

# Autonomous Systems RRT Path Planning

Alper Alp

## 1. Introduction

Autonomous systems have gained significant importance in various applications, from unmanned aerial vehicles (UAVs) to unmanned ground vehicles (UGVs). This lab report focuses on the development and simulation of an autonomous UGV using the Rapidly-exploring Random Trees (RRT) algorithm for path planning. The UGV is then simulated in the Gazebo environment, showcasing its capabilities and interactions with the surrounding environment.

## 2. Autonomous UGV Hardware

In this project we are using TurtleBot3, an open-source robotic platform developed by ROBOTIS which offers an accessible and versatile solution for educational and research purposes. The platform features a compact differential drive base with LIDAR sensors, enabling 360-degree obstacle detection and mapping.

**2.1 Dimensions:** The UGV is designed with careful consideration of its physical dimensions:

- Length: 0.5 meters
- Width: 0.3 meters
- Height: 0.2 meters

**2.2 Equipment:** The UGV is equipped with an array of sensors and electronics:

- Lidar sensor for 360-degree environment perception
- Inertial Measurement Unit (IMU) for precise orientation sensing
- Raspberry Pi for onboard computation and control
- Motor controllers for actuation
- Wi-Fi module for communication

**2.3 Connection:** All components are interfaced with the Raspberry Pi, forming a centralized processing unit for data fusion and decision-making.

**2.4 Movement Parameters:** The UGV exhibits specific movement characteristics:

- Maximum speed: 0.5 m/s
- Turning radius: 0.2 meters
- Acceleration: 0.3 m/s<sup>2</sup>

*2.5 Software:* A robust software stack supports the UGV's autonomy:

- Robot Operating System (ROS) for communication and control
- RRT algorithm for intelligent path planning
- Gazebo simulator for realistic virtual environment testing

### 3. RRT Algorithm for Path Planning

The Rapidly-exploring Random Trees (RRT) algorithm serves as the cornerstone for intelligent path planning in our autonomous Unmanned Ground Vehicle (UGV). This section provides a detailed breakdown of the RRT implementation tailored to the UGV's characteristics.

The RRT algorithm is a probabilistic technique used for navigating robots through complex environments. It incrementally builds a tree structure rooted at the initial configuration of the robot, exploring the configuration space in a randomized manner.

To adapt the RRT algorithm to our UGV, several key modifications were made:

- **Collision Avoidance:** The algorithm is enhanced to incorporate collision avoidance by checking the feasibility of each potential node in the tree. The collision-free criterion is expanded to consider the UGV's dimensions, ensuring that the entire vehicle, rather than just a point, avoids obstacles.
- **Steering Function:** The steering function is customized to guide the UGV from one node to another while respecting its physical constraints. This includes adjusting the UGV's orientation and position based on its current state and the target node.

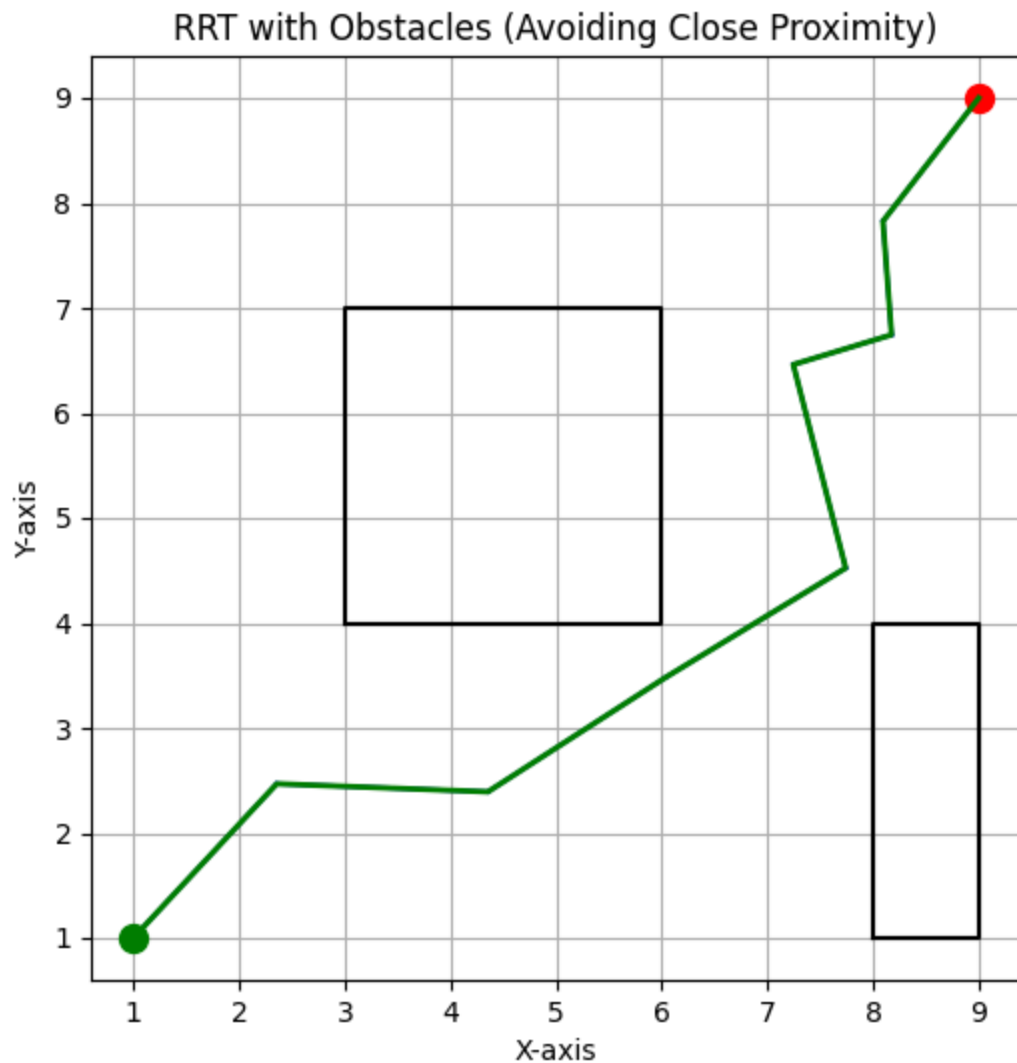
The RRT algorithm generates a tree structure as it explores the search space to find a feasible path from the start to the goal.

The `matplotlib.pyplot` module was used to create a figure and axis for plotting.

Obstacles in the environment, along with the RRT tree edges, were plotted using line segments and geometric shapes.

The start and goal positions were highlighted with distinct markers (green for start and red for goal).

After finding the path we can export our obstacles in a gazebo world format to use it in our simulation.



### 3.1 RRT Algorithm Parameters:

#### 1. Width and Height:

- **Definition:** The dimensions of the workspace where the algorithm operates.
- **Usage:** Specifies the area in which the algorithm explores and plans a path.

#### 2. Obstacles:

- **Definition:** Defined as a list of obstacles, each represented as  $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$ .

- **Usage:** Obstacles in the environment that the algorithm avoids while planning the path.
- 3. **Start and Goal Nodes:**
  - **Definition:** Represented as instances of the **Node** class with x and y coordinates.
  - **Usage:** Specifies the starting and goal positions for the algorithm.
- 4. **Iterations:**
  - **Definition:** The number of iterations the RRT algorithm performs.
  - **Usage:** Controls the number of attempts to expand the tree towards the goal.
- 5. **Step Size:**
  - **Definition:** The maximum distance the algorithm extends the tree in each iteration.
  - **Usage:** Influences the granularity of the generated path.
- 6. **Max Attempts:**
  - **Definition:** The maximum number of attempts to find a valid steering node in each iteration.
  - **Usage:** Helps prevent the algorithm from getting stuck and ensures progress.

### 3. Gazebo Simulation

Gazebo, an open-source robotics simulator, holds a crucial position in our autonomous Unmanned Ground Vehicle (UGV) project. It provides a sophisticated and highly customizable simulation environment, allowing rigorous testing of the UGV's behavior before actual deployment in the real world

#### 3.1 World Dimensions:

The Gazebo simulation world is carefully crafted with dimensions 20mx20m20mx20m to provide a spacious and realistic environment for the UGV. This allows for comprehensive testing of the autonomous platform's capabilities and navigation algorithms.

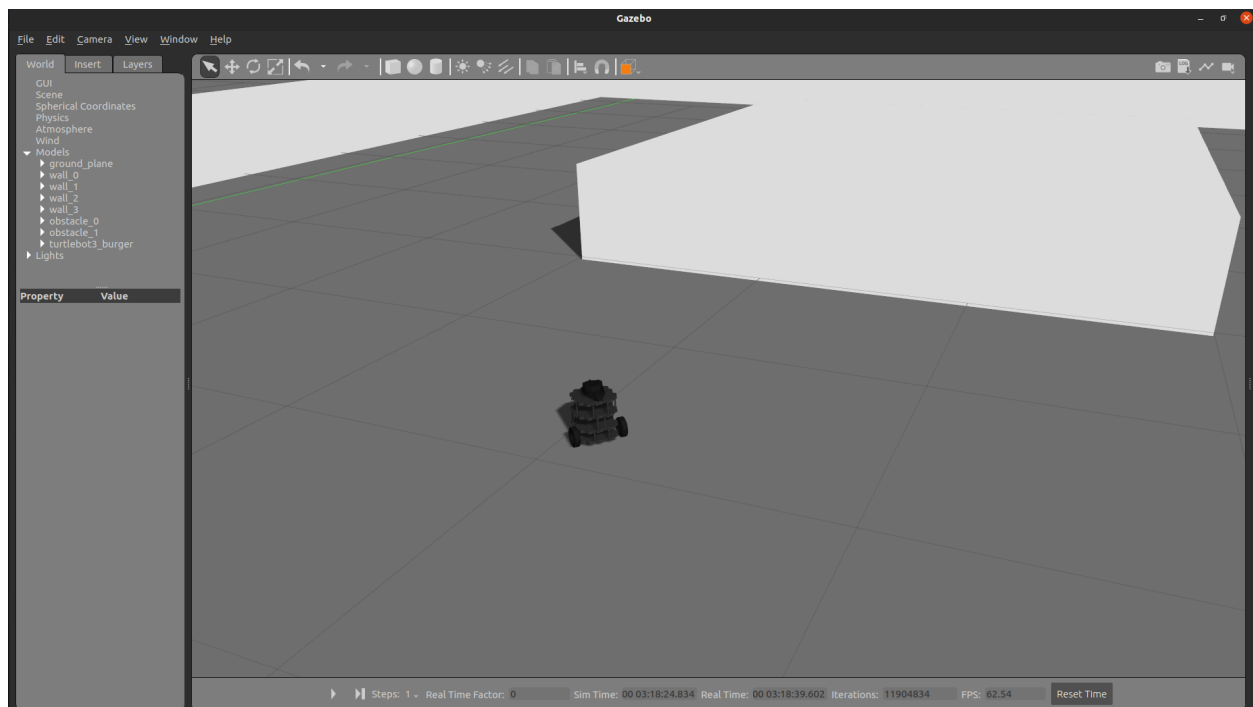
#### 3.2 Obstacles (in Gazebo):

The Gazebo simulation includes strategically placed physical obstacles to emulate real-world challenges for the UGV. These obstacles are configured to vary in shape, size, and placement, simulating different terrains and scenarios that the autonomous platform might encounter.

### 3.3 Gazebo World File and Node Coordinates File:

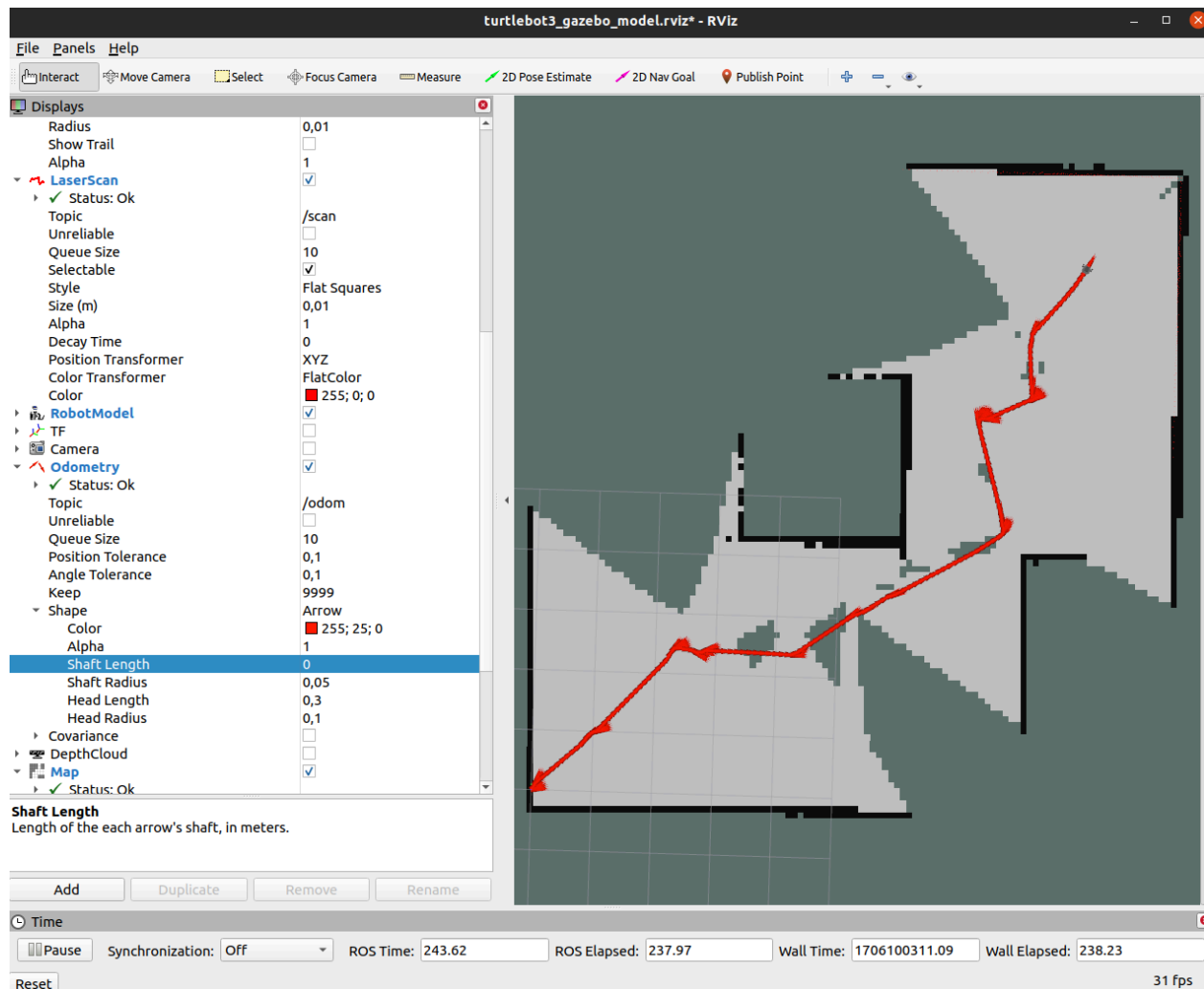
- **World File:** The Gazebo world file, named "rrt\_world.world," encapsulates the entire simulated environment. It defines the physical properties, obstacles, and other elements crucial for an accurate representation of the UGV's operational space.
- **Nodes Coordinates File:** To facilitate further analysis and evaluation, the coordinates of the RRT nodes are recorded in the "rrt\_nodes.txt" file. This file provides insight into the path-planning process and aids in debugging or simulating the RRT algorithm.

Once the simulation setup is complete, we utilize our custom-created map and generate a launch file. This launch file facilitates the seamless deployment of our map and robot at specific positions within the simulation environment. Subsequently, we employ a straightforward moving script to guide the robot along the planned route. This script serves as a navigational tool, enabling the robot to accurately follow the predefined path.



Now that we've set up our simulation, it's time to see our robot in action. We'll run a movement script using the text files we created. This step allows us to watch how our robot follows the map we designed, moving step by step until it reaches the goal. It's like a real-world test for our robot's ability to understand the route we planned and navigate through the environment. This part helps us check if everything works smoothly and lets us

make any needed improvements.



We noticed that our robot can follow the planned path, even though our movement script is basic and not very accurate.

## 4. Multi-Agent Systems

In the realm of autonomous systems, multi-agent systems (MAS) play a crucial role. MAS involves multiple intelligent agents, each capable of independent decision-making and interacting with the environment and other agents. These systems are designed to collaborate, coordinate, or compete to achieve specific goals. The coordination among agents enables them to perform tasks more efficiently and handle complex scenarios.

## 5. Conclusion

In conclusion, our project embarked on the exploration and development of a simple yet effective autonomous platform, focusing on an Unmanned Ground Vehicle (UGV). We meticulously detailed the construction of the UGV, emphasizing construction dimensions, equipment specifications, connectivity methods, movement parameters, and the underlying software architecture.

The implementation of the Rapidly Exploring Random Tree (RRT) algorithm added a layer of intelligence to our autonomous platform, enabling path planning in diverse scenarios with varying environmental parameters. The utilization of Gazebo, a powerful simulator, allowed us to simulate and observe the UGV's performance in a controlled environment, enhancing our understanding of its capabilities.

Furthermore, we delved into the realm of multi-agent systems, showcasing how collaborative entities can amplify the efficiency and adaptability of autonomous platforms. The project provided hands-on experience in scripting, simulation setup, and the integration of path-planning algorithms.

In essence, this endeavor provided a comprehensive understanding of autonomous systems, from the intricacies of algorithmic decision-making to the practical aspects of simulation and collaborative multi-agent systems. Through this project, we gained valuable insights into the exciting and evolving field of autonomous technology.

## 6. Scripts

### 6.1 RRT Planner and World Generator

```
import matplotlib.pyplot as plt
import numpy as np
import random

class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None

def is_collision_free(x, y, obstacles, safety_distance=0.5):
    for obstacle in obstacles:
```



```

        if (obstacle[0] - safety_distance) < x < (obstacle[1] +
safety_distance) and \
            (obstacle[2] - safety_distance) < y < (obstacle[3] +
safety_distance):
            return False # Collision detected
        return True # Collision-free

def generate_random_node(width, height):
    return Node(random.uniform(0, width), random.uniform(0, height))

def nearest_neighbor(random_node, nodes):
    return min(nodes, key=lambda node: np.hypot(node.x - random_node.x,
node.y - random_node.y))

def steer(from_node, to_node, step_size, obstacles):
    distance = np.hypot(to_node.x - from_node.x, to_node.y - from_node.y)

    if distance < step_size:
        return to_node
    else:
        theta = np.arctan2(to_node.y - from_node.y, to_node.x - from_node.x)
        new_x = from_node.x + step_size * np.cos(theta)
        new_y = from_node.y + step_size * np.sin(theta)

        # Check if the new node is collision-free and at a safe distance
        from obstacles
        if is_collision_free(new_x, new_y, obstacles):
            return Node(new_x, new_y)
        else:
            return from_node

def rrt(width, height, obstacles, start, goal, iterations=1000,
step_size=0.5, max_attempts=100):
    nodes = [start]

    for _ in range(iterations):
        random_node = generate_random_node(width, height)

        if not is_collision_free(random_node.x, random_node.y, obstacles):
            continue # Skip if collision

        nearest_node = nearest_neighbor(random_node, nodes)

```

```

    attempts = 0
    while attempts < max_attempts:
        new_node = steer(nearest_node, random_node, step_size,
obstacles)
        if new_node != nearest_node:
            break
        attempts += 1

    if attempts >= max_attempts:
        continue # Skip if unable to find a valid steering node

    if is_collision_free(new_node.x, new_node.y, obstacles):
        new_node.parent = nearest_node
        nodes.append(new_node)

# Check if goal is reached
    if np.hypot(new_node.x - goal.x, new_node.y - goal.y) < step_size:
        goal.parent = new_node
        nodes.append(goal)

# Extract the selected path
    selected_path = [goal]
    current = goal
    while current.parent:
        selected_path.append(current.parent)
        current = current.parent
    selected_path = selected_path[::-1] # Reverse the path to start
from the beginning

    return selected_path # Return only the correct path

return [] # Return an empty list if no valid path is found

def visualize_rrt(nodes, obstacles, start, goal):
    plt.figure(figsize=(6, 6))

    # Plot obstacles
    for obstacle in obstacles:
        plt.plot([obstacle[0], obstacle[1], obstacle[1], obstacle[0],
obstacle[0]],
                [obstacle[2], obstacle[2], obstacle[3], obstacle[3],
obstacle[2]], 'k-')

```

```

# Plot edges in the tree
for node in nodes:
    if node.parent:
        plt.plot([node.x, node.parent.x], [node.y, node.parent.y], 'b-')

# Plot start and goal
plt.plot(start.x, start.y, 'go', markersize=10)
plt.plot(goal.x, goal.y, 'ro', markersize=10)

# Highlight the path
path_x = []
path_y = []
current = goal
while current.parent:
    path_x.append(current.x)
    path_y.append(current.y)
    current = current.parent
path_x.append(start.x)
path_y.append(start.y)
plt.plot(path_x, path_y, 'g-', linewidth=2)

plt.title('RRT with Obstacles (Avoiding Close Proximity)')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)
plt.show()

def create_gazebo_world(width, height, obstacles,
world_filename='rrt_world.world', nodes_filename='rrt_nodes.txt'):

    with open(nodes_filename, 'w') as nodes_file:
        for node in nodes:
            nodes_file.write(f'{node.x} {node.y}\n')

    with open(world_filename, 'w') as world_file:
        world_file.write('<?xml version="1.0" ?>\n')
        world_file.write('<sdf version="1.6">\n')
        world_file.write('<world name="default">\n')

        # Include the ground plane
        world_file.write('\t<include>\n')
        world_file.write('\t\t<uri>model://ground_plane</uri>\n')

```

```

world_file.write('\t</include>\n')

# Add walls around the map
wall_thickness = 1.0 # You can adjust the thickness of the walls
walls = [
    [-1.5, 11.5, -1.5, wall_thickness - 1.5], # Bottom wall
    [-1.5, wall_thickness - 1.5, -1.5, 11.5], # Left wall
    [11.5 - wall_thickness, 11.5, -1.5, 11.5], # Right wall
    [-1.5, 11.5, 11.5 - wall_thickness, 11.5], # Top wall
]

for i, wall in enumerate(walls):
    world_file.write(f'\t<model name="wall_{i}">\n')
    world_file.write('\t\t<static>true</static>\n')
    world_file.write('\t\t<link name="link">\n')
    world_file.write('\t\t\t<collision name="collision">\n')
    world_file.write('\t\t\t\t<geometry>\n')
    world_file.write('\t\t\t\t\t<box>\n')
    world_file.write(f'\t\t\t\t\t\t<size>{wall[1] - wall[0]}
{wall[3] - wall[2]} 1.0</size>\n')
    world_file.write('\t\t\t\t\t</box>\n')
    world_file.write('\t\t\t\t</geometry>\n')
    world_file.write('\t\t\t</collision>\n')
    world_file.write('\t\t\t<visual name="visual">\n')
    world_file.write('\t\t\t\t<geometry>\n')
    world_file.write('\t\t\t\t\t<box>\n')
    world_file.write(f'\t\t\t\t\t\t<size>{wall[1] - wall[0]}
{wall[3] - wall[2]} 1.0</size>\n')
    world_file.write('\t\t\t\t\t</box>\n')
    world_file.write('\t\t\t\t</geometry>\n')
    world_file.write('\t\t\t</visual>\n')
    world_file.write('\t\t</link>\n')
    world_file.write(f'\t\t<pose>{:.2f} {:.2f} 0.0 0 0
0</pose>\n'.format(
        (wall[0] + wall[1]) / 2, (wall[2] + wall[3]) / 2))
    world_file.write('\t</model>\n')

# Add obstacles to the world
for i, obstacle in enumerate(obstacles):
    world_file.write(f'\t<model name="obstacle_{i}">\n')
    world_file.write('\t\t<static>true</static>\n')
    world_file.write('\t\t<link name="link">\n')

```

```

# Collision
world_file.write('\t\t\t<collision name="collision">\n')
world_file.write('\t\t\t\t<geometry>\n')
world_file.write('\t\t\t\t\t<box>\n')
world_file.write(f'\t\t\t\t\t\t\t<size>{obstacle[1] - obstacle[0]}
{obstacle[3] - obstacle[2]} 1.0</size>\n')
world_file.write('\t\t\t\t\t\t</box>\n')
world_file.write('\t\t\t\t</geometry>\n')
world_file.write('\t\t\t</collision>\n')

# Visual
world_file.write('\t\t\t<visual name="visual">\n')
world_file.write('\t\t\t\t<geometry>\n')
world_file.write('\t\t\t\t\t<box>\n')
world_file.write(f'\t\t\t\t\t\t\t<size>{obstacle[1] - obstacle[0]}
{obstacle[3] - obstacle[2]} 1.0</size>\n')
world_file.write('\t\t\t\t\t\t</box>\n')
world_file.write('\t\t\t\t\t</geometry>\n')
world_file.write('\t\t\t</visual>\n')

world_file.write('\t\t</link>\n')
world_file.write('\t\t<pose>{:.2f} {:.2f} 0.0 0 0
0</pose>\n'.format(
    (obstacle[0] + obstacle[1]) / 2, (obstacle[2] + obstacle[3])
/ 2))
world_file.write('\t</model>\n')

# Add sunlight
world_file.write('\t<light name="sun" type="directional">\n')
world_file.write('\t\t<cast_shadows>true</cast_shadows>\n')
world_file.write('\t\t<pose>0 0 10 0 0 0</pose>\n')
world_file.write('\t\t<diffuse>0.8 0.8 0.8 1</diffuse>\n')
world_file.write('\t\t<specular>0.2 0.2 0.2 1</specular>\n')
world_file.write('\t\t<attenuation>\n')
world_file.write('\t\t\t<range>1000</range>\n')
world_file.write('\t\t\t<constant>0.9</constant>\n')
world_file.write('\t\t\t<linear>0.01</linear>\n')
world_file.write('\t\t\t<quadratic>0.001</quadratic>\n')
world_file.write('\t\t</attenuation>\n')
world_file.write('\t\t<direction>-0.5 0.5 -0.5</direction>\n')
world_file.write('\t</light>\n')

```

```

        world_file.write('</world>\n')
        world_file.write('</sdf>\n')

    print(f'Gazebo world file "{world_filename}" created.')

if __name__ == "__main__":
    # Define workspace dimensions
    width, height = 10, 10

    # Define obstacles as [x_min, x_max, y_min, y_max]
    obstacles = [[3, 6, 4, 7], [8, 9, 1, 4]]

    # Define start and goal nodes
    start = Node(1, 1)
    goal = Node(9, 9)

    # Run RRT algorithm
    nodes = rrt(width, height, obstacles, start, goal, iterations=1000,
step_size=2, max_attempts=100)

    # Visualize the RRT with highlighted path
    visualize_rrt(nodes, obstacles, start, goal)

    # Create Gazebo world file
    create_gazebo_world(width, height, obstacles)

```

## 6.2 Robot Controller

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

```

import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from math import atan2, sqrt, pow
import sys

class TurtleBotController:
    def __init__(self):
        rospy.init_node('turtlebot_controller', anonymous=True)

```

```

        self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist,
queue_size=10)
        self.pose_subscriber = rospy.Subscriber('/odom', Odometry,
self.update_pose)

        self.pose = None

    def update_pose(self, data):
        self.pose = data.pose.pose

    def get_distance(self, goal_x, goal_y):
        current_x = self.pose.position.x
        current_y = self.pose.position.y

        distance = sqrt(pow((goal_x - current_x), 2) + pow((goal_y -
current_y), 2))
        return distance

    def move_turtlebot(self, goal_x, goal_y):
        rate = rospy.Rate(10)

        while not rospy.is_shutdown():
            # Wait until pose information is received
            while self.pose is None and not rospy.is_shutdown():
                rospy.loginfo("Waiting for odometry information...")
                rate.sleep()

            if rospy.is_shutdown():
                break

            distance = self.get_distance(goal_x, goal_y)

            if distance < 0.1:
                rospy.loginfo("Goal reached!")
                self.stop_movement()
                break

            goal_angle = atan2(goal_y - self.pose.position.y, goal_x -
self.pose.position.x)

            if abs(goal_angle - self.get_orientation()) > 0.1:

```

```

        self.rotate_turtlebot(goal_angle)
    else:
        self.stop_movement()
        self.linear_turtlebot()

    rate.sleep()

def stop_movement(self):
    twist = Twist()
    twist.angular.z = 0
    self.velocity_publisher.publish(twist)

def get_orientation(self):
    orientation_q = self.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y,
orientation_q.z, orientation_q.w]
    _, _, yaw = euler_from_quaternion(orientation_list)
    return yaw

def rotate_turtlebot(self, goal_angle):
    twist = Twist()
    twist.angular.z = 0.15 if goal_angle > self.get_orientation() else
-0.15
    self.velocity_publisher.publish(twist)


def linear_turtlebot(self):
    twist = Twist()
    twist.linear.x = 0.2
    self.velocity_publisher.publish(twist)

if __name__ == '__main__':
    try:
        controller = TurtleBotController()

        # Read goal coordinates from a file
        filename = sys.argv[1]

        with open(filename, 'r') as file:
            lines = file.readlines()
            goals = [(float(line.split()[0]), float(line.split()[1])) for
line in lines]
```





```
# Move to each goal sequentially
for goal_x, goal_y in goals:
    controller.move_turtlebot(goal_x, goal_y)

except rospy.ROSInterruptException:
    pass
```