# 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:

A floor plan of a house

Description automatically generated

1. 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).

A computer screen shot of a building

Description automatically generated

A white rectangular object with black lines

Description automatically generated with medium confidenceA computer screen shot of a blueprint

Description automatically generated

1. 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.

A white rectangular object with black lines

Description automatically generated with medium confidence

Dilated Occupancy Map

Occupancy Map

A black and white maze

Description automatically generated

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:

A map of a building

Description automatically generated

A computer screen shot of a blueprint

Description automatically generated

A computer screen shot of a building

Description automatically generatedA map of a building

Description automatically generated

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

# 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

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)

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

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()