Project 4b Report Group 38 : Robotics and Spatial Intelligence

Part 1(A) Description of the Overall Approach:

System Overview

In expanding the differential drive robot simulator from Project 4a, the focus was on integrating an occupancy grid map and a LiDAR sensor to enhance the simulation's fidelity and utility. The occupancy grid provides a discrete representation of the environment, marking free, occupied, and unknown spaces, which allows the robot to make informed navigation decisions. The addition of a simulated LiDAR sensor equips the robot with the ability to perceive its surroundings, detect obstacles, and adjust its trajectory in real-time. Together, these enhancements enable the simulation of more complex scenarios and behaviors, such as obstacle avoidance and path planning. .

Node 1: Velocity Translator Node

This remains the same from the previous project (4a)

Node 2: Simulator Node

The DifferentialDriveSimulator node simulates a differential drive robot in a two-dimensional environment, as defined by the provided robot_name and world_file_name parameters. The robot is represented by a URDF (Unified Robot Description Format) file, which outlines its physical characteristics, such as body radius, wheel distance, and laser scanner specifications. Additionally, the robot's behavior is influenced by parameters such as error variance for wheel velocities, update rates for error calculation, and laser scanner properties including range, angle, and failure probabilities.

Upon initialization, the simulator reads in a world file that specifies the layout of the environment, including obstacles and the initial pose of the robot. This file is used to generate an OccupancyGrid, which is a grid-based representation of the environment where each cell can be free, occupied, or unknown. The grid is then published to the /map topic, allowing visualization in RViz or other ROS tools.

```python
def read_world_file(self,filename):
    with open(filename) as file:
        world = yaml.safe_load(file)

    resolution = world['resolution']
    # print(f'the {resolution = }')
    # robot_pose = world['robot_pose']
    map_data = world['map']
    # Split the map string into rows
    map_rows = map_data.split('\n')
    # print(f'the {map_data = }')
    # Strip out newline characters and create a 2D list of map cells

    map_array = [list(row) for row in map_rows]

    # print(f'the {len(map_array) = }')
    origin_x = 0.0
    origin_y = 0.0
    #creating the occupancy grid message

    grid = OccupancyGrid()
    grid.header.stamp = self.get_clock().now().to_msg()
    grid.header.frame_id = 'world'
    grid.info.resolution = resolution
    grid.info.width = len(map_array[0])
    grid.info.height = len(map_array)
    grid.info.origin.position.x = origin_x
    grid.info.origin.position.y = origin_y
    grid.info.origin.orientation.w = 1.0 # No rotation

    # Flattening the map data and converting it to occupancy values

    flat_map = []
    for row in reversed(map_array): # reversed to match the occupancy grid coordinate system
        # print(f'the {row = }')
        for char in row:
            if char == '#':
                flat_map.append(100)
            elif char =='.':
                flat_map.append(0)
            elif char == '\n':
                continue

    grid.data = flat_map
    # print(f'the {flat_map = }')
    # print(f'the {grid.data = }')
    return grid
```

The node subscribes to /vl and /vr topics to receive velocity commands for the left and right wheels, respectively. These velocities are subject to simulated random errors based on the predefined variances. The simulator uses these velocities to compute the robot's new pose at regular intervals, considering the robot's current orientation and the specified wheel velocities.

```
    # Subscribers
    self.subscription_vl = self.create_subscription(Float64, '/vl', self.vl_callback, 10)
    self.subscription_vr = self.create_subscription(Float64, '/vr', self.vr_callback, 10)
```

Collision detection is an integral part of the simulation. Before updating the robot's pose, a collision check is performed by creating a set of boundary points around the robot's circumference and verifying if any of these points would lie within an obstacle in the next timestep. If a collision is detected, the robot's movement is halted.

```
def check_collision(self, next_x, next_y):
    map_resolution = self.map_msg.info.resolution
    map_origin = self.map_msg.info.origin.position

    # Calculate the bounds of the robot considering its radius
    bounds = [
        (next_x + math.cos(angle) * self.radius, next_y + math.sin(angle) * self.radius)
        for angle in np.linspace(0, 2 * math.pi, num=60)  # Check around the circumference
    ]

    for bx, by in bounds:
        # Convert each bound point to map coordinates
        map_x = int((bx - map_origin.x) / map_resolution)
        map_y = int((by - map_origin.y) / map_resolution)

        # Check if the cell is within map bounds
        if (map_x >= self.map_msg.info.width or map_x < 0 or
            map_y >= self.map_msg.info.height or map_y < 0):
            return True  # Collision with boundary

        # Check if the cell is occupied
        index = map_y * self.map_msg.info.width + map_x
        if index < len(self.map_msg.data) and self.map_msg.data[index] == 100:
            return True  # Collision with obstacle

    return False  # No collision
```

The laser scanner simulation is achieved through a ray-casting method. For each laser beam, an angle is calculated relative to the robot's orientation, and a ray is cast into the environment. If the ray encounters an obstacle within the laser's range, the distance is recorded; otherwise, the maximum range is returned. To add realism, each measurement can fail with a certain probability, resulting in a NaN value, and Gaussian noise is added to the distance measurements.

```python
def ray_cast(self, angle):

    if random.random() < self.laser_fail_probability:
        return float('nan')  # Return NaN for a failed measurement
# Start at the robot's location
    x = self.x
    y = self.y

    # Calculate the ray's direction in the world frame
    world_angle = self.theta + angle
    dx = math.cos(world_angle)
    dy = math.sin(world_angle)

    # Step size for the ray increments, depending on the map resolution
    step_size = self.map_msg.info.resolution / 2.0
    max_distance = self.range_max
    distance = 0

    # Incrementally step along the ray
    while distance < max_distance:
        # Check the occupancy grid cell at the current ray position
        map_x = int((x - self.map_msg.info.origin.position.x) / self.map_msg.info.resolution)
        map_y = int((y - self.map_msg.info.origin.position.y) / self.map_msg.info.resolution)

        # Check if out of bounds
        if map_x < 0 or map_y < 0 or map_x >= self.map_msg.info.width or map_y >= self.map_msg.info.height:
            return max_distance  # Return max range if out of bounds

        # Calculate the index in the occupancy grid data array
        index = map_y * self.map_msg.info.width + map_x

        # Check if the occupancy grid cell is occupied
        if self.map_msg.data[index] == 100:
            # Add some noise to the measurement
            noise = np.random.normal(0, math.sqrt(self.laser_error_variance))
            measured_distance = distance + noise
            return min(max(measured_distance, self.range_min), max_distance)

        # Move to the next position along the ray
        x += dx * step_size
        y += dy * step_size
        distance += step_size

    # If no obstacle was hit, return the max range
    return max_distance
```

Lastly, the laser scan data is published to the /scan topic at a frequency determined by the laser's specified rate, enabling the visualization of the environment from the robot's perspective in RViz.

```python
def publish_laser_scan(self):
    scan = LaserScan()
    scan.header.stamp = self.get_clock().now().to_msg()
    scan.header.frame_id = 'laser'
    scan.angle_min = self.angle_min
    scan.angle_max = self.angle_max
    scan.angle_increment = (scan.angle_max - scan.angle_min) / self.laser_count
    scan.time_increment = 0.0
    scan.range_min = self.range_min
    scan.range_max = self.range_max

    scan.ranges = [self.ray_cast(angle) for angle in np.linspace(scan.angle_min, scan.angle_max, self.laser_count)]

    self.laser_scan_publisher.publish(scan)
```

Project 4b Report Group 38 : Robotics and Spatial Intelligence

Launch File and System Integration

To ensure synchronized operation, a launch file is used to initialize the system. It sets up the necessary arguments for input and output bag files, starts the playback of recorded sensor data, initiates all the required nodes, and captures the system's output to the specified output bag file.

The choices made in designing the program:

When designing the enhanced differential drive robot simulator, several deliberate choices were made to ensure a robust and realistic simulation environment:

1. **Occupancy Grid Map Integration:** The decision to incorporate an occupancy grid map was driven by the need to simulate a more realistic navigation scenario. The grid represents the spatial layout of the environment and is fundamental for path planning and collision detection. It allows the robot to make decisions based on the structure of the simulated world, enhancing the realism of the simulation.

2. **LiDAR Sensor Simulation:** Simulating a LiDAR sensor was a pivotal choice to provide the robot with environmental awareness. Ray-casting algorithms were employed to mimic the sensor's behavior, allowing the robot to detect obstacles and measure distances to them. This simulation includes the addition of Gaussian noise and a failure rate to the sensor readings to closely emulate real-world uncertainties and sensor imperfections.

3. **Error Simulation:** Incorporating error simulation in wheel velocities and sensor measurements was a deliberate choice to prepare for the imperfect nature of real-world robotics. By considering these factors, the simulator provides a more challenging and instructive platform for developing and testing control algorithms.

4. **Real-Time Updates and Feedback:** The system was designed to operate in real-time, continuously responding to control inputs, updating the robot's state, and publishing sensor data. This dynamic nature is crucial for testing the responsiveness of navigation algorithms under simulated real-world conditions.

5. **Modularity and Parameterization:** The program was structured with modularity in mind, allowing individual components, such as the robot's physical parameters or the environmental map, to be easily modified or replaced. This design choice facilitates

experimentation with different robot configurations or maps without extensive code changes.

Part 1(B): Did the results meet our expectations?

In evaluating the performance of the differential drive simulator node, the results met our expectations. The integration of the occupancy grid map and the LiDAR sensor simulation functioned as intended, providing a comprehensive representation of the robot's environment. The system successfully interpreted the YAML world file to construct a simulated map, accurately reflecting the placement of obstacles and free spaces within the grid.

The simulated LiDAR sensor, powered by a ray-casting algorithm, effectively identified obstacles by publishing detailed scan data to the /scan topic. This allowed for a realistic depiction of how a physical robot's sensor would operate, taking into account various environmental factors and sensor noise.

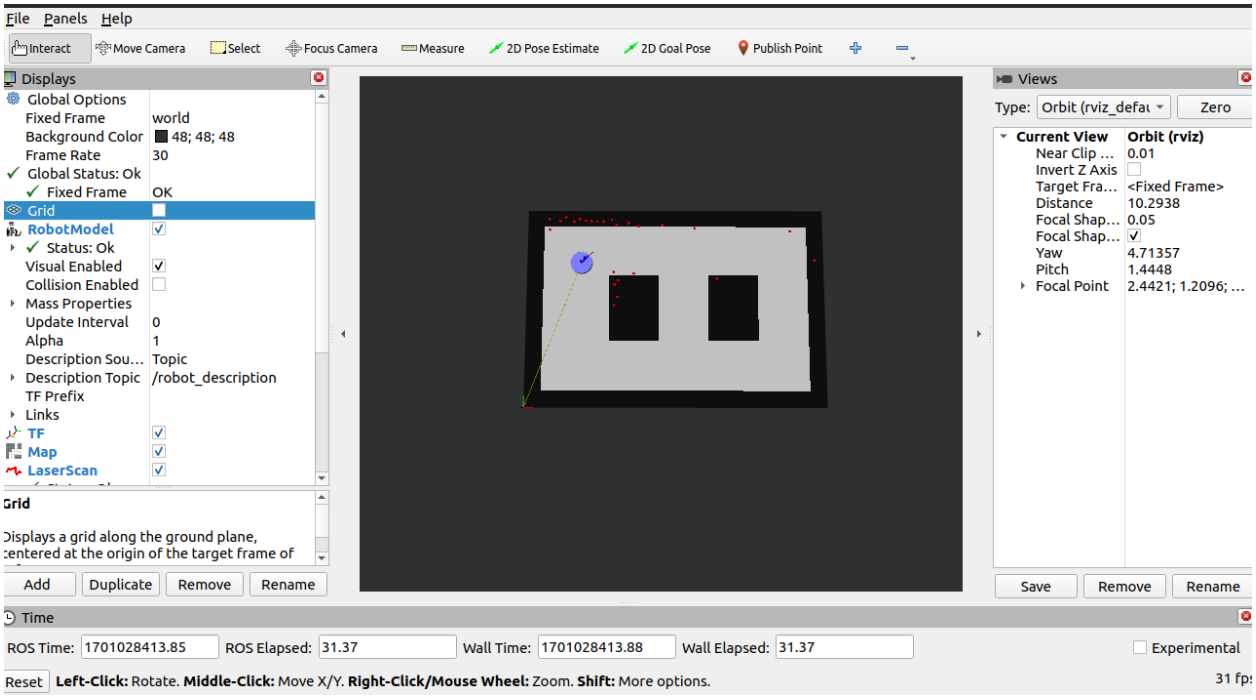
Collision detection was another aspect that performed as expected. The simulator preemptively checked the robot's proposed movements against the map data, preventing it from moving into occupied grid cells.

Overall, the simulator node achieved its goal of creating a dynamic and interactive environment for the differential drive robot. It allowed for the testing of control strategies, sensor integration, and autonomous behavior in a setting that closely mimics the challenges faced in actual robotic applications. The successful implementation of these features underscores the effectiveness of the simulation in providing a reliable platform for robotics research and development.

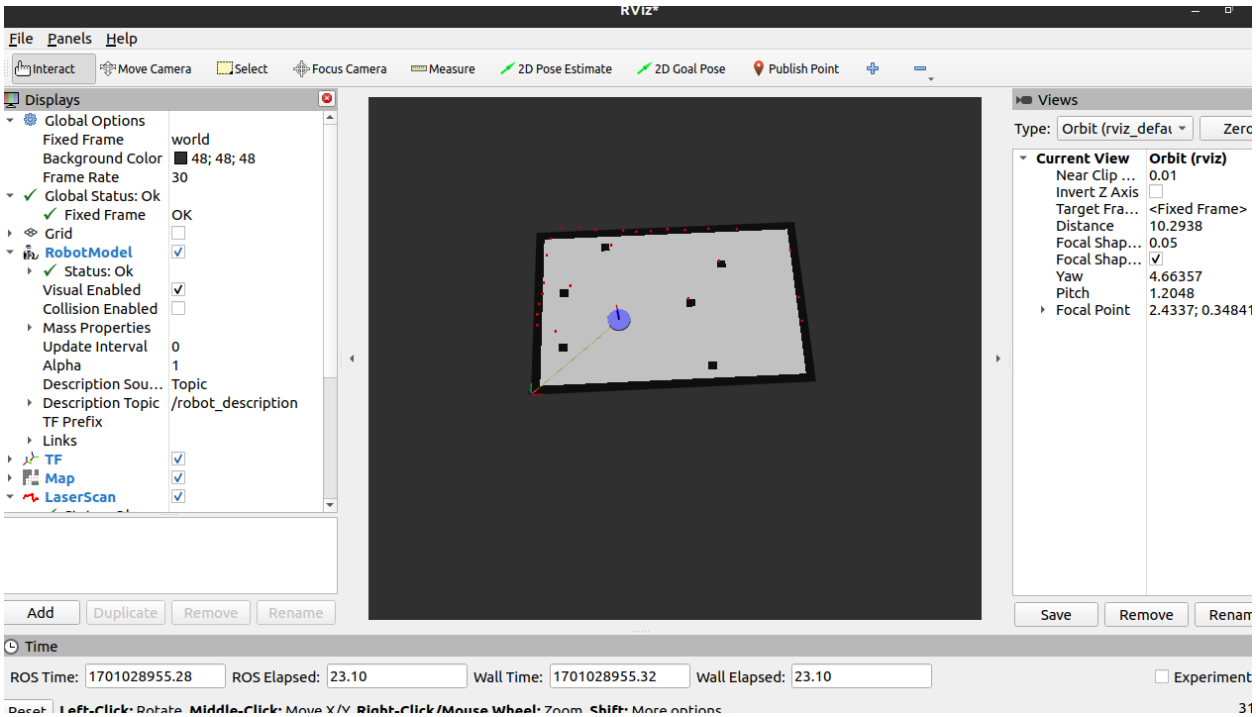
Project 4b Report Group 38 : Robotics and Spatial Intelligence

Screenshots showing the final configuration in rviz:

Test case 1 robot name: normal.robot world name: brick.world bag input: input5

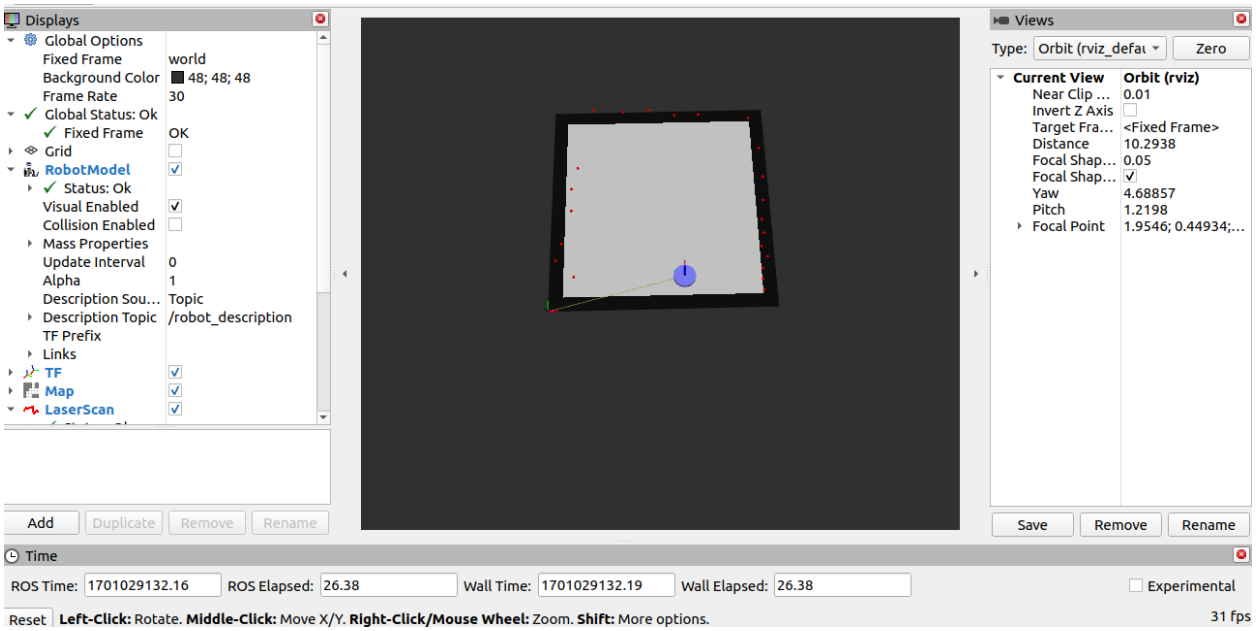


Test case 2 robot name: ideal.robot world name: pillars.world bag input: input6

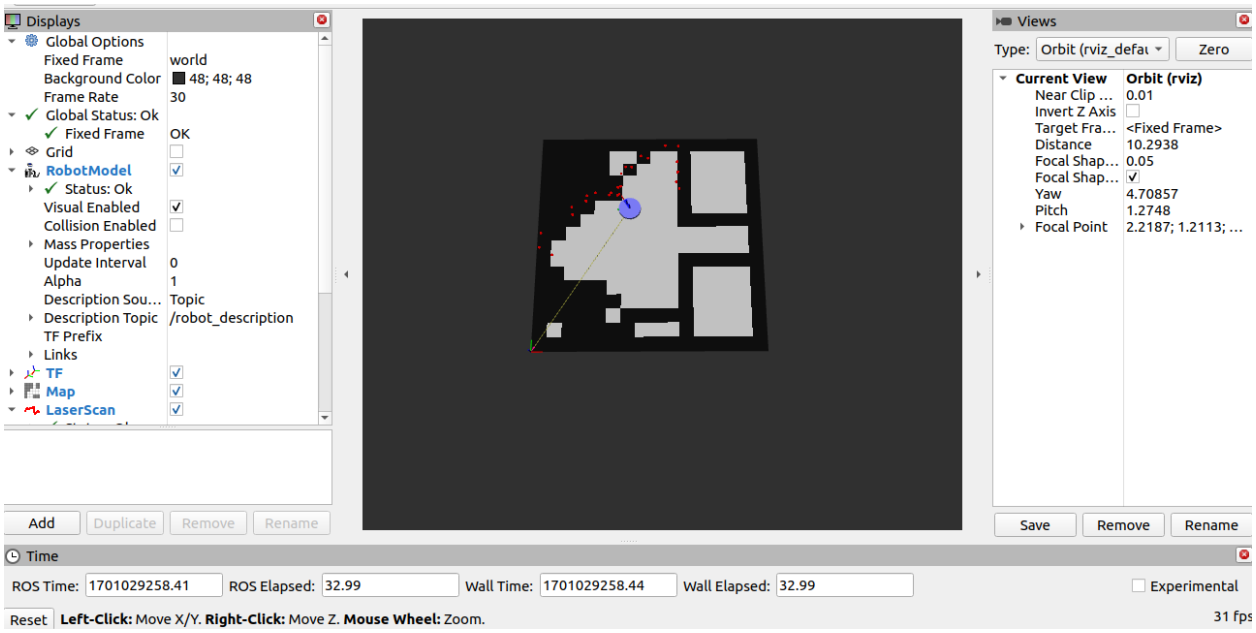


Project 4b Report Group 38 : Robotics and Spatial Intelligence

Test case 3 robot name: normal.robot world name: open.world bag input: input7

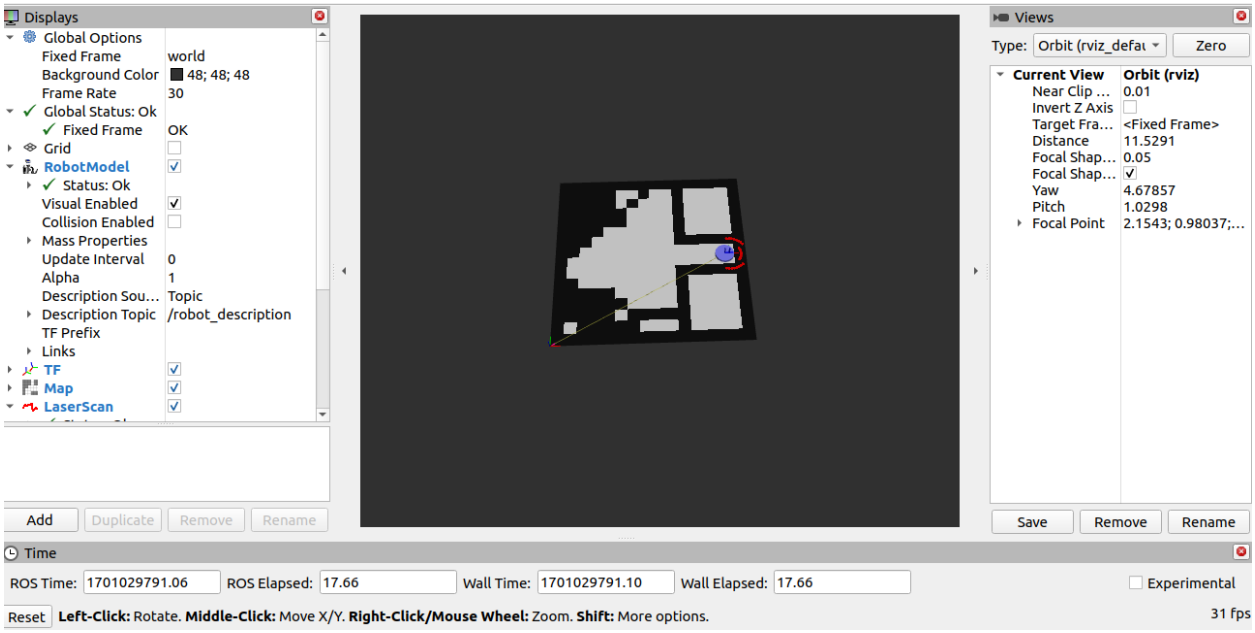


Test case 4 robot name: ideal.robot world name: ell.world bag input: input8



Project 4b Report Group 38 : Robotics and Spatial Intelligence

Test case 5 robot name: ideal.robot world name: ell.world bag input: input9



Test case 6 robot name: bad.robot world name: ell.world bag input: input9