

```
1 # This is a sample Python script.
2
3 # Press Shift+F10 to execute it or replace it with your code.
4 # Press Double Shift to search everywhere for classes, files, tool windows, actions
  , and settings.
5 import cv2
6 import numpy as np
7 import heapq
8 import math
9
10 def move_Left60(parent_node, step_size): #60 degree counter clockwise movement
11     x = parent_node[0]
12     y = parent_node[1]
13     theta = parent_node[2]
14     if theta + 60 >= 360:
15         new_theta=theta + 60-360
16     else:
17         new_theta=theta + 60
18     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)
19     return Left60
20
21 def move_Left30(parent_node, step_size): #30 degree counter clockwise movement
22     x = parent_node[0]
23     y = parent_node[1]
24     theta = parent_node[2]
25     if theta + 30 >= 360:
```

```
26         new_theta=theta + 30-360
27     else:
28         new_theta=theta + 30
29     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)
30     return Left30
31
32 def move_straight(parent_node, step_size): #Applies straight forward movement
33     x = parent_node[0]
34     y = parent_node[1]
35     theta = parent_node[2]
36     new_theta=theta
37     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)
38     return Straight
39
40 def move_Right30(parent_node, step_size): #30 degree clockwise movement
41     x = parent_node[0]
42     y = parent_node[1]
43     theta = parent_node[2]
44     if theta - 30 < 0:
45         new_theta=theta - 30+360
46     else:
47         new_theta=theta - 30
48     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)
49     return Right30
```

```
50
51 def move_Right60(parent_node, step_size): #60 degree clockwise movement
52     x = parent_node[0]
53     y = parent_node[1]
54     theta = parent_node[2]
55     if theta - 60 < 0:
56         new_theta=theta - 60+360
57     else:
58         new_theta=theta - 60
59     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)
60     return Right60
61
62 def generate_path(reverse_path):
63     next_node = []
64     while next_node != "N/A":
65         search_for = reverse_path[-1]
66         reverse_path.append(cost_to_come[search_for]['parent node'])
67         next_node = cost_to_come[search_for]['parent node']
68
69     print("This is the reverse path from goal to start", reverse_path)
70
71     # This loop creates the forward path to goal
72     t = 0
73     forward_path = []
74     for t in range(len(reverse_path)):
75         forward_path.append(reverse_path.pop(-1))
```

```
76
77     ## this eliminates the start node from forward_path
78     forward_path.pop(0)
79     return forward_path
80
81 def check_for_goal(parent_node, goal_node):
82     SolutionFound=False
83     x = parent_node[0]
84     y = parent_node[1]
85     goal_x = goal_node[0]
86     goal_y = goal_node[1]
87     if (((goal_x-x)**2 + (goal_y-y)**2)**(0.5)) <= (1.5*step_size):
88         SolutionFound=True
89     return SolutionFound
90
91 def obstacle_check(parent_node, clearance,obstacle_match):    #used to confirm user
inputs
92     obstacle_match=True
93     x = parent_node[0]
94     y = parent_node[1]
95     if (x >= (100-clearance)) and (x <= (150+clearance)) and (y >= 0) and (y <= (
100+clearance)):    # Obstacle A check
96         obstacle_match=True
97     else:
98         if (x >= (100-clearance)) and (x <= (150+clearance)) and (y >= (150-
100-clearance)) and (y <= 250):    # Obstacle B check
99         obstacle_match = True
```

```

100         else:
101             # Obstacle C1 check
102             if (x >= 300 - 37.5 * 3 ** 0.5 - clearance) and (x <= 300) and (
103                 y >= -(1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance**2+
clearance**2) ** (0.5))) and (
104                 y <= (1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance**2+
clearance**2) ** (0.5))):
105                 obstacle_match = True
106         else:
107             # Obstacle C2 check
108             if (x <= 300 + 37.5 * 3 ** 0.5 + clearance) and (x >= 300) and (
109                 y >= (1 / 3) ** (0.5) * (x - 300) + (125 - 75 - (clearance
**2+clearance**2) ** (0.5))) and (
110                 y <= -(1 / 3) ** (0.5) * (x - 300) + (125 + 75 + (clearance
**2+clearance**2) ** (0.5))):
111                 obstacle_match = True
112         else:
113             # Obstacle D check
114             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)):
115                 obstacle_match=True
116         else:
117             # Walls check
118             if (x <= clearance) or (x >= (600 - clearance)) or (y <=
clearance) or (y >= (250 - clearance)):
119                 obstacle_match=True

```

```
120                 else:
121                     obstacle_match = False
122             return obstacle_match
123
124 #Initialize the starting robot parameters
125 clearance=0
126 step_size=0
127 start_cost=0.0
128
129 theta_choices=set([330, 300, 270, 240, 210, 180, 150, 120, 90, 60, 30, 0])
130 clearance=float(input('What is the radius of the robot?\n'))
131 step_size=float(input('How far does the robot move each action?\n'))
132
133 #This part of the code builds the map and obstacles based on inputed robot size
134 visual_map=np.zeros((500, 1200, 3), np.uint8)
135 visual_map[0:500,0:1200,:] = [0,0,255]
136 visited_nodes=np.zeros((500,1200,12))
137
138 cost_to_come=[]
139 x=0
140 y=0
141 node = 0
142 cost_to_come={}
143 obstacles=[]
144 #Loop builds obstacle list and primes np array for visualization of the space
145 for i in range(1200):
146     for j in range(500):
```

```

147         x=i*0.5
148         y=j*0.5
149
150         if (x >= (100 - clearance)) and (x <= (150 + clearance)) and (y >= 0) and (
y <= (100 + clearance)): # Obstacle A check
151             visited_nodes[j][i][:]=-1.0
152             obstacles.append((x,y))
153         else:
154             if (x >= (100 - clearance)) and (x <= (150 + clearance)) and (y >= (150
- clearance)) and (y <= 250): # Obstacle B check
155                 visited_nodes[j][i][:] = -1.0
156                 obstacles.append((x,y))
157             else:
158                 #Obstacle C1 check
159                 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))))):
160                     visited_nodes[j][i][:] = -1.0
161                     obstacles.append((x,y))
162                 else:
163                     #Obstacle C2 check
164                     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))))):
165                         visited_nodes[j][i][:] = -1.0

```

```

166             obstacles.append((x,y))
167         else:
168             #Obstacle D check
169             if (x >= (460 - clearance)) and (y >= (((100 + clearance
170 ) / (50 + 2 * clearance) * (x - (510 + clearance)) + 125))) and (y <= ((-1 * (100
171 + clearance) / (50 + 2 * clearance) * (x - (510 + clearance)) + 125))):
172                 visited_nodes[j][i][:] = -1.0
173                 obstacles.append((x,y))
174             else:
175                 #Walls check
176                 if (x <= clearance) or (x >= (600 - clearance)) or (y
177 <= clearance) or (y >= (250 - clearance)):
178                     visited_nodes[j][i][:] = -1.0
179                     obstacles.append((x,y))
180                 else:
181                     # visited_nodes[j][i][:] = 0
182                     #cost_to_come[node]={'x': x, 'y': y, 'parent node': "
183 N/A", 'cost to come': float('inf')}}
184                     visual_map[j,i,:] = [100,100,100]
185
186
187 #obstacle_list used when new nodes are created to check if they are in obstacle
space
188 obstacle_list=set(obstacles)
189
190
191 #Visualize Space

```



```
188 cv2.imshow("Zeros matx", visual_map) # show numpy array
189 cv2.waitKey(0) # wait for any key to exit window
190 cv2.destroyAllWindows() # close all windows
191
192 #Define start node by x,y, and theta coordinates
193 bad_choice = True
194 obstacle_match = True
195 while bad_choice == True:
196     start_x=input('What is the x coordinate (integer only) of your starting point?\n')
197     start_x=int(start_x)
198     start_y=input('What is the y coordinate (integer only) of your starting point?\n')
199     start_y=int(start_y)
200     start_theta = input('What is the initial facing of the robot? Enter a positive or negative multiple 30 between 0 and 330.\n')
201     start_theta = int(start_theta)
202     start_node = (start_x, start_y, start_theta)
203     bad_choice_check=obstacle_check(start_node,clearance, obstacle_match)
204     if bad_choice_check == True:
205         print('This is an invalid starting position, please try again.\n')
206     else:
207         if abs(start_theta) in theta_choices:
208             bad_choice=False
209         else: print('This is an invalid initial facing of the robot, please try again.\n')
210
```

```
211 start_theta_index=int(start_theta/30)
212 visited_nodes[start_y][start_x][start_theta_index]=0.0
213
214 #Define goal node by x,y, and theta coordinates
215 bad_choice = True
216 obstacle_match = True
217 while bad_choice == True:
218     goal_x = input('What is the x coordinate (integer only) of your target position
        ?\n')
219     goal_x=int(goal_x)
220     goal_y = input('What is the y coordinate (integer only) of your target position
        ?\n')
221     goal_y=int(goal_y)
222     goal_theta = input('What is the final facing of the robot? Enter a positive or
        negative multiple 30 between 0 and 330.\n')
223     goal_theta = int(goal_theta)
224     goal_node = (goal_x, goal_y, goal_theta)
225     bad_choice_check=obstacle_check(goal_node,clearance, obstacle_match)
226     if bad_choice_check == True:
227         print('This is an invalid goal position, please try again.\n')
228     else:
229         if abs(goal_theta) in theta_choices:
230             bad_choice=False
231         else: print('This is an invalid final facing of the robot, please try again
        .\n')
232
233 #Initializing node map
```

```
234 distance_to_goal = ((goal_x - start_x) ** 2 + (goal_y - start_y) ** 2) ** (0.5)
235 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}
236
237 SolutionFound = False
238 counter=0
239
240
241
242 #Initialize list of nodes that need to be expanded/investigated/have moves applied
243 queue=[]
244 queue=[[cost_to_come[start_node]['total cost'],start_node]]
245
246 closed_list=[]
247 closed_list_check = set()
248
249 while SolutionFound!=True:                                #counter<100: #
250
251     heapq.heapify(queue)
252
253     # identify the parent node and eliminate it from the list of nodes that need to
be investigated
254     parent_node=heapq.heappop(queue)
255     parent_node=parent_node[1]
256
257     # check if the last popped node is a match for goal
258     # if it's a match, initializes reverse_path list and adds node to node map
```

```
259
260     SolutionFound=check_for_goal(parent_node, goal_node)
261     if SolutionFound == True:
262         print(cost_to_come[parent_node])
263         reverse_path = [goal_node,parent_node]
264         print(reverse_path)
265         break
266
267     closed_list.append(parent_node)
268     closed_list_check.add(parent_node)
269 # #         #perform "moves" on "parent" node to create "new" nodes
270
271     Left60=move_Left60(parent_node, step_size)
272
273     Left30=move_Left30(parent_node, step_size)
274
275     Straight=move_straight(parent_node, step_size)
276
277     Right30=move_Right30(parent_node, step_size)
278
279     Right60=move_Right60(parent_node, step_size)
280
281
282     #stores the "new" nodes in another dictionary for future use in loop
283     action_dict={0:Left60, 1:Left30, 2:Straight, 3:Right30, 4:Right60}
284
285     #initialize variable before loop begins
```

```

286     match=False
287
288     #for each action in action_dict loop runs
289     for j in range(len(action_dict)):
290         match=False
291
292         #Setting up Rounded node to determine if node has been found
293         action_node=action_dict[j][0]
294         action_x=action_node[0]
295         action_y=action_node[1]
296         #if statement checks if new node is outside map (can happen with larger
step sizes)
297         if action_x >= 600 or action_x <= 0 or action_y >= 250 or action_y <= 0:
298             break
299         action_theta=action_node[2]
300         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
301         rounded_y=int(round(action_y * 2))
302         action_coord = (rounded_x/2, rounded_y/2)
303         theta_index=int(action_theta/30)
304
305         #Determining new cost-to-go, cost-to-come, and total cost
306         distance_to_goal = ((goal_x - action_x)**2 + (goal_y - action_y)**2)**0.5
307         new_cost_to_come = cost_to_come[parent_node]['cost to come'] + action_dict[
    j][1]
308         new_cost = new_cost_to_come + distance_to_goal
309

```

```
310
311         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:
312             match=True
313
314         else:
315             #Adds node to node map and open list if nearest node has not been
visited (zero value)
316             if visited_nodes[rounded_y][rounded_x][theta_index]==0:
317                 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 }
318                 queue.append([cost_to_come[action_dict[j][0]]['total cost'],
action_dict[j][0]])
319                 visited_nodes[rounded_y][rounded_x][theta_index]=1
320
321         counter=counter+1
322         #print(counter)
323 #This loop creates the reverse path by searching for the next parent node until the
start node's parent "NA" is found
324 forward_path=generate_path(reverse_path)
325
326 cv2.imshow("Zeros matx", visual_map)  # show numpy array
327 cv2.waitKey(0)  # wait for ay key to exit window
328 cv2.destroyAllWindows()  # close all windows
329
330
```

```
331 x=[]
332 y=[]
333 visual_map_explore=visual_map
334 for i in range(len(closed_list)):
335     coord=closed_list[i]
336     Left60 = move_Left60(coord, step_size)
337     node=Left60[0]
338     x1=2*int(round(node[0]))
339     y1=500-(int(round(node[1])))*2
340     point1=(x1, y1)
341     Left30 = move_Left30(coord, step_size)
342     node = Left30[0]
343     x2=2*int(round(node[0]))
344     y2=500-(int(round(node[1])))*2
345     point2 = (x2, y2)
346     Straight = move_straight(coord, step_size)
347     node = Straight[0]
348     x3 = 2 * int(round(node[0]))
349     y3 = 500 - (int(round(node[1])) * 2
350     point3 = (x3, y3)
351     Right30 = move_Right30(coord, step_size)
352     node = Right30[0]
353     x4=2*int(round(node[0]))
354     y4=500-(int(round(node[1])))*2
355     point4 = (x4, y4)
356     Right60 = move_Right60(coord, step_size)
357     node = Right60[0]
```

```
358     x5=2*int(round(node[0]))
359     y5=500-(int(round(node[1])))*2
360     point5 = (x5,  y5)
361
362     y=500-(int(coord[1]))*2
363     x=2*(int(coord[0]))
364
365     x_arr=(250-int(coord[1]))*2
366     y_arr=int(coord[0])*2
367     visual_map_explore[x_arr][y_arr]=0
368
369     cv2.arrowedLine(visual_map_explore, (x,y), point1, (255,0,0),1 )
370     cv2.arrowedLine(visual_map_explore, (x, y), point2, (255, 0, 0), 1)
371     cv2.arrowedLine(visual_map_explore, (x, y), point3, (255, 0, 0), 1)
372     cv2.arrowedLine(visual_map_explore, (x, y), point4, (255, 0, 0), 1)
373     cv2.arrowedLine(visual_map_explore, (x, y), point5, (255, 0, 0), 1)
374     cv2.imshow("Zeros matx", visual_map_explore) # show numpy array
375     cv2.waitKey(1) # wait for ay key to exit window
376
377
378
379
380 cv2.destroyAllWindows() # close all windows
381
382 x=[]
383 y=[]
384 for i in range(len(forward_path)):
```



```
385     coord=forward_path[i]
386     x_arr=(250-int(coord[1]))*2
387     y_arr=int(coord[0])*2
388     y=500-(int(coord[1]))*2
389     x=2*(int(coord[0]))
390     if i != 0:
391         cv2.arrowedLine(visual_map, prev_node, (x,y), (0, 255, 0), 1)
392
393     visual_map[x_arr,y_arr,:] = [0,255,0]
394     cv2.imshow("Zeros matx", visual_map) # show numpy array
395     cv2.waitKey(50) # wait for ay key to exit window
396     prev_node = (x, y)
397
398 cv2.waitKey(0) # wait for ay key to exit window
399
400 cv2.destroyAllWindows() # close all windows
401
402
```