```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Created on Tue Mar 21 16:05:08 2023

@author: caitlin.p.conn

"""

# Required imported libraries
# Additional packages were installed, tried to capture requirements in readme
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry

from collections import OrderedDict
from datetime import datetime

import matplotlib.pyplot as plt
import numpy as np
import imutils
import rospy
import math
import time
import cv2

from tf.transformations import euler_from_quaternion, quaternion_from_euler

##########################################################################################
# Visualization/Plotting Helper Functions

# Function creates the map grid image with the appriopriate obstacle and freespace boundaries
# Resource: quora.com/How-do-you-find-the-distance-between-the-edges-and-the-center-of-a-regula
def create_map_grid(clearance_color, obstacle_space_color, free_space_color, robot_clearance, r

    # Define map grid shape
    map_grid = np.ones((map_height,map_width,3), dtype = np.uint8)

    # Set obstacle color
    obstacle_color = obstacle_space_color

    # Define total clearance value in pixels using user robot radius and robot clearance inputs
    c = robot_clearance + robot_radius

    #############################################################
    # Change image pixels to reflect map boundaries

    for y in range(map_height):
        for x in range(map_width):

            ##################################################################
            # Display rectangles

            # Plot left rectange clearance
            if x >= ((0.5+1)*100) - c and x < ((0.5+1+0.15)*100)+ c and y >= ((0.75)*100) - c a
                map_grid[y,x] = clearance_color

            # Plot left rectange obstacle space
            if x >= ((0.5+1)*100) and x <= ((0.5+1+0.15)*100) and y >= ((0.75)*100) and y <= ((
                map_grid[y,x] = obstacle_color

            # Plot right rectange clearance
            if x >= ((0.5+1+1)*100) - c and x < ((0.5+1+1+0.15)*100)  + c and y >= 0 - c and y
                map_grid[y,x] = clearance_color
```

```python
        # Plot right rectange obstacle space
        if x >= ((0.5+1+1)*100) and x <= ((0.5+1+1+0.15)*100) and y >= 0 and y <= ((1.25)*1
            map_grid[y,x] = obstacle_color

        ###################################################################

        # Plot horizontal walls bloated by c
        if (x >= 0 and x < map_width and y >= 0 and y < c) or (x >= 0 and x < map_width and
            map_grid[y,x] = clearance_color

        # Plot vertical walls bloated by c
        if (x >= 0 and x < c and y >= 0 and y < map_height) or (x >= map_width - c and x <
            map_grid[y,x] = clearance_color


    ###################################################################
    # Display circle
    x, y = np.meshgrid(np.arange(map_grid.shape[1]), np.arange(map_grid.shape[0]))

    circle_center_x = (0.5+1+1+1.5)*100
    circle_center_y = (1.1)*100
    circle_radius = (0.5)*100 + c

    dist_point_to_circle_center = np.sqrt((x - circle_center_x) ** 2 + (y - circle_center_y) **

    # Use boolean mask to extract pixels in circle
    mask = dist_point_to_circle_center <= circle_radius
    map_grid[mask] = clearance_color

    circle_radius = (0.5)*100

    # Use boolean mask to extract pixels in circle
    mask = dist_point_to_circle_center <= circle_radius
    map_grid[mask] = obstacle_color

    ###################################################################

    # REMAP COORDINATES


    ###################################################################

    # Plot the resized map grid with the obstacle and free space
    plt.figure()
    plt.title('Original Map Grid')
    plt.imshow(map_grid.astype(np.uint8), origin="lower")
    plt.show()

    return map_grid, map_height, map_width

#####################################################################################################
# Function checks if node is in the defined obstacle space
def check_node_in_obstacle_space(child_node_x, child_node_y, obstacle_matrix):
    # print("Obstacle DB: ", obstacle_matrix[int(child_node_y)][int(child_node_x)] == -1)
    # print()
    return obstacle_matrix[int(child_node_y)][int(child_node_x)] == -1

# Function checks if node is within map_grid bounds
def check_node_in_map(child_node_x, child_node_y, map_height, map_width):
    # print("Map DB: ", 0 <= child_node_x < map_width and 0 <= child_node_y < map_height)
    # print()
    return 0 <= child_node_x < map_width and 0 <= child_node_y < map_height
```

```
#########################################################################################

# Function handles negative angles and angles greater than 360 degrees
def handle_theta_angle(input_angle):
    return input_angle % 360.0

# Function determines the validity of the action to produce the child node and checks if the re
def generate_child_node(obstacle_matrix, map_boundary_matrix, parent_node, goal_node_coord, act

    valid_move = False # boolean truth value of valid swap
    is_node_obstacle = False # boolean to check if node is in obstacle space

    child_node_x, child_node_y, child_theta, cost_of_action, angle_to_add = 0, 0, 0, 0, 0

    parent_node_x = parent_node[0][0] # parent x coordinate
    parent_node_y = parent_node[0][1] # parent y coordinate
    parent_theta = parent_node[1][4] # parent theta value
    parent_cost_to_come = parent_node[1][5] # parent cost to come (not total cost)

    #####################################################################################

    pt_list = []
    t = 0 # time 0 seconds
    r = robot_wheel_radius
    L = wheel_distance
    dt = 0.1 # increment of seconds
    Thetan = 3.14 * parent_theta / 180# convert angle to radians

    # Action logic using dictionary
    action_dict = {1: [0, RPM1], 2: [RPM1, 0], 3: [RPM1, RPM1], 4: [0, RPM2], 5: [RPM2, 0], 6:
    child_rpm_actions = action_dict.get(action, 0)
    RPM1_val, RPM2_val = child_rpm_actions

    # Convert between revolutions per minute and radians per second to convert from RPM values
    # directly specifies the two angular wheel velocities (e.g., in radians per second).
    UL = RPM1_val * 2 * math.pi / 60.0
    UR = RPM2_val * 2 * math.pi / 60.0

    D = 0
    Xn = parent_node_x
    Yn = parent_node_y
    while t < 0.5: #t < 1:
        t = round(t + dt, 2)
        X = Xn + (0.5 * r * (UL + UR) * math.cos(Thetan) * dt)
        Y = Yn + (0.5 * r * (UL + UR) * math.sin(Thetan) * dt)
        Thetan += (r / L) * (UR - UL) * dt
        D += math.sqrt(math.pow((X-Xn),2) + math.pow((Y-Yn),2))

        xdot = 0.5 * r * (UL + UR) * math.cos(Thetan)
        ydot = 0.5 * r * (UL + UR) * math.sin(Thetan)
        thetadot = (r / L) * (UR - UL)

        Xn, Yn = X,Y

        pt_list.append([Xn, Yn, Thetan, xdot, ydot, thetadot])


    # Set child angle based on the action value
    angle_to_add = 180 * (Thetan) / 3.14 # convert back to degrees
    child_theta = handle_theta_angle(child_theta)
    child_theta = angle_to_add

    # Set child coordinates
    child_node_x = X
```

```python
        child_node_y = Y

        ########################################################################################

        # Check if node is within map bounds and if it is in the obstacle space for later computati
        is_node_in_map = check_node_in_map(child_node_x, child_node_y, map_height, map_width)
        if is_node_in_map:
            is_node_obstacle = check_node_in_obstacle_space(child_node_x, child_node_y, obstacle_ma
        else:
            is_node_obstacle = False
        valid_move = is_node_in_map and not is_node_obstacle

        ########### COST LOGIC ###################

        # Compute child node's cost to come value -> CostToCome(x') = CostToCome(x) + L(x,u)
        cost_of_action = D
        # child_cost_to_come = round(cost_of_action + parent_cost_to_come, 1)
        child_cost_to_come = cost_of_action + parent_cost_to_come


        pta = (child_node_x, child_node_y)
        ptb = (goal_node_coord[0], goal_node_coord[1])

        # Euclidean Distance Heuristic
        cost_to_go = np.linalg.norm(np.array(pta)-np.array(ptb))

        weight = 2
        # Can change this value to further bias the results toward the goal node ex. enter 4
        total_cost = child_cost_to_come + (weight*cost_to_go)

        ###########################################

        # returned node is the resulting child node of the requested action
        return pt_list, child_cost_to_come, total_cost, valid_move, child_node_x, child_node_y, chi

    ##########################################################################################
    # Function takes the visited queue as an input and computes the optimal path from the start to

    def compute_optimal_path(visited_queue, initial_node_coord, goal_node_coord, closest_node_to_go
        path_list = [] # list to store coordinates of optimal path


        closest_node_to_goal = (closest_node_to_goal_x, closest_node_to_goal_y)

        first_parent_coord = visited_queue[closest_node_to_goal][3] # set first parent to search fo
        curr_elem_x = closest_node_to_goal[0] # get x coordinate of the goal node, first node added
        curr_elem_y = closest_node_to_goal[1] # get y coordinate of the goal node, first node added

        path_list.append(closest_node_to_goal) # add goal node to the path list
        parent_coord = first_parent_coord # set variavle equal to current node's coordinates

        # Continue while loop logic until the initial node is reached
        while(not((curr_elem_x == initial_node_coord[0]) and (curr_elem_y == initial_node_coord[1])
            for visited_elem in visited_queue: # loop through each node in the visited queue; visit
                curr_elem_x = visited_elem[0] # current node's x coordinate
                curr_elem_y = visited_elem[1] # current node's y coordinate
                curr_coord = (curr_elem_x, curr_elem_y) # store coordinate as variable
                if curr_coord == parent_coord: # check if the current node is the node being search
                    path_list.append(visited_elem) # add the current element to the path list
                    parent_coord = visited_queue[visited_elem][3] # search for the current node's p
                    break

        # Debug Statements
        # for elem in visited_queue:
```

```python
    #     print("Visited Queue Current Element: ", elem)
    # print()

    # for p in path_list:
    #     print("Path List Current Element: ", p)
    # print()

    path_list = np.flipud(path_list)

    return path_list

###################################################################################################

# Function keeps track of obstacle and free space boundaries of the map
# Matrix is only created or edited once at the beginning of the code file
def map_obstacle_freespace_matrix(map_grid, map_height, map_width):
    color_channel = map_grid[..., 2].astype(float)  # Convert color channel to float

    map_boundary_matrix = np.full_like(color_channel, -1, dtype=float)

    map_boundary_matrix = np.where(color_channel == 1, np.inf, map_boundary_matrix)

    return map_boundary_matrix

###################################################################################################
# Function returns node with the lowest cost to come in the visited queue

def get_node_lowest_total_cost(open_list):

    node, min_total_cost = min(open_list.items(), key=lambda x: x[1][0])

    return node

###################################################################################################
# Function computes final logic if the goal node has been found, goal node is added to the visi

def goal_node_found(visited_nodes, goal_found, visited_queue, child_node_x_valid, child_node_y_
    goal_found = True

    node_idx = node_idx + 1

    child_node = ((child_node_x_valid, child_node_y_valid),(total_cost, node_idx, curr_parent_i

    visited_queue[(child_node_x_valid, child_node_y_valid)] = (total_cost, node_idx, curr_parer
    visited_nodes.add((child_node_x_valid, child_node_y_valid))  # Add the node to the visited

    closest_node_to_goal_x = child_node_x_valid
    closest_node_to_goal_y = child_node_y_valid

    print("Last Child Node (Goal Node): \n", child_node)
    print()
    print("Problem solved, now backtrack to find optimal path!")
    print()
    print("_________________________________________________________________________________
    print()

    return visited_queue, goal_found, closest_node_to_goal_x, closest_node_to_goal_y

###################################################################################################
# Function gets the user defined input to define the initial and goal nodes

def get_user_input(map_width, map_height, check_node_in_obstacle_space, obstacle_matrix):

    # Get user defined initial node
```

```
    while True:
        x_initial = eval(input("Enter start node's x coordinate. x coordinate can range from 0
        y_initial = eval(input("Enter start node's y coordinate. y coordinate can range from 0
        th_initial = eval(input("Enter the orientation of the robot at the start point in degre

        if not(0 <= x_initial <= map_width - 1) or (not(0 <= y_initial <= map_height - 1)):
            print("Re-enter initial coordinates, coordinates not within bounds.")
            print()

        else:
            print("Start node x-coordinate:", x_initial)
            print("Start node y-coordinate:", y_initial)
            print()

            is_initial_obstacle = check_node_in_obstacle_space(x_initial, y_initial, obstacle_m
            if (is_initial_obstacle == True):
                print("Re-enter initial node, coordinates are within bounds but are not within
                print()
                continue

            if (th_initial % 30) != 0:
                print("Re-enter initial theta, angle not multiple of 30. (k * 30), i.e. {.., -6
                print()
                continue
            else:
                break

    # Get user defined goal node
    while True:
        x_goal = eval(input("Enter goal node's x coordinate. x coordinate can range from 0 to '
        y_goal = eval(input("Enter goal node's y coordinate. y coordinate can range from 0 to '

        if not(0 <= x_goal <= map_width - 1) or (not(0 <= y_goal <= map_height - 1)):
            print("Re-enter goal coordinates, coordinates not within bounds.")
            print()

        else:
            print("Goal node x-coordinate:", x_goal)
            print("Goal node y-coordinate:", y_goal)
            print()

            is_goal_obstacle = check_node_in_obstacle_space(x_goal, y_goal, obstacle_matrix)
            if (is_goal_obstacle == True):
                print("Re-enter goal node, coordinates are within bounds but are not within fre
                print()
                continue
            else:
                break

    # Make sure input initial and goal angles are non-negative and are <= 360 degrees
    th_initial = handle_theta_angle(th_initial)

    print("___________________________________________________________________________________
    print()

    return x_initial, y_initial, x_goal, y_goal, th_initial

# Function gets the user defined input to define robot radius and clearance
def get_user_robot_input():

    # Get user defined robot clearance
    while True:
        robot_clearance = eval(input("Enter robot clearance between 0 and 15" + ": "))
```

```python
        if not(0 <= robot_clearance <= 15):
            print("Re-enter robot clearance as value between 0 and 0.2")
            print()
            continue

        else:
            print("Robot Clearance:", robot_clearance)
            print()
            break

        # print("Robot Clearance:", robot_clearance)
        # print()
        # break

    # Get user defined robot RPM values: Revolutions per Minute
    # https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#data-of-turtlebot3-waf
    # https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#specifications
    # 57 [rev/min] (at 11.1 [V])
    # 61 [rev/min] (at 12.0 [V])
    while True:
        RPM1 = eval(input("Enter robot Revolutions per Minute (RPM1) integer value between 10 a

        if not(10 <= RPM1 <= 55):
            print("Re-enter robot RPM1 as integer value between 10 and 55")
            print()
            continue

        else:
            print("Robot RPM1:", RPM1)
            print()
            break

    while True:
        RPM2 = eval(input("Enter robot Revolutions per Minute (RPM2) integer value between 10 a

        if not(10 <= RPM1 <= 55):
            print("Re-enter robot RPM2 as integer value between 10 and 55")
            print()
            continue

        else:
            print("Robot RPM2:", RPM2)
            print()
            break

    return robot_clearance, RPM1, RPM2

###############################################################################################

# Function takes numerical input and rounds it to the nearest half
def round_num_nearest_half(num, thresh_val):
    return (2*(round(num / thresh_val) * thresh_val))

def is_node_in_visited_queue(child_node_x_valid, child_node_y_valid, visited_matrix):
    thresh = 0.5
    # Compute the indices of the visited region for the node
    i = int(round_num_nearest_half(child_node_x_valid, thresh))
    j = int(round_num_nearest_half(child_node_y_valid, thresh))

    # print(i)
    # print(j)
    # print(child_node_x_valid)
    # print(child_node_y_valid)
```

```
    if (0 <= i < 1200) and (0 <= j < 400):
        if visited_matrix[i][j] == 0:
            visited_matrix[i][j] = 1
            return False
        else:
            return True

    return True

################################################################################################

# Function uses backtracking logic to find the traversal pathway from the initial node to goal
# Function that calls subfunctions to perform search operations
# Resource: https://numpy.org/doc/stable/reference/generated/numpy.fliplr.html
# Must use flipud function to ensure using a forward search strategy!!

def astar_approach_alg(obstacle_matrix, map_boundary_matrix, initial_node_coord, goal_node_coor
    visited_matrix = np.zeros((1200, 400))

    explored_node_color = (255, 255, 255) # White arrows

    closest_node_to_goal_x = 0
    closest_node_to_goal_y = 0

    ##########################################################################
    robot_wheel_radius = 33*0.1 # mm to cm
    robot_radius = 105*0.1 # mm to cm
    wheel_distance = 160*0.1 # mm to cm

    line_points = {}
    visited_nodes = set()
    ##########################################################################

    # Thresholds defined per problem statement
    euclid_dist_threshold = 5

    fig, ax = plt.subplots() # keeps track of figure and axis for map grid image

    curr_parent_idx = 0 # Parent index
    node_idx = 1 # Current node index

    debug_counter = 0
    # debug_counter2 = 0

    # Create empty data structures
    visited_queue = {} # explored/visited/closed, valid nodes
    visited_queue = OrderedDict(visited_queue)
    open_dict = {} # keeps track of the node queue to be processed

    show_grid = True # Debug boolean
    goal_found = False # When true, stop search

    ##############################################################################
    # Add initial node to the open node dictionary, initial node has no parent
    open_dict[initial_node_coord] = [0, node_idx, None, initial_node_coord, th_initial, 0]

    # Check if the initial node is the goal node, if so stop search
    pt1 = (initial_node_coord[0], initial_node_coord[1])
    pt2 = (goal_node_coord[0], goal_node_coord[1])
    euclidean_dist = np.linalg.norm(np.array(pt1)-np.array(pt2))

    if (initial_node_coord[0] == goal_node_coord[0] and initial_node_coord[1] == goal_node_coor

        curr_node_coord = (initial_node_coord[0], initial_node_coord[1])
```

```
    visited_queue, goal_found, closest_node_to_goal_x, closest_node_to_goal_y = goal_node_f
    return [], visited_queue, goal_found, fig, ax, closest_node_to_goal_x, closest_node_to_

##############################################################################
print("Main code execution has started...")
print()

# Process next node in the open dictionary with the lowest cost to come
while (len(open_dict) != 0): # Stop search when node queue is empty

    debug_counter = debug_counter + 1 # Debug variable

    curr_node = get_node_lowest_total_cost(open_dict)

    curr_node_x = curr_node[0]
    curr_node_y = curr_node[1]
    curr_coord = (curr_node_x,curr_node_y) # Returns current node's (x,y) coordinates
    curr_node_list = open_dict.pop(curr_coord) # Returns (cost_to_come, current_node_idx, p
    curr_node_coord = (curr_node_x, curr_node_y)

    curr_node = (curr_coord, curr_node_list) # Creates a tuple, first element is the node's

    visited_queue[curr_node_coord] = curr_node_list
    visited_nodes.add(curr_node_coord)  # Add the node to the visited set


    #######################################################################
    # Debug statements

    debug_runs = 5000

    if show_grid == True and debug_counter % debug_runs == 0:
        print("Debug Counter: ", debug_counter)
        print("Current Parent Node:")
        print(curr_node)
        print()

    #######################################################################

    # Evaluate children
    curr_parent_idx = curr_node_list[1] # Update parent node index
    i = 1 # Start with first child/move/action

    while i < 9: # Iterate for 8 times -> 8 moves

        # Generate child node
        pt_list, child_cost_to_come, total_cost, valid_move, child_node_x_valid, child_node

        ###################################################################

        if valid_move == False:
            i = i + 1 # Update action variable to evaluate the next move
            continue

        pt11 = [child_node_x_valid, child_node_y_valid]
        pt22 = goal_node_coord
        euclidean_dist_b4 = np.linalg.norm(np.array(pt11)-np.array(pt22))
        is_visited_check = is_node_in_visited_queue(child_node_x_valid, child_node_y_valid,
        if is_visited_check:
            i = i + 1
            continue

        # Check if child node is in open dictionary
        is_in_open =  (child_node_x_valid, child_node_y_valid) in set(open_dict)
```

```
            ##################################################################

            if valid_move == True: # Child node is valid but has not been explored yet

                # Check if the initial node is the goal node, if so stop search
                pt1 = [child_node_x_valid, child_node_y_valid]
                pt2 = goal_node_coord
                euclidean_dist = np.linalg.norm(np.array(pt1)-np.array(pt2))

                # Check if current child node is goal node
                if (euclidean_dist <= euclid_dist_threshold): # Child node equals goal node
                    visited_queue, goal_found, closest_node_to_goal_x, closest_node_to_goal_y =
                    line_points[(child_node_x_valid,child_node_y_valid)] = pt_list

                    return line_points, visited_queue, goal_found, fig, ax, closest_node_to_goa

                else: # Goal node/state not found yet

                    if (is_in_open == False): # New child, child has not been expored and is no
                        node_idx = node_idx + 1 # Create new child index
                        open_dict[(child_node_x_valid, child_node_y_valid)]=(total_cost, node_i
                        line_points[(child_node_x_valid,child_node_y_valid)] = pt_list

                    elif (is_in_open == True): # Child has not been explored but is in open dic

                        # child_total_cost_stored = open_dict[(x_replace, y_replace)][0]
                        child_total_cost_stored = open_dict[(child_node_x_valid, child_node_y_v


                        # Update node
                        if total_cost < child_total_cost_stored:
                            total_cost_new = total_cost
                            existing_child_idx = open_dict[(child_node_x_valid, child_node_y_va

                            child_node_list = (total_cost_new, existing_child_idx, curr_parent_
                            open_dict[(child_node_x_valid, child_node_y_valid)] = child_node_li

                            line_points[(child_node_x_valid,child_node_y_valid)] = pt_list


            i = i + 1 # Update action variable to evaluate the next move

            first_parent = visited_queue[curr_node_coord][3]
            first_parent_x = first_parent[0]
            first_parent_y = first_parent[1]

    # End of outer while loop
    print("Goal Found Boolean: ", goal_found)
    print()

    return line_points, visited_queue, goal_found, fig, ax, closest_node_to_goal_x, closest_noc

##########################################################################################
def handle_root_node(root_x, root_y, child_x, child_y, robot_wheel_radius, wheel_distance, RPM1
    t = 0 # time 0 seconds
    r = robot_wheel_radius
    L = wheel_distance
    dt = 0.1 # increment of seconds
    Thetan = 3.14 * 0 / 180# convert angle to radians
    pt_list = []

    # Convert between revolutions per minute and radians per second to convert from RPM values
    # directly specifies the two angular wheel velocities (e.g., in radians per second).
```

```
    UL = RPM1_val * 2 * math.pi / 60.0
    UR = RPM2_val * 2 * math.pi / 60.0

    # Xi, Yi,Thetai: Input point's coordinates
    # Xs, Ys: Start point coordinates for plot function
    # Xn, Yn, Thetan: End point coordintes
    D = 0
    Xn = root_x
    Yn = root_y
    while t < 0.5:#t < 1:
        t = round(t + dt, 2)
        Xn += 0.5 * r * (UL + UR) * math.cos(Thetan) * dt
        Yn += 0.5 * r * (UL + UR) * math.sin(Thetan) * dt
        if t == 0.5:
            Xn = round(2*Xn)/2
            Yn = round(2*Yn)/2
        Thetan += (r / L) * (UR - UL) * dt
        D += math.sqrt(math.pow((Xn),2) + math.pow((Yn),2))
        pt_list.append([Xn,Yn])

    # Set child coordinates
    # child_node_x = Xn
    # child_node_y = Yn
    # Round coordinates to nearest half to reduce node search
    # child_node_x = round(2*child_node_x)/2
    # child_node_y = round(2*child_node_y)/2

    # pt_list[4] = [child_node_x,child_node_y]

    return pt_list

# Resrouce: https://www.theconstructsim.com/ros-qa-how-to-convert-quaternions-to-euler-angles/
def odom_callback(msg): # rostopic type /odom # rosmsg show nav_msgs/Odometry
    # geometry_msgs/Quaternion orientation
    #   float64 x
    #   float64 y
    #   float64 z
    #   float64 w

    global odom_x, odom_y, odom_theta

    odom_x = msg.pose.pose.position.x
    odom_y = msg.pose.pose.position.y

    orientation_q = msg.pose.pose.orientation
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (roll, pitch, yaw) = euler_from_quaternion (orientation_list)
    odom_theta = yaw

def gazebo_publisher(robot_path):

    ##########################################################################
    rospy.init_node('astar_node', anonymous=True)

    # Publish to /cmd_vel ros topic
    vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
    velocity_msg = Twist()
    rate = rospy.Rate(10)  # 10 Hz = 1/.1 = 10 Hz Frequency Publish Rate

    ##########################################################################

    global odom_x, odom_y, odom_theta
    odom_x, odom_y, odom_theta = 0.0, 0.0, 0.0
```

```
# Subscribe to /odom topic
rospy.Subscriber('/odom', Odometry, odom_callback)

##########################################################################

# pt_list.append([Xn, Yn, Thetan, xdot, ydot, thetadot])
total_nodes = len(robot_path)
while not rospy.is_shutdown():

    for i, pt in enumerate(robot_path):

        print("Processing point", i)

        if (math.sqrt((pt[0] - odom_x)**2 + (pt[1] - odom_y)**2) < 0.1):

            print("Next point achieved. No corrections needed.")
            print("Odom X: ", odom_x, "x: ", pt[0])
            print("Odom Y: ", odom_y, "y: ", pt[1])
            print("Diff: ", math.sqrt((pt[0] - odom_x)**2 + (pt[1] - odom_y)**2) )
            print()

            # Publish new values
            velocity_msg.linear.x = math.sqrt((pt[3])**2 + (pt[4])**2) # xdot and ydot
            velocity_msg.linear.y = 0
            velocity_msg.linear.z = 0
            velocity_msg.angular.x = 0
            velocity_msg.angular.y = 0
            velocity_msg.angular.z = pt[5]  # thetadot

            vel_pub.publish(velocity_msg)

            rate.sleep()

        else:

            # Adjust orientation

            threshold = 0.1  # Initial threshold
            scale_factor = 0.5
            pt2 = pt[2]

            # if odom_theta > pt2:
            #     adjustment = -1
            # else:
            #     adjustment = 1


            while abs(odom_theta - pt2) > threshold:
                print("Odom Theta: ", odom_theta, "Theta: ", pt2, "Node: ", i, "/", total_r
                print("Diff Theta: ", abs(odom_theta - pt2))
                print()

                # theta_to_pub = odom_theta + (adjustment*scale_factor)
                theta_to_pub = abs(odom_theta - pt2) * scale_factor

                # Publish new values
                velocity_msg.linear.x = 0
                velocity_msg.linear.y = 0
                velocity_msg.linear.z = 0
                velocity_msg.angular.x = 0
                velocity_msg.angular.y = 0
                velocity_msg.angular.z = theta_to_pub

                vel_pub.publish(velocity_msg)
```

```
                rate.sleep()

            print("Theta corrected.")
            print()


            # Adjust pose
            dist_factor = 0.1 # 0.1
            x_desired = pt[0]
            y_desired = pt[1]

            euclid_dist = (math.sqrt((x_desired - odom_x) ** 2 + (y_desired - odom_y) ** 2)

            while (euclid_dist) > 0.1: #(odom_x != pt[0]) and (odom_y != pt[1]):

                print("Odom X: ", odom_x, "X: ", pt[0], "Node: ", i, "/", total_nodes)
                print("Odom Y: ", odom_y, "Y: ", pt[1], "Node: ", i, "/", total_nodes)
                print("Diff Pose: ", math.sqrt((pt[0] - odom_x)**2 + (pt[1] - odom_y)**2) )
                print()

                # Publish vel values to move in straight line
                velocity_msg.linear.x = dist_factor * euclid_dist
                velocity_msg.linear.y = 0
                velocity_msg.linear.z = 0
                velocity_msg.angular.x = 0
                velocity_msg.angular.y = 0
                velocity_msg.angular.z = 0

                vel_pub.publish(velocity_msg)

                rate.sleep()

                euclid_dist = (math.sqrt((x_desired - odom_x) ** 2 + (y_desired - odom_y) *

            velocity_msg.linear.x = 0
            velocity_msg.linear.y = 0
            velocity_msg.linear.z = 0
            velocity_msg.angular.x = 0
            velocity_msg.angular.y = 0
            velocity_msg.angular.z = 0

            vel_pub.publish(velocity_msg)

            print("Pose corrected.")

            print("Corrections made. Process next point.")
            print("##################################################")
            print()

        print("Last coordinate reached, terminating program.")
        break

    return

# Function gets user input, prints results of dijkstra_approach_alg function, optimal path is a
# Resource for time functions: https://stackoverflow.com/questions/27779677/how-to-format-elaps
# Resouce for RBG map grid color selection: https://www.rapidtables.com/web/color/RGB_Color.htm
# Resource for creation animation video: https://docs.opencv.org/3.4/dd/d9e/classcv_1_1VideoWri

def main_func():

    gazebo_coordinates = []
```

```
robot_radius = 105*0.1 # mm to cm

# Map/grid dimensions defined per problem statements
map_height = 2 # meters
map_width = 6 # meters
map_height = map_height * 100 # convert meters to cm
map_width = map_width * 100 # convert meters to cm

# Define RGB colors and attributes for map grid image
obstacle_space_color = (156,14,38)
free_space_color = (0,0,0)
clearance_color = (102, 0, 0)
optimal_path_color = (0,204,204)
start_node_color = (255, 255, 0)
goal_node_color = (0,153,0)
line_color = (0, 255, 0)

text_font = cv2.FONT_HERSHEY_SIMPLEX
plt_origin = (85, 10)
video_origin = (85, 25)
font_scale = 0.5
color = (255, 0, 0)
thickness = 1 # assume pixels

start_goal_pt_thickness = 3
traversal_thickness = 1

robot_clearance, robot_radius = 0, 0
robot_clearance, RPM1, RPM2 = get_user_robot_input()

# Create map grid and store original width and height of map image
map_grid, map_height, map_width = create_map_grid(clearance_color, obstacle_space_color, fr
original_map_height = map_height
original_map_width = map_width

#######************************************************************#######

# Resize map image using imutils resize function to speed up processing time, can alter wid
# associated height will automatically be computed to maintain aspect ratio
# map_grid = imutils.resize(map_grid, width = 300)
map_height, map_width, _ = np.shape(map_grid)

# plt.figure()
# plt.title('Resized Map Grid')
# plt.imshow(map_grid.astype(np.uint8), origin="lower")
# plt.show()

#######************************************************************#######

# Set up matrix to keep track of obstacle and free spaces
map_boundary_matrix = map_obstacle_freespace_matrix(map_grid, map_height, map_width)
obstacle_matrix = map_boundary_matrix.copy()

# Get user defined initial and goal node coordinates
x_initial, y_initial, x_goal, y_goal, th_initial = get_user_input(map_width, map_height, ch
initial_node_coord = (x_initial, y_initial)
goal_node_coord = (x_goal, y_goal)

print("Map Created and User Input Saved")
print()

# Plot the resized map grid with the obstacle and free space
# plt.figure()
# plt.title('Resized Map Grid')
```

```
# plt.imshow(map_grid.astype(np.uint8), origin="lower")
# plt.show()

##########################################################################################

# Use two different ways to compute the time the dijkstra algorithm takes to solve the give
start1 = time.time()
start2 = datetime.now()

arrow_img = np.copy(map_grid)

line_points, visited_queue, goal_found, fig, ax, closest_node_to_goal_x, closest_node_to_gc

end1 = time.time()
end2 = datetime.now()

print("Was goal found ? -> ", goal_found)
print()

if goal_found == False:
    print("No solution.")
    print()
    #return

##########################################################################################

# Execution Time Computed - Method 1
hrs, remain = divmod(end1 - start1, 3600)
mins, secs = divmod(remain, 60)
print("- Problem solved in (hours:min:sec:milliseconds) (Method 1): {:0>2}:{:0>2}:{:05.2f}'

# Execution Time Computed - Method 2
runtime=end2-start2
print("- Problem solved in (hours:min:sec:milliseconds) (Method 2): " + str(runtime))
print()

print("Start node coordinate input:", (x_initial,y_initial))
print("Goal node coordinate input:", (x_goal,y_goal))
print()

##########################################################################################

# Call function to compute optimal path
if goal_found:
    optimal_path = compute_optimal_path(visited_queue, initial_node_coord, goal_node_coord,

##########################################################################################

# Debug Statement
# Plot the resized map grid with the obstacle and free space
# plt.figure()
# plt.title('Resized Map Grid')
# plt.imshow(map_grid.astype(np.uint8), origin="lower")
# plt.show()

# Create animation visualization video
out = cv2.VideoWriter('conn_astar_algorithm_video.avi', cv2.VideoWriter_fourcc(*'XVID'), 50

##########################################################################################

# Plots start and goal nodes
map_grid = cv2.circle(map_grid, (x_initial,y_initial), radius = 0, color=start_node_color,
map_grid = cv2.circle(map_grid, (x_goal,y_goal), radius = 0, color=goal_node_color, thickne
```

```
last_parent_x = next(iter(visited_queue))[0]
last_parent_y = next(iter(visited_queue))[1]
last_parent_x = int(last_parent_x)
last_parent_y = int(last_parent_y)

# Skip first element as we do not have a case to handle it yet
db_cnt = 1
visited_queue_iter = iter(visited_queue)
next(visited_queue_iter)

arrow_img = np.copy(map_grid)
output_frames = []

for visited_node in visited_queue_iter:

    arrow_img = np.copy(map_grid)


    # Skip first element as we do not have a case to handle it yet
    if db_cnt == 1:
        db_cnt+= 1
        continue

    pc = visited_queue[visited_node][3]
    px = int(pc[0])
    py = int(pc[1])
    traverse_list = line_points[visited_node]
    # pt_list.append([X,Y, xdot, ydot, thetadot])


    # Convert traverse_list to NumPy array
    traverse_arr = np.array(traverse_list, dtype=int)

    traverse_float = np.array(traverse_list)

    gazebo_coordinates.extend(traverse_float)

    # Draw lines
    line_start = np.column_stack((px, py))
    line_end = traverse_arr[:, :2].astype(int)

    line_thickness = 1
    cv2.polylines(arrow_img, [line_start, line_end], isClosed=False, color=line_color, thic

    # Overlay the arrow on the original map
    map_grid = cv2.addWeighted(map_grid, 0.5, arrow_img, 0.5, 0)

    # Draw circles
    goal_node_color = (153, 0, 0)
    circle_radius = 0
    circle_thickness = 2
    map_grid = cv2.circle(map_grid, (x_goal, y_goal), radius=circle_radius, color=goal_node
    map_grid = cv2.circle(map_grid, (x_initial, y_initial), radius=circle_radius, color=sta
    map_grid = cv2.circle(map_grid, (line_end[4][0],line_end[4][1]), radius=circle_radius,

    # Necessary image processing for proper color display
    output_frame = cv2.flip(map_grid, 0)  # change y axis of image
    output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR)  # change color space to r
    output_frame = cv2.resize(output_frame, (original_map_width, original_map_height))  # r
    output_frames.append(output_frame)

# Write all frames to animation video
for frame in output_frames:
    out.write(frame)
```

```
################################################################################
if goal_found:
    last_parent_x = optimal_path[0][0]
    last_parent_y = optimal_path[0][1]

    last_parent_x = int(last_parent_x)
    last_parent_y = int(last_parent_y)

    for optimal_node in optimal_path:

        # Define the color of the arrow
        color = optimal_path_color

        # Add the arrow to the plot
        arrow_img = np.copy(map_grid)

        cv2.line(arrow_img, (last_parent_x, last_parent_y), (int(optimal_node[0]), int(opti
        map_grid = cv2.circle(map_grid, (int(optimal_node[0]), int(optimal_node[1])), radiu

        # Overlay the arrow on the original map
        map_grid = cv2.addWeighted(map_grid, 0.5, arrow_img, 0.5, 0)


        map_grid[int(optimal_node[1]),int(optimal_node[0])] = optimal_path_color # optimal
        map_grid = cv2.circle(map_grid, (x_initial,y_initial), radius=0, color=start_node_c
        map_grid = cv2.circle(map_grid, (x_goal,y_goal), radius=0, color=goal_node_color, t

        output_frame = cv2.flip(map_grid, 0) # change y axis
        output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR) # change color space t

        output_frame = cv2.resize(output_frame, (original_map_width, original_map_height))

        out.write(output_frame) # write image framr to animation video

        last_parent_x = optimal_node[0]
        last_parent_y = optimal_node[1]

        last_parent_x = int(last_parent_x)
        last_parent_y = int(last_parent_y)

################################################################################

output_frame = cv2.flip(map_grid, 0) # change y axis
output_frame = cv2.cvtColor(output_frame, cv2.COLOR_RGB2BGR) # change color space to reflec

# Output to terminal/console
if goal_found:
    output_frame = cv2.putText(output_frame, 'Solution Found', video_origin, text_font, fon
    map_grid = cv2.putText(map_grid, 'Solution Found', plt_origin, text_font, font_scale, (

else:
    output_frame = cv2.putText(output_frame, 'No Solution', video_origin, text_font, font_s
    map_grid = cv2.putText(map_grid, 'No Solution', plt_origin, text_font, font_scale, (0,

output_frame = cv2.resize(output_frame, (original_map_width, original_map_height)) # resize
out.write(output_frame) # write image framr to animation video
out.release() # release video object, done writing frames to video

################################################################################

# Display last frame to Python IDE
plt.figure()
```

```
    plt.title('Final Map Grid')
    plt.imshow(map_grid.astype(np.uint8), origin="lower")
    plt.show()

    # Display last frame to video
    ax.set_title('Final Map Grid')
    ax.axis('off')
    ax.imshow((map_grid).astype(np.uint8), animated=True, origin="lower")

    print("Astar and Visual Code Complete.")

    ###############################################################################################

    # print("gazebo_coordinates:", gazebo_coordinates[:15])

    # pt_list.append([Xn, Yn, Thetan, xdot, ydot, thetadot])

    # Convert coordinates from cm to meters and scale points to gazebo origin
    gazebo_coordinates[2] = 3.14 * gazebo_coordinates[2] / 180 # convert to radians
    gazebo_coordinates_scaled = [((x * 0.01) - 0.5, (y * 0.01) - 1, _, xdot * 0.01, ydot * 0.01
    robot_path = gazebo_coordinates_scaled

    gazebo_publisher(robot_path)

    print("Gazebo Code Complete.")

    # End of algorithm

################################################################################################
# Call main function

main_func()

# End of code file
################################################################################################
```