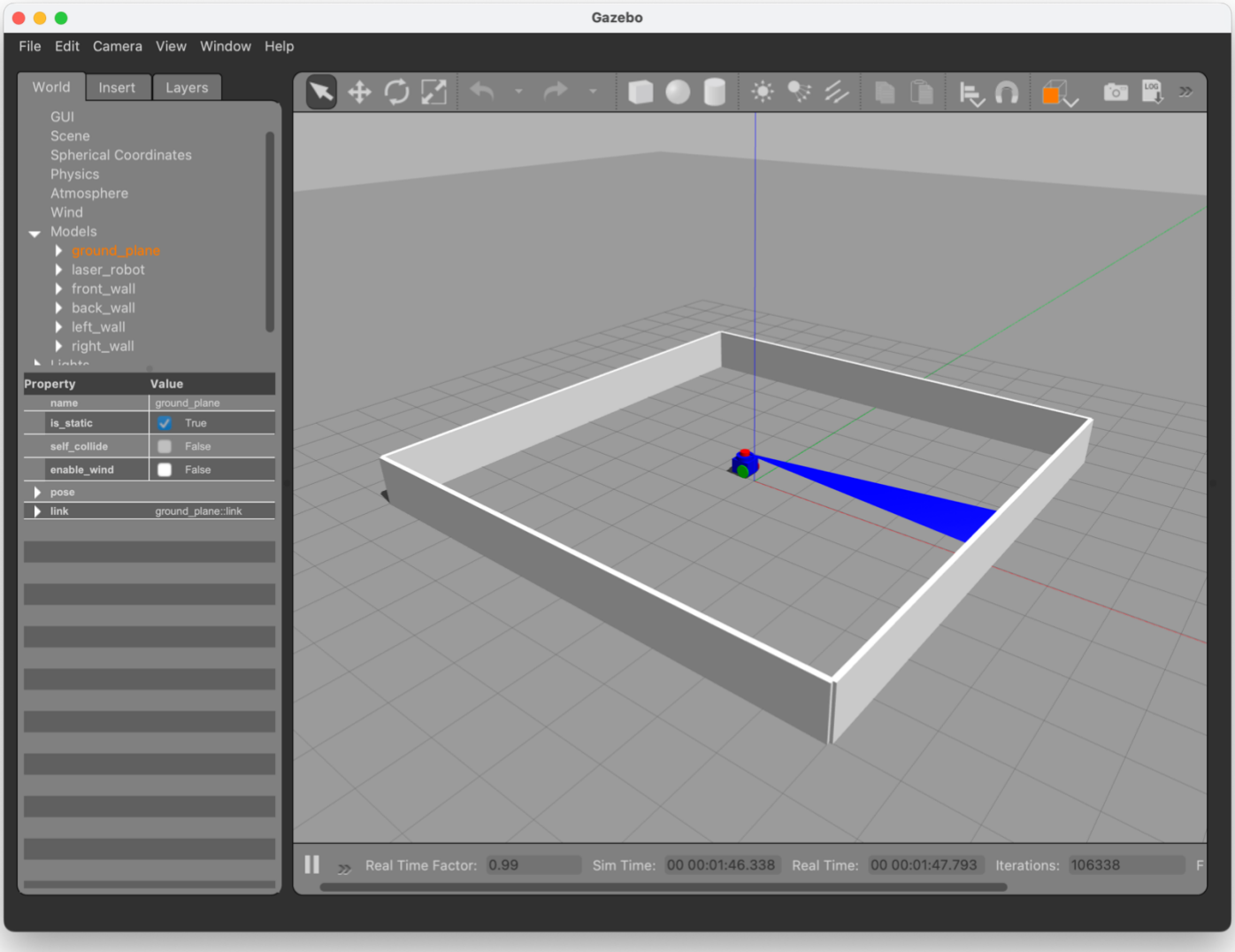
EECS 4421 LAB2

Name: Mahfuz Rahman

Student No.: 217847518

1. Gazebo View of the world



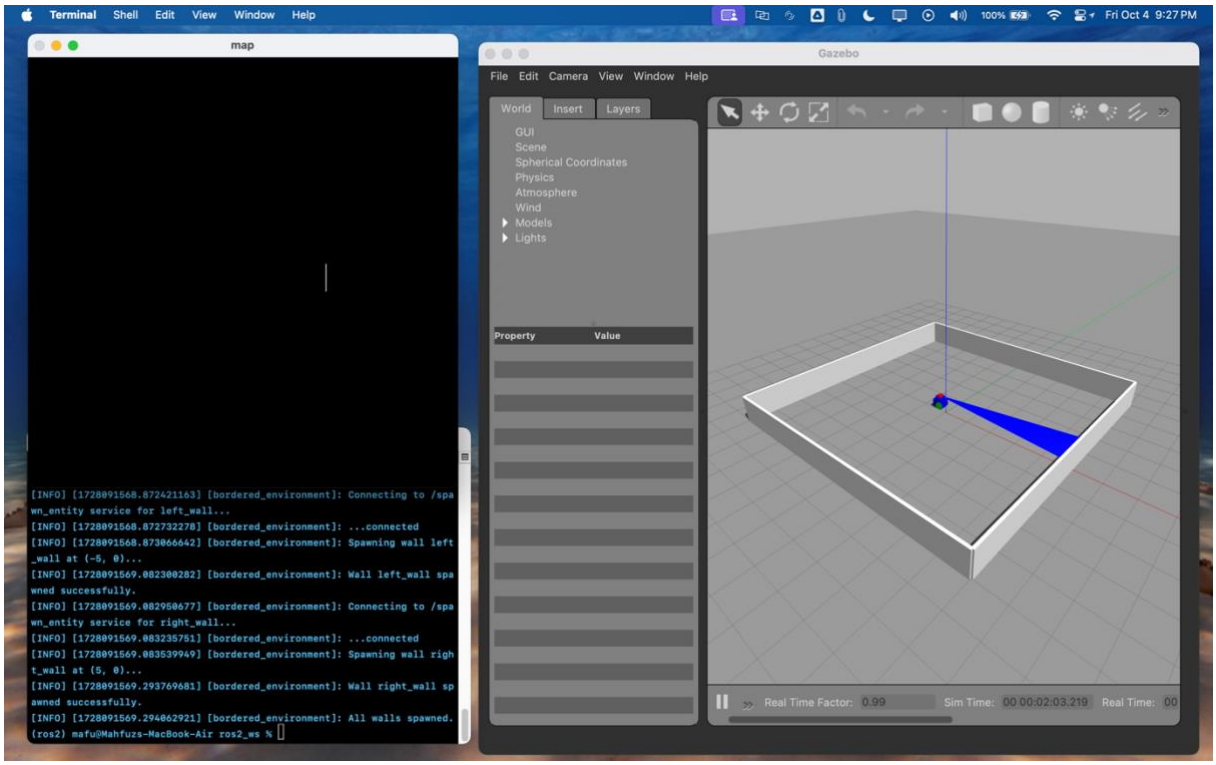
Code that generates the map (collect_lidar.py) :

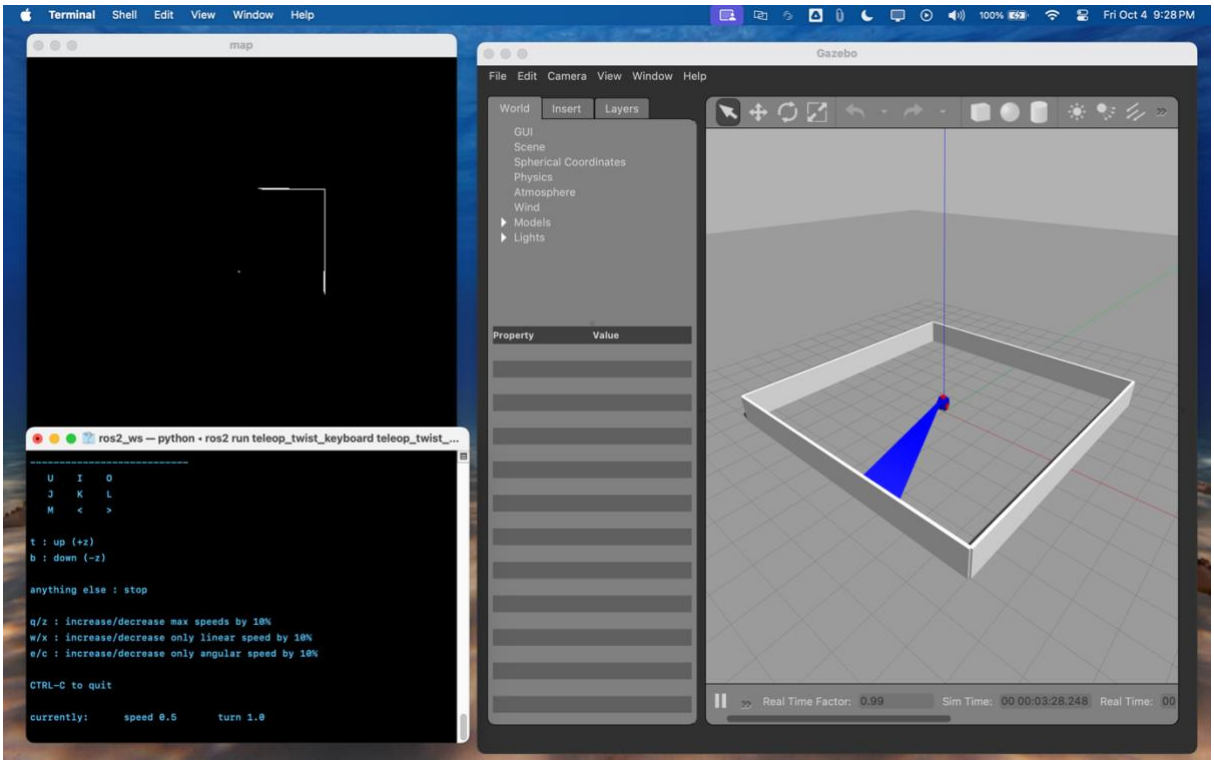
```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from sensor_msgs.msg import LaserScan
import cv2

def euler_from_quaternion(quaternion):
    """
    Converts quaternion (w in last place) to euler roll, pitch, yaw
    quaternion = [x, y, z, w]
    """
    x = quaternion.x
    y = quaternion.y
    z = quaternion.z
    w = quaternion.w

    sinr_cosp = 2 * (w * x + y * z)
    cosr_cosp = 1 - 2 * (x * x + y * y)
    roll = np.arctan2(sinr_cosp, cosr_cosp)

    sinp = 2 * (w * y - z * x)
    pitch = np.arcsin(sinp)

    siny_cosp = 2 * (w * z + x * y)
    cosy_cosp = 1 - 2 * (y * y + z * z)
    yaw = np.arctan2(siny_cosp, cosy_cosp)

    return roll, pitch, yaw

class CollectLidar(Node):
    _WIDTH = 513
    _HEIGHT = 513
    _M_PER_PIXEL = 0.05

    def __init__(self):
        super().__init__('collect_lidar')
```

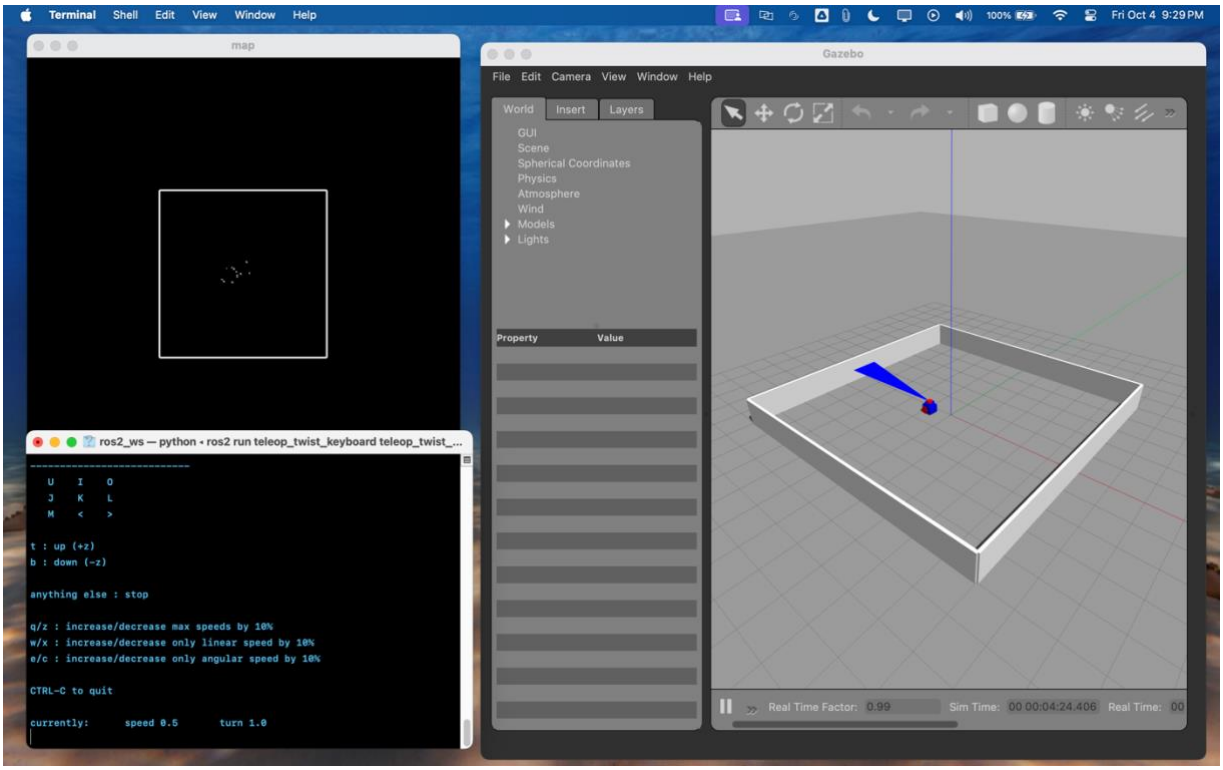
```
        self.get_logger().info(f'{self.get_name()} created')

        self._map = np.zeros((CollectLidar._HEIGHT, CollectLidar._WIDTH), dtype=np.uint8)

        self._cur_x = 0.0
        self._cur_y = 0.0
        self._cur_yaw = 0.0

        self.create_subscription(Odometry, "/odom", self._odom_callback, 1)
        self.create_subscription(LaserScan, "/scan", self._scan_callback, 1)

    def _scan_callback(self, msg):
        angle_min = msg.angle_min
        angle_increment = msg.angle_increment
        ranges = msg.ranges

        for i, r in enumerate(ranges):
            if r < msg.range_max:
                angle = angle_min + i * angle_increment

                # Converting from LIDAR polar coordinates to Cartesian coordinates
                x_car = r * math.cos(angle)
                y_car = r * math.sin(angle)

                # Transform from robot-relative cartesian coordinates to world coordinates
                x_world = self._cur_x + x_car * math.cos(self._cur_yaw) - y_car * math.sin(self._cur_yaw)
                y_world = self._cur_y + x_car * math.sin(self._cur_yaw) + y_car * math.cos(self._cur_yaw)

                # Convert world coordinates to grid coordinates
                grid_x = int(x_world / CollectLidar._M_PER_PIXEL + CollectLidar._WIDTH // 2)
                grid_y = int(y_world / CollectLidar._M_PER_PIXEL + CollectLidar._HEIGHT // 2)

                # Mark the grid cell as occupied if within bounds
                if 0 <= grid_x < CollectLidar._WIDTH and 0 <= grid_y < CollectLidar._HEIGHT:
                    self._map[grid_y, grid_x] = 255  # Occupied

        cv2.imshow('map', self._map)
        cv2.waitKey(10)

    def _odom_callback(self, msg):
```

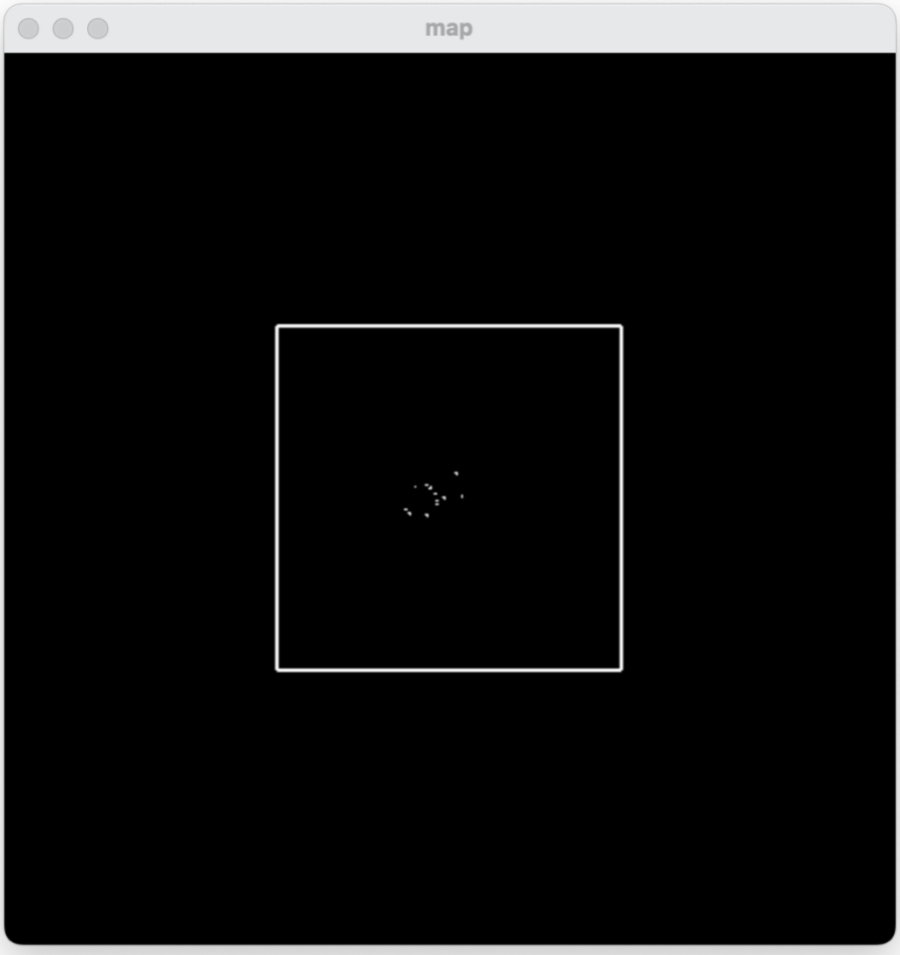
```
        pose = msg.pose.pose
        self._cur_x = pose.position.x
        self._cur_y = pose.position.y
        _, _, self._cur_yaw = euler_from_quaternion(pose.orientation)
        self.get_logger().info(f"Robot at ({self._cur_x}, {self._cur_y}, {self._cur_yaw})")

def main(args=None):
    rclpy.init(args=args)
    node = CollectLidar()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    rclpy.shutdown()

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

2. Mapping on a 2D map using LIDAR






Terminal Shell Edit View Window Help
Fri Oct 4 9:29 PM
map
Gazebo
File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Atmosphere
Wind
Models
Lights
Property Value
ros2_ws — python • ros2 run teleop_twist_keyboard teleop_twist_...
U I O
J K L
M < >
t : up (+z)
b : down (-z)
anything else : stop
q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
CTRL-C to quit
currently: speed 0.5 turn 1.0
Real Time Factor: 0.99
Sim Time: 00 00:04:24.406
Real Time: 00


map

1. **How did you decide to represent the map and update it when sensor measurements are obtained?**

   Since we are using a robot equipped with a LIDAR scanner, we get polar coordinates or basically a list of distances and angles that represents the distance to an object at a specific angle from the robot's position at the time.

   Thus, we can convert the polar coordinates to cartesian coordinates and plot onto a 2D map. For this, I used the following conversion:

$$x = r \times \cos(\theta)$$
$$y = r \times \sin(\theta)$$

   r is the range (distance) from the LIDAR.
   θ is the angle of the LIDAR beam. This is computed using *angle_min + i * angle_increment,* where *i* is the index of the range.

   However, the LIDAR data is relative to the robot's current position and orientation. We thus use the robot's global position which is x, y, and yaw angle from odometry to transform the LIDAR points and plot onto the map. We calculate this using the following:

$$X_{world} = X_{robot} + x \times \cos(\theta_{robot}) - y \times \sin(\theta_{robot})$$

$$Y_{world} = Y_{robot} + x \times \sin(\theta_{robot}) - y \times \cos(\theta_{robot})$$

   Now that we have the coordinates, we need to plot them onto our 2D map. To achieve this, we use the following method:

$$grid_x = \left\lfloor \frac{X_{world}}{resolution} \right\rfloor$$

$$grid_y = \left\lfloor \frac{Y_{world}}{resolution} \right\rfloor$$

   We then mark these grids onto the map as occupied.

2. **How did you decide where to drive the robot?**

   We can drive around the robot using a number of ways. I explored two methods, where one approach was simply to use the *teleop_twist_keyboard* node. The other approach was to drive the robot automatically utilizing the sensor data and a path planning algorithm to navigate around obstacles. The robot can leverage Dijkstra's algorithm to decide on a path to a goal location. If an obstacle is detected within a certain range, the robot adjusts its trajectory by steering away from the detected object.

3. **Was the model consistent over all measurement obtained?**

   Considering the simplicity of our environment and robot, the model was generally consistent over time. Since our environment was static, the readings from the robot's sensors were always same. However, I observed some factors that affected consistency due to the robot's movements. For instance sudden movements of the robot at high speed or sudden stopping. The robot's drifts over time (due to wheel slip or other factors), introduced inconsistencies in how sensor readings were integrated into the map.