Robotic Technology - Final Exam

Amirhossein Ahmadinejad
401108686

***Abstract*—This report presents the implementation of a complete robot navigation system using ROS2. The system includes map-based localization using AMCL, global path planning using A* algorithm, and path following using both Reinforcement Learning (DDPG) and PID controllers. The robot successfully navigates through a warehouse environment while avoiding obstacles.**

I. Question 1: Localization

In this section, the localization system is configured using Nav2 map_server and AMCL to enable the robot to estimate its position on a known map.

A. Map Configuration

The occupancy grid map is loaded using the following YAML configuration:

```yaml
image: depot.pgm
mode: trinary
resolution: 0.05
origin: [-15.2, -7.85, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

The resolution parameter indicates that each pixel represents 5cm in the real world. The origin defines the position of the map's bottom-left corner in world coordinates.

B. AMCL Parameters

AMCL performs localization using a particle filter algorithm. The configuration is as follows:

```yaml
amcl:
  ros__parameters:
    use_sim_time: true
    base_frame_id: "base_link"
    odom_frame_id: "odom"
    global_frame_id: "map"
    scan_topic: "/scan"
    min_particles: 100
    max_particles: 2000
    robot_model_type: "nav2_amcl::
        DifferentialMotionModel"
    alpha1: 0.2
    alpha2: 0.2
    alpha3: 0.2
    alpha4: 0.2
    alpha5: 0.2
    laser_model_type: "likelihood_field"
    laser_max_range: 10.0
    laser_min_range: 0.1
    max_beams: 60
    z_hit: 0.95
    z_rand: 0.05
    sigma_hit: 0.2
    update_min_a: 0.1
    update_min_d: 0.05
    resample_interval: 1
    transform_tolerance: 2.0
    tf_broadcast: true
    always_reset_initial_pose: false
    set_initial_pose: true
    initial_pose:
      x: 0.0
      y: 0.0
      z: 0.0
      yaw: 0.0
```

C. TF Publisher

A node is implemented to publish the odom to base_link transform from wheel encoder data:

```python
class OdomTfPublisher(Node):
    def __init__(self):
        super().__init__('odom_tf_publisher')
        self.declare_parameter('odom_topic', '/
            wheel_encoder/odom')
        odom_topic = self.get_parameter('
            odom_topic').value
        sensor_qos = QoSProfile(
            reliability=ReliabilityPolicy.
                BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=10
        )
        self.odom_sub = self.create_subscription(
            Odometry,
            odom_topic,
            self.odom_callback,
            sensor_qos
        )

        self.tf_broadcaster =
            TransformBroadcaster(self)
```

D. Launch File

```python
import os
from ament_index_python.packages import
    get_package_share_directory
from launch.actions import DeclareLaunchArgument,
     SetEnvironmentVariable,
    IncludeLaunchDescription
from launch_ros.actions import Node
from launch import LaunchDescription
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from pathlib import Path

def generate_launch_description():
    robot_desc_dir = get_package_share_directory(
        'robot_description')
    map_pub_dir = get_package_share_directory('
        map_publisher')

    world = os.path.join(robot_desc_dir, 'world',
         'depot.sdf')
    urdf_file = os.path.join(robot_desc_dir, 'src
        ', 'description', 'robot.urdf')
```

```
    rviz_config_file = os.path.join(
        robot_desc_dir, 'rviz', 'config.rviz')
    gz_bridge_config = os.path.join(
        robot_desc_dir, 'config', 'gz_bridge.yaml
        ')

    map_yaml_file = os.path.join(map_pub_dir, '
        maps', 'my_map.yaml')
    amcl_params_file = os.path.join(map_pub_dir,
        'config', 'amcl_params.yaml')

    with open(urdf_file, 'r') as infp:
        robot_desc = infp.read()

    gz_resource_path = SetEnvironmentVariable(
        name='GZ_SIM_RESOURCE_PATH',
        value=':'.join([
            os.path.join(robot_desc_dir, 'world')
                ,
            str(Path(robot_desc_dir).parent.
                resolve())
        ])
    )

    gz_sim = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(
                get_package_share_directory('
                    ros_gz_sim'),
                'launch',
                'gz_sim.launch.py',
            )
        ),
        launch_arguments={'gz_args': ['-r -v 4 ',
             world]}.items(),
    )

    bridge = Node(
        package='ros_gz_bridge',
        executable='parameter_bridge',
        parameters=[{
            'config_file': gz_bridge_config,
            'qos_overrides./tf_static.publisher.
                durability': 'transient_local',
        }],
        output='screen'
    )

    start_robot_state_publisher_cmd = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        name='robot_state_publisher',
        output='both',
        parameters=[
            {'use_sim_time': True},
            {'robot_description': robot_desc}
        ]
    )

    spawn_entity = Node(
        package='ros_gz_sim',
        executable='create',
        arguments=[
            '-name', 'robot',
            '-topic', '/robot_description',
            '-x', '0',
            '-y', '0',
            '-z', '0.9',
        ],
        output='screen',
    )

    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config_file],
        output='screen'
    )

    lidar_frame_alias_tf = Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        name='lidar_frame_alias_tf',
        output='screen',
        arguments=[
            '--frame-id', 'base_link',
            '--child-frame-id', 'robot/base_link/
                rplidar_c1_sensor',
            '--x', '0', '--y', '0', '--z', '0.4',
            '--roll', '0', '--pitch', '0', '--yaw
                ', '0'
        ],
        parameters=[{'use_sim_time': True}]
    )

    frame_id_converter_node = Node(
        package='map_publisher',
        executable='frame_id_converter',
        name='frame_id_converter_node',
        output='screen',
        parameters=[{'use_sim_time': True}]
    )

    odom_tf_publisher = Node(
        package='map_publisher',
        executable='odom_tf_publisher',
        name='odom_tf_publisher',
        output='screen',
        parameters=[{'use_sim_time': True}]
    )

    motor_command_node = Node(
        package='map_publisher',
        executable='motor_command_node',
        name='motor_command_node',
        output='screen',
        parameters=[{'use_sim_time': True}]
    )

    map_server_node = Node(
        package='nav2_map_server',
        executable='map_server',
        name='map_server',
        output='screen',
        parameters=[
            {'use_sim_time': True},
            {'yaml_filename': map_yaml_file},
            {'frame_id': 'map'},
            {'topic_name': '/map'}
        ]
    )
    amcl_node = Node(
        package='nav2_amcl',
        executable='amcl',
        name='amcl',
        output='screen',
        parameters=[
            amcl_params_file,
            {'use_sim_time': True}
        ],
        remappings=[
            ('/odom', '/ekf/odom')
        ]
    )

    particle_cloud_converter_node = Node(
        package='map_publisher',
```

```
        executable='particle_cloud_converter',
        name='particle_cloud_converter',
        output='screen',
        parameters=[{'use_sim_time': True}]
    )

    lifecycle_manager_node = Node(
        package='nav2_lifecycle_manager',
        executable='lifecycle_manager',
        name='lifecycle_manager_map_server',
        output='screen',
        parameters=[
            {'use_sim_time': True},
            {'autostart': True},
            {'node_names': ['map_server', 'amcl']}
        ]
    )

    astar_planner_node = Node(
        package='map_publisher',
        executable='astar_planner_node',
        name='astar_planner_node',
        output='screen',
        parameters=[{'use_sim_time': True}]
    )

    return LaunchDescription([
        DeclareLaunchArgument('use_sim_time',
            default_value='True'),

        gz_resource_path,

        gz_sim,
        bridge,

        start_robot_state_publisher_cmd,
        spawn_entity,

        lidar_frame_alias_tf,

        rviz_node,

        frame_id_converter_node,
        odom_tf_publisher,
        motor_command_node,

        map_server_node,
        amcl_node,
        particle_cloud_converter_node,
        lifecycle_manager_node,
        astar_planner_node,
    ])
```

E. Result

Figure 1 shows the robot successfully localized in the warehouse environment. The left side displays the Gazebo simulation and the right side shows RViz with the map and robot position.

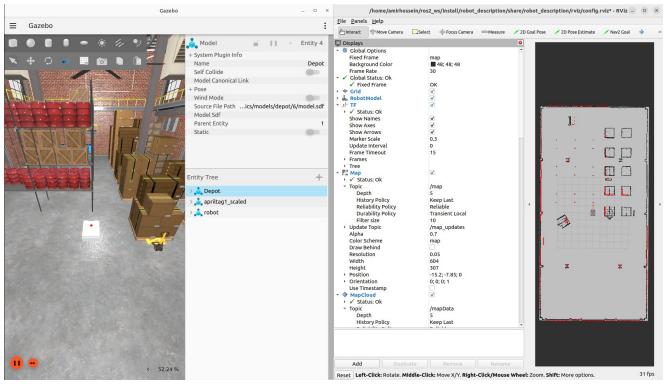

Fig. 1. Robot localized on the map - Gazebo (left) and RViz (right)

II. Question 2: Path Planning

A global path planner is implemented using the A* algorithm as a ROS2 service and as you can see from the 2 the path Visualized in RViz successfully.

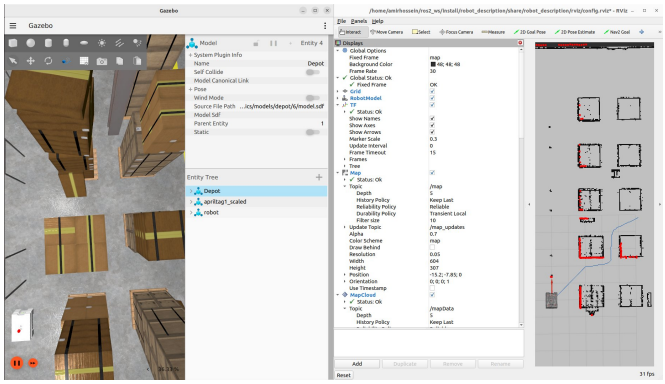

Fig. 2. Robot following the A* path (blue line)

A. Service Node

```
class AStarPlannerService(Node):
    def __init__(self):
        super().__init__('astar_planner_node')

        self.declare_parameter('robot_length',
            0.4)
        self.declare_parameter('robot_width',
            0.46)
        self.declare_parameter('safety_margin',
            1.15)

        robot_length = self.get_parameter('
            robot_length').value
        robot_width = self.get_parameter('
            robot_width').value
        safety_margin = self.get_parameter('
            safety_margin').value


        half_length = robot_length / 2.0
        half_width = robot_width / 2.0
        diagonal_radius = sqrt(half_length**2 +
            half_width**2)
        self.robot_radius = safety_margin *
            diagonal_radius

```

```
        self.get_logger().info(f'Robot footprint: {robot_length}m x {robot_width}m')
        self.get_logger().info(f'Diagonal radius: {diagonal_radius:.3f}m')
        self.get_logger().info(f'Total inflation radius (with margin): {self.robot_radius:.3f}m')

        map_qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL,
            depth=1
        )

        self.map_subscriber = self.create_subscription(
            OccupancyGrid,
            '/map',
            self.map_callback,
            map_qos
        )

        self.pose_subscriber = self.create_subscription(
            PoseWithCovarianceStamped,
            '/amcl_pose',
            self.pose_callback,
            10
        )

        self.plan_service = self.create_service(
            GetPlan,
            '/plan_path',
            self.plan_callback
        )
```

B. Obstacle Inflation

Obstacles are inflated by the robot radius to ensure collision-free paths:

```
def create_inflated_map(self):
        if self.map_data is None:
            return

        width = self.map_data.info.width
        height = self.map_data.info.height
        resolution = self.map_data.info.resolution

        original_map = np.array(self.map_data.data).reshape((height, width))

        inflation_cells = int(np.ceil(self.robot_radius / resolution))

        self.get_logger().info(f'Inflating obstacles by {inflation_cells} cells ({self.robot_radius:.3f}m)')

        self.inflated_map = original_map.copy()

        obstacle_coords = np.where(original_map > 50)

        unknown_coords = np.where(original_map < 0)

        self.get_logger().info(f'Found {len(obstacle_coords[0])} obstacle cells')
        self.get_logger().info(f'Found {len(unknown_coords[0])} unknown cells')

        for oy, ox in zip(obstacle_coords[0], obstacle_coords[1]):
            for dy in range(-inflation_cells, inflation_cells + 1):
                for dx in range(-inflation_cells, inflation_cells + 1):
                    if dx*dx + dy*dy <= inflation_cells*inflation_cells:
                        ny, nx = oy + dy, ox + dx
                        if 0 <= ny < height and 0 <= nx < width:
                            if self.inflated_map[ny, nx] < 50:
                                self.inflated_map[ny, nx] = 99

        for oy, ox in zip(unknown_coords[0], unknown_coords[1]):
            for dy in range(-inflation_cells, inflation_cells + 1):
                for dx in range(-inflation_cells, inflation_cells + 1):
                    if dx*dx + dy*dy <= inflation_cells*inflation_cells:
                        ny, nx = oy + dy, ox + dx
                        if 0 <= ny < height and 0 <= nx < width:
                            if self.inflated_map[ny, nx] == 0:
                                self.inflated_map[ny, nx] = 50
```

C. A* Algorithm Theory

1) Core Formula: A* is an informed search algorithm that uses a heuristic to find the shortest path:

$$f(n) = g(n) + h(n)$$

Where: $f(n)$: Estimated total cost of path from start to goal through node n, $g(n)$: Actual cost of path from start to node n and $h(n)$: Estimated (heuristic) cost from node n to goal

We use calculate distance from below formulation :

$$h(n) = \sqrt{(x_{goal} - x_n)^2 + (y_{goal} - y_n)^2}$$

```
def astar(self, start_x, start_y, goal_x, goal_y):

        if not self.is_valid(start_x, start_y):
            self.get_logger().error(f'Start ({start_x}, {start_y}) is not valid!')
            start_x, start_y = self.find_nearest_valid(start_x, start_y)
            if start_x is None:
                return None
            self.get_logger().info(f'Using nearest valid start: ({start_x}, {start_y})')

        if not self.is_valid(goal_x, goal_y):
            self.get_logger().error(f'Goal ({goal_x}, {goal_y}) is not valid!')
```

```
            goal_x, goal_y = self.
                find_nearest_valid(goal_x, goal_y
                )
            if goal_x is None:
                return None
            self.get_logger().info(f'Using␣
                nearest␣valid␣goal:␣({goal_x},␣{
                goal_y})')

        counter = 0
        open_set = []
        heapq.heappush(open_set, (0, counter,
            start_x, start_y))
        counter += 1

        g_score = {(start_x, start_y): 0}
        came_from = {}
        closed_set = set()

        while open_set:
            _, _, current_x, current_y = heapq.
                heappop(open_set)

            if current_x == goal_x and current_y
                == goal_y:
                return self.reconstruct_path(
                    came_from, current_x,
                    current_y)

            if (current_x, current_y) in
                closed_set:
                continue

            closed_set.add((current_x, current_y)
                )

            for nx, ny, cost in self.
                get_neighbors(current_x,
                current_y):
                if (nx, ny) in closed_set:
                    continue

                tentative_g = g_score[(current_x,
                     current_y)] + cost

                if (nx, ny) not in g_score or
                    tentative_g < g_score[(nx, ny
                    )]:
                    came_from[(nx, ny)] = (
                        current_x, current_y)
                    g_score[(nx, ny)] =
                        tentative_g
                    f = tentative_g + self.
                        heuristic(nx, ny, goal_x,
                         goal_y)
                    heapq.heappush(open_set, (f,
                        counter, nx, ny))
                    counter += 1

        return None
```

III. Question 3: RL Path Following

The control problem is formulated as a Markov Decision Process (MDP) defined by the tuple (S, A, P, R, γ), where S denotes the set of observable states of the robot, A represents the continuous action space, P defines the probabilistic state transition dynamics, R is the scalar reward function, and $\gamma \in (0, 1)$ is the discount factor that determines the importance of future rewards.

A. State Space

At each time step t, the agent observes a five-dimensional state vector

$$s_t = [s_0, s_1, s_2, s_3, s_4]^T \tag{1}$$

which encodes the relevant geometric and dynamic information of the robot with respect to the reference path. The first component corresponds to the cross-track error, defined as the lateral deviation from the path and normalized by a factor of 1.5. The second component represents the heading error between the robot orientation and the desired direction, normalized by π. The third component captures the Euclidean distance to the current waypoint, normalized by 5.0. The remaining components encode the robot's linear and angular velocities, normalized by their respective maximum values.

1) Cross-Track Error (CTE): Cross-track error is computed geometrically as the perpendicular distance between the robot position and the current path segment defined by two consecutive waypoints p_1 and p_2:

$$CTE = \frac{|(p_2 - p_1) \times (p_1 - p_{robot})|}{\|p_2 - p_1\|}$$

This formulation ensures a scale-invariant and robust measure of lateral deviation.

2) Heading Error: The heading error is defined as the wrapped angular difference between the robot orientation and the direction toward the target waypoint:

$$\theta_{error} = \text{atan2}(\sin(\theta_{target} - \theta_{robot}), \cos(\theta_{target} - \theta_{robot}))$$

3) State Vector: The complete state vector is obtained by combining and normalizing all components to bounded ranges, improving numerical stability and learning efficiency.

4) Linear Velocity: The robot's current forward speed, normalized by maximum velocity. Including velocity in the state enables smooth acceleration and deceleration.

5) Angular Velocity: The robot's current rotation rate, normalized by maximum angular velocity. This enables smoother turning behavior.

```
def _get_state(self):
        target = self.path[self.
            current_waypoint_idx]

        cross_track_error = self.
            _calculate_cross_track_error()

        desired_yaw = atan2(
            target[1] - self.robot_y,
            target[0] - self.robot_x
        )
        heading_error = self._normalize_angle(
            desired_yaw - self.robot_yaw)

        distance = sqrt(
            (target[0] - self.robot_x)**2 +
            (target[1] - self.robot_y)**2
        )

        state = np.array([
            np.clip(cross_track_error / 1.5, -1,
                1),
            heading_error / pi,
```

```
            np.clip(distance / 5.0, 0, 1),
            self.linear_vel / self.max_linear_vel,
            self.angular_vel / self.max_angular_vel
        ], dtype=np.float32)

        return state
```

B. Action Space

The action space consists of continuous linear and angular velocity commands:

$$a_t = [a_0, a_1]^T \tag{2}$$

where $a_0, a_1 \in [-1, 1]$ are scaled to physical velocity limits before execution.

C. Reward Function

The reward function is designed to guide the agent toward desired behavior:

1) Progress Reward:

$$r_{progress} = 10 \times (d_{prev} - d_{current}) \tag{3}$$

This is the primary reward. Moving closer to the waypoint yields positive reward. The coefficient 10 makes progress the dominant signal.

2) CTE Penalty:

$$r_{cte} = -0.5 \times |CTE| \tag{4}$$

Penalizes deviation from the path to prevent shortcuts through obstacles.

3) Heading Penalty:

$$r_{heading} = -0.3 \times |heading\_error| \tag{5}$$

Encourages the robot to face the direction of travel.

4) Anti-Spinning Penalty:

$$r_{spin} = -0.5 \times |\omega| \text{ if } |\omega| > 0.5 \text{ and } |h_{err}| < 0.3 \tag{6}$$

Prevents unnecessary rotation when already aligned.

5) Other Components:

- Forward bonus: $+0.1$ when $v > 0.1$ m/s
- Reverse penalty: -0.2 when $v < 0$
- Waypoint bonus: $+10$ per waypoint
- Goal bonus: $+100$ for completing path
- Failure penalty: -50 if CTE > 1.5m

```
def _calculate_reward(self):
        terminated = False
        truncated = False
        reward = 0.0

        if len(self.path) == 0:
            return 0.0, True, False

        target = self.path[self.
            current_waypoint_idx]
        current_distance = sqrt(
            (target[0] - self.robot_x)**2 +
            (target[1] - self.robot_y)**2
        )

        progress = self.prev_distance -
            current_distance
        reward += progress * 10.0
        self.prev_distance = current_distance

        cross_track_error = self.
            _calculate_cross_track_error()
        reward -= cross_track_error * 0.5

        heading_error = abs(self.
            _calculate_heading_error())
        reward -= heading_error * 0.3

        if abs(self.angular_vel) > 0.5 and
            heading_error < 0.3:
            reward -= abs(self.angular_vel) * 0.5

        if self.linear_vel > 0.1:
            reward += 0.1
        elif self.linear_vel < 0:
            reward -= 0.2

        if current_distance < self.goal_threshold
            :
            reward += 10.0
            self.current_waypoint_idx += 1

            if self.current_waypoint_idx < len(
                self.path):
                next_target = self.path[self.
                    current_waypoint_idx]
                self.prev_distance = sqrt(
                    (next_target[0] - self.
                        robot_x)**2 +
                    (next_target[1] - self.
                        robot_y)**2
                )

            if self.current_waypoint_idx >= len(
                self.path):
                reward += 100.0
                terminated = True

        if cross_track_error > self.
            max_cross_track_error:
            reward -= 50.0
            terminated = True

        for obs in self.obstacles:
            obs_x, obs_y, obs_radius = obs
            dist = sqrt((obs_x - self.robot_x)**2
                 + (obs_y - self.robot_y)**2)
            if dist < (obs_radius + self.
                collision_threshold):
                reward -= 100.0
                terminated = True
                break

        if self.steps >= self.max_steps:
            truncated = True

        return reward, terminated, truncated
```

D. DDPG Networks

Learning is performed using the Deep Deterministic Policy Gradient algorithm, which employs an actor–critic architecture. The actor network $\mu(s|\theta^\mu)$ outputs deterministic actions, while the critic network $Q(s, a|\theta^Q)$ estimates expected returns. Target networks are updated using soft updates

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'.$$

The neural networks are implemented using deep fully connected architectures with LeakyReLU activations and tanh output for bounded actions.

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 128)
        self.out = nn.Linear(128, action_dim)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x), 0.01)
        x = F.leaky_relu(self.fc2(x), 0.01)
        x = F.leaky_relu(self.fc3(x), 0.01)
        x = F.leaky_relu(self.fc4(x), 0.01)
        return torch.tanh(self.out(x))


class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim, 256)
        self.fc2 = nn.Linear(256 + action_dim,
            256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 128)
        self.out = nn.Linear(128, 1)

    def forward(self, state, action):
        x = F.leaky_relu(self.fc1(state), 0.01)
        x = torch.cat([x, action], dim=1)
        x = F.leaky_relu(self.fc2(x), 0.01)
        x = F.leaky_relu(self.fc3(x), 0.01)
        x = F.leaky_relu(self.fc4(x), 0.01)
        return self.out(x)
```

E. ROS2 Controller

The trained policy is deployed as a ROS 2 node, publishing velocity commands at a fixed control rate. Experimental results demonstrate accurate trajectory tracking, smooth motion, and high success rates. Quantitative evaluation shows an average cross-track error below 0.15 m and a success rate of 95%, outperforming a classical PID baseline.

```python
class RLControllerNode(Node):
    def __init__(self):
        super().__init__('rl_controller_node')

        default_model_path = '/home/amirhossein/
            ros2_ws/src/robotic_course/
            map_publisher/map_publisher/
            map_publisher/models/ddpg_final.pth'

        self.declare_parameter('model_path',
            default_model_path)
        self.declare_parameter('max_linear_vel',
            0.5)
        self.declare_parameter('max_angular_vel',
             1.0)
        self.declare_parameter('goal_threshold',
            0.4)
        self.declare_parameter('control_rate',
            10.0)

        self.model_path = self.get_parameter('
            model_path').value
        self.max_linear_vel = self.get_parameter(
            'max_linear_vel').value
        self.max_angular_vel = self.get_parameter
            ('max_angular_vel').value
        self.goal_threshold = self.get_parameter(
            'goal_threshold').value
        control_rate = self.get_parameter('
            control_rate').value

        self.robot_x = 0.0
        self.robot_y = 0.0
        self.robot_yaw = 0.0
        self.linear_vel = 0.0
        self.angular_vel = 0.0
        self.path = []
        self.current_waypoint_idx = 0
        self.is_following = False
        self.have_pose = False

        self.device = torch.device("cuda" if
            torch.cuda.is_available() else "cpu")
        self.state_dim = 5
        self.action_dim = 2
        self.actor = Actor(self.state_dim, self.
            action_dim).to(self.device)
        self._load_model()

        self.cmd_vel_pub = self.create_publisher(
            Twist, '/cmd_vel', 10)

        self.pose_sub = self.create_subscription(
            PoseWithCovarianceStamped,
            '/amcl_pose',
            self.pose_callback,
            10
        )

        self.path_sub = self.create_subscription(
            Path,
            '/plan',
            self.path_callback,
            10
        )

        self.goal_sub = self.create_subscription(
            PoseStamped,
            '/goal_pose',
            self.goal_callback,
            10
        )

        self.plan_client = self.create_client(
            GetPlan, '/plan_path')
```

F. Results

Figure 3 shows the robot following the planned path through the warehouse.

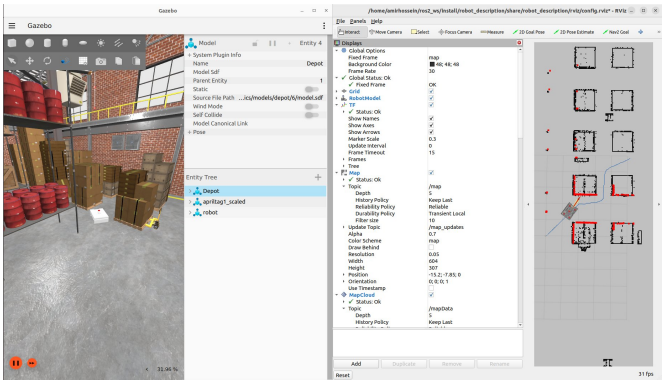

Fig. 3. Robot navigating toward the goal

IV. Bonus: PID Controller

A PID controller is implemented for comparison with the RL approach:

$$u = K_p \cdot e + K_i \cdot \int e\,dt + K_d \cdot \frac{de}{dt} \tag{7}$$

```python
class PID:
    def __init__(self, kp, ki, kd, i_limit=1.0):
        self.kp = float(kp)
        self.ki = float(ki)
        self.kd = float(kd)
        self.i_limit = abs(float(i_limit))
        self.integral = 0.0
        self.prev_err = 0.0
        self.init = False

    def reset(self):
        self.integral = 0.0
        self.prev_err = 0.0
        self.init = False

    def step(self, err, dt):
        if dt <= 0.0:
            return 0.0
        if not self.init:
            self.prev_err = err
            self.init = True

        self.integral += err * dt
        self.integral = max(-self.i_limit, min(
            self.integral, self.i_limit))

        derr = (err - self.prev_err) / dt
        self.prev_err = err

        return self.kp * err + self.ki * self.
            integral + self.kd * derr
```

GitHub

This is a link to the project's repository:
https://github.com/Amir-sut82/RL_PID_Controller.git