EECS 4421 Assignment 3

Mahfuz Rahman

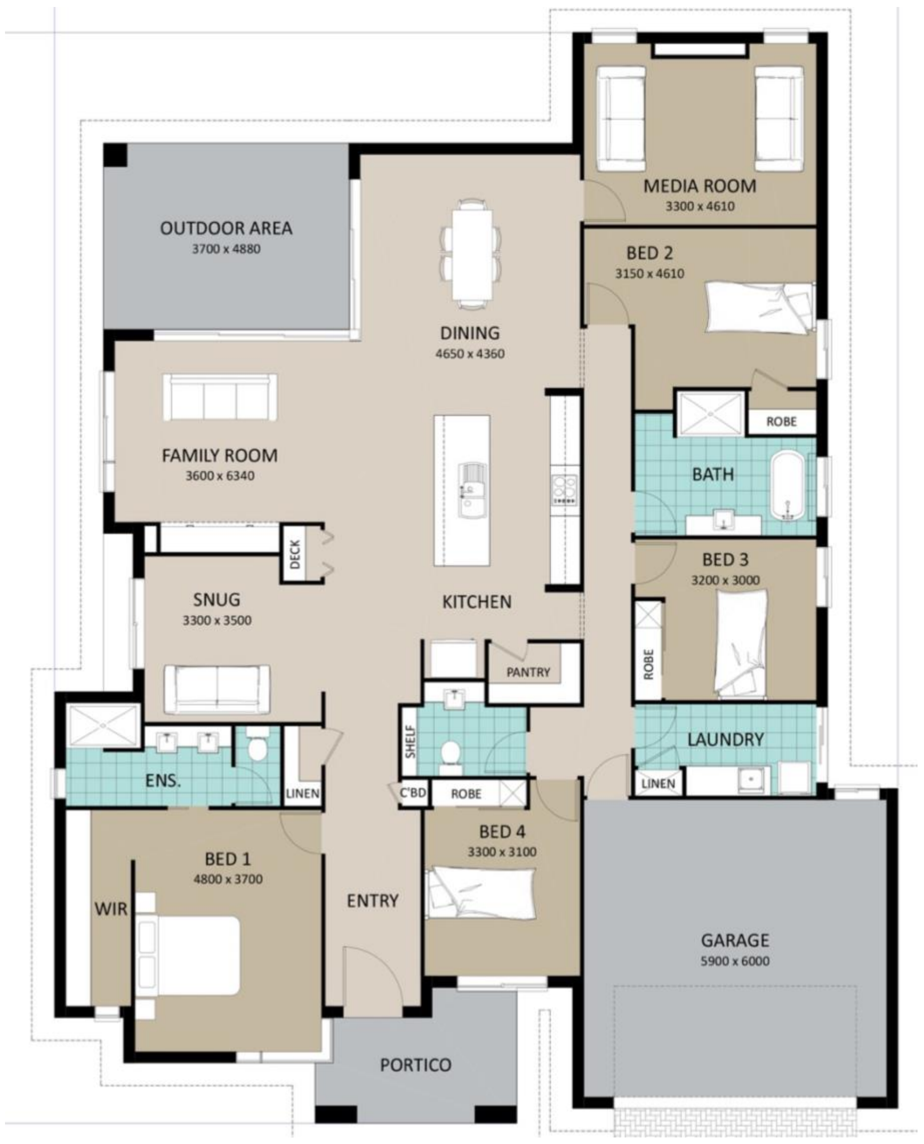
217847518

Assignment 3 Submission

1. Obtain a jpeg or similar blueprint of some constructed space. Something of reasonable complexity, but at least 20m x 20m as a jpeg or similar image. Scan this as an image where the size of a pixel is approximately .1m. This means your image will be at least 200x200 pixels in size. You will use this blueprint in two ways. (i) to build a Gazebo model that is consistent with the blueprint, and (ii) to do planning for a laser-equipped robot in this space. [no marks]

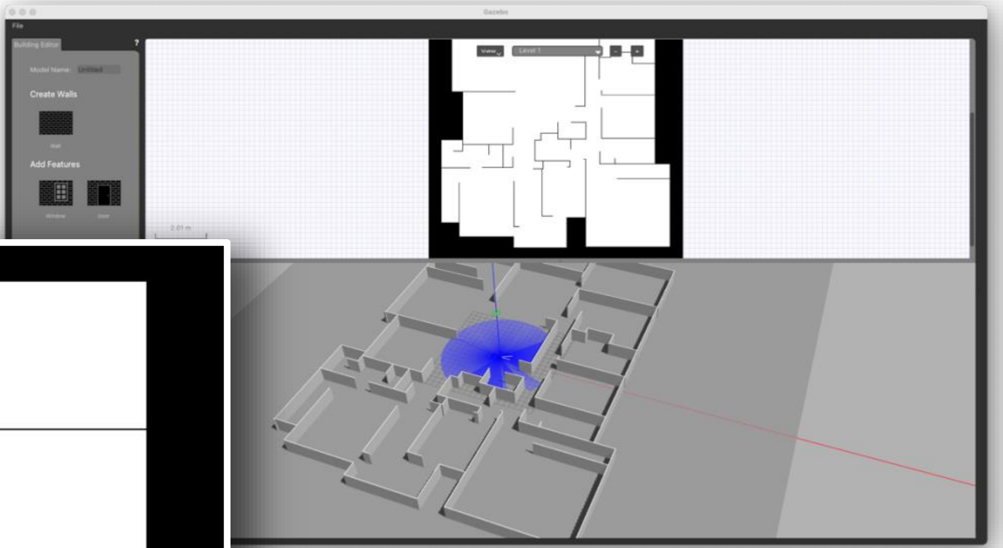
I found a blueprint of an apartment where the image size was 2480 x 3508 pixels. For simplicity, I scaled the image into 1000 x 1250 pixels.

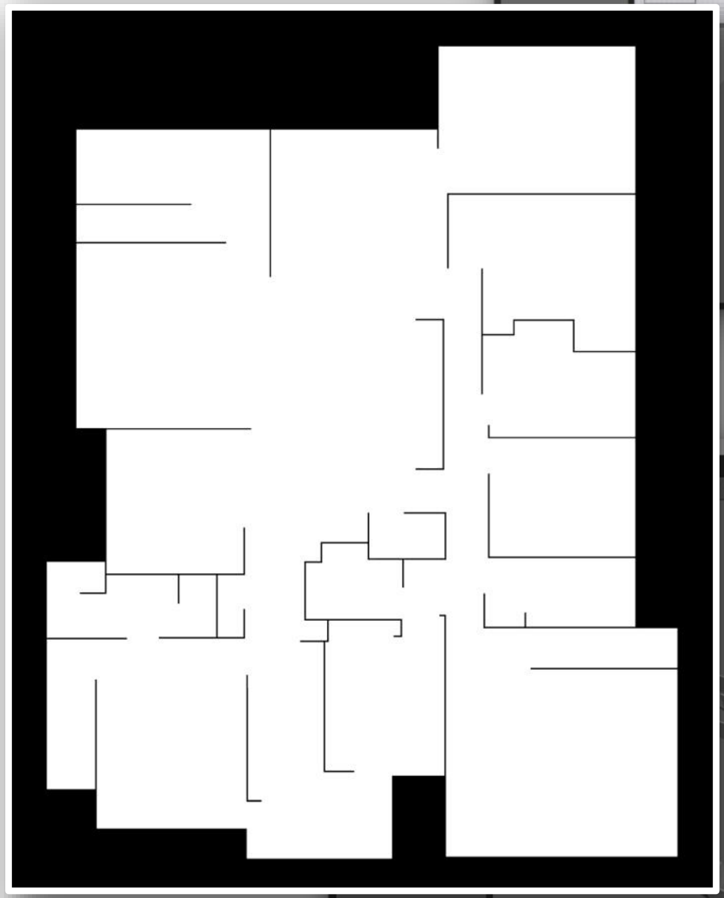
Blueprint:

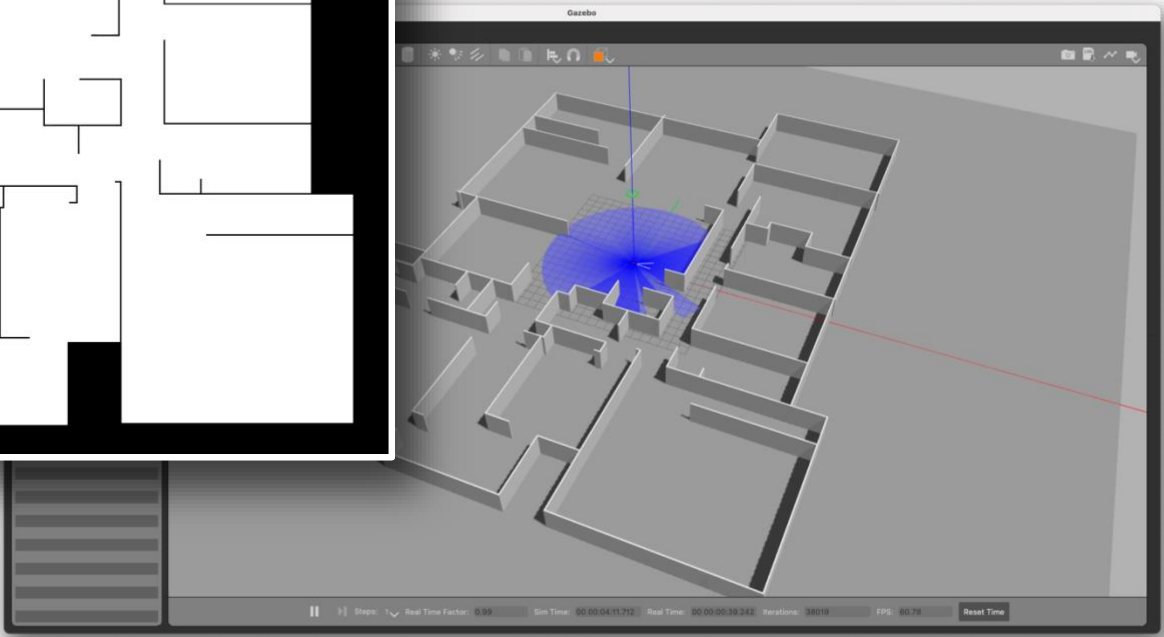


2. Construct a Gazebo world from the blueprint. To do this, define a north on your blueprint and run gazebo. Gazebo has a "Building Editor" within it which you can use to import the blueprint and draw walls on it. So for the example above, one would import that and draw 'interesting walls' for the robot. One might leave out all the uninteresting furniture, only keep walls (and doorways which are assumed to be open), remove doors, etc. For ease of use, assume a single floor. Note that you cannot edit what you have created once you exit the editor, but you can merge worlds together and you can manually edit the world file. To launch gazebo with no world just run 'ros2 launch gazebo_ros gazebo.launch.py'. And once you have built a world, you can launch gazebo with the word using 'ros2 launch gazebo_ros gazebo.launch.py world:=yourworld.world'. Hand in a pdf of your blueprint and views of the resulting Gazebo world. Show the Laser Robot operating in the world. Where is your origin in the world? [10 marks]

I used the above blueprint to develop a black and white image with only the walls and the necessary structures. This would be my occupancy map. I then used the building editor to build the world in gazebo. The origin in the world is the center where the robot spawns, i.e., (0,0,0). This corresponds directly to the map's center which is (500px, 625px).



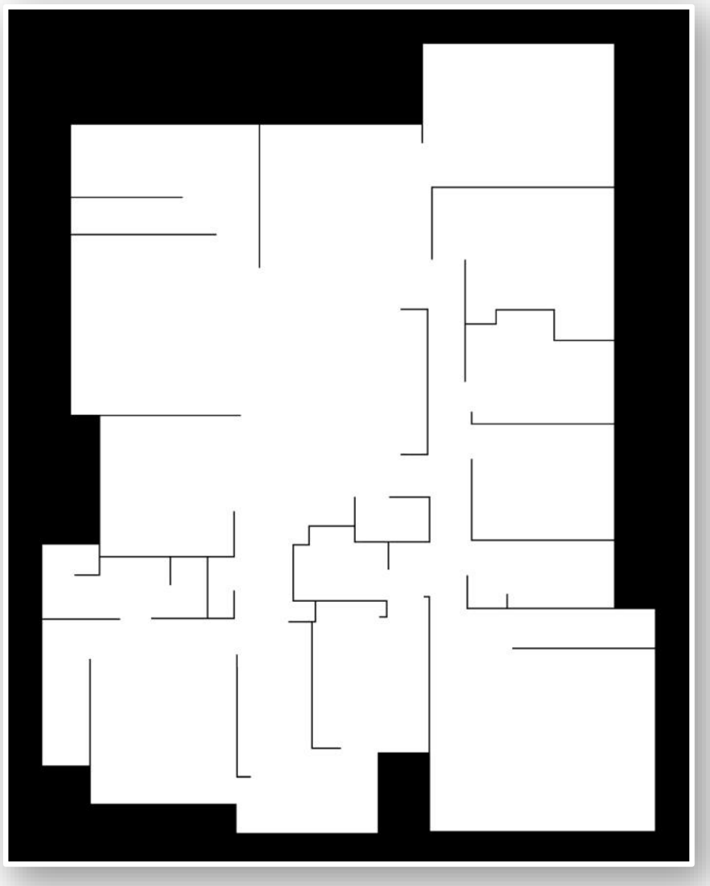




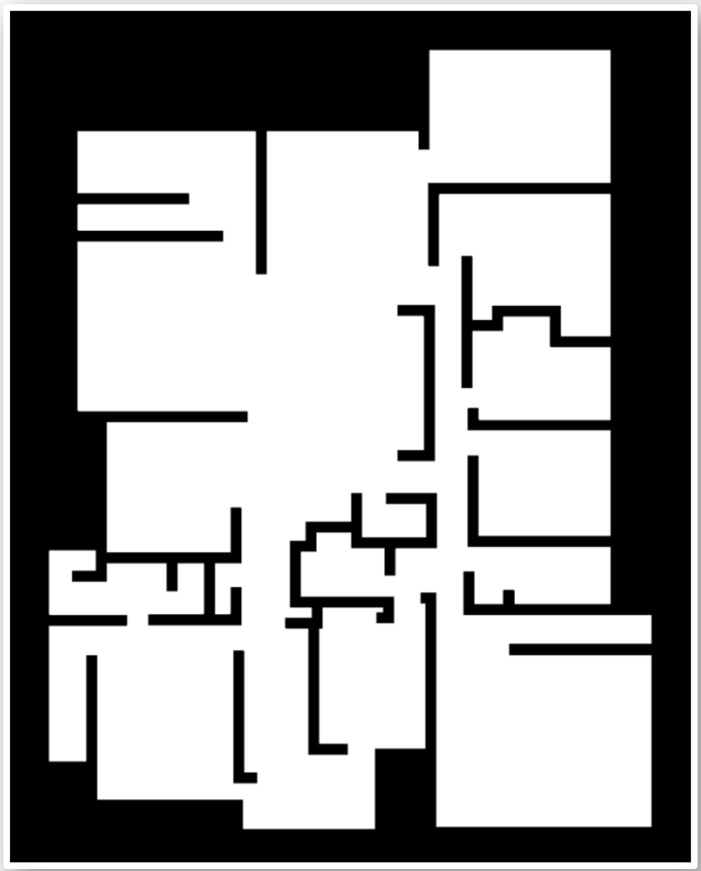
3. Take your blueprint and build an occupancy map that you can use with the RRT code from one of the labs. You should take the jpeg of your map and manually mark the walls (eliminating the objects that you did not include in your Gazebo world model). Once you have an occupancy map (similar to the one you created in the earlier lab), write code to/use some paint program to dilate all obstacles by a distance r, which corresponds to the radius of the Laser Robot. Dilation is relatively straightforward. Process every pixel in the occupancy map, and if it is adjacent to an occupied location, make it occupied after a pass through the map. This will dilate the map by one cell (whatever size cell you might have used. Dilate by the radius of the robot. Note that OpenCV has built in functions to do dilation, but you can easily do this by writing the code yourself. Augment your code from the lab so that you can take the pose information from Gazebo and use that as a start location on the map, and choose a goal location on the map, and produce a set of waypoints that if the robot was to follow using straight lines would get the robot from the start to the goal [40 marks].

As mentioned above, I built the occupancy map, and the corresponding dilated map as follows. The dilation is done during runtime, so it is convenient to always use the main occupancy map image during development.

Occupancy Map



Dilated Occupancy Map



So, overall, I use the RRT/RRT* algorithm to generate a set of waypoints. I use these set of waypoints to navigate around the map. Here's how the whole process works:

- I developed on my previous lab work to incorporate the **RRT Star** algorithm to plan a path from the robot's current location to its goal.
- This algorithm randomly samples points in the space and grows a tree of reachable points.
- The environment is scanned for walls or obstacles (represented by black pixels in the image).
- I also incorporated a 10% bias towards the goal, so the RRT algorithm samples points towards the goal 10% of the time.

- As a bonus, I incorporated obstacle detection from Lab8 and trigger an obstacle avoidance behavior.

- The robot's position is continuously updated using **odometry data** (robot's x, y coordinates and orientation (theta), which is received via the /odom topic.

- Since my path planning and world representation is based on image coordinates, I need to transform those coordinates into **Gazebo coordinates**. The function image_to_gazebo_coordinates() converts the 2D pixel-based coordinates (used for the image map) into the simulation space.

- Similarly, I use gazebo_to_image_coordinates() to convert the robot's position in Gazebo back to image coordinates. This allows me to visualize the robot's movement and path on the grid-like map I've created.

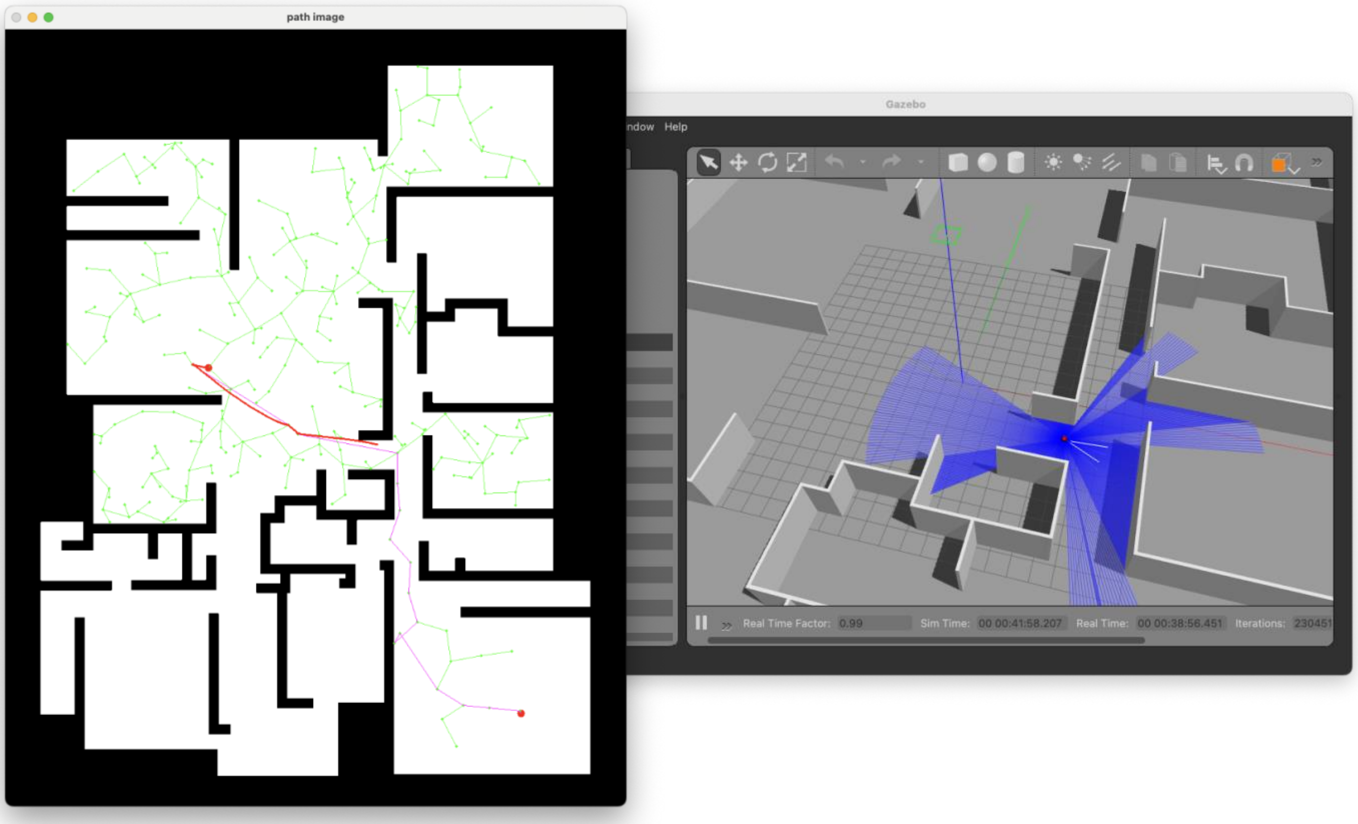
- I also incorporated utilizing the **ROS 2 service** which I use to start or stop the robot movement.
- When I receive a service request, I update the target coordinates, regenerate the path, and the robot begins moving towards the new goal. Currently, the start coordinates are taken based on the robot's position at the time, and the goal coordinates are hard coded into the code. I can later update to include the goal coordinates as parameters.

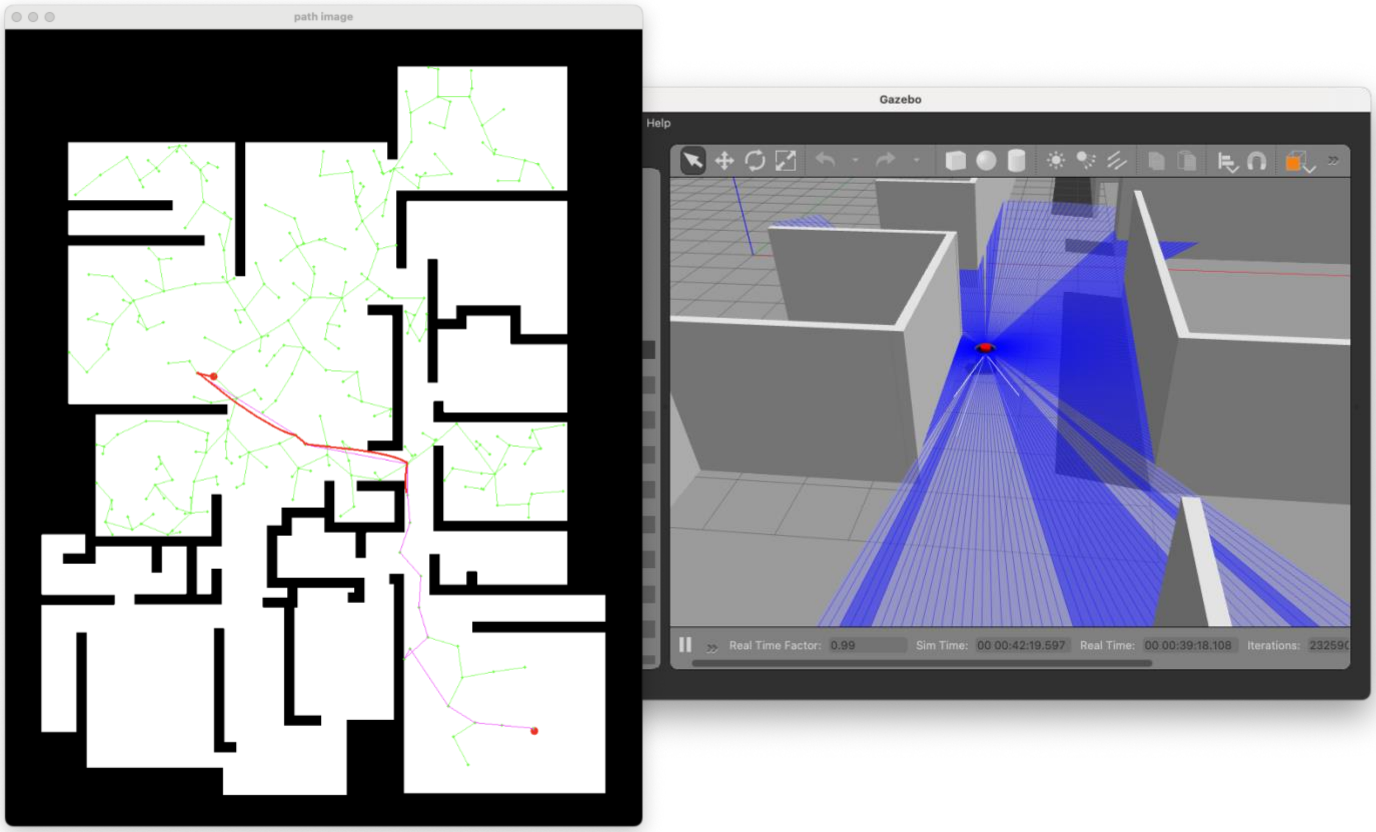
- To control the robot's movement, I adjust its **linear speed** (twist.linear.x) and **angular speed** (twist.angular.z) based on its position relative to the goal. If the robot is not facing the goal (as determined by comparing its current orientation with the heading toward the goal), I make it rotate until it is aligned with the target.

- As the robot moves towards each waypoint, I continuously update its position and check if it's within range of the next waypoint. Once it reaches a waypoint (within a tolerance distance), I move on to the next one. If all waypoints are reached, the robot stops and the task is complete.

- Running the code overall, showed that the algorithm found a path most of the time. In some exception cases with tight corners or where the robot's dimensions does not satisfy the space, the code lets the user know if path is not found.

Some snapshots. The magenta line shows the path calculated by RRT. The red line shows robots path so far:





The whole code:

```
import math
import numpy as np
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rcl_interfaces.msg import SetParametersResult
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist, Pose, Point, Quaternion
from nav_msgs.msg import Odometry
from std_srvs.srv import SetBool
from sensor_msgs.msg import LaserScan


import cv2
import json
import random
from datetime import datetime

WORLD_WIDTH = 1000
WORLD_HEIGHT = 1250

N = 10000  # Number of random samples
BACKGROUND_COLOR = (255, 255, 255)  # White background
OCCUPIED_COLOR = (0, 0, 0)  # Black for obstacles

ROBOT_RADIUS = 0.5
PIXEL_RESOLUTION = 0.07  # Each pixel represents 0.07 meters

def euler_from_quaternion(quaternion):
    """Converts quaternion (w in last place) to euler roll, pitch, yaw"""
    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

def load_image(image_path):    #loads the image and returns dilated the world
    blueprint_img = cv2.imread(image_path)

    gray_img = cv2.cvtColor(blueprint_img, cv2.COLOR_BGR2GRAY)
    _, thresholded_img = cv2.threshold(gray_img, 120, 255, cv2.THRESH_BINARY_INV)
    world = np.full((WORLD_HEIGHT, WORLD_WIDTH, 3), BACKGROUND_COLOR, dtype=np.uint8)
    world[thresholded_img == 0] = (0, 0, 0)

    robot_radius_pixels = int(ROBOT_RADIUS / PIXEL_RESOLUTION)
    kernel_size = robot_radius_pixels * 2 + 1
    kernel = np.ones((kernel_size, kernel_size), np.uint8)

    dilated_world = cv2.dilate(world, kernel, iterations=1)

    dilated_world = cv2.bitwise_not(dilated_world)

    return dilated_world

def is_line_free(world, point1, point2, step_size=1):
    """Check if the line between two points is free of obstacles."""
    x1, y1 = point1
    x2, y2 = point2

    # Calculate the total distance between the points
    distance = math.dist(point1, point2)

    # Calculate the number of steps based on the step size
    num_steps = int(distance / step_size)

    if num_steps == 0:
        return False

    # Calculate the increments for each step
    x_step = (x2 - x1) / num_steps
    y_step = (y2 - y1) / num_steps
```

```python
    # Iterate through each step along the line
    for i in range(num_steps + 1):  # +1 to include the endpoint
        # Calculate the current point along the line
        x = int(x1 + i * x_step)
        y = int(y1 + i * y_step)

        # Check if the current point is within an obstacle
        if np.array_equal(world[y, x], [0, 0, 0]):  # Black indicates an obstacle
            return False

    # If no obstacles were encountered, return True
    return True

def display_map(name, world):
    cv2.imshow(name, world)
    cv2.waitKey(10)

def image_to_gazebo_coordinates(path, image_width=WORLD_WIDTH, image_height=WORLD_HEIGHT,
scale_factor=0.056):
    """Converts image coordinates to Gazebo coordinates."""
    gazebo_path = []
    for image_x, image_y in path:
        gazebo_x = (image_x - image_width / 2) * scale_factor
        gazebo_y = (image_height / 2 - image_y) * scale_factor
        gazebo_path.append((gazebo_x, gazebo_y))
    return gazebo_path

def gazebo_to_image_coordinates(gazebo_x, gazebo_y, image_width=WORLD_WIDTH,
image_height=WORLD_HEIGHT, scale_factor=17.86):
    image_x = (gazebo_x * scale_factor) + image_width / 2
    image_y = (image_height / 2) - (gazebo_y * scale_factor)
    return (int(image_x), int(image_y))

class FindPath(Node):

    def _short_angle(self, angle):
        """Normalize an angle to be within the range [-pi, pi]."""
        if angle > math.pi:
            angle = angle - 2 * math.pi
        if angle < -math.pi:
            angle = angle + 2 * math.pi
        return angle

    def _compute_speed(self, diff, max_speed, min_speed, gain):
        """Compute the speed based on the difference."""
        speed = abs(diff) * gain
        speed = min(max_speed, max(min_speed, speed))
        return math.copysign(speed, diff)

    def _laser_callback(self, msg, mind=1.5):
        min_range = mind * 10
        for i, r in enumerate(msg.ranges):
            angle = msg.angle_min + i * msg.angle_increment
            if (abs(angle) < math.pi/4) and (r < min_range):
                min_range = r
        self._min_r = min_range

    def _startup_callback(self, request, resp):
        self.get_logger().info(f'Got a request {request}')
        if request.data:
            self._waypoints = self.process_path()
            if(self._waypoints != []):
                self.get_logger().info(f'robot starting')
                self._run = True
                resp.success = True
                resp.message = "Architecture running"
            else:
                resp.message = "Filed to obtain path. Try again"
        else:
            self.get_logger().info(f'robot suspended')
            self._run = False
            resp.success = True
            resp.message = "Architecture suspended"
        return resp

    def _avoid_obstacle(self, minr = 0.5):
        """ if there is an obstacle within mind of the front of the robot, stop and rotate"""
        if self._min_r < minr:
            twist = Twist()
            twist.linear.x = 0.0
            twist.angular.z = math.pi / 10
            return twist
        return None
```

```python
def __init__(self):
    super().__init__('find_path')
    self.get_logger().info(f'{self.get_name()} created')

    self._min_r = 10000
    self._goal_x = 830
    self._goal_y = 1100
    self._cur_x = 0.0
    self._cur_y = 0.0
    self._cur_theta = 0.0

    self._waypoints = [(0, 0)]
    self._current_waypoint_index = 0

    self._subscriber = self.create_subscription(Odometry, "/odom", self._listener_callback, 1)
    self.create_subscription(LaserScan, "/scan", self._laser_callback, 1)

    self._publisher = self.create_publisher(Twist, "/cmd_vel", 1)

    self.create_service(SetBool, '/startup', self._startup_callback)
    self._run = False

    self._world = load_image('blueprint_model_resized.jpg')



def process_path(self):  # returns the set of waypoints in gazebo coordinates
    start = gazebo_to_image_coordinates(self._cur_x, self._cur_y)
    goal = (self._goal_x, self._goal_y)

    cv2.circle(self._world, start, 2, (0, 165, 255), -1) #Orange
    cv2.circle(self._world, goal, 2, (255, 192, 203), -1) #Pink

    self.get_logger().info(f'calculating path...')
    #self._world, path = self.grow_rrt(self._world, start, goal, 50)
    self._world, path = self.grow_rrt_star(self._world, start, goal, 50)

    if path:
        for i in range(len(path) - 1):
            cv2.line(self._world, path[i], path[i + 1], (255, 0, 255), 1)  # Magenta line for path
    display_map("path image", self._world)
    path = image_to_gazebo_coordinates(path)

    return path

def grow_rrt(self, world_image, start, goal, d):
    world = world_image.copy()

    root = start
    tree = [root]
    edges = []

    parents = {root: None}  # Dictionary to store parent of each node

    goal_reached = False

    while not goal_reached:
        # Random sampling with goal bias
        if random.random() < 0.1:  # 10% chance to sample the goal
            new_point = goal
        else:
            x = random.randint(0, world.shape[1] - 1)
            y = random.randint(0, world.shape[0] - 1)
            new_point = (x, y)

        # Obstacle check
        if np.array_equal(world[new_point[1], new_point[0]], BACKGROUND_COLOR):
            # Find the nearest point in the tree to this new point
            nearest_point = min(tree, key=lambda node: math.dist(node, new_point))

            # Calculate distance and determine if new point should be at distance d
            distance = math.dist(nearest_point, new_point)
            if distance > d:
                # Calculate new point at distance d from the nearest point
                x3 = nearest_point[0] + (new_point[0] - nearest_point[0]) * d / distance
                y3 = nearest_point[1] + (new_point[1] - nearest_point[1]) * d / distance
                new_point = (int(x3), int(y3))

            # Check if the line from nearest_point to new_point is free of obstacles
            if is_line_free(world, nearest_point, new_point):
                cv2.line(world, nearest_point, new_point, (0, 255, 0), 1)  # Green line for connection
                cv2.circle(world, new_point, 1, (0, 255, 0), -1)
```

```python
                    tree.append(new_point)
                    edges.append((nearest_point, new_point))
                    parents[new_point] = nearest_point  # Store the parent of new_point

                    # Check if the goal has been reached
                    if math.dist(new_point, goal) <= 20:
                        cv2.line(world, new_point, goal, (0, 255, 0), 1)
                        print("Goal reached!")
                        goal_reached = True

                        # Add the goal to the tree and path
                        tree.append(goal)
                        parents[goal] = new_point

                        # Backtrack from the goal to start to get the path
                        path = []
                        current = goal
                        while current is not None:
                            path.append(current)
                            current = parents[current]
                        path.reverse()  # Reverse the path to go from start to goal

                        return world, path

        return world, []

    def grow_rrt_star(self, world_image, start, goal, d):
        radius = 200
        world = world_image.copy()

        cv2.circle(world, start, 6, (0, 0, 255), -1)
        cv2.circle(world, goal, 6, (0, 0, 255), -1)

        root = start
        tree = [root]
        edges = []

        parents = {root: None}  # Dictionary to store parent of each node
        costs = {root: 0}

        goal_reached = False
    #for _ in range(N):
    while not goal_reached:
        # Random sampling with goal bias
        if random.random() < 0.3:  # 10% chance to sample the goal
            new_point = goal
        else:
            x = random.randint(0, world.shape[1] - 1)
            y = random.randint(0, world.shape[0] - 1)
            new_point = (x, y)

        # Obstacle check
        if np.array_equal(world[new_point[1], new_point[0]], BACKGROUND_COLOR):
            # Find the nearest point in the tree to this new point
            nearest_point = min(tree, key=lambda node: math.dist(node, new_point))

            # Calculate distance and determine if new point should be at distance d
            distance = math.dist(nearest_point, new_point)
            if distance > d:
                # Calculate new point at distance d from the nearest point
                x3 = nearest_point[0] + (new_point[0] - nearest_point[0]) * d / distance
                y3 = nearest_point[1] + (new_point[1] - nearest_point[1]) * d / distance
                new_point = (int(x3), int(y3))

            # Check if the line from nearest_point to new_point is free of obstacles
            if is_line_free(world, nearest_point, new_point):
                cv2.line(world, nearest_point, new_point, (0, 255, 0), 1)  # Green line for connection
                cv2.circle(world, new_point, 2, (0, 255, 0), -1)

                tree.append(new_point)
                edges.append((nearest_point, new_point))
                parents[new_point] = nearest_point  # Store the parent of new_point
                costs[new_point] = costs[nearest_point] + distance

                # Rewiring step: Check nearby nodes for potential shorter paths
                nearby_nodes = [
                    node for node in tree
                    if node != new_point and math.dist(node, new_point) <= radius
                ]
                for node in nearby_nodes:
                    potential_cost = costs[new_point] + math.dist(new_point, node)
                    if potential_cost < costs[node] and is_line_free(world, new_point, node):
                        # Update parent and cost for the node if a shorter path is found
                        parents[node] = new_point
                        costs[node] = potential_cost

                # Check if the goal has been reached
                if math.dist(new_point, goal) <= 20:
                    cv2.line(world, new_point, goal, (0, 255, 0), 1)
                    print("Goal reached!")
                    goal_reached = True

                    # Add the goal to the tree and path
                    parents[goal] = new_point
                    costs[goal] = costs[new_point] + math.dist(new_point, goal)


                    # Backtrack from the goal to start to get the path
                    path = []
                    current = goal
                    while current is not None:
                        path.append(current)
                        current = parents[current]
                    path.reverse()  # Reverse the path to go from start to goal

                    return world, path
    return world, []

def _listener_callback(self, msg):
    """Callback to update the robot's current position and drive towards the next waypoint."""
    pose = msg.pose.pose
    roll, pitch, yaw = euler_from_quaternion(pose.orientation)
    self._cur_x = pose.position.x
    self._cur_y = pose.position.y
    self._cur_theta = self._short_angle(yaw)

    #self.get_logger().info(f"x: {self._cur_x} y: {self._cur_y}")

    if self._run and (self._current_waypoint_index < len(self._waypoints)):
        target_x, target_y = self._waypoints[self._current_waypoint_index]

        cv2.circle(self._world, (gazebo_to_image_coordinates(self._cur_x, self._cur_y)), 1, (0, 0, 255), -1)
        display_map("path image", self._world) # draw the path live

        #avoid = self._avoid_obstacle()
        avoid = None
```

```python
            if avoid is not None:
                self.get_logger().info(f'avoiding')
                self._publisher.publish(avoid)
                return
            elif self._drive_to_goal(target_x, target_y):
                 self._current_waypoint_index += 1

            if self._current_waypoint_index >= len(self._waypoints):
                self.get_logger().info('All waypoints reached!')
                self._run = False

        else:
            twist = Twist()
            twist.linear.x = 0.0
            twist.angular.z = 0.0
            self._publisher.publish(twist)



    def _drive_to_goal(self, goal_x, goal_y, heading_tol=0.15, range_tol=0.15):
        """Drive the robot to the target goal."""
        twist = Twist()

        x_diff = goal_x - self._cur_x
        y_diff = goal_y - self._cur_y
        dist = math.sqrt(x_diff * x_diff + y_diff * y_diff)

        if dist > range_tol:
            #self.get_logger().info(f'{self.get_name()} driving to goal x: {goal_x} y: {goal_y}')
            heading = math.atan2(y_diff, x_diff)
            diff = self._short_angle(heading - self._cur_theta)

            if abs(diff) > heading_tol:
                twist.angular.z = self._compute_speed(diff, 0.5, 0.5, 0.2)
                # self.get_logger().info(f'{self.get_name()} turning towards goal heading {heading} current {self._cur_theta} diff {diff}')
                self._publisher.publish(twist)
                return False

            twist.linear.x = self._compute_speed(dist, 0.7, 0.1, 0.2)
            self._publisher.publish(twist)
            # self.get_logger().info(f'{self.get_name()} moving forward, distance: {dist}')
            return False
        else:
            self.get_logger().info(f'{self.get_name()} reached waypoint ({goal_x}, {goal_y})')
            return True

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

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