Coursework 2 - PLAN

1 C-Space Representations

In motion planning for robotics, the concept of Configuration Space (C-Space) is fundamental. C-Space is a mathematical representation where the robot's configuration is described as a single point, and each point in this space corresponds to a feasible configuration of the robot in the real world. For DE NIRO, For DE NIRO, moving in a 2D plane, the C-Space is also 2D, representing its position in x and y coordinates. The map is scaled at 16 pixels per meter, with the origin (0, 0) at the center. The direct conversion between pixels and real-world dimensions is calculated by one pixel equaling 0.0625 meters.

1.1 Task A Part 1: Square C-Space Dilation

"Dilation" is an image processing technique used to expand obstacles in the C-space representation, ensuring a collision-free path for the robot. In this task, a square mask approximates DE NIRO's footprint, assuming it occupies a square region on the map. The width of the square is approximated by the width of DE NIRO (`robot_px`). The following code initializes a square array of ones, representing the robot's dimensions, to serve as the `robot_mask`, as shown in Figure 2.

```
def expand_map(img, robot_width):
    robot_px = int(robot_width * scale)   #size of the robot in pixels x axis
    robot_mask = np.ones((robot_px, robot_px)) #create a square mask using an array of ones
    plt.imshow(robot_mask, vmin=0, vmax=1, origin='lower')
    plt.show() #show the square mask plot
    expanded_img = binary_dilation(img, robot_mask) #perform binary dilation on the mask created

    return expanded_img
```

The square mask undergoes binary dilation, returning `expanded_img` and expanding the C-Space, as shown in Figure 3. Compared to the original map in Figure 1, obstacles (yellow parts) are inflated, narrowing the robot's pathways. While this prevents collisions, the oversized mask may unnecessarily restrict feasible routes.

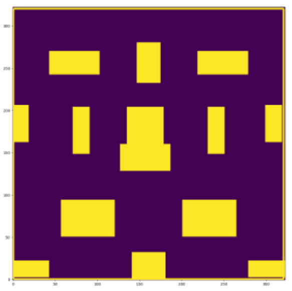


Figure 1: Original C-Space map (no dilation)

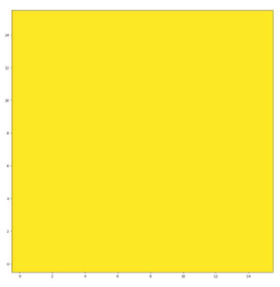


Figure 2:Square Mask Plot

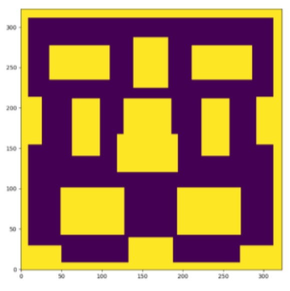


Figure 3: Dilated C-Space map (square dilation)

1.2 Task A Part 2: Circular C-Space Dilation

A circular mask provides a more accurate approximation of footprint of DE NIRO than a square mask, as shown in Figure 4. The `expand_map` function is modified to iterate through a zero-initialized matrix, computing each pixel’s Euclidean distance from the center as Equation 1 below.

If the distance is within the robot's radius, the corresponding pixel is set to one, forming a circular mask.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \qquad [1](1)$$

As shown in Figure 5, the circular mask results in rounded obstacle edges and provides slightly more navigable space than the square representation, offering a better approximation of boundaries of DE NIRO. However, it does not fully capture sharp corners which potentially eliminates feasible paths. Additionally, it incurs higher computational costs due to distance calculations. Despite these limitations, circular representation remains the preferred choice in this case.

```python
def expand_map(img, robot_width):
    robot_px = int(robot_width * scale)   #size of the robot in pixels x axis

    robot_mask = np.zeros((robot_px, robot_px)) #create a matrix of zeros
    # iterate the matrix
    for i in range(robot_px):
        for j in range(robot_px):
            # since Python arrays are zero-indexed, a shift is applied to adjust the point coordinates
            x = i - (robot_px - 1) / 2
            y = j - (robot_px - 1) / 2
            # calculate the distance from the center using Pythagorean theorem
            distance = np.sqrt(x**2 + y**2)
            # If the distance is within the robot's radius, mark it as 1
            if distance <= robot_px / 2:
                robot_mask[i, j] = 1

    plt.imshow(robot_mask, vmin=0, vmax=1, origin='lower')
    plt.show() #show the square mask plot
    expanded_img = binary_dilation(img, robot_mask) # image is expanded using the mask we created

    return expanded_img
```

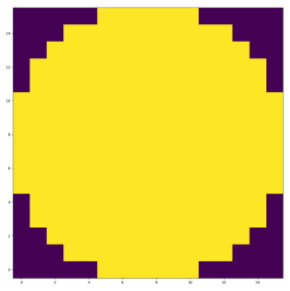


Figure 4: Circular Mask Plot

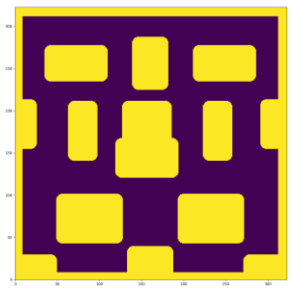


Figure 5: Dilated C-Space map (circular dilation)

2 Waypoint Navigation

2.1 Task B Part 1: Adding Waypoints by Hand

A simple waypoint-based motion planning approach is employed to manually define waypoints around obstacles. While this helps avoid collisions, it does not guarantee an optimal path. The process was carried out iteratively using the following code.

```python
# Create an array of waypoints for the robot to navigate via to reach the goal
# For all navigation tasks, DE NIRO will be navigated
# from a starting position of (0.0, -6.0) m to (8.0, 8.0) m
waypoints = np.array([[0, -6],
                      [0, 0],
                      [8, 8]])
```

Initially, we tested waypoint navigation using only the start and end coordinates, with a midpoint at (0,0). The resulting visualized planned path on the C-space map is presented in Figure 6. A feasible path was manually sketched based on visual inspection (Figure7) nd estimated the corresponding pixel coordinates through the interactive plotting tool in ROS. The waypoint selection was informed by the visibility graph method, ensuring that the path closely follows obstacle edges where appropriate. Table 1 shows the conversion of the pixel coordinates to the world coordinates by subtracting 162 and then dividing by 16.

Table 1: Waypoint Coordinates

Points	Pixel co-ordinates (px)	World co-ordinates (m)
1 (Start)	(162, 66)	(0.0, -6.0)
2	(162, 117)	(0, -2.8)
3	(194,117)	(2.0, -2.8)
4	(194, 290)	(2.0, 8.0)
5 (End)	(290, 290)	(8.0, 8.0)

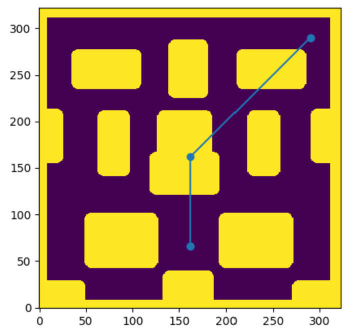


Figure 6: Route Irritation 1

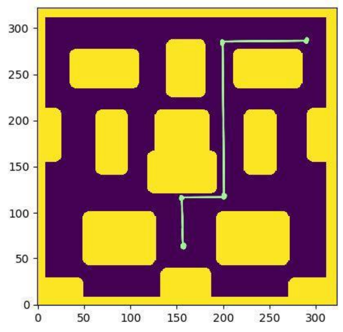


Figure 7: Sketched Waypoint Navigation

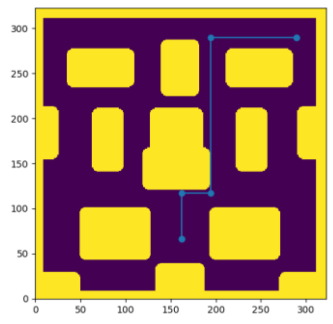


Figure 8: ROS Waypoint Navigation

2.2 Task B Part 2: Finding the Shortest Path

To determine the optimal path for DE NIRO, a small region of the map was selected. Each vertex of the surrounding obstacles was connected, forming a graph representation, as shown in Figure 9. To compute the shortest route, Dijkstra’s algorithm was applied to the visibility graph, identifying ABEGH and ACDGH as the two shortest paths to the goal (two pink routes in Figure 10).

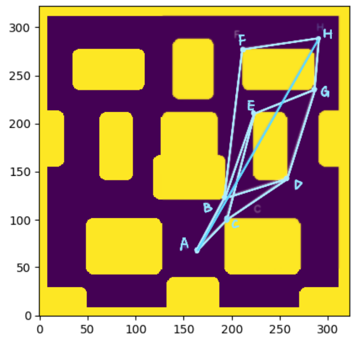


Figure 9:Dijkstra’s Algorithm

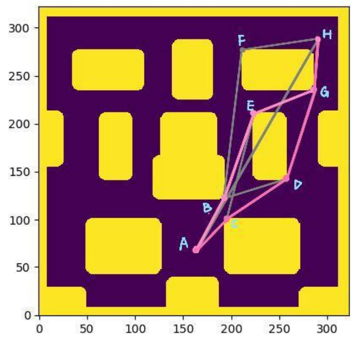


Figure 10:Two Short Paths

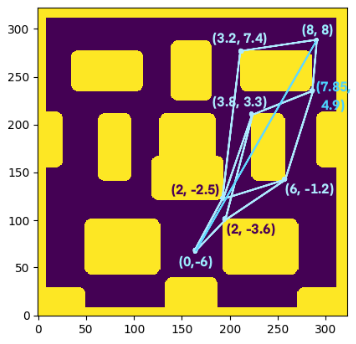


Figure 11: Visibility Graph Coordinates

To determine and validate the shortest path, the total path length must be computed. NumPy-based calculations were added, enabling the measurement of distances between consecutive waypoint vectors, in order to find the shortest path, as demonstrated below.

```
# calculating the length of each individual path between 2 points
path_length_1 = np.linalg.norm((waypoints[1] - waypoints[0]))
path_length_2 = np.linalg.norm((waypoints[2] - waypoints[1]))
path_length_3 = np.linalg.norm((waypoints[3] - waypoints[2]))
path_length_4 = np.linalg.norm((waypoints[4] - waypoints[3]))
path_length_5 = np.linalg.norm((waypoints[5] - waypoints[4]))
# Calculating and printing the total path length
path_length_total = path_length_1 + path_length_2 + path_length_3 + path_length_4 + path_length_5
print("Total path length = ", path_length_total)
```

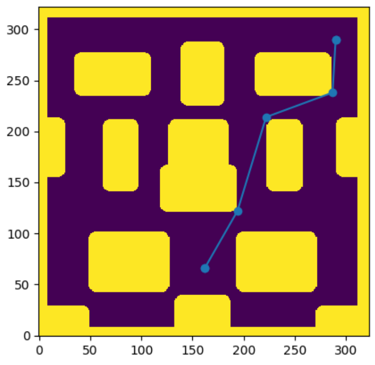


Figure 12: Route ABEGH

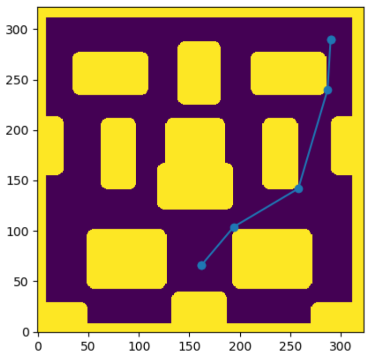


Figure 13: Route ACDGH

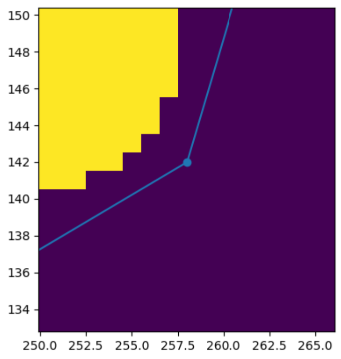


Figure 14: Close up – no collisions

The waypoints for each path were processed, and the results are summarized in Table 2. Although the paths were similar, Route ACEGH was determined to be shorter, with a total length of 17.267 meters.

Table 2: Comparison of two potential optimal routes

Route	Waypoints	Distance
ABEGH	(0,0, 6.0), (2.0, -3.6), (6.0, -1.2), (7.85, 4.9), (8.0, 8.0)	17.562
ACDGH	(0,0, 6.0), (2.0, -2.5), (3.8, 3.3), (7.85, 4.9), (8.0, 8.0)	17.267

3 Potential Field Algorithm

Potential Field Algorithm is a reactive method that works by simulating the agent as a charged particle which is attracted to the goal and repelled by obstacles.This approach has advantages that the trajectory is produced with little computation. However, they can become trapped in local minima of the potential field and fail to find a path or can find a non-optimal path.

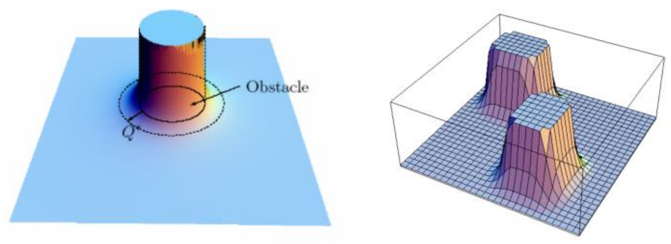


Figure 15: Visual explanation of potential field algorithm

Fatt is the “attractive” potential ——— move to the goal

Frep is the “repulsive” potential ——— avoid obstacles

There are many different strategies to assign potentials for obstacles. They are all based on the distance between the objects and the goal.

A simple strategy for calculating the attractive force is to make it inversely proportional to the robot’s distance from goal.

$$f_{att} = K_{att}\frac{\hat{X}_g}{d_g}$$

K_{att}: The attraction parameter (to be turened).

$\hat{X}_g$: The unit vector from the robot’s current position to the goal point (direction).

d_g: The current distance between the robot to the goal.

These strategies become problematic when numerous obstacles are present or when the goal is far away. In such cases, we aim to establish a linear attractive force, where the goal exerts a constant magnitude of attraction, independent of distance.

$$f_{att} = K_{att} \cdot \hat{X}_g \tag{3}$$

Calculating the repulsive force, making it inversely proportional to the robot’s distance from the obstacles.

$$f_{rep} = -K_{rep}\frac{1}{N}\sum_{i=1}^{N}\frac{\hat{X}_i}{d_i} \tag{4}$$

K_{rep}: The repulsion parameter (to be turened).

$\hat{X}_i$: The unit vector from the robot’s current position to obstacle i (direction).

d_g: The current distance between the robot to obstacle i.

In general, all potential forces can be formulated in the following way:

$$f = K\hat{X} \cdot p(x) \tag{5}$$

Except for the basic one, we have some more complicated potential strategies which will be discussed in Task C part 2. This can be changed to introduce constant, proportional or inversely proportional relationships between the relative distance and force values.

3.1 Task C: Implementing the Potential Field Algorithm

3.1.1 Task C Part 1: Defining force magnitudes

Initially, the expression for the positive attractive force(the force from the goal) and the repulsive force (the force from the obstacles) is written. The attractive potential is constant, as in Equation 3, and the repulsive potential is inversely proportional to distance in Equation 2.

Attractive Force and tuned positive parameter:

```
        # potential function
        pos_force_magnitude =  1  # Attractive force parameter is chosen to be constant
        # tuning parameter
        K_att = 1      # will tune the value to be 1, 10, 100
        # normalised positive force
        positive_force = K_att * pos_force_direction * pos_force_magnitude

```

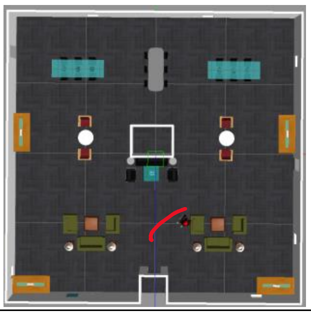
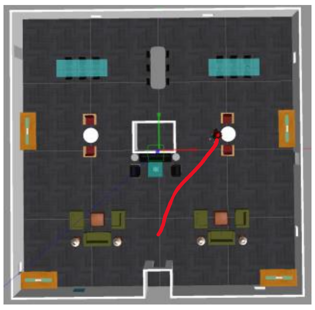
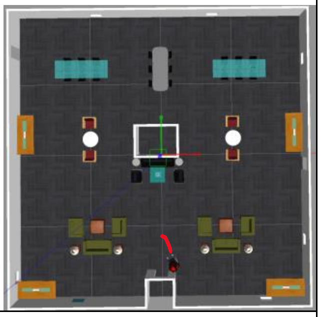
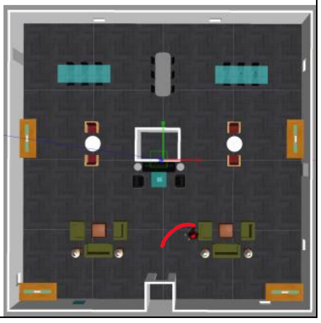
Repulsive Force and tuned negative parameter:

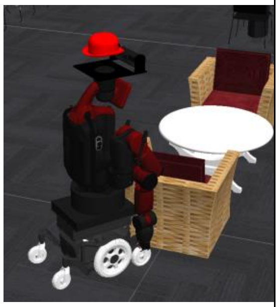
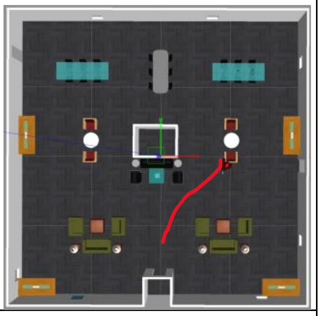
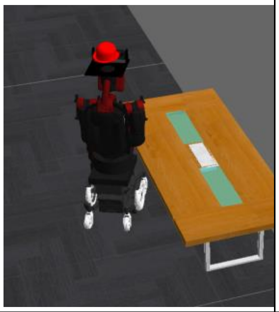
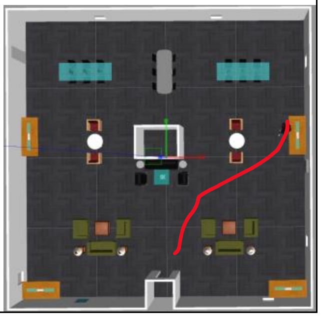
```

        # Repulsive force parameter from each obstacle is inversely proportional to the distance
        force_magnitude = -1.0 / distance_to_obstacle  # Repulsive force magnitude
        # tuning parameter
        K_rep = 15

```

Achieving perfect parameter tuning was highly challenging. Experiments with different values of Katt and Krep were conducted to observe their interactions with various obstacles, as shown in Table 2.

Initially, we broadly explored parameter values. We found that when Katt was very large, the agent moved straight ahead and collided with the green sofa. Conversely, when Katt was small and Krep was very large, the agent turned 180 degrees and exited the room. Through further experimentation, we determined that maintaining an approximate Katt to Krep ratio of 1:15 enabled the agent to navigate more effectively. Each time, we ran the command `python3 set_deniro_pose.py` to reset Deniro to its original position.

Table 3 – Potential Field Values

K_{att}	K_{rep}	Results	Image of DeNiro	Map view
1	100	First, we tested for the default value. Deniro turned 70 degrees and collided with the green sofa. We thought the attractive force might be too small to attract Deniro to the goal. Next time, we increased the value of Katt.		
50	100	We found that Deniro went further and collided with the red sofa. This means the attractive force was strong enough to make it reach the goal. However, the repulsive force was too weak. We wanted to find the optimal ratio of Katt to Krep.		
1	50	As we wanted to control the variables, we kept Katt at 1 and only adjusted Krep. Deniro turned around 150 degrees and hit the right side of the doorframe, which means the repulsive force was too strong. We needed to decrease it.		
1	20	Deniro collided with the green sofa, which means the repulsive force was better than last time (when it moved backward). This shown that we were moving in the right direction.		

1	10	It collided with the red sofa, indicating that the repulsive force is insufficient. Therefore, we increased Krep slightly to improve the balance between attractive and repulsive forces.		
1	15	We conducted multiple adjustments, testing values such as 14.9, 14.7, 14.3, 14.5, 14.55, 15.5, 16.0, 16.5, and 16.7. The best result obtained was a collision with the orange table.		

3.1.2 Task C Part 1: Modifying p(x)

When implementing the algorithm from Part 1, it became evident that Part 1 lacked sufficient adjustments to effectively avoid nearby obstacles

Potential Strategies Analysis:

For linear attractive force, use code:

```
Pos_force_magnitude=1.0
```

For inverse-distance attraction, use code:

```
Pos_force_magnitude=1.0/distance_to_goal
```

The proportional distance attraction form:

```
Pos_force_magnitude = distance_to_goal
```

For the repulsive function：

Softer avoidance, use code:

```
Force_magnitudes=-1.0/distance_to_obstacle
```

Strong Avoidance near obstacles, use code:

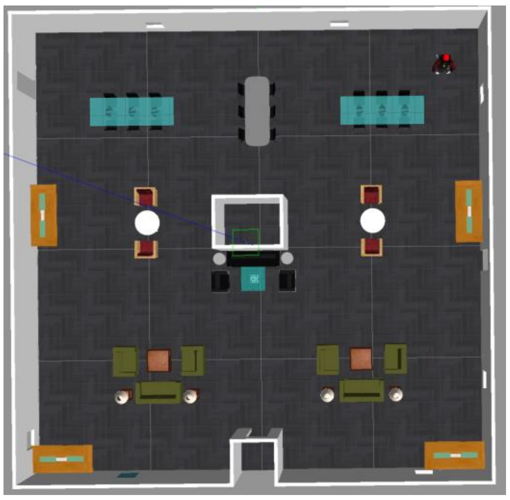
```
Force_magnitudes =-1.0/ (distance_to_obstacle**2)
```

In this case, the inverse proportional function did not sufficiently amplify the repulsive forces of nearby objects. To enhance this effect, an additional squared term was introduced to further increase repulsive force for closer objects. Consequently, the force magnitude was modified as follows:

```
force_magnitude = -1.0/ (distance_to_obstacle**2)
```

The parameters that achieved a completer path were Katt =1 and Krep = 18 and the path can be seen below.

```python
        # Repulsive force parameter from each obstacle is inversely proportional to the distance
        #Strong Avoidance near obstacles
        force_magnitude = -1.0 / (distance_to_obstacle**2)  # Repulsive force magnitude
        # tuning parameter
        K_rep = 18

        # force from an individual obstacle pixel
        obstacle_force = force_direction * force_magnitude
        # total negative force on DE NIRO
        negative_force = K_rep * np.sum(obstacle_force, axis=0) / obstacle_pixel_locations.shape[0]
```



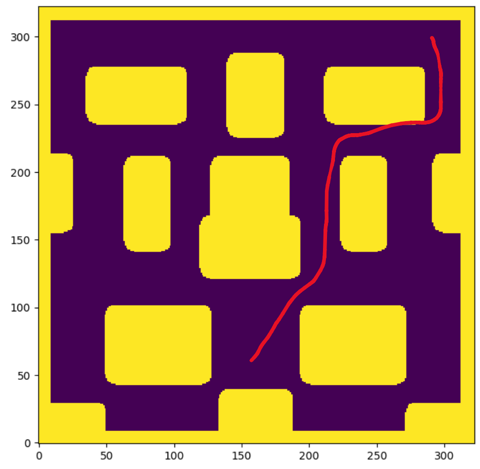


Figure 16 & 17 – Map view of completed path and image of DE NIRO in goal position

The path appeared to stay very close to the obstacles. DE NIRO remained to the left of the second obstacle, as the goal exerted a stronger pull in the y-direction than in the x-direction. The approximate length of the path was 20 meters. This path was not as short as the waypoints in Task B for two reasons.

First, curved paths tend to add unnecessary length, whereas straight paths are more efficient and faster. Second, this behavior represented a locally optimal decision rather than a globally optimal path. The waypoint-based approach resulted in a path that was 2.5 meters shorter than the potential field approach.

4 Probabilistic Roadmap

The Probabilistic Roadmap (PRM) is a sampling-based motion planning method. It is slower to compute than the potential field algorithm but probabilistically complete. This means that, given enough time, PRM is guaranteed to find an optimal path if one exists.

A PRM is constructed through the following steps:

1. **Randomly Sampling from the Map (Task D):** The process begins by randomly sampling coordinates within C-space to explore possible configurations. Each sampled configuration is then evaluated for collisions with obstacles, and only collision-free configurations are retained as milestones.
2. **Creating the Graph (Task E):** The milestones are connected to their nearest neighbours using straight-line paths. Each link is collision checked and any invalid links are removed. The remaining collision-free links together form a roadmap, creating the PRM graph, which represents feasible paths through the environment.

These two steps are discussed in detail in the following section.

4.1 Task D: Randomly Sampling from the Map

4.1.1 Task D Part 1: Uniform Sampling Method

The objective of this task is to generate N_{point} samples that do not collide with any obstacle in the C- space. To achieve this, a uniform sampling method was employed, where each generated point has an equal probability of appearing anywhere in C-space. A while loop is used to iteratively generate random points until the required number is reached. In each iteration, the loop produces a set of random points equal to the number of remaining samples needed outside the obstacle region ($N_{point} – N_{accept}$).

A for loop within the while loop iterates through each generated point to determine if it collides with an obstacle. This is achieved by checking whether the corresponding C-space value for the point is 1, indicating a collision. If a collision is detected, the corresponding entry in the rejection flag array is updated to 1. Otherwise, the point is added to the set of accepted points and the N_{accept} counter is incremented accordingly.

The following code snippet was used for the for loop:

```python
# Loop through the generated points and check if their pixel location corresponds to an obstacle in self.pixel_map
for i in range(len(pixel_points)):
    # get the point location (integer) x and y on our map seperately
    px_y = int(pixel_points[i, 1])
    px_x = int(pixel_points[i, 0])
    # self.pixel_map[px_y, px_x] = 1 when an obstacle is present
    # Check if the generated point is inside an obstacle
    if self.pixel_map[px_y,px_x] == 1:
        # If yes, set the corresponding entry in the rejection flag array to 1
        rejected[i] = 1
```

Figure 18 presents the plot of the accepted and rejected points after the required number of points outside the obstacle was achieved ($N_{point} = N_{accept}$).

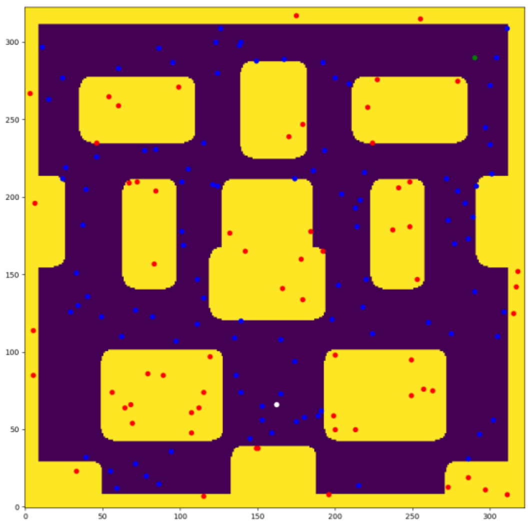


Figure 18: Plot of the accepted (labelled blue) and rejected (labelled red) points on the map

4.1.2 Task D Part 2: Bridge Test Sampling

In the previous section, a uniform sampling method was used. While this approach is simple and easy to implement, it becomes ineffective in regions with a high obstacle density. Since points are evenly distributed across C-space, many will collide with obstacles. Therefore, an alternative approach, Bridge Test Sampling, should be employed to improve sampling efficiency.

Bridge Test Sampling is particularly effective in narrow passages. It generates two random points as endpoints and then calculates their midpoint. If both endpoints lie within an obstacle while the midpoint remains collision-free, the midpoint is accepted as a new node. This method increases sampling density in narrow passages, thereby improving the connectivity of the generated roadmap [2].

Figure 19 shows the process of building a short bridge in narrow passages. The midpoint is positioned over the free space, indicating that it has successfully passed the bridge test and can be accepted as a milestone in the roadmap.

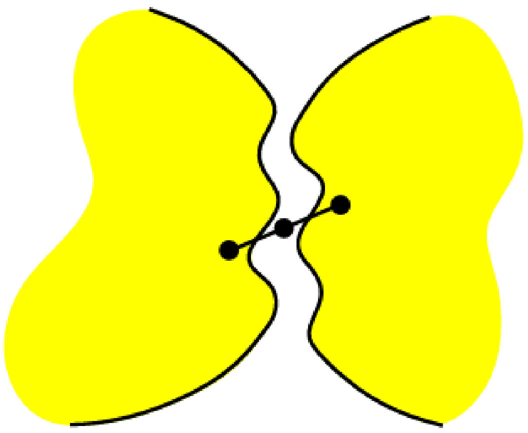


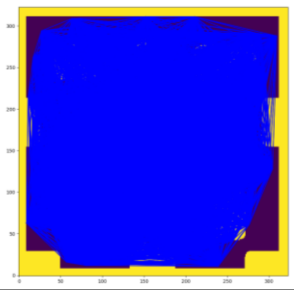
Figure 19: Building the short bridge in narrow passages [2]

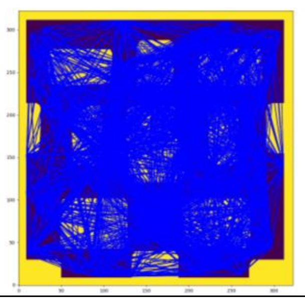
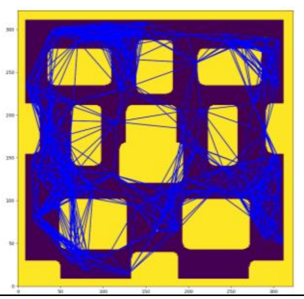
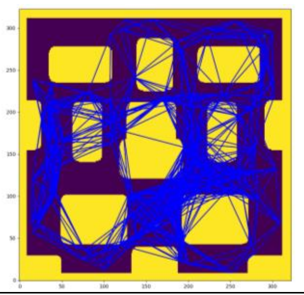
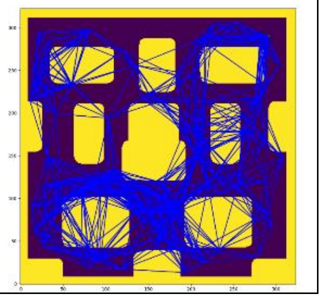
4.2 Task E: Creating the Graph

4.2.1 Task E Part 1: Edge Tuning

Once the sampling process is complete, a roadmap can be constructed. The generated points serve as nodes, with edges connecting them based on a predefined distance threshold. This threshold must be carefully balanced to ensure effective path planning. If it is too large, there is a high probability that an obstacle lies between the points. Conversely, if it is too small, the connections become impractical for navigation. As shown in Table 3, the minimum and maximum distances between points have been fine-tuned to 0.5 and 5.0, respectively, to achieve an optimal balance.

Table 3: Tuning the distance threshold for node connection

Min Distance	Max Distance	Description	Plot
0.0	20.0	Initially, the minimum and maximum allowed distances for links were set to 0.0 and 20.0 respectively. However, this resulted in an overly dense network with frequent collisions. To address this problem, the maximum distance was prioritised for tuning first.	

0.0	10.0	The maximum distance was reduced to 10.0, but the network remained overly dense, requiring further reduction.	
0.0	5.0	The maximum distance was further reduced to 5.0, which significantly decreased collisions. As a result, 5.0 was selected as the final value. However, the minimum distance was too small, leading to unnecessary connections, and thus had to be increased.	
1.0	5.0	The minimum distance was increased to 1.0, resulting in a longer connection. However, the number of edges generated was not large enough to find an optimal path. The value needs to be reduced.	
0.5	5.0	The minimum distance was adjusted to 0.5 to provide a sufficient number of edges while filtering out unnecessarily short links. As a result, 0.5 was chosen as the final value for the minimum distance.	

4.2.2 Task E Part 2: Collision Checking

It is important to note that even after fine-tuning the distance threshold to an optimal value, edges may still collide with obstacles. To prevent this, each edge must be systematically checked for collisions at an appropriate resolution.

Collision checking requires both the direction vector and the distance between nodes. NumPy library in Python was employed to compute these values efficiently. The vector between two nodes points from point A to point B, while the distance is determined by its magnitude. Since a unit vector only represents direction, it is obtained by normalising the vector - dividing it by its magnitude.

The following code snippet was used for the calculation:

```
 # Calculate the distance between the two point
vector = np.subtract(pointB, pointA)  # vector of point A to point B
distance = np.linalg.norm(vector)  # distance = mangitude of vector from A to B
# Calculate the UNIT direction vector pointing from pointA to pointB
direction = vector / distance # unit vector = vector / magnitude
# Choose a resolution for collision checking
resolution = 0.5   # resolution to check collision to in m
```

The resolution value must be carefully tuned for effective collision checking. If it is set too high, colliding edges may not be detected and removed. Conversely, if it is too low, the computational time increases significantly. The tuning process began with a resolution value of 2, which was iteratively halved until a suitable value was found. Finally, 0.05 was selected as the optimal resolution, as it effectively removes all colliding edges while maintaining efficient processing time.

Figures 20 to 23 show the results of collision checking for resolution values of 2 and 0.05 respectively. A zoomed-in view was provided to illustrate detailed collision detection.

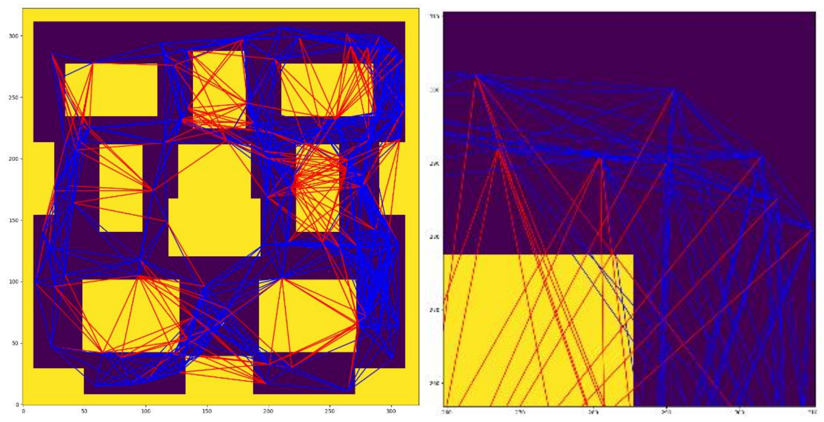


Figure 20 & 21: Roadmap generated using a resolution value of 2, presented with both an overall view and a zoomed-in view. Blue lines represent accepted edges, while red lines represent rejected edges

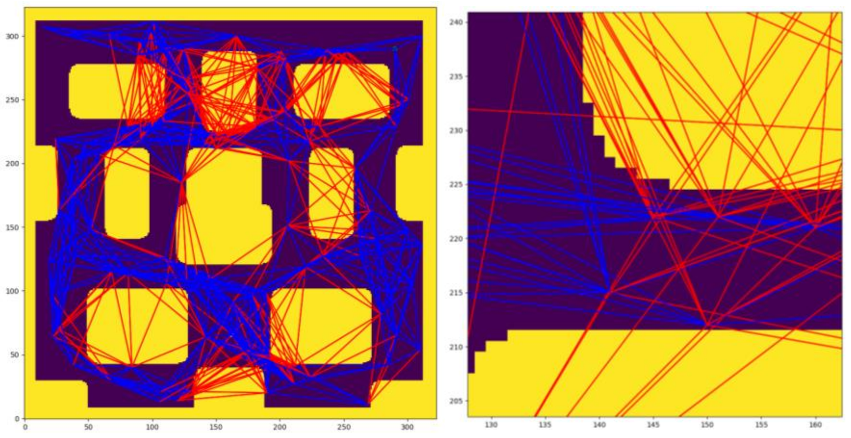


Figure 22 & 23: Roadmap generated using a resolution value of 0.05, presented with both an overall view and a zoomed-in view. Blue lines represent accepted edges, while red lines represent rejected edges

4.2.3 Task E Part 3: Searching Near Edges for Suitable Locations For New Nodes

A common problem in PRM is that there are not enough edges to form a complete graph connecting the initial node to the goal node. While increasing the number of sampling nodes at the start may help, it is computationally expensive and not always effective. Therefore, a more efficient approach is needed to identify suitable locations near existing edges for additional nodes.

The adopted method, edge-based resampling, is applied after edge collision detection. For each failed edge, the algorithm evaluates its midpoint. If the midpoint is collision-free, it is accepted as a new milestone. This method effectively adds new nodes while maintaining an optimal distance from existing ones. By creating new nodes between obstacles, it densifies the sampled graph and increases the number of collision-free edges. As a result, it significantly improves the likelihood of generating a complete path from the initial to the goal node.

5. Task F: Dijkstra's Algorithm

The PRM graph presents numerous possible routes between the initial to goal position. To find the shortest path, the Dijkstra's algorithm is implemented as follows.

5.1 Task F Part 1: Runing Dijkstra's Algorithm

5.1.1 Explanation of the Implementation

Step 1 of Dijkstra's algorithm initializes the unvisited nodes data frame, which tracks the cost and path for each node. Two data frames are created: one for unvisited nodes and another for visited nodes. Each entry in the unvisited data frame includes three columns: 'Node' (the node identifier), 'Cost' (the total distance to reach the node), and 'Previous' (the preceding node in the path). The first data frame includes all unvisited nodes, with their costs initially set to a very high value (1e6) which simulates infinity, except for the initial node, which has a cost of 0. The visited data frame starts empty, as no nodes have been processed. Once they are initialized, the initial state of both the unvisited and visited data frames is displayed to verify the setup before running the algorithm.

```python
def dijkstra(self, graph, edges):

    ############################################################### TASK F

    goal_node = goal  # Define the goal node

     # Step 1: Create a dataframe of unvisited nodes
    nodes = list(graph.keys())  # List all nodes in the graph

    # Initialise each cost to a very high number
    initial_cost = 1e6  # Large number to represent infinity

    # Create a dataframe for unvisited nodes with three columns: 'Node', 'Cost', and 'Previous'
    unvisited = pd.DataFrame(
        {'Node': nodes,  # List of all nodes in the graph
        'Cost': [initial_cost for node in nodes],  # Initialize all node costs to a very high value (infinity)
        'Previous': ['' for node in nodes]})# Track the previous node in the optimal path
    # Set 'Node' as the index to allow direct lookups and updates using node names
    unvisited.set_index('Node', inplace=True)

    # Set the first node's cost to zero
    unvisited.loc[[str(initial_position)], ['Cost']] = 0.0

    # Create a dataframe of visited nodes (initially empty)
    visited = pd.DataFrame({'Node':[''], 'Cost':[0.0], 'Previous':['']})
    visited.set_index('Node', inplace=True)

    # Display initial state of dataframes
    print('--------------------------------')
    print('Unvisited nodes')
    print(unvisited.head())
    print('--------------------------------')
    print('Visited nodes')
    print(visited.head())
    print('--------------------------------')
    print('Running Dijkstra')
```

In Step 2, the node with the lowest cost in the unvisited data frame is selected as the current node. This node represents the next step in the path with the smallest accumulated cost from the starting position.

```python
    # Dijkstra's algorithm!
    # Step 2: Run Dijkstra's algorithm until the goal node is reached
    while str(goal_node) not in visited.index.values:

        # Select the node with the minimum cost from the unvisited nodes
        current_node = unvisited[unvisited['Cost']==unvisited['Cost'].min()]
        current_node_name = current_node.index.values[0]    # the node's name (string)
        current_cost = current_node['Cost'].values[0]       # the distance from the starting node to this node (float)
        current_tree = current_node['Previous'].values[0]   # a list of the nodes visited on the way to this one (string)

        # Get all connected nodes and their edge costs
        connected_nodes = graph[current_node.index.values[0]]   # get all of the connected nodes to the current node (array)
        connected_edges = edges[current_node.index.values[0]]   # get the distance from each connected node to the current node
```

In Step 3, the algorithm evaluates each unvisited node connected to the current node. For each connection, it calculates a potential new cost (next_cost_trial) by adding the cost of reaching the current node to the edge cost between the current and connected nodes. If this new cost is lower than the previously recorded cost for the connected node, the algorithm updates both the cost and the path. The path is updated to include the sequence of nodes taken to reach the current node to ensure that the algorithm always maintains the shortest known path to each node.

```
 # Step 3: Loop through connected nodes to update costs if a shorter path is found
for next_node_name, edge_cost in zip(connected_nodes, connected_edges):
    next_node_name = str(next_node_name)    # the next node's name (string)

    if next_node_name not in visited.index.values:  # if we haven't visited this node before

        # update this to calculate the cost of going from the initial node to the next node via the current node
        next_cost_trial = current_cost + edge_cost # set this to calculate the cost of going from the initial node to the next node
via the current node
        next_cost = unvisited.loc[[next_node_name], ['Cost']].values[0] # the previous best cost we've seen going to the next node

        # if it costs less to go the next node from the current node, update then next node's cost and the path to get there
        if next_cost_trial < next_cost:
            unvisited.loc[[next_node_name], ['Cost']] = next_cost_trial
            unvisited.loc[[next_node_name], ['Previous']] = current_tree + current_node_name    # update the path to get to that node
```

In Step 4, the current node is removed from the unvisited data frame and added to the visited data frame. This ensures the node is not considered in future updates, thereby preventing revisit of previous nodes.

```
# Step 4: Move current node from unvisited to visited
unvisited.drop(current_node_name, axis=0, inplace=True)     # remove current node from the unvisited list
visited.loc[current_node_name] = [current_cost, current_tree]   # add current node to the visited list
```

After finalising the node states, the visited nodes and unvisited nodes are then displayed for verification. The optimal path is retrieved then plotted on the map, with the start and goal positions highlighted. Finally, the waypoints are set up for navigation, and the motion planning process is completed.

5.1.2 The Result of Dijkstra's Algorithm

The result paths are similar to the graph planned in task B which demonstrates the capability of the algorithm to closely approximate human route-planning strategies. However, none of the path planned achieved the same or shorter path compared to task B (17.27m). This is because Dijkstra's algorithm finds the optimal path regarding a pre-planned graph generated by the PRM mapping, and the optimality is therefore constrained by the proximity between the sampled nodes and the optimal nodes.

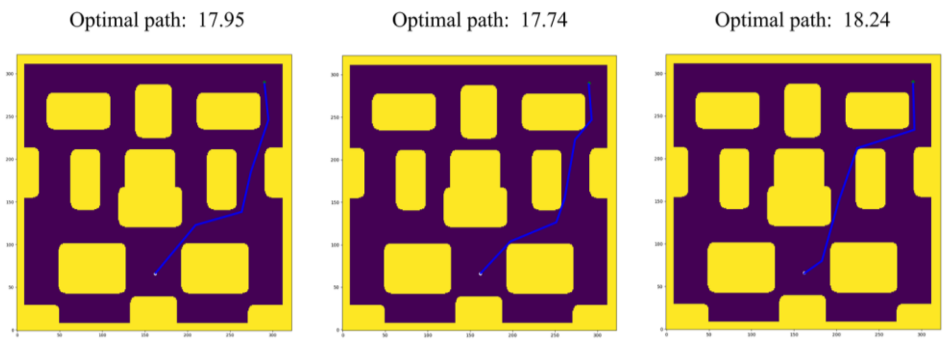


Figure 24 & 25 & 26: Planned path results when sample size N = 100

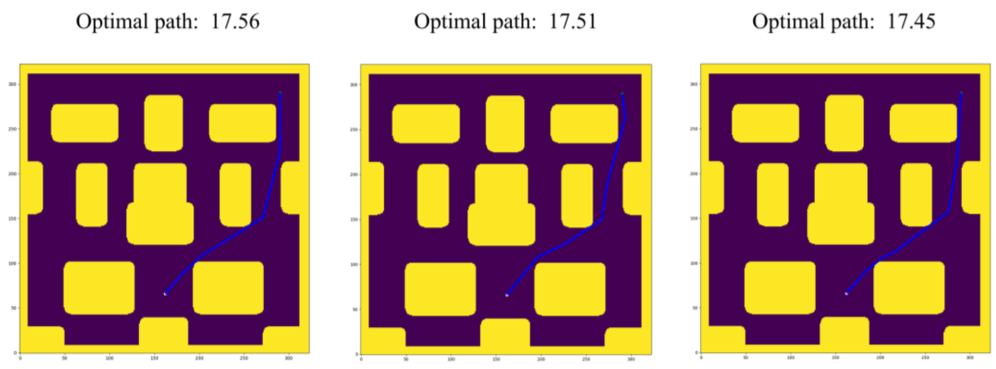


Figure 27 & 28 &29: Final path results when sample size N = 300

By increasing the number of sampled nodes from 100 to 300, the algorithm takes significantly longer time to compute. The mean length of the path planned when N=100 is **17.98m**, while that of N = 300 is **17.51m**. The reduction of **0.47m** shows improvement in the path planned. This is because, with a greater sample rate, the point plotted is closer to the corner of the obstacles and therefore closer to the optimal waypoints.

However, increasing the sample size to the pixel size of the map is impossible considering the enormous time consumption and computational force needed to compute the path. Therefore, finding the optimal path by just increasing the sample size is impossible in practice. To reduce the computational force required, a non-uniform (biased) sampling method can be used. The optimal waypoints determined by both the human brain (Task B) and Dijkstra's algorithm are highly close to the corner of the obstacles. By utilising this, the sampled point can be biased toward the edge of the obstacle to achieve greater possibilities of sampling optimal waypoints.

Nonetheless, the path planned in PRM and Dijkstra's method are straight lines with sharp turning on waypoints before moving to the next ones. This although achieves the shortest distance, is not necessarily the most efficient method, considering the time wasted turning to the next direction when staying on the waypoints. Therefore, extra processing of smoothing the generated path can be applied to allow turning to the next direction when travelling on the curved route without stopping midway. The trade-off between extra time taken on a curved path and the time consumed by turning at a point can be further tuned to plan the best route in practice.

5.2 Task F Part 2: One-to-many and One-to-one Planning

The advantages of a one-to-many planning algorithm, such as Dijkstra's algorithm, include its ability to efficiently solve multiple motion planning problems after an initial preprocessing. Once the c-space is pre-processed by generating the PRM or other mapping methods, different path planning problems can be solved by connecting different start to goal configurations and using Dijkstra's algorithm to find a good path through the pre-planned graph. This approach enables efficient and flexible motion planning across multiple queries without needing to regenerate the entire roadmap. However, the new goal needs to be readily sampled and can be retrieved from the visited dataframe, and the starting point should not be changed. If the new goal and starting points are unsampled or any obstacle is changed, the mapping needs to be re-processed, otherwise, the graph can be reused by running the Dijkstra's algorithm to the new goal.

While Dijkstra's algorithm is effective for solving multiple motion planning problems, the Rapidly-exploring Random Tree (RRT) is better suited for single-query scenarios. Unlike PRM, which constructs a roadmap in advance for repeated use, RRT incrementally builds a tree from the starting point to efficiently explore the space until it reaches the goal. The pseudocode by step of implementation is below:

```
GENERATE_RRT(x_init, K, Δt)
1  T.init(x_init);
2  for k = 1 to K do
3      x_rand ←RANDOM_STATE();
4      x_near ←NEAREST_NEIGHBOR(x_rand, T);
5      u ←SELECT_INPUT(x_rand, x_near);
6      x_new ←NEW_STATE(x_near, u, Δt);
7      T.add_vertex(x_new);
8      T.add_edge(x_near, x_new, u);
9  Return T
```

The process begins by setting DENIRO’s current position as the root of the tree. A random target position within its operational space is selected to ensure broad path exploration. The algorithm then identifies the closest node in the existing tree to the selected random state, determining DENIRO’s nearest reachable configuration. To extend the tree, a potential movement toward the sampled position is generated while fixing to step size constraints to maintain stable and controlled motion.

Before integrating the new state, a collision check is performed to prevent movement through obstacles. If the new node is valid, it is added to the tree and connected to its nearest neighbor. This iterative process continues, expanding the tree with validated movements until the robot reaches a position close to the goal. Once the target is within range, it is linked to the tree, completing the motion plan. Finally, the system backtracks from the goal to the start, extracting the optimal sequence of movements for DENIRO to follow.

RRT is highly efficient for single-query motion planning because it does not require precomputing a roadmap. Instead, it constructs a path on demand, making it more effective for scenarios where a single problem needs to be solved quickly. The algorithm is particularly well-suited for exploring large and high-dimensional spaces, as its structure allows it to rapidly expand in an unstructured environment. Since RRT grows incrementally, it minimizes computational overhead while adapting dynamically to new obstacles and environmental changes. However, RRT does not guarantee the most optimal path, as it prioritizes rapid exploration over path efficiency, often resulting in solutions that are longer or less smooth compared to graph-based algorithms like Dijkstra’s.

Reference

[1] Valerie Boor, Mark H. Overmars, and A. Frank van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In 1999 IEEE International Conference on Robotics & Automation, Detroit, Michigan, May 1999. IEEE.

[2] Hsu D, Jiang T, Reif JH, Sun Z. The bridge test for sampling narrow passages with probabilistic roadmap planners. 2003 Nov 10;