```
PART
01: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!

#!/usr/bin/env python3

import numpy as np
import cv2
import heapq
import copy
import time
import functools

HEIGHT = 300 # cm
WIDTH = 600 # cm

## turtle 3 wafflepi
# Wheel Radius (R): 33 mm
# Robot Radius (r): 220 mm
# Wheel Distance (L): 287 mm

WRADIUS = .033
RRADIUS = .22
WDIS = .287

# Timer decorator to measure execution time of functions
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()  # Start time
        result = func(*args, **kwargs)  # Execute the wrapped function
        end_time = time.perf_counter()  # End time
        run_time = end_time - start_time  # Calculate runtime
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return result  # Return the result of the wrapped function
    return wrapper

class Node:
    def __init__(self, Node_Cost, Node_Cost_est, Node_x, Node_y, Node_theta,
Parent_Node_x, Parent_Node_y, Parent_Node_theta, rpmL, rpmR):
        self.Node_Cost = Node_Cost # The cost to reach this node.
        self.Node_Cost_est = Node_Cost_est # The estimated cost to the goal.
        self.Node_x = Node_x # The node's x location.
        self.Node_y = Node_y # The node's y location.
        self.Node_theta = Node_theta # The node's angle.
        self.Parent_Node_x = Parent_Node_x # The node's parent's x location.
        self.Parent_Node_y = Parent_Node_y # The node's parent's y location.
        self.Parent_Node_theta = Parent_Node_theta # The node's parent's angle.
        self.rpmL = rpmL # The speed of the left motor to reach this point.
rads/sec
        self.rpmR = rpmR # The speed of the right motor to reach this point.
rads/sec

    # This method allows the heapq module to compare Node objects by their cost
when sorting.
    # This ensures that the node with the smallest cost plus heuristic is popped
first.
    # The heuristic used is Euclidean distance.
    def __lt__(self, other):
        return self.Node_Cost + self.Node_Cost_est < other.Node_Cost +
other.Node_Cost_est


# create_new_node creates a new node from a given node and a set of wheel
```

```
velocities.
# given_Node is the node we are travelling from. revs = (left wheel speed, right
wheel speed)
# which we use to determine how the robot is moving. goal_x and goal_y are the
end goal points
# for the path planner. We use this to calculate the heuristic to the goal.
obstacles is the image
# map of where all the obstacles are located. It is gray scale and equals 255
whenever an
# obstacle is present. We use this to determine if the motion from our start
point to our final
# point is obstacle free. The function returns the newNode generated from the
motion along with
# whether the path is obstacle free. If there are no obstacles encountered
during the motion, then
# valid_path is True.
def create_new_node(given_Node, revs, goal_x, goal_y, obstacles):

    # valid_curve takes in the entire array of x and y points. I then compares
all points
    # to the obstacle map to see if any lie out of the bounds of the map or in
an obstacle.
    # It returns true if all points in the path are obstacle free and in the
map.
    def valid_curve(x_arr, y_arr, map_shape, obstacles):
        # Round each point to nearest integer and convert to an int.
        x_arr = np.round(x_arr).astype(int)
        y_arr = np.round(y_arr).astype(int)

        # Check that all points are within the bounds of the map.
        in_bounds = (0 <= x_arr) & (x_arr < map_shape[1]) & (0 <= y_arr) &
(y_arr < map_shape[0])

        # Clip indices to avoid indexing errors.
        x_arr_clipped = np.clip(x_arr, 0, map_shape[1] - 1)
        y_arr_clipped = np.clip(y_arr, 0, map_shape[0] - 1)

        # Check that all points within the map are not obstacles.
        obstacle_free = obstacles[y_arr_clipped, x_arr_clipped] == 0

        # Rreturn True if all points are valid/obstacle free.
        return np.all(in_bounds & obstacle_free)

    # Get the left and right wheel velocities in rads/sec.
    UL, UR = revs

    r = WRADIUS  # Wheel radius
    L = WDIS  # Distance between wheels

    # Get the starting X, Y, and theta position.
    Xi = given_Node.Node_x
    Yi = given_Node.Node_y
    Thetai = given_Node.Node_theta

    # Time array. Get time steps of .1 seconds up until 3 seconds.
    t = np.linspace(0, 3, 100)
    dt = t[1] - t[0]

    # Angular velocity.
    # Note: Thetan = np.deg2rad(Thetai) + w * t was changed to Thetan =
np.deg2rad(Thetai) - w * t
    # because the sign convention was incorrect by observation.
    w = (r / L) * (UR - UL)
    Thetan = np.deg2rad(Thetai) - w * t
    # Linear velocity.
    v = 0.5 * r * (UL + UR)

    # Compute deltas.
    dx = v * np.cos(Thetan) * dt
    dy = v * np.sin(Thetan) * dt

    # Integrate position.
    x_points = np.cumsum(dx)*100 + Xi
    y_points = np.cumsum(dy)*100 + Yi

    # Total distance traveled.
    total_distance = np.sum(np.hypot(np.diff(x_points), np.diff(y_points)))

    # Final orientation.
    final_theta = np.rad2deg(Thetan[-1]) % 360

    # Get the parent.
    parent_x = given_Node.Node_x
    parent_y = given_Node.Node_y
    parent_theta = given_Node.Node_theta

    # Increment the cost from the movement.
    cost = given_Node.Node_Cost + total_distance

    # Get the final position and orientation.
    x_pos = x_points[-1]
    y_pos = y_points[-1]
    theta_pos = final_theta

    # Get the estimated cost to the goal.
    cost_est = np.sqrt((given_Node.Node_x - goal_x)**2 + (given_Node.Node_y - 
goal_y)**2)

    # Generate the new node.
    newNode = 
Node(cost,cost_est,x_pos,y_pos,theta_pos,parent_x,parent_y,parent_theta, 
rpmL=UL, rpmR=UR)

    # Validate entire path.
    map_shape = obstacles.shape
    valid_path = valid_curve(x_points, y_points, map_shape, obstacles) # True or 
False

    # Return the new node and if it is valid (obstacle free movement).
    return newNode, valid_path

# Convert the angle given, in degrees, to an index from 0-7.
def angle_to_index(angle):
    # Normalize angle to the range [0, 360).
    angle = angle % 360

    # Compute the index by dividing the angle by 45.
    return int(angle // 45)

def gen_obstacle_map():
    # Set the height and width of the image in pixels.
    height = HEIGHT
    width = WIDTH
    # Create blank canvas.
    obstacle_map = np.zeros((height,width,3), dtype=np.uint8 )

    # Define polygons for rectangle obstacles.
    def l_obstacle1(x,y):
        return (100 <= x <= 110) and (0 <= y <= 200)

    def l_obstacle2(x,y):
        return (210 <= x <= 220) and (100 <= y <= 300)

    def l_obstacle3(x,y):
        return (320 <= x <= 330) and (0 <= y <= 100)

    def l_obstacle4(x,y):
        return (320 <= x <= 330) and (200 <= y <= 300)

    def l_obstacle5(x,y):
        return (430 <= x <= 440) and (0 <= y <= 200)

    # For every pixel in the image, check if it is within the bounds of any
obstacle.
    # If it is, set it's color to red.
    for y in range(height):
        for x in range(width):
            if (l_obstacle1(x, y) or l_obstacle2(x,y) or l_obstacle3(x,y) or
l_obstacle4(x,y)
                or l_obstacle5(x,y)):
                obstacle_map[y, x] = (0, 0, 255)


    # The math used assumed the origin was in the bottom left.
    # The image must be vertically flipped to satisy cv2 convention.
    return np.flipud(obstacle_map)

# expand_obstacles takes the obstacle map given by gen_obstacle_map as an image,
along with
# the scale factor sf, and generates two images. The first output_image, is a
BGR image
# to draw on used for visual display only. expanded_mask is a grayscale image
with white
# pixels as either obstacles or clearance space around obstacles. This function
will take
# the given obstacle image and apply a specified radius circular kernel to the
image. This ensures
# an accurate clearance around every obstacle.
def expand_obstacles(image, radius):

    # Convert image to HSV.
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define color mask for red and create grayscale image.
    lower_red = np.array([0, 200, 200])
    upper_red = np.array([25, 255, 255])
    obstacle_mask = cv2.inRange(hsv, lower_red, upper_red)

    # Create circular structuring element for expansion
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2 * radius + 1, 2 *
radius + 1))
    # Apply kernel to get dilation around all elements.
    expanded_mask = cv2.dilate(obstacle_mask, kernel, iterations=1)

    # Apply dilation to all of the borders.
    h, w = expanded_mask.shape
    expanded_mask[:radius+1, :] = 255  # Top border
    expanded_mask[h-radius:, :] = 255  # Bottom border
    # expanded_mask[:, :radius+1] = 255  # Left border
    # expanded_mask[:, w-radius:] = 255  # Right border
    # Create the output image and apply color orange to all obstacle and
clearance
    # pixels.
    output_image = image.copy()
    output_image[np.where(expanded_mask == 255)] = [0, 165, 255]  # Color orange

    # Restore original red pixels. This creates an image with red obstacles,
    # and orange clearance zones.
    output_image[np.where(obstacle_mask == 255)] = [0, 0, 255]

    return output_image, expanded_mask

# Prompt the user for a point. prompt is text that specifies what
# type of point is to be given. prompt is solely used for terminal text output.
# obstacles is used to dictate the map's bounadires. This ensures points given
lie
# within the map's bounds.
# The function returns the user's points as integers. It also prompts the user
for
# an angle if the prompt is "start" and ensures that the angle is between
[0,360).

def get_point(prompt, obstacles):
    valid_input = False

    # Prompt the user until a valid input is given.
    while not valid_input:
        try:
            x = int(input(f"Enter the x-coordinate for {prompt} (int): "))
            y = int(input(f"Enter the y-coordinate for {prompt} (int): ")) + 150
        except ValueError:
            print("Invalid input. Please enter a numerical value.")
            continue

        if prompt == "start":
            # Ensure theta meets constraints.
            while True:
                try:
                    theta = int(input(f"Enter the theta-coordinate for {prompt}
(must be 0-359): "))
                    if 0 <= theta < 360:
                        break
                    else:
                        print("Invalid theta. It must be between 0 and 359.
Please try again.")
                except ValueError:
                    print("Invalid input. Please enter an integer value for
theta.")

        # Correct the y value to account for OpenCV having origin in top left.
        obstacle_y = obstacles.shape[0] - y

        # Validate position against obstacles
        if valid_move(x, obstacle_y, obstacles.shape, obstacles):
            valid_input = True
        else:
            print("Invalid Input. Within Obstacle. Please try again.")
    if prompt == "start":
        return int(x), int(y), int(theta)
    else:
        return int(x), int(y)

# valid_move checks if a given point lies within the map bounds and
# if it is located within an obstacle. If the point is in the image and NOT in
```

```
an obstacle,
# it returns True, meaning the position is valid/Free/open space.
def valid_move(x, y, map_shape, obstacles):
    return 0 <= x < map_shape[1] and 0 <= y < map_shape[0] and obstacles[int(y),
int(x)] == 0

# Check if we are at the goal position. This checks if the point is within a
circular region around
# the goal.
def goal_check(x, y, end_x, end_y):
    # Compute Euclidean distance to goal.
    dis = np.sqrt(((end_x - x))**2 + (end_y - y)**2)
    # Check if position is within 5 cms.
    if dis < 5:
        return True
    else:
        return False

# gen_node_key takes a node and uses its properties to generate a 'node_key'.
This is an index
# used for the closed and seen sets.
def gen_node_key(node):
    node_key = (int(round(node.Node_y)), int(round(node.Node_x)),
angle_to_index(node.Node_theta))
    return node_key

# A_star_search uses the A star method to explore a map and find the path from
start to end.
# map is the image to draw on. Obstacles is a grayscale image of 1 cm resolution
to path plan on.
# start consists of the desired starting x, y, and theta positions. end includes
the desired x, y
# ending positions. Revs is a list of eight wheel revolution combinations to
determine the action set
# of the robot.
@timer
def A_star_search(map, obstacles, start, end, Revs):

    # Convert y coordinates from origin bottom left (user input) to origin top
left (cv2 convention).
    height, width, _ = map.shape
    start_x, start_y, start_theta = start
    end_x, end_y = end
    start_y = height - start_y
    end_y = height - end_y
    start = (start_x, start_y, start_theta)
    end = (end_x, end_y)

    # Open video file to write path planning images to.
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video_filename = "A_star_Proj3_phase2_video.mp4"
    fps = 60
    video_out = cv2.VideoWriter(video_filename, fourcc, fps, (width, height))

    # Create the start node.
    start_node = Node(0, 0, start[0], start[1], start[2], start[0], start[1],
start[2], 0, 0)

    open_set = []  # Priority queue. Used to extract nodes with smallest cost
from forward search.
    heapq.heappush(open_set, start_node)

    # The path planning occurs on a positional resolution of 1 cm. This sets the
dimensions for our
```

```
    # sets to check if points are duplicates.
    height = int(height)
    width = int(width)

    # The seen set is how I track if a node has already been given a cost. It is
a boolean matrix that checks if
    # every possible position/orientation combination has been seen. True =
seen. seen nodes are in the open or closed set.
    seen = np.full((height, width, 8), False, dtype=bool)
    start_angle_index = angle_to_index(start_theta)
    seen[start_y, start_x, start_angle_index] = True

    # This is my seen set, but as a dictionary to store all node information.
    visited = {}
    visited[(start_y, start_x, start_angle_index)] = start_node

    # This is the closed set. It stores nodes that have been fully explored.
    closed_set = np.full((height, width, 8), False, dtype=bool)

    # Create a list of functions of all the types of moves we can execute.
    RPM1 = Revs[0]
    RPM2 = Revs[1]
    directions = [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2, RPM2],
[RPM1,RPM2],[RPM2,RPM1]]

    # Draw the start and end points as a magenta and cyan circle, respectively.
    cv2.circle(map, (start[0],start[1]), 5, (255, 0, 255), -1)
    cv2.circle(map, (end[0], end[1]), 5, (255, 255, 0), -1)

    # Used to store only every 100th frame to the video. Otherwise the video is
hours long.
    # Additionally, writing frames takes the most computation for every loop.
    video_frame_counter = 0

    # Continue to search while the open_set is not empty.
    while open_set:

        # Get the node with the smallest cost from the open_set.
        current_node = heapq.heappop(open_set)
        # Extract it's x and y position.
        current_x, current_y, current_theta = current_node.Node_x,
current_node.Node_y, current_node.Node_theta

        # Verify that this position is not in the closed set.
        # Skip this iteration if it is in the closed_set as the position
        # has already been fully explored. This is required because
        # there is no efficient implementation to updating nodes within a heapq.
        # As such, a node may be added to the heapq, then added again to the
heapq if
        # a better parent was found.
        if closed_set[int(current_y), int(current_x),
angle_to_index(current_theta)] == True:
            continue

        # Add the current node to the closed set.
        closed_set[int(current_y), int(current_x),
angle_to_index(current_theta)] = True

        # If the node currently popped is within the goal region, get the path.
        if goal_check(current_x, current_y, end_x, end_y):
            print("Goal Reached!")
            # Generate final path.
            path_xys, path_rpms = get_final_path(visited, current_node)
            print(path_xys)
            print(f"end_x: {end_x}")
            print(f"end_y: {end_y}")

            print(path_rpms)

            # For each pixel in the path, draw a magenta line
            for i in range(len(path_xys) - 1):
                x1, y1 = path_xys[i]
                x2, y2 = path_xys[i + 1]
                cv2.line(map, (int(x1), int(y1)), (int(x2), int(y2)), (255, 0, 
255), 2)
                video_out.write(map)

            # Release the video file.
            video_out.release()
            # Terminate search and return the final map with the path and area 
explored.
            return map

        # Increment the video_frame_counter and save a frame if it is the 100th 
frame.
        video_frame_counter += 1
        if video_frame_counter == 100:
            # Redraw start and end circles.
            cv2.circle(map, (start_x, start_y), 5, (255, 0, 255), -1)
            cv2.circle(map, (end_x, end_y), 5, (255, 255, 0), -1)
            # Save current map state as a frame in the final video.
            video_out.write(map)
            # Reset the frame counter.
            video_frame_counter = 0

        # For the current node, apply each of the eight movement directions and 
examine
        # the newNode generated from moving in each direction.
        for revs in directions:
            # Get newNode from current move.
            newNode, valid_path = create_new_node(current_node, revs, end_x, 
end_y, obstacles)

            if valid_path: # Check that it isn't in an obstacle.

                # Get the node_key / index for this node.
                node_key = gen_node_key(newNode)

                if closed_set[node_key] == False: # Check that it is not in the 
closed set.
                    if seen[node_key] == False: # Check that it isn't in the 
open nor closed lists.

                        # Add it to the seen set.
                        seen[node_key] = True

                        # Add it to the visited set.
                        visited[node_key] = newNode
                        heapq.heappush(open_set, newNode)

                    # If the node is in the open list AND the new cost is 
cheaper than the old cost to this node, rewrite it
                    # within visited and add the newNode to the open_set. The 
old version will be safely skipped.
                    elif seen[node_key] == True:
                        if visited[node_key].Node_Cost > newNode.Node_Cost:
                            visited[node_key] = newNode
                            heapq.heappush(open_set, newNode)

                    # Draw each of the movement directions. The lines drawn are
straight, but the actual movements are curved.
                    cv2.line(map, (int(newNode.Node_x), int(newNode.Node_y)),
(int(current_node.Node_x), int(current_node.Node_y)),(155,155,155),1)

    # Release video and alert the user that no path was found.
    video_out.release()
    print("Path not found!")
    return map

# get_final_path backtracks the position to find the path.
# It also creates a file, path_rpms.txt which stores the commands
# that the turtlebot will need to follow to recreate the path.
def get_final_path(visited, end_node):

    # create a list of x and y positions and rpms.
    path_rpms = []
    path_xys = []
    current_x, current_y, current_theta = end_node.Node_x, end_node.Node_y,
end_node.Node_theta

    # Node key is used for indexing to get nodes.
    node_key = (int(round(current_y)), int(round(current_x)),
angle_to_index(current_theta))

    while node_key in visited:  # Ensure the node exists in visited.
        # Get the motors speeds used to get to this node.
        rpmL = visited[node_key].rpmL
        rpmR = visited[node_key].rpmR
        path_rpms.append((rpmL, rpmR))

        # Add the current x and y.
        path_xys.append((current_x, current_y))

        # Get the current parent's positon.
        parent_x = visited[node_key].Parent_Node_x
        parent_y = visited[node_key].Parent_Node_y
        parent_theta = visited[node_key].Parent_Node_theta

        # Stop when we reach the starting node.
        if (current_x, current_y, current_theta) == (parent_x, parent_y,
parent_theta):
            break

        # Update for the next iteration.
        current_x, current_y, current_theta = parent_x, parent_y, parent_theta
        node_key = (int(round(current_y)), int(round(current_x)),
angle_to_index(current_theta))

    path_rpms.reverse()  # Reverse to get the correct order.
    path_rpms.append((0,0)) # Add a robot stop command to the end.
    path_xys.reverse()

    # Write RPMs to a text file.
    with open("path_rpms.txt", "w") as f:
        for rpmL, rpmR in path_rpms:
            f.write(f"{rpmL},{rpmR}\n")

    return path_xys, path_rpms

def main(args=None):
    print("Program Start")
    print("Please enter the start and end coordinates.")
    print("Coordinates should be given as integers in units of cm from the
bottom left origin.")
    print("Image Width is 600 cm. Image Height is 250 cm.")

    # Generate and expand the obstacle map.
    obstacle_map = gen_obstacle_map()

    # Prompt the user for robot radius, clearance, and step size.
    robot_radius = int(np.ceil(RRADIUS*100)) # in cm / pixels
    clearance = int(np.ceil(float(input(f"Enter the clearance radius in mm:
")))/10)

    # Prompt user for motor RPMs.
    rpm1 = int(input(f"Enter the first radius/sec value for the motors (int) (1-
10 suggested): "))
    rpm2 = int(input(f"Enter the second radius/sec value for the motors (int)
(1-10 suggested): "))

    # Expand the obstacle space for the robot radius then for the clearance.
    expanded_obstacle_map, obs_map_gray = expand_obstacles(obstacle_map,
robot_radius)
    expanded_obstacle_map2, obs_map_gray =
expand_obstacles(expanded_obstacle_map, clearance)

    # Prompt the user for the start and end points for planning.
    start_x, start_y, start_theta = get_point(prompt="start",
obstacles=obs_map_gray)
    end_x, end_y = get_point(prompt="end", obstacles=obs_map_gray)

    # Correct the angles to align with OpenCV's top left origin system.
    start_theta = -start_theta % 360

    print("Planning Path...")

    # Apply A* search
    final_path_image = A_star_search(map=expanded_obstacle_map2,
obstacles=obs_map_gray, start=(start_x, start_y, start_theta), end=(end_x,
end_y), Revs=(rpm1, rpm2) )
    # total_time = time.time() - start_time
    print("Program Finished.")
    print("Click image and press keyboard to close program and final image.")

    # Show the solution.
    cv2.imshow("Map", final_path_image)
    cv2.waitKey(0)


if __name__ == "__main__":
    main()

PART
02: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!/usr/bin/env python3

import numpy as np
import cv2
import heapq
import copy
import time
import functools
import rclpy
from rclpy.node import Node as ROSNode
from geometry_msgs.msg import Twist
from pathlib import Path

import sys

HEIGHT = 300 # cm
WIDTH = 600 # cm

## turtle 3 wafflepi
# Wheel Radius (R): 33 mm
# Robot Radius (r): 220 mm
# Wheel Distance (L): 287 mm

WRADIUS = .033
RRADIUS = .22
WDIS = .287

# Timer decorator to measure execution time of functions
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()  # Start time
        result = func(*args, **kwargs)  # Execute the wrapped function
        end_time = time.perf_counter()  # End time
        run_time = end_time - start_time  # Calculate runtime
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return result  # Return the result of the wrapped function
    return wrapper

class Node:
    def __init__(self, Node_Cost, Node_Cost_est, Node_x, Node_y, Node_theta,
Parent_Node_x, Parent_Node_y, Parent_Node_theta, rpmL, rpmR):
        self.Node_Cost = Node_Cost # The cost to reach this node.
        self.Node_Cost_est = Node_Cost_est # The estimated cost to the goal.
        self.Node_x = Node_x # The node's x location.
        self.Node_y = Node_y # The node's y location.
        self.Node_theta = Node_theta # The node's angle.
        self.Parent_Node_x = Parent_Node_x # The node's parent's x location.
        self.Parent_Node_y = Parent_Node_y # The node's parent's y location.
        self.Parent_Node_theta = Parent_Node_theta # The node's parent's angle.
        self.rpmL = rpmL # The speed of the left motor to reach this point.
rads/sec
        self.rpmR = rpmR # The speed of the right motor to reach this point.
rads/sec

    # This method allows the heapq module to compare Node objects by their cost
when sorting.
    # This ensures that the node with the smallest cost plus heuristic is popped
first.
    # The heuristic used is Euclidean distance.
    def __lt__(self, other):
        return self.Node_Cost + self.Node_Cost_est < other.Node_Cost +
other.Node_Cost_est


# create_new_node creates a new node from a given node and a set of wheel
velocities.
# given_Node is the node we are travelling from. revs = (left wheel speed, right
wheel speed)
# which we use to determine how the robot is moving. goal_x and goal_y are the
end goal points
# for the path planner. We use this to calculate the heuristic to the goal.
```

```
obstacles is the image
# map of where all the obstacles are located. It is gray scale and equals 255
whenever an
# obstacle is present. We use this to determine if the motion from our start
point to our final
# point is obstacle free. The function returns the newNode generated from the
motion along with
# whether the path is obstacle free. If there are no obstacles encountered
during the motion, then
# valid_path is True.
def create_new_node(given_Node, revs, goal_x, goal_y, obstacles):

    # valid_curve takes in the entire array of x and y points. I then compares
all points
    # to the obstacle map to see if any lie out of the bounds of the map or in
an obstacle.
    # It returns true if all points in the path are obstacle free and in the
map.
    def valid_curve(x_arr, y_arr, map_shape, obstacles):
        # Round each point to nearest integer and convert to an int.
        x_arr = np.round(x_arr).astype(int)
        y_arr = np.round(y_arr).astype(int)

        # Check that all points are within the bounds of the map.
        in_bounds = (0 <= x_arr) & (x_arr < map_shape[1]) & (0 <= y_arr) &
(y_arr < map_shape[0])

        # Clip indices to avoid indexing errors.
        x_arr_clipped = np.clip(x_arr, 0, map_shape[1] - 1)
        y_arr_clipped = np.clip(y_arr, 0, map_shape[0] - 1)

        # Check that all points within the map are not obstacles.
        obstacle_free = obstacles[y_arr_clipped, x_arr_clipped] == 0

        # Rreturn True if all points are valid/obstacle free.
        return np.all(in_bounds & obstacle_free)

    # Get the left and right wheel velocities in rads/sec.
    UL, UR = revs

    r = WRADIUS  # Wheel radius
    L = WDIS  # Distance between wheels

    # Get the starting X, Y, and theta position.
    Xi = given_Node.Node_x
    Yi = given_Node.Node_y
    Thetai = given_Node.Node_theta

    # Time array. Get time steps of .1 seconds up until 3 seconds.
    t = np.linspace(0, 3, 100)
    dt = t[1] - t[0]

    # Angular velocity.
    # Note: Thetan = np.deg2rad(Thetai) + w * t was changed to Thetan =
np.deg2rad(Thetai) - w * t
    # because the sign convention was incorrect by observation.
    w = (r / L) * (UR - UL)
    Thetan = np.deg2rad(Thetai) - w * t

    # Linear velocity.
    v = 0.5 * r * (UL + UR)

    # Compute deltas.
    dx = v * np.cos(Thetan) * dt
    dy = v * np.sin(Thetan) * dt

    # Integrate position.
    x_points = np.cumsum(dx)*100 + Xi
    y_points = np.cumsum(dy)*100 + Yi

    # Total distance traveled.
    total_distance = np.sum(np.hypot(np.diff(x_points), np.diff(y_points)))

    # Final orientation.
    final_theta = np.rad2deg(Thetan[-1]) % 360

    # Get the parent.
    parent_x = given_Node.Node_x
    parent_y = given_Node.Node_y
    parent_theta = given_Node.Node_theta

    # Increment the cost from the movement.
    cost = given_Node.Node_Cost + total_distance

    # Get the final position and orientation.
    x_pos = x_points[-1]
    y_pos = y_points[-1]
    theta_pos = final_theta

    # Get the estimated cost to the goal.
    cost_est = np.sqrt((given_Node.Node_x - goal_x)**2 + (given_Node.Node_y -
goal_y)**2)

    # Generate the new node.
    newNode =
Node(cost,cost_est,x_pos,y_pos,theta_pos,parent_x,parent_y,parent_theta,
rpmL=UL, rpmR=UR)

    # Validate entire path.
    map_shape = obstacles.shape
    valid_path = valid_curve(x_points, y_points, map_shape, obstacles) # True or
False

    # Return the new node and if it is valid (obstacle free movement).
    return newNode, valid_path

# Convert the angle given, in degrees, to an index from 0-7.
def angle_to_index(angle):
    # Normalize angle to the range [0, 360).
    angle = angle % 360

    # Compute the index by dividing the angle by 45.
    return int(angle // 45)

def gen_obstacle_map():
    # Set the height and width of the image in pixels.
    height = HEIGHT
    width = WIDTH
    # Create blank canvas.
    obstacle_map = np.zeros((height,width,3), dtype=np.uint8 )

    # Define polygons for rectangle obstacles.
    def l_obstacle1(x,y):
        return (100 <= x <= 110) and (0 <= y <= 200)

    def l_obstacle2(x,y):
        return (210 <= x <= 220) and (100 <= y <= 300)
    def l_obstacle3(x,y):
        return (320 <= x <= 330) and (0 <= y <= 100)

    def l_obstacle4(x,y):
        return (320 <= x <= 330) and (200 <= y <= 300)

    def l_obstacle5(x,y):
        return (430 <= x <= 440) and (0 <= y <= 200)

    # For every pixel in the image, check if it is within the bounds of any
obstacle.
    # If it is, set it's color to red.
    for y in range(height):
        for x in range(width):
            if (l_obstacle1(x, y) or l_obstacle2(x,y) or l_obstacle3(x,y) or
l_obstacle4(x,y)
                or l_obstacle5(x,y)):
                obstacle_map[y, x] = (0, 0, 255)


    # The math used assumed the origin was in the bottom left.
    # The image must be vertically flipped to satisy cv2 convention.
    return np.flipud(obstacle_map)

# expand_obstacles takes the obstacle map given by gen_obstacle_map as an image,
along with
# the scale factor sf, and generates two images. The first output_image, is a
BGR image
# to draw on used for visual display only. expanded_mask is a grayscale image
with white
# pixels as either obstacles or clearance space around obstacles. This function
will take
# the given obstacle image and apply a specified radius circular kernel to the
image. This ensures
# an accurate clearance around every obstacle.
def expand_obstacles(image, radius):

    # Convert image to HSV.
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define color mask for red and create grayscale image.
    lower_red = np.array([0, 200, 200])
    upper_red = np.array([25, 255, 255])
    obstacle_mask = cv2.inRange(hsv, lower_red, upper_red)

    # Create circular structuring element for expansion
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2 * radius + 1, 2 *
radius + 1))
    # Apply kernel to get dilation around all elements.
    expanded_mask = cv2.dilate(obstacle_mask, kernel, iterations=1)

    # Apply dilation to all of the borders.
    h, w = expanded_mask.shape
    expanded_mask[:radius+1, :] = 255  # Top border
    expanded_mask[h-radius:, :] = 255  # Bottom border
    # expanded_mask[:, :radius+1] = 255  # Left border
    # expanded_mask[:, w-radius:] = 255  # Right border

    # Create the output image and apply color orange to all obstacle and
clearance
    # pixels.
    output_image = image.copy()
    output_image[np.where(expanded_mask == 255)] = [0, 165, 255]  # Color orange
    # Restore original red pixels. This creates an image with red obstacles,
    # and orange clearance zones.
    output_image[np.where(obstacle_mask == 255)] = [0, 0, 255]

    return output_image, expanded_mask

# Prompt the user for a point. prompt is text that specifies what
# type of point is to be given. prompt is solely used for terminal text output.
# obstacles is used to dictate the map's bounadires. This ensures points given
lie
# within the map's bounds.
# The function returns the user's points as integers. It also prompts the user
for
# an angle if the prompt is "start" and ensures that the angle is between
[0,360).

def get_point(prompt, obstacles):
    valid_input = False

    # Prompt the user until a valid input is given.
    while not valid_input:
        try:
            x = int(input(f"Enter the x-coordinate for {prompt} (int): "))
            y = int(input(f"Enter the y-coordinate for {prompt} (int): ")) + 150
        except ValueError:
            print("Invalid input. Please enter a numerical value.")
            continue

        if prompt == "start":
            # Ensure theta meets constraints.
            while True:
                try:
                    theta = int(input(f"Enter the theta-coordinate for {prompt}
(must be 0-359): "))
                    if 0 <= theta < 360:
                        break
                    else:
                        print("Invalid theta. It must be between 0 and 359.
Please try again.")
                except ValueError:
                    print("Invalid input. Please enter an integer value for
theta.")

        # Correct the y value to account for OpenCV having origin in top left.
        obstacle_y = obstacles.shape[0] - y

        # Validate position against obstacles
        if valid_move(x, obstacle_y, obstacles.shape, obstacles):
            valid_input = True
        else:
            print("Invalid Input. Within Obstacle. Please try again.")
    if prompt == "start":
        return int(x), int(y), int(theta)
    else:
        return int(x), int(y)

# valid_move checks if a given point lies within the map bounds and
# if it is located within an obstacle. If the point is in the image and NOT in
an obstacle,
# it returns True, meaning the position is valid/Free/open space.
def valid_move(x, y, map_shape, obstacles):
    return 0 <= x < map_shape[1] and 0 <= y < map_shape[0] and obstacles[int(y),
int(x)] == 0
```

```
# Check if we are at the goal position. This checks if the point is within a
circular region around
# the goal.
def goal_check(x, y, end_x, end_y):
    # Compute Euclidean distance to goal.
    dis = np.sqrt(((end_x - x))**2 + (end_y - y)**2)
    # Check if position is within 5 cms.
    if dis < 5:
        return True
    else:
        return False

# gen_node_key takes a node and uses its properties to generate a 'node_key'.
This is an index
# used for the closed and seen sets.
def gen_node_key(node):
    node_key = (int(round(node.Node_y)), int(round(node.Node_x)),
angle_to_index(node.Node_theta))
    return node_key

# A_star_search uses the A star method to explore a map and find the path from
start to end.
# map is the image to draw on. Obstacles is a grayscale image of 1 cm resolution
to path plan on.
# start consists of the desired starting x, y, and theta positions. end includes
the desired x, y
# ending positions. Revs is a list of eight wheel revolution combinations to
determine the action set
# of the robot.
@timer
def A_star_search(map, obstacles, start, end, Revs):

    # Convert y coordinates from origin bottom left (user input) to origin top
left (cv2 convention).
    height, width, _ = map.shape
    start_x, start_y, start_theta = start
    end_x, end_y = end
    start_y = height - start_y
    end_y = height - end_y
    start = (start_x, start_y, start_theta)
    end = (end_x, end_y)

    # Open video file to write path planning images to.
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video_filename = "A_star_Proj3_phase2_video.mp4"
    fps = 60
    video_out = cv2.VideoWriter(video_filename, fourcc, fps, (width, height))

    # Create the start node.
    start_node = Node(0, 0, start[0], start[1], start[2], start[0], start[1],
start[2], 0, 0)

    open_set = []  # Priority queue. Used to extract nodes with smallest cost
from forward search.
    heapq.heappush(open_set, start_node)

    # The path planning occurs on a positional resolution of 1 cm. This sets the
dimensions for our
    # sets to check if points are duplicates.
    height = int(height)
    width = int(width)

    # The seen set is how I track if a node has already been given a cost. It is
a boolean matrix that checks if
```

```
    # every possible position/orientation combination has been seen. True =
seen. seen nodes are in the open or closed set.
    seen = np.full((height, width, 8), False, dtype=bool)
    start_angle_index = angle_to_index(start_theta)
    seen[start_y, start_x, start_angle_index] = True

    # This is my seen set, but as a dictionary to store all node information.
    visited = {}
    visited[(start_y, start_x, start_angle_index)] = start_node

    # This is the closed set. It stores nodes that have been fully explored.
    closed_set = np.full((height, width, 8), False, dtype=bool)

    # Create a list of functions of all the types of moves we can execute.
    RPM1 = Revs[0]
    RPM2 = Revs[1]
    directions = [[0,RPM1],[RPM1,0],[RPM1,RPM1],[0,RPM2],[RPM2,0],[RPM2, RPM2],
[RPM1,RPM2],[RPM2,RPM1]]

    # Draw the start and end points as a magenta and cyan circle, respectively.
    cv2.circle(map, (start[0],start[1]), 5, (255, 0, 255), -1)
    cv2.circle(map, (end[0], end[1]), 5, (255, 255, 0), -1)

    # Used to store only every 100th frame to the video. Otherwise the video is
hours long.
    # Additionally, writing frames takes the most computation for every loop.
    video_frame_counter = 0

    # Continue to search while the open_set is not empty.
    while open_set:

        # Get the node with the smallest cost from the open_set.
        current_node = heapq.heappop(open_set)
        # Extract it's x and y position.
        current_x, current_y, current_theta = current_node.Node_x,
current_node.Node_y, current_node.Node_theta

        # Verify that this position is not in the closed set.
        # Skip this iteration if it is in the closed_set as the position
        # has already been fully explored. This is required because
        # there is no efficient implementation to updating nodes within a heapq.
        # As such, a node may be added to the heapq, then added again to the
heapq if
        # a better parent was found.
        if closed_set[int(current_y), int(current_x),
angle_to_index(current_theta)] == True:
            continue

        # Add the current node to the closed set.
        closed_set[int(current_y), int(current_x),
angle_to_index(current_theta)] = True

        # If the node currently popped is within the goal region, get the path.
        if goal_check(current_x, current_y, end_x, end_y):
            print("Goal Reached!")
            # Generate final path.
            path_xys, path_rpms = get_final_path(visited, current_node)

            print(path_xys)
            print(f"end_x: {end_x}")
            print(f"end_y: {end_y}")

            print(path_rpms)
            # For each pixel in the path, draw a magenta line
            for i in range(len(path_xys) - 1):
                x1, y1 = path_xys[i]
                x2, y2 = path_xys[i + 1]
                cv2.line(map, (int(x1), int(y1)), (int(x2), int(y2)), (255, 0,
255), 2)
                video_out.write(map)

            # Release the video file.
            video_out.release()
            # Terminate search and return the final map with the path and area
explored.
            return map

        # Increment the video_frame_counter and save a frame if it is the 100th
frame.
        video_frame_counter += 1
        if video_frame_counter == 100:
            # Redraw start and end circles.
            cv2.circle(map, (start_x, start_y), 5, (255, 0, 255), -1)
            cv2.circle(map, (end_x, end_y), 5, (255, 255, 0), -1)
            # Save current map state as a frame in the final video.
            video_out.write(map)
            # Reset the frame counter.
            video_frame_counter = 0

        # For the current node, apply each of the eight movement directions and
examine
        # the newNode generated from moving in each direction.
        for revs in directions:
            # Get newNode from current move.
            newNode, valid_path = create_new_node(current_node, revs, end_x,
end_y, obstacles)

            if valid_path: # Check that it isn't in an obstacle.

                # Get the node_key / index for this node.
                node_key = gen_node_key(newNode)

                if closed_set[node_key] == False: # Check that it is not in the
closed set.
                    if seen[node_key] == False: # Check that it isn't in the
open nor closed lists.

                        # Add it to the seen set.
                        seen[node_key] = True

                        # Add it to the visited set.
                        visited[node_key] = newNode
                        heapq.heappush(open_set, newNode)

                    # If the node is in the open list AND the new cost is
cheaper than the old cost to this node, rewrite it
                    # within visited and add the newNode to the open_set. The
old version will be safely skipped.
                    elif seen[node_key] == True:
                        if visited[node_key].Node_Cost > newNode.Node_Cost:
                            visited[node_key] = newNode
                            heapq.heappush(open_set, newNode)

                    # Draw each of the movement directions. The lines drawn are
straight, but the actual movements are curved.
                    cv2.line(map, (int(newNode.Node_x), int(newNode.Node_y)),
(int(current_node.Node_x), int(current_node.Node_y)),(155,155,155),1)
```

```
    # Release video and alert the user that no path was found.
    video_out.release()
    print("Path not found!")
    return map

# get_final_path backtracks the position to find the path.
# It also creates a file, path_rpms.txt which stores the commands
# that the turtlebot will need to follow to recreate the path.
def get_final_path(visited, end_node):

    # create a list of x and y positions and rpms.
    path_rpms = []
    path_xys = []
    current_x, current_y, current_theta = end_node.Node_x, end_node.Node_y, 
end_node.Node_theta

    # Node key is used for indexing to get nodes.
    node_key = (int(round(current_y)), int(round(current_x)), 
angle_to_index(current_theta))

    while node_key in visited:  # Ensure the node exists in visited.
        # Get the motors speeds used to get to this node.
        rpmL = visited[node_key].rpmL
        rpmR = visited[node_key].rpmR
        path_rpms.append((rpmL, rpmR))

        # Add the current x and y.
        path_xys.append((current_x, current_y))

        # Get the current parent's positon.
        parent_x = visited[node_key].Parent_Node_x
        parent_y = visited[node_key].Parent_Node_y
        parent_theta = visited[node_key].Parent_Node_theta

        # Stop when we reach the starting node.
        if (current_x, current_y, current_theta) == (parent_x, parent_y, 
parent_theta):
            break

        # Update for the next iteration.
        current_x, current_y, current_theta = parent_x, parent_y, parent_theta
        node_key = (int(round(current_y)), int(round(current_x)), 
angle_to_index(current_theta))

    path_rpms.reverse()  # Reverse to get the correct order.
    path_rpms.append((0,0)) # Add a robot stop command to the end.
    path_xys.reverse()

    # Write RPMs to a text file.
    with open("path_rpms.txt", "w") as f:
        for rpmL, rpmR in path_rpms:
            f.write(f"{rpmL},{rpmR}\n")

    return path_xys, path_rpms

# Create a ROS node that publishes to the turtlebot. It will command the 
turtlebot to follow
# the velocity profile set within the file path_rpms.txt.
class PathFollower(ROSNode):
    def __init__(self):
        # Create the node
        super().__init__('path_follower')
        # Create a publisher that publishes to control the turtlebot's velocity.
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 10)

        self.timer_period = 3.0  # How frquently commands should get updated.

        # Load RPM path
        self.rpm_path = self.load_rpm_path('path_rpms.txt')
        self.current_index = 0

        # Start publishing commands
        self.timer = self.create_timer(self.timer_period, self.publish_velocity)

    # Function that reads in the path_rpms.txt file.
    def load_rpm_path(self, filename):
        path = Path(filename)
        if not path.exists():
            self.get_logger().error(f"File {filename} not found.")
            return []
        with open(filename, 'r') as file:
            lines = file.readlines()
            rpm_values = [tuple(map(float, line.strip().split(','))) for line in
lines if line.strip()]
        return rpm_values

    def rpm_to_mps(self, rpm):
        # Convert rads/sec to m/s.
        WHEEL_RADIUS = 0.033  # meters
        return (rpm * WHEEL_RADIUS)

    # Publishes velocity commands to the turtlebot.
    def publish_velocity(self):
        # If all the commands have been published, finish.
        if self.current_index >= len(self.rpm_path):
            self.get_logger().info('Path complete.')
            self.timer.cancel()
            return

        # Get the next command.
        left_rpm, right_rpm = self.rpm_path[self.current_index]
        self.get_logger().info(f"Publishing RPMs - Left: {left_rpm}, Right:
{right_rpm}")

        # Convert from rads/sec to m/s.
        left_mps = self.rpm_to_mps(left_rpm)
        right_mps = self.rpm_to_mps(right_rpm)

        # Robot base size. Distance between wheels.
        WHEEL_BASE = 0.287  # meters

        # cmd_vel topic wants linear and angular velocity. Convert to desired
format
        # from individual wheel velocities.
        linear_x = (right_mps + left_mps) / 2.0
        angular_z = (right_mps - left_mps) / WHEEL_BASE

        # Create Twist message. Set message data for publishing.
        twist = Twist()
        twist.linear.x = linear_x
        twist.angular.z = angular_z

        # Publish the message to the turtlebot.
        self.publisher_.publish(twist)
        self.current_index += 1
```

```
def main(args=None):
    print("Program Start")
    print("Please enter the start and end coordinates.")
    print("Coordinates should be given as integers in units of cm from the
bottom left origin.")
    print("Image Width is 600 cm. Image Height is 250 cm.")

    rclpy.init(args=args)

    # Generate and expand the obstacle map.
    obstacle_map = gen_obstacle_map()

    # Prompt the user for robot radius, clearance, and step size.
    robot_radius = int(np.ceil(RRADIUS*100)) # in cm / pixels
    clearance = int(np.ceil(float(input(f"Enter the clearance radius in mm:
")))/10)

    # Prompt user for motor speeds.
    rpm1 = int(input(f"Enter the first radius/sec value for the motors (int) (1-
10 suggested): "))
    rpm2 = int(input(f"Enter the second radius/sec value for the motors (int)
(1-10 suggested): "))

    # Expand the obstacle space for the robot radius then for the clearance.
    expanded_obstacle_map, obs_map_gray = expand_obstacles(obstacle_map,
robot_radius)
    expanded_obstacle_map2, obs_map_gray =
expand_obstacles(expanded_obstacle_map, clearance)

    # Prompt the user for the start and end points for planning.
    start_x, start_y, start_theta = get_point(prompt="start",
obstacles=obs_map_gray)
    end_x, end_y = get_point(prompt="end", obstacles=obs_map_gray)

    # Correct the angles to align with OpenCV's top left origin system.
    start_theta = -start_theta % 360

    print("Planning Path...")

    # Apply A* search
    final_path_image = A_star_search(map=expanded_obstacle_map2,
obstacles=obs_map_gray, start=(start_x, start_y, start_theta), end=(end_x,
end_y), Revs=(rpm1, rpm2) )
    print("Program Finished.")

    # Generate the node, run it, and shutdown when finished.
    node = PathFollower()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

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