```python
# This is a sample Python script.

# Press Shift+F10 to execute it or replace it with your code.
# Press Double Shift to search everywhere for classes, files, tool windows, actions, and settings.
import cv2
import numpy as np
import heapq
import math

def move_Left60(parent_node, step_size): #60 degree counter clockwise movement
    x = parent_node[0]
    y = parent_node[1]
    theta = parent_node[2]
    if theta + 60 >= 360:
        new_theta=theta + 60-360
    else:
        new_theta=theta + 60
    Left60 = ((x+step_size*math.cos((new_theta*math.pi/180)),y+step_size*math.sin((new_theta*math.pi/180)), new_theta), step_size)
    return Left60

def move_Left30(parent_node, step_size): #30 degree counter clockwise movement
    x = parent_node[0]
    y = parent_node[1]
    theta = parent_node[2]
    if theta + 30 >= 360:
```

```
        new_theta=theta + 30-360
    else:
        new_theta=theta + 30
    Left30 = ((x+step_size*math.cos((new_theta*math.pi/180)),y+step_size*math.sin((new_theta*math.pi/180)), new_theta), step_size)
    return Left30

def move_straight(parent_node, step_size): #Applies straight forward movement
    x = parent_node[0]
    y = parent_node[1]
    theta = parent_node[2]
    new_theta=theta
    Straight = ((x+step_size*math.cos((new_theta*math.pi/180)),y+step_size*math.sin((new_theta*math.pi/180)), new_theta), step_size)
    return Straight

def move_Right30(parent_node, step_size): #30 degree clockwise movement
    x = parent_node[0]
    y = parent_node[1]
    theta = parent_node[2]
    if theta - 30 < 0:
        new_theta=theta - 30+360
    else:
        new_theta=theta - 30
    Right30 = ((x+step_size*math.cos((new_theta*math.pi/180)),y+step_size*math.sin((new_theta*math.pi/180)), new_theta), step_size)
    return Right30
```

```python

def move_Right60(parent_node, step_size): #60 degree clockwise movement
    x = parent_node[0]
    y = parent_node[1]
    theta = parent_node[2]
    if theta - 60 < 0:
        new_theta=theta - 60+360
    else:
        new_theta=theta - 60
    Right60 = ((x+step_size*math.cos((new_theta*math.pi/180)),y+step_size*math.sin((
new_theta*math.pi/180)), new_theta), step_size)
    return Right60

def generate_path(reverse_path):
    next_node = []
    while next_node != "N/A":
        search_for = reverse_path[-1]
        reverse_path.append(cost_to_come[search_for]['parent node'])
        next_node = cost_to_come[search_for]['parent node']

    print("This is the reverse path from goal to start", reverse_path)

    # This loop creates the forward path to goal
    t = 0
    forward_path = []
    for t in range(len(reverse_path)):
        forward_path.append(reverse_path.pop(-1))
```

```python

    # # this eliminates the start node from forward_path
    forward_path.pop(0)
    return forward_path

def check_for_goal(parent_node, goal_node):
    SolutionFound=False
    x = parent_node[0]
    y = parent_node[1]
    goal_x = goal_node[0]
    goal_y = goal_node[1]
    if (((goal_x-x)**2 + (goal_y-y)**2)**(0.5)) <= (1.5*step_size):
        SolutionFound=True
    return SolutionFound

def obstacle_check(parent_node, clearance,obstacle_match):   #used to confirm user inputs
    obstacle_match=True
    x = parent_node[0]
    y = parent_node[1]
    if (x >= (100-clearance)) and (x <= (150+clearance)) and (y >= 0) and (y <= (100+clearance)):  # Obstacle A check
        obstacle_match=True
    else:
        if (x >= (100-clearance)) and (x <= (150+clearance)) and (y >= (150-clearance)) and (y <= 250):  # Obstacle B check
            obstacle_match = True
```

```
        else:
            # Obstacle C1 check
            if (x >= 300 - 37.5 * 3 ** 0.5 - clearance) and (x <= 300) and (
                    y >= -(1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance**2+clearance**2) ** (0.5))) and (
                    y <= (1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance**2+clearance**2) ** (0.5))):
                obstacle_match = True
            else:
                # Obstacle C2 check
                if (x <= 300 + 37.5 * 3 ** 0.5 + clearance) and (x >= 300) and (
                        y >= (1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance**2+clearance**2) ** (0.5))) and (
                        y <= -(1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance**2+clearance**2) ** (0.5))):
                    obstacle_match = True
                else:
                    # Obstacle D check
                    if (x >= (460 - clearance)) and (y >= ((100+clearance) / (50+2*clearance) * (x - (510 + clearance)) + 125)) and (y <= (-1 * (100+clearance) / (50+2*clearance) * (x - (510 + clearance)) + 125)):
                        obstacle_match=True
                    else:
                        # Walls check
                        if (x <= clearance) or (x >= (600 - clearance)) or (y <= clearance) or (y >= (250 - clearance)):
                            obstacle_match=True
```

```
                        else:
                            obstacle_match = False
    return obstacle_match

#Initialize the starting robot parameters
clearance=0
step_size=0
start_cost=0.0

theta_choices=set([330, 300, 270, 240, 210, 180, 150, 120, 90, 60, 30, 0])
clearance=float(input('What is the radius of the robot?\n'))
step_size=float(input('How far does the robot move each action?\n'))

#This part of the code builds the map and obstacles based on inputed robot size
visual_map=np.zeros((500, 1200, 3), np.uint8)
visual_map[0:500,0:1200,:] = [0,0,255]
visited_nodes=np.zeros((500,1200,12))

cost_to_come=[]
x=0
y=0
node = 0
cost_to_come={}
obstacles=[]
#Loop builds obstacle list and primes np array for visualization of the space
for i in range(1200):
    for j in range(500):
```

```
        x=i*0.5
        y=j*0.5

        if (x >= (100 - clearance)) and (x <= (150 + clearance)) and (y >= 0) and (y <= (100 + clearance)):  # Obstacle A check
            visited_nodes[j][i][:]=-1.0
            obstacles.append((x,y))
        else:
            if (x >= (100 - clearance)) and (x <= (150 + clearance)) and (y >= (150 - clearance)) and (y <= 250):  # Obstacle B check
                visited_nodes[j][i][:] = -1.0
                obstacles.append((x,y))
            else:
                #Obstacle C1 check
                if (x >= (300 - 37.5 * 3 ** 0.5 - clearance)) and (x <= 300) and (y >= (-(1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance ** 2 + clearance ** 2) ** (0.5)))) and (y <= ((1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance ** 2 + clearance ** 2) ** (0.5)))):
                    visited_nodes[j][i][:] = -1.0
                    obstacles.append((x,y))
                else:
                    #Obstacle C2 check
                    if (x <= (300 + 37.5 * 3 ** 0.5 + clearance)) and (x >= 300) and (y >= ((1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance ** 2 + clearance ** 2) ** (0.5)))) and (y <= (-(1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance ** 2 + clearance ** 2) ** (0.5)))):
                        visited_nodes[j][i][:] = -1.0
```

```
                        obstacles.append((x,y))
                    else:
                        #Obstacle D check
                        if (x >= (460 - clearance)) and (y >= (((100 + clearance) / (50 + 2 * clearance) * (x - (510 + clearance)) + 125))) and (y <= ((-1 * (100 + clearance) / (50 + 2 * clearance) * (x - (510 + clearance)) + 125))):
                            visited_nodes[j][i][:] = -1.0
                            obstacles.append((x,y))
                        else:
                            #Walls check
                            if (x <= clearance) or (x >= (600 - clearance)) or (y <= clearance) or (y >= (250 - clearance)):
                                visited_nodes[j][i][:] = -1.0
                                obstacles.append((x,y))
                            else:
                                # visited_nodes[j][i][:] = 0
                                #cost_to_come[node]={'x': x,'y': y,'parent node': "N/A", 'cost to come': float('inf')}
                                visual_map[j,i,:] = [100,100,100]


#obstacle_list used when new nodes are created to check if they are in obstacle space
obstacle_list=set(obstacles)


#Visualize Space
```

```
cv2.imshow("Zeros matx", visual_map)  # show numpy array
cv2.waitKey(0)  # wait for ay key to exit window
cv2.destroyAllWindows()  # close all windows

#Define start node by x,y, and theta coordinates
bad_choice = True
obstacle_match = True
while bad_choice == True:
    start_x=input('What is the x coordinate (integer only) of your starting point?\n')
    start_x=int(start_x)
    start_y=input('What is the y coordinate (integer only) of your starting point?\n')
    start_y=int(start_y)
    start_theta = input('What is the initial facing of the robot?  Enter a positive or negative multiple 30 between 0 and 330.\n')
    start_theta = int(start_theta)
    start_node = (start_x, start_y, start_theta)
    bad_choice_check=obstacle_check(start_node,clearance, obstacle_match)
    if bad_choice_check == True:
        print('This is an invalid starting position, please try again.\n')
    else:
        if abs(start_theta) in theta_choices:
            bad_choice=False
        else: print('This is an invalid initial facing of the robot, please try again.\n')

```

```python
start_theta_index=int(start_theta/30)
visited_nodes[start_y][start_x][start_theta_index]=0.0

#Define goal node by x,y, and theta coordinates
bad_choice = True
obstacle_match = True
while bad_choice == True:
    goal_x = input('What is the x coordinate (integer only) of your target position?\n')
    goal_x=int(goal_x)
    goal_y = input('What is the y coordinate (integer only) of your target position?\n')
    goal_y=int(goal_y)
    goal_theta = input('What is the final facing of the robot?  Enter a positive or negative multiple 30 between 0 and 330.\n')
    goal_theta = int(goal_theta)
    goal_node = (goal_x, goal_y, goal_theta)
    bad_choice_check=obstacle_check(goal_node,clearance, obstacle_match)
    if bad_choice_check == True:
        print('This is an invalid goal position, please try again.\n')
    else:
        if abs(goal_theta) in theta_choices:
            bad_choice=False
        else: print('This is an invalid final facing of the robot, please try again.\n')

#Initializing node map
```

```
distance_to_goal = ((goal_x - start_x) ** 2 + (goal_y - start_y) ** 2) ** (0.5)
cost_to_come[start_node]={'x':start_x, 'y':start_y, 'theta': start_theta, 'parent node':"N/A", 'cost to come':0, 'total cost':distance_to_goal}

SolutionFound = False
counter=0



#Initialize list of nodes that need to be expanded/investigated/have moves applied
queue=[]
queue=[[cost_to_come[start_node]['total cost'],start_node]]

closed_list=[]
closed_list_check = set()

while SolutionFound!=True:           #counter<100: #

    heapq.heapify(queue)

    # identify the parent node and eliminate it from the list of nodes that need to be investigated
    parent_node=heapq.heappop(queue)
    parent_node=parent_node[1]

    # check if the last popped node is a match for goal
    # if it's a match, initializes reverse_path list and adds node to node map
```

```
    SolutionFound=check_for_goal(parent_node, goal_node)
    if SolutionFound == True:
        print(cost_to_come[parent_node])
        reverse_path = [goal_node,parent_node]
        print(reverse_path)
        break

    closed_list.append(parent_node)
    closed_list_check.add(parent_node)
# #     #perform "moves" on "parent" node to create "new" nodes

    Left60=move_Left60(parent_node, step_size)

    Left30=move_Left30(parent_node, step_size)

    Straight=move_straight(parent_node, step_size)

    Right30=move_Right30(parent_node, step_size)

    Right60=move_Right60(parent_node, step_size)


    #stores the "new" nodes in another dictionary for future use in loop
    action_dict={0:Left60, 1:Left30, 2:Straight, 3:Right30, 4:Right60}

    #initialize variable before loop begins
```

```
    match=False

    #for each action in action_dict loop runs
    for j in range(len(action_dict)):
        match=False

        #Setting up Rounded node to determine if node has been found
        action_node=action_dict[j][0]
        action_x=action_node[0]
        action_y=action_node[1]
        #if statement checks if new node is outside map (can happen with larger step sizes)
        if action_x >= 600 or action_x <= 0 or action_y >= 250 or action_y <= 0:
            break
        action_theta=action_node[2]
        rounded_x=int(round(action_x * 2)) #since input to array must be integer, this multiplies the rounded value by 2 to get the proper matrix placement
        rounded_y=int(round(action_y * 2))
        action_coord = (rounded_x/2, rounded_y/2)
        theta_index=int(action_theta/30)

        #Determining new cost-to-go, cost-to-come, and total cost
        distance_to_goal = ((goal_x - action_x)**2 + (goal_y - action_y)**2)**0.5
        new_cost_to_come = cost_to_come[parent_node]['cost to come'] + action_dict[j][1]
        new_cost = new_cost_to_come + distance_to_goal

```

```python

        if action_dict[j][0] in closed_list_check or action_coord in obstacle_list or visited_nodes[rounded_y][rounded_x][theta_index]==1:  #obs_check==True:
            match=True

        else:
             #Adds node to node map and open list if nearest node has not been visited (zero value)
            if visited_nodes[rounded_y][rounded_x][theta_index]==0:
                cost_to_come[action_dict[j][0]] = {'x': action_x, 'y': action_y, 'theta': action_theta, 'parent node': parent_node, 'cost to come': new_cost_to_come, 'total cost': new_cost }
                queue.append([cost_to_come[action_dict[j][0]]['total cost'], action_dict[j][0]])
                visited_nodes[rounded_y][rounded_x][theta_index]=1

    counter=counter+1
    #print(counter)
#This loop creates the reverse path by searching for the next parent node until the start node's parent "NA" is found
forward_path=generate_path(reverse_path)

cv2.imshow("Zeros matx", visual_map)  # show numpy array
cv2.waitKey(0)  # wait for ay key to exit window
cv2.destroyAllWindows()  # close all windows


```

```
x=[]
y=[]
visual_map_explore=visual_map
for i in range(len(closed_list)):
    coord=closed_list[i]
    Left60 = move_Left60(coord, step_size)
    node=Left60[0]
    x1=2*int(round(node[0]))
    y1=500-(int(round(node[1])))*2
    point1=(x1, y1)
    Left30 = move_Left30(coord, step_size)
    node = Left30[0]
    x2=2*int(round(node[0]))
    y2=500-(int(round(node[1])))*2
    point2 = (x2,  y2)
    Straight = move_straight(coord, step_size)
    node = Straight[0]
    x3 = 2 * int(round(node[0]))
    y3 = 500 - (int(round(node[1]))) * 2
    point3 = (x3,  y3)
    Right30 = move_Right30(coord, step_size)
    node = Right30[0]
    x4=2*int(round(node[0]))
    y4=500-(int(round(node[1])))*2
    point4 = (x4,  y4)
    Right60 = move_Right60(coord, step_size)
    node = Right60[0]
```

```
    x5=2*int(round(node[0]))
    y5=500-(int(round(node[1])))*2
    point5 = (x5,  y5)

    y=500-(int(coord[1]))*2
    x=2*(int(coord[0]))

    x_arr=(250-int(coord[1]))*2
    y_arr=int(coord[0])*2
    visual_map_explore[x_arr][y_arr]=0

    cv2.arrowedLine(visual_map_explore, (x,y), point1, (255,0,0),1 )
    cv2.arrowedLine(visual_map_explore, (x, y), point2, (255, 0, 0), 1)
    cv2.arrowedLine(visual_map_explore, (x, y), point3, (255, 0, 0), 1)
    cv2.arrowedLine(visual_map_explore, (x, y), point4, (255, 0, 0), 1)
    cv2.arrowedLine(visual_map_explore, (x, y), point5, (255, 0, 0), 1)
    cv2.imshow("Zeros matx", visual_map_explore)  # show numpy array
    cv2.waitKey(1)  # wait for ay key to exit window




cv2.destroyAllWindows()  # close all windows

x=[]
y=[]
for i in range(len(forward_path)):
```

```
    coord=forward_path[i]
    x_arr=(250-int(coord[1]))*2
    y_arr=int(coord[0])*2
    y=500-(int(coord[1]))*2
    x=2*(int(coord[0]))
    if i != 0:
        cv2.arrowedLine(visual_map, prev_node, (x,y), (0, 255, 0), 1)

    visual_map[x_arr,y_arr,:] = [0,255,0]
    cv2.imshow("Zeros matx", visual_map)  # show numpy array
    cv2.waitKey(50)  # wait for ay key to exit window
    prev_node = (x, y)

cv2.waitKey(0)  # wait for ay key to exit window

cv2.destroyAllWindows()  # close all windows
```