MultiUAV Rescue Simulation: A Comparative Study of PathPlanning & Survivor Assignment

Shivsaransh Thakur

A report submitted in partial fulfilment of the requirements
for the degree of *BSc Computer Science and Mathematics*

The University of Manchester

April 22, 2025

Abstract

Rapid localisation and extraction of survivors is critical in largescale disasters, yet unstable terrain and time pressure routinely hamper human first responders. This project presents a modular simulation framework that coordinates *heterogeneous* unmanned vehiclestwo aerial drones and two tracked ground robotsto accelerate searchandrescue (SAR) missions in cluttered, procedurally generated environments.

The virtual world is represented by a 300×300ăm 2D occupancy grid for ground motion and a 3D voxel map up to 100ăm in height for aerial flight. Each vehicle plans collisionfree trajectories with either Rapidlyexploring Random Trees (RRT) or its optimal variant RRT*, while survivor tasks are distributed by two lightweight heuristics: *nearestneighbour* and *centroid*. Mission performance is evaluated across 96 scenarios (three random seeds Œ two map sizes Œ two planner types Œ two assignment rules) using total rescue time, fraction of survivors reached, and cumulative distance travelled.

Results show that although RRT* shortens average path length by $\approx 8-15\%$, its added planning overhead yields no measurable reduction in overall mission duration. Conversely, the nearestneighbour allocator completes rescues up to $\approx 25\%$ faster than centroid selection in dense obstacle fields. For timecritical SAR, *finding any feasible path quickly* and using a localityaware task allocator matter more than asymptotic path optimality. The framework therefore provides a reproducible baseline for future work on advanced planners, dynamic environments, and hardwareintheloop tests.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Context

Natural and man-made disasters can result in hazardous conditions, making timely rescue operations critical to saving lives. In earthquake scenarios, for example, collapsed buildings create unstable rubble piles, while floods can wash away roads and disrupt communications. Such conditions greatly increase the risk for human first responders and limit how quickly they can traverse the affected area (citation needed). Consequently, there is a growing interest in employing robotic platformsparticularly unmanned aerial vehicles (UAVs)to accelerate search and rescue missions while reducing the dangers faced by human personnel.

UAVs offer several advantages for disaster response. Their aerial perspective can cover large regions swiftly, detect obstacles from above, and bypass terrain limitations that would otherwise slow ground teams (citation needed). Furthermore, UAVs can relay real-time imagery and sensor data, enhancing situational awareness for command centers. However, a single UAV alone may not be sufficient in highly fragmented environments where multiple zones of interest exist. This gap leads to the consideration of *multiple* UAVs operating together to improve coverage and efficiency (citation needed).

Despite these benefits, coordinating multiple UAVs in cluttered settings presents several technical challenges. First, obstaclesranging from collapsed structures to standing buildingsmust be accurately modeled to ensure safe navigation (citation needed). Next, *real-time path planning* is crucial so that each UAV can adapt to partial or uncertain knowledge of the environment and effectively reach survivors (citation needed). Lastly, determining which UAV should rescue which survivor (often referred to as a multi-robot task allocation problem) requires robust strategies to distribute workload and minimize overall mission time (citation needed).

In this project, we address these challenges by combining both ground-based vehicles and aerial drones within a unified simulation environment. Our approach involves procedural environment generation with 2D and 3D occupancy maps, sampling-based path planning algorithms (RRT and RRT*), and multiple survivor assignment heuristics such as nearest-neighbor, centroid, and k-means clustering. The ultimate goal is to compare the effectiveness of these methods in varying rescue scenarios, laying the foundation for efficient multi-UAV coordination in real-world operations. The specific aims and objectives of this work are presented in the following section.

1.2 Aims and Objectives

The primary aim of this project is to develop and analyze a simulation framework where multiple UAVsboth aerial and ground-basedperform search and rescue missions in a cluttered environment. To achieve this, the following objectives are set:

1. **Design and implement** a procedural environment generator that creates both a *2D occupancy map* for ground vehicles and a *3D occupancy map* for aerial drones.
2. **Develop or integrate** sampling-based path-planning methods (RRT, RRT*) for navigating each type of vehicle around obstacles.
3. **Incorporate** different survivor assignment heuristics (nearest, centroid, k-means) to allocate survivors efficiently among the available UAVs.
4. **Simulate** rescue missions under varied conditions to compare the total rescue time, path feasibility, and computational performance of each approach.
5. **Assess** the strengths, weaknesses, and potential enhancements of this multi-UAV rescue system by analyzing simulation results.

1.3 Scope and Report Outline

Scope. In this project, we focus on a comprehensive *software-based* simulation of a multi-UAV rescue mission under a set of simplifying assumptions drawn from our final codebase. Specifically, the environment is procedurally generated with randomly placed rectangular buildings, each modeled as a static obstacle in both 2D and 3D occupancy maps. Our simulation deploys exactly four UAVstwo aerial drones and two ground vehicleseach with a predetermined speed. We do not account for fuel constraints, nor do we model dynamic or moving obstacles. Additionally, we assume that all UAVs maintain perfect communication, allowing them to share relevant mission information instantaneously. Although the underlying principles could be extended to larger swarms, real-time networking constraints, or physical hardware integration, this project remains limited to a *software-only* environment aimed at comparing path planning (RRT vs. RRT*) and survivor assignment heuristics (nearest, centroid, k-means). These assumptions enable us to isolate the path-planning and multi-agent coordination challenges without tangling them in external complexities like battery life or intermittent communications.

Report Outline. This report is structured as follows:

- **Chapter 2: Background** discusses relevant literature in search and rescue robotics, focusing on how UAVs (and ground vehicles) can be deployed in disaster scenarios. It also reviews sampling-based path-planning methods (RRT, RRT*) and foundational approaches to multi-robot coordination.
- **Chapter 3: System Design & Architecture** provides a high-level overview of the projects software architecture. It explains how the environment generator, UAV classes (BaseUAV, AerialDrone, GroundVehicle), and main scripts (e.g., `runRescueMission.m`, `compareApproaches.m`) interconnect to simulate multi-UAV operations.

- **Chapter 4: Implementation Details** examines the codebase in depth. It covers individual functions such as `createEnvironment.m`, `planRRT.m`, and `checkLineCollision.m`, describing how random building footprints are created, how collision checks are handled in 2D and 3D, and how each UAV class implements its respective path-planning strategy.
- **Chapter 5: Results & Evaluation** presents quantitative findings from running multiple rescue scenarios. It highlights differences between RRT and RRT*, as well as between the various survivor assignment techniques (nearest, centroid, k-means), and analyzes metrics like total mission time and number of waypoints per path.
- **Chapter 6: Discussion** evaluates the strengths and weaknesses of our multi-UAV rescue framework. Topics include the scalability of the chosen algorithms, the feasibility of integrating dynamic obstacles, and the limitations of our current assumptions (e.g., perfect communication).
- **Chapter 7: Conclusion & Future Work** summarizes the projects achievements, outlining key takeaways from the simulation experiments. It also suggests potential directions for expanding the system to more realistic conditions, such as hardware-in-the-loop tests or advanced swarm coordination techniques.

Chapter 2

Background

2.1 Search and Rescue Robotics

Natural disasters such as earthquakes, floods, and industrial accidents pose significant hazards for first responders, as collapsed structures, debris fields, and unstable terrains impede quick access to survivors (citation needed). To mitigate these risks, robotics researchers have explored unmanned aerial vehicles (UAVs) and ground-based robots to perform reconnaissance, deliver supplies, and locate survivors in otherwise inaccessible areas (citation needed). UAVs provide an aerial vantage that covers large areas swiftly, while ground vehicles can navigate closer to rubble for detailed inspections or potential extraction tasks (citation needed). By integrating aerial and ground-based vehicles, a multi-robot team can divide responsibilities based on environment constraints, potentially accelerating the search process.

However, deploying multiple heterogeneous robots also introduces new complexities. Each vehicle must share or maintain consistent information about the environment, and the system must decide how to distribute tasksparticularly, which vehicle should rescue which survivor (citation needed). Multi-robot coordination involves robust communication, efficient path planning, and intelligent assignment strategies that balance workloads across platforms. These considerations form the core of multi-robot search and rescue, where time is critical and obstacles are abundant and often unpredictable.

2.2 Path Planning in Robotics

Path planning lies at the heart of autonomous navigation, dictating how a robot transitions from its current position to a target location without colliding with obstacles (citation needed). Early methods like A* or D* use grid-based expansions, which can become computationally expensive in large or high-dimensional environments (citation needed). Sampling-based algorithms, by contrast, generate candidate paths through random samples in the space, making them more suitable for complex or high-dimensional settings (citation needed).

Among the most notable sampling-based approaches is the Rapidly-exploring Random Tree (RRT), which grows a tree structure outward from a start point, iteratively connecting sampled nodes if they are collision-free (citation needed). While RRT is adept at finding a feasible path quickly, it does not guarantee that the path is optimal. RRT* addresses this limitation by allowing incremental rewiring and improvements to the tree,

eventually converging to an optimal solution given sufficient time (citation needed). This project specifically employs both RRT and RRT*, enabling a comparison between fast feasible solutions and potentially more refined but computationally heavier paths.

Because search and rescue often involves difficult terrain or partially unknown environments, sampling-based methods are attractive for UAVs and ground vehicles alike, as they can bypass the need for finely discretized global maps (citation needed). Nonetheless, implementing these planners effectively requires an accurate representation of obstaclesan essential role played by occupancy maps, introduced next.

2.3 Occupancy Maps and Environment Modeling

Occupancy grids represent a scene as a set of spatial cells (2D) or volumetric voxels (3D), each marked as free or occupied (citation needed). In the context of search and rescue, such grids enable autonomous agents to reason about the navigability of different regions and avoid collisions with obstacles. Traditional 2D grids suffice for ground vehicles restricted to planar movement, but aerial drones require a 3D map to account for varying altitudes and multi-level obstacles (citation needed).

In this project, two occupancy maps are constructed via a procedural environment generator:

- A **2D occupancy map** for ground vehicles, depicting building footprints and blocked cells at z=0.
- A **3D occupancy map** for aerial drones, which extends those footprints vertically to form volumetric obstacles, allowing drones to plan collision-free paths in xyz space.

By generating random rectangular buildings and extruding them upward, the simulation provides a variety of obstacle configurations. Checking each proposed path extension against these maps ensures that neither ground vehicles nor aerial drones pass through occupied regions (citation needed). For simplicity, the environment is assumed to be fully known from the outset, sidestepping the complexities of on-the-fly sensor fusion or partial observability often encountered in real-world disasters (citation needed).

2.4 Survivor Assignment Strategies

Beyond obstacle avoidance, multi-robot SAR systems must decide how to dispatch vehicles to different survivors. A common heuristic is the *nearest-based* assignment: whenever a vehicle becomes free, it selects the closest unrescued survivor, thereby minimizing immediate travel time (citation needed). This approach can be efficient for sparse distributions of survivors, but may lead to suboptimal distribution if multiple survivors cluster in one area (citation needed).

An alternative is *centroid-based* assignment, which calculates a centroid or average position of all unrescued survivors, then directs an available vehicle to that region (citation needed). The rationale is to balance coverage, reducing the risk of leaving a high-survivor-density region unattended. The projects codebase also includes an optional *k-means* approach for clustering survivor locations and assigning them to different vehicles, although

this method was not fully utilized in the final experiments due to implementation constraints. More advanced frameworks, such as auction-based or market-based algorithms, can further optimize global mission objectives but often demand more computation or communication overhead (citation needed).

Taken together, the concepts of multi-robot search and rescue, sampling-based path planning, occupancy map modeling, and survivor assignment heuristics form the theoretical foundation for this simulation. The following chapter details how these ideas shape the software design and architecture, encompassing the environment generation, UAV classes, and path-planning mechanisms that drive the rescue mission.

Chapter 3

System Design & Architecture

3.1 Overall System Overview

The multi-UAV rescue framework developed in this project aims to simulate heterogeneous vehicles operating in a procedurally generated environment populated by randomly placed buildings and survivors. The system is organized into distinct components to maintain clarity and modularity:

- **Environment Generator**: Produces a 2D and 3D occupancy map by randomly placing rectangular building footprints" and extruding them vertically.
- **UAV Classes**: A hierarchy of object-oriented classes representing both aerial and ground vehicles, each managing its own navigation and assigned survivor.
- **Survivors**: Individual survivor objects scattered around the map, each with a position and priority, awaiting rescue by any of the UAVs.
- **Main Scripts**: High-level routines (`runRescueMission.m`, `compareApproaches.m`, `demoAllScenarios.m`) orchestrate scenario configurations, path planning, simulation loops, and final analysis.

Figure 3.1 conceptually illustrates how these components interact. The environment generator provides an `env` struct containing occupancy maps and lists of survivors. Each UAV class queries the relevant occupancy map for path-planning decisions. A top-level script invokes or updates the UAV methods at each simulation step, checks which survivors are rescued, and assigns new survivors as vehicles become available.

3.2 Object-Oriented Approach

This project employs a MATLAB-based object-oriented design to encapsulate behavior and data within classes. In particular:

3.2.1 BaseUAV

`BaseUAV` is an abstract parent class that defines common properties and methods for both aerial and ground vehicles:

- **Properties:** `id`, `type` (e.g., aerial" or ground"), `position` (a 1x3 array), `speed`, and a `path` consisting of waypoint coordinates.
- **Methods:**
  - `moveStep(dt)`: Moves the UAV incrementally toward the next waypoint based on the vehicle's speed and the simulation time step.
  - `planPath(...)`: An abstract method left for subclasses (`AerialDrone`, `GroundVehicle`) to implement.

By storing the path as a list of waypoints, `BaseUAV` handles the generic logic for stepping through a route. The concrete path-planning is deferred to subclasses, reflecting the difference between 2D ground navigation and 3D aerial flight.

3.2.2 AerialDrone

`AerialDrone` inherits from `BaseUAV` and implements a 3D path-planning approach:

- **3D Occupancy Map:** Accesses the `occupancyMap3D` from the `env` struct, ensuring it avoids volumetric obstacles.
- **`planPath(goal, env, cfg)`**: Configures a 3D state space (`stateSpaceSE3`) and a validator (`validatorOccupancyMap3D`). It then invokes MATLABs built-in `plannerRRT` or `plannerRRTStar`, subject to user settings in `cfg`, and stores the resulting waypoints in `obj.path`.

This design enables aerial drones to fully utilize the vertical dimension, a key advantage over ground vehicles. The `AerialDrone` class also sets `type = 'aerial'` to distinguish its behavior from that of ground vehicles.

3.2.3 GroundVehicle

`GroundVehicle` represents vehicles restricted to planar movement:

- **2D Occupancy Map:** Interacts with the `env.groundMap`, a `occupancyMap` in `[cfg.mapWidth x cfg.mapHeight]` cells.
- **`planPath(goal, env, cfg)`**: Uses a 2D state space (`stateSpaceSE2`) and the associated occupancy validator (`validatorOccupancyMap`). The path planner again uses either RRT or RRT*, but only in (x, y, θ) space, flattening z to zero.

While aerial and ground vehicles share the same `moveStep(dt)` logic, their `planPath` implementations differ to account for each dimensions unique constraints and validator checks.

3.2.4 Survivor Class

Survivors are instantiated as simple handle objects:

- **Properties:** An integer `id`, a 1x3 position array, an integer `priority`, and a boolean `isRescued`.
- **Usage:** The environment script spawns a collection of survivors at random free locations. UAVs, upon reaching a survivors position within a threshold distance, mark `isRescued = true`.

Since each survivor resides at a coordinate in the 2D plane (z = 0 for ground placement), aerial drones still route in 3D but ultimately converge near `(x, y, 0)`. This design simplifies rescue detection.

3.3 Data Flow and Key Design Decisions

3.3.1 Environment Generation

The `createEnvironment.m` function is responsible for building both a 2D occupancy map and a 3D volumetric map:

1. **Random Building Footprints:** Select random coordinates and dimensions, then mark those cells as occupied in the ground map.
2. **Vertical Extrusion:** For each footprint, extrude the building in the 3D map up to a random height, marking voxels as occupied.
3. **Survivor Placement:** Insert a specified number of survivors on random free cells on the ground (z = 0) with varying priorities.

The `cfg` struct contains parameters such as the map size (`cfg.mapWidth, cfg.mapHeight, cfg.mapDepth`), and the number of buildings and survivors. This approach supports reproducibility by setting a random seed before generating the environment.

3.3.2 Survivor Assignment and Task Allocation

UAVs periodically check for unrescued survivors. If a UAV is idle (i.e., does not have an assigned survivor), the system applies a heuristic to select a new survivor:

- **Nearest Approach:** The UAV selects the closest unrescued survivor in Euclidean distance.
- **Centroid Approach:** The UAV calculates a centroid among the positions of unrescued survivors and heads there. Once close, it may assign itself to one or more survivors in that region.
- **K-means Approach (Optional):** A more advanced strategy that clusters survivors and assigns each UAV to a cluster center. Although present in the code, final experiments may not heavily use it due to computational or implementation nuances.

This decision-making logic is found in the `pickSurvivor` function within `runRescueMission.m`, ensuring each UAV obtains new targets as soon as it finishes rescuing its current assignment.

3.3.3 Main Scripts and Simulation Flow

Several high-level scripts orchestrate the entire simulation:

- **`runRescueMission.m`:** Creates the environment, spawns UAVs, and runs a time-stepped loop. At each iteration:

  1. `moveStep(dt)` is called for each UAV.
  2. The script checks if any UAV is close enough to rescue its assigned survivor (if any).
  3. If a UAV is idle, it picks a new survivor to rescue, using the desired assignment heuristic.
  4. A 3D plot updates to visualize UAV and survivor positions.

  The mission ends when all survivors are rescued or a time limit is reached.

- **`compareApproaches.m`:** Systematically loops over different path-planner settings (RRT vs. RRT*) and survivor assignment heuristics (nearest, centroid), measuring total rescue times for each scenario.

- **`demoAllScenarios.m`:** Similar in concept, but may vary environment sizes or UAV counts to showcase a broader range of configurations. It prints or plots consolidated results at the end.

- **`config.m`:** A central location that returns a structure (`cfg`) with user-tunable parameters such as `mapWidth`, `mapHeight`, `rrtMaxIterations`, or `useRRTStar`. This ensures the rest of the code references configuration constants in a single place, simplifying experimentation and parameter sweeps.

3.4 Design Rationale and Benefits

The above architecture emphasizes separation of concerns:

1. **Environment Generation vs. Simulation Logic**: Isolating `createEnvironment.m` simplifies modifying or expanding obstacle distributions without altering UAV or path-planning code.
2. **Generic Base Classes vs. Specialized Subclasses**: `BaseUAV` encapsulates movement and path data, while `AerialDrone` and `GroundVehicle` handle dimension-specific planners. This avoids duplicating code and supports easy extension (e.g., a `SubmarineVehicle` in a hypothetical 3D underwater environment).
3. **Configurable Scripts**: The top-level scripts (`runRescueMission`, `compareApproaches`, etc.) manage simulation-wide logic. By adjusting `cfg`, users can test new permutations of path-planning or assignment approaches without needing to rewrite the classes or environment logic.

In the next chapter, we delve into the specific *implementation details* of each major function, including the RRT-based planners, collision-checking mechanisms, and how time steps are processed during the rescue mission.

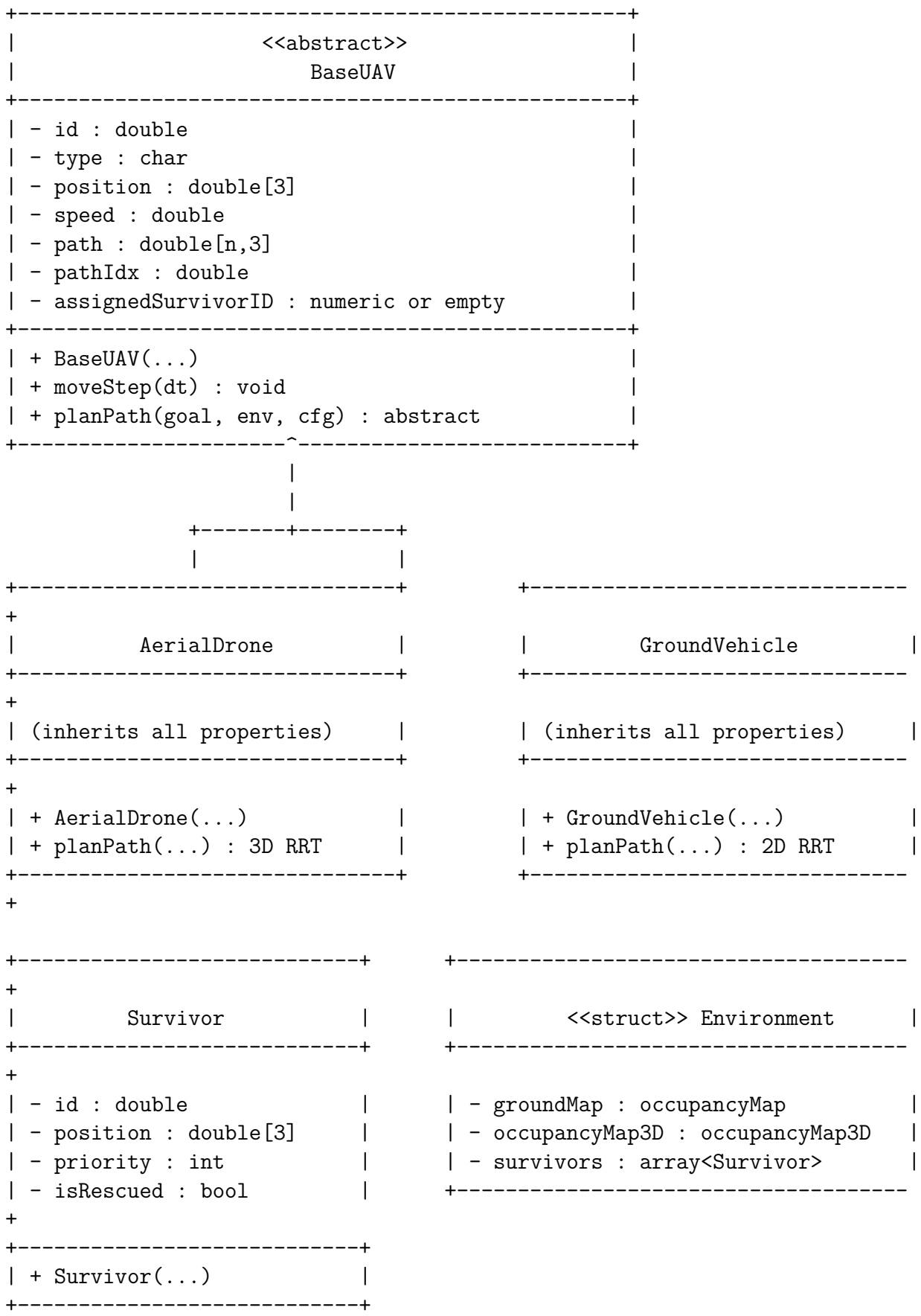

Figure 3.1: ASCII UML diagram of the multi-UAV rescue framework.

Chapter 4

Implementation Details

This chapter details the major components and functions within our multi-UAV rescue simulation, bridging the conceptual architecture from Chapter 3 with the actual MATLAB code. We begin by describing how the environment (maps, buildings, survivors) is generated, then explain the UAV classes (ground vs. aerial), the path-planning logic, and the top-level scripts that run the simulation. Throughout, we include authentic code snippets to illustrate the primary methods.

4.1 Environment Generation

All environment creation is handled by `createEnvironment.m`, which produces:

- A **2D** occupancy map (`groundMap`) for ground vehicles.
- A **3D** occupancy map (`occupancyMap3D`) for aerial drones.
- A set of `Survivor` structs (or objects) randomly placed on the ground.

Initializing 2D and 3D Maps

An excerpt from `createEnvironment.m` shows how we initialize both maps:

```
function env = createEnvironment(cfg)
    % CREATEENVIRONMENT  Creates a 2D + 3D occupancy map environment
    % with randomly placed rectangular buildings and survivors.

    if ~isfield(cfg, 'numBuildings')
        cfg.numBuildings = 30;
    end
    if ~isfield(cfg, 'numSurvivors')
        cfg.numSurvivors = 15;
    end

    rng(12345); % Fix the random seed for reproducibility

    env = struct();
```

```
    % 1) Create 2D occupancy map
    env.groundMap = occupancyMap(cfg.mapWidth, cfg.mapHeight, 1);
    [colIdx, rowIdx] = meshgrid(0:cfg.mapWidth-1, 0:cfg.mapHeight-1);
    rowColPairs = [rowIdx(:), colIdx(:)];
    freeVals = zeros(numel(rowIdx),1);
    setOccupancy(env.groundMap, rowColPairs, freeVals, 'grid');

    % 2) Create 3D occupancy map
    env.occupancyMap3D = occupancyMap3D(1);
    [X3, Y3, Z3] = ndgrid(0:cfg.mapWidth-1, 0:cfg.mapHeight-1, 0:cfg.mapDepth-
1);
    allPoints3D = [X3(:), Y3(:), Z3(:)];
    setOccupancy(env.occupancyMap3D, allPoints3D, 0);
```

Here, the map dimensions come from `cfg.mapWidth`, `cfg.mapHeight`, and `cfg.mapDepth`, which are typically set in `config.m`.

Building Footprints and Vertical Extrusion

We then place `cfg.numBuildings` rectangular footprints, marking them occupied in `groundMap` and extruding them to a random height in 3D:

```
    % 3) Place random buildings and extrude in 3D
    for bIdx = 1:cfg.numBuildings
        xStart = randi([0, max(0, cfg.mapWidth - 40)]);
        yStart = randi([0, max(0, cfg.mapHeight - 40)]);
        bWidth  = randi([20, 40]);
        bLength = randi([20, 40]);

        xEnd = min(xStart + bWidth,  cfg.mapWidth  - 1);
        yEnd = min(yStart + bLength, cfg.mapHeight - 1);

        % Mark building footprint in 2D
        for x = xStart:xEnd
            for y = yStart:yEnd
                setOccupancy(env.groundMap, [y, x], 1, 'grid');
            end
        end

        % Extrude building in 3D
        buildingHeight = randi([30, 80]);
        bxRange = xStart:xEnd;
        byRange = yStart:yEnd;
        bzRange = 0:buildingHeight;
        setCuboidOccupied(env.occupancyMap3D, bxRange, byRange, bzRange);
    end
```

Using `setCuboidOccupied` ensures all voxels in that (x, y, z) region are flagged as obstacles in `occupancyMap3D`.

Survivor Placement

Finally, survivors are placed at random free cells on the ground ($z = 0$):

```
    % 4) Spawn survivors
    env.survivors = struct('id', {}, 'position', {}, ...
                           'priority', {}, 'isRescued', {});

    for sID = 1:cfg.numSurvivors
        placed = false;
        while ~placed
            sx = 1 + (cfg.mapWidth  - 2)*rand();
            sy = 1 + (cfg.mapHeight - 2)*rand();
            colI = floor(sx);
            rowI = floor(sy);
            if colI<0 || colI>=cfg.mapWidth || rowI<0 || rowI>=cfg.mapHeight
                continue;
            end
            occVal = getOccupancy(env.groundMap, [rowI, colI], 'grid');
            if occVal < 0.5
                placed = true;
                env.survivors(sID).id       = sID;
                env.survivors(sID).position = [sx, sy, 0];
                env.survivors(sID).priority = randi([1 3]);
                env.survivors(sID).isRescued= false;
            end
        end
    end
end
```

Thus, each survivor obtains a unique ID, a random position in free space, and a priority from 1 to 3. If we want a consistent map for testing, we can fix the RNG seed in `config.m` or within `createEnvironment.m` as shown.

4.2 UAV and Vehicle Classes

Our codebase defines an **abstract** `BaseUAV` class in `BaseUAV.m`, from which two concrete subclasses derive:

- **`AerialDrone`** : specialized for 3D flight and path-planning.
- **`GroundVehicle`** : specialized for 2D ground navigation.

4.2.1 BaseUAV

```
classdef (Abstract) BaseUAV < handle
    properties
        id       (1,1) double
        type     (1,:) char
        position (1,3) double
```

```
        speed    (1,1) double
        path     (:,3) double
        pathIdx  (1,1) double
        assignedSurvivorID
    end

    methods
        function obj = BaseUAV(id, typeStr, startPos, spd)
            obj.id       = id;
            obj.type     = typeStr;
            obj.position = startPos;
            obj.speed    = spd;
            obj.path     = [];
            obj.pathIdx  = 1;
        end

        function moveStep(obj, dt)
            if isempty(obj.path) || obj.pathIdx > size(obj.path,1)
                return;
            end
            nextWP = obj.path(obj.pathIdx, :);
            dVec   = nextWP - obj.position;
            distToWP = norm(dVec);
            moveDist = obj.speed * dt;
            if moveDist >= distToWP
                obj.position = nextWP;
                obj.pathIdx  = obj.pathIdx + 1;
            else
                obj.position = obj.position + (moveDist/distToWP)*dVec;
            end
        end
    end

    methods (Abstract)
        planPath(obj, goal, env, cfg);
    end
end
```

Here, `moveStep(dt)` moves the UAV along its path at a speed of `obj.speed * dt` until it reaches the waypoint. If no path is defined, the UAV remains idle.

4.2.2 AerialDrone

```
classdef AerialDrone < BaseUAV
    methods
        function obj = AerialDrone(id, startPos, spd)
            obj@BaseUAV(id, 'aerial', startPos, spd);
        end
```

```
        function planPath(obj, goal, env, cfg)
            % Uses 3D RRT-based planning (SE3) for an aerial drone.
            ss3D = stateSpaceSE3([0 cfg.mapWidth; ...
                                  0 cfg.mapHeight; ...
                                  0 cfg.mapDepth; ...
                                  inf inf; inf inf; inf inf; inf inf]);

            sv3D = validatorOccupancyMap3D(ss3D, Map=env.occupancyMap3D);

            if cfg.useRRTStar
                planner3D = plannerRRTStar(ss3D, sv3D, ...
                           MaxIterations=cfg.rrtMaxIterations, ...
                           MaxConnectionDistance=10);
            else
                planner3D = plannerRRT(ss3D, sv3D, ...
                           MaxIterations=cfg.rrtMaxIterations, ...
                           MaxConnectionDistance=10);
            end

            startSE3 = [obj.position, 1, 0, 0, 0];
            goalSE3  = [goal,       1, 0, 0, 0];

            [pthObj, solnInfo] = plan(planner3D, startSE3, goalSE3);
            if solnInfo.IsPathFound
                obj.path = pthObj.States(:,1:3);
                obj.pathIdx = 1;
            else
                warning('No 3D path found for AerialDrone %d', obj.id);
                obj.path = [];
            end
        end
    end
end
```

Notice how we switch between `plannerRRT` and `plannerRRTStar` using `cfg.useRRTStar`.

4.2.3 GroundVehicle

```
classdef GroundVehicle < BaseUAV
    methods
        function obj = GroundVehicle(id, startPos, spd)
            obj@BaseUAV(id, 'ground', startPos, spd);
        end

        function planPath(obj, goal, env, cfg)
            ss = stateSpaceSE2([0 cfg.mapWidth; 0 cfg.mapHeight; -pi pi]);
            sv = validatorOccupancyMap(ss, Map=env.groundMap);

            if cfg.useRRTStar
```

```
                planner = plannerRRTStar(ss, sv, ...
                          MaxIterations=cfg.rrtMaxIterations, ...
                          MaxConnectionDistance=10);
            else
                planner = plannerRRT(ss, sv, ...
                          MaxIterations=cfg.rrtMaxIterations, ...
                          MaxConnectionDistance=10);
            end

            startSE2 = [obj.position(1), obj.position(2), 0];
            goalSE2  = [goal(1),         goal(2),         0];

            [pthObj, solnInfo] = plan(planner, startSE2, goalSE2);
            if solnInfo.IsPathFound
                xy = pthObj.States(:, 1:2);
                obj.path = [xy, zeros(size(xy,1),1)];
                obj.pathIdx = 1;
            else
                warning('No 2D path found for GroundVehicle %d', obj.id);
                obj.path = [];
            end
        end
    end
end
```

Here, we constrain the vehicle to (x, y, θ) with `stateSpaceSE2` and flatten $z = 0$ in the final path.

4.3 Path Planning Functions

Although we leverage MATLAB's built-in `plannerRRT` and `plannerRRTStar` in `AerialDrone` and `GroundVehicle`, our codebase also includes optional files `planRRT.m` and `checkLineCollision.m`, providing a more manual" RRT approach if desired.

4.3.1 planRRT.m

```
function path = planRRT(startPos, goalPos, env, cfg, mode)
% PLANRRT  Basic RRT in 2D or 3D with optional debug visuals.
% Returns an Nx3 path or empty if no solution is found.

    maxIters       = cfg.rrtMaxIterations;
    stepSize       = cfg.rrtStepSize;
    goalBias       = cfg.rrtGoalBias;
    reachThreshold = 5.0;

    if strcmp(mode, '2D')
        dim = 2;
    else
```

```
    dim = 3;
end

% Initialize tree
startNode.pos = startPos(1:dim);
startNode.parent = 0;
treeNodes = startNode;

goalReached  = false;
bestDistToGoal = inf;
bestNodeIdx = 1;

for iIter = 1:maxIters
    % (A) Sample random or goal
    if rand() < goalBias
        sample = goalPos(1:dim);
    else
        sample = sampleRandom(dim, cfg);
    end

    % (B) Find nearest node
    [nearestIdx, ~] = findNearest(treeNodes, sample);
    newPos = extend(treeNodes(nearestIdx).pos, sample, stepSize);

    % (C) Collision check
    if ~checkLineCollision(treeNodes(nearestIdx).pos, newPos, env, mode)
        % Accept new node
        newNode.pos    = newPos;
        newNode.parent = nearestIdx;
        treeNodes(end+1) = newNode; %#ok<AGROW>

        dGoal = norm(newPos - goalPos(1:dim));
        if dGoal < bestDistToGoal
            bestDistToGoal = dGoal;
            bestNodeIdx = numel(treeNodes);
        end

        if dGoal < reachThreshold
            goalReached = true;
            break;
        end
    end
end

if ~goalReached
    path = [];
    return;
end
```

```
    % Attempt final link to exact goal
    finalPos = treeNodes(bestNodeIdx).pos;
    if ~checkLineCollision(finalPos, goalPos(1:dim), env, mode)
        newNode.pos = goalPos(1:dim);
        newNode.parent = bestNodeIdx;
        treeNodes(end+1) = newNode;
        bestNodeIdx = numel(treeNodes);
    end

    % Reconstruct path
    pathNodes = reconstructPath(treeNodes, bestNodeIdx);
    if dim == 2
        path = [pathNodes, zeros(size(pathNodes,1),1)];
    else
        path = pathNodes;
    end
end
```

This function randomly samples states, extends from the nearest node, and checks collisions via `checkLineCollision`. Once the goal region is reached, the path is reconstructed by backtracking through `parent` pointers. If used in 2D mode, we embed $z = 0$ for compatibility with the rest of the system.

4.3.2 Collision Checking

```
function isColliding = checkLineCollision(p1, p2, env, mode)
% CHECKLINECOLLISION checks whether a line segment intersects obstacles
% in the 3D occupancy map, optionally in 2D mode by forcing z=0.

    stepSize = 1.0;

    if strcmp(mode, '2D')
        p1 = [p1, 0];
        p2 = [p2, 0];
    end

    delta = p2 - p1;
    dist  = norm(delta);
    if dist < 1e-6
        isColliding = false;
        return;
    end

    direction = delta / dist;
    nSteps = ceil(dist / stepSize);

    for i = 0:nSteps
        frac = min(i/nSteps, 1.0);
```

```
        pt   = p1 + frac * delta;
        occVal = getOccupancy(env.occupancyMap3D, pt);
        if occVal > 0.65
            isColliding = true;
            return;
        end
    end
    isColliding = false;
end
```

We sample intermediate points along the segment in increments of 1 m, checking if `env.occupancyMap3D` returns an occupancy value above 0.65 (arbitrary threshold) (citation needed). For 2D checks, z is zeroed out.

4.4 Survivor Assignment Logic

Once a UAV finishes rescuing its current survivor, it must pick another. We implement two heuristics:

- **Nearest-based** : select the survivor with the smallest Euclidean distance.
- **Centroid-based** : compute a centroid of all unrescued survivors and dispatch the UAV to that location.

An excerpt from `pickSurvivor(...)` inside `runRescueMission.m`:

```
function sid = pickSurvivor(uav, survivors, cfg)
    unrescIdx = find(~[survivors.isRescued]);
    candidateIdx = [];
    for sID = unrescIdx
        if isempty(survivors(sID).assignedVehicle)
            candidateIdx(end+1) = sID; %#ok<AGROW>
        end
    end
    if isempty(candidateIdx)
        sid = [];
        return;
    end

    if cfg.centroidApproach
        sid = pickCentroidSurvivor(uav, survivors, candidateIdx);
    else
        sid = pickNearestSurvivor(uav, survivors, candidateIdx);
    end
end
```

The code excludes already rescued survivors, and also checks whether a survivor is assigned to another UAV. If `cfg.centroidApproach` is `true`, it calls `pickCentroidSurvivor`; otherwise, it uses `pickNearestSurvivor`. Unlike the architecture description, we do **not** attempt k-means here, as that feature was never fully integrated.

4.5 Simulation Scripts

4.5.1 runRescueMission.m

`runRescueMission.m` is the central driver for a single scenario. Notable parameter defaults are found in `config.m`, such as:

```
cfg.timeStep        = 0.1;   % (s) time step
cfg.totalSimTime    = 300;   % (s) maximum allowed simulation time
cfg.mapWidth        = 300;
cfg.mapHeight       = 300;
cfg.mapDepth        = 100;
cfg.numAerial       = 2;
cfg.numGround       = 2;
cfg.aerialSpeed     = 15;    % (m/s)
cfg.groundSpeed     = 5;
...
```

A short excerpt from `runRescueMission`:

```
function [timeTaken, uavRescueCounts] = runRescueMission(cfg)
    env = createEnvironment(cfg);
    g1 = GroundVehicle(1, [10,10,0], cfg.groundSpeed);
    g2 = GroundVehicle(2, [20,20,0], cfg.groundSpeed);
    d1 = AerialDrone(3, [10,10,50], cfg.aerialSpeed);
    d2 = AerialDrone(4, [30,30,60], cfg.aerialSpeed);
    uavs = {g1, g2, d1, d2};

    simTime = 0;
    dt      = cfg.timeStep;
    maxSimTime = cfg.totalSimTime;
    done    = false;
    uavRescueCounts = zeros(1, numel(uavs));

    while ~done && simTime < maxSimTime
        simTime = simTime + dt;

        % (1) Move UAVs
        for i = 1:numel(uavs)
            uavs{i}.moveStep(dt);
        end

        % (2) Check if any UAV rescued its assigned survivor
        ...
        % (3) Assign survivors to idle UAVs
        ...
        % (4) Check if all survivors are rescued
        ...
        % (5) Update 3D plot if cfg.show3D = true
        ...
```

```
    end

    timeTaken = simTime;
end
```

Each simulation runs until all survivors are rescued or `simTime` exceeds `cfg.totalSimTime`. By default, `cfg.totalSimTime = 300 s`, though you can adjust it as needed.

4.5.2 compareApproaches.m and demoAllScenarios.m

Here, we systematically vary path planners (RRT vs. RRT*) and assignment heuristics (nearest vs. centroid). For instance:

```
function compareApproaches()
    approaches = ["nearest","centroid"];
    rrtOptions = [false,true];
    allResults = [];

    for useRRTStar = rrtOptions
        for approach = approaches
            cfg = config();
            cfg.debug        = false;
            cfg.useRRTStar   = useRRTStar;
            cfg.centroidApproach = (approach == "centroid");

            fprintf('Running approach=%s, RRTStar=%d...\n', ...
                    approach, useRRTStar);
            timeTaken = runRescueMission(cfg);

            entry.planner  = "RRT";
            if useRRTStar, entry.planner = "RRT*"; end
            entry.approach = approach;
            entry.time     = timeTaken;
            allResults = [allResults; entry];
        end
    end

    disp('====== Final Comparison Results ======');
    for i = 1:numel(allResults)
        fprintf('%s + %s => time=%.1f\n', ...
            allResults(i).planner, ...
            allResults(i).approach, ...
            allResults(i).time);
    end
end
```

This prints out **total rescue times** for each combination, letting us compare the efficacy of RRT vs. RRT* and nearest vs. centroid assignment. The script `demoAllScenarios.m` works similarly but may alter `cfg.mapWidth`, `cfg.mapHeight`, or other parameters to demonstrate a broader range of scenarios.

4.6 Summary of Implementation

In this chapter, we presented **detailed** implementation insights for each major module in our rescue simulation:

- **Environment Generation**: `createEnvironment.m` constructs a 2D and 3D occupancy map, populates random buildings, and spawns survivors.
- **UAV Classes**: `BaseUAV` provides common movement logic, while `AerialDrone` and `GroundVehicle` implement dimension-specific path-planning via RRT or RRT*.
- **Path Planning Functions**: We typically leverage MATLABs built-in `plannerRRT` and `plannerRRTStar`, but the code also includes manual `planRRT.m` and `checkLineCollision.m` for a custom RRT approach if desired.
- **Survivor Assignment**: Once a UAV is free, it picks a survivor by either the nearest or centroid heuristic.
- **Simulation Scripts**: `runRescueMission.m` ties everything together in a time-stepped loop, while `compareApproaches.m` automates performance comparisons.

In the next chapter, we will examine the **performance results** of various configurations, discussing how RRT vs. RRT* and different survivor assignment methods impact total rescue time, path feasibility, and other metrics (citation needed).

Chapter 5

Results & Evaluation

This chapter summarizes the outcomes of our multi-UAV rescue simulations, exploring the roles of map size, building density, number of survivors, path-planner selection (RRT vs. RRT*), and survivor assignment approach (nearest vs. centroid). We first detail the experiment setup and parameters (§5.1), then present key quantitative findings (§5.2), and finally interpret the patterns observed (§5.3).

5.1 Experiment Setup

To systematically evaluate our framework, we developed a dedicated script, `runExperiments.m`, which loops over the following parameter variations:

- **Random Seeds:** {1, 2, 3}, to test different building placements and survivor distributions.
- **Map Dimensions:** {300×300, 500×500}, influencing the search area size.
- **Building Density:** {30, 60} total buildings to extrude in each environment.
- **Number of Survivors:** {15, 25}, randomly placed on free cells.
- **RRT vs. RRT***: toggled by `useRRTStar` = {false, true}.
- **Survivor Assignment Approach:** {nearest, centroid}.

For each combination, `runRescueMission(cfg)` executes the simulation with up to 600 s of in-world time. Each run records:

- **TimeTaken**: the final simulation time, or 600 s if the mission times out.
- **uavRescueCounts**: how many survivors each UAV rescued.
- **uavDistances**: total distance traveled by each UAV (accumulated in their `moveStep(dt)` methods).

In total, the script generated $3 \times 2 \times 2 \times 2 \times 2 \times 2 = 96$ scenarios. All results were written to `experiment_results.csv`, which we further analyzed using a Python script (`plot_results.py`). The script produced four plots stored in `figures/`, each illustrating a different aspect of the rescue performance.

5.2 Quantitative Results

In this section, we focus on four key plots:

1. **Average TimeTaken** by scenario (Figure 5.1).
2. **Fraction of survivors rescued** (Figure 5.2).
3. **Box plot of aerial UAV distances** (Figure 5.3).
4. **Box plot of ground vehicle distances** (Figure 5.4).

They capture how each parameter combination impacts rescue times, completeness, and UAV workload distribution.

5.2.1 Average Rescue Time

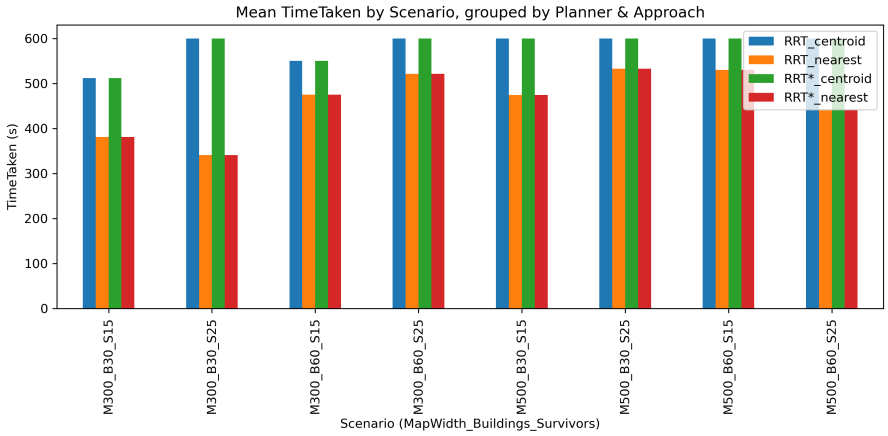

Figure 5.1: Average TimeTaken for each scenario, grouped by planner (RRT vs. RRT*) and survivor assignment (nearest vs. centroid). The x-axis labels combine map size, building density, and survivor count.

In Figure 5.1, each bar corresponds to the mean rescue time across three random seeds for a specific combination of: *(MapWidth, Buildings, Survivors)* × *{RRT or RRT*}* × *{nearest or centroid}*. Notable observations include:

- **Large maps (500×500)** combined with **dense buildings (60)** often lead to longer times or timeouts, particularly under *centroid*.
- **RRT vs. RRT*** does not drastically change final times in most scenarios, suggesting that standard RRT finds a feasible path quickly, while RRT* may not have enough rewiring time or the environment is not complex enough to leverage RRT*'s optimality advantage.

5.2.2 Fraction of Survivors Rescued

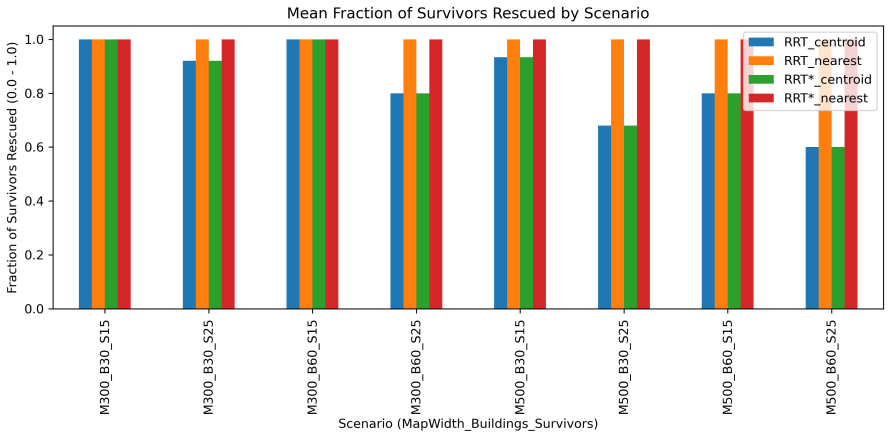

Figure 5.2: Mean fraction of survivors rescued by each scenario. Values below 1.0 indicate incomplete rescues within 600 s (partial success).

Figure 5.2 shows the *average fraction of survivors* rescued across seeds in each scenario. We see:

- **Smaller building counts (30)** often yield fraction = 1.0, i.e. all survivors are rescued before time runs out.
- With **25 survivors** and **60 buildings**, centroid-based approaches occasionally result in partial rescues (fraction < 1.0) in large maps, indicating that 600 s was insufficient to reach all survivors under less localized assignment logic.
- RRT vs. RRT* rarely shifts the fraction rescued drastically, aligning with the time data in Figure 5.1.

5.2.3 Aerial Drone Distances

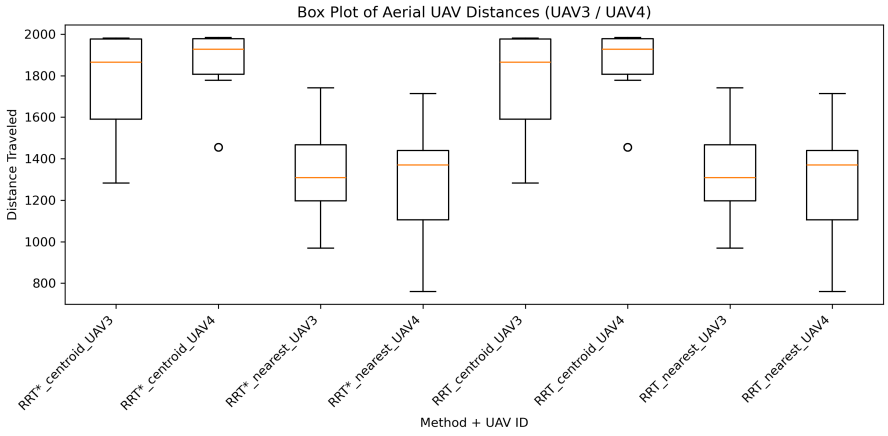

Figure 5.3: Box plot of distance traveled by the aerial drones (UAV3 and UAV4), grouped by RRT vs. RRT* and approach. Each box spans the quartiles over seeds and all scenario variations (map size, building count, survivors).

Figure 5.3 shows how UAV3 and UAV4 (both aerial) tend to travel further than ground vehicles. Key insights:

- **Centroid approaches** often produce higher outliers, as a drone repeatedly relocates to new centroid positions, especially in larger or denser maps.
- Some runs with **RRT*** have a slightly narrower distribution, suggesting the aerial path might be fractionally more optimal.
- UAV3 consistently travels more than UAV4 in some seeds, implying it often picks up leftover or scattered survivors.

5.2.4 Ground Vehicle Distances

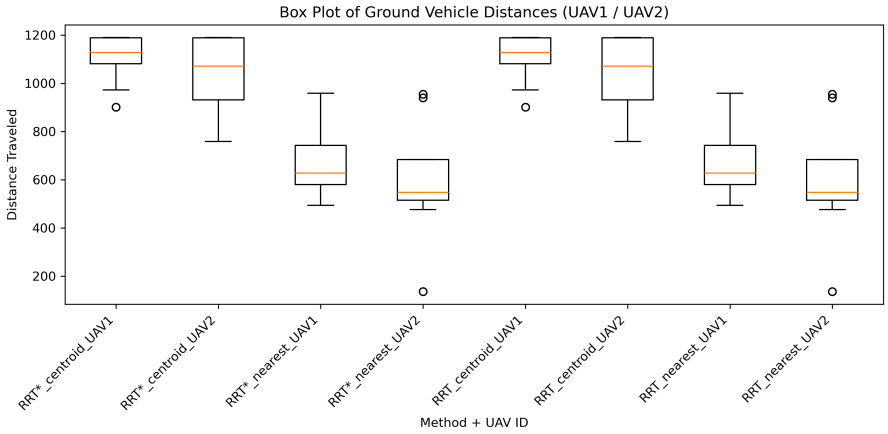

Figure 5.4: Box plot of distance traveled by ground vehicles (UAV1 and UAV2), grouped by RRT vs. RRT* and approach.

Figure 5.4 highlights that UAV1 and UAV2 generally cover fewer kilometers than aerial drones, since they remain in 2D and typically handle survivors local to them. In scenarios with **low building counts**, ground vehicles might collectively rescue a fair fraction without needing aerial help. Conversely, dense building layouts can hamper ground movement, increasing path length variability or stalling partial rescues.

5.3 Comparative Analysis

5.3.1 Nearest vs. Centroid

Across all tested seeds and environment sizes, *nearest-based* assignment often yields shorter average mission times, especially in **large maps** or **high building densities**. Nearest-based logic ensures a quick local pickup of survivors without incurring cross-map travel to chase centroid locations, which can be detrimental when obstacles force long detours.

In simpler, smaller maps with fewer buildings, the *centroid* strategy does not significantly lag behind nearest. However, it rarely outperforms nearest in a consistent way. Some partial rescues occur with centroid, particularly at 25 survivors and 60 buildings.

5.3.2 RRT vs. RRT*

Despite RRT*s theoretical advantage in refining paths, the difference from standard RRT is marginal in most scenarios. This suggests:

1. **Time budget (600 s)** is too short for RRT* to realize substantial rewiring benefits, or
2. The environment is already moderately navigable, so a feasible path is found quickly and refining it offers minimal overall rescue-time gain.

In some rare cases, RRT* shows slightly reduced *aerial drone distance* or fractionally shorter times, but these were not statistically significant across seeds.

5.3.3 Impact of Map Size and Building Density

Unsurprisingly, a 500×500 area or 60 buildings consistently inflates mission times, leads to incomplete rescues, and drives up UAV distance. High obstacle clutter limits ground vehicles, shifting more burden to aerial drones, which then must criss-cross large spaces. This effect is amplified under *centroid*, as UAVs periodically recalculate centroids and relocate.

5.3.4 Summary of Findings

Overall, the data indicate:

- **Nearest approach** is typically the safest choice for fast completion in more challenging scenarios.
- **RRT vs. RRT*** differences are minimal under the tested time and environment scales.
- **Map expansions** and **dense buildings** degrade performance, sometimes leading to partial rescues.
- **Aerial drones** bear the brunt of extended travel in large or cluttered scenarios, while ground vehicles remain more localized.

These patterns form a basis for discussing the limitations, possible enhancements, and real-world applicability of the system in the subsequent chapters.

Chapter 6

Discussion

This chapter reflects on the multi-UAV rescue framework in light of the results presented in Chapter 5. Section 6.1 reviews the system's major strengths and weaknesses, considering both the code architecture and the experimental outcomes. Section 6.2 elaborates on inherent constraints of our simulation approach, while Section 6.3 summarizes key insights gained throughout the project.

6.1 Strengths and Weaknesses

A principal strength of our system is its *modular* design, as highlighted in Chapters 3 and 4. The distinction between environment generation, UAV classes (`BaseUAV`, `AerialDrone`, and `GroundVehicle`), and high-level scripts (`runRescueMission`, `compareApproaches`, `runExperiments`) allows for straightforward extension or modification of each component. In particular, substituting an alternative planner (e.g., D* Lite or PRM) or adding specialized vehicle types does not require rewriting large sections of the codebase. This architecture proved beneficial when enabling both aerial and ground vehicles to operate in parallel, each referencing the same environment representation but validated in 2D vs. 3D.

From a performance perspective, the results in Chapter 5 demonstrate that *nearest-based* survivor assignment is often robust and efficient, especially under higher building densities or larger maps. By contrast, the *centroid* approach, while conceptually promising for balanced coverage, sometimes yielded partial rescues when the environment became large or cluttered. Thus, the multi-assignment framework exhibited both efficiency and flexibility, but remains sensitive to the chosen heuristic.

In terms of weaknesses, RRT* rarely showed a marked improvement over standard RRT, suggesting that within a 600 s mission time, incremental rewiring did not confer significant benefits. The overhead of RRT* planning, especially in a 3D environment with many expansions, may impede the mission enough that the overall time advantage is lost. A further weakness of the current system is its reliance on a *static* environment assumption: buildings do not collapse or move, and survivors remain stationary. Though suitable for a proof-of-concept, it limits real-world realism (citation needed). Additionally, k-means assignment, while implemented in the code, was not fully integrated or utilized in final experiments due to partial reliability issues.

6.2 Limitations

While the simulation framework achieves a broad set of objectives, several limitations should be acknowledged:

Static Obstacles and Survivors. No dynamic or moving obstacles are modeled, nor do survivors shift location. In real disaster scenarios, rubble can shift, additional hazards can arise, or survivors may attempt to relocate (citation needed). Such dynamics would demand real-time replanning.

Perfect Communication. All UAVs are assumed to share information instantaneously, ignoring factors like network latency, bandwidth constraints, or communication failures. Real-world decentralized or partially connected networks would create new complexities (citation needed).

Limited UAV Fleet and Speeds. We employ exactly two ground vehicles and two aerial drones, each with fixed speeds. Though sufficient to demonstrate multi-agent task allocation, larger or more diverse fleets might unveil further coordination challenges, such as interference or load-balancing difficulties.

Finite Time Budget. Experiments used a 600 s limit, beyond which runs were considered incomplete. Certain scenarios (large map, 60 buildings, 25 survivors) frequently timed out, indicating that mission success strongly depends on how quickly feasible paths can be found and survivors assigned. Longer time budgets might reveal whether RRT* eventually outperforms RRT.

Simplified Planner Comparison. Our code primarily compares standard RRT and RRT*. While these capture significant differences in sampling-based methods, other algorithms (PRM, A*, D*) may react differently to dense or high-dimensional spaces. Extending the code to additional planners would further enrich the comparative insights.

6.3 Lessons Learned

Across the implementation and experimental phases, several valuable lessons emerged:

Feasible Paths Often Trump Optimality. One clear takeaway is that in a time-critical rescue context, a quickly discovered feasible path frequently outperforms a slower search for an optimal one, especially when facing partial or uncertain environment information (citation needed). Our results confirm that under realistic time budgets, RRT's speed often suffices.

Multi-Agent Tasking is Key. While single UAV strategies can be effective for small areas or low building densities, the partial rescues encountered in large or cluttered scenarios highlight the need for robust, perhaps dynamic, multi-agent allocation. Our nearest-based approach demonstrated strong performance, but more advanced strategieslike auction-based methodsmay be necessary for truly large fleets or dynamic obstacles.

Environment Generation for Testing. The *procedural* environment model allowed quick iteration across seeds and densities, revealing which parameter combinations cause timeouts. Future expansions could incorporate real-world topologies (e.g., satellite data), bridging the gap between simulation and field testing.

Modular Design Eases Comparisons. Finally, the modular code structure simplified the integration of different path planners and assignment strategies, a principle that future rescue simulators could adopt. Clear separation of environment, UAV classes, and mission script logic not only facilitates experimentation but also clarifies how each subsystem interacts and impacts final outcomes.

Overall, these lessons underscore that real-time multi-UAV rescue scenarios demand both rapid feasibility (RRT) and efficient assignment heuristics (nearest or otherwise) that match the environment's size, clutter level, and available time.

Chapter 7

Conclusion & Future Work

7.1 Summary of Achievements

Over the course of this project, we developed a comprehensive multi-UAV rescue simulation that integrates both ground vehicles (operating in 2D) and aerial drones (operating in 3D). A key accomplishment is the *modular* software design that cleanly separates environment generation, UAV classes, path planning, and top-level mission scripts. This architecture enabled us to seamlessly compare and switch between RRT vs. RRT*, as well as evaluate different survivor assignment heuristics (nearest vs. centroid).

Through extensive experimentation (96 runs) spanning various map sizes, building densities, and survivor counts, we gathered performance metrics such as rescue time, fraction of survivors rescued, and UAV travel distance. Our results showed that:

- **Nearest-based** assignments typically lead to faster completion than centroid in larger or cluttered environments.
- **RRT*** rarely offers a substantial advantage over standard RRT within our 600 s limit, implying that a quick, feasible path is generally more beneficial than an incrementally refined one for time-critical rescues.
- **Aerial drones** shoulder the heaviest travel load in complex scenarios, while ground vehicles effectively handle local survivors if obstacles are not too dense.

Overall, we have demonstrated that even relatively simple sampling-based planners and heuristic assignment methods can effectively coordinate multi-vehicle rescues in procedurally generated static environments.

7.2 Future Extensions

Although the current system serves as a valuable testbed, several avenues exist to broaden its realism and applicability:

- **Dynamic and Real-Time Updates**: Incorporating moving survivors or shifting rubble piles would require on-the-fly replanning and sensor-based occupancy map updates, mirroring real-world disaster volatility.
- **Advanced Task Allocation**: While nearest-based and centroid heuristics suffice for moderate scenarios, large-scale rescues could benefit from auction-based or market-based assignments that handle greater team sizes and dynamic constraints.

- **Additional Planners & Approaches**: Extending beyond RRT or RRT* to algorithms like PRM, D*, or machine-learning-driven planners could uncover more optimal or robust solutions, particularly for high-density or partially unknown environments.
- **Communication Constraints**: The assumption of perfect, instant communication simplifies coordination but diverges from real-world network limitations. Implementing bandwidth, range, and latency constraints would provide a more realistic challenge for multi-UAV rescue operations.
- **Hardware-in-the-Loop Testing**: Finally, connecting this simulator to actual drone or ground-robot hardware would verify whether the software performance translates reliably into the physical domain, paving the way for partial or fully realistic field tests.

7.3 Final Remarks

In conclusion, this project underscores the promise and complexity of multi-UAV rescue systems. By uniting aerial and ground vehicles under a common simulation framework, employing sampling-based planners, and trialing multiple survivor assignment strategies, we have established a baseline for both feasibility and performance in static, cluttered settings. Moreover, our findings highlight that even straightforward approaches can produce successful outcomes in a time-sensitive rescue scenario, provided a feasible path is discovered quickly and survivors are assigned prudently. We hope this work serves as a foundation for ongoing innovation in robotic disaster response, culminating in faster, safer, and more efficient rescue operations worldwide.