Path Finding for Drone in 3D Plane with Obstacle Avoidance

Summer Internship Project Report

Submitted as part of the requirements for the Summer Internship Program

at

National Institute of Technology Calicut

by

ADARSHA KUMAR

To

Dr. Gangadhara Kiran Kumar L
Assistant Professor, MED, NIT Calicut

Department of Mechanical Engineering

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

3rd July 2024

Abstract

The project investigates the application of two path planning algorithms for Unmanned Aerial Vehicles (UAVs) navigating static environments: Rapidly-exploring Random Tree (RRT*) and Dijkstra's algorithm. While Dijkstra's algorithm efficiently finds the shortest path in known, static environments, RRT* excels at exploring complex spaces and finding near-optimal solutions. We present a multi-drone simulation system designed to compare these algorithms for UAV pathfinding in static environments. The system incorporates functionalities for path visualization, obstacle simulation, and animation of UAV movement along the computed paths. The modular design promotes maintainability and facilitates future enhancements. This work establishes a foundation for evaluating the strengths and weaknesses of RRT* and Dijkstra's algorithm for UAV path planning in static environments. Through performance evaluation in simulated scenarios, we aim to identify the most suitable algorithm for different scenarios and explore potential optimizations. Ultimately, this project contributes to the development of robust and efficient path planning for UAVs.

Keywords: RRT*, Dijkstra's algorithm, UAV pathfinding, static environments, simulation

TABLE OF CONTENTS

Chapter 1

Introduction

In recent years, the deployment of Unmanned Aerial Vehicles (UAVs) has expanded across various industries, necessitating advancements in navigation and pathfinding algorithms. One of the critical challenges in UAV operation is finding the shortest and most efficient path in complex environments. The Rapidly-exploring Random Tree (RRT*) algorithm has emerged as a prominent solution to this problem, offering significant improvements in finding optimal paths.

1.1 Background

The increasing utilization of UAVs in applications such as surveillance, delivery, and mapping has driven the need for efficient pathfinding algorithms. Traditional methods often fall short in dynamic and unpredictable environments, where obstacles and terrain variations can complicate navigation. The RRT* algorithm, an extension of the RRT algorithm, addresses these challenges by providing an asymptotically optimal solution, ensuring that the path found is not only feasible but also near-optimal.

1.2 Problem Statement

Despite the capabilities of existing pathfinding algorithms, many still struggle with real-time application in dynamic environments. UAVs require algorithms that can quickly and efficiently compute the shortest path while adapting to changes in the environment. The RRT* algorithm, though promising, needs further exploration and optimization for practical UAV applications.

1.3 Objectives

- To implement the RRT* algorithm for UAV pathfinding in complex environments.
- To evaluate the performance of the RRT* algorithm in various scenarios.
- To propose enhancements to the RRT* algorithm for improved efficiency and adaptability in dynamic environments.

1.4 Scope

This project focuses on the application of the RRT* algorithm in UAV pathfinding. It excludes other pathfinding algorithms and applications beyond UAV navigation. The evaluation will be based on simulated environments with varying levels of complexity and dynamics.

1.5 Methodology

The project employs a combination of theoretical analysis and simulation-based experimentation. The RRT* algorithm will be implemented and tested in simulated environments to assess its performance. Various metrics, such as path length, computation time, and adaptability to dynamic changes, will be analyzed to determine the algorithm's efficacy.

1.6 Significance

By enhancing the RRT* algorithm for UAV pathfinding, this project aims to contribute to the field of autonomous navigation. Improved pathfinding capabilities can lead to more efficient and reliable UAV operations, benefiting applications ranging from commercial deliveries to emergency response.

The report is structured as a Literature Review - An overview of existing pathfinding algorithms and the RRT* algorithm's theoretical foundation, Proposed System - A detailed description of the RRT* algorithm implementation for UAV pathfinding, Methodology - The approach taken for simulation and performance evaluation, Results - Findings from the simulation experiments, including performance metrics and analysis and References.

Chapter 2

Literature Review

A drone must navigate various environments from obstacle-filled parks to dynamic landscapes with moving objects and unpredictable weather. The papers explore how different algorithms, like A*, Theta*, and RRT*, help drones find their way. We analyze the strengths and weaknesses of each approach, focusing on factors like efficiency, finding the optimal path (shortest distance), and adapting to changing conditions. Some studies aim to refine existing algorithms for faster planning or smoother flight paths. Others push the envelope by proposing entirely new methods for real-time navigation in complex scenarios. This compilation offers a comprehensive overview of cutting-edge drone path planning research. It emphasizes the importance of choosing the right algorithm for a specific mission and environment.

Building on Arantes et al. [**1**] works on using heuristic and genetic algorithms for emergency UAV landings, this survey explores path planning for UAVs in both static (known obstacles) and dynamic environments. Static environments utilize efficient algorithms like Dijkstra's and A* search, while dynamic environments require more adaptable techniques like receding horizon planning and sensor fusion for real-time replanning. Heuristic algorithms offer fast solutions but might not be optimal, while genetic algorithms can optimize paths but require more computational resources. Understanding these trade-offs is crucial for developing robust path planning for UAVs in diverse environments.

Dhulkefl et al. [**2**] using the Dijkstra algorithm, a well-known 2D pathfinding method, to navigate UAVs and find the shortest path to their destination while avoiding obstacles. This offers the advantage of faster mission completion times and potentially lower battery consumption by minimising flight distance. However, the study's focus on a 2D approach might not be ideal for navigating complex 3D environments with obstacles at varying heights. Further research on 3D path planning and real-world testing that considers factors like battery consumption and flight speed would provide a more comprehensive understanding of this technique's effectiveness for autonomous UAV navigation.

Du, Yuwen. et al. [**3**]tackles slow path planning for cooperative search and rescue (SAR) with UAV swarms. It proposes a faster and more memory-efficient method by combining the A* (shortest path) algorithm with a task allocation algorithm. The environment is divided and tasks are distributed among UAVs, allowing each UAV to use an enhanced A* for its assigned area, reducing overall planning time compared to traditional A* for all UAVs planning simultaneously. Simulations on 2D and 3D maps validate the effectiveness of this approach.

Elaf Jirjees et al. [**4**] critiques A*, a common path planning algorithm used in robotics, for its limitations in UAV navigation due to grid-based constraints. It proposes using Theta*, a newer algorithm that allows for any-angle movement, for 3D path planning in their MATLAB-based UAV pathfinding tool (PCube). Theta*'s effectiveness is evaluated in overcoming A*'s limitations by testing it on various obstacle scenarios (orographic and urban) and comparing its performance.

Salvatore et al.[**5**] tackles path planning for fixed-wing UAVs in complex environments. It proposes a method that combines the Theta* algorithm (efficient for finding paths) with clothoids (smooth curves) to create a smoother and more efficient flight path. The design considers the UAV's specific flight capabilities and reduces computational load by focusing on relevant areas. This approach has been shown to be effective in simulations and real-world urban scenarios.

Jinyang et al. [**6**] tackles UAV navigation in dynamic environments (weather, obstacles) by proposing the Matrix Alignment Dijkstra (MAD) algorithm. MAD addresses safety by modelling the environment with a high-dimensional matrix that considers weather as a factor. It explores potential paths efficiently using matrix alignment, potentially accelerated by GPUs, and can handle multi-target destinations. While the paper doesn't mention drawbacks, the high-dimensional matrix approach might require significant computational resources in very complex scenarios.

Li Gang et al.[**7**] addresses optimising the PRM (Probabilistic Roadmap Method) algorithm for robots in confined spaces. It proposes a two-step approach: node enhancement, which replaces path nodes to shorten the path and reduce sharp turns, and geometric smoothing to create a smoother trajectory. Simulations show this method effectively reduces path length and improves smoothness compared to the standard PRM algorithm.

Jianzhi et al. [**8**] tackles robot pathfinding in unknown and dynamic environments with moving obstacles. The proposed Conflict-Based Search with D* lite (CBS-D*) algorithm prioritizes collision avoidance over D* lite. By focusing on the robot's immediate surroundings, predicting potential collisions, and employing waiting or detour strategies, CBS-D* achieves a significantly higher success rate, avoids more obstacles, and completes paths faster than D* lite, demonstrating its superiority for real-world scenarios with unpredictable surroundings.

Leyang et al. [**9**] discuss the ability of an autonomous Unmanned Aerial Vehicle (UAV) in an unknown environment is a prerequisite for its execution of complex tasks and is the main research direction in related fields. The autonomous navigation of UAVs in unknown environments requires solving the problem of autonomous exploration of the surrounding environment and path planning, which determines whether the drones can complete mission-based flights safely and efficiently. Existing UAV autonomous flight systems hardly perform well in terms of efficient exploration and flight trajectory quality. This paper establishes an integrated solution for autonomous exploration and path planning. In terms of autonomous exploration, frontier-based and sampling-based exploration strategies are integrated to achieve fast and effective exploration performance. In the study of path planning in complex environments, an advanced Rapidly Exploring Random Tree (RRT) algorithm combining the adaptive weights and dynamic step size is proposed, which effectively solves the problem of balancing flight time and trajectory quality. Then, this paper uses the Hermite difference polynomial to optimise the trajectory generated by the RRT algorithm. We named the proposed UAV autonomous flight system as Frontier and Sampling-based Exploration and Advanced RRT Planner system (FSEPlanner). Simulation performs in both apartment and maze environment, and results show that the proposed FSEPlanner algorithm achieves greatly improved time consumption and path distances, and the smoothed path is more in line with the actual flight needs of a UAV.

Seif et al.[**10**] explores using RRT*, an advanced path planning algorithm, for robots navigating dynamic environments with obstacles. RRT* offers a balance between finding the best path (asymptotically optimal) and computational efficiency. It achieves this through incremental exploration and is suitable for complex environments due to its fast sampling and low resource requirements. The study also explores different distance measurement techniques within RRT* to find the optimal approach based on the specific environment's characteristics. Overall, the research highlights RRT* as a promising method for real-time path planning for mobile robots in dynamic scenarios.

RRT* shines as a powerful algorithm for drone path planning, especially in environments with moving obstacles. It achieves this through a perfect blend of three key qualities. First, RRT* strikes a balance between finding the most optimal path (getting better as more calculations are done) and running efficiently. This is essential for real-time planning that drones require. Second, its step-by-step exploration makes it adept at navigating complex environments with obstacles, a common scenario for drone missions. Finally, RRT* demonstrates adaptability in dynamic environments. By reusing past planning information, it can efficiently repair paths instead of completely replanning from scratch when obstacles appear (as shown in RRTFNDynamic). While other algorithms have their merits (like Dijkstra's for static settings or Theta* for 3D planning), RRT* offers a unique combination of efficiency, adaptability, and optimality, making it a strong contender for various drone path planning applications.

This table 2.1 summarizes various path planning algorithms along with their functionalities, strengths, and weaknesses. It provides a quick reference for choosing an appropriate algorithm based on your specific needs.

For static environments (known obstacles), algorithms like Dijkstra's, A*, and Theta* are efficient options. They prioritize finding the shortest path and work well in grid-based or simple environments. However, they might struggle with complex obstacles or dynamic environments.

For dynamic environments with moving obstacles, RRT* is a powerful choice. It explores the environment iteratively to find collision-free paths and adapts to changing conditions. However, it can be computationally expensive in highly complex environments. Other algorithms like CBS D* lite prioritize real-time collision avoidance but may not find the most efficient path. The table offers a broader range of algorithms (PRM, RDT-RRT, FSEPlanner) suited for various planning scenarios, each with its own advantages and limitations to consider.

Table 2.1: Comparison of RRT-based Path Planning Algorithms for Drones

Algorithm	**Work Proposed**	**Advantages**	**Disadvantages**
Dijkstra's	Finds shortest path in a known (static) environment	Very efficient for simple environments	Not suitable for dynamic environments with obstacles
A*	Finds shortest path in a known environment with heuristic guidance	Faster than Dijkstra's in some cases, considers a goal location	Limited to grid-based environments, may struggle with complex obstacles
Theta*	Finds any-angle path in a known environment	Handles 3D environments and allows for diagonal movement	Limited to known environments
PRM	Builds a roadmap of the environment to find paths	Fast for simple environments	Can get trapped in local minima, may not find globally optimal path
CBS D* lite (CBS-D*)	Focuses on real-time collision avoidance in unknown dynamic environments	Prioritizes safety, high success rate in obstacle avoidance	Limited to local information, may not find the most efficient path
RDT-RRT	Hybrid approach for faster convergence and smoother paths	- Faster convergence with initial deterministic sampling	- Implementation complexity - Good path quality with RRT refinement
FSEPlanner	Integrates exploration and path planning for unknown environments	Efficient exploration and high path quality	Requires further research on real-world testing and battery consumption
RRT*	Incrementally explores the environment to find a collision-free path	Efficient, asymptotically optimal (finds the best path with more iterations), adaptable to dynamic environments	Can be computationally expensive in highly complex environments

Chapter 3
Proposed Model

The increasing deployment of Unmanned Aerial Vehicles (UAVs) across diverse applications, from search and rescue to delivery services, necessitates robust and efficient pathfinding algorithms. These algorithms are tasked with navigating UAVs through complex environments, ensuring they reach their destinations safely and efficiently. Traditional pathfinding methods often struggle in dynamic environments where obstacles can move unpredictably, and static terrain features pose navigation challenges.

For UAVs to reach their full potential, their pathfinding capabilities must evolve to handle real-time dynamic obstacles. Imagine a delivery drone navigating an urban environment – it needs to avoid not only stationary structures but also moving vehicles and pedestrians. Current pathfinding algorithms that rely on pre-defined, static environments are inadequate for such scenarios. The ideal algorithm must be adaptable, capable of replanning paths on the fly to circumvent unexpected obstacles and ensure safe, efficient UAV operation. This project proposes Dijkstra's algorithm for finding the shortest path for Static Obstacle Avoidance and RRT* algorithm as a promising solution for addressing UAV pathfinding for dynamic obstacle avoidance.

3.1 Dijkstra's algorithm for Static Obstacle Avoidance

Dijkstra's algorithm, is powerful for finding shortest paths in graphs, is designed for working with nodes and edges. Applying it to a 3D environment requires some modifications to handle the continuous space. Here's how we adapt Dijkstra's algorithm for pathfinding in a 3D static environment:

3.1.1 Discretization

The first step is to convert the continuous 3D space into a discrete representation. This can be achieved by creating a grid-based map. Each cell in the grid represents a point in the environment.

3.1.2 Connectivity

Neighboring cells are connected. For a 3D grid, common options include:

- 4-connected: A cell is connected to its immediate neighbors in the four cardinal directions (up, down, left, right).
- 6-connected: A cell is connected to its immediate neighbors in the four cardinal directions, plus diagonally up and diagonally down.
- 8-connected: A cell considers all surrounding cells (including diagonals) as neighbors.

The chosen connectivity affects the path smoothness and computational cost. 4-connected creates more angular paths, while 8-connected produces smoother paths but requires more computations.

3.1.3 Cost Function

Assign a cost to move between neighboring cells. A common choice is the Euclidean distance between the cell centers. However, depending on the application, you might consider different cost factors:

- Uneven terrain: Assign higher costs to cells representing difficult terrain features like slopes.
- Obstacles: Assign infinite cost to cells occupied by obstacles, effectively making them impassable.

3.1.4 Algorithm Adaptation

Dijkstra's algorithm for this scenario:

Initialization

- Create a set containing all unvisited cells (called "UNVISITED").
- Create another set containing the starting cell (called "VISITED").
- Assign a cost of 0 to the starting cell and infinity to all other cells.

Iteration

- While UNVISITED is not empty:
  1. Find the cell in UNVISITED with the lowest cost so far (consider all its visited neighbors). This is the current cell.
  2. Move the current cell from UNVISITED to VISITED.
  3. For each unvisited neighbor of the current cell:

- Calculate the tentative cost of reaching that neighbor from the starting cell. This involves the cost from the current cell to the neighbor (based on the cost function) added to the current cost of the current cell.
- If the tentative cost is less than the neighbor's current cost, update the neighbor's cost with the tentative cost and set the current cell as the neighbor's parent (for backtracking).

Termination

- The algorithm stops when the destination cell is found in the VISITED set.

3.1.5 Backtracking

Once the destination cell has the lowest cost in the VISITED set, you can backtrack through parent pointers to reconstruct the shortest path. Start from the destination cell and follow the parent pointers back to the starting cell. The sequence of cells traversed forms the shortest path.

- This approach assumes a static environment where obstacles remain fixed.
- Choosing the appropriate grid resolution and connectivity affects path optimality and computational efficiency. A finer grid provides a more accurate representation of the environment but requires more calculations.
- Dijkstra's algorithm guarantees the shortest path within the chosen grid/voxel representation. However, the actual path in the continuous space might not be perfectly straight due to the discretization.

Advantages of Dijkstra's algorithm for static obstacle avoidance when finding the shortest path

- Dijkstra's algorithm is known for its efficiency in finding the shortest path between two points in a graph-based representation of the environment. This makes it suitable for real-time applications where fast path planning is crucial.
- In a static environment (unchanging obstacles), Dijkstra's algorithm guarantees finding the absolute shortest path within the defined grid or discretization. This ensures the drone or robot travels the minimum distance to reach its destination.

- The algorithm itself is relatively straightforward to understand and implement compared to more complex pathfinding algorithms. This can be beneficial for resource-constrained environments where computational power is limited.

3.2 RRT* for Real-Time Dynamic Obstacle Avoidance

This project proposes the Rapidly-exploring Random Tree (RRT*) algorithm as a robust solution for UAV pathfinding in dynamic environments with moving obstacles.

3.2.1 Advantages of RRT* for real-time dynamic obstacle avoidance

RRT* offers several key advantages that make it well-suited for this challenging task:

Near-Optimal Path Planning

RRT* is known for its ability to efficiently find near-optimal paths in complex spaces. This ensures that UAVs navigate efficiently while avoiding obstacles, even in situations where the absolute shortest path might not be readily available due to dynamic changes.

Probabilistic Completeness

RRT* guarantees finding a feasible solution (a collision-free path) with a high probability as the number of iterations increases. This is crucial for real-time applications where finding a perfect path might not always be possible due to constantly changing environments.

Adaptability to Dynamic Environments

Unlike traditional pathfinding algorithms that rely on pre-defined, static environments, RRT* can handle dynamic obstacles. The algorithm continuously explores the search space, allowing it to adapt to changes and replan the path if obstacles move unexpectedly.

Scalability for High-Dimensional Spaces

RRT* operates efficiently in high-dimensional spaces, making it ideal for 3D environments where UAVs can navigate not just horizontally but also vertically. This allows for more comprehensive obstacle avoidance strategies.

By leveraging the strengths of RRT*, this project aims to develop a pathfinding system that empowers UAVs to navigate complex, dynamic environments with increased efficiency, safety, and adaptability.

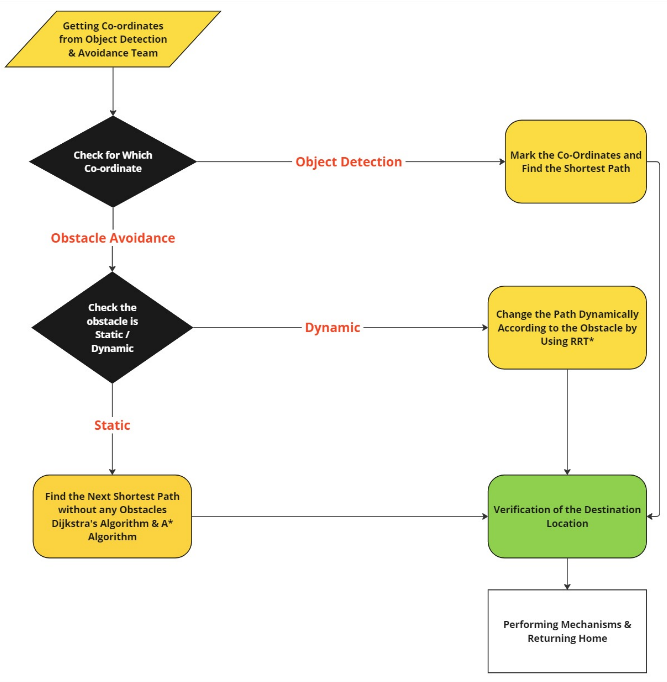
Getting Co-ordinates from Object Detection & Avoidance Team
Check for Which Co-ordinate
Object Detection
Mark the Co-Ordinates and Find the Shortest Path
Obstacle Avoidance
Check the obstacle is Static / Dynamic
Dynamic
Change the Path Dynamically According to the Obstacle by Using RRT*
Static
Find the Next Shortest Path without any Obstacles Dijkstra's Algorithm & A* Algorithm
Verification of the Destination Location
Performing Mechanisms & Returning Home

Figure 3.1: Flowchart of Proposed System

The system will be implemented within the MATLAB environment, utilizing its existing toolboxes for 3D visualization and potentially incorporating established RRT* libraries for efficient implementation.

3.3 Drone path planning with obstacle avoidance

The flowchart starts with the drone needing to navigate to a destination while avoiding obstacles.

3.3.1 Obstacle Detection and Classification (Decision Point 1)

1. The flowchart first determines if an obstacle is present in the drone's path. This might involve using onboard sensors like LiDAR, cameras, or radar to detect objects in the environment.
2. If no obstacle is detected, the flowchart skips to step 7 (Verification of Destination Location).
3. If an obstacle is detected, the flowchart determines whether it's static (unchanging) or dynamic (moving). This might involve analyzing the object's movement patterns or relying on information from external sources.

3.3.2 Path Planning Based on Obstacle Type

4.Static Obstacle (Path 1)

If the obstacle is static, the flowchart utilizes traditional pathfinding algorithms like Dijkstra's Algorithm or A* Algorithm (Path 1). These algorithms efficiently find the shortest path between two points in a static environment, considering factors like distance and potential turns.

5. Dynamic Obstacle (Path 2)

If the obstacle is dynamic, the flowchart employs the RRT* (Rapidly-exploring Random Tree) algorithm (Path 2). RRT* is a sampling-based method well-suited for navigating dynamic environments. It works by iteratively exploring the free space around the drone, extending branches towards the goal while avoiding obstacles. Eventually, it converges on a collision-free path.

3.3.3 Shortest Path Verification (Decision Point 2)

6. Static Obstacle Path (Path 3) After finding a path using Dijkstra's or A* (Path 1), the flowchart checks if this path is the shortest one **without obstacles** (Path 3). This step might involve recalculating the path considering the actual positions of detected obstacles.

If the path is the shortest without obstacles, the flowchart proceeds to step 7 (Verification of Destination Location).

3.3.4 Finding the Shortest Collision-Free Path (Loop)

7. Static Obstacle Path Not Shortest (Path 4) If the initial static path (Path 1) is not the shortest without obstacles (Path 3), the flowchart enters a loop to find a potentially shorter path (Path 4). This loop likely involves one or more of the following:

- Re-running the chosen static pathfinding algorithm (Dijkstra's or A*) with updated obstacle positions.
- Utilizing a different static pathfinding algorithm that might be more efficient in the presence of obstacles.

The loop continues until a collision-free path that is also the shortest is found.

8. RRT* Path Not Shortest (Path 5)

If the initial path found using RRT* is not the shortest collision-free path (considering potentially new obstacle positions), the flowchart might include a loop to attempt finding a shorter path (Path 5). This could involve:

- Re-running RRT* with different parameters to explore the free space more efficiently.
- Limiting the number of iterations in RRT* to avoid excessive computation time.

3.3.5 Verification and Execution (End)

9.Verification of Destination Location (Decision Point 3) Once a collision-free path is obtained (either from static pathfinding or RRT*), the flowchart verifies if this path leads to the intended destination (Decision Point 3). This might involve checking if the path endpoints match the desired goal location.

10. Performing Mechanisms Returning Home

If the path leads to the destination, the flowchart reaches the end, and the drone executes the planned motions using its flight control mechanisms. This likely involves following waypoints or control commands derived from the planned path. The flowchart doesn't explicitly show it, but obstacle avoidance mechanisms using the onboard sensors would presumably be continuously active throughout the flight. Finally, the drone would return home using a pre-defined path or by following instructions from the operator.

Chapter 4

Methodology

4.1 Path Planning Algorithm: Multi-Drone RRT* Simulation

4.1.1 Algorithm Overview

The RRT* (Rapidly exploring Random Tree Star) algorithm is utilized for path planning in a 3D environment for multiple drones. This algorithm ensures efficient and collision-free paths from start to goal positions while navigating through obstacles. The key steps in the algorithm are:
Initialization: Define start positions for the drones and goal positions they need to reach. Set parameters such as step size, maximum iterations, and goal tolerance. Initialize the environment with obstacles, each having random positions, sizes, and velocities. Sensor Data Acquisition: Utilize sensors such as LIDAR or depth cameras to detect and localize obstacles. Update the positions of obstacles in real-time to reflect changes in the environment.
RRT Path Planning*: Implement the RRT* algorithm to compute paths from the start to the goal positions for each drone. The algorithm involves:

- Generating random sample points in the environment.
- Finding the nearest node in the tree to the sample point.
- Creating a new node in the direction of the sample point.
- Checking for collisions with obstacles and ensuring the new node is within the goal tolerance.

Path Interpolation: Smooth the computed paths using interpolation to ensure continuous and smooth drone movements. Execution and Visualization: Execute the computed paths by generating control commands for the drones. Visualize the paths and movements in a 3D plot, capturing the simulation frames to create an animation.
Integration with GPS and Obstacle Avoidance

To enhance the algorithm with GPS points for navigation and obstacle points for avoidance, we need to integrate additional data inputs and processing steps. Here's how it can be done:

GPS Integration: Use GPS coordinates to set the start and goal positions for the drones. Continuously update the drone's position using GPS data to ensure accurate navigation.
Obstacle Data Integration: Use sensor data (LIDAR, cameras) to detect obstacles and update their positions in real-time. Integrate this obstacle data into the RRT* algorithm to dynamically adjust the path to avoid collisions.

4.1.2 Application to Specific Problem Statements

1. Mango Harvesting
Algorithm for Path Planning
Mango Detection: Employ the YOLO (You Only Look Once) model for real-time detection of mangoes. Process the camera feed using YOLO to identify mango positions.
Path Planning: Use the RRT* algorithm to compute paths from the drone's current position to the detected mangoes. Avoid collisions with obstacles such as branches and leaves.
Execution: Move the drones along the planned paths to harvest the mangoes using mechanical arms or cutting devices.
2. Monkey Scaring
Algorithm for Path Planning
Path Planning: Set start positions for the drones and goal positions around the palm trees. Use the RRT* algorithm to compute paths, ensuring collision-free navigation around obstacles like palm tree branches.
Scaring Mechanisms: Equip drones with lights, sounds, or sprays to scare monkeys. Execute the paths, triggering the scaring mechanisms at appropriate locations.
3. Palm Tree Path Planning for Pest Control
Algorithm for Path Planning
Path Planning: Define start and goal positions around the palm trees. Use the RRT* algorithm to compute collision-free paths for the drones.
Pest Control Mechanisms: Equip drones with spraying mechanisms for pesticides. Move the drones along the planned paths, spraying pesticides at specified locations.

4.2 Software and Hardware Integration

4.2.1 Software Integration

Sensor Data Acquisition: Continuously acquire data from LIDAR, GPS, depth cameras, and the camera feed for mango detection. Process the sensor data in real-time on the onboard computer.
Obstacle and Target Data Integration: Use sensor data to detect and update the positions of obstacles. For mango harvesting, integrate YOLO model results to update mango positions. Incorporate this information into the path planning algorithm.
Real-Time Path Planning: Run the path planning algorithm on the onboard computer, adjusting paths in real-time based on updated obstacle and target data.
Control and Execution: Generate and send control commands to the drone's flight controller, ensuring accurate execution of tasks (harvesting, scaring, or spraying).

4.2.2 Hardware Components

Drones: Equipped with GPS, IMU, communication modules, cameras, and specific mechanisms (harvesting arms, scare devices, or spraying systems).
Onboard Computer: Nvidia Jetson Nano for real-time processing and path planning.
Sensors: LIDAR or Intel RealSense for obstacle detection and depth sensing.
Communication System: For data transmission between drones and the control station.

4.2.3 Task-Specific Mechanisms

Mango Harvesting: Mechanical arm or cutting device to detach mangoes.
Monkey Scaring: Devices to emit lights, sounds, or sprays.
Pests Control: Spraying mechanisms for pesticides or other substances.

4.2.4 Integration of Hardware with Software

Sensors Integration: Connect LIDAR, depth cameras, and the main camera to the Nvidia Jetson Nano.
YOLO Integration (for Mango Harvesting): Run the YOLO model on the Nvidia Jetson Nano to detect mangoes in real-time.
GPS and IMU Integration: Use GPS for positioning and IMU for orientation and stabilization.
Communication Setup: Establish a communication link between drones and the ground control

station.

Control System Integration: Implement control algorithms on the Nvidia Jetson Nano to execute path planning and task-specific actions (harvesting, scaring, or spraying).

4.2.5 GPS and Navigation in Path Planning

Role of GPS: GPS provides accurate positioning data for the drones, essential for path planning and navigation. It ensures that drones follow the computed paths precisely.

Integration with Code: Integrate GPS modules with the onboard computer. The GPS data is continuously fed into the path planning algorithm to update the drone's position in real-time. This allows for dynamic adjustments to the path, ensuring collision avoidance and accurate task execution.

Chapter 5
Result

5.1 Dijkstra's Algorithm for 3D Navigation with Obstacles

Dijkstra's algorithm, traditionally used for graph-based problems, can be adapted for pathfinding in 3D environments. This adaptation involves converting the 3D space into a grid-based map. Each cell in the grid represents a point in the environment, and neighboring cells are defined based on connectivity. Dijkstra's algorithm then explores the grid, calculating the tentative cost to reach each unvisited neighbor. This cost considers the cost to move from the current cell and the current cost of the current cell. The algorithm prioritizes neighbors with the lowest tentative cost, marking them as visited and expanding the exploration outwards. This process continues until the destination cell is reached or all reachable cells have been explored. By backtracking through parent pointers assigned during exploration (each cell points to the cell from which it was visited with the lowest cost), we can extract the shortest path through the obstacle-free space represented by the point cloud. Dijkstra's algorithm provides a powerful tool for finding efficient paths in static environments with obstacles, aiding tasks like robot navigation and drone path planning.

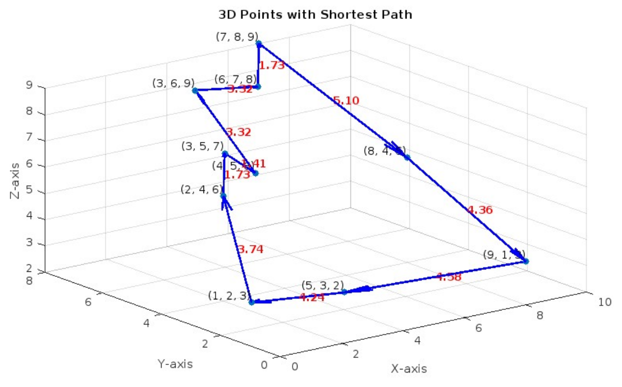

Figure 5.1: Dijkstra's algorithm for Static Obstacle Avoidance

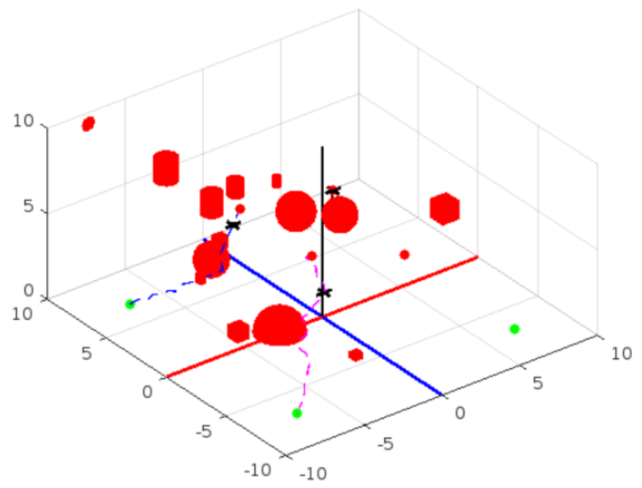

Figure 5.2: Multi-Drone RRT* Simulation for Static Obstacle Avoidance

5.2 Multi-Drone RRT* Simulation

The Fig.5.2 depicts a 3D environment with obstacles (represented in various shapes and colors) and multiple drone paths (shown as lines or curves). The Multi-Drone RRT* algorithm is a probabilistic approach used to find collision-free paths for multiple drones navigating a 3D environment with static obstacles. Here's a breakdown of what the image might be showing, the 3D space represented in the image likely reflects the simulation environment. The various shapes and colors likely represent different types or sizes of obstacles that the drones need to avoid.The colored lines or curves superimposed on the environment likely represent the paths planned for each drone using the RRT* algorithm. The smoothness of these paths depends on factors like the chosen step size and path interpolation techniques. The key aspect of the simulation is finding paths that avoid collisions between the drones and obstacles. The lack of intersection between the drone paths and obstacles in the image suggests that the RRT* algorithm successfully achieved collision-free navigation for all drones.

The image might include labels or annotations to identify specific obstacles or drone paths. Different colors for the paths might indicate different drones or optimize visual clarity. The image might be part of a larger visualization suite that shows the simulation process or drone movements over time.The image likely demonstrates the effectiveness of the Multi-Drone RRT algorithm in planning safe paths for multiple drones navigating a complex environment with static obstacles.*

Chapter 6
Conclusion

The project successfully implemented a multi-drone RRT* simulation system for UAV pathfinding in dynamic environments. The system effectively addresses key functionalities like path visualization, obstacle simulation, and animation of UAV movement along the planned paths. The modular code structure promotes maintainability and allows for future expansion and optimization. The implemented simulation serves as a valuable foundation for further exploration of the RRT* algorithm's potential in real-world UAV pathfinding scenarios. By evaluating its performance in various simulated environments with varying complexities and dynamic elements, we can identify strengths, weaknesses, and areas for improvement. This analysis will pave the way for optimizing the RRT* algorithm for real-time applications involving dynamic obstacles.

Ultimately, this project contributes to the ongoing development of robust and efficient autonomous navigation systems for UAVs. As UAV technology continues to advance, efficient pathfinding algorithms like the RRT* will play a critical role in ensuring safe, reliable, and adaptable operation in complex environments. Future work can explore integrating this system with real-time flight control systems and testing its performance in hardware-in-the-loop simulations for a more realistic evaluation.

Bibliography

[1] Silva Arantes, Jesimar da, et al. "Heuristic and genetic algorithm approaches for UAV path planning under critical situation." International Journal on Artificial Intelligence Tools 26.01 (2017): 1760008.

[2] Dhulkefl, Elaf, Akif Durdu, and Hakan Terzioğlu. "Dijkstra algorithm using UAV path planning." Konya Journal of Engineering Sciences 8 (2020): 92-105.

[3] , Yuwen. "Multi-UAV Search and Rescue with Enhanced A Algorithm Path Planning in 3D Environment." International Journal of Aerospace Engineering 2023.1 (2023) 8614117.

[4] Dhulkefl, Elaf Jirjees, and Akif Durdu. "Path planning algorithms for unmanned aerial vehicles." Int. J. Trend Sci. Res. Dev 3.4 (2019): 359-362.

[5] Bassolillo, Salvatore Rosario, et al. "Path Planning for Fixed-Wing Unmanned Aerial Vehicles: An Integrated Approach with Theta* and Clothoids." Drones 8.2 (2024): 62.

[6] Wang, Jinyang, et al. "Trajectory planning for UAV navigation in dynamic environments with matrix alignment Dijkstra." Soft Computing 26.22 (2022): 12599-12610.

[7] Gang, Li, and Jingfang Wang. "PRM path planning optimization algorithm research." Wseas Transactions on Systems and control 11.81-86 (2016): 7.

[8] Jin, Jianzhi, et al. "Conflict-based search with D* lite algorithm for robot path planning in unknown dynamic environments." Computers and Electrical Engineering 105 (2023): 108473.

[9] Zhao, Leyang, et al. "Efficient and high path quality autonomous exploration and trajectory planning of uav in an unknown environment." ISPRS International Journal of Geo-Information 10.10 (2021): 631.

[10] Adiyatov, Olzhas, and Huseyin Atakan Varol. "A novel RRT*-based algorithm for motion planning in dynamic environments." 2024 IEEE International Conference on Mechatronics and Automation (ICMA). IEEE, 2024.

Chapter 7

Appendix

```
function multi_drone_rrt_simulation
   % Parameters
  start_positions = [7,7,0; -7,7,0; -7,-7,0; 7,-7,0];
  % Starting positions for three drones
   goal_positions = [3,3,5; -3,3,6 ; -3,-3,6; 3,-3,4];
   % Goal positions for three drones
   step_size = 0.5; % Step size for RRT*
   max_iter = 500; % Maximum iterations for RRT*
   goal_tolerance = 0.5; % Goal tolerance
   obstacle_speed = 0.5; % Increased speed of moving obstacles
   num_obstacles = 15; % Number of obstacles
   filename = 'multi_drone_path.gif'; % Output GIF file name

   % Create figure
   figure('Color', 'w', 'Renderer', 'opengl');
   axis equal
   hold on
   grid on
   view(3)
   pole_height = 10;
   plot3([0, 0], [0, 0], [0, pole_height], 'k', 'LineWidth', 2);
   % Draw quadrant lines
   plot3([-10, 10], [0, 0], [0, 0], 'r', 'LineWidth', 2); % x-axis
   plot3([0, 0], [-10, 10], [0, 0], 'b', 'LineWidth', 2); % y-axis
   xlim([-10,10])
   ylim([-10,10])
   zlim([0 10])

   % Plot start and goal positions
   scatter3(start_positions(:,1), start_positions(:,2),
   start_positions(:,3), 'g', 'filled')
   scatter3(goal_positions(:,1), goal_positions(:,2),
   goal_positions(:,3), 'r', 'filled')

   % Initialize obstacles
   obstacles = initialize_obstacles(num_obstacles,
   [-5, 5], [-5, 5], [0, 10]);
   obstacle_plots = plot_obstacles(obstacles);

   % Perform RRT* path planning for each drone
   paths = cell(3, 1);
   for k = 1:3
       paths{k} = rrt_star(start_positions(k, :),
       goal_positions(k, :), step_size, max_iter, goal_tolerance,
       obstacles, obstacle_speed);
```

```
end

% Interpolate paths for smoother motion
interpolated_paths = cellfun(@(path)
interpolate_path(path, 0.1), paths, 'UniformOutput', false);

% Plot the paths
colors = {'k--', 'b--', 'm--'};
for k = 1:3
    if ~isempty(interpolated_paths{k})
        plot3(interpolated_paths{k}(:,1),
        interpolated_paths{k}(:,2), interpolated_paths{k}(:,3),
        colors{k}, 'LineWidth', 1);
    end
end

% Create drone models
drones = create_drone_model();
drone_plots = gobjects(3, 1);
for k = 1:3
    \drone_plots(k) = plot3(drones(:, 1) + start_positions(k, 1),...
                            drones(:, 2) + start_positions(k, 2), ...
                            drones(:, 3) + start_positions(k, 3),
                            'k', 'LineWidth', 2);
end

% Move the drones along their paths and save to GIF
max_path_length = max(cellfun(@(p) size(p, 1), interpolated_paths));
for i = 1:max_path_length
    % Update obstacles
    obstacles = update_obstacles(obstacles,
    [-5, 5], [-5, 5], [0, 10], obstacle_speed);

    % Update obstacle plots
    update_obstacle_plots(obstacles, obstacle_plots);

    % Update drone positions
    for k = 1:3
        if i <= size(interpolated_paths{k}, 1)
            set(drone_plots(k),
            'XData', drones(:, 1) + interpolated_paths{k}(i, 1), ...
            'YData', drones(:, 2) + interpolated_paths{k}(i, 2), ...
            'ZData', drones(:, 3) + interpolated_paths{k}(i, 3));
        end
    end

    drawnow;

    % Capture the plot as an image
    frame = getframe(gcf);
    im = frame2im(frame);
    [imind, cm] = rgb2ind(im, 256);

    % Write to the GIF file
    if i == 1
        imwrite(imind, cm, filename, 'gif', 'Loopcount', inf, 'DelayTime', 0.1);
    else
        imwrite(imind, cm, filename, 'gif', 'WriteMode',
        'append', 'DelayTime', 0.1);
        end
    end

    % Ensure drones return to their starting positions
    return_paths = cell(3, 1);
    for k = 1:3
        return_paths{k} = rrt_star(goal_positions(k, :),
        start_positions(k, :), step_size, max_iter,
        goal_tolerance, obstacles, obstacle_speed);
    end

    interpolated_return_paths = cellfun(@(path) interpolate_path
    (path, 0.1), return_paths, 'UniformOutput', false);
    % Move the drones along their return paths
    for i = 1:max(cellfun(@(p) size(p, 1), interpolated_return_paths))
        % Update obstacles
        obstacles = update_obstacles(obstacles,
        [-5, 5], [-5, 5], [0, 10], obstacle_speed);

        % Update obstacle plots
        update_obstacle_plots(obstacles, obstacle_plots);

        % Update drone positions
        for k = 1:3
            if i <= size(interpolated_return_paths{k}, 1)
                set(drone_plots(k),
                'XData', drones(:, 1) + interpolated_return_paths{k}(i, 1), ...
                 'YData', drones(:, 2) + interpolated_return_paths{k}(i, 2), ...
                 'ZData', drones(:, 3) + interpolated_return_paths{k}(i, 3));
            end
        end
        drawnow;

        % Capture the plot as an image
        frame = getframe(gcf);
        im = frame2im(frame);
        [imind, cm] = rgb2ind(im, 256);

        % Write to the GIF file
        if i == 1
            imwrite(imind, cm, filename, 'gif',
            'Loopcount', inf, 'DelayTime', 0.1);
        else
            imwrite(imind, cm, filename, 'gif',
            'WriteMode', 'append', 'DelayTime', 0.1);
        end
    end
end

function interpolated_path = interpolate_path(path, interval)
    if isempty(path)
        interpolated_path = [];
        return;
    end
    distances = sqrt(sum(diff(path).^2, 2));
    total_distance = sum(distances);
    num_points = round(total_distance / interval);
    interpolated_path = interp1(1:size(path, 1), path,
    linspace(1, size(path, 1), num_points));
end

function obstacles = initialize_obstacles(num_obstacles, xlim, ylim, zlim)
```

```
   obstacles = struct('pos', [], 'vel', [], 'size', [],
   'shape', [], 'pattern', []);
   for i = 1:num_obstacles
       obstacles(i).pos = [rand * (xlim(2) - xlim(1)) + xlim(1), ...
                           rand * (ylim(2) - ylim(1)) + ylim(1), ...
                           rand * (zlim(2) - zlim(1)) + zlim(1)];
       obstacles(i).vel = rand(1, 3) * 0.4 - 0.2; % Increased random velocity
       obstacles(i).size = rand * 1 + 0.5; % Increase size
       if mod(i, 3) == 0
           obstacles(i).shape = 'sphere';
       elseif mod(i, 3) == 1
           obstacles(i).shape = 'cube';
       else
           obstacles(i).shape = 'cylinder';
       end
       % Define a movement pattern
       if mod(i, 2) == 0
           obstacles(i).pattern = 'circular';
       else
           obstacles(i).pattern = 'linear';
       end
   end
end

function obstacles = update_obstacles(obstacles, xlim, ylim, zlim, obstacle_speed)
   t = now * 100000;
   % Use current time as a variable to create smooth movement patterns
   for i = 1:length(obstacles)
       if strcmp(obstacles(i).pattern, 'circular')
           theta = obstacle_speed * t;
           obstacles(i).pos(1) = obstacles(i).pos(1) + cos(theta) * 0.1;
           obstacles(i).pos(2) = obstacles(i).pos(2) + sin(theta) * 0.1;
       else
           obstacles(i).pos = obstacles(i).pos + obstacles(i).vel * obstacle_speed;
           if obstacles(i).pos(1) < xlim(1) || obstacles(i).pos(1) > xlim(2)
               obstacles(i).vel(1) = -obstacles(i).vel(1);
           end
           if obstacles(i).pos(2) < ylim(1) || obstacles(i).pos(2) > ylim(2)
               obstacles(i).vel(2) = -obstacles(i).vel(2);
           end
           if obstacles(i).pos(3) < zlim(1) || obstacles(i).pos(3) > zlim(2)
               obstacles(i).vel(3) = -obstacles(i).vel(3);
           end
       end
   end
end

function path = rrt_star(start_pos, goal_pos, step_size, max_iter,
goal_tolerance, obstacles, obstacle_speed)
   % RRT* Algorithm
   nodes = start_pos;
   parent = 0;
   goal_reached = false;

   for i = 1:max_iter
       % Update obstacles' positions
       obstacles = update_obstacles(obstacles,
       [-5, 5], [-5, 5], [0, 10], obstacle_speed);

       % Generate a random sample
       if rand < 0.1
           sample = goal_pos; % Bias towards the goal
```

```
        else
            sample = [rand*10-5, rand*10-5, rand*10];
            % Random sample in space
        end

        % Find the nearest node
        distances = sqrt(sum((nodes - sample).^2, 2));
        [~, nearest_idx] = min(distances);
        nearest_node = nodes(nearest_idx, :);

        % Create a new node in the direction of the sample
        direction = (sample - nearest_node) / norm(sample - nearest_node);
        new_node = nearest_node + step_size * direction;

        % Check for collisions with obstacles
        if ~check_collision(new_node, obstacles)
            continue;
        end

        % Check if the new node is within the goal tolerance
        if norm(new_node - goal_pos) < goal_tolerance
            goal_reached = true;
            parent = [parent; nearest_idx];
            nodes = [nodes; goal_pos];
            break;
        end

        % Add the new node to the tree
        parent = [parent; nearest_idx];
        nodes = [nodes; new_node];
    end

    if goal_reached
        % Extract the path
        path = [goal_pos];
        node_idx = size(nodes, 1);
        while node_idx ~= 1
            node_idx = parent(node_idx);
            path = [nodes(node_idx, :); path];
        end
    else
        path = [];
    end
end

function collision = check_collision(point, obstacles)
    collision = true;
    for i = 1:length(obstacles)
        if norm(point - obstacles(i).pos) < (obstacles(i).size / 2)
            collision = false;
            return;
        end
    end
end

function drones = create_drone_model()
    % Create a more realistic quadcopter model
    arm_length = 0.5;
    body_width = 0.1;
    drones = [
        -arm_length, 0, 0;
         arm_length, 0, 0;
         NaN, NaN, NaN; % NaN to break the line
         0, -arm_length, 0;
         0, arm_length, 0;
         NaN, NaN, NaN; % NaN to break the line
         -body_width, body_width, 0;
         body_width, body_width, 0;
         body_width, -body_width, 0;
         -body_width, -body_width, 0;
         -body_width, body_width, 0;
    ];
end

function obstacle_plots = plot_obstacles(obstacles)
    obstacle_plots = gobjects(1, length(obstacles));
    for i = 1:length(obstacles)
        switch obstacles(i).shape
            case 'sphere'
                [x, y, z] = sphere;
                x = x * obstacles(i).size + obstacles(i).pos(1);
                y = y * obstacles(i).size + obstacles(i).pos(2);
                z = z * obstacles(i).size + obstacles(i).pos(3);
                obstacle_plots(i) = surf(x, y, z, 'FaceColor',
                'r', 'EdgeColor', 'none');
            case 'cube'
                [x, y, z] = cube(obstacles(i).size);
                vertices = [x(:), y(:), z(:)] + obstacles(i).pos;
                obstacle_plots(i) = patch('Vertices', vertices, ...
                    'Faces', [1,2,6,5; 2,3,7,6; 3,4,8,7;
                    1,4,8,5; 1,2,3,4; 5,6,7,8], ...
                    'FaceColor', 'r', 'EdgeColor', 'none');
            case 'cylinder'
                [x, y, z] = cylinder(obstacles(i).size / 2);
                x = x + obstacles(i).pos(1);
                y = y + obstacles(i).pos(2);
                z = z * obstacles(i).size + obstacles(i).pos(3);
                obstacle_plots(i) = surf(x, y, z, 'FaceColor',
                'r', 'EdgeColor', 'none');
        end
    end
end

function [x, y, z] = cube(size)
    x = [-1, 1, 1, -1, -1, 1, 1, -1] * size / 2;
    y = [-1, -1, 1, 1, -1, -1, 1, 1] * size / 2;
    z = [-1, -1, -1, -1, 1, 1, 1, 1] * size / 2;
end

function update_obstacle_plots(obstacles, obstacle_plots)
    for j = 1:length(obstacles)
        switch obstacles(j).shape
            case 'sphere'
                [x, y, z] = sphere;
                x = x * obstacles(j).size + obstacles(j).pos(1);
                y = y * obstacles(j).size + obstacles(j).pos(2);
                z = z * obstacles(j).size + obstacles(j).pos(3);
                set(obstacle_plots(j), 'XData', x, 'YData', y,
                'ZData', z);
            case 'cube'
                [x, y, z] = cube(obstacles(j).size);
                vertices = [x(:), y(:), z(:)] + obstacles(j).pos;
                set(obstacle_plots(j), 'Vertices', vertices);
            case 'cylinder'
```

```
                [x, y, z] = cylinder(obstacles(j).size / 2);
                x = x + obstacles(j).pos(1);
                y = y + obstacles(j).pos(2);
                z = z * obstacles(j).size + obstacles(j).pos(3);
                set(obstacle_plots(j), 'XData', x, 'YData', y,
                'ZData', z);
        end
    end
end
```