

a_star_manas_ronen_vaibhav.py

```

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

```

```

49     RIGHT30 = (1, 30)
50     RIGHT60 = (1, 60)
51
52
53 #####
54 ##### DEFINE THE COORDINATE TRANSFORMATIONS #####
55 #####
56 def coordinate_transformation(
57     standard_frame: Tuple, spatial_resolution, angular_resolution
58 ) -> Tuple:
59     """
60     Convert a standard frame (mm, mm, deg) to a grid frame (grid_x, grid_y,
61     grid_theta).
62
63     Args:
64         standard_frame (Tuple): A tuple representing the standard frame (x, y, theta)
65         in mm and degrees.
66         spatial_resolution (): _spatial resolution in mm per unit_
67         angular_resolution (_type_): _angular resolution in degrees per unit_
68
69     Returns:
70         Tuple: A tuple representing the grid frame (grid_x, grid_y, grid_theta) in
71         grid units.
72     """
73     return (
74         int(standard_frame[0] * spatial_resolution),
75         int(standard_frame[1] * spatial_resolution),
76         int(standard_frame[2] / angular_resolution),
77     )
78
79 def coordinate_transformation_2dof(standard_frame_2d, spatial_resolution) -> Tuple:
80     """
81     Convert a 2D standard frame (mm, mm) to a grid frame (grid_x, grid_y).
82
83     Args:
84         standard_frame_2d (Tuple): A tuple representing the 2D standard frame (x, y)
85         in mm.
86         spatial_resolution (): _spatial resolution in mm per unit_
87
88     Returns:
89         Tuple: A tuple representing the grid frame (grid_x, grid_y) in grid units.
90     """
91     return (
92         int(standard_frame_2d[0] * spatial_resolution),
93         int(standard_frame_2d[1] * spatial_resolution),
94     )
95
96 def coordinate_transformation_inverse(
97     grid_frame, spatial_resolution, angular_resolution

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

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

```

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

```

```

639
640     # Logging
641     if logging:
642         animate_search(color_occupancy_grid, search_array_history, path)
643
644     # Return the path and cost
645     return path, cost
646
647     path("No path found")
648     return None, None
649
650
651 #####
652 ##### GENERATE AN ANIMATION FUNCTION FOR VISUALIZATION OF A* RESULTS ##
653 #####
654 def animate_search(color_occ_grid, search_array_history, path):
655     # Append 10 copies of the final frame to ensure the final frame is shown on
screen for a sufficient amount of time.
656     final_frame = search_array_history[-1]
657     for _ in range(10):
658         search_array_history.append(final_frame)
659
660     # Create the figure and axis.
661     fig, ax = plt.subplots(figsize=(8, 6))
662
663     # Plot the colored occupancy grid as the background.
664     ax.imshow(
665         color_occ_grid.T,
666         origin="lower",
667         cmap="inferno",
668         alpha=1.0,
669         extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
670     )
671
672     # Create a discrete colormap to show the search evolution
673     # 0: unvisited (transparent), 1: Closed (blue), 2: Open (yellow)
674     search_cmap = ListedColormap([(0, 0, 0, 0), "blue", "yellow"])
675     search_norm = BoundaryNorm([0, 0.5, 1.5, 2.5], search_cmap.N)
676
677     # Code that ensures that each x, y location displays on the plot as VISITED
(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
688     else:
689         first_frame_list_of_lists[i][j] = 0
690
691     # Display the initial search space
692     search_space = ax.imshow(
693         np.array(first_frame_list_of_lists).T,
694         origin="lower",
695         cmap=search_cmap,
696         norm=search_norm,
697         alpha=0.6,
698         extent=[0, color_occ_grid.shape[0], 0, color_occ_grid.shape[1]],
699     )
700
701     # Extract start and goal coordinates from the path.
702     start_coors = path[0]
703     goal_coors = path[-1]
704
705     # Plot start and goal as static objects
706     ax.scatter(start_coors[0], start_coors[1], s=50, color="chartreuse")
707     ax.scatter(goal_coors[0], goal_coors[1], s=50, color="magenta")
708
709     # Initialize a line object to eventually contain the path line
710     (path_line,) = ax.plot([], [], "r-", linewidth=2)
711
712     # Set title and axis labels.
713     ax.set_title("A* Search Evolution")
714     ax.set_xlabel("X-Coordinate Grid Frame")
715     ax.set_ylabel("Y-Coordinate Grid Frame")
716
717     # Create the legend markers
718     # CLOSED SET patch (blue)
719     closed_patch = Patch(facecolor="blue", edgecolor="blue", label="Closed")
720     # OPEN SET patch (yellow)
721     open_patch = Patch(facecolor="yellow", edgecolor="yellow", label="Open")
722     # START marker
723     start_marker = Line2D(
724         [0],
725         [0],
726         marker="o",
727         color="w",
728         markerfacecolor="chartreuse",
729         markersize=8,
730         label="Start",
731     )
732     # GOAL marker
733     goal_marker = Line2D(
734         [0],
735         [0],
736         marker="o",
```

```

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

```



```

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

```

```

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

```

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

```
926
927     # Show the color occupancy grid
928     plt.imshow(
929         occupancy_grid_for_plotting.T,
930         origin="lower",
931         cmap="inferno",
932         extent=[0, occupancy_grid.shape[0], 0, occupancy_grid.shape[1]],
933     )
934
935     # Make a chart title
936     plt.title("Workspace with Path")
937
938     # Overlay the path on the occupancy grid
939     xs_path = [node[0] for node in path]
940     ys_path = [node[1] for node in path]
941     us_path = [np.cos(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]
942     vs_path = [np.sin(np.deg2rad(node[2] * ANGULAR_RESOLUTION)) for node in path]
943
944     # Include orientation in the static visualiation
945     plt.quiver(
946         xs_path,
947         ys_path,
948         us_path,
949         vs_path,
950         color="r",
951         width=0.005,
952         scale_units="xy",
953         angles="xy",
954         scale=0.5,
955     )
956     plt.show()
957
```