HW 4: Path Planning with RRTs

In this assignment we'll explore a fundamental problem in robotics: path planning. That is, the problem of moving a robot from an initial configuration A to a goal configuration B, while avoiding obstacles along the way.

You can think of this as a literal path planning problem where a mobile base must navigate to a goal, or in a more abstract application, such as finding a sequence of joint angles that will move an arm from one position to another.

This problem has two primary issues that make it difficult to apply shortest-path algorithms from graph theory, like Djikstra's or A-star:

1. The search space is continuous: we dont have a predefined set of discrete nodes to search through. Any kind of discretized planning will require us to create our own nodes.
2. There may not be an accurate measure of distance. For example, how do we determine the distance between two configurations for a robot arm? Do we look at the end effector pose? The joint angles? There is no clear answer.

1 RRT

One method to solve this problem is an algorithm called Rapidly-exploring Random Trees (RRT). The algorithm itself is quite simple, and only has 1 hyperparameter to tune!

Consider a 2D search problem where we must find a path from a starting point N_{start} to a goal point N_{goal}, while avoiding the obstacles. To do so, we must build a path (a sequence of points) connecting N_{start} to N_{goal}. We'll be assuming a euclidean distance function for our implementation.

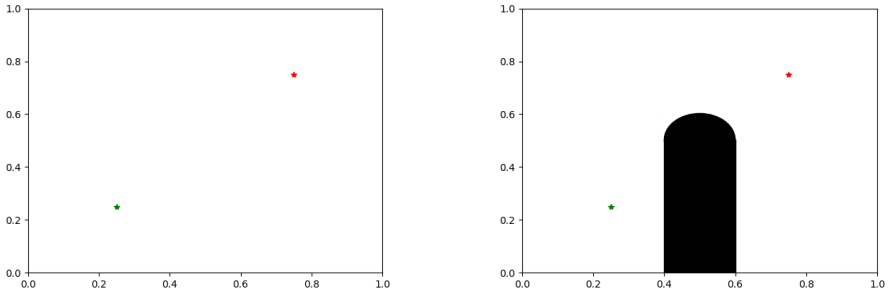
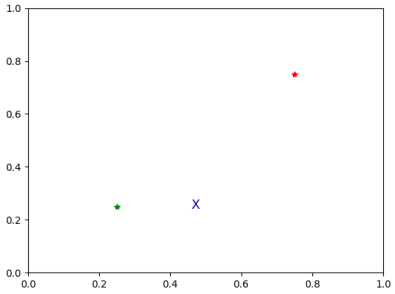


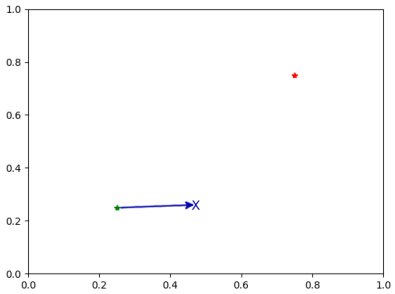
Figure 1: A 2D RRT search space with no obstacles (left) and a single obstacle (right). N_{start} is in green, while N_{goal} is in red.

An RRT is an iteratively built tree with clever use of random sampling that is likely (though not guaranteed) to build one such path from N_{start} to N_{goal}.

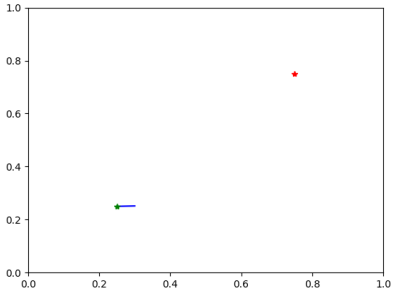
We first randomly sample a point in the search space. Let's call it P_{sample}.



We then compute the direction vector between the closest node N_{close} in our search tree and P_{sample}.



Now we create a new node that is a fixed distance δ away from N_{close} along our direction vector, which we'll call N_{new}.



If N_{new} is not in collision with an obstacle, we add it to our search tree.

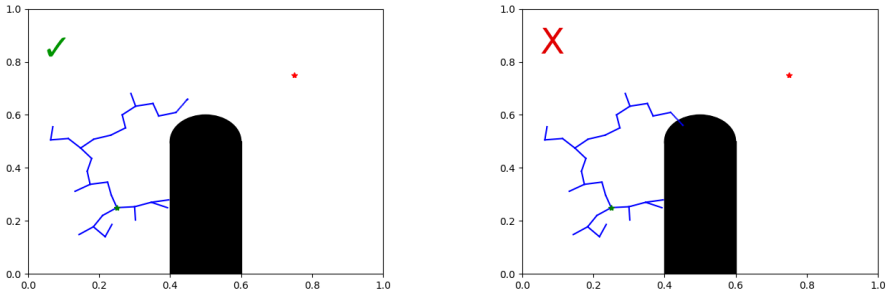


Figure 2: A valid N_{new} will not collide with any obstacles (left). A N_{new} that does collide (right) should be discarded.

After every new addition, we check if that point is within δ of the goal position. If so, we connect to the goal and draw our path.

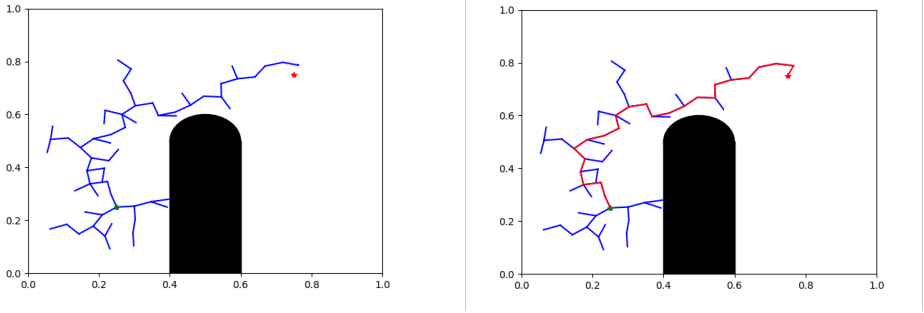


Figure 3: Once N_{new} is close enough to N_{goal}, connect the two nodes and trace the path from N_{start} to N_{goal}.

Note that the path we build is quite convoluted. RRT will only build a valid path, not necessarily the shortest path! The algorithm provides no proof of optimality, but it is relatively fast to compute compared to other path planning algorithms.

The example above is restricted to a 2-dimensional search problem, but the code you will write will be able to work across N-dimensional search spaces.

2 RRT

Implement the empty methods in the RRT, CollisionBox, and CollisionSphere classes in `src/rrt.py` and `src/collision.py`. The skeleton code lays out the basic structure, alongside a Node implementation you may find helpful as a data structure. You are free to modify the RRT class as needed, but you MUST implement the methods provided.

3 Testing

Unit tests are provided to help you check and bugfix your implementation. Look through the tests in `test/test_rrt.py` to understand what each unit test is looking for.

You can run unit tests for your RRT with the following commands:

```
cd code/rrt
# Run all tests for RRT
nose2 test.test_rrt
# Run a single test class
nose2 test.test_rrt.TestRRTInit
# Run a single test case
nose2 test.test_rrt.TestRRTInit.test_rrt_init_basic
```

The `TestRRTBuild` test case will run an end-to-end test of your RRT. Make sure you've passed this final test case before moving on.

4 Visualization

To get a visual understanding of what your RRT is doing, we have implemented a special case of the RRT for 2-dimensional search problems with a visualization method.

Finish the implementation in `src/planar_rrt.py` and examine the unit test in `test/test_planar_rrt.py`. You can change the input values to experiment with your PlanarRRT, but first try out the default configuration. You should see a visualization similar to the ones illustrated in this document!

```
# Run the PlanarRRT
nose2 test.test_planar_rrt
```

5 Questions

In the pdf file `answers.pdf` answer each of the following questions in a couple sentences.

1. Why is the path returned by the RRT not guaranteed to be optimal (i.e. not the shortest feasible path)?
2. What effect will increasing δ have on the performance of the RRT?

3. What effect will increasing the bounds of the search space have on the performance of the RRT? How about increasing the number of dimensions of the search space?

4. Why is it important to have a relatively small δ? Hint: think about what would happen if we have lots of small obstacles in our search space