the leave point
        self.distance_to_goal_from_leave_point = math.sqrt(
            pow(self.goal_x_coordinates[self.goal_idx]
            - self.leave_point_x, 2)
            + pow(self.goal_y_coordinates[self.goal_idx]
            - self.leave_point_y, 2))

        # Is the leave point closer to the goal than the hit point?
        # If yes, go to goal.
```

```
            diff = self.distance_to_goal_from_hit_point - self.distance_to_goal_from
            if diff > self.leave_point_to_hit_point_diff:

                # Change the mode. Go to goal.
                self.robot_mode = "go to goal mode"


            # Exit this function
            return

    # Logic for following the wall
    # >d means no wall detected by that laser beam
    # <d means an wall was detected by that laser beam
    d = self.dist_thresh_wf

    if self.leftfront_dist > d and self.front_dist > d and self.rightfront_dist
    > d:
        self.wall_following_state = "search for wall"
        msg.linear.x = self.forward_speed
        msg.angular.z = -self.turning_speed_wf_slow # turn right to find wall

    elif self.leftfront_dist > d and self.front_dist < d and self.rightfront_dist
    > d:
        self.wall_following_state = "turn left"
        msg.angular.z = self.turning_speed_wf_fast


    elif (self.leftfront_dist > d and self.front_dist > d and self.rightfront_dist
    < d):
        if (self.rightfront_dist < self.dist_too_close_to_wall):
            # Getting too close to the wall
            self.wall_following_state = "turn left"
            msg.linear.x = self.forward_speed
            msg.angular.z = self.turning_speed_wf_fast
        else:
            # Go straight ahead
            self.wall_following_state = "follow wall"
            msg.linear.x = self.forward_speed

    elif self.leftfront_dist < d and self.front_dist > d and self.rightfront_dist
    > d:
        self.wall_following_state = "search for wall"
        msg.linear.x = self.forward_speed
        msg.angular.z = -self.turning_speed_wf_slow # turn right to find wall

    elif self.leftfront_dist > d and self.front_dist < d and self.rightfront_dist
```

```
        < d:
            self.wall_following_state = "turn left"
            msg.angular.z = self.turning_speed_wf_fast

        elif self.leftfront_dist < d and self.front_dist < d and self.rightfront_dist
        > d:
            self.wall_following_state = "turn left"
            msg.angular.z = self.turning_speed_wf_fast

        elif self.leftfront_dist < d and self.front_dist < d and self.rightfront_dist
        < d:
            self.wall_following_state = "turn left"
            msg.angular.z = self.turning_speed_wf_fast

        elif self.leftfront_dist < d and self.front_dist > d and self.rightfront_dist
        < d:
            self.wall_following_state = "search for wall"
            msg.linear.x = self.forward_speed
            msg.angular.z = -self.turning_speed_wf_slow # turn right to find wall

        else:
            pass

        # Send velocity command to the robot
        self.publisher_.publish(msg)

    def bug2(self):

        # Each time we start towards a new goal, we need
        # to calculate the start-goal line
        if self.start_goal_line_calculated == False:

            # Make sure go to goal mode is set.
            self.robot_mode = "go to goal mode"

            self.start_goal_line_xstart = self.current_x
            self.start_goal_line_xgoal = self.goal_x_coordinates[self.goal_idx]
            self.start_goal_line_ystart = self.current_y
            self.start_goal_line_ygoal = self.goal_y_coordinates[self.goal_idx]

            # Calculate the slope of the start-goal line m
            self.start_goal_line_slope_m = (
                (self.start_goal_line_ygoal - self.start_goal_line_ystart) / (
                self.start_goal_line_xgoal - self.start_goal_line_xstart))

            # Solve for the intercept b
```

```
            self.start_goal_line_y_intercept = self.start_goal_line_ygoal - (
                    self.start_goal_line_slope_m * self.start_goal_line_xgoal)

            # We have successfully calculated the start-goal line
            self.start_goal_line_calculated = True

        if self.robot_mode == "go to goal mode":
            self.go_to_goal()
        elif self.robot_mode == "wall following mode":
            self.follow_wall()
def main(args=None):

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

    # Create the node
    controller = PlaceholderController()

    # Spin the node so the callback function is called
    # Pull messages from any topics this node is subscribed to
    # Publish any pending messages to the topics
    rclpy.spin(controller)

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

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

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

3.3 URDF Model

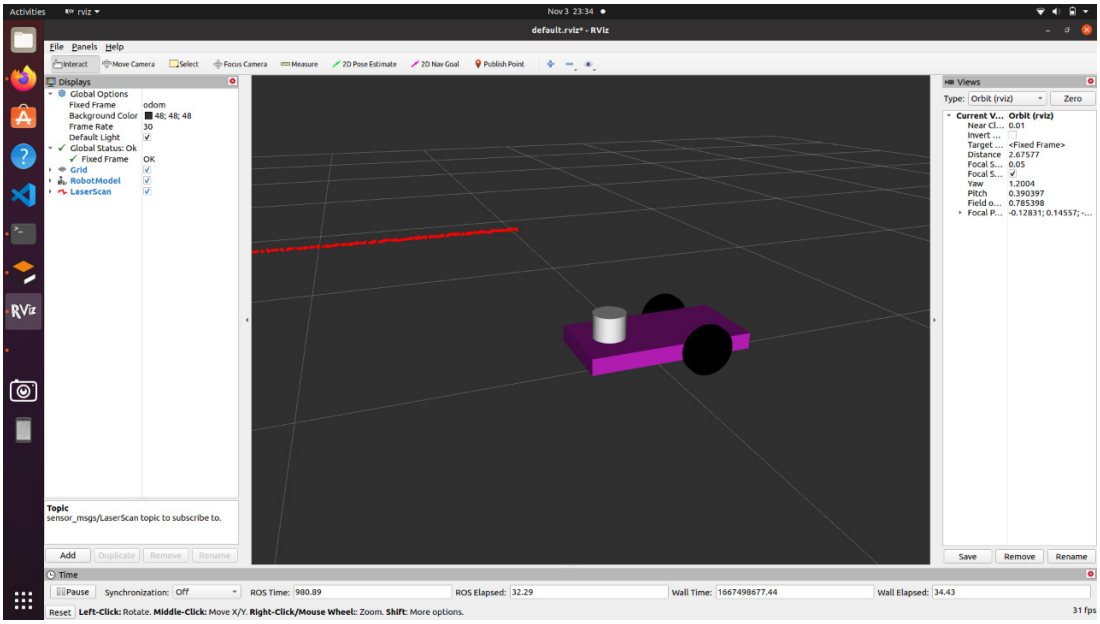

Fig. 3.2: Bot used for simulation

The bot was programmed with a differential drive controller, with a command topic of cmd vel. The LiDAR plugin used was a ray type LiDAR with a range of 0.12 to 3.5, having a resolution of 0.015.

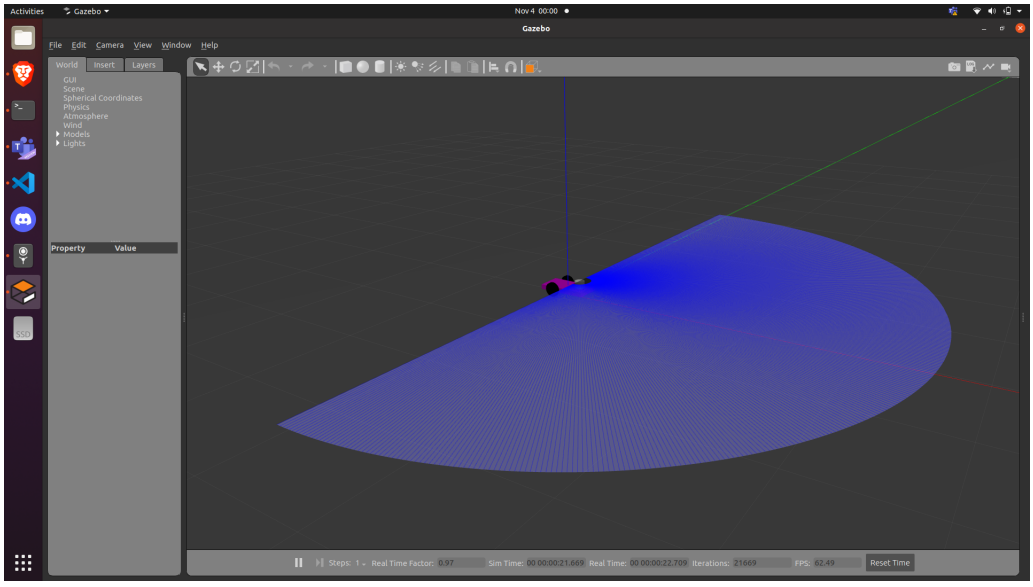

Fig. 3.3: Bot spawned in Gazebo

3.4 Gazebo Implementation

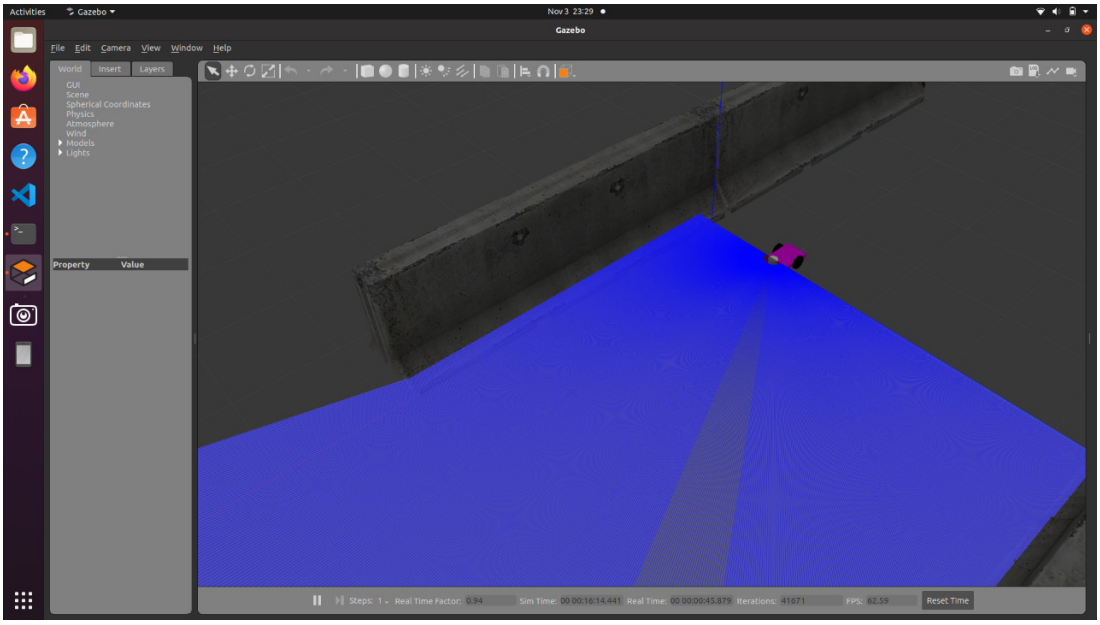

Fig. 3.4: Simulated Environment with Bot

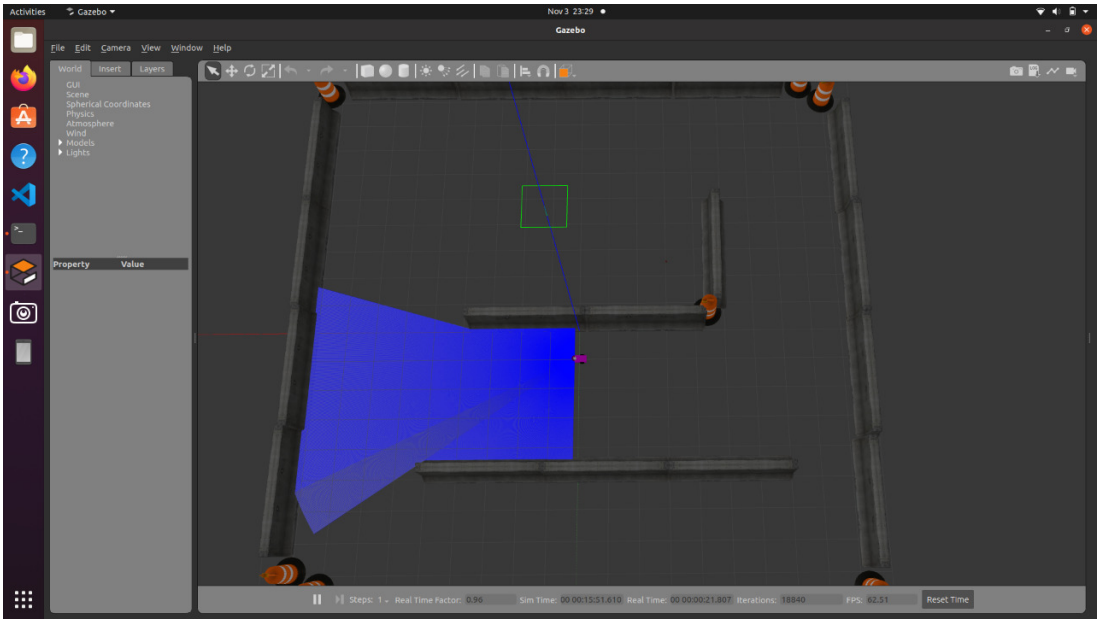

Fig. 3.5: Bot with Bug2 Algorithm

3.5 ROS2 Implementation

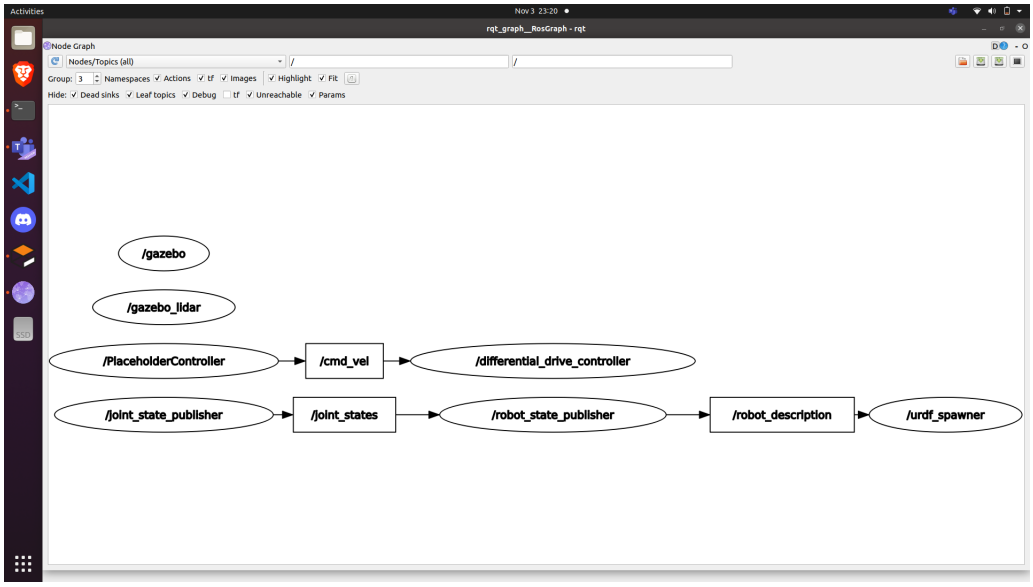

Fig. 3.6: RQT Graph showing all Topics

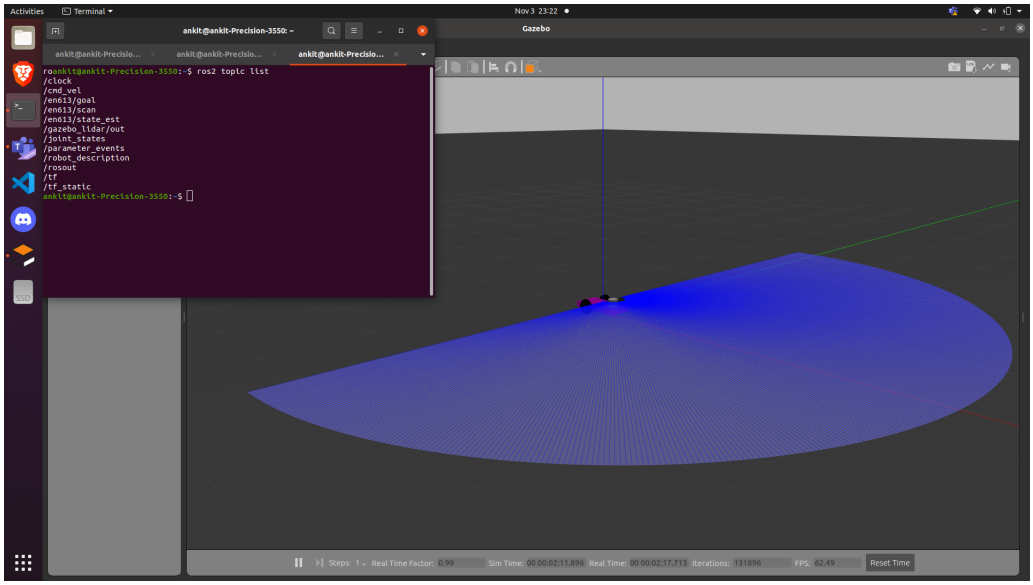

Fig. 3.7: All active topics during simulation

CHAPTER 4
Contribution of Each Student

Ankit Aggarwal

Contributions

- **Research on Differential Drive Controller**
- **Created Launch file for Simulation**
- **Implementation on Gazebo**

Analysis

If both the wheels are driven in the same direction and speed, the robot will go in a straight line. If both wheels are turned with equal speed in opposite directions, the robot will rotate about the central point of the axis. Since the direction of the robot is dependent on the rate and direction of rotation of the two driven wheels, these quantities should be sensed and controlled precisely. Algorithms were tested multiple times in Gazebo, that is in a virtual environment to debug errors and ensure smooth functioning of robot in the simulation.

Srishti Gupta

Contributions

- **Research on Algorithms to be used for Navigation Stack**
- **Optimised Navigation Algorithm**
- **Created the URDF model**

Analysis

Bug1 is an exhaustive search algorithm, it looks at all choices before committing. Bug2 is a greedy algorithm, it takes the first opportunity that looks better. In many cases, Bug2 will out perform Bug1 but Bug1 has a more predictable performance overall. The Bug2 Algorithm was implemented according to the drive system chosen. A CAD model is designed and converted into URDF file using SolidWorks.

Naman Bhat

Contributions

- **Completed all Courses**
- **Conducted extensive Literature Review**
- **Compiled research papers related to this project**

Analysis

Coursera Course - Collaborative Robot Safety: Design & Deployment. This course equips you to assess the safety of a collaborative robot workcell and prevent the chances of injury or harm. It imparts industry-endorsed safety standards, technical report recommendations and best practices from the International Organization for Standardization (ISO), Robotic Industries Association (RIA) and Occupational Safety and Health Administration (OSHA).

Meet Jain

Contributions

- **Researched upon Sensors to be used**
- **Created report in Overleaf and PPT for presentation**

Analysis

A distance sensor is required that can detect the distances to objects and walls in the environment (e.g. like an ultrasonic sensor or a laser distance sensor.) A Wheel encoder that the robot can use to estimate how far the robot has traveled from the starting location.

CHAPTER 5
Results and Discussion

5.1 Fault Analysis

Throughout the project, numerous obstacles arose at each level, and new approaches to problem solving were required. Below we address the challenges faced:

- **Package build error**
- **Debugging code of Bug2 Algorithm**
- **Launch file for a Gazebo World**

5.1.1 Concept Revision

The simulation of point to point bug algorithm is carried out using ROS2. The algorithm is simulated on maze based environment in Gazebo. The algorithm operates in dynamic environment because information about the environment can be obtained immediately from the range sensor during the movement of the robot. The performance of the algorithm depends on total sudden points detected. The less number of sudden points detected is better.

CHAPTER 6
Conclusion and Future Scope

6.1 Conclusion

This project shows a 2 wheeled differential drive system robot. Only 3D Lidar and encoders are utilized as input commands in the system. The main objective of this study is to build a autonomous robot which can navigate through a maze based environment effortlessly. The new automatic control approach has been set up to navigate the robot to a desired destination, in unknown environments with avoiding obstacles. This approach gives the user the ability to look at the surrounding environments during the navigation process. Bug2 algorithm is simulated and implemented to control the robot in auto navigation method. The simulation shows that the user can enjoy looking around while the auto controlling method navigate to the goal point with avoiding obstacles, however there are cases where a manual controlling method is more efficient to use. However, Bug2 is applied due to its cost effectiveness and simplicity. [5]

Note: For references, a few more online resources were used that are included in the zip folder.

Bibliography

[1] N. Buniyamin, W. W. Ngah, N. Sariff, Z. Mohamad, *et al.*, "A simple local path planning algorithm for autonomous mobile robots," *International journal of systems applications, Engineering & development*, vol. 5, no. 2, pp. 151–159, 2011.

[2] K. Mcguire, G. Croon, and K. Tuyls, "A comparative study of bug algorithms for robot navigation," *Robotics and Autonomous Systems*, vol. 121, p. 103 261, Aug. 2019. DOI: 10.1016/j.robot.2019.103261.

[3] A. Al-Haddad, R. Sudirman, C. Omar, K. Y. Hui, and M. R. bin Jimin, "Wheelchair motion control guide using eye gaze and blinks based on bug 2 algorithm," in *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*, IEEE, vol. 2, 2012, pp. 438–443.

[4] J. Zhao, S. Liu, and J. Li, *Research and implementation of autonomous navigation for mobile robots based on slam algorithm under ros*, May 2022. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9185640/.

[5] A. Yufka and O. Parlaktuna, "Performance comparison of bug algorithms for mobile robots," in *Proceedings of the 5th international advanced technologies symposium, Karabuk, Turkey*, 2009, pp. 13–15.