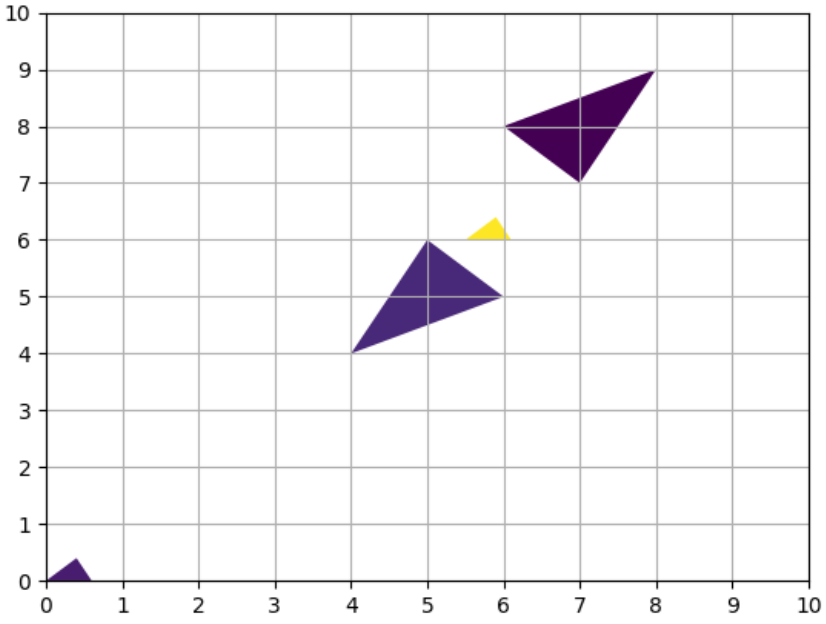
Kaivalya Mishra CS460

Assignment 2 Report

1 2D Geometric Motion Planning

For this section, I mainly used this environment for testing the various functions:

```
Robot: [(0.6, 0.0), (0.4, 0.4), (0.0, 0.0)]
Obstacle 0: [(6.0, 5.0), (5.0, 6.0), (4.0, 4.0)]
Obstacle 1: [(8.0, 9.0), (6.0, 8.0), (7.0, 7.0)]
Start: (0.0, 0.0)
Goal: (8.0, 8.0)
```



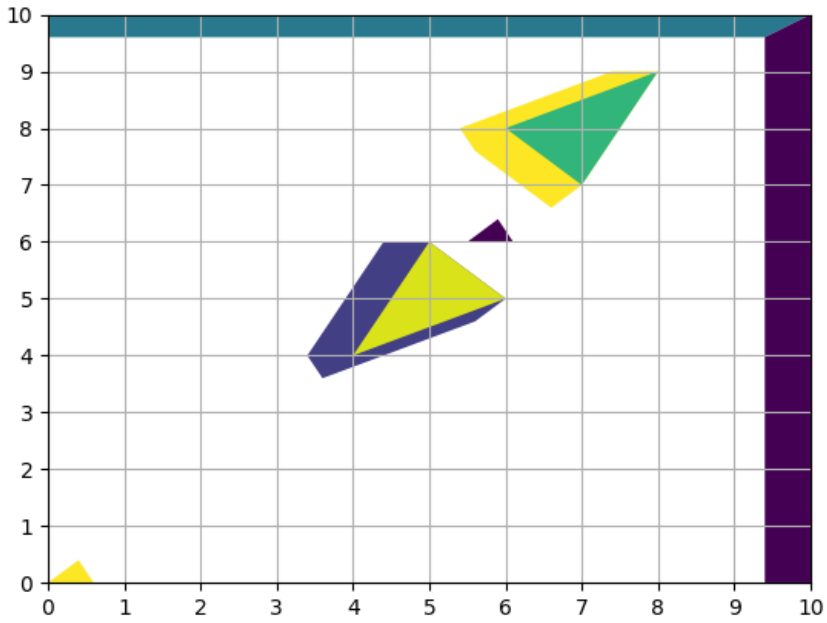
For generating the configuration space, I created a method `define_cspace` in `rrt.py`. This function uses the methodolgy of Andrew's monotone chain convex hull algorithm to generate the convex hulls of the obstacles.
In the algorithm, the the upper and lower halves of the convex hull are made via the set of points sorted by their x-values (descending for lower hull, ascending for upper hull) and then checking a series of three points a, b, and p (with the order of the points being $a \rightarrow b \rightarrow p$) for what kind of turn is made from vector ab to bp. If the turn is counter-clockwise, b is popped out from the current hull list. Eventually, the resulting hulls are then combined to give the convex hull of the set of points.

```
for obs in obstacles:
    points = []
    for i in obs:
        for j in invert_bot:
            points.append((i[0]+j[0],i[1]+j[1]))
    points = sorted(set(points))

    if (len(points) <= 1):
        new_obstacles.append(points)
        continue

    lowHull = []
    for p in points:
        while len(lowHull) >= 2 and
              cross_prod(lowHull[-2], lowHull[-1], p) <= 0:
            lowHull.pop()
        lowHull.append(p)
    upHull = []
    for p in reversed(points):
        while len(upHull) >= 2 and
              cross_prod(upHull[-2], upHull[-1], p) <= 0:
            upHull.pop()
        upHull.append(p)

    new_obstacles.append(lowHull[:-1] + upHull[:-1])
```



Base environment over configuration space

As for implementing the `Tree` structure, I created a new field named `data` which was a dictionary storing all nodes as keys and the nodes' parents, costs, and children in a list as the value. The dictionary can be represented by this layout:

```
{node: [node_parent, node_cost, node_children]}
```

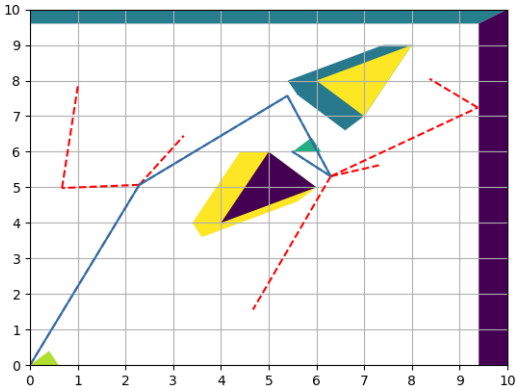
I implemented a complete version of the `extend` function, having two helper methods `path_intersection` and `seg_intersect`. In `path_intersection`, I checked to see if the line segement (using `seg_intersect`) from `point1` to `point2` intersected any obstacle edges in the c-space. I used a valid intersection closest to `point1` if the path to `point2` was not clear, otherwise I simply extended to `point2`.

Search trees and solution paths are animated for both this section and the next (Geometric Motion Planning for a Car) when calling `visualize_rrt` and `visualize_rrt_star`. The animations for search tree growth and robot traversal across the solution path (if it exists) are shown one after another.

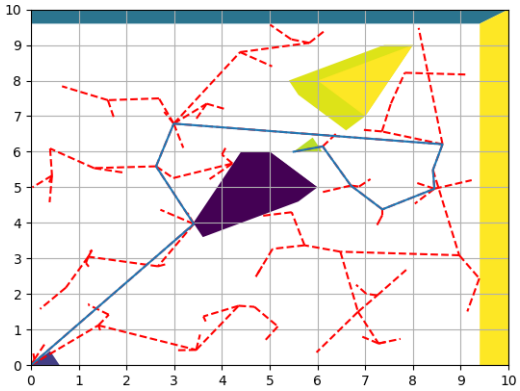
RRT Performance

In measuring the success rate compared to iteration count for `rrt`, I conducted the algorithm (using the previously environment for the tests) at iteration levels of 10, 20, 50, and 100 for 10 runs each:

Iteration Level	Success Rate
10	7/10
20	7/10
50	9/10
100	10/10



10 Iterations Example



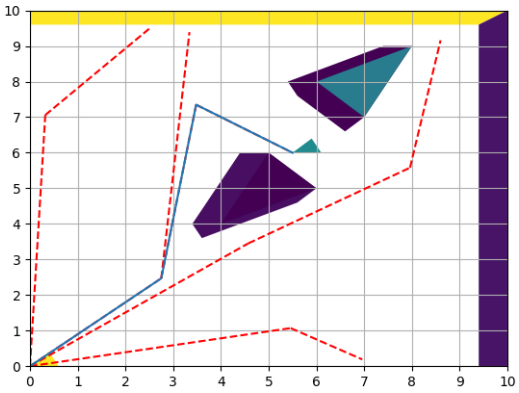
100 Iterations Example

Success rate seemed to increase alongside number of iterations taken. However, path optimality was essentially not present regardless of iteration count, as can be seen with the examples above.

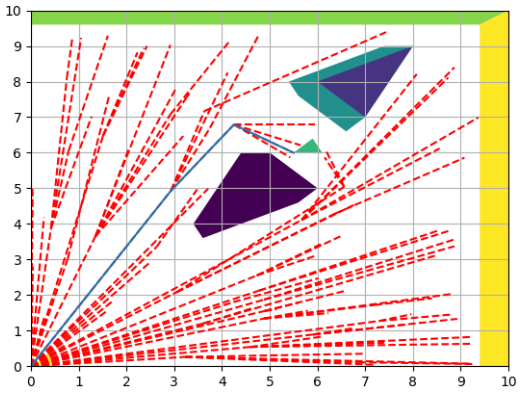
RRT* Performance

I used the same experimental procedure for `rrt*` from `rrt`:

Iteration Level	Success Rate
10	7/10
20	8/10
50	10/10
100	10/10



10 Iterations Example



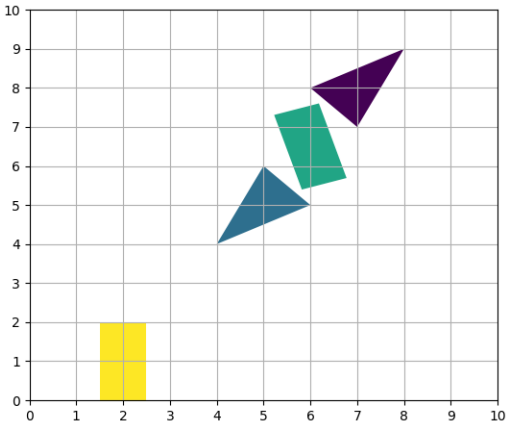
100 Iterations Example

As was the case with `rrt`, success rate seemed to increase alongside number of iterations taken, even more so for this algorithm. Path optimality also appeared to increase with iteration count as well, as can be seen from the above graphs. The search trees, in comparison to `rrt`'s search trees, seem to radiate out in far less winding paths and do not self intersect. It can also be seen that the paths tend to 'wrap' around the objects in c-space.

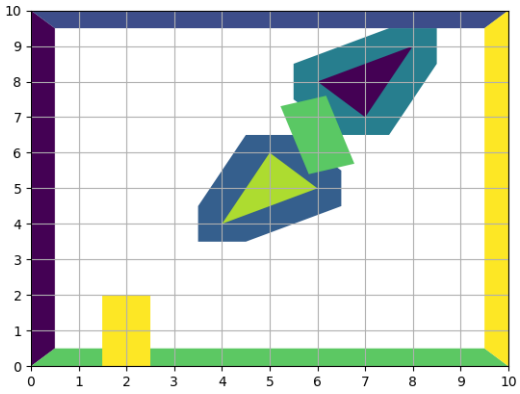
2 Geometric Motion Planning for a Car

For this section, I mainly used this environment for testing the various functions:

```
Robot Dimensions: (1.0, 2.0)
Obstacle 0: [(6.0, 5.0), (5.0, 6.0), (4.0, 4.0)]
Obstacle 1: [(8.0, 9.0), (6.0, 8.0), (7.0, 7.0)]
Start Pose: (2.0, 1.0, 0.0)
End Pose: (6.0, 6.5, 0.3)
```



Base environment



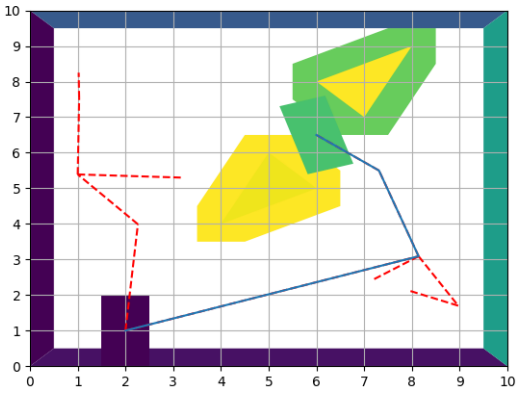
Base environment over c-space

I adapted most of the code over from section 1 for this section. Some things did have to be changed however, considering the `robot` variable had to be passed as its class rather than a list, alongside a few other details due to the robot being symmetric and being able to rotate.

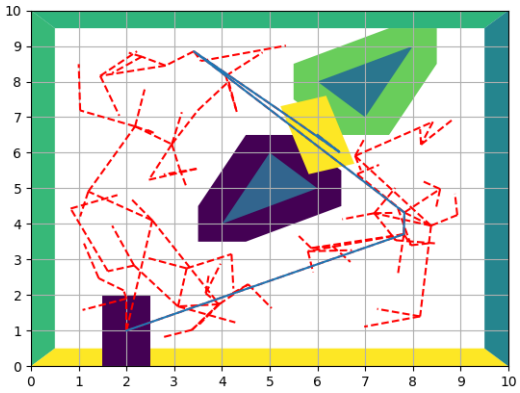
RRT Performance

I used the same experimental procedure as in the previous section:

Iteration Level	Success Rate
10	6/10
20	8/10
50	9/10
100	10/10



10 Iterations Example



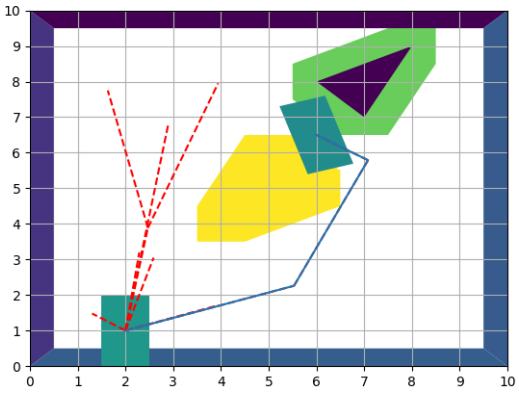
100 Iterations Example

As was the case in section 1, success rate seemed to increase alongside number of iterations taken but not path optimality.

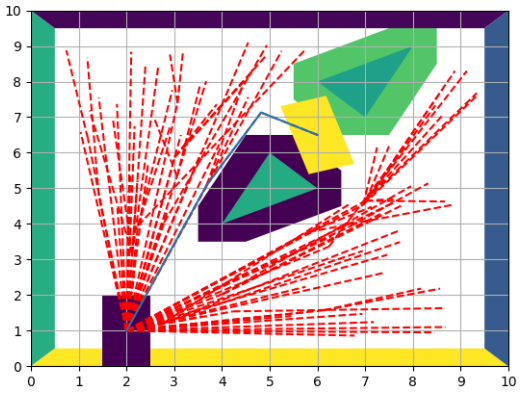
RRT* Performance

I used the same experimental procedure as in the previous section:

Iteration Level	Success Rate
10	7/10
20	7/10
50	9/10
100	10/10

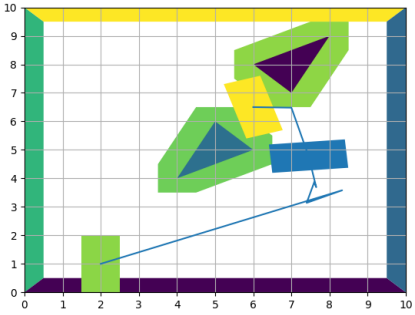


10 Iterations Example

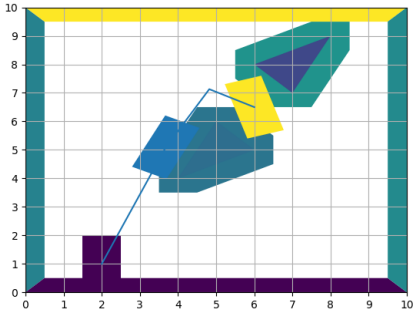


100 Iterations Example

Like in the previous section, success rate and path optimality seemed to increase alongside number of iterations taken. In addition to the search trees taking far less winding paths and not self intersecting, the points also lead the robot rotate less along its path as shown below.



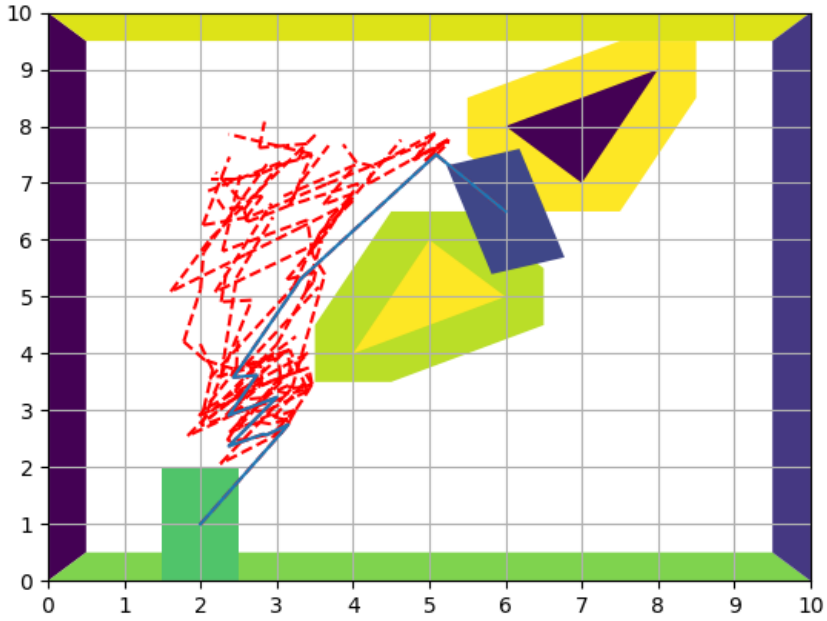
RRT Turns



RRT* Turns

3 Kinematic Motion Planning for a Holonomic Car

In implementing the modified `extend` method, I chose $n_1 = 0.25$ and $n_2 \geq 3$ (with $dt = 0.25$) along with $u_1 = (-3, -\frac{\pi}{4})$ and $u_2 = (3, \frac{\pi}{4})$. My reasoning behind the duration number choices (and step size) was because I wanted to the robot to be able to take precise steps along its path but also not cover too little ground. My choices for the controls also were due to this, but also because I wanted to restrain the motion of the robot so it did not wander around too much as in the previous sections with RRT.
The effects of my variable choices can be observed in the visual below:



30 Iteration RRT

The tree mainly tends to grow towards one side of the obstacle in this environment, potentially due to my rotation restraints on the robot. There also appears to be larger amount of nodes with shorter distances between them when the robot trajectory approches or tries to wrap around a nearby obstacle. It also appears to greatly change its position shortly after its motion becomes clear of said obstacles.

RRT Performance

Iteration Level	Success Rate
10	3/10
20	4/10
50	7/10*
100	8/10

* Performance at iteration level 50 was very inconsistent.

Success rate did appear to increase with a greater number of iterations, though they are lower when compared to their respective counterparts in the previous sections.