

## a\_star\_manas\_ronen\_vaibhav.py

```

1  # Github link: https://github.com/manasdesai/Project3_Phase1_Planning
2  #!/usr/bin/env python3
3
4  """
5  ## Team Members
6  1. Manas Desai (120998973) (mdesai01)
7  2. Ronen Aniti (112095116) (raniti)
8  3. Vaibhav Yuwaraj Shende (121206817) (svaibhav)
9
10 """
11
12 # Import Libraries
13 import time
14 import matplotlib.colors as mcolors
15 import numpy as np
16 import matplotlib.pyplot as plt
17 from enum import Enum
18 from queue import PriorityQueue
19 from matplotlib.animation import FuncAnimation
20 from matplotlib.colors import ListedColormap, BoundaryNorm
21 from matplotlib.patches import Patch
22 from matplotlib.lines import Line2D
23 from typing import Tuple
24
25 #####
26 ##### CONSTANTS #####
27 #####
28 LOGGING = False
29
30
31 #####
32 ##### DEFINE THE ACTIONS #####
33 #####
34 class Action(Enum):
35     """
36     Enum to represent the actions for the A* search algorithm.
37
38     """
39
40     def __init__(self, r: int, theta: int) -> None:
41         """Represents unit vector version of each action"""
42         self.r = r
43         self.theta = theta
44
45     LEFT60 = (1, -60)
46     LEFT30 = (1, -30)
47     STRAIGHT = (1, 0)
48     RIGHT30 = (1, 30)

```

```

49     RIGHT60 = (1, 60)
50
51
52 #####
53 ##### DEFINE THE COORDINATE TRANSFORMATIONS #####
54 #####
55 def coordinate_transformation(
56     standard_frame: Tuple, spatial_resolution, angular_resolution
57 ) -> Tuple:
58     """
59     Convert a standard frame (mm, mm, deg) to a grid frame (grid_x, grid_y,
60     grid_theta).
61
62     Args:
63         standard_frame (Tuple): A tuple representing the standard frame (x, y, theta)
64         in mm and degrees.
65         spatial_resolution (): _spatial resolution in mm per unit_
66         angular_resolution (_type_): _angular resolution in degrees per unit_
67
68     Returns:
69         Tuple: A tuple representing the grid frame (grid_x, grid_y, grid_theta) in
70         grid units.
71     """
72     return (
73         int(standard_frame[0] * spatial_resolution),
74         int(standard_frame[1] * spatial_resolution),
75         int(standard_frame[2] / angular_resolution),
76     )
77
78 def coordinate_transformation_2dof(standard_frame_2d, spatial_resolution) -> Tuple:
79     """
80     Convert a 2D standard frame (mm, mm) to a grid frame (grid_x, grid_y).
81
82     Args:
83         standard_frame_2d (Tuple): A tuple representing the 2D standard frame (x, y)
84         in mm.
85         spatial_resolution (): _spatial resolution in mm per unit_
86
87     Returns:
88         Tuple: A tuple representing the grid frame (grid_x, grid_y) in grid units.
89     """
90     return (
91         int(standard_frame_2d[0] * spatial_resolution),
92         int(standard_frame_2d[1] * spatial_resolution),
93     )
94
95 def coordinate_transformation_inverse(
96     grid_frame, spatial_resolution, angular_resolution
97 ) -> Tuple:

```

```

96     return (
97         int(grid_frame[0] / spatial_resolution),
98         int(grid_frame[1] / spatial_resolution),
99         int(grid_frame[2] * angular_resolution),
100     )
101
102
103 def coordinate_transformation_inverse_2dof(grid_frame, spatial_resolution) -> Tuple:
104     return (
105         int(grid_frame[0] / spatial_resolution),
106         int(grid_frame[1] / spatial_resolution),
107     )
108
109
110 def angle_to_index(angle_deg, angular_resolution) -> int:
111     return int(angle_deg / angular_resolution)
112
113
114 def index_to_angle(angle_index, angular_resolution) -> int:
115     return int(angle_index * angular_resolution)
116
117
118 #####
119 ##### DEFINE COLLISION DETECTION #####
120 #####
121 def collision(x: int, y: int, scale: int, safety: int) -> bool:
122     """
123     Check if a point (x, y) collides with the obstacles in the workspace.
124
125     Args:
126         x (int): x coordinate in mm
127         y (int): y coordinate in mm
128         scale (int): scale factor for the workspace
129         safety (int): safety margin around obstacles in mm
130
131     Returns:
132         bool: True if the point collides with an obstacle.
133     """
134     regions = []
135
136     # Wall buffer region
137     regions.append(
138         (x - safety <= 0)
139         or (x - 180 * scale + safety >= 0)
140         or (y - safety <= 0)
141         or (y - 50 * scale + safety >= 0)
142     )
143
144     # E
145     #####

```

```
146 # E, vertical rectangle primitive
147 regions.append(
148     (x - 20 * scale + safety >= 0)
149     and (x - 25 * scale - safety <= 0)
150     and (y - 10 * scale + safety >= 0)
151     and (y - 35 * scale - safety <= 0)
152 )
153
154 # E, bottom rectangle primitive
155 regions.append(
156     (x - 20 * scale + safety >= 0)
157     and (x - 33 * scale - safety <= 0)
158     and (y - 10 * scale + safety >= 0)
159     and (y - 15 * scale - safety <= 0)
160 )
161
162 # E, middle rectangle primitive
163 regions.append(
164     (x - 20 * scale + safety >= 0)
165     and (x - 33 * scale - safety <= 0)
166     and (y - 20 * scale + safety >= 0)
167     and (y - 25 * scale - safety <= 0)
168 )
169
170 # E, top rectangle primitive
171 regions.append(
172     (x - 20 * scale + safety >= 0)
173     and (x - 33 * scale - safety <= 0)
174     and (y - 30 * scale + safety >= 0)
175     and (y - 35 * scale - safety <= 0)
176 )
177
178 # N
179 #####
180 # N, left vertical rectangle primitive
181 regions.append(
182     (x - 43 * scale + safety >= 0)
183     and (x - 48 * scale - safety <= 0)
184     and (y - 10 * scale + safety >= 0)
185     and (y - 35 * scale - safety <= 0)
186 )
187
188 # N, middle in between two segment region
189 regions.append(
190     (y + 3 * x - 179 * scale - safety * np.sqrt(10) <= 0)
191     and (y + 3 * x - 169 * scale + safety * np.sqrt(10) >= 0)
192     and (x - 48 * scale + safety >= 0)
193     and (x - 53 * scale - safety <= 0)
194     and (y - 10 * scale + safety >= 0)
195     and (y - 35 * scale - safety <= 0)
```

```
196     )
197
198     # N, right vertical rectangle primitive
199     regions.append(
200         (x - 53 * scale + safety >= 0)
201         and (x - 58 * scale - safety <= 0)
202         and (y - 10 * scale + safety >= 0)
203         and (y - 35 * scale - safety <= 0)
204     )
205
206     # P
207     #####
208     # P, left vertical bar
209     regions.append(
210         (x - 68 * scale + safety >= 0)
211         and (x - 73 * scale - safety <= 0)
212         and (y - 10 * scale + safety >= 0)
213         and (y - 35 * scale - safety <= 0)
214     )
215
216     # P, semi-circular region
217     regions.append(
218         (
219             (x - 73 * scale) ** 2
220             + (y - 28.75 * scale) ** 2
221             - (6.25 * scale + safety) ** 2
222             <= 0
223         )
224         and (x - 73 * scale - safety >= 0)
225     )
226
227     # M
228     #####
229     # M, first vertical bar (left vertical)
230     regions.append(
231         (x - 85 * scale + safety >= 0)
232         and (x - 90 * scale - safety <= 0)
233         and (y - 10 * scale + safety >= 0)
234         and (y - 35 * scale - safety <= 0)
235     )
236     # M, left diagonal region
237     regions.append(
238         (5 * x + y - 485 * scale - safety * np.sqrt(26) <= 0)
239         and (5 * x + y - 483 * scale + safety * np.sqrt(26) >= 0)
240         and (x - 90 * scale + safety >= 0)
241         and (x - 95 * scale - safety <= 0)
242         and (y - 10 * scale + safety >= 0)
243         and (y - 35 * scale - safety <= 0)
244     )
245     # M, right diagonal region
```

```

246     regions.append(
247         (5 * x - y - 465 * scale - safety * np.sqrt(26) <= 0)
248         and (5 * x - y - 463 * scale + safety * np.sqrt(26) >= 0)
249         and (x - 95 * scale + safety >= 0)
250         and (x - 100 * scale - safety <= 0)
251         and (y - 10 * scale + safety >= 0)
252         and (y - 35 * scale - safety <= 0)
253     )
254 # M, second vertical bar (right vertical)
255 regions.append(
256     (x - 100 * scale + safety >= 0)
257     and (x - 105 * scale - safety <= 0)
258     and (y - 10 * scale + safety >= 0)
259     and (y - 35 * scale - safety <= 0)
260 )
261
262 # 6
263 #####
264 # 6, circle primitive
265 regions.append(
266     (
267         ((x - 120 * scale) ** 2 + (y - 17.5 * scale) ** 2)
268         <= (7.5 * scale + safety) ** 2
269     )
270 )
271 # 6, vertical rectangle on top of the circle
272 regions.append(
273     (x - 112.5 * scale + safety >= 0)
274     and (x - 117.5 * scale - safety <= 0)
275     and (y - 17.5 * scale + safety >= 0)
276     and (y - 35 * scale - safety <= 0)
277 )
278 # 6, horizontal rectangle attached to the right of the vertical rectangle
279 regions.append(
280     (x - 117.5 * scale + safety >= 0)
281     and (x - 123 * scale - safety <= 0)
282     and (y - 33 * scale + safety >= 0)
283     and (y - 35 * scale - safety <= 0)
284 )
285
286 # 6 (second 6)
287 #####
288 # 6 (second 6), circle primitive
289 regions.append(
290     (
291         ((x - 139.5 * scale) ** 2 + (y - 17.5 * scale) ** 2)
292         <= (7.5 * scale + safety) ** 2
293     )
294 )
295 # 6 (second 6), vertical rectangle

```

```

296     regions.append(
297         (x - 132 * scale + safety >= 0)
298         and (x - 137 * scale - safety <= 0)
299         and (y - 17.5 * scale + safety >= 0)
300         and (y - 35 * scale - safety <= 0)
301     )
302     # 6 (second 6), horizontal rectangle
303     regions.append(
304         (x - 137 * scale + safety >= 0)
305         and (x - 143 * scale - safety <= 0)
306         and (y - 33 * scale + safety >= 0)
307         and (y - 35 * scale - safety <= 0)
308     )
309
310     # 1
311     #####
312     # 1, rectangle primitive
313     regions.append(
314         (x - 155 * scale + safety >= 0)
315         and (x - 160 * scale - safety <= 0)
316         and (y - 10 * scale + safety >= 0)
317         and (y - 35 * scale - safety <= 0)
318     )
319
320     return any(regions)
321
322
323     #####
324     ## DEFINE FUNCTIONALITY TO CREATE DIFFERENT TYPES OF OCCUPANCY GRIDS #####
325     #####
326     def generate_occupancy_grid(workspace_dimension, spatial_resolution, scale, safety):
327         """
328         Generate an occupancy grid for the workspace based on the specified dimensions,
329         spatial resolution, scale, and safety margin.
330
331         Args:
332             workspace_dimension (): The dimensions of the workspace (width, height) in
333             mm.
334             spatial_resolution (): The spatial resolution in mm per unit.
335             scale (): The scale factor for the workspace, used to determine the size of
336             obstacles.
337             safety (): The safety margin around obstacles in mm.
338
339         Returns:
340             np.ndarray: A 2D boolean occupancy grid where True indicates an obstacle and
341             False indicates free space.
342         """
343         occupancy_grid_shape = (
344             int(workspace_dimension[0] * spatial_resolution) + 1,
345             int(workspace_dimension[1] * spatial_resolution) + 1,
346         )

```

```

343
344     occupancy_grid = np.zeros(occupancy_grid_shape, dtype=bool)
345
346     for x in range(occupancy_grid_shape[0]):
347         for y in range(occupancy_grid_shape[1]):
348             grid_coord = (x, y)
349             frame_coord = coordinate_transformation_inverse_2dof(
350                 grid_coord, spatial_resolution
351             )
352             if collision(frame_coord[0], frame_coord[1], scale, safety):
353                 occupancy_grid[x, y] = True
354
355     return occupancy_grid
356
357
358 def generate_occupancy_grid_for_plotting(
359     workspace_dimension, spatial_resolution, scale, safety
360 ):
361     occupancy_grid_shape = (
362         int(workspace_dimension[0] * spatial_resolution) + 1,
363         int(workspace_dimension[1] * spatial_resolution) + 1,
364     )
365
366     occupancy_grid_for_plotting = np.zeros(occupancy_grid_shape, dtype=int)
367
368     for x in range(occupancy_grid_shape[0]):
369         for y in range(occupancy_grid_shape[1]):
370             grid_coord = (x, y)
371             frame_coord = coordinate_transformation_inverse_2dof(
372                 grid_coord, spatial_resolution
373             )
374             if collision(frame_coord[0], frame_coord[1], scale, 0.0):
375                 occupancy_grid_for_plotting[x, y] = (
376                     2 # Obstacle region but not in padding
377                 )
378             elif collision(frame_coord[0], frame_coord[1], scale, safety):
379                 occupancy_grid_for_plotting[x, y] = (
380                     1 # Padded region but not in obstacle region
381                 )
382
383     return occupancy_grid_for_plotting
384
385
386 #####
387 ##### DEFINE A HEURISTIC FUNCTION FOR A* SEARCH #####
388 #####
389 def euclidean_ground_distance(node1, node2):
390     """
391     Get the Euclidean distance between 2 points
392

```



```

393     Args:
394         node1 (tuple): coordinates of first point
395         node2 (tuple): coordinates of second point
396
397     Returns:
398         float: euclidean distance
399     """
400     return round(np.sqrt((node1[0] - node2[0]) ** 2 + (node1[1] - node2[1]) ** 2), 2)
401
402
403     #####
404     # DEFINE A NEIGHBOR EXPANSION FUNCTION FOR A* SEARCH #####
405     #####
406     def find_valid_neighbors(
407         occupancy_grid, current, step_size, spatial_resolution, angular_resolution
408     ):
409         # The output of this must be two lists in grid units only
410
411         # Define a list of valid actions
412         actions_list = [
413             Action.LEFT60,
414             Action.LEFT30,
415             Action.STRAIGHT,
416             Action.RIGHT30,
417             Action.RIGHT60,
418         ]
419
420         # Define lists for valid neighbors and distances
421         valid_neighbors_list = [] # grid frame
422         distances_list = [] # grid distances
423
424         # Convert to standard frame (mm, mm, deg) for collision checking
425         current_standard_frame = coordinate_transformation_inverse(
426             current, spatial_resolution, angular_resolution
427         )
428
429         for action in actions_list:
430             # Get the dimensionless radius of expansion
431             r = action.r
432
433             # The action angular displacement in degrees:
434             delta_theta = action.theta
435
436             # The new x coordinate in mm
437             new_node_standard_x = current_standard_frame[0] + r * step_size * np.cos(
438                 np.deg2rad(current_standard_frame[2] + delta_theta)
439             )
440
441             # The new y coordinate in mm
442             new_node_standard_y = current_standard_frame[1] + r * step_size * np.sin(

```

```

443         np.deg2rad(current_standard_frame[2] + delta_theta)
444     )
445
446     # The new theta coordinate in degrees
447     new_node_standard_theta = current_standard_frame[2] + delta_theta
448     new_node_standard_theta = (
449         new_node_standard_theta % 360
450     ) # Wrap around 360 degrees
451
452     # The new node in (mm, mm, deg) format
453     new_node_standard = (
454         new_node_standard_x,
455         new_node_standard_y,
456         new_node_standard_theta,
457     )
458
459     # The new node in grid units
460     new_node_grid = coordinate_transformation(
461         new_node_standard, spatial_resolution, angular_resolution
462     )
463
464     # If the grid coordinate of the new node is out of bounds, skip
465     if new_node_grid[0] < 0 or new_node_grid[0] >= occupancy_grid.shape[0]:
466         continue
467
468     if new_node_grid[1] < 0 or new_node_grid[1] >= occupancy_grid.shape[1]:
469         continue
470
471     # Mark the new node as valid if its in bounds and not an obstacle
472     if not occupancy_grid[new_node_grid[0], new_node_grid[1]]:
473         valid_neighbors_list.append(new_node_grid)
474         distances_list.append(
475             r * step_size * spatial_resolution
476         ) # The cost is measured in grid units
477
478     return valid_neighbors_list, distances_list
479
480
481 #####
482 ##### DEFINE A BACKTRACKING FUNCTION FOR A* SEARCH #####
483 #####
484 def backtrack(predecessors, start, goal):
485     """
486     Simple backtracking routine to extract the path from the branching dictionary
487     """
488     path = [goal]
489     current = goal
490     while predecessors[current] != None:
491         parent = predecessors[current]
492         path.append(parent)

```

```

493         current = parent
494     return path[::-1]
495
496
497 #####
498 ##### IMPLEMENTATION OF A* PATHFINDING #####
499 #####
500 def astar(
501     occupancy_grid,
502     color_occupancy_grid,
503     start,
504     goal,
505     scale_factor,
506     safety_margin,
507     step_size,
508     spatial_resolution,
509     angular_resolution,
510     h=euclidean_ground_distance,
511     logging=False,
512 ):
513     # *Assume start and goal are in grid units*
514
515     # Convert the start node to standard coordinate from for validity check
516     start_standard_frame = coordinate_transformation_inverse(
517         start, spatial_resolution, angular_resolution
518     )
519
520     # Convert the goal node to standard coordinate frame for validity check
521     goal_standard_frame = coordinate_transformation_inverse(
522         goal, spatial_resolution, angular_resolution
523     )
524
525     # Check the validity of the start node
526     if collision(
527         start_standard_frame[0], start_standard_frame[1], scale_factor, safety_margin
528     ):
529         print("The start node is invalid")
530         return None, None
531
532     # Check the validity of the goal node
533     if collision(
534         goal_standard_frame[0], goal_standard_frame[1], scale_factor, safety_margin
535     ):
536         print("The goal node is invalid")
537         return None, None
538
539     # Construct lists to store open set and closed set history for animation
540     if logging:
541         search_array_history = []
542

```

```
543     # Initialize A* data structures
544     parents = dict()
545
546     # Construct an array data structure to combine open set and closed set
information.
547     # Unvisited and unopened = 0
548     # Visited = 1
549     # Open = 2
550     search_array_shape = (
551         occupancy_grid.shape[0],
552         occupancy_grid.shape[1],
553         int(360 / angular_resolution),
554     )
555     search_array = np.zeros(search_array_shape, dtype=np.int8)
556
557     g_scores = dict()
558     queue = PriorityQueue()
559     goal_is_found = False
560
561     # Handle the start node
562     search_array[start[0], start[1], start[2]] = 2 # 2 -> open
563     parents[start] = None
564     g_scores[start] = 0.0
565     f_score_start = g_scores[start] + h(start, goal)
566     queue.put((f_score_start, start))
567
568     # Logging
569     iteration = 1
570
571     # Begin the A* main loop
572     while not queue.empty():
573         # Logging
574         iteration += 1
575         if logging and (iteration % 5000 == 0):
576             print(f"Logging the {iteration}th iteration of A*")
577             search_array_history.append(search_array.copy())
578
579         # POP the most promising node
580         f_current, current = queue.get()
581
582         # ASSUME that some queued nodes may be visited nodes. Skip them.
583         if search_array[current[0], current[1], current[2]] == 1: # 1 -> visited
584             continue
585
586         # ONLY proceed to visit and process unvisited nodes:
587
588         # Mark the current node as visited
589         search_array[current[0], current[1], current[2]] = 1 # 1-> visited
590
591         # Stop search if the goal is found
```

```

592     if current[:2] == goal[:2]:
593         print("A* has found the goal")
594         found_goal = current # May have different orientation, but that's OK
595         goal_is_found = True
596
597     if logging:
598         print(f"Logging the {iteration}th iteration of A*")
599         search_array_history.append(search_array.copy())
600     break
601
602     # If this current node is NOT the goal node:
603
604     # Expand the neighbors of this current node:
605     valid_neighbors_list, distances_list = find_valid_neighbors(
606         occupancy_grid, current, step_size, spatial_resolution,
angular_resolution
607     )
608     for i, neighbor in enumerate(valid_neighbors_list):
609         # ASSUME that some NEIGHBORS may be VISITED already. Skip them.
610         if search_array[neighbor[0], neighbor[1], neighbor[2]] == 1: # 1 ->
visited
611             continue
612
613         # ASSUME that some NEIGHBORS may already be in the OPEN set. Process
them, but IF AND ONLY IF a better partial plan would result.
614         if search_array[neighbor[0], neighbor[1], neighbor[2]] == 2: # 2 -> open
615             g_current = g_scores[current] # g-score of current node
616             g_tentative = g_current + distances_list[i]
617             if g_tentative < g_scores[neighbor]:
618                 g_scores[neighbor] = g_tentative
619                 parents[neighbor] = current
620                 f_score_neighbor = g_tentative + h(neighbor, goal)
621                 queue.put((f_score_neighbor, neighbor))
622
623         # ASSUME that some NEIGHBORS may be NOT in the OPEN SET and NOT in the
CLOSED SET.
624         if (
625             search_array[neighbor[0], neighbor[1], neighbor[2]] == 0
626         ): # 0 -> unvisited and unseen
627             search_array[neighbor[0], neighbor[1], neighbor[2]] = 2 # 2 -> open
628             g_tentative = g_scores[current] + distances_list[i]
629             parents[neighbor] = current
630             g_scores[neighbor] = g_tentative
631             f_score_neighbor = g_tentative + h(neighbor, goal)
632             queue.put((f_score_neighbor, neighbor))
633
634     if goal_is_found:
635         cost = g_scores[found_goal] # cost in grid units
636
637         path = backtrack(parents, start, found_goal) # path is in grid units
638

```

```

639     # Logging
640     if logging:
641         animate_search(color_occupancy_grid, search_array_history, path)
642
643     # Return the path and cost
644     return path, cost
645
646 path("No path found")
647 return None, None
648
649 #####
650 ##### GENERATE AN ANIMATION FUNCTION FOR VISUALIZATION OF A* RESULTS ##
651 #####
652 def animate_search(color_occ_grid, search_array_history, path):
653     # Append 10 copies of the final frame to ensure the final frame is shown on
654     # screen for a sufficient amount of time.
655     final_frame = search_array_history[-1]
656     for _ in range(10):
657         search_array_history.append(final_frame)
658
659     # Create the figure and axis.
660     fig, ax = plt.subplots(figsize=(8, 6))
661
662     # Plot the colored occupancy grid as the background.
663     ax.imshow(
664         color_occ_grid.T,
665         origin="lower",
666         cmap="inferno",
667         alpha=1.0,
668         extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
669     )
670
671     # Create a discrete colormap to show the search evolution
672     # 0: unvisited (transparent), 1: Closed (blue), 2: Open (yellow)
673     search_cmap = ListedColormap([(0, 0, 0, 0), "blue", "yellow"])
674     search_norm = BoundaryNorm([0, 0.5, 1.5, 2.5], search_cmap.N)
675
676     # Code that ensures that each x, y location displays on the plot as VISITED
677     # (blue) if ANY orientation at that location has been VISITED
678     first_frame = search_array_history[0]
679     H, W, 0 = first_frame.shape
680     first_frame_list_of_lists = [[0 for _ in range(W)] for _ in range(H)]
681     for i in range(H):
682         for j in range(W):
683             location_2d = first_frame[i, j, :]
684             if 1 in location_2d:
685                 first_frame_list_of_lists[i][j] = 1
686             elif 2 in location_2d:
687                 first_frame_list_of_lists[i][j] = 2

```

```
687         else:
688             first_frame_list_of_lists[i][j] = 0
689
690     # Display the initial search space
691     search_space = ax.imshow(
692         np.array(first_frame_list_of_lists).T,
693         origin="lower",
694         cmap=search_cmap,
695         norm=search_norm,
696         alpha=0.6,
697         extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
698     )
699
700     # Extract start and goal coordinates from the path.
701     start_coords = path[0]
702     goal_coords = path[-1]
703
704     # Plot start and goal as static objects
705     ax.scatter(start_coords[0], start_coords[1], s=50, color="chartreuse")
706     ax.scatter(goal_coords[0], goal_coords[1], s=50, color="magenta")
707
708     # Initialize a line object to eventually contain the path line
709     (path_line,) = ax.plot([], [], "r-", linewidth=2)
710
711     # Set title and axis labels.
712     ax.set_title("A* Search Evolution")
713     ax.set_xlabel("X-Coordinate Grid Frame")
714     ax.set_ylabel("Y-Coordinate Grid Frame")
715
716     # Create the legend markers
717     # CLOSED SET patch (blue)
718     closed_patch = Patch(facecolor="blue", edgecolor="blue", label="Closed")
719     # OPEN SET patch (yellow)
720     open_patch = Patch(facecolor="yellow", edgecolor="yellow", label="Open")
721     # START marker
722     start_marker = Line2D(
723         [0],
724         [0],
725         marker="o",
726         color="w",
727         markerfacecolor="chartreuse",
728         markersize=8,
729         label="Start",
730     )
731     # GOAL marker
732     goal_marker = Line2D(
733         [0],
734         [0],
735         marker="o",
736         color="w",
```

```

737     markerfacecolor="magenta",
738     markersize=8,
739     label="Goal",
740 )
741 # PATH marker
742 path_marker = Line2D([0], [0], color="red", lw=2, label="Final Path")
743 # PLACE the legend in the top right corner
744 ax.legend(
745     handles=[closed_patch, open_patch, start_marker, goal_marker, path_marker],
746     loc="upper right",
747 )
748
749 # Define an update function for the animation of the A* search evolution
750 def update_frame(frame_index):
751     frame = search_array_history[frame_index]
752     H, W, O = frame.shape
753     # Collapse the 3D frame into 2D explicitly.
754     frame_as_list_of_lists = [[0 for _ in range(W)] for _ in range(H)]
755     for i in range(H):
756         for j in range(W):
757             location_2d = frame[i, j, :]
758             if 1 in location_2d:
759                 frame_as_list_of_lists[i][j] = 1
760             elif 2 in location_2d:
761                 frame_as_list_of_lists[i][j] = 2
762             else:
763                 frame_as_list_of_lists[i][j] = 0
764
765     # Add the frame data to the visualization
766     search_space.set_data(np.array(frame_as_list_of_lists).T)
767
768     # Show the final path for the final 10 iterations of the animation
769     if frame_index >= len(search_array_history) - 10:
770         xs = [p[0] for p in path]
771         ys = [p[1] for p in path]
772         path_line.set_data(xs, ys)
773
774     return search_space, path_line
775
776 # Create the animation.
777 anim = FuncAnimation(
778     fig,
779     update_frame,
780     frames=len(search_array_history),
781     interval=200,
782     blit=False,
783     repeat=True,
784 )
785
786 print("Saving the animation (gif)...")

```



```

787     # Save the animation as a GIF.
788     anim.save("astar_animation.gif", writer="pillow", fps=5)
789     print("Animation saved as astar_animation.gif")
790
791     # Save the animation as an MP4.
792     print("Saving the animation (mp4)...")
793     anim.save("astar_animation.mp4", writer="ffmpeg", fps=5)
794     print("Animation saved as astar_animation.mp4")
795
796     plt.show()
797
798
799     #####
800     ##### THE MAIN FUNCTION #####
801     #####
802     if __name__ == "__main__":
803         # This project is an extension of Project 2.
804         # The obstacles from Project 2 will be used.
805         # Recall the exact x, y dimensions of the workspace from Project 2
806         # However they will be dilated by a fixed factor.
807         ORIGINAL_WORKSPACE_DIMENSION = (180, 50)
808         SCALE_FACTOR = 3.0
809
810         # The resulting workspace dimension is this:
811         NEW_WORKSPACE_DIMENSION = (
812             ORIGINAL_WORKSPACE_DIMENSION[0] * SCALE_FACTOR,
813             ORIGINAL_WORKSPACE_DIMENSION[1] * SCALE_FACTOR,
814         )
815
816         # The Project 2 specifications also call for a spatial resolution of 0.5mm/ unit
817         # (2 units/mm) and for an angular resolution of 30 degrees per unit. So I am setting
818         # that here.
819         # The coordinates of the workspace are assumed to be
820         # (mm, mm, degrees) for (x, y, theta).
821         SPATIAL_RESOLUTION = 2.0
822         ANGULAR_RESOLUTION = 30.0
823
824         # Define the step size for the branch expansion process for the A* component of
825         # the pathfinding operation. This will be in mm.
826         step_size = float(input("Enter the step size (mm): "))
827
828         # Define a safety margin around obstacles
829         safety_margin = float(input("Enter a safety margin around obstacles (mm): "))
830
831         robot_radius = float(input("Enter the robot radius (mm): "))
832
833         # Use the larger of the input `robot_radius` and the input `safety_margin` to
834         # serve as the actual `safety_margin` used in pathfinding.
835         safety_margin = max(safety_margin, robot_radius)
836
837         # Define the start pose (mm, mm, deg)

```

```

834     x_start, y_start, theta_start = map(
835         float,
836         input(
837             "Enter the (mm, mm, deg) coordinates for the start pose, separated by
only spaces:"
838         ).split(),
839     )
840     start = (x_start, y_start, theta_start)
841
842     # Define the goal pose (mm, mm, deg)
843     x_goal, y_goal, theta_goal = map(
844         float,
845         input(
846             "Enter the (mm, mm, deg) coordinates for the goal pose, separated by only
spaces:"
847         ).split(),
848     )
849     goal = (x_goal, y_goal, theta_goal)
850
851     # Convert the start and goal poses to the grid frame
852     start = coordinate_transformation(start, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION)
853
854     goal = coordinate_transformation(goal, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION)
855
856     # Define an occupancy grid for Project 3 having the structure:
857     # grid[i][j] -> i -> units of mm * spatial_resolution
858     #               j -> units of mm * spatial_resolution
859     # The grid will be a Numpy bool array and will be constructed using
860     # a `collision` function having the hard-coded obstacle information
861     # specified in Project 2
862     print("Constructing the occupancy grid...")
863     occupancy_grid = generate_occupancy_grid(
864         NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
865     )
866     print("The occupancy grid construction is complete.")
867
868     # This is a step non-necessary for pathfinding but necessary for static
visualization and for animation. We will proceed with static visualization to aid the
grader in assessing the result. But we will skip animation generation, instead
including an animation from a prior run with our submission.
869     print("Generating a color occupancy grid for plotting purposes")
870     color_occupancy_grid = generate_occupancy_grid_for_plotting(
871         NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
872     )
873
874     #####
875     #####CALL TO A* #####
876     #####
877     # Executes A* search, return path and cost
878     print("Executing A* search from start to goal.")

```

```
879     # ONLY TURN LOGGING TO `True` IF YOU WANT TO GENERATE ANIMATION FILES AND HAVE
MATPLOTLIB ALSO SHOW THE ANIMATION.
880     # RECOMMENDED: KEEP FALSE. THIS PROCESS IS TIME CONSUMING.
881     START_TIME = time.perf_counter()
882     path, cost = astar(
883         occupancy_grid,
884         color_occupancy_grid,
885         start,
886         goal,
887         SCALE_FACTOR,
888         safety_margin,
889         step_size,
890         SPATIAL_RESOLUTION,
891         ANGULAR_RESOLUTION,
892         logging=LOGGING,
893     )
894     END_TIME = time.perf_counter()
895     RUN_TIME = END_TIME - START_TIME
896
897     print(f"Total time for execution: {RUN_TIME} seconds.")
898
899     # Proceed with static visualization to aid the grader in assessing the results.
This shows the color occupancy grid with the final path.
900     if path:
901         # Print the path in grid units
902         print(f"The path, in grid units, is as follows: {path}")
903
904         # Print the cost in grid units
905         print(f"The total cost (length) of the path, in grid units, is {cost}")
906
907         # Print the path in units of (mm, mm deg)
908         print(
909             f"The final path in units of (mm, mm, deg) is {[coordinate_transform-
ation_inverse(p, SPATIAL_RESOLUTION, ANGULAR_RESOLUTION) for p in path]}"
910         )
911
912         # Print the cost in standard units (mm)
913         print(f"The path length in mm is {cost / SPATIAL_RESOLUTION}")
914
915         # Construct a static plot of the output to assist the grader in assessing
performance
916         print(
917             "Constructing an occupancy grid for plotting to highlight the padded
region..."
918         )
919
920         occupancy_grid_for_plotting = generate_occupancy_grid_for_plotting(
921             NEW_WORKSPACE_DIMENSION, SPATIAL_RESOLUTION, SCALE_FACTOR, safety_margin
922         )
923
924         print("The occupancy grid for plotting is complete.")
```

```
925
926     # Show the color occupancy grid
927     plt.imshow(
928         occupancy_grid_for_plotting.T,
929         origin="lower",
930         cmap="inferno",
931         extent=[0, occupancy_grid.shape[0], 0, occupancy_grid.shape[1]],
932     )
933
934     # Make a chart title
935     plt.title("Workspace with Path")
936
937     # Overlay the path on the occupancy grid
938     xs_path = [node[0] for node in path]
939     ys_path = [node[1] for node in path]
940     us_path = [np.cos(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]
941     vs_path = [np.sin(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]
942
943     # Include orientation in the static visualiation
944     plt.quiver(
945         xs_path,
946         ys_path,
947         us_path,
948         vs_path,
949         color="r",
950         width=0.005,
951         scale_units="xy",
952         angles="xy",
953         scale=0.5,
954     )
955     plt.show()
956
```