Assignment 2 Report

Testing commands used:

component_1.py

```
python component_1.py
```

nearest_neighbors.py

```
python nearest_neighbors.py --robot arm --target 0 0 -k 3 --configs config1.txt
python nearest_neighbors.py --robot freeBody --target 7 2 1.57 -k 3 --configs config.txt
```

collision_checking.py

```
python collision_checking.py --robot arm --map environment_2.txt
python collision_checking.py --robot freeBody --map environment_5.txt
```

PRM: prm.py

```
python prm.py --robot arm --start 0 0 --goal 1.57 1.57 --map environment_1.txt
python prm.py --robot arm --start -2.8 3.14 --goal 0 0 --map environment_2.txt

python prm.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.0 --map  environment_5.txt
python prm.py --robot freeBody --start -5 9 0 --goal 5 -2.5 1.57 --map  environment_4.txt
```

RRT: rrt.py

```
python rrt.py --robot arm --start 0 0 --goal 1.57 1.57 --goal_rad 0.2 --map environment_1.txt
python rrt.py --robot arm --start -2.8 3.14 --goal 0 0 --goal_rad 0.2 --map environment_2.txt

python rrt.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.7 --goal_rad 0.2 --map
environment_5.txt
python rrt.py --robot freeBody --start -5 9 0 --goal 5 -2.5 .98 --goal_rad 0.2 --map
environment_4.txt
```

RRT*: AO_planner.py

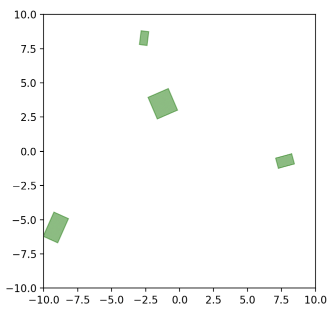
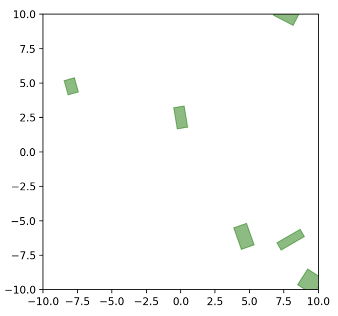
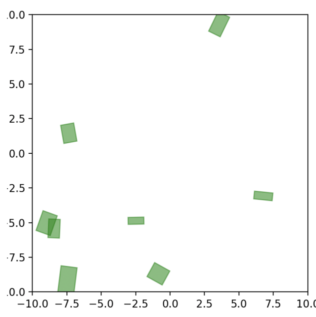
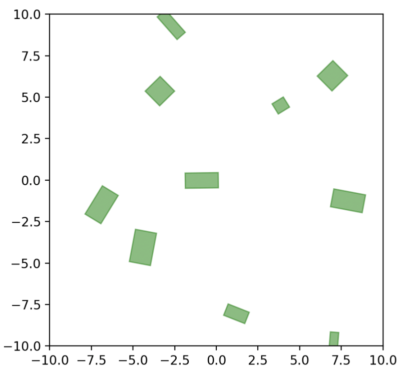
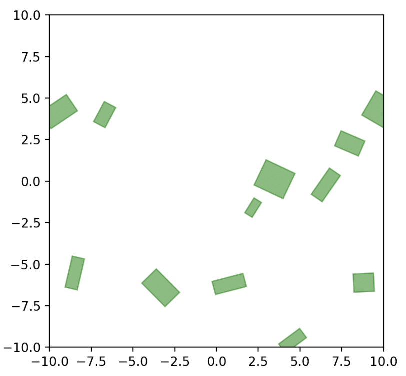
```
python AO_planner.py --robot arm --start 0 0 --goal 1.57 1.57 --goal_rad 0.2 --map environment_1.txt
python AO_planner.py --robot arm --start -2.8 3.14 --goal 0 0  --goal_rad 0.2 --map environment_2.txt

python AO_planner.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.0 --goal_rad 0.2 --map environment_5.txt
python AO_planner.py --robot freeBody --start -5 9 0 --goal 5 -2.5 .98 --goal_rad 0.2 --map environment_4.txt
```

RRT car: rrt_car.py

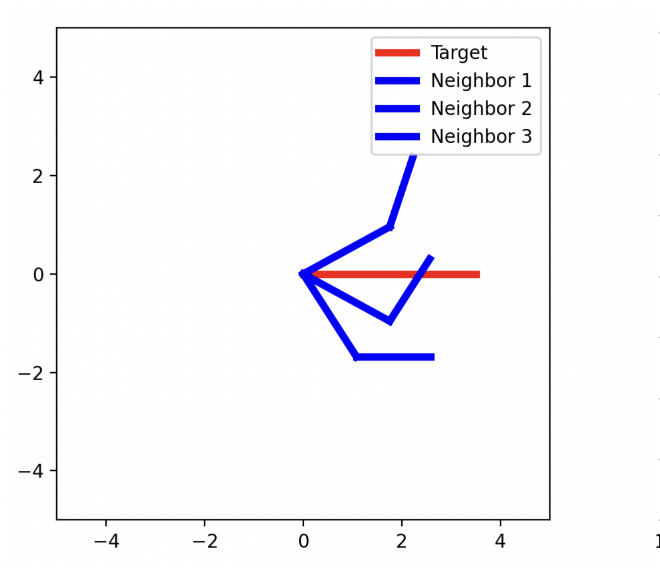
```
python rrt_car.py --start -5 -5 0 --goal 8.5 8.5 2.0 --goal_rad 0.2 --velocity 1 --steering_angle .9 --map  environment_5.txt
python rrt_car.py --start -5 9 0 --goal 5 -2.5 1.57 --goal_rad 0.2 --velocity 1 --steering_angle .9 --map  environment_4.txt
```

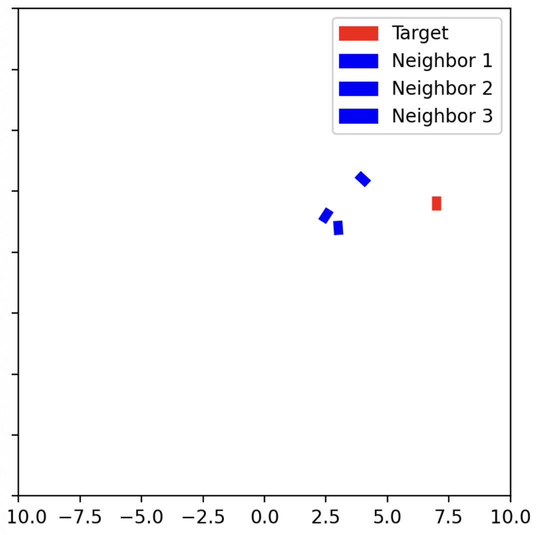
1 - Generating Environments:

Scene 1-5 in order from left to right

2 - Nearest neighbors:





0.5 0.75	1.0 2.0 0.5
1.0 1.25	3.0 1.0 1.5
0.0 2.0	2.5 1.5 -1.0
-0.5 1.5	0.0 0.0 0.0
1.5 0.5	4.0 3.0 0.75
-1.0 1.0	

2. Nearest Neighbors with Linear Search (5%)

The Python script nearest_neighbors.py determines the closest configurations for two types of robots: the "arm" and the "freeBody."

Distance Metric

- Arm Robot: Uses Euclidean distance between joint angles:

- FreeBody Robot: Uses weighted Euclidean distance for position and orientation:

Naive Linear Search Approach

- The script iterates through all configurations, computes distances to the target, and keeps track of the k closest configurations. This simple approach works well for a small number of configurations.

Visualization

Arm Robot: Visualizes the arm in a 2D plane with the k nearest configurations overlaid.

FreeBody Robot: Shows position and orientation in a 2D plane with the k nearest configurations.

Visualizations are attached in zip and shown above

3 - Collision Detection:

The collision_checking.py script implements collision detection for the arm and freeBody robots in a given environment.

Collision Detection Implementation

The approach used for collision checking involves spawning the robot in the environment and determining if it overlaps with any obstacles. The robot is spawned at random poses every second for a total of 10 poses. Each pose is checked against the obstacle positions to detect collisions.

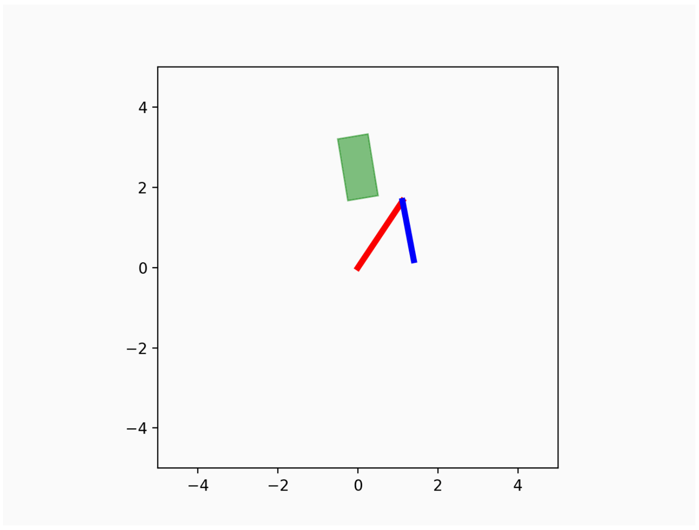
- Arm Robot: The arm is modeled as a series of line segments, and collision checking involves determining if any of these segments intersect with the obstacles. This is done by calculating intersections between the arm's segments and the rectangular obstacles.

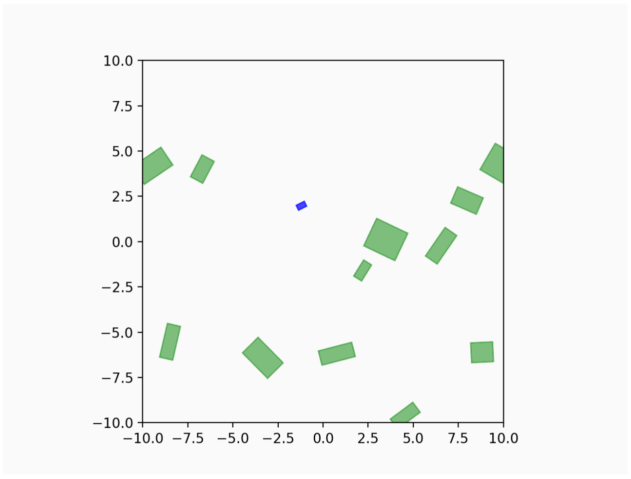
- FreeBody Robot: For the freeBody robot, the collision check determines if the bounding box of the robot overlaps with any of the obstacles in the environment.

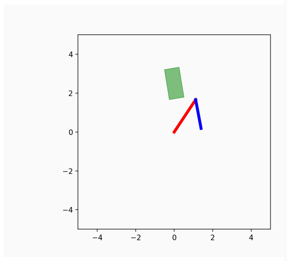
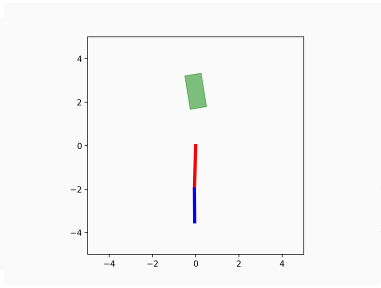
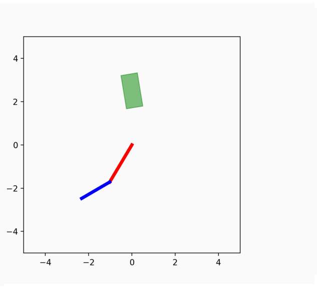
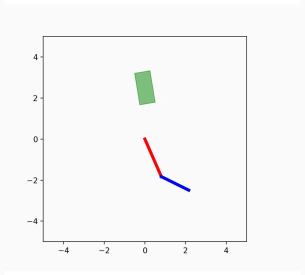
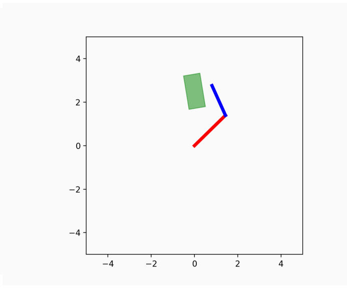
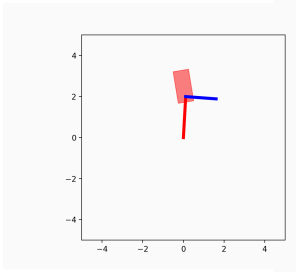
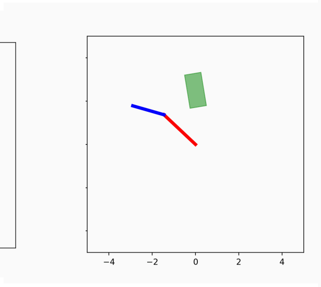
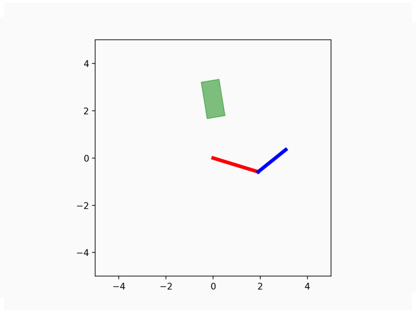
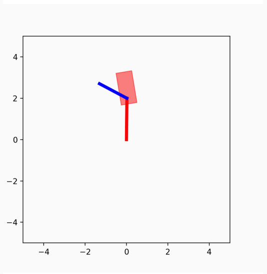
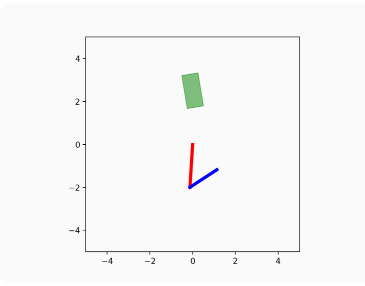
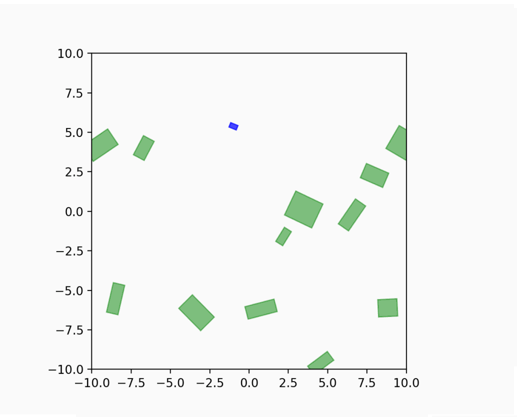
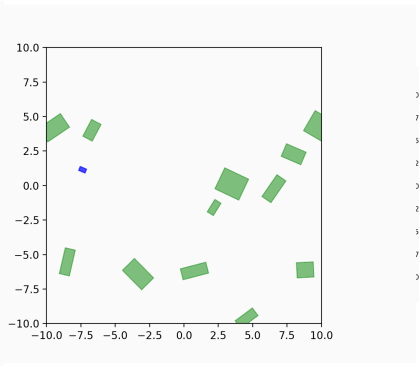
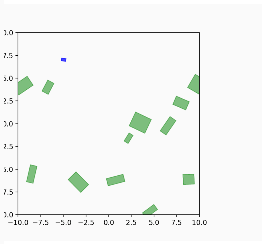
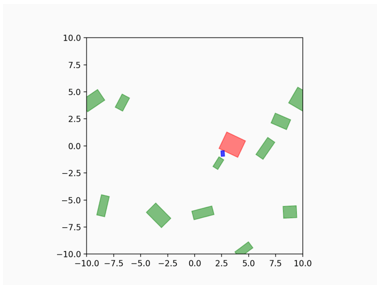
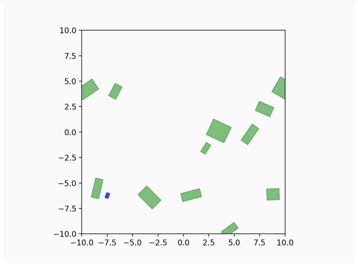
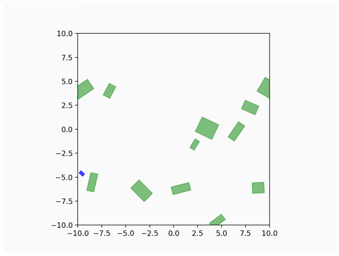
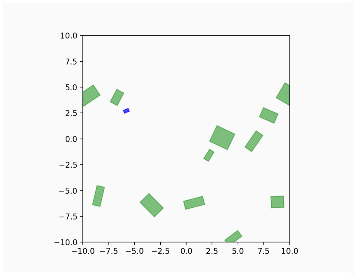
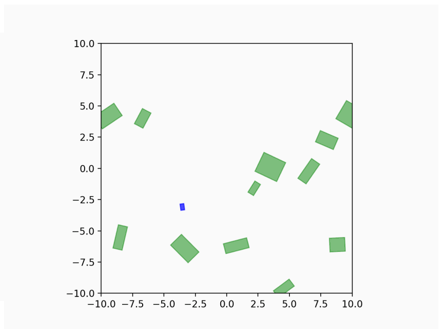
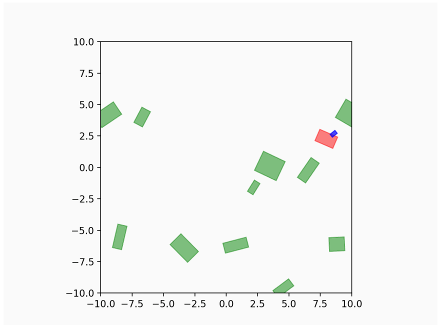
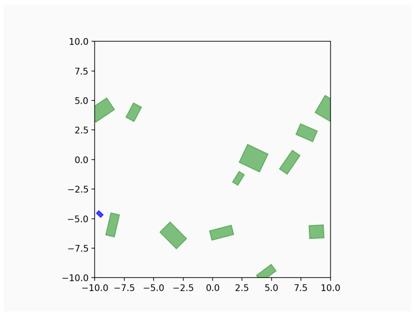
If a collision is detected, the colliding obstacles are marked in red, while the non-colliding obstacles are marked in green.

Visualization

The script includes a visualization feature that shows the environment, the robot's pose, and the obstacles, which can be seen below and in report as gif






4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

4 2 0 −2 −4
−4 −2 0 2 4

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

10.0 7.5 5.0 2.5 0.0 −2.5 −5.0 −7.5 −10.0
−10.0 −7.5 −5.0 −2.5 0.0 2.5 5.0 7.5 10.0

4.1 PRM Implementation

The PRM algorithm follows these main steps:

- Sampling Configurations: Random configurations are generated within the robot's configuration space. Configurations are rejected if they are in collision with obstacles.

- Building the Roadmap: The algorithm builds a graph where each node represents a collision-free configuration, and edges represent feasible paths between configurations.

- Connecting Neighbors: For each new node, the algorithm connects it to its k nearest neighbors (using a distance metric similar to the one described in the nearest neighbors component) if a direct path exists that avoids obstacles.

- Pathfinding: After constructing the graph, Dijkstra's or A* search is used to find a path from the start configuration to the goal configuration.

The PRM implementation uses a maximum of 5000 nodes for the roadmap and a constant number of neighbors (k = 6) for connecting nodes.

Visualization

The script generates two visualizations which are included in the report:

- Roadmap Visualization: Shows the nodes and edges of the PRM alongside the obstacles. Nodes are represented as circles, and edges as lines connecting nodes. Obstacles are displayed in the environment for reference.

- Solution Path Animation: Displays the robot moving from the start configuration to the goal configuration along the solution path. The solution path is shown without displaying the entire roadmap.

4.2 RRT Implementation

The RRT algorithm follows these main steps:

- Tree Initialization: The tree is initialized with the start configuration as the root node.

- Growing the Tree: The algorithm iteratively adds nodes to the tree:
    - A random configuration is sampled in the configuration space.

        - The nearest existing node in the tree is found using the appropriate distance metric.
        - A new node is added by moving a small step from the nearest node towards the random sample. This helps to gradually grow the tree in the direction of unexplored areas.
        - Occasionally, the goal configuration is sampled (5% of the time) to bias the tree growth towards the goal.
        - Nodes are added to the tree as long as they do not result in collisions with obstacles.
    - Termination: The search stops after adding 1000 nodes or if a valid path to the goal is found.
    - The RRT implementation also considers goal biasing to improve the likelihood of finding a path to the goal within the limited number of iterations.

Visualization

The script generates two visualizations upon completion which can be seen in the zip file:

- Tree Growth Animation: Displays the RRT tree growing over time, with nodes representing the sampled configurations and edges representing connections between nodes.

- Solution Path Animation: Shows the robot moving from the start configuration to the goal configuration along the found path, without any collisions with obstacles.

5. RRT* Implementation

The RRT* algorithm follows these main steps:

- Tree Initialization: The tree is initialized with the start configuration as the root node.

- Growing the Tree: The algorithm incrementally grows the tree:
    - A random configuration is sampled in the configuration space.
    - The nearest node in the tree is found, and a new node is added by moving a small step towards the sampled configuration. The new node is added only if the path is collision-free.
    - After adding a new node, the algorithm re-evaluates connections to neighboring nodes within a radius to minimize path cost. This rewiring step ensures that the path from the root to each node remains optimal.
- Termination: The algorithm stops after a fixed number of iterations (e.g., 1000) or when the goal configuration is reached and further improvements are minimal.

Differences from RRT

The key difference between RRT and RRT* is the rewiring step. In RRT*, each new node evaluates whether it can provide a lower-cost path to existing nodes within a certain radius. This makes RRT* capable of finding asymptotically optimal paths, unlike RRT which aims only for feasibility.

Visualization

The script generates two visualizations upon completion which are visible in the zip file:

- Tree Growth and Rewiring Animation: Displays the RRT* tree growing and rewiring over time. Nodes and edges are shown in the configuration space, highlighting how the tree is adjusted to find optimal paths.

- Solution Path Animation: Shows the robot moving from the start configuration to the goal configuration along the optimal path found by RRT*.

Visualizations for two different environments are included in the report, each with two different start and goal pairs. These visualizations help illustrate the improvements made by RRT* in finding more efficient paths compared to standard RRT.

6. Comparison of Planners

Success Rate

- **PRM**: The PRM algorithm builds a roadmap of the configuration space before attempting to find a path. Its success rate depends on the density of the nodes and the complexity of the environment. In challenging environments with many obstacles, the PRM may struggle if the roadmap does not adequately cover the free space, but for 5000 obstacles it will produce an efficient path
- **RRT**: RRT is effective at quickly finding a feasible path in complex environments. It is particularly good at exploring large spaces, but the paths found are not always efficient or optimal. The success rate is limited by the number of nodes added; if the goal is difficult to reach, RRT may not find a solution within a fixed number of iterations.
- **RRT***: RRT* extends RRT by optimizing paths through the rewiring step. It has a similar success rate to RRT but produces a path of higher quality since it not only grows towards the goal but also refines the tree to improve coverage and quality.

Path Quality

- **PRM**: PRM can produce relatively high-quality paths if the roadmap has good coverage. However, it is not guaranteed to be optimal, and might miss shorter paths or optimal connections.

- **RRT**: The paths generated by RRT are often suboptimal, with sharp turns and unnecessary detours. This is because RRT focuses on expanding the tree towards unexplored areas without considering path efficiency.
- **RRT***: RRT* significantly improves path quality compared to RRT by rewiring nodes to minimize path costs. The paths generated by RRT* are asymptotically optimal, meaning that as more nodes are added, the solution converges towards the optimal path.

Computation Time

- **PRM**: The computation time for PRM is the longest out of the three. It can be summed up in two phases: making the roadmap and path finding. The graph construction can be time-consuming, especially for dense maps with many nodes. However, once the roadmap is built, querying for paths is relatively fast.
- **RRT**: RRT has lower computation time initially since it incrementally grows the tree and searches for a feasible path. The time taken depends on the number of nodes and the complexity of the environment. It generally performs well for quick, feasible pathfinding.
- **RRT***: RRT* requires more computation time than RRT due to the additional rewiring step that optimizes the path. As the number of nodes increases, the cost of maintaining and optimizing the tree also increases, making RRT* slower compared to RRT but producing higher-quality paths.

Statistics

For below, all 3 planners found the goal

RRT

1. python rrt.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.0 --goal_rad 0.2 --map environment_5.txt
2. python rrt.py --robot freeBody --start -7 3 0.5 --goal 4 7 1.2 --goal_rad 0.2 --map environment_5.txt
3. python rrt.py --robot freeBody --start -8 -3 1.57 --goal 5 5 0.9 --goal_rad 0.2 --map environment_5.txt

RRT*

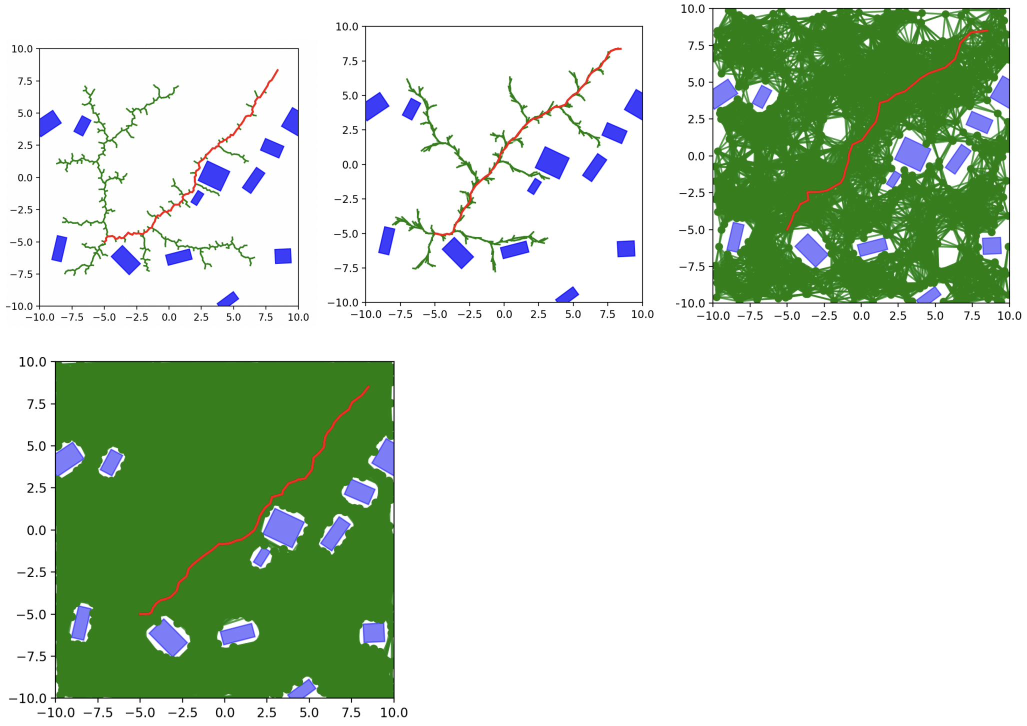
1. python AO_planner.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.0 --goal_rad 0.2 --map environment_5.txt
2. python AO_planner.py --robot freeBody --start -7 3 0.5 --goal 4 7 1.2 --goal_rad 0.2 --map environment_5.txt
3. python AO_planner.py --robot freeBody --start -8 -3 1.57 --goal 5 5 0.9 --goal_rad 0.2 --map environment_5.txt

PRM

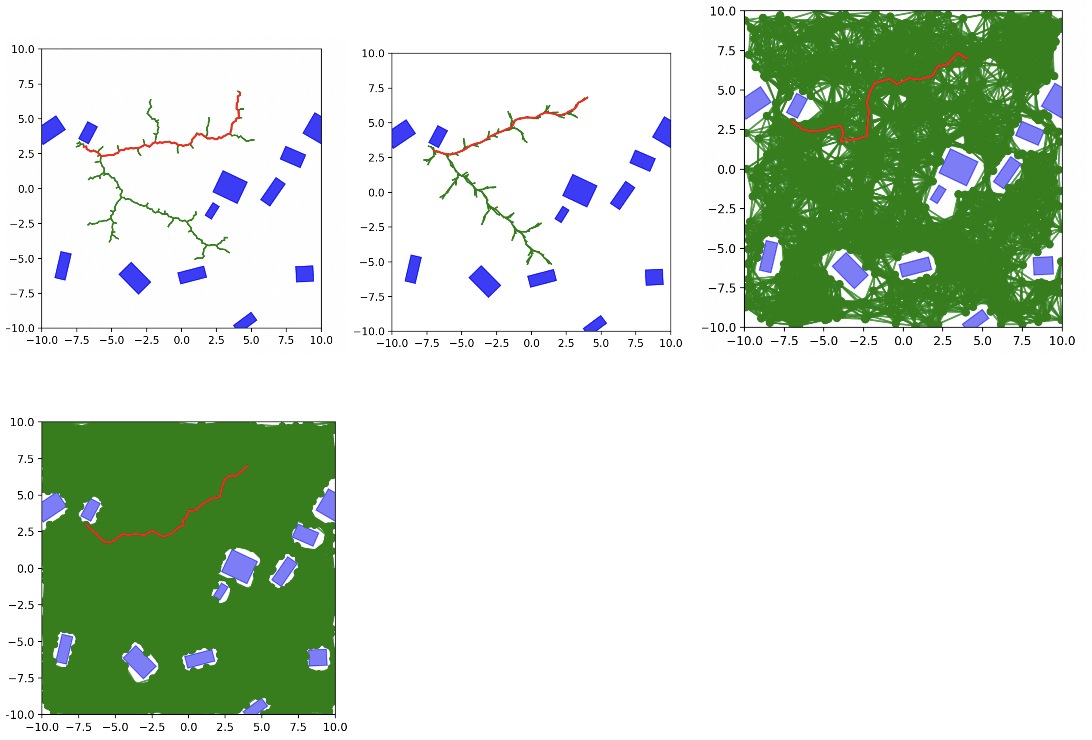
1. python prm.py --robot freeBody --start -5 -5 0 --goal 8.5 8.5 2.0 --map environment_5.txt
2. python prm.py --robot freeBody --start -7 3 0.5 --goal 4 7 1.2 --map environment_5.txt
3. python prm.py --robot freeBody --start -8 -3 1.57 --goal 5 5 0.9 --map environment_5.txt

For PRM, sampled 1000 nodes for 1-3 and 5000 nodes for 1-2

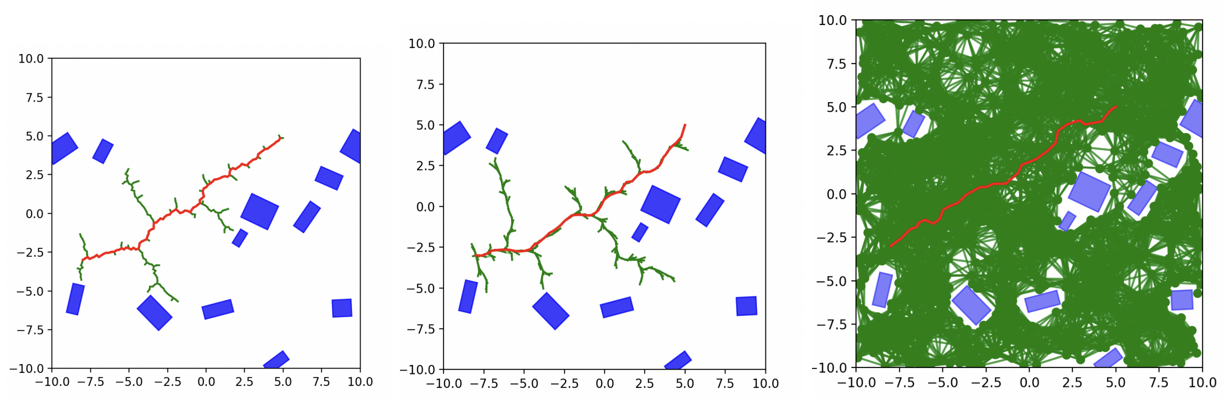
1)



2)

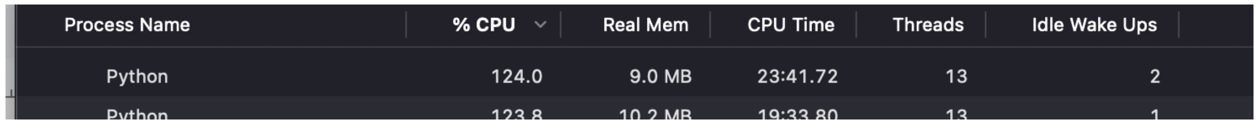


3)



Analysis

- PRM, if given enough samples, produces the straightest path, but takes exponentially longer than the others



Process Name	% CPU	Real Mem	CPU Time	Threads	Idle Wake Ups
Python	124.0	9.0 MB	23:41.72	13	2
Python	123.8	10.2 MB	19:33.80	13	1

  - (the above was for 5000 samples and it didn't finish until after 30 minutes)
  - RRT and RRT* ran in under a minute
  - if not given enough samples, can sometimes produce curvy angles
- RRT and RRT* for low amount of iterations produce nearly identical paths, but RRT* is more straight with less jumps as opposed to the bumpy, sharper turns of RRT
- Not shown in images, but if the goal was in between obstacles (around 4.8 -.25 1.57), RRT would have difficulty finding a path when limited to 1000. For 10 iterations, found in only 1/12 for RRT and 1/10 in RRT*, while PRM was able to do it every time

7. Motion Planning for a Car-Like Robot

The rrt_car.py script implements the RRT algorithm for a car-like mobile robot. Unlike holonomic robots, the car-like robot is subject to non-holonomic constraints, meaning it cannot move directly sideways, and its motion is defined by velocity and steering inputs. This introduces additional complexity into path planning.

Motion Model

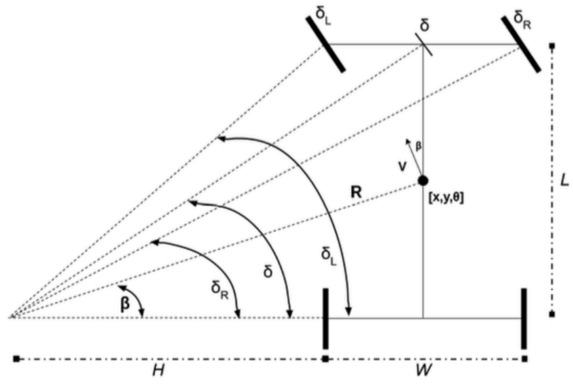
The car-like robot's motion model is defined by its velocity (V) and steering angle (δ), governed by the following differential equations:

Screenshot 2024-10-27 at 12.25.45 AM

Car-like Robot

$$\dot{x} = V \cdot cos(\theta + \beta)$$
$$\dot{y} = V \cdot sin(\theta + \beta)$$
$$\dot{\theta} = \omega$$
$$\omega = \frac{V}{R}$$
$$R = \frac{L}{2\sin(\beta)}$$
$$\beta = tan^{-}1(\frac{1}{2}\tan(\delta))$$

$$u = [V, \delta]'$$



Nonholonomic constraint

$$-\sin(\theta + \beta)\dot{x} + \cos(\theta + \beta)\dot{y} + L\dot{\theta}\cos\beta \equiv 0$$

RRT Adaptation for Car-like Robot

The RRT algorithm was adapted to accommodate the car-like robot's constraints:

- Steering Function: To connect nodes in the RRT tree, a steering function is implemented that simulates the car's motion under sampled control inputs. The steering function respects the car's kinematics by using the velocity and steering angle to guide the robot towards a target configuration while ensuring that the movement follows feasible trajectories.

- Collision Checking: Collision detection is performed by simulating the trajectory generated by the steering function and verifying if the car's path intersects with any obstacles. The collision checking ensures that the car avoids obstacles while navigating through the environment.

Visualization

- Tree Growth Animation: Displays the growth of the RRT tree in the configuration space, showing how the car-like robot explores feasible paths while adhering to its motion constraints.

- Solution Path Animation: Shows the car-like robot following the computed path from the start to the goal configuration. This path respects the car's non-holonomic constraints and avoids obstacles.