Introduction

In this notebook, we explore the problem of path planning within a defined environment using the Rapidly-exploring Random Tree (RRT) algorithm. The environment is a 10 x 10 square the goal and starting state predefined. These conditions can be changed freely. In this script, the object being contolled will be three dimensional, x,y and theta.

By Seif & Mark

Importing the necessary libraries

In [53]:
```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle
import numpy as np
import random
from scipy.spatial import distance
```

Initializing the environment and the L Shape

In [54]:
```
# Setting the environment dimensions, 10 by 10 square
x_min, x_max, y_min, y_max = 0, 10, 0, 10
initial_orientation = 0

# Initializing the goal region
goal_center = (9, 9)
goal_radius = 0.5

# Initializing the obstacles
obstacles = [
    ((3.5, 6), 0.5, 4),  # Rectangle (3.5,10),(4,10),(4,6),(3.5,6)
    ((3.5, 0), 0.5, 4)   # Rectangle (3.5,4),(4,4),(4,0),(3.5,0)
]
# Initializng the L shape
L_shape = np.array([
    [0, 0],         # Top-left corner
    [0.3, 0],       # Top-right corner of the vertical part
    [0.3, -2.7],    # Inner corner point of the joint
    [3, -2.7],      # Upper-right corner of the horizontal part
    [3, -3],        # Bottom-left corner of the horizontal part
    [0, -3]         # Bottom-left corner of the vertical part
])
```

Defining the RRT helper functions needed

In [55]:
```
# Plotting the environment
def plot_environment(ax, nodes, final_node=None, L_shape=[]):
```

```
    # Plotting all the obstacles
    for obstacle in obstacles:
        ax.add_patch(Rectangle(obstacle[0], obstacle[1], obstacle[2], color='r

    # Ploting the goal region
    ax.add_patch(Circle(goal_center, goal_radius, color='green', alpha=0.5))

    # Plotting all nodes and paths
    for node in nodes:
        if node.parent:
            ax.plot([node.x, node.parent.x], [node.y, node.parent.y], 'r-')

    # Highlight path to goal if found
    if final_node:
        path_node = final_node

        while path_node.parent:
            ax.plot([path_node.x, path_node.parent.x], [path_node.y, path_node
            path_node = path_node.parent

    # Draw the L-shaped object for each node
    for node in nodes:
        draw_L_shape(ax, node.x, node.y, node.theta, L_shape)

    ax.plot(nodes[0].x, nodes[0].y, 'bo')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('RRT Path Planning with L-shaped Object')
    plt.grid(True)
    plt.show()

def draw_L_shape(ax, x, y, theta, L_shape):

    # Create the transformation matrix for the L-shape, generic tranformation m
    transform = np.array([[np.cos(theta), -np.sin(theta), x],
                          [np.sin(theta), np.cos(theta), y],
                          [0, 0, 1]])

    # Make the dimensions match for calcualtion
    L_shape_homogenized = np.hstack((L_shape, np.ones((L_shape.shape[0], 1))))

    # Apply the transformation to each point of the L-shape
    transformed_shape = (transform @ L_shape_homogenized.T).T

    # Extract the transformed vertices and plot the L-shape
    xs, ys = transformed_shape[:, 0], transformed_shape[:, 1]
    ax.fill(xs, ys, alpha=0.5, closed=True)

# Defining each node with x, y, and theta
class Node:

    def __init__(self, x, y, theta):
        self.x = x
        self.y = y
        self.theta = theta
        self.parent = None

def is_inside_any_obstacle(x, y, theta, L_shape, obstacles):
```

```
    # Create the transformation matrix for the L-shape from above
    transform = np.array([[np.cos(theta), -np.sin(theta), x],
                          [np.sin(theta), np.cos(theta), y],
                          [0, 0, 1]])

    # Make the dimensions match for calcualtion
    L_shape_homogenized = np.hstack((L_shape, np.ones((L_shape.shape[0], 1))))

    # Apply the transformation to each point of the L-shape
    transformed_shape = (transform @ L_shape_homogenized.T).T

    # Function to check if two line segments intersect
    def line_segments_intersect(p1, p2, q1, q2):
        def ccw(a, b, c):
            return (c[1] - a[1]) * (b[0] - a[0]) > (b[1] - a[1]) * (c[0] - a[0]
        return ccw(p1, q1, q2) != ccw(p2, q1, q2) and ccw(p1, p2, q1) != ccw(p1

    # Check each edge of the L-shape for collision with each obstacle edge
    for i in range(len(transformed_shape) - 1):
        l_start = transformed_shape[i, :2]
        l_end = transformed_shape[i + 1, :2]

        for obstacle in obstacles:
            (ox, oy), width, height = obstacle
            # Define the four corners of the rectangle obstacle
            rect_corners = [(ox, oy), (ox + width, oy), (ox + width, oy + heigh

            # Check each edge of the rectangle
            for j in range(4):
                o_start = rect_corners[j]
                o_end = rect_corners[(j + 1) % 4]  # Wrap around to the first c

                if line_segments_intersect(l_start, l_end, o_start, o_end):
                    return True  # Collision detected between L-shape edge and

    return False


# Function for random sampling with orientation and goal biasing
def random_sample():
    if random.random() > 0.1:
                return random.uniform(x_min, x_max), random.uniform(y_min, y_ma
    else:
        return goal_center[0], goal_center[1], random.uniform(0, 2*np.pi)

# Function to find the nearest node
def nearest_node(nodes, sample):
    return min(nodes, key=lambda node: distance.euclidean((node.x, node.y), (sa
# Function to Choosing where the new node will be created
def steer(x_nearest, y_nearest, theta_nearest, x_sample, y_sample, theta_sample(
    # Calculate the direction and distance to the sampled point
    direction_to_sample = np.arctan2(y_sample - y_nearest, x_sample - x_nearest
    distance_to_sample = np.hypot(x_sample - x_nearest, y_sample - y_nearest)

    # Check if the sampled point is within the goal bias threshold distance to
    if dist.euclidean((x_sample, y_sample), goal_center) < goal_bias_threshold
        # Steer directly towards the goal if within goal bias threshold
        direction = np.arctan2(goal_center[1] - y_nearest, goal_center[0] - x_n
        distance = min(delta_distance, dist.euclidean((x_nearest, y_nearest), g
    else:
```

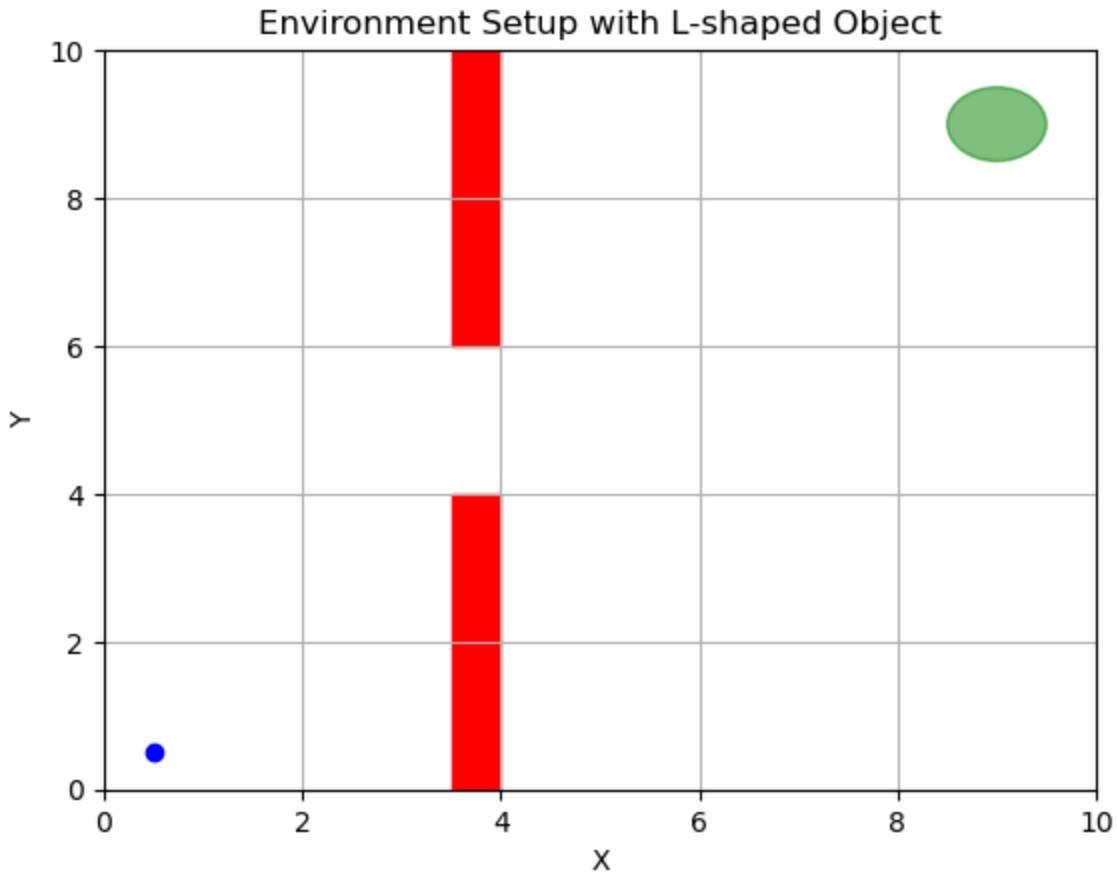
```
        # Steer towards the sampled point otherwise
        direction = direction_to_sample
        distance = min(delta_distance, distance_to_sample)

    # Calculate the new position
    x_new = x_nearest + distance * np.cos(direction)
    y_new = y_nearest + distance * np.sin(direction)
    # For theta, you can choose to keep it aligned with the direction of moveme
    theta_new = (theta_nearest + np.arctan2(np.sin(theta_sample - theta_neares

    return x_new, y_new, theta_new




# Function to check if a point is inside the goal region
def is_inside_goal(x, y, theta, L_shape):

    # Create the transformation matrix
    transform = np.array([[np.cos(theta), -np.sin(theta), x],
                          [np.sin(theta), np.cos(theta), y],
                          [0, 0, 1]])

    # Make the dimensions match for calcualtion
    L_shape_homogenized = np.hstack((L_shape, np.ones((L_shape.shape[0], 1))))

    # Apply the transformation to each point of the L-shape
    transformed_shape = (transform @ L_shape_homogenized.T).T

    # Check if any point of the L-shape is within the goal region
    for point in transformed_shape[:, :2]:
        if distance.euclidean(point, goal_center) <= goal_radius:
            return True
    return False

# The main RRT Function
def rrt(max_iterations=10000, delta_distance=0.5, delta_theta=np.pi/18):
    nodes = [Node(0.5, 0.5, initial_orientation)]  # Start with the starting n

    for _ in range(max_iterations):
        x_rand, y_rand, theta_rand = random_sample()
        nearest = nearest_node(nodes, (x_rand, y_rand, theta_rand))
        x_new, y_new, theta_new = steer(nearest.x, nearest.y, nearest.theta, x_

        if not is_inside_any_obstacle(x_new, y_new, theta_new, L_shape, obstac
            new_node = Node(x_new, y_new, theta_new)
            new_node.parent = nearest
            nodes.append(new_node)

            if is_inside_goal(x_new, y_new, theta_new, L_shape):
                return nodes, new_node  # Path found

    return nodes, None  # Path not found within max_iterations
```

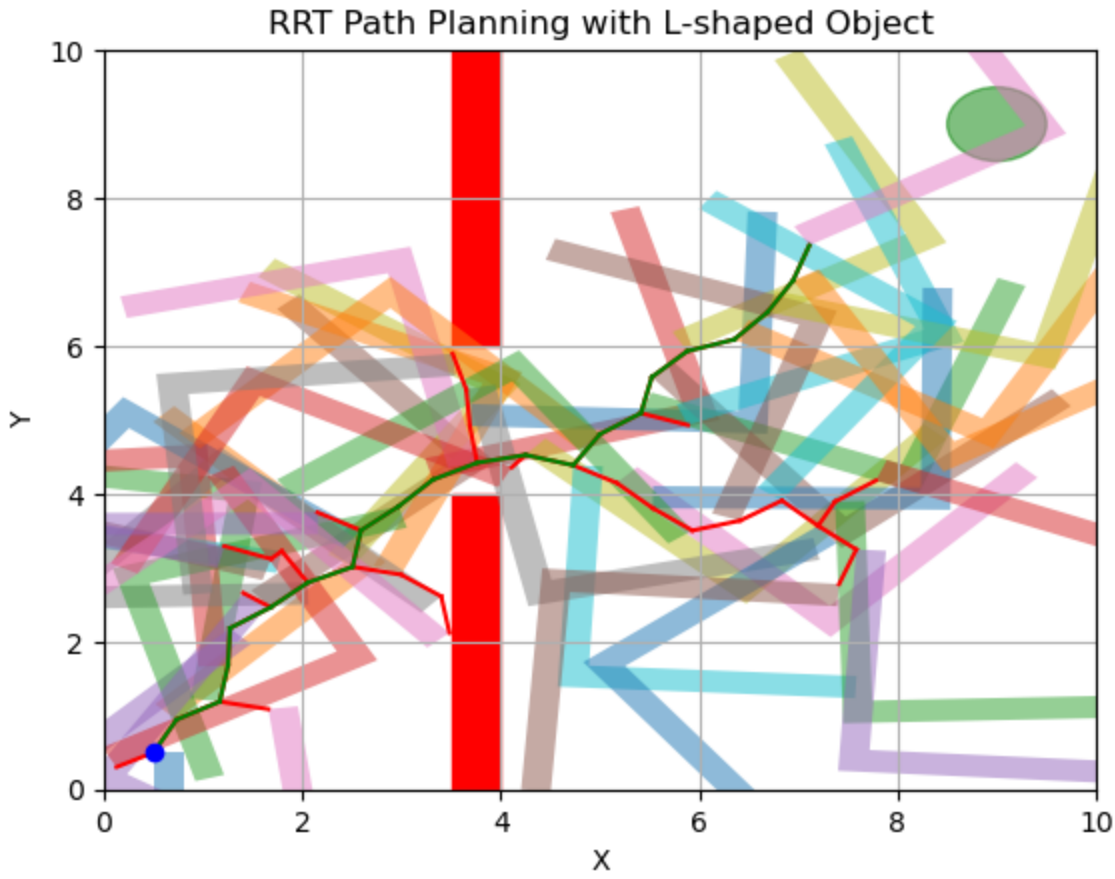
Running the RRT algorithim

In [56]:
```
# Running the RRT algorithm
nodes, final_node = rrt()
```

Showing visual results of RRT Algorithim

In [57]:
```
# Plotting the environment with just the obstacles
fig, ax = plt.subplots()

for obstacle in obstacles:
    ax.add_patch(Rectangle(obstacle[0], obstacle[1], obstacle[2], color='red'))
ax.add_patch(Circle(goal_center, goal_radius, color='green', alpha=0.5))
ax.plot(0.5, 0.5, 'bo')  # Initial state
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Environment Setup with L-shaped Object')
plt.grid(True)
plt.show()

# Plotting the result
fig, ax = plt.subplots()

plot_environment(ax, nodes, final_node, L_shape)
```

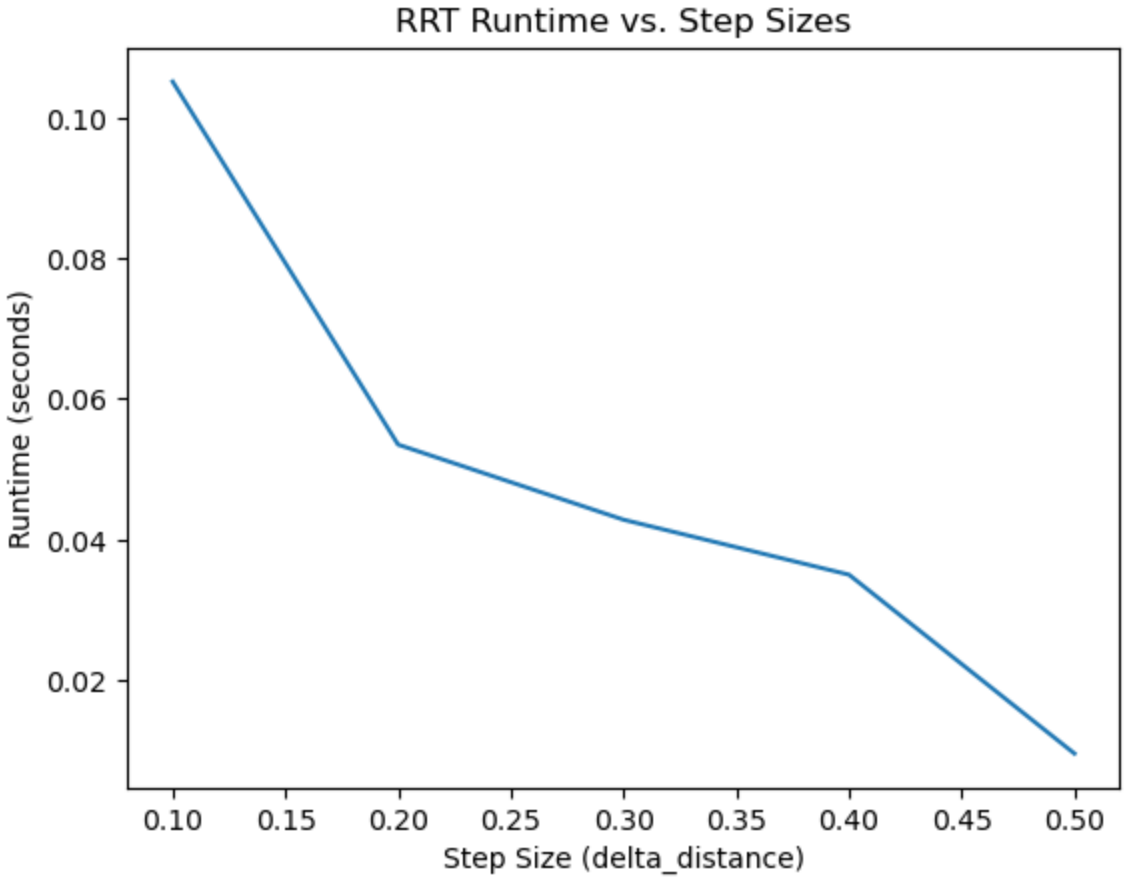




Comparing Runtime in relation to step size

In [59]:
```
def run_experiments(delta_distance_list, max_iterations=1000):
    results = []

    for delta_distance in delta_distance_list:
        start_time = time.time()
        # Call the rrt function without the 'obstacles' parameter
        nodes, final_node = rrt(max_iterations=max_iterations, delta_distance=(
        end_time = time.time()

        runtime = end_time - start_time
        path_found = final_node is not None

        results.append({
            'delta_distance': delta_distance,
            'runtime': runtime,
            'path_found': path_found
        })

    return results

# The obstacles
obstacles = [((3.5, 6), 0.5, 4), # Rectangle (3.5,10),(4,10),(4,6),(3.5,6)
             ((3.5, 0), 0.5, 4)] # Rectangle (3.5,4),(4,4),(4,0),(3.5,0)

# Choosing step sizes to compare
delta_distance_list = [0.1, 0.2, 0.3, 0.4, 0.5]
```

```
# Running the experiment function
experiment_results = run_experiments(delta_distance_list)

# Converting the results to a DataFrame
df_results = pd.DataFrame(experiment_results)

# Plotting the results
sns.lineplot(data=df_results, x='delta_distance', y='runtime')
plt.title('RRT Runtime vs. Step Sizes')
plt.xlabel('Step Size (delta_distance)')
plt.ylabel('Runtime (seconds)')
plt.show()
```



Conclusion

This problem posed a much greater challange, as the function that we used to determine whether or not a node is allowed became very involved, as all the dimensional aspects of the object needed to be accounted for. With that, it became increasing difficult for the object to find its way toward the goal in time, so a heaveir goal bias was introduced into the steering function.