Assignment 2

Ravi Raghavan
rr1133

Michelle Han
mmh255

November 14, 2023

1 Motion Planning for a 2-link Planar Arm

1.1 Sampling Random Collision-Free Configurations

To generate a plot of random collision-free configurations, $\theta 1$ and $\theta 2$ were uniformly sampled. After each sample, the robot arm was updated and checked for collisions. If a collision occurs, the joint angles are repeatedly sampled until a collision-free configuration results. Each collision-free configuration is added to a list and the list of configurations are plotted at the end.

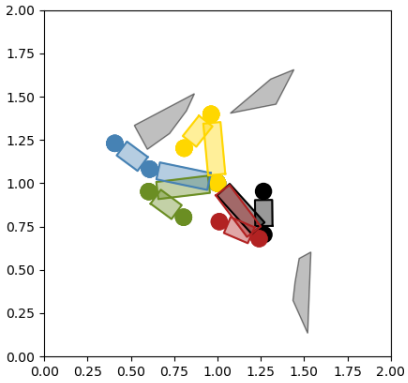

Figure 1: Provided environment

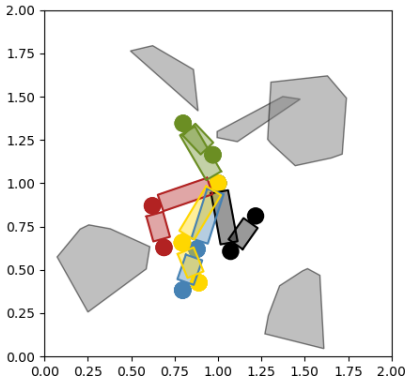

Figure 2: Assignment 1 environment

1.2 Nearest Neighbors

The distance metric was computed in two parts. First, the difference between $\theta 1$ and $\theta 2$ of the configurations was computed. The differences were then both squared and summed. The square root of the sum was then taken to get the Euclidean distance. The result, the Euclidean distance, was used as the distance metric for part 1.2 and the rest of the project.

1.2.1 With Linear Search Approach

To implement the Linear Search, the distance between the target and each configurations in the configurations list was computed. The distances were added to a list along with the configurations and this list was sorted in ascending order. The first k elements of the list gets returned.

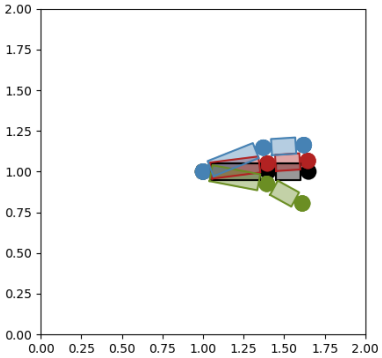

Figure 3: target = [0,0], k=3

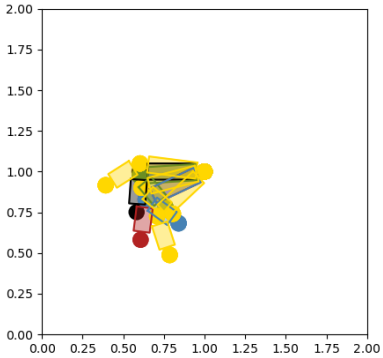

Figure 4: target = [3.14,1.5], k=6

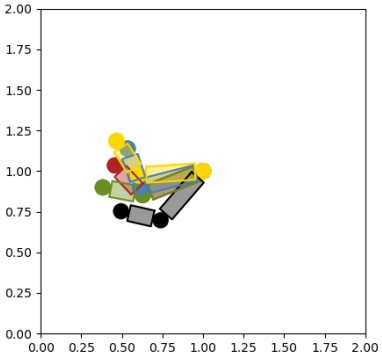

Figure 5: target = [4,5.2], k=4

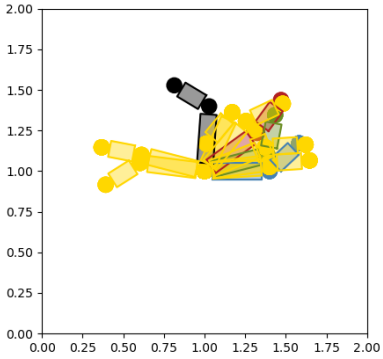

Figure 6: target = [1.5,1.1], k=10

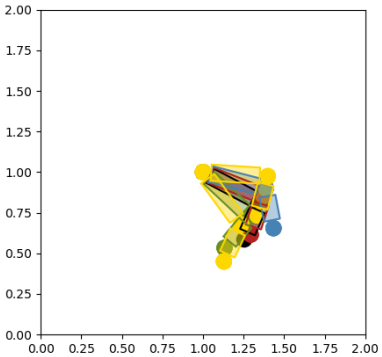

Figure 7: target = [5.8,-1.5], k=5

	Number of Nearest Neighbors			
	3	5	10	20
Running Time (in seconds)	0.0001890659332	0.0001862049103	0.0001668930054	0.0001678466797
	0.0002138614655	0.0002160072327	0.0002150535583	0.0002288818359

LINEAR SEARCH

KD-TREE SEARCH

Figure 8: Running time vs. number of nearest neighbors

	Number of Random Configurations			
	100	500	1000	2000
Running Time (in seconds)	0.0001680850983	0.0001730918884	0.0001730918884	0.0001680850983
	0.0002369880676	0.0002019405365	0.0002019405365	0.0001969337463

LINEAR SEARCH

KD-TREE SEARCH

Figure 9: Running time vs. number of configurations

1.2.2 With KD-Tree Search Approach

To implement the KD-Tree Search, a KD-Tree was implemented where the data was split based on where $\theta 1$ or $\theta 2$ was being partitioned. To search the KD-Tree, π was determined to be the split that determined whether to go left or right on the KD-Tree. This contrasts a KD-Tree with Euclidean topology where 0 is used.

1.2.3 Search Comparison

To compare running times between Linear search and KD-tree search, the target was [3.14, 0] for arm_configs.npy. When comparing the running time as the number of configurations increased, the k-nearest neighbors was set to 3 (see fig.10-13). When comparing the running time as the number of neighbors increased, the number of random configurations was 100 (see fig.14-17). Based on the figures generated from the search comparisons (see fig.8 and fig.9), the running times were consistent throughout for Linear search and KD-tree search

1.2.4 Visualizations

1.3 Interpolation Along the Straight Line in the C-Space

To move the arm from the start to goal, the difference between the joints were computed. The distance was normalized to be within [0, π] to account for the topology. The distance between $\theta 1$ of the start and goal was how much the joint 1 needed to rotate to achieve the goal position. Similarly, the distance between $\theta 2$ of the start and goal was how much the joint 2 needed to rotate to achieve the goal position. To make the animation look smooth, the step size of $\theta 1$ was $\theta 1$'s distance/20 and the step size of $\theta 2$ was $\theta 2$'s distance/20. The direction $\theta 1$ and $\theta 2$ move in is determined by the unnormalized difference between the goal and the start. If the difference is less than 0, this meant that the arm has to move in a clockwise direction. If the difference was greater than 0, this meant that the arm has to move in a counterclockwise direction.

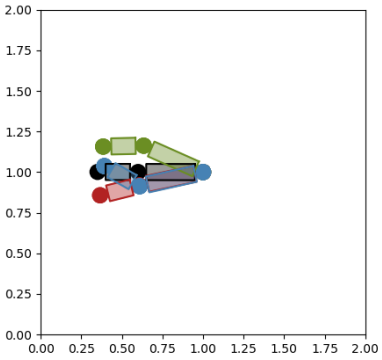

Figure 10: Search using 100 configurations

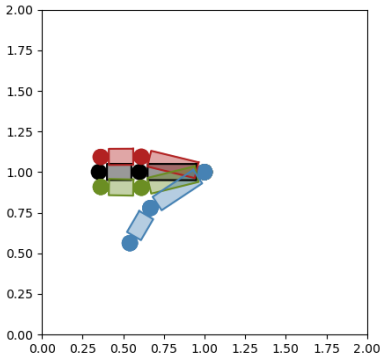

Figure 11: Search using 500 configurations

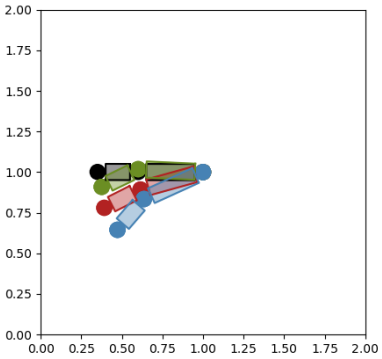

Figure 12: Search using 1000 configurations

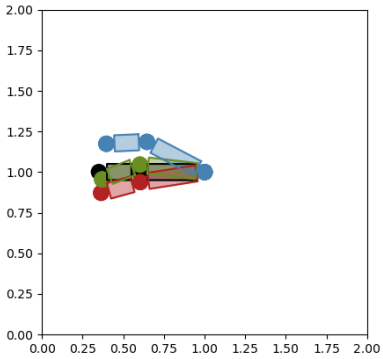

Figure 13: Search using 2000 configurations

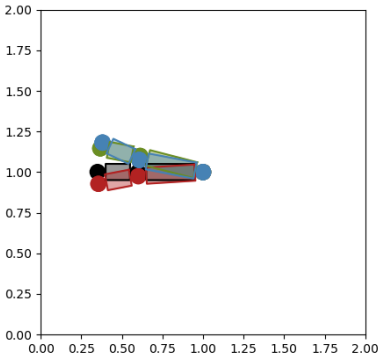

Figure 14: Search with k=3

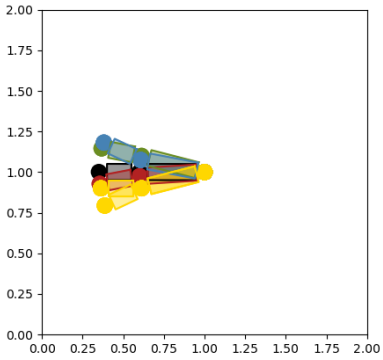

Figure 15: Search with k=5

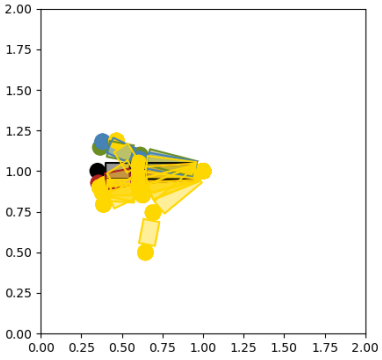

Figure 16: Search with k=10

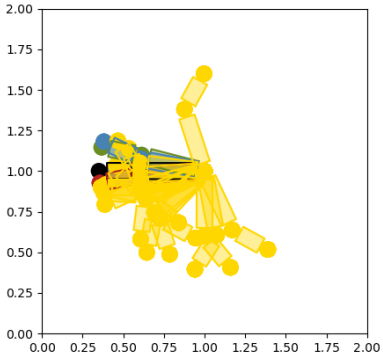

Figure 17: Search with k=20

1.4 RRT Implementation

To implement RRT, V was the set of vertices and E was the set of edges. To represent the graph structure, a Dictonary was used. Using the RRT algorithm, sampled configurations can be added until there were 1000 stored in the graph. The step size for $\theta 1$ and $\theta 2$ was determined by using the half-distance between the nearest configuration's and the random configuration's $\theta 1$ and $\theta 2$, respectively. The nearest configuration to random sample was found using Linear Search. To aid the search process, the goal was sampled 5 percent of the time

1.4.1 Edges

To add an edge, the path from the nearest configuration in the direction of a random configuration was interpolated. The step size along this path was the distance of $\theta 1/5$ and distance of $\theta 2/5$. Again the direction that theta moves in is determined by whether or not the difference between the "goal" and "start" is less than 0. If the path to the random configuration is collision-free at each step, then the edge is able to be added to the graph.

1.4.2 Wrap Around

Regarding the torodial topology of a 2-link arm, the edges require a "wrap" around. More specifically, if a configuration goes past 0, it wraps around to 2π. Similarly if a configuration goes past 2π, it wraps around to 0. If the start configuration is ¡= the goal configuration, wrap around occurs if the goal-start angles are greater than π. If the start configuration is ¿= the goal configuration, wrap around occurs if the start-goal angles are greater than π. This was computed for $\theta 1$ and $\theta 2$ between the configurations.

1.5 PRM Implementation

PRM consists of two parts, the roadmap construction and query process. To build the roadmap, N, collision-free configurations were added to the graph. The graph is again represented using a Dictionary. However, unlike RRT where two configurations were stored together without their distance, PRM was implemented to contain the distance. This was done so that the during the query process, dijkstra's could be applied. Using Dijkstra's a shortest path was found between the start and goal (if possible).

1.5.1 Further Notes

PRM took a considerably longer about of time in comparison to RRT. With more nodes in the graph, PRM had to check every configurations for it's k neighbors and form edges if possible. As a result, animation arm1.5-tree1 had to be computed using 100 nodes. This contrasted RRT which just connects to the nearest neighbor. To represent the configuration space, a "wrap around" for the edges was used as metnioned in section 1.4.

1.5.2 PRM*

To change PRM to implement PRM*, k was changed so that it was not a constant k = 3. Instead, PRM* involves setting k = log(n). As a result, with more nodes in the graph, more neighbors are being connected.

1.5.3 Planner Comparison

Overall, RRT had the highest success rate. PRM and PRM* both encounted the problem of not being able to find a feasible path within 1000 nodes or just taking too long. Though, with lower nodes, PRM and PRM* seemed to run far more faster. In terms of path quality, PRM and PRM* provided the more optimal paths. The paths provided by RRT seemed to move back and forth before finding the goal. PRM and PRM* presented a more direct path. For 5 environments, the respective pairs of start/goals: (-0.5, -0.5) to (1.5,1.5), (0, 0) to (1.5,1.5), (0, 3.14) to (3.14,1.57), (-1.57, -0.5) to (1.5,1.5), (-0.5, -1.57) to (1.5,1.5). PRM and PRM* had very low success rates with finding a path within 1000 nodes.

2 Motion Planning for a Rigid Body in 2D

2.1 Sampling Random Collision-Free Configurations

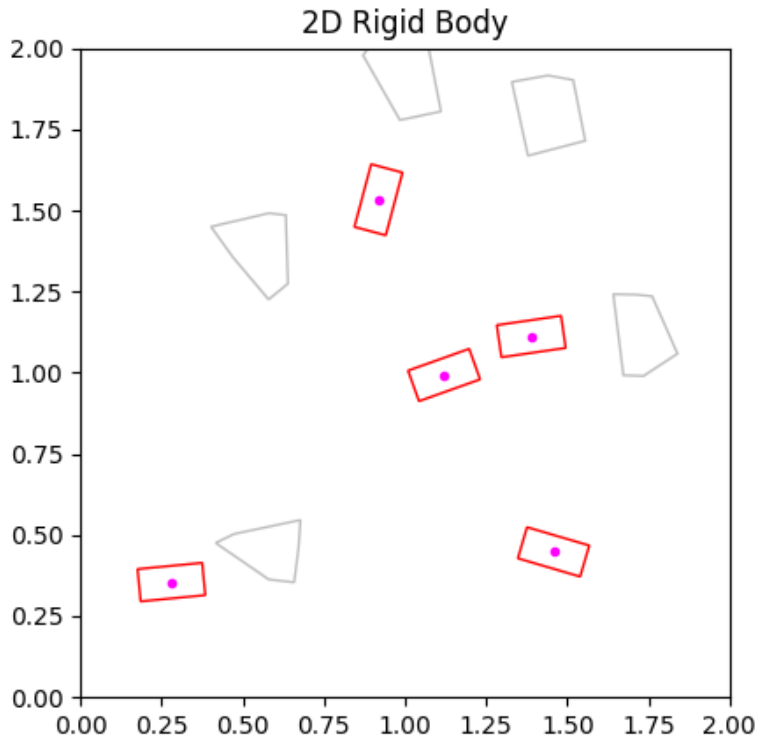

Figure 18: 1st Collision-Free Sampling for Environment 1

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration.

To generate this plot, I ran:
python rigid_body_1.py -- map "others/rigid_polygons.npy"

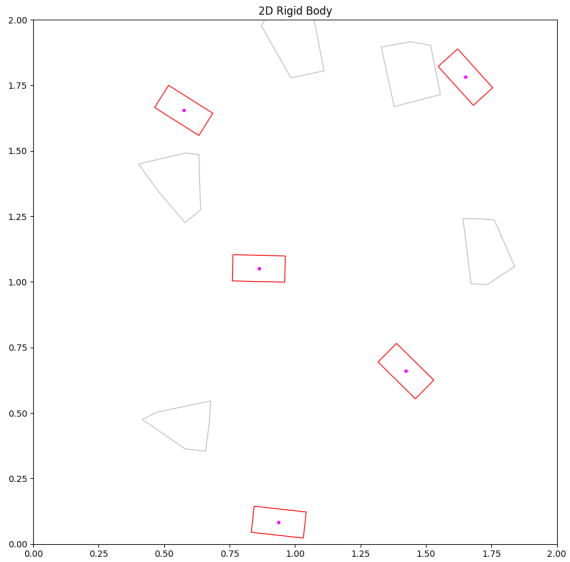

Figure 19: 2nd Collision-Free Sampling for Environment 1

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration.

To generate this plot, I ran:
python rigid_body_1.py -- map "others/rigid_polygons.npy"

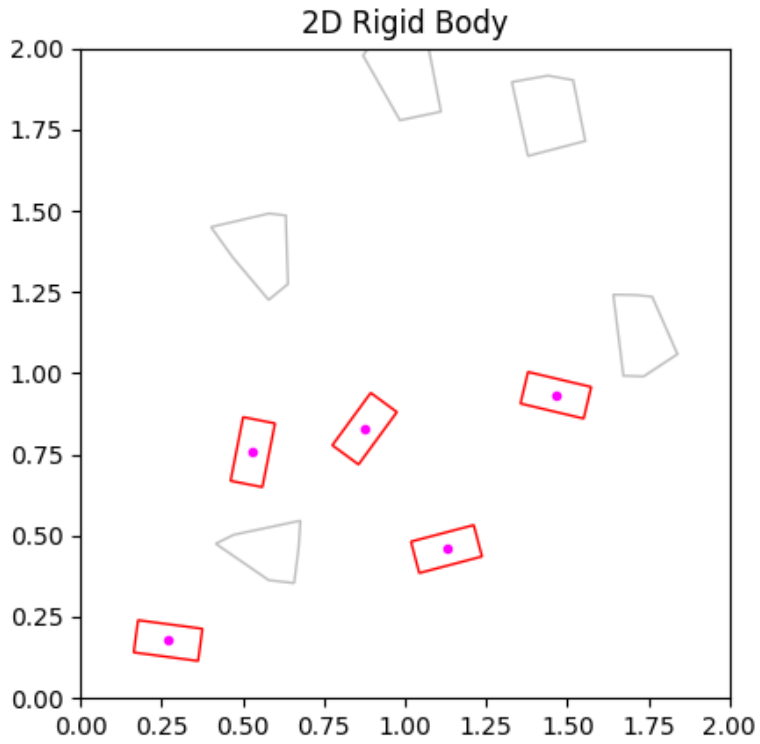

Figure 20: 3rd Collision-Free Sampling for Environment 1

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration.

To generate this plot, I ran:
python rigid_body_1.py -- map "others/rigid_polygons.npy"

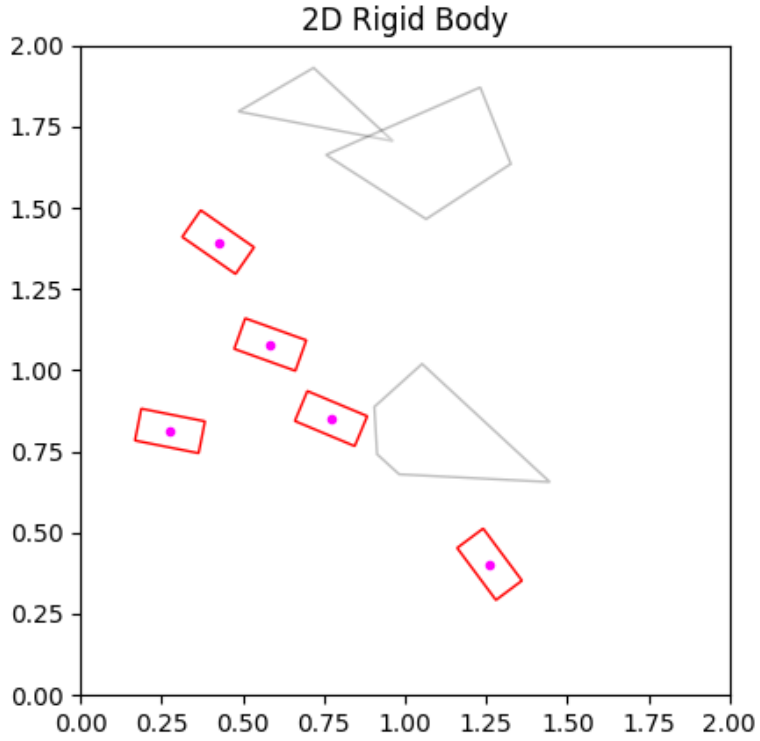

Figure 21: 1st Collision-Free Sampling for Environment 2

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene1.npy"

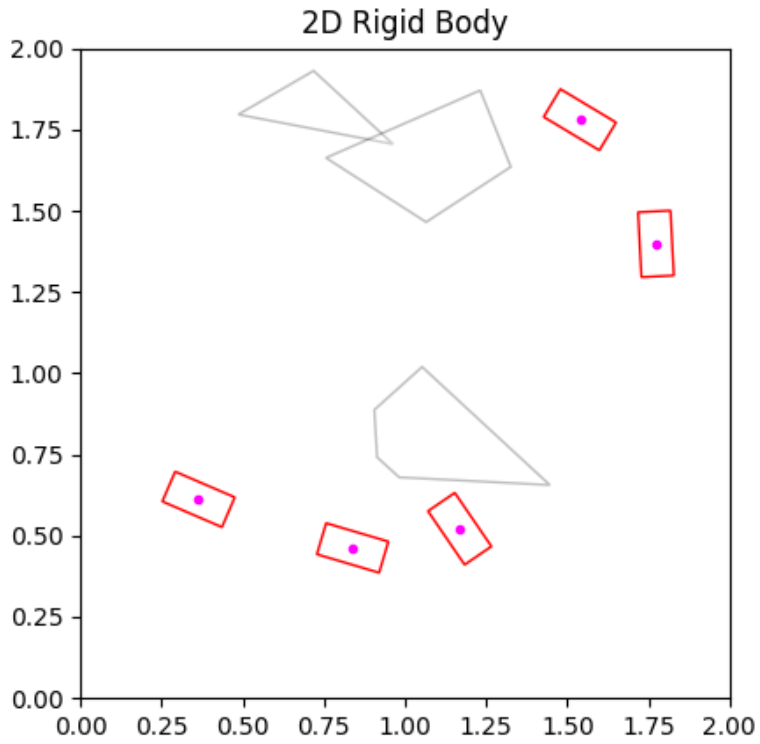

Figure 22: 2nd Collision-Free Sampling for Environment 2

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene1.npy"

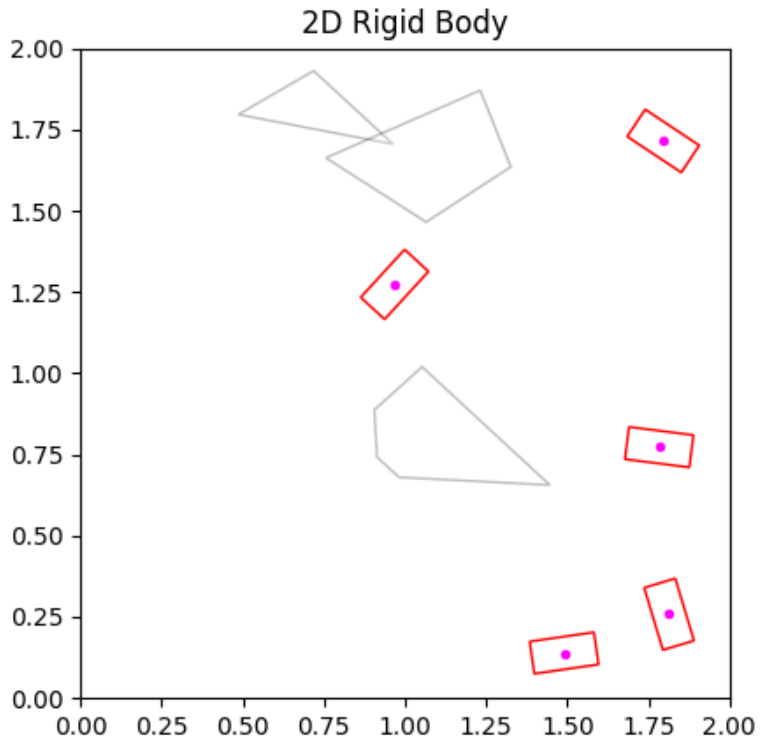

Figure 23: 3rd Collision-Free Sampling for Environment 2

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene1.npy"

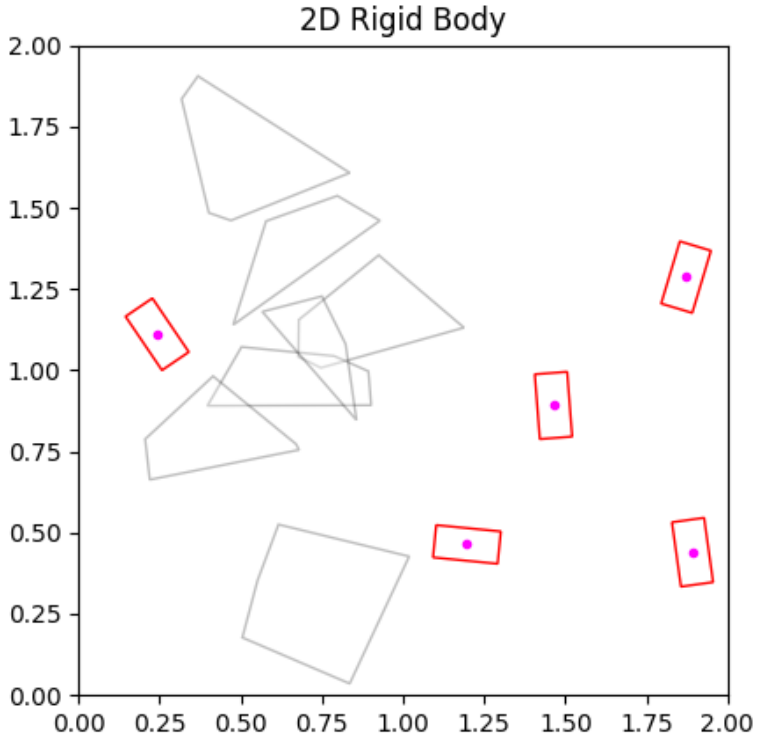

Figure 24: 1st Collision-Free Sampling for Environment 3

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene8.npy"

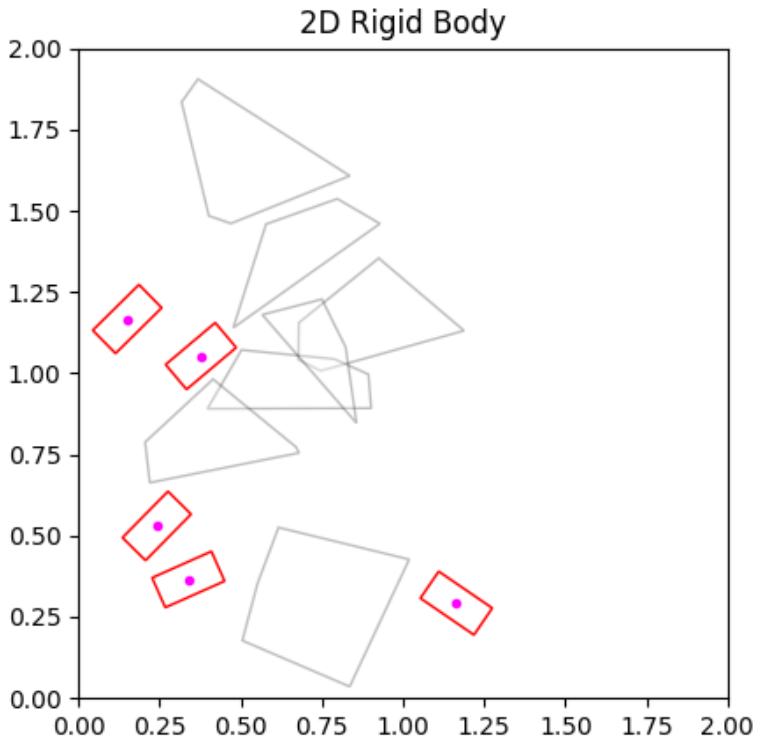

Figure 25: 2nd Collision-Free Sampling for Environment 3

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene8.npy"

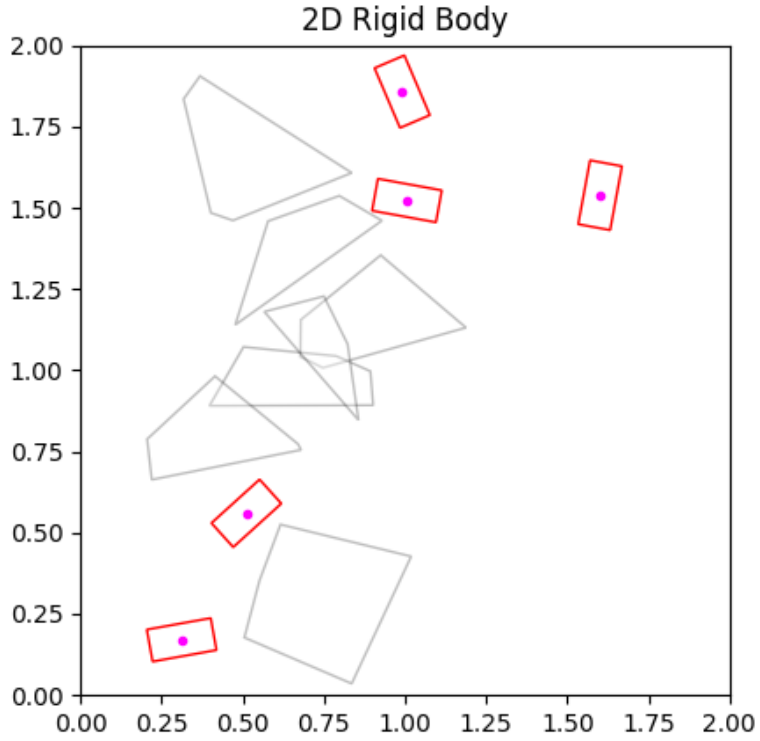

Figure 26: 3rd Collision-Free Sampling for Environment 3

Shown in the image above is a plot of the environment with 5 random collision-free configurations of the Rigid Body. For each configuration, the green dot represents the geometric center of the rigid body when it is in that configuration. To generate this plot, I ran, python rigid_body_1.py -- map "others/p2_scene8.npy"

2.1.1 Analysis

For Problem 2, we know that our configuration q for the 2D Rigid Body can be represented as (x, y, θ) where (x, y) are the coordinates of the rigid body's geometric center and θ is the degree of rotation of the rigid body with respect to its center.

Hence, to obtain 5 samples of random collision-free configurations, I followed the following basic steps 5 times! :

- I sampled x_{rand} uniformly from the range $(0, 2)$, I sampled y_{rand} uniformly from the range $(0, 2)$, and I sampled θ_{rand} uniformly from the range $(-\pi, \pi)$
- The configuration I obtained was $(x_{rand}, y_{rand}, \theta_{rand})$
- To determine whether this configuration was in the Free Space of the Configuration Space, I converted the configuration to workspace coordinates and applied collision checking(same algorithms I used in Assignment 1). I have included a brief description of these collision detection algorithms in the following section
- If the configuration was in the Free Space of the Configuration Space, I kept it

2.1.2 Collision Detection Review

To implement collision detection, my methodology can be broken down into two major portions. Let's say we are trying to detect if Polygon P and Q collide. The first thing I did was to construct bounding boxes around each polygon. Essentially, a bounding box for a polygon is akin to drawing a rectangle around the polygon that covers all its corners

Once the bounding boxes are constructed, when checking for collision, the first thing I check to see is if the bounding boxes for P and Q intersect. If no intersection is detected, I simply return False to indicate that there is NO collision.

However, if there is a collision detected between the bounding boxes of P and Q, I need to verify this collision. To do this verification, I use the Separating Axis Theorem(SAT).

Essentially, the main principle behind the Separating Axis Theorem is as follows. If a line/axis exists such that the projections of two polygons onto this line don't overlap, then the polygons don't collide.

For each edge of both polygons, I computed the normal vector to that edge. Then, I projected each polygon onto that normal vector. If there was a gap in the projections(i.e. there was no overlap in the projections), then I could conclude that there was NO intersection between the two polygons.

If there was no such gap, I would have to keep continuing this process until I found a normal vector where there was a gap in the projections of the polygons onto this vector.

At the end of the process, if I did not find any such gap of the projections onto any of the normal vectors, I could conclude that there was a collision between the polygons

2.2 Nearest Neighbors with Linear Search Approach

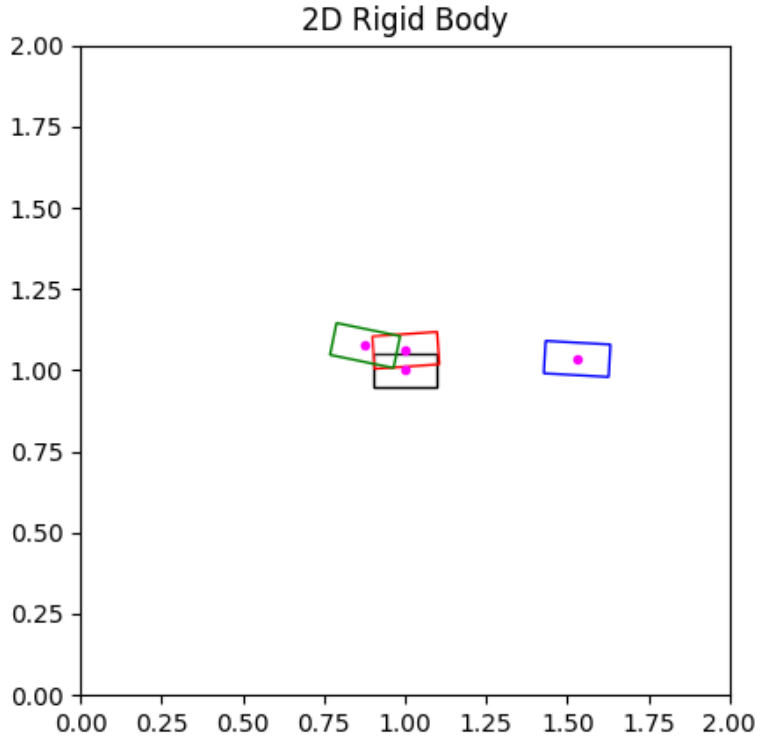

Figure 27: Nearest Neighbors(1)

Program Input: Target Configuration was $[1, 1, 0]$, number of nearest neighbors was 3, and the input list was specified by others/rigid_configs.npy

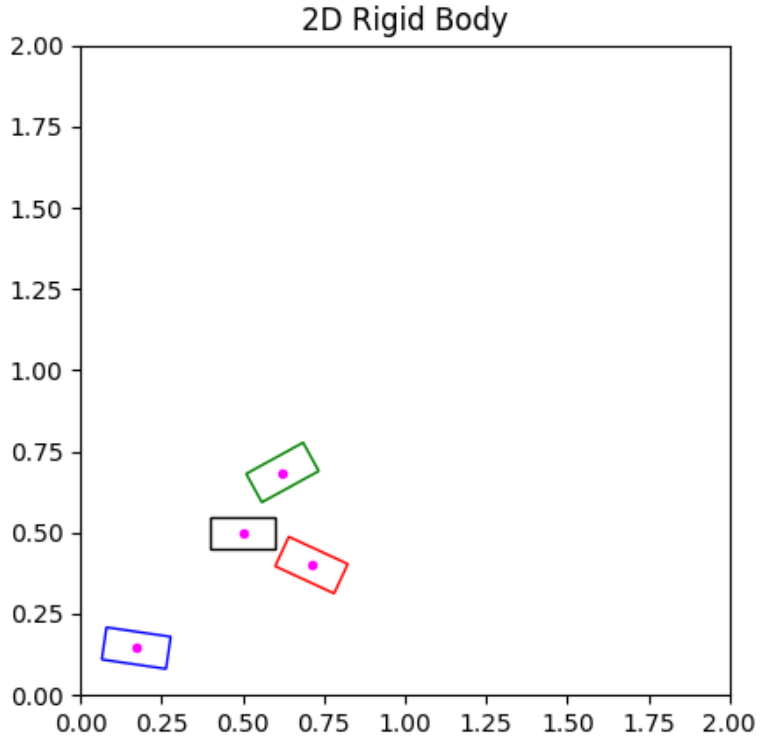

Figure 28: Nearest Neighbors(2)

Program Input: Target Configuration was $[0.5, 0.5, 0]$, number of nearest neighbors was 3, and the input list was specified by others/rigid_configs.npy

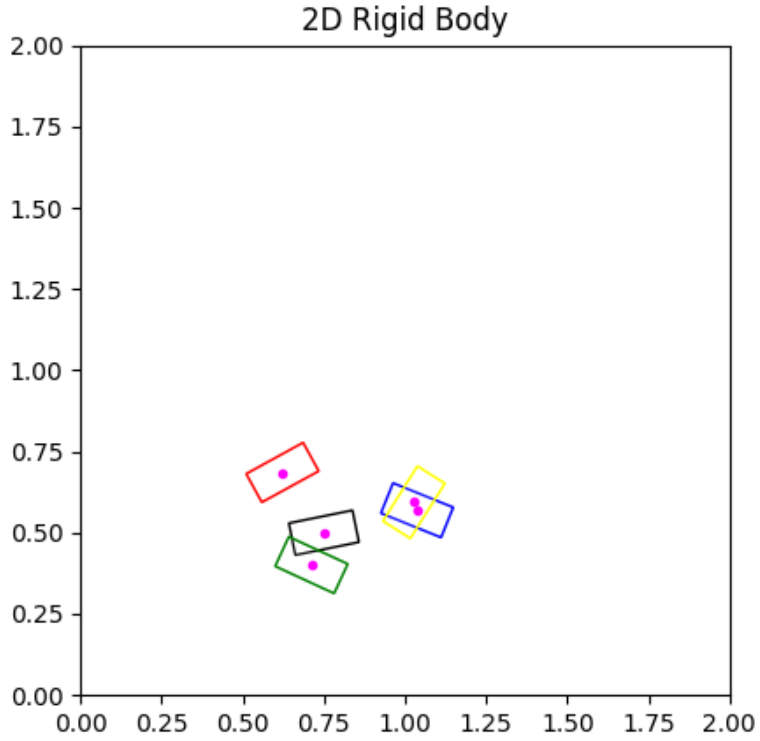

Figure 29: Nearest Neighbors(3)

Program Input: Target Configuration was $[0.75, 0.5, 0.2]$, number of nearest neighbors was 4, and the input list was specified by others/rigid_configs.npy

Note: The yellow configurations are there. You may have to squint a bit to see them :)

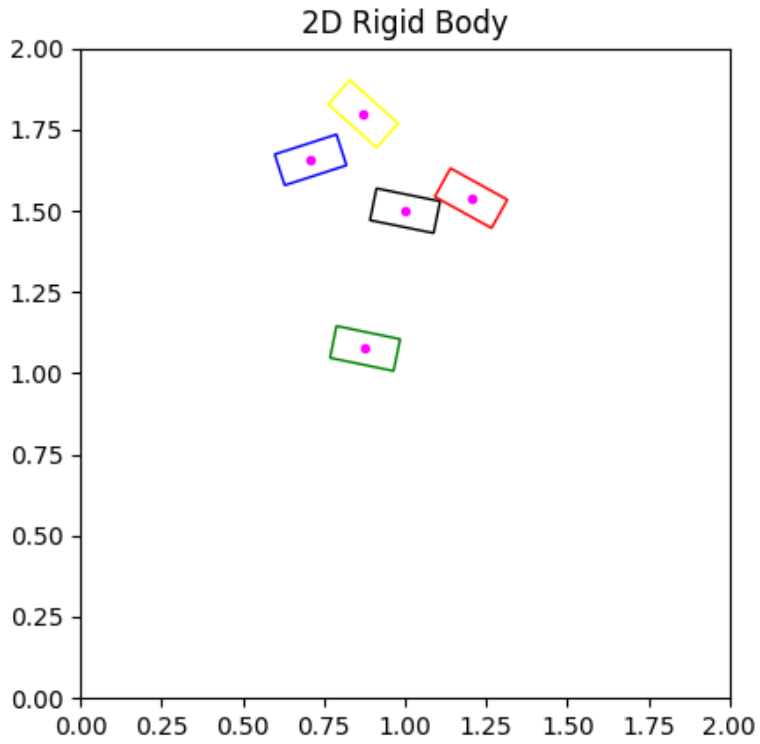

Figure 30: Nearest Neighbors(4)

Program Input: Target Configuration was $[1, 1.5, -0.2]$, number of nearest neighbors was 4, and the input list was specified by others/rigid_configs.npy

Note: The yellow configurations are there. You may have to squint a bit to see them :)

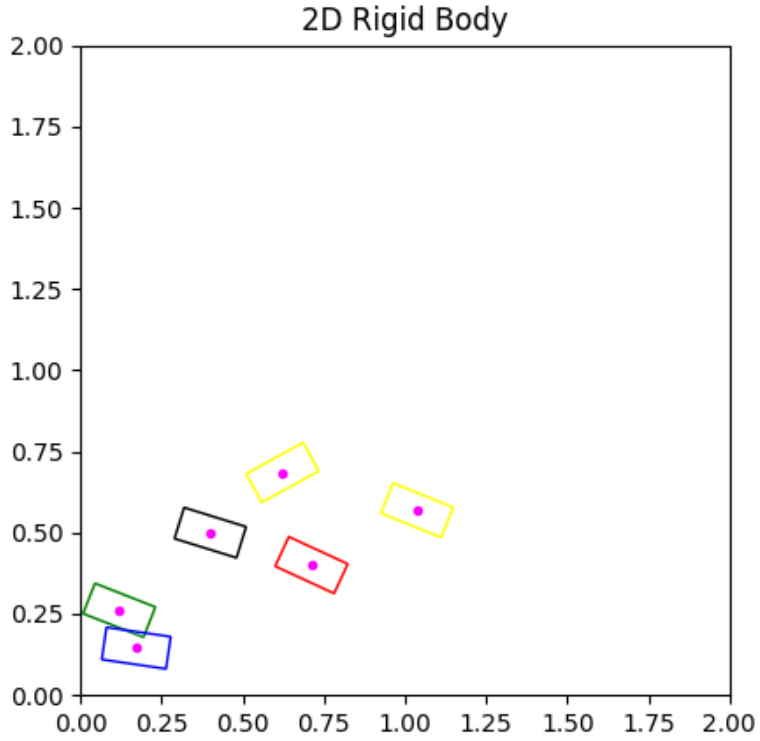

Figure 31: Nearest Neighbors(5)

Program Input: Target Configuration was $[0.4, 0.5, -0.3]$, number of nearest neighbors was 5, and the input list was specified by others/rigid_configs.npy

Note: The yellow configurations are there. You may have to squint a bit to see them :)

2.2.1 Analysis

First, I will explain the distance metric to compute the distance between two configurations and how I went about computing this distance.

Distance Metric: $D = \alpha d_t + (1 - \alpha)(d_r)$

where $\alpha = 0.7$, d_t is the Euclidean Distance between two configurations, d_r is the Rotational Distance between two configurations.

Let's say we have two configurations: (x_1, y_1, θ_1) and (x_2, y_2, θ_2), here is how I would go about computing the distance between the two configurations:

$$d_t = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Calculating d_r was also rather straightforward but I had to account for the topology here. The topology, for the 2D Rigid Body, was $SE(2)$. Hence, for θ, the angles can "wrap around" in a circle.

Hence, To calculate d_r, given θ_1 and θ_2, here is a rough algorithm of what I followed

Algorithm 1 Calculation of d_r

Parameters: $theta_1$, $theta_2$

$d_r = min(abs(\theta_1 - \theta_2), 2\pi - abs(\theta_1 - \theta_2))$

2.2.2 Search for Nearest Neighbors

Once my distance function was well defined, all I had to do was, given the query, perform a search over the given configurations(i.e. configurations given in others/rigid_configs.npy file).First I computed the distance to each configuration specified in others/rigid_configs.npy, then I sorted the resulting distances in ascending order, and then I output the k neighbors with the smallest distances(i.e. the first k values). I do this because we want "nearest" neighbors!.

2.3 Interpolation Along the Straight Line in the C-Space

2.3.1 Analysis

Let's say we have two configurations: (x_1, y_1, θ_1) and (x_2, y_2, θ_2). To interpolate along the straight line from (x_1, y_1, θ_1) to (x_2, y_2, θ_2), I would connect (x_1, y_1, θ_1) to (x_2, y_2, θ_2) and then divide the line into multiple segments.

For example, I would divide the line segment from (x_1, y_1, θ_1) to (x_2, y_2, θ_2) into n segments, and animate the rigid body moving along those n segments.

Let $q_{start} = (x_1, y_1, \theta_1)$ and let $q_{goal} = (x_2, y_2, \theta_2)$. Let D_t represent the translational distance from q_{start} to q_{goal} and let D_r represent the rotational distance from q_{start} to q_{goal}. Over each of the n segments, we will be moving a translational distance of $\frac{D_t}{n}$ and a rotational distance of $\frac{D_r}{n}$. If we have to move clockwise to go from start to goal, our animated rigid body will also move clockwise along each of the n segments. If we have to move counterclockwise to go from start to goal, our animated rigid body will also move counterclockwise along each of the n segments.

2.3.2 Interpolation With Respect to Topology

The topology, for the 2D Rigid Body, was $SE(2)$. For θ, the angles can "wrap around" in a circle. Hence, as I mentioned earlier, I accounted for this with my calculation of the rotational distance needed to go from θ_1 to θ_2. Please refer to Algorithm 1 where I outlined specifically how d_r would be calculated in such a case

2.4 RRT Implementation

Let V be the set of vertices of the RRT and E be the set of edges of the RRT.

Algorithm 2 RRT Algorithm

Parameters: N (number of nodes we want to have in our RRT), q_{start}(Node to start our RRT From), q_{goal} (Goal Node)
$V = V \cup \{q_{start}\}$
i $\rightarrow 0$
while $i < N$ **do**
  Sample q_{rand} from the free configuration space
  Let q_{near} be the point on the RRT that is closest to q_{rand}.
  From q_{near}, draw a straight line from q_{near} to q_{rand}. Make sure to step towards q_{rand} in small steps from q_{near}.
  If we hit an obstacle, let q_{steer} be the last point we visited on the straight line interpolation with q_{rand} where we didn't collide with an obstacle.
  if A point like q_{steer} could NOT found on the straight line interpolated path from q_{rand} to q **then**
    Continue to next Iteration
  else
    $V = V \cup \{q_{steer}\}$, $E = E \cup \{(q_{near}, q_{steer})\}$
    i $\rightarrow i + 1$
  end if
end while
if $q_{goal} \notin V$ **then**
  Let q_{near} be the point on the RRT that is closest to q_{goal}.
  Output the Path from q_{start} to q_{near}
end if

2.5 PRM Implementation

Let V be the set of vertices of the PRM and E be the set of edges of the PRM.

Algorithm 3 PRM Algorithm

Parameters: N (number of nodes we want to have in our PRM), k (how many neighbors we want to analyze when forming edges)
i $\rightarrow 0$
while $i < N$ **do**
  Sample q_{rand} from the free configuration space
  $V = V \cup \{q_{rand}\}$
  Let $Q_{neighbors}$ be the k nearest neighbors of q_{rand} out of all the nodes currently in the PRM. To calculate the "nearest" neighbors, the same methodology used in Section 2.2 was used.
  for $q_n \in Q_{neighbors}$ **do**
    if There is a collision free path between q_n and q_{rand} **then**
      $E = E \cup \{(q_n, q_{rand})\}$
    end if
  end for
  i $\rightarrow i + 1$
end while

Algorithm 4 PRM Query

Parameters: q_{start}(Start Node), q_{goal} (Goal Node), k (how many neighbors we want to analyze when forming edges)
Let $Q_{neighbors}$ be the k nearest neighbors of q_{start} out of all the nodes currently in the PRM. To calculate the "nearest" neighbors, the same methodology used in Section 2.2 was used.
for $q_n \in Q_{neighbors}$ **do**
  if There is a collision free path between q_n and q_{start} **then**
    $E = E \cup \{(q_n, q_{start})\}$
  end if
end for
Let $Q^2_{neighbors}$ be the k nearest neighbors of q_{goal} out of all the nodes currently in the PRM. To calculate the "nearest" neighbors, the same methodology used in Section 2.2 was used.
for $q_n \in Q^2_{neighbors}$ **do**
  if There is a collision free path between q_n and q_{goal} **then**
    $E = E \cup \{(q_n, q_{goal})\}$
  end if
end for
Run A^* Search Algorithm on the PRM Graph to find a path from q_{start} to q_{goal} in the Configuration space via the nodes on our PRM

2.5.1 Details on A* Implementation

Notation: $G(node)$ represents the G value of the node(i.e. current best distance from the start node to the current node. $H(node)$ is the heuristic estimate from the current node to the goal node. $D(q_1, q_2)$ is the distance between two configurations q_1 and q_2. Please refer to earlier discussions regarding the calculation of the distance function in this topological space.

Algorithm 5 A* Algorithm Implementation

```
Parameters: q_start(Start Configuration), q_goal(Goal Configuration)
Add q_start to the fringe heap
while Fringe is NOT empty do
  Pop Fringe and get node. Let's call it q_n
  if q_n is the goal then
    return
  end if
  Put q_n in the closed list
  Let the children of q_n be Q_children
  for q_c ∈ Q_children do
    if q_c is NOT in the closed list and NOT in the Fringe then
      Add q_c to the Fringe.
      H(q_c) = D(q_c, q_goal)
      G(q_c) = G(q_n) + D(q_n, q_c)
    end if
    if q_c is NOT in the closed list and is already in the Fringe then
      if G(q_n) + D(q_n, q_c) < G(q_c) then
        Set G(q_c) = G(q_n) + D(q_n, q_c)
      end if
    end if
  end for
end while
```

2.5.2 Proof that A* Heuristic is Consistent

As we learned in prior lectures, when it pertains to graph search, when the heuristic is consistent, then we can be certain that our A* Algorithm can find the optimal path(i.e. path with least cost). In our case, "cost" is path with smallest distance.

To recap, in order to prove that a heuristic is consistent, the following must be established:

Given three nodes, q_1, q_2, and q_{goal}, $h(q_1) \leq Cost(q_1, q_2) + h(q_2)$

Since, in this topological space, $Cost(q_1, q_2) = D(q_1, q_2)$, we can rewrite this as:
$h(q_1) \leq D(q_1, q_2) + h(q_2)$

Since we have set our $h(q) = D(q, q_{goal})$, this would require proving that:

$D(q_1, q_{goal}) \leq D(q_1, q_2) + D(q_2, q_{goal})$

Let $D_t(q, q_{goal})$ represent the translational distance from q to q_{goal}. Let $D_r(q, q_{goal})$ represent the rotational distance from q to q_{goal}. As mentioned earlier, $D(q, q_{goal}) = 0.7D_t(q, q_{goal}) + 0.3D_r(q, q_{goal})$

We already know, from Euclidean Geometry, that $D_t(q_1, q_{goal}) \leq D_t(q_1, q_2) + D_t(q_2, q_{goal})$

It is also intuitive to see that $D_r(q_1, q_{goal}) \leq D_r(q_1, q_2) + D_r(q_2, q_{goal})$

Let's say that we have to rotate θ degrees, either clockwise or counterclockwise, to get from q_1 to q_{goal}. There are two cases we can encounter. The first case is that the rotational configuration for q_2 is in between q_1 and q_{goal}. Then, $D_r(q_1, q_{goal}) = D_r(q_1, q_2) + D_r(q_2, q_{goal})$ and, by extension, $D_r(q_1, q_{goal}) \leq D_r(q_1, q_2) + D_r(q_2, q_{goal})$

The second case is where q_2 is NOT in between q_1 and q_{goal}. In this case, either $D_r(q_1, q_2)$ is greater than $D_r(q_1, q_{goal})$, $D_r(q_2, q_{goal})$ is greater than $D_r(q_1, q_{goal})$, or $D_r(q_1, q_2)+D_r(q_2, q_{goal}) = 2\pi-D_r(q_1, q_{goal})$. Since D_r always returns a value that is $\leq \pi$, we can state that, for the situation where $D_r(q_1, q_2) + D_r(q_2, q_{goal}) = 2\pi - D_r(q_1, q_{goal})$, $D_r(q_1, q_2) + D_r(q_2, q_{goal}) \geq D_r(q_1, q_{goal})$ Hence, the equation $D_r(q_1, q_{goal}) \leq D_r(q_1, q_2) + D_r(q_2, q_{goal})$ is still always satisfied.

Since $D_t(q_1, q_{goal}) \leq D_t(q_1, q_2) + D_t(q_2, q_{goal})$ and $D_r(q_1, q_{goal}) \leq D_r(q_1, q_2) + D_r(q_2, q_{goal})$

it is easy to see that $0.7D_t(q_1, q_{goal}) + 0.3D_r(q_1, q_{goal}) \leq 0.7(D_t(q_1, q_2) + D_t(q_2, q_{goal})) + 0.3(D_r(q_1, q_2) + D_r(q_2, q_{goal}))$

$0.7D_t(q_1, q_{goal}) + 0.3D_r(q_1, q_{goal}) \leq 0.7D_t(q_1, q_2) + 0.7D_t(q_2, q_{goal}) + 0.3D_r(q_1, q_2) + 0.3D_r(q_2, q_{goal})$

$0.7D_t(q_1, q_{goal}) + 0.3D_r(q_1, q_{goal}) \leq 0.7D_t(q_1, q_2) + 0.3D_r(q_1, q_2) + 0.7D_t(q_2, q_{goal}) + 0.3D_r(q_2, q_{goal})$

$0.7D_t(q_1, q_{goal}) + 0.3D_r(q_1, q_{goal}) \leq (0.7D_t(q_1, q_2) + 0.3D_r(q_1, q_2)) + (0.7D_t(q_2, q_{goal}) + 0.3D_r(q_2, q_{goal}))$

$D(q_1, q_{goal}) \leq D(q_1, q_2) + D(q_2, q_{goal})$

Since we have set our $h(q) = D(q, q_{goal})$, we have now officially proved that
$h(q_1) \leq D(q_1, q_2) + h(q_2)$.

By proving that my heuristic is consistent, it is also admissible, meaning that it is an optimistic estimate of the true cost to go. Furthermore, this would also show that the path returned by A* is optimal for this problem.

3 Motion Planning for a First-Order Car

3.1 Implementation of Dynamics

Configurations are defined as such:

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} \tag{1}$$

Controls are defined as such:

$$u(t) = \begin{bmatrix} v(t) \\ \phi(t) \end{bmatrix} \tag{2}$$

Derivative of Configuration is defined as such:

$$\dot{q}(t) = \begin{bmatrix} v\cos\theta(t) \\ v\sin\theta(t) \\ \frac{v}{L}\tan\phi \end{bmatrix} \tag{3}$$

System Dynamics:

$$q(t+dt) = q(t) + \dot{q}(t)dt \tag{4}$$

Instructions to Control Keyboard Animation:

- Up: Increase Velocity
- Down: Decrease Velocity
- Right: Increase value of ϕ
- Left: Decrease Value of ϕ
- 'c': Keep Controls the same and move along trajectory

Please Note that the control starts at $v = 0$, $\phi = 0$. Hence, when starting the animation, you might want to press the up/down keys to give the car some velocity

This is equivalent, in a real life scenario, to actually starting the car. Usually, to start off, you car is in a resting state with the engine off.

3.2 Integration of Dynamics

Given a starting configuration(i.e. $q(0)$) and a constant control (i.e. μ), I was able to perform the integration of the above system dynamic equation to determine the trajectory of the car.

To achieve this, I would simply set $dt = \Delta t = 0.01$. Then, the system dynamic equation can be rewritten as such:

$$q(t+\Delta t) = q(t) + \dot{q}(t)\Delta t \tag{5}$$

$$q(t + 0.01) = q(t) + \dot{q}(t)(0.01) \tag{6}$$

The Initial Value of this equation is

$$q(0) = \begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \end{bmatrix} \tag{7}$$

The control is constant and can be denoted as such:

$$u = \begin{bmatrix} v \\ \phi \end{bmatrix} \tag{8}$$

To solve for the Derivative of Configuration:

$$\dot{q}(t) = \begin{bmatrix} v\cos\theta(t) \\ v\sin\theta(t) \\ \frac{v}{L}\tan\phi \end{bmatrix} \tag{9}$$

The problem description denotes $L = 0.2$. Now, all the necessary information is needed to solve the equation $q(t + 0.01) = q(t) + \dot{q}(t)(0.01)$ for the value of the configuration at the next time step!

3.3 RRT and Planning with Dynamics

Algorithm 6 Modified RRT For Non-Holonomic Car

Randomly sample configurations q_{rand} and identify the closest node q_{close} on the tree to the random sample q_{rand}. Occasionally, the goal configuration will be sampled in order to assist RRT in finding a solution (e.g., 5 % of the time).

Instead of trying to connect with an edge the closest node q_{close} to the random sample q_{rand}, in this version of RRT, a blossom b of random controls $\{u_1, \ldots, u_b\}$ are sampled.

For each of the b sampled controls u_i, integrate forward in time the dynamics from the closest node q_{close} for the same duration ΔT. This results in b possible new states $\{q^1_{new}, \ldots, q^b_{new}\}$ that are reachable from q_{close}. Accept a new state only if the integration of the dynamics for time ΔT does not result in a collision.

Check the distance between all the new states $\{q^1_{new}, \ldots, q^b_{new}\}$ to the random sample q_{rand} and identify which state q_{cn} among them is the closest to q_{rand} given the distance metric in this configuration space. add the edge from q_{close} to q_{cn} and store the corresponding control on the edge.

3.3.1 Implementation Details

Here, I will explain the implementation details from my project that I used to help my RRT Algorithm converge to a solution quicker

For each of the sampled controls, to ensure that my RRT converged to a solution quicker, I integrated forward in time the dynamics from the closest node q_{close} for a total of 250 integration steps. Please note that I would stop integrating when I hit an obstacle and the "new" state that I attained(i.e. q_{new}) would be the state along the integration path just prior to hitting the obstacle.

Furthermore, to ensure that my RRT reached the goal region, I sampled the goal node 5% of the time.

3.3.2 Experimentation

To experiment with different design choices, my strategy was to change the value of b and observe the success rate of the RRT Algorithm, the resulting path quality of the path returned by the RRT Algorithm, and the overall computational cost.

I have run these experiments on various polygonal scenes. For each polygonal scene, I ran 5 trials for different values of b(i.e. 1, 3, and 5). For each trial on each value of b, I report a tuple (i, j, k)

$i = 0$ if we could NOT find a successful path from the start to the goal region. $i = 1$ if we could find a successful path from the start to the goal region.

j is equal to the total path distance as computed by the distance metric in this configuration space

k is the time required to build the RRT in seconds

Scene 1: *rigid_polygons.npy*
Scene 2: *p2_scene1.npy*
Scene 3: *p2_scene8.npy*

For each experiment, I ran:
Scene 1: python car_2.py –start 0.5 0.25 0 –goal 1.75 1.5 0.0 –map "rigid_polygons.npy"

Scene 2: python car_2.py –start 0.5 0.25 0 –goal 1.75 1.5 0.0 –map "p2_scene1.npy"

Scene 3: python car_2.py –start 0.25 0.25 -3 –goal 1.2 0.75 0.25 –map "p2_scene8.npy"

Another thing to note is that, since RRT is probabilistically complete, if we let RRT run with no limit on the number of iterations, it will eventually find a solution(i.e. a path from start to the goal vertex). However, to compare whether a solution can be found across different values of b, I wanted to put a maximum limit on the number of iterations we run. I set the maximum limit for the number of iterations as 10000

Trial #	b = 1	b = 3	b = 5
1	(1,8.62,105)	(1,4.87,131)	(1,6.32,14)
2	(1,7.81,92)	(1,7.37,67)	(1,5.58,147)
3	(1,9.39,261)	(1,3.77,68)	(1,5.59,176)
4	(1,4.97,219)	(1,5.81,52)	(1,3.44,21)
5	(1,6.71,43)	(1,8.81,321)	(1,4.45,104)
Avg	(1, 7.5, 144)	(1, 6.126, 127.8)	(1, 5.076, 92.4)

Table 1: Table for Scene 1

Trial #	b = 1	b = 3	b = 5
1	(1, 6.83, 261)	(1, 5.47, 61)	(1, 12.57, 574)
2	(1, 8.63, 559)	(1, 6.19, 46)	(1, 8.60, 128)
3	(1, 11.99, 267)	(1, 4.43, 3)	(1, 8.13, 125)
4	(1, 9.11, 1256)	(1, 8.36, 163)	(1, 7.78, 282)
5	(1, 6.84, 102)	(1, 7.89, 46)	(1, 7.40, 86)
Avg	(1, 8.68, 489)	(1, 6.468, 63.8)	(a, 8.896, 239)

Table 2: Table for Scene 2

Trial #	b = 1	b = 3	b = 5
1	(1, 8.78, 232)	(1, 3.95, 150)	(1, 11.13, 186)
2	(1, 3.82, 135)	(1, 7.56, 184)	(1, 6.32, 9)
3	(1, 3.90, 182)	(1, 7, 264)	(1, 2.99, 239)
4	(1, 8.05, 144)	(1, 6.76, 128)	(1, 3.65, 16)
5	(1, 6.61, 156)	(1, 3.56, 31)	(1, 3.25, 281)
Avg	(1, 6.232, 169.8)	(1, 5.766, 151.4)	(1, 5.468, 146.2)

Table 3: Table for Scene 3

3.3.3 Scene 1 Analysis

Impact on RRT Build Time

Based on my analysis for Scene 1, the time taken to construct the RRT seems to decrease as I increase the value of b. Increasing the value of b enables me to sample more controls during each iteration of the RRT.

An important thing to keep in mind is that we sample the goal node 5% of the time. For I iterations of RRT, the expected number of times we sample the goal node is $0.05 * I$. Due to the fact that we sample more controls as b increases, we will be integrating along more pathways as dictated by the dynamic equation and the value of the control. Hence, due to the simple fact that we are exploring more pathways, we are more likely to add a node to the RRT that is closer to the goal node. Hence, as b increases, we are likely to get to the goal region a lot quicker.

Impact on Path Distance

Based on my analysis for Scene 1, the path distance seems to decrease as I increase the value of b. Increasing the value of b enables me to sample more controls during each iteration of the RRT. Sampling more controls would allow the RRT Algorithm to explore more areas of the free Configuration Space and, thus, find shorter paths to the goal region

Impact on Success Rate

My experiments show that my RRT Implementation was always successful in finding a path. This showed that 10000 iterations, as a max threshold, was sufficient in letting my RRT Algorithm converge to a solution

3.3.4 Scene 2 Analysis

Impact on RRT Build Time

Based on my analysis for Scene 2, the time taken to construct the RRT seems to decrease when I increase b from 1 to 3. However, the time taken to construct the RRT increases when I increase b from 3 to 5. Here is my hypothesis for why this is occurring.

An important thing to keep in mind is that we sample the goal node 5% of the time. For I iterations of RRT, the expected number of times we sample the goal node is $0.05 * I$. Due to the fact that we sample more controls as b increases, we will be integrating along more pathways as dictated by the dynamic equation and the value of the control. Hence, due to the simple fact that we are exploring more pathways, we are more likely to add a node to the RRT that is closer to the goal node. Hence, as b increases, we are likely to get to the goal region a lot quicker.

However, when we increase b past a certain point, due to the fact that Scene 2 is rather sparse(i.e. has a few polygonal obstacles) when compared to Scene 1, sampling too many controls may lead us to consider far too many potential directions. Furthermore, since the free space is much bigger, we must consider the Voronoi Bias of the RRT.

For a recap, here is the definition of the Voronoi Bias of the RRT: Voronoi Bias refers to the fact that the probability of a tree node being selected as the "closest" tree node is proportional to the volume of its Voronoi Region. Our Search is biased towards those nodes with the largest Voronoi Regions(i.e. Unexplored regions of the configuration space)

Hence, since RRT is more biased towards the nodes with the largest Voronoi Regions and since our Scene is more sparse(i.e. less polygonal obstacles and bigger free space), we may have the situation where our RRT is exploring regions of the free space that is away from the goal.

This, in my opinion, would explain the increase in RRT Build Time when b was increased from 3 to 5

Impact on Path Distance

Based on my analysis for Scene 2, the path distance seems to decrease when I increase b from 1 to 3. However, the path distance increases when I increase b from 3 to 5. Here is my hypothesis for why this is occurring

Intuitively, when we increase b, we would expect the path distance to decrease. Increasing the value of b enables me to sample more controls during each iteration of the RRT. Sampling more controls would allow the RRT Algorithm to explore more areas of the free Configuration Space and, thus, find shorter paths to the goal region

However, once b crosses a certain threshold, sampling too many controls may lead us to consider far too many potential directions. This may lead the RRT Algorithm astray from the goal node

Impact on Success Rate

My experiments show that my RRT Implementation was always successful in finding a path. This showed that 10000 iterations, as a max threshold, was sufficient in letting my RRT Algorithm converge to a solution

3.3.5 Scene 3 Analysis

Impact on RRT Build Time

Based on my analysis for Scene 3, the time taken to construct the RRT seems to decrease as I increase the value of b. Increasing the value of b enables me to sample more controls during each iteration of the RRT.

An important thing to keep in mind is that we sample the goal node 5% of the time. For I iterations of RRT, the expected number of times we sample the goal node is $0.05 * I$. Due to the fact that we sample more controls as b increases, we will be integrating along more pathways as dictated by the dynamic equation and the value of the control. Hence, due to the simple fact that we are exploring more pathways, we are more likely to add a node to the RRT that is closer to the goal node. Hence, as b increases, we are likely to get to the goal region a lot quicker.

Impact on Path Distance

Based on my analysis for Scene 3, the path distance seems to decrease as I increase the value of b. Increasing the value of b enables me to sample more controls during each iteration of the RRT. Sampling more controls would allow the RRT Algorithm to explore more areas of the free Configuration Space and, thus, find shorter paths to the goal region

Impact on Success Rate

My experiments show that my RRT Implementation was always successful in finding a path. This showed that 10000 iterations, as a max threshold, was sufficient in letting my RRT Algorithm converge to a solution