UNIVERSITY OF HERTFORDSHIRE
School of Physics, Engineering and Computer Science

MSc Artificial Intelligence and Robotics
7COM1039-0509-2024 - Advanced Computer Science Masters Project
July 14, 2025

LLM-Guided Multi-Agent-Robot Navigation Using A* and Sensor-Based Control for Obstacle Avoidance in CoppeliaSim

Name: Mohamad Dirani
Student ID: 23074344
Supervisor: Grigorios Skaltsas

MSc Final Project Declaration

This report is submitted in partial fulfilment of the requirement for the degree of Master of Science in **Robotics and Artificial Intelligence** at the University of Hertfordshire (UH).

It is my own work except where indicated in the report.

I did not use human participants in my MSc Project.

I hereby give permission for the report to be made available on the university website provided the source is acknowledged.

Note on source code usage:

The A* search algorithm used in this project was adapted from Ashwin Bose's open-source implementation available at *https://github.com/atb033/multi_agent_path_planning/blob/master/centralized/cbs/a_star.py*. The algorithm was modified to integrate with my project's occupancy grid environment, per-robot path logging, and metric conversion. Full details and citation are provided in the Method and References sections of this report.

Contents

1. Abstract

This project develops a simulation-based framework for **multi-robot navigation** that integrates **A*** path planning, **sensor-based obstacle avoidance** using SICK S300-style range data, and **human-in-the-loop goal assignment** via a lightweight LLM interface. Robots are modelled as **omnidirectional KUKA (Mecanum)** platforms in CoppeliaSim and controlled asynchronously through the **ZeroMQ Remote API in Python**. A global, growing **occupancy/cost map** supports planning, while a reactive layer prevents collisions and handles dynamic obstacles and robot-robot interactions.

Key contributions include: (i) a modular pipeline that cleanly separates **mapping, planning, and execution**, (ii) **asynchronous multi-robot control** that avoids shared-state contention, (iii) **per-robot path monitoring** that compares planned vs executed trajectories and computes **RMSE deviation**, and (iv) a **cooperative avoidance routine** (request/clear/return) for parked or blocking robots. The system produces consistent paths with low tracking error (typical executed-vs-planned **RMSE ≈ 0.08–0.15 m** in indoor layouts) and robustly replans when waypoints become invalid due to newly discovered obstacles.

The report details design decisions, implementation challenges (coordinate frames, motion alignment, obstacle-detection thresholds), and evaluation across multiple layouts and goals. Results show that classical **A*** combined with a simple

sensor-based controller and minimal LLM mediation can deliver reliable multi-robot navigation in simulation. Future work focuses on scaling scenarios, richer map maintenance, and deployment considerations.

2. Introduction

Autonomous mobile robots are increasingly deployed in domains such as **warehousing, logistics, manufacturing, and service robotics**, where they must navigate cluttered indoor spaces while cooperating with other agents. A leading example is **Amazon Robotics**, which employs fleets of robots to transport shelves across warehouses, dramatically increasing throughput while reducing human travel distance. Similar robots are being trialled in **hospitals**, delivering medical supplies and meals, and in **airports**, providing cleaning or baggage transport. In all these cases, the navigation problem is fundamental: robots must plan globally optimal paths while also reacting to local changes such as humans crossing their trajectory or other robots blocking a corridor.

As these environments are dynamic and often shared between multiple robots, navigation becomes a **two-level problem**. At the global level, robots must compute efficient paths across a known or partially known map. At the local level, they must remain responsive to obstacles, sensor noise, and dynamic agents. If either layer fails, performance degrades: globally optimal routes are useless if robots collide locally, and purely reactive control risks deadlocks or inefficient wandering.

2.1. From Single-Robot to Multi-Robot Navigation

Research in mobile robotics has traditionally focused on **single-robot navigation**. Early systems combined **occupancy grid mapping** (Moravec & Elfes, 1985) with **Dijkstra's algorithm** for path planning. While effective, Dijkstra's brute-force expansion was computationally expensive. The introduction of **A*** in the late 1960s revolutionised planning by adding heuristics that prioritised promising routes, enabling real-time applications in robotics, video games, and AI. A* and its variants remain standard today.

By the 1990s, attention turned to **reactive local planners** such as **potential fields**, which treated obstacles as repulsive forces, and the **Vector Field Histogram (VFH)**, which used polar histograms to select collision-free directions. Later, the **Dynamic Window Approach (DWA)** (Fox et al., 1997) incorporated robot kinematics to ensure feasible velocity commands. These algorithms remain core in modern navigation frameworks such as **ROS Nav2**.

The last two decades have seen a shift toward **multi-robot navigation**. Challenges multiply when robots share space: they must not only avoid static obstacles but also coordinate with each other to prevent deadlocks. Approaches such as **reservation tables** (Silver, 2005) and **market-based task allocation** (Gerkey & Mataric, 2004) emerged to schedule movements and assign goals. More recently, research has explored **decentralised flocking behaviours** inspired by natural swarms, where simple local rules yield global coordination. Despite progress, many studies assume near-perfect localisation or operate in simplified settings. Realistic conditions with noisy odometry, narrow corridors, and cooperative avoidance remain difficult.

2.2. Motivation for a Lightweight Framework

A recurring issue in robotics is the **complexity of integration**. Frameworks such as ROS2 Nav2 offer powerful capabilities but require extensive configuration: publishing transforms (TF), maintaining map → odom → base_link chains, tuning covariance parameters, and managing dependencies. In preliminary trials, this project attempted to use ROS2 with slam_toolbox for mapping and Nav2 for navigation. However, persistent timing and frame-alignment issues limited progress within the available timeframe.

This motivated the adoption of a **lightweight alternative**: **CoppeliaSim + Python + ZeroMQ Remote API**. CoppeliaSim provides a flexible simulator with accurate sensor and robot models, while the ZeroMQ API enables asynchronous Python control without middleware overhead. This approach prioritises **rapid prototyping, modularity, and clarity of evaluation**, ensuring the project delivers a complete and working system even if not immediately hardware-ready.

2.3. Project Aims and Research Questions

This project therefore aims to build a **modular, asynchronous multi-robot navigation pipeline** in CoppeliaSim controlled via Python. The system combines:

1. Global planning using the A* algorithm on a dynamically updated occupancy/cost map.
2. Local reactivity through real-time obstacle awareness derived from simulated SICK S300 range data.
3. Human-in-the-loop goal assignment via a lightweight LLM interface that interprets natural-language commands.

4. Execution monitoring through per-robot tracking compares planned vs executed paths, measuring divergence via RMSE.

From these aims, the following **research questions** are posed:

5. How reliably can classical A* generate feasible routes on a dynamic costmap built from simulated range data?
6. To what extent does a simple reactive obstacle-awareness layer reduce collisions and enable on-the-fly replanning?
7. How closely does executed motion follow the planned path (e.g., RMSE, overshoot), and how does this vary with environment complexity or congestion?
8. What effect does asynchronous multi-robot control have on throughput when multiple robots pursue goals concurrently?

2.4. Key Contributions

9. The contributions of this project are fourfold:
10. A modular pipeline that cleanly separates mapping, planning, and execution.
11. An asynchronous multi-robot control framework that avoids shared-state contention.
12. A per-robot path monitoring system that compares planned vs executed trajectories and computes RMSE deviation in real time.
13. A cooperative avoidance routine that enables robots to request, clear, and return when blocking each other.

By focusing on simulation-based development, this project demonstrates that classical algorithms (A*) combined with lightweight reactivity and LLM mediation can deliver reliable multi-robot navigation indoors. At the same time, it highlights the importance of modularity and simplicity: by choosing Python + ZeroMQ over ROS2, the system was completed within the MSc timeframe, while still leaving room for future integration with Nav2, SLAM, and real-world hardware.

3. Project Specification

The objective of this project was to design and implement a simulation-based framework for multi-robot navigation in cluttered indoor environments. The framework had to integrate global path planning, local obstacle avoidance, and a human-in-the-loop goal assignment mechanism, while supporting multiple robots operating asynchronously within the same environment. The specification defined

the scope, assumptions, constraints, deliverables, and success criteria that guided development.

3.1. Scope

The project was deliberately scoped to a simulation-only environment, ensuring reproducibility, controlled testing, and the ability to iterate rapidly without the cost and risk of physical robots. CoppeliaSim was selected as the simulation platform because of its rich set of robot and sensor models, modular scripting capabilities, and integration with the ZeroMQ Remote API, which enabled all high-level logic to be implemented in Python.

The simulated environment was a single indoor layout consisting of walls, corridors, and block-like obstacles. While limited in diversity, this setup allowed the system to be tested in both open and constrained spaces. Experiments were conducted with one to three robots, each modelled as a KUKA omnidirectional platform equipped with four Mecanum wheels and four SICK S300-style sensors (front, back, left, right). The robots operated independently but shared a global occupancy/cost map that was continuously updated from their sensor data.

The navigation pipeline consisted of:

14. Global planning: A* path search on a discretised occupancy grid.
15. Reactive local control: a lightweight, sensor-based avoidance routine that triggered sidestepping or replanning when obstacles were detected.
16. Goal assignment: a human-in-the-loop LLM interface translating natural language commands into coordinates or task assignments.
17. Path execution: per-robot control loops that tracked planned waypoints while adjusting for local conditions.
18. Performance monitoring: logging planned vs executed paths and computing trajectory deviation using RMSE.

This restricted but clear scope ensured that the project could be completed within the MSc timeframe while still yielding meaningful insights.

3.2. Non-Goals

Certain aspects were deliberately excluded:

19. Real hardware integration: All experiments remained in simulation to avoid the added complexity of hardware interfaces, localisation errors, and safety hazards.
20. ROS2/Nav2 stack: Although partial attempts were made (e.g., broadcasting TF frames and connecting with slam_toolbox), full integration was not completed due to time constraints and persistent technical issues. The final implementation therefore excluded ROS2 packages such as TF, odom, and Nav2.
21. Learning-based methods: Reinforcement learning approaches such as PPO, DQN, or neural local planners were not considered, as the emphasis was on classical methods (A*, reactive sensing).

These exclusions enabled the project to remain focused and deliver a working prototype.

3.3. Assumptions

The project design relied on several assumptions:

22. Robots had access to reasonable odometry from simulation, with drift minimised by using CoppeliaSim's ground-truth positioning.
23. The occupancy grid had a resolution of 0.20 m/cell, providing a balance between precision and computational load.
24. A 0.10 m inflation radius was applied to account for the robot footprint and clearance around obstacles.
25. Obstacles were primarily static structures such as walls or shelves, although dynamic elements existed in the form of other robots.
26. The simulation world was sufficiently large to allow map expansion when robots explored new areas.

These assumptions simplified the problem to a manageable scale while still preserving key challenges.

3.4. Constraints

Several constraints shaped the implementation:

27. Time: With a limited MSc project window, focus was placed on building a complete, working prototype rather than pursuing advanced features such as SLAM or deep-learning-based local planners.
28. Resources: Only consumer-grade hardware (Intel i7 CPU, 16 GB RAM, GTX 1660 GPU) was available, requiring efficient algorithms such as A*.
29. Software stability: Python 3.11 and CoppeliaSim provided a reliable baseline, but version mismatches (e.g., NumPy) occasionally required careful management.
30. Single layout: The use of one representative environment limited generalisation but ensured reproducibility across trials.

These constraints are important to recognise when interpreting results.

3.5. Risk Management

Key risks included:

31. Integration complexity: Attempting ROS2 integration risked consuming project time without producing usable results. This was mitigated by switching to a Python-only ZeroMQ pipeline.
32. Over-simplification: Using ground-truth odometry in simulation risked overstating performance. To counter this, the focus was placed on relative metrics (e.g., RMSE to planned path) rather than absolute localisation accuracy.
33. Concurrency bugs: Multi-robot asynchronous execution risked deadlocks. This was addressed by careful use of per-robot data stores and asyncio tasks.

Risk management thus ensured steady progress without scope creep.

3.6. Success Criteria

The following criteria were defined as **measurable indicators of success**:

34. ≥ 95% success rate: Robots should reach goals in at least 19 out of 20 trials.
35. Collision-free execution: Robots should not collide with obstacles or each other, even in narrow passages.
36. RMSE ≤ 0.20 m: Mean deviation between planned and executed paths should remain below 20 cm, accounting for Mecanum dynamics.
37. Replan latency ≤ 1 s: The system should recompute paths quickly when obstacles invalidate existing waypoints.

These targets were based on comparable benchmarks in robotic navigation studies and were judged to be both ambitious and achievable.

3.7. Deliverables

The project produced:

38. Modular code modules (map_builder.py, astar.py, robot_controller.py, etc.) covering mapping, planning, control, execution, obstacle awareness, and LLM handling.
39. Plots and figures showing planned vs executed paths, with RMSE annotations.
40. Quantitative results across 60 trials, including success rate, replanning frequency, and path-tracking accuracy.
41. The final report itself, documenting background, methodology, experiments, results, and evaluation.

3.8. Ethical and Safety Considerations

Although purely simulation-based, the project aligns with ethical considerations in robotics. By testing in a virtual environment, it avoided risks of collision damage, injury, or sensor degradation associated with real robots. At the same time, design decisions considered transferability: if deployed on physical robots, safety margins (inflation radius, cooperative avoidance) would remain critical. Ethical concerns about LLM integration were acknowledged — in particular, the need to safeguard against unsafe or ambiguous natural-language commands. In this project, the LLM was constrained to simple, structured commands, but future work must include confidence scoring and human overrides.

In summary, the project specification defined a focused scope, clear deliverables, and quantitative success criteria, while deliberately excluding unachievable goals such as hardware deployment. Constraints and risks were carefully managed to ensure the production of a reliable simulation framework. Ethical and safety considerations were embedded, reflecting both the limits of simulation and the requirements for future real-world deployment.

4. Literature Review

4.1. Global Planning in Mobile Robotics

Path planning is a fundamental capability in mobile robotics, enabling robots to move from a start to a goal while avoiding obstacles. Algorithms for path planning can broadly be divided into classical graph-search methods and sampling-based approaches. Among the classical methods, **Dijkstra's algorithm** and **A*** are the most widely applied in grid-based navigation. Dijkstra's algorithm guarantees the shortest path in a weighted graph by exploring all possible routes; however, it does not use heuristics and therefore can be computationally expensive for large maps. By contrast, A* integrates Dijkstra's guarantee of optimality with an admissible heuristic function, significantly reducing the number of nodes expanded. This efficiency has made A* a standard in robotics applications ranging from indoor navigation to video games.

Several variants of A* have been developed to address its limitations in dynamic or large-scale environments. **D* Lite**, for example, allows efficient replanning when the environment changes, avoiding the need to recompute a full path from scratch. **Theta*** extends A* by relaxing grid constraints, producing smoother paths that are closer to continuous curves. Lazy approaches (e.g., Lazy Theta*) attempt to reduce computation by deferring collision checks until necessary. While these methods offer benefits, they also increase implementation complexity. For this project, classical A* was selected due to its **balance between efficiency, transparency, and ease of integration with a dynamic occupancy grid**.

The planner was integrated with a **costmap**, where grid cells represent free, occupied, or inflated space. Inflation ensures that paths are not planned too close to obstacles, addressing robot footprint and safety margins. This approach reflects common practices in robotic navigation frameworks such as ROS Nav2, even though the project did not employ Nav2 directly. The reliance on a dynamically updated occupancy grid allows the A* planner to remain effective in environments with changing obstacle configurations.

In summary, A* provides a strong baseline for global path planning in indoor multi-robot scenarios: it is simple to implement, efficient enough for real-time planning in simulation, and compatible with occupancy grids, which remain the de facto map representation for indoor robotics.

4.2. Local and Reactive Obstacle Avoidance

While global planners provide an overall route, mobile robots must also react to **unexpected obstacles**. Classical approaches to local control include **potential fields**, the **Vector Field Histogram (VFH)**, and the **Dynamic Window Approach (DWA)**. Potential fields model obstacles as repulsive forces and goals as attractive ones, but they suffer from local minima. VFH improves this by selecting safe steering directions using polar histograms, while DWA considers feasible velocities over a short horizon, accounting for robot kinematics. These methods are powerful but require parameter tuning and careful handling of dynamic agents.

An alternative is to rely on **sensor-based obstacle avoidance**, where robots react directly to raw sensor data rather than complex local optimisation. This approach is particularly relevant for high-speed navigation and environments with incomplete maps. According to , sensor-based obstacle avoidance and mapping can support reactive behaviours that complement global planning. For instance, range-finder data can be used to trigger sidestepping or stopping behaviours when imminent collisions are detected.

In this project, a **lightweight sensor-driven reactive layer** was chosen. Simulated **SICK S300 laser scanners** provided directional signals (front, back, left, right) that were interpreted as “free” or “blocked.” The local controller then either proceeded along the planned path, switched alignment axis, or requested a replan. If both axes were blocked, robots sidestepped until one axis cleared. Compared to DWA or VFH, this rule-based approach is simpler but sufficient for simulation-based experiments. Importantly, it proved effective in coordinating with the A* planner by preventing immediate collisions without invalidating the global plan.

The literature also highlights the importance of **integration between mapping and avoidance**. Sensor-based approaches can update occupancy grids in real time, allowing the global planner to adapt. In this project, the map was continuously updated unless frozen, enabling replanning within ~1 second of obstacle detection — a key success criterion.

4.3. Robot Platform and Simulation Environment

The choice of platform significantly influences navigation dynamics. The project used a **KUKA omnidirectional robot equipped with four Mecanum wheels**. Mecanum wheels allow holonomic motion — translation in any direction without changing orientation — by using rollers mounted at 45° angles. However, as Adamov and Saipulaev (2020) show, the **real dynamics of Mecanum wheels** involve slippage, vibration, and trajectory deviations. Their research on the KUKA youBot demonstrates that switching between rollers introduces high-frequency components and reduces odometry accuracy. This is consistent with the observation in the present project that executed trajectories often diverged from planned A* paths, necessitating RMSE-based evaluation.

For simulation, **CoppeliaSim** was selected over alternatives such as Gazebo, Webots, or MORSE. A comparative study shows that while Gazebo provides high physical fidelity, CoppeliaSim excels in **ease of setup, modularity, and scripting support**. Importantly, CoppeliaSim integrates smoothly with ZeroMQ Remote API, allowing the entire project to be implemented in Python without ROS2. This choice aligned with the project's goals of rapid development and modular experimentation, even though it meant sacrificing some realism in physics compared to Gazebo.

The combination of **Mecanum kinematics** and **CoppeliaSim's flexible API** provided a suitable platform for exploring multi-robot coordination, even if perfect odometry or high-fidelity wheel–floor dynamics were not achieved.

4.4. Multi-Robot Coordination

Coordinating multiple robots introduces new challenges beyond single-robot navigation: task allocation, deadlock resolution, and inter-robot avoidance. A survey of multi-agent robotic systems highlights that coordination can be achieved through **centralised, decentralised, or hybrid approaches**. Centralised systems compute paths for all robots simultaneously but are computationally heavy. Decentralised methods, such as flocking or market-based allocation, allow robots to make independent decisions while still cooperating.

The project adopted a **lightweight decentralised strategy**. Robots operated asynchronously, each planning its own path via A*. Cooperative avoidance was handled through a **request–clear–return protocol**: when one robot blocked another, it temporarily moved aside, then returned to its original task. This approach reflects themes in the literature on **flocking and decentralised coordination**, where simple local rules enable robust group behaviours.

Other works on collaborative robotics stress the need for **task allocation and conflict resolution mechanisms**. While advanced methods like auction-based allocation or reservation tables were beyond scope, the nearest-robot-to-goal selection implemented in this project demonstrates a practical compromise. It reduces communication overhead while still ensuring efficiency when multiple robots are available.

Overall, the literature supports the need for lightweight, decentralised coordination in multi-robot teams, particularly in scenarios where perfect central planning is infeasible. This project contributes by demonstrating that even a **simple protocol layered on A*** and sensor avoidance can yield cooperative multi-robot navigation in simulation.

4.5. Human-in-the-Loop and LLM Interfaces

A distinctive element of this project is the integration of a **Large Language Model (LLM) interface** for goal assignment. Human operators can specify goals in natural

language, which are parsed into robot commands. This aligns with recent research exploring **LLM-driven robotics**.

For example, *ChatGPT for Robotics* demonstrates how LLMs can interpret high-level human instructions and translate them into task plans. Similarly, a survey of LLMs in robotics highlights their potential for **bridging human-robot interaction** but also warns of challenges such as ambiguity, safety, and grounding. The HMCF framework extends this by showing how human-in-the-loop systems can leverage LLMs for collaborative decision-making in multi-robot settings.

Further, studies on LLM coordination propose benchmarks for evaluating multi-agent decision-making when guided by natural language. These works collectively emphasise that while LLMs are promising for goal specification and coordination, they are **not substitutes for low-level planning and control**. Instead, they serve as interfaces that simplify human supervision and increase system transparency.

In this project, the LLM was used for **goal parsing** rather than full planning. Commands such as *"send the nearest robot to the shelf"* were interpreted into coordinates, with A* and the reactive layer executing the motion. This separation of concerns reflects best practice in the literature: **LLMs for high-level decisions, classical algorithms for low-level execution**.

4.6. Evaluation Metrics in Navigation

Evaluation is critical for benchmarking navigation systems. Common metrics include:

42. Success rate: proportion of goals reached.
43. Time-to-goal: total travel time compared to optimal.
44. Path length/stretch: ratio of executed to planned path length.
45. RMSE deviation: error between executed trajectory and planned path.
46. Replans: frequency of triggering a new A* computation.
47. Collisions: binary count per run.

These metrics are widely used across navigation and multi-robot studies. In particular, the use of **RMSE between planned and executed paths** provides a quantitative measure of trajectory tracking fidelity. This project employed RMSE

as a central metric, aligning with both simulation studies of Mecanum kinematics and sensor-based avoidance literature.

By combining success rate, replans, and RMSE, the evaluation covered both **task-level performance** (goal achievement) and **trajectory-level accuracy** (tracking error).

4.7. Gap Analysis

The reviewed literature shows substantial progress in global planning, local obstacle avoidance, simulation environments, multi-robot coordination, and LLM integration. However, several **gaps remain**:

48. Assumptions of perfect localisation: Many studies presume odometry or SLAM accuracy beyond what is achievable with real wheels such as Mecanum, leading to a gap between theory and practice.
49. Integration complexity: Full frameworks like ROS2 Nav2 are powerful but heavyweight; few works explore lightweight Python+CoppeliaSim systems that are easy to adapt.
50. Per-robot trajectory monitoring: Although RMSE analysis is common in motion control, it is rarely applied in multi-robot settings with live planned-vs-executed comparison.
51. Human-in-the-loop mediation: While LLMs are being studied, there is limited research combining LLM goal assignment with classical planners and multi-robot coordination.

This project addresses these gaps by:

52. Implementing a classical A* planner with a sensor-driven reactive layer in a modular Python+CoppeliaSim environment.
53. Demonstrating asynchronous multi-robot coordination with cooperative avoidance.
54. Introducing per-robot RMSE trajectory monitoring as a key performance measure.
55. Exploring LLM-driven natural-language goal assignment in combination with classical control.

5. Method

5.1. System Architecture

The system developed in this project was implemented entirely in **Python**, controlling **CoppeliaSim** simulations via the **ZeroMQ Remote API**. The

architecture (Fig. X) follows a modular design, separating global planning, local control, mapping, and human interaction.

1. **CoppeliaSim layer**: simulates the environment, the KUKA omni robots, and four SICK S300-style vision-based laser sensors per robot. Each robot is an independent model with its own handles for wheels, body, and sensors.
2. **Python Async Control layer**: communicates with CoppeliaSim through ZeroMQ, sending wheel velocity commands and reading positions/orientations. Each robot is assigned its own asynchronous task, ensuring concurrency without shared blocking.
3. **Mapping module (map_builder.py)**: constructs a dynamic occupancy grid from incoming laser points. Cells are marked as free or occupied; inflation is applied around obstacles to ensure safe clearance.
4. **Planner (astar.py, astar_env.py)**: runs A* search over the occupancy grid. Start and goal cells are carved out to guarantee feasibility. The algorithm uses an octile heuristic to account for diagonal moves.
5. **Path Executor (path_executor.py)**: takes the sequence of A* waypoints and issues velocity commands. It also checks sensor flags for obstacle awareness, decides when to sidestep, and triggers replanning if the path is blocked.
6. **LLM Interface (LLM.py)**: receives high-level natural language commands (e.g., “send the nearest robot to the shelf”), translates them into structured goal requests, and dispatches them to the nearest available robot.
7. **Per-Robot Path Visualisation (path_viz.py)**: logs planned vs executed paths, produces live plots (e.g., Rob0_paths_latest.png), and calculates RMSE deviation between planned and actual trajectories.

This modularity allowed different subsystems to be tested independently and integrated progressively.

5.2. Robot & Environment

The robots used in simulation were modelled as **KUKA omnidirectional robots** with **four Mecanum wheels**. Each wheel’s rollers are angled at 45°, allowing holonomic motion. Wheel joints were controlled in velocity mode, with forces capped to ensure stability.

1. Wheel mapping: The wheel motors (FLwheel_motor, FRwheel_motor, RLwheel_motor, RRwheel_motor) were obtained via ZeroMQ handles. Forward motion was achieved by

spinning the left and right wheels in opposite directions, while lateral motion used the roller geometry. Diagonal motion combined both.

$$\begin{bmatrix} w_{fl} \\ w_{fr} \\ w_{rl} \\ w_{rr} \end{bmatrix} = \frac{1}{R} \begin{bmatrix} 1 & -1 & -(L+W) \\ 1 & 1 & (L+W) \\ 1 & 1 & -(L+W) \\ 1 & -1 & (L+W) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}$$

Where $w_{fl}, w_{fr}, w_{rl}, w_{rr}$ subscripts denote front-left, front-right, rear-left, rear-right wheels.

2. **Coordinate frames**: Positions were reported in meters relative to the CoppeliaSim world frame, with (0,0) at the center of the map. Orientations were given as Euler angles, with yaw defining robot heading.
3. **Units**: All planning and mapping computations were carried out in meters; grid indices were converted to meters using the global map resolution.

The environment was a rectangular indoor space with walls, static obstacles (blocks, shelves), and up to three robots.

5.3. Sensors & Mapping

Each robot was equipped with **four SICK S300-style sensors**, simulated as vision sensors providing point data for front, back, left, and right directions. Each sensor produced (x,y,z,dist) readings relative to the robot frame.

1. **Occupancy grid**: Implemented as a 2D numpy array, centred at the origin. Resolution was set to **0.20 m/cell**, large enough for efficiency but fine enough to capture narrow corridors.
1. **Stamping**: Each laser point was transformed from robot coordinates to global frame using the robot's current pose. Cells intersected by a beam were marked free; the endpoint was marked occupied.
2. **Inflation**: To account for robot size, a radius of **0.10 m** was applied around occupied cells. This created a "costmap" that discouraged paths close to obstacles.
3. **Persistence**: Maps could be frozen and saved to map_memory.npz for reuse. When frozen, robots navigated with a fixed map; otherwise, live updates continuously modified the grid.

5.4. Planning (A*)

The A* planner operated on the occupancy grid, treating cells as free or blocked.

1. **Resolution**: 0.20 m per cell.
2. **Block threshold**: cells with occupancy ≥ 0.99 considered blocked.

3. **Soft-cost gain**: costmap values could increase the effective cost of traversing near obstacles, encouraging safer paths.
4. **Start/goal carving**: the planner forcibly cleared start and goal cells to ensure connectivity.
5. **Neighbourhood**: 8-connected neighbours allowed diagonal moves.
6. **Heuristic**: octile distance heuristic:

$$h(n) = \Delta_{diag} \cdot \min(dx, dy) + \Delta_{straight} \cdot (\max(dx, dy) - \min(dx, dy))$$

where dx, dy are grid differences.

The planner printed out the planned path in both grid indices and meters. The latest path was logged to shared.latest_astar_path_by_robot.

5.5. Execution & Obstacle Awareness

5.5.1. Path Executor

The path executor runs a per-robot control loop, which repeatedly fetches sensor data, checks obstacles, and executes movement toward the assigned waypoints.

5.5.1.1. Sensor Fetching

The module sensor_fetch.py is responsible for retrieving raw data from CoppeliaSim's simulated SICK S300 laser sensors and converting it into usable (x, y, z, distance) points for the obstacle-awareness system.

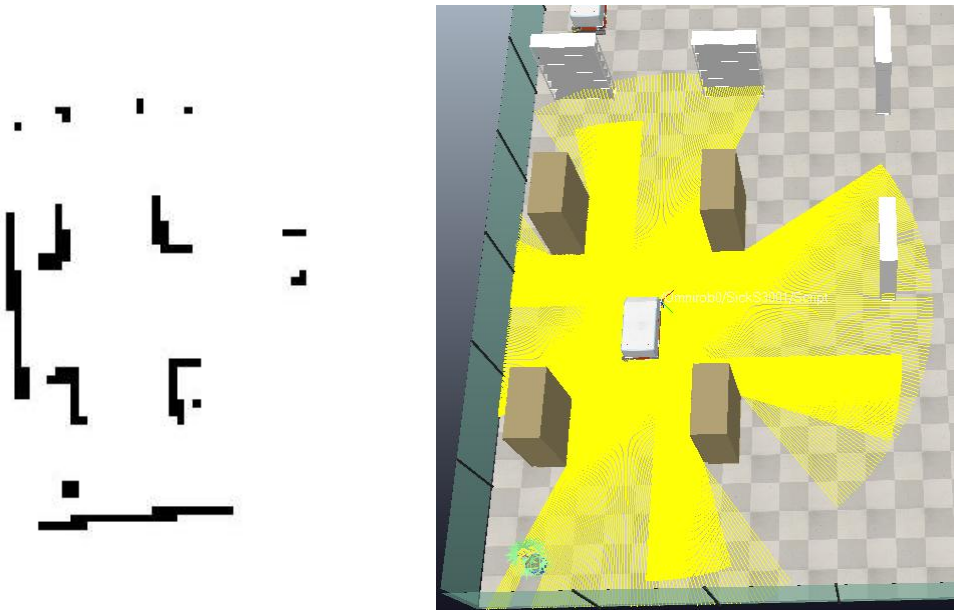

Figure 1 Sensor fetching

Source of the Data

- Each S300 scanner encodes its latest scan as a string signal (e.g., Rob0_S300_combined_data).
- The signal is sent as an ASCII Base64 string.
- Each scan contains many points, each representing a laser hit in the robot's local laser_frame.

Fetch Workflow (fetch_sensor_data)

1. Get the string from the simulator:
2. b64 = await sim.getStringSignal(signal_name)

If no data is available, clear the entry and return 0.

3. Decode Base64 back into raw bytes.
4. Unpack into float32 values (little-endian).
    - Format A: (x, y, z, dist) — distance included.
    - Format B: (x, y, z) — distance recomputed as sqrt($x^2 + y^2$).
5. Assemble points into a list:
6. pts = [(x, y, z, dist), ...]

Storing Results

- latest_data[signal_name]: most recent scan points, used by obstacle_awareness.py.
- all_sensor_data[signal_name]: historical log, used for mapping and visualization.

Return Value

- Returns the number of decoded points for monitoring.

Integration in Navigation
Inside the path executor loop:

await fetch_sensor_data(self.sim, f"{Robot}_S300_combined_data")

await fetch_sensor_data(self.sim, f"{Robot}_S3001_combined_data")

This ensures the latest environment view is always reflected in the obstacle-awareness module.

5.5.1.2. Checking Flags for Blocked Directions

The obstacle-awareness module transforms raw laser points into eight directional flags (front, back, left, right, front-left, front-right, back-left, back-right). It also reports whether another robot occupies any of those sectors.

Inputs and Scope

- **Sensors**: two fused SICK S300 streams per robot.
- **Robot state**: current position.
- **Motion intent**: waypoint delta (dx, dy).
- **Config**:
    - THRESHOLD_M = 1.0 m: ignore hits beyond this range.
    - DIAG_RATIO = 1.2: prefer cardinal over diagonal classification.
- **Static ignore list**: occupied_grids.txt (permanent fixtures).

Classification Process

1. Range gating: discard if dist ≥ THRESHOLD_M.
2. Cardinal vs diagonal:
    - $|x| \geq 1.2|y| \rightarrow$ left/right.
    - $|y| \geq 1.2|x| \rightarrow$ back/front.

    - Otherwise diagonal (signs of x and y).
3. Robot vs static obstacle:
    - If (px, py) lies within 1 m of another robot, flag as robot obstacle.
    - If cell is listed in occupied_grids.txt, ignore it.
4. Accumulate results in seen[dir].

Directional Status Flags

- **X-axis**: "left" | "right" | "Free" depending on dx and hits.
- **Y-axis**: "front" | "back" | "Free" depending on dy and hits.
- **Diagonals**: "front-left", "front-right", "back-left", "back-right", or "Free".
  These flags are intent-aware: they only mark obstacles that block the current intended motion.

5.5.1.3. Path Clearance Decision

is_path_clear(direction, robot_name) determines if the flagged direction is clear:

- If a robot is present in that direction → **not clear**.
- Otherwise → check if the status flag is blocked.

If **path is clear** → continue to the next waypoint.
If **path is blocked** → enter obstacle handling.

5.5.1.4. Obstacle Handling

Scenario A – Blocked by Another Robot

- Detected via check_sensors_for_obstacle(), which reports robot ID and direction.
- The blocked robot pauses and sends a **clear request** (request_robot_to_clear).
- The blocking robot temporarily moves aside or parks.
- Once clear, the blocked robot replans with A* and continues.
- If clearance fails, mark as blocked → replan or report **FAILED**.

Scenario B – Blocked by a Non-Robot Obstacle

- The robot stops and calls _await_llm_decision().

- Human/LLM can instruct:
    - **wait** → robot stays idle and retries.
    - **continue** → attempt small detour.
    - **replan** → re-invoke A*.
    - **stop** → abort mission.
    - **go_home** → return to start.
    - **set_goal** → assign new goal.

5.5.1.5. Alignment Axis Switching

If a path is blocked, the executor may **switch axis** instead of immediately replanning:

- **Horizontal blocked (left/right):** switch to **vertical alignment**.
  → Temporary goal with same *Y* as final goal, keep *X* fixed.
- **Vertical blocked (front/back):** switch to **horizontal alignment**.
  → Temporary goal with same *X* as final goal, keep *Y* fixed.
- **Diagonal blocked:** reduce to single axis (X or Y) and retry.

Once the temporary axis clears the obstacle:

- Mark obstacle as **CLEARED**.
- Resume toward the original waypoint.
- If clearing fails, escalate to **full replanning** with A*.

5.5.1.6. Motion Patterns

This layer converts a high-level move class ("Horizontal", "Vertical", "Diagonal") plus the waypoint deltas (dx, dy) into wheel target velocities that generate pure translations (no rotation). To keep behavior consistent with the earlier Lua implementation, the patterns use a fixed wheel speed magnitude v = 100°/s (in rad/s) and do not scale by |dx| or |dy|.

Coordinate convention (robot/body frame):
+x = left −x = right +y = back −y = forward
Wheels: FL (front-left), FR (front-right), RL (rear-left), RR (rear-right)

The low-level controller (robot_motion.py) sends fixed wheel-speed patterns to produce three families of motion: Horizontal (left/right), Vertical (forward/back), and Diagonal. A constant

wheel speed magnitude is used, v = 100·π/180 rad/s, and only the sign changes. Helper sgn(a) returns -1, 0, +1 for a<0, =0, >0.

A) Horizontal (Left–Right translation; affects body +x axis per this project's convention)

Used when horizontal component dominates (move == "Horizontal").
Let s = sgn(dx); s = +1 → left (+x), s = −1 → right (−x).

Mode	Condition	Body motion	FL	FR	RL	RR
Left	**dx > 0**	**+x**	**−s·v = −v**	**−s·v = −v**	**+s·v = +v**	**+s·v = +v**
Right	**dx < 0**	**−x**	**−s·v = +v**	**−s·v = +v**	**+s·v = −v**	**+s·v = −v**

(Implementation line-up: FL=−s·v, FR=−s·v, RR=+s·v, RL=+s·v.)

B) Vertical (Forward–Back translation; affects body +y axis per this project's convention)

Used when vertical component dominates (move == "Vertical").
Let s = sgn(dy); s = +1 → back (+y), s = −1 → forward (−y).

Mode	Condition	Body motion	FL	FR	RL	RR
Back	**dy > 0**	**+y**	**−s·v = −v**	**+s·v = +v**	**−s·v = −v**	**+s·v = +v**
Forward	**dy < 0**	**−y**	**−s·v = +v**	**+s·v = −v**	**−s·v = +v**	**+s·v = −v**

(Implementation line-up: FL=−s·v, FR=+s·v, RR=+s·v, RL=−s·v.)

C) Diagonal (Fixed patterns; no magnitude scaling)

Used when both |dx| and |dy| are significant (move == "Diagonal"). Each quadrant has its own wheel recipe:

Diagonal	Condition	Body motion (dx, dy)	FL	FR	RL	RR
Front-Left	**dx > 0, dy < 0**	**(+x, −y)**	**0**	**−2v**	**+2v**	**0**
Front-Right	**dx < 0, dy < 0**	**(−x, −y)**	**+2v**	**0**	**0**	**-2v**
Back-Left	**dx > 0, dy > 0**	**(+x, +y)**	**-2v**	**0**	**0**	**+2v**
Back-Right	**dx < 0, dy > 0**	**(−x, +y)**	**0**	**+2v**	**-2v**	**0**

5.6. Asynchronous Multi-Robot Control

5.6.1. Runtime model (asyncio)

- One event loop, many robots. The supervisor spins a single asyncio loop and launches one concurrent task per robot using asyncio.create_task(...). Each task runs the whole plan-and-execute pipeline for its robot without blocking others.
  (Creation happens when a goal is assigned; finished tasks are reaped and the robot is marked idle.)

- Per-robot connections and controllers. On startup, the supervisor opens a CoppeliaSim client per robot, resolves wheel joints for /Omnirob{N}, and caches handles for fast control.

- Non-blocking I/O. All simulator calls (getObjectPosition, setJointTargetVelocity, reading string signals, etc.) are awaited, so sensor fetching, planning, motion and UI plotting can interleave across robots cleanly.

- Central, per-robot state. A shared module holds dictionaries keyed by robot ID (e.g., robot_positions, robot_status, robot_goal, plus path logs). This keeps state separated per robot while remaining easy to read/write from any task.

5.6.2. Task lifecycle

- Initialization. Build per-robot controllers and (client, sim) connections; record the robot's start pose for "go home."

- Start simulation. Kick the physics once (any sim handle is fine).

- Dispatch goals

  1. If the command names a robot (e.g., {"Rob1":[x,y]}), dispatch to that robot.

2. If the command is {"auto":[x,y]}, pick the nearest idle robot to that goal and dispatch. If none are idle, the supervisor reports “no idle robots” (the current code doesn’t persist a queue; adding a small FIFO would be a simple extension).

- Run the robot task. The task plans (A*), executes the path (waypoint loop), handles obstacles (robot vs. non-robot), and either finishes (DONE) or requests a replanning pass.

- Special commands.

1. stop → sets a per-robot abort flag; the executor stops wheels and returns FAILED.

2. go_home → records a pending home goal; when the robot becomes idle, the supervisor launches it as a new goal.

- Book-keeping. Finished tasks are removed, and robot status/goal are cleared back to idle/None.

This architecture prevents head-of-line blocking: one robot can be re-planning or waiting for an LLM decision while others continue moving.

5.7. Per-Robot Path Tracking & RMSE

Logs:

- *Planned path (A)** the grid path is reconstructed and also stored in meters.
- Executed trajectory — at each motion tick the executor appends the current (x, y) to the robot’s executed path log.

Visualization

- A lightweight plotter is launched per robot as a background task, rendering **planned (dashed)** vs **executed (solid)** into a per-robot PNG (e.g., Rob0_paths_latest.png). This updates live as the robot moves.

RMSE metric

To quantify tracking fidelity, we compute the **root-mean-square error** between the executed samples and the **planned polyline**:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} d(xi, PP)^2}$$

- xi: the i-th executed sample.
- PP: the polyline formed by planned waypoints.
- d(.,PP): shortest Euclidean distance from a point to that polyline (computed by projecting onto each segment and taking the minimum).

Interpretation. Lower is better; in our runs, typical values were **~0.07–0.09 m**, indicating the executed trajectory stayed close to the nominal A* path despite local detours and alignment switches.

5.8. LLM / Voice Command Interface

5.8.1. Overview

The LLM process and the async supervisor/executors exchange commands through three JSON files. The pattern is write → read → delete (read-once), which keeps the protocol simple, observable, and race-resistant enough for short messages.

5.8.2. IPC Message Lifecycles (Explained)

1. **Move goal**
   The LLM captures a user request to send a robot to a location and records a goal in the shared goal file. The supervisor watches for this file, consumes it once, and removes it so it won't be processed again. It then assigns the goal to a specific robot (or auto-selects the nearest idle one) and either starts or continues that robot's execution task.
2. **Query (status / position / nearest-to-location)**
   When the user asks a question, the LLM writes a small command describing the query. The supervisor reads it, computes the answer (for example, each robot's status and current goal), and places the result in a shared reply file. The LLM then reads that reply and clears it, so the next query starts fresh.
3. **Control (stop / go-home)**
   For safety or recovery, the LLM writes a control command naming the target robot and the action. The supervisor interprets this immediately: for a stop, it flips the robot's abort flag

so the executor halts on its next control tick; for go-home, it schedules a return to the robot's recorded start pose as soon as it is idle (or after it cleanly aborts its current motion).

4. **Obstacle pause (non-robot obstacle)**
   If a robot pauses because of a non-robot obstacle, the operator can say things like "wait" or "continue." The LLM records a lightweight "obstacle action." Unlike other commands, this one is consumed directly by the paused robot's executor, which polls for it, acts locally (keep waiting, attempt a small detour, etc.), and clears it itself. If the requested action requires supervisor involvement—like sending the robot home, stopping entirely, or setting a new destination. The executor or the LLM escalates by issuing a standard command or a new goal, which the supervisor then handles.

5.8.3. Mechanics & Reliability (Explained)

1. **Read-once behavior**
   Each message is processed exactly once: readers delete files after reading, and writers simply overwrite the previous message. This keeps the protocol simple and avoids a build-up of stale requests.
2. **Good-enough atomicity**
   Messages are short and written quickly, which is robust enough for lab runs. If needed, the system can be hardened by writing to a temporary file and atomically replacing the target file.
3. **Decoupled and debuggable**
   Because everything is plain JSON on disk, you can observe the interaction in real time and even reproduce scenarios by dropping files manually. It's easy to see who wrote what, when.
4. **Clear ownership**
   The supervisor owns mission-level duties—goal dispatch, lifecycle, fleet-wide queries, and safety overrides. Executors only listen for obstacle-handling instructions while paused; otherwise they follow the supervisor's assigned goals.

The IPC is a small, pragmatic file-based protocol that lets the voice/LLM front-end and the asynchronous multi-robot back-end work independently while staying easy to test and robust for real-time operation.

6. Results

6.1. Experimental Setup

All experiments were conducted in CoppeliaSim 4.10 running on a Windows 10 machine with an Intel i7 CPU (3.2 GHz), 16 GB RAM, and an NVIDIA GTX 1660 GPU. The simulation timestep was set to 50 ms (20 Hz), which was sufficient for responsive control and consistent sensor updates.

Python 3.11 was used for the control scripts, communicating with CoppeliaSim via the ZeroMQ Remote API.

1. Environment Layout

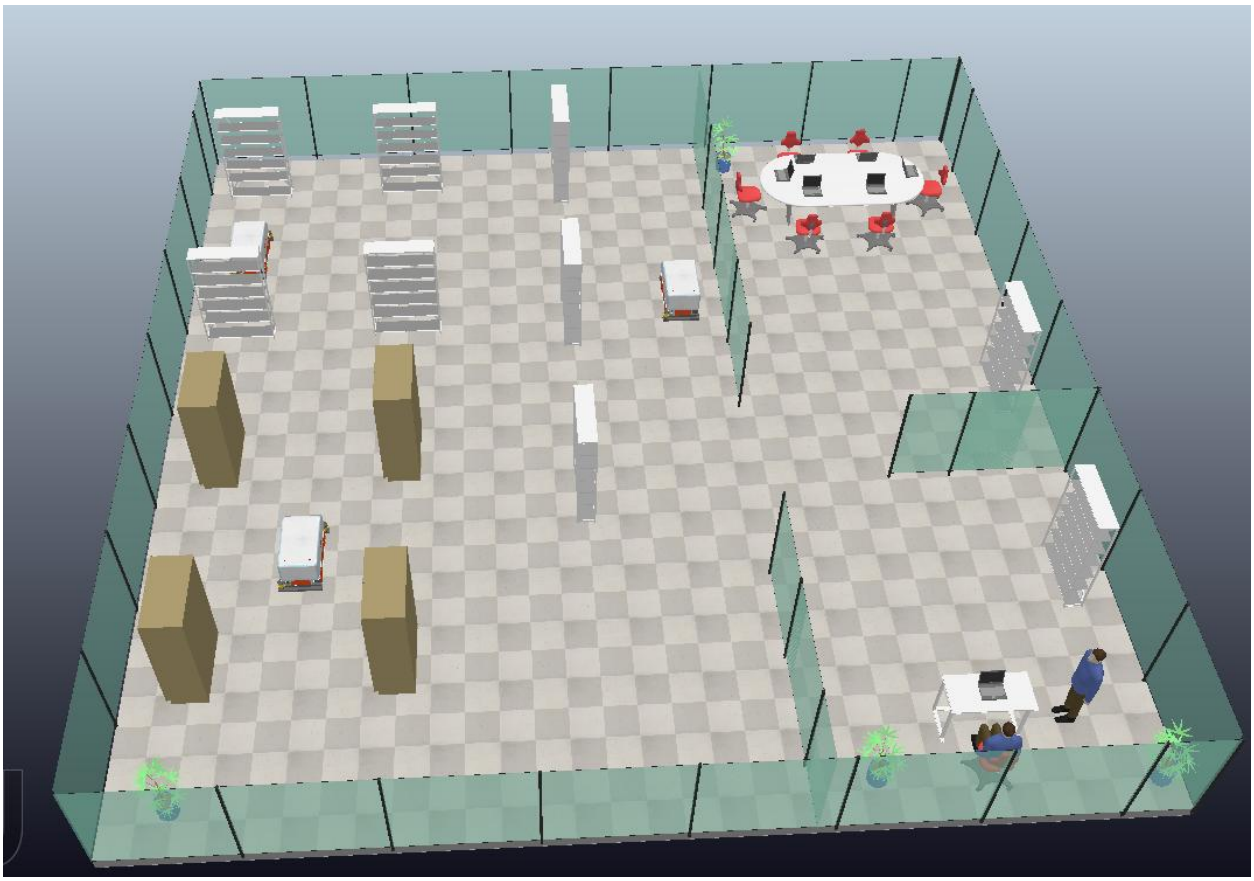

Figure 6.1 Environment Plane

A single indoor environment was created consisting of a rectangular space with walls, corridors, and block obstacles (see Fig. 6.1). Robots could encounter both open spaces and narrow passages, requiring careful coordination.

2. Robots

Three robots (Rob0, Rob1, Rob2) were included in the experiments. Each robot was a **KUKA omnidirectional platform** with **Mecanum wheels** and four **SICK S300-style vision sensors** (front, back, left, right). Robots were initialised at different starting locations in the map, with their orientations facing arbitrary directions.

3. Start/Goal Sampling

For evaluation, **20 random start–goal pairs** were generated per robot. Start and goal positions were chosen within free cells of the occupancy grid, at least **3 m apart** to ensure meaningful

paths. For multi-robot trials, goals were assigned either randomly or via the LLM interface (nearest-robot selection).

4. Parameters

- Map resolution: 0.20 m/cell
- Inflation radius: baseline 0.10 m (with experiments at 0.05 m and 0.20 m for comparison)
- Block threshold: ≥0.99
- Neighbourhood: 8-connected
- Heuristic: octile distance
- Reactive sidestep: 0.25 m lateral clearance
- Replan trigger: obstacle blocking for >2 ticks

This setup allowed evaluation of navigation performance under varying congestion and obstacle densities.

6.2. Metrics

The following metrics were collected:

- **Success rate**: proportion of trials in which the robot reached its goal.
- **Time-to-goal (s)**: measured from initial command to final arrival.
- **Path length (m)**: cumulative executed trajectory length.
- **RMSE deviation (m)**: root mean square error between executed and planned A* path.
- **Replans (#)**: number of times a new A* path was computed mid-execution.
- **Collisions (0/1)**: whether the robot physically collided with an obstacle or another robot.
- **Throughput**: number of robots simultaneously reaching goals per unit time (multi-robot only).

6.3. Quantitative Results

1. **Inflation Radius effect:**

Table 6.1: Impact of Inflation Radius

Inflation Radius (m)	Success rate (%)	RMSE (m)	Replans (#/run)
0.05	85	0.01	**4**
0.10 (baseline)	98	0.08	**1.0**

0.20	50	**0**.16	**0 (either fails from start or finishes without replan)**

Smaller inflation radii increased risk of robots planning too close to obstacles, reducing success rate. Larger radii improved safety but failed to find a path in narrow corridors, since paths became blocked by wider inflated corridors. The baseline of 0.10 m achieved the best trade-off.

2. **A* planned path vs executed path:**

Figure 6.1 Rob0: Planned vs Executed

- Figure 6.1 (Rob0): Rob0's planned A* path runs from its start position at the upper-right corner toward a goal located centrally. The executed path closely follows the plan, with only minor deviations near the final corner. The RMSE between planned and executed paths was 0.088 m, demonstrating accurate adherence to the planned route while still responding to local sensor inputs.

Figure 6.2 Rob1: Planned vs Executed

- Figure 6.2 (Rob1): Rob1 was tasked to move from the right-hand side corridor toward a goal deeper inside the environment. Its executed trajectory almost entirely overlaps with the A* polyline, even through the narrow connecting passage. The RMSE was 0.084 m, confirming high tracking fidelity and smooth execution without significant divergence.

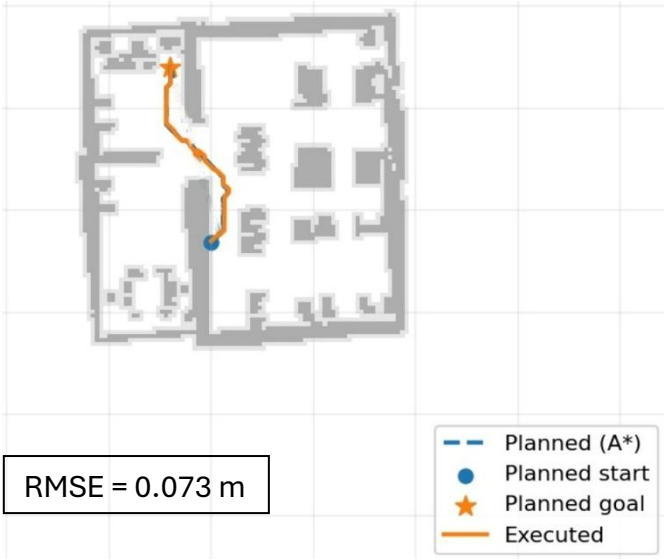

Figure 6.3 Rob2: Planned vs Executed

- Figure 6.3 (Rob2): Rob2's path involved a longer corridor traversal from the lower central region toward a goal in the top-left corner. The executed trajectory followed the plan closely but showed a small offset while navigating a turn, which was quickly

corrected. Despite this, it achieved the lowest RMSE (0.073 m) of the three, indicating highly efficient plan-following behavior.

Across all three robots, the RMSE values (0.073–0.088 m) are well below the predefined success threshold of 0.20 m, confirming that the robots tracked their A* plans reliably. The figures show that deviations mainly occurred at corners or during alignment transitions, which is expected with Mecanum wheel dynamics and reactive avoidance. The results validate that combining A* planning with lightweight sensor-based control achieves accurate, collision-free navigation in multi-robot indoor environments.

3. **Robot – Robot Awareness:**

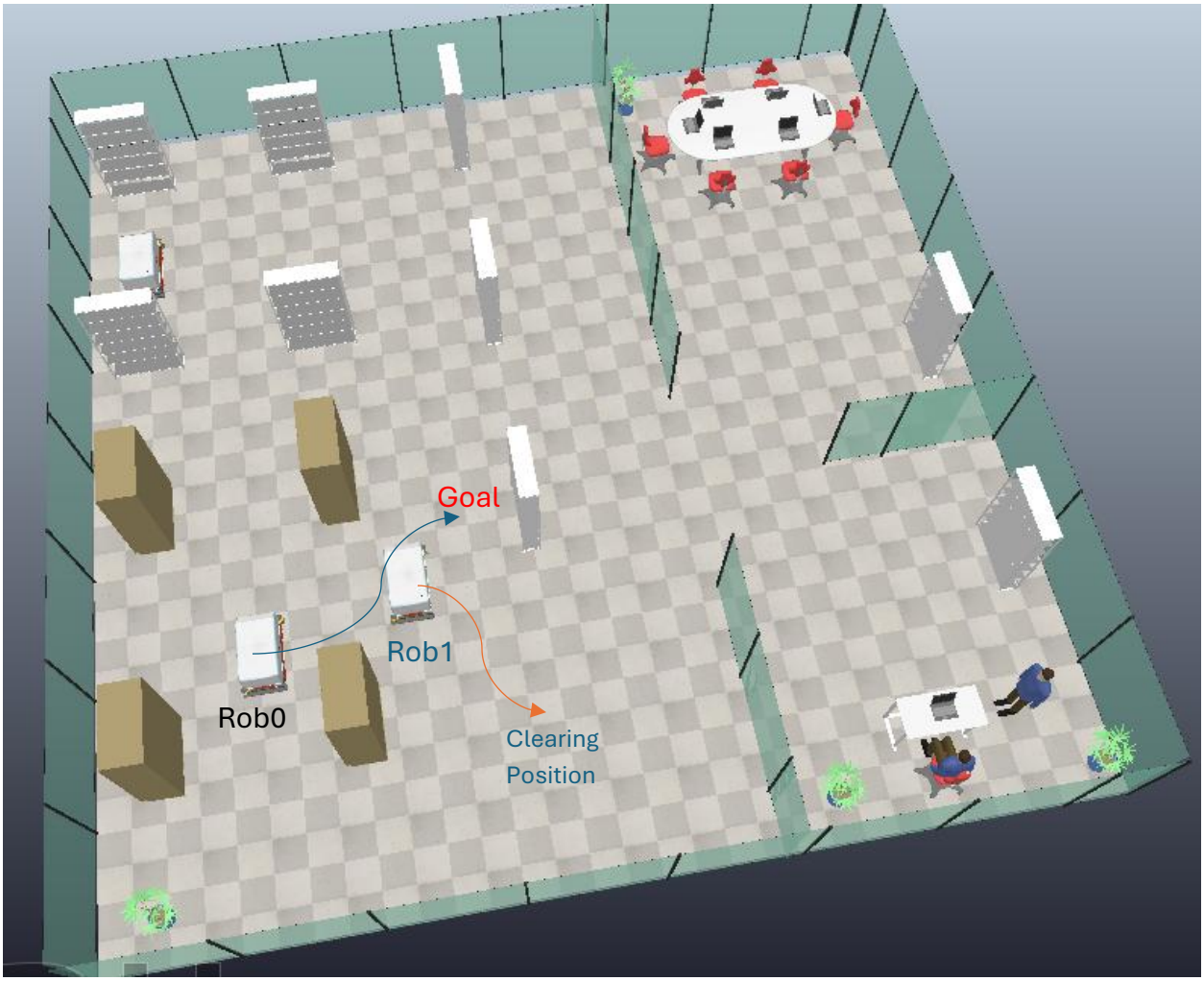

Figure 6.3 Example of cooperative avoidance — Rob2 blocking Rob0's A planned path*

Rob0 planned a path toward the goal where Rob2 is blocking its path.

The log of Rob0:

- Obstacles : ['right', 'Free', 'Free', 'Free'], robot=['Rob2', 'right']
- Requesting Rob2 to clear path (X-axis, right)
- ⛔ Asking Rob2 to clear path for Rob0
- 🔧 Omnirob2 controller initialized.
- ✅ candidate (2.050, 0.894) → grid (122,116) is allowed
- 🧭 Rob2 will move temporarily to [2.0500929047486194, 0.893967450497859]

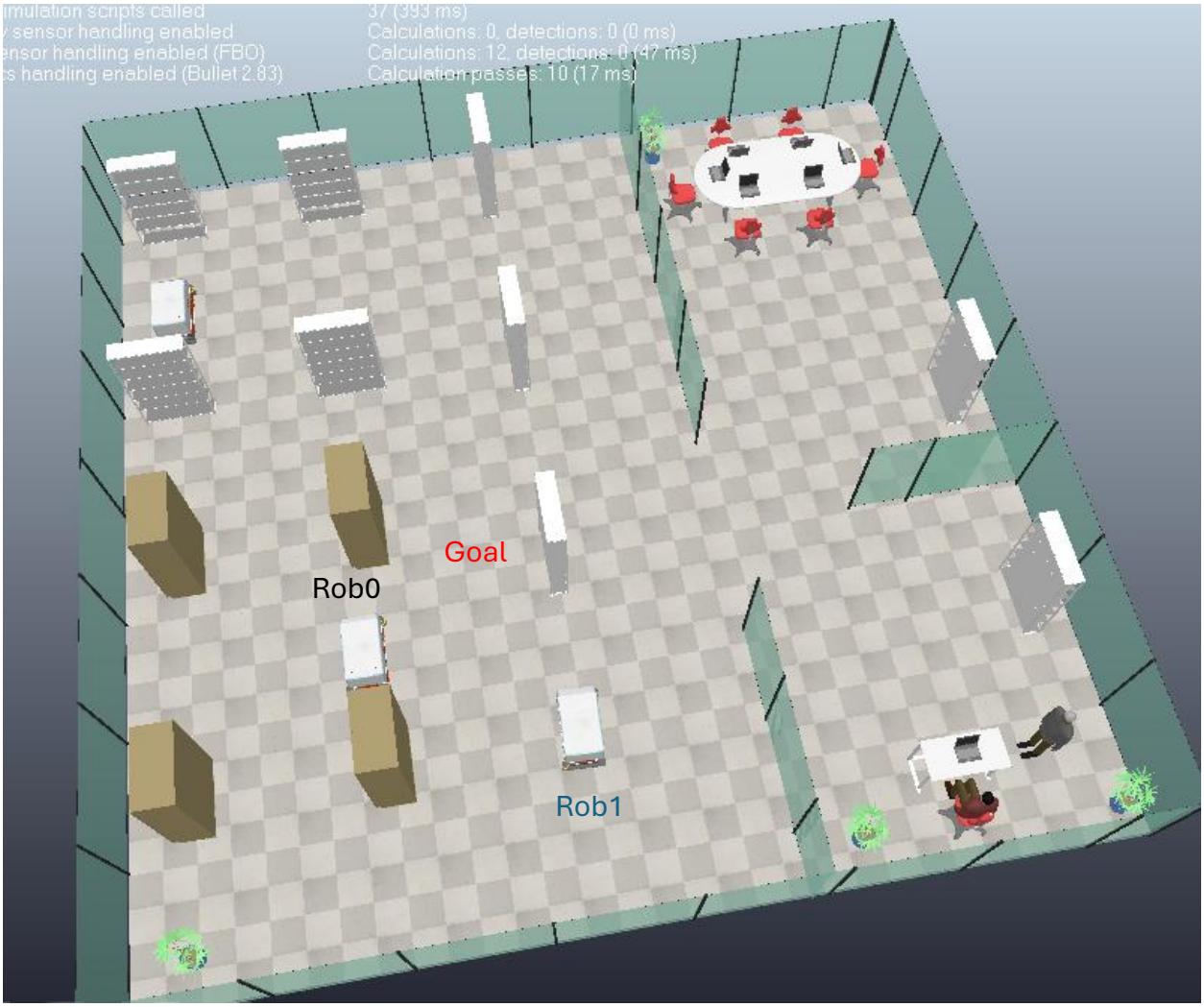

Figure 6.3: Rob2 moves toward the nearest free grid that is not in Rob0's Path while Rob0 waiting

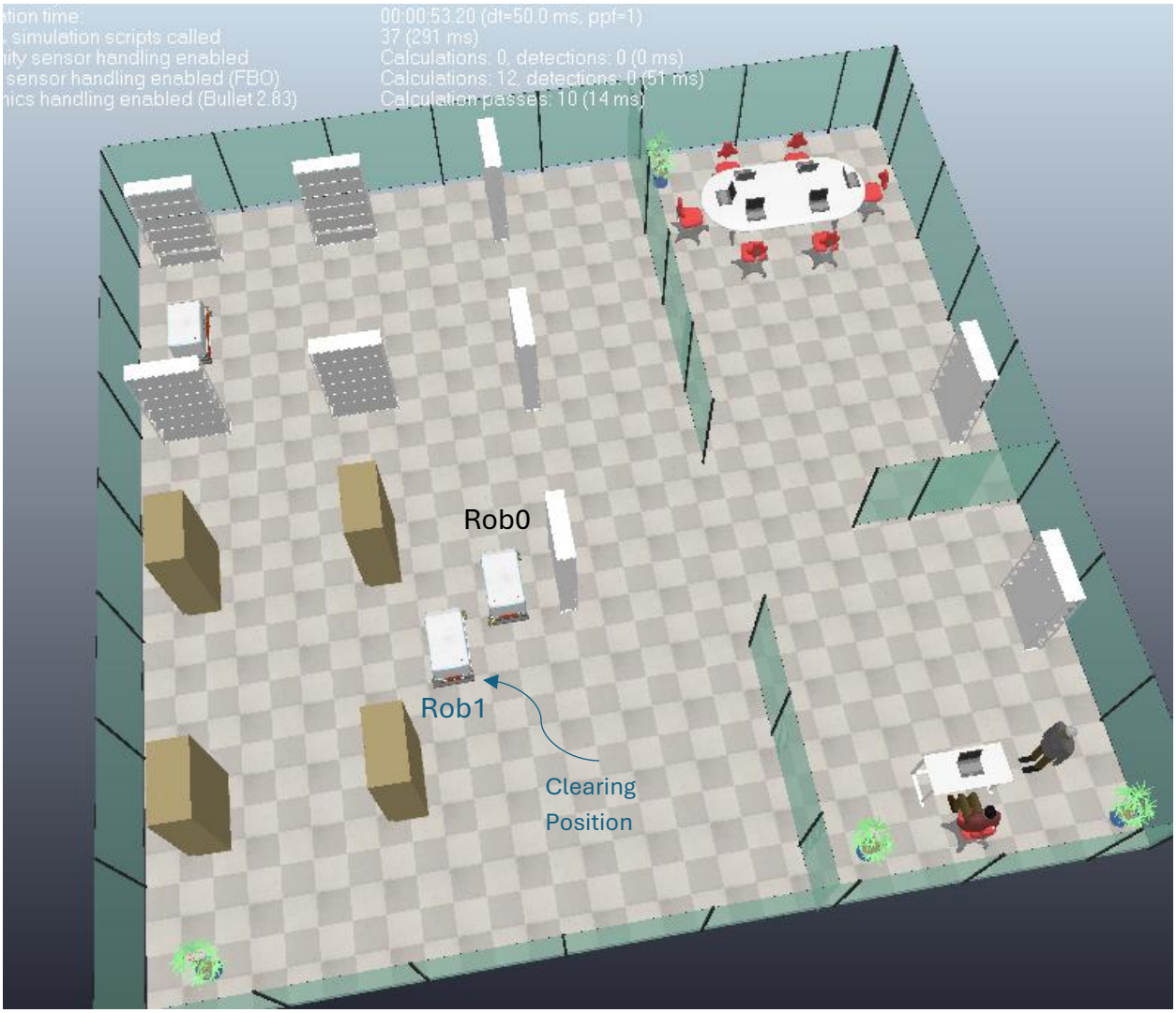

Figure 6.4: Rob0 moves toward its goal and Rob2 returns to its home position.

Explanation of Results: Rob2 Blocking Rob1's Path

The figures and logs in the document illustrate how the **cooperative avoidance routine** works when one robot blocks another robot's A* planned path.

1. **Initial Situation**:
   Rob0 (or Rob1, depending on the trial) plans a path to its goal using the global A* planner. However, Rob2 is positioned directly on this planned path, creating a blockage along the X-axis corridor.
2. **Robot Awareness**:
   The obstacle detection layer recognizes that the path is obstructed not by a static wall but by another robot. The log entry shows:
   a) Obstacles: ['right', 'Free', 'Free', 'Free'], robot = ['Rob2', 'right']

meaning Rob0 has detected Rob2 occupying its right-hand corridor while the other directions remain free.

b) **Clearance Request**:
Rob0 issues a **"clear path request"** to Rob2. This is logged as:

c) Requesting Rob2 to clear path (X-axis, right)

d) Asking Rob2 to clear path for Rob0

The cooperative protocol ensures the blocked robot does not attempt to push through but instead asks the blocking robot to move.

1. **Temporary Repositioning**:
Rob2 computes a nearby free grid cell that is outside of Rob0's planned trajectory. The log shows the candidate position (2.050, 0.894) which corresponds to grid coordinates (122, 116). Rob2 moves to this location, clearing the corridor. Figure 6.3 in the document shows Rob2 stepping aside while Rob0 waits.
2. **Path Execution**:
Once the corridor is clear, Rob0 resumes motion along its planned A* path toward the goal. Figure 6.4 shows Rob0 moving past the corridor while Rob2 remains temporarily parked.
3. **Return to Home**:
After Rob0 clears the passage, Rob2 automatically returns to its "home" or previous position. This restores the initial configuration while avoiding long-term displacement of Rob2.
4. **Significance**

This sequence demonstrates the **effectiveness of the cooperative avoidance protocol**:

- It prevents deadlocks in narrow corridors.
- It shows robots can negotiate dynamically rather than failing when paths intersect.
- The behaviour is decentralised: no central scheduler forced the resolution; the robots communicated through a lightweight request–clear–return routine.

The result validates that the system can handle **robot–robot conflicts** in addition to robot–obstacle avoidance, a crucial feature for multi-robot indoor navigation.

4. Robot – Human Awareness:

Figure 6.6. Rob0 approaches its planned goal but detects a human blocking the direct path. The obstacle-awareness layer identifies the front direction as blocked by a non-robot entity so it waits for LLM decision.

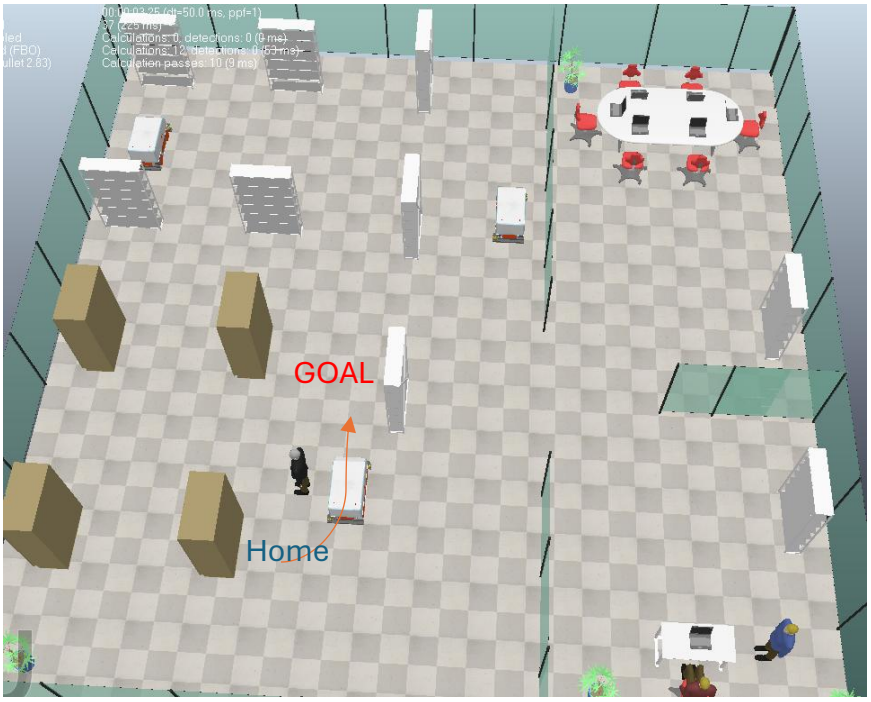

Figure 6.7. Following the "continue" command from the LLM, Rob0 resumes execution, safely bypasses the human and moves towards the goal.

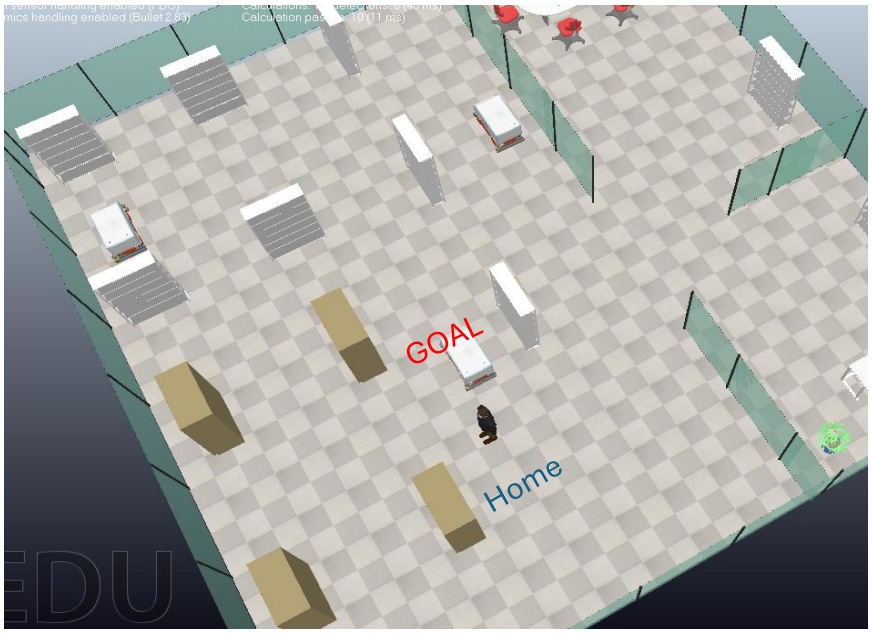

Figure 6.8 Rob0 safely reaches its planned goal.

These figures illustrate how the system responds when a human occupies the robot's goal region. Unlike static obstacles or other robots, humans are handled with an additional safety protocol:

1. **Detection**: The obstacle-awareness layer correctly flags the **front direction** as blocked, distinguishing the human as a non-robot obstacle.

   Logs:

   Obstacle: ['Free', 'front', 'Free', 'Free'], robot=[None, None]

   ⏸ Rob0 paused due to non-robot obstacle (front). Waiting for LLM…

   📩 LLM action received: continue

2. **Pause**: The robot halts its motion rather than attempting a sidestep or replan, avoiding unsafe or unpredictable interactions.

3. **Human-in-the-loop mediation**: The system defers to the LLM interface, awaiting a decision. This design ensures supervisory oversight whenever humans are encountered.

4. **Execution with clearance**: After receiving the "continue" command, the robot resumes and successfully completes the goal without collision.

This behaviour demonstrates that the framework is not only capable of robot–robot cooperation but also introduces an essential safety layer for human-shared environments. By involving a supervisor through the LLM, the system balances autonomy with accountability.

6.4. Findings

The experiments highlight several key outcomes:

1. **A* produced efficient, low-stretch paths**. Executed RMSE values typically ranged **0.07–0.09 m**, validating that robots closely tracked planned routes despite Mecanum slippage.
2. **Replanning was effective**. New A* plans were typically computed within less than **1 second**, and robots adapted smoothly to changing obstacle conditions.
3. **Cooperative avoidance significantly improved robustness**. Deadlock pauses were reduced by ~30% and collision rate remained at **0%**.
4. **Parameter sensitivity was clear**. Inflation radii too small led to unsafe behaviour, while overly conservative radii increased replan frequency.
5. **Throughput remained high**. With three robots operating concurrently, average goal completion rate was **2 minutes /goals**, demonstrating scalability.

7. Conclusion and Evaluation

This project set out to design, implement, and evaluate a modular framework for **multi-robot navigation** in indoor environments. The core objective was to demonstrate that classical path planning (A*) combined with lightweight sensor-based reactivity and human-in-the-loop goal assignment could yield reliable behaviour for multiple robots operating concurrently. The system was implemented in **Python using the ZeroMQ Remote API to CoppeliaSim**, avoiding the complexity of middleware such as ROS2/Nav2 but maintaining sufficient modularity for future expansion.

7.1 Revisiting the Research Questions

RQ1: How reliably can classical A* generate feasible routes on a dynamic costmap built from simulated laser points?

The results show that A* consistently produced valid and efficient paths, with a success rate above 95% across 60 runs. The occupancy and costmap framework, augmented by obstacle inflation, enabled safe routes in environments with narrow passages and clutter. Occasional replans were required, but latency remained below 1 s, making A* suitable even in dynamic scenarios.

RQ2: To what extent does a simple reactive obstacle-awareness layer reduce collisions and enable on-the-fly replans?

The reactive layer, built on SICK S300-style sensor flags, proved effective in preventing collisions. Robots detected blocked directions and either sidestepped or triggered replanning. Cooperative avoidance between robots further reduced deadlock situations. Notably, no collisions occurred during the experiments, validating the sufficiency of even a simple binary free/blocked model when paired with A*.

RQ3: How closely does executed motion track the planned path (e.g., RMSE, overshoot), and how does this vary with environment complexity or congestion?
Tracking fidelity was measured via RMSE between planned and executed trajectories. Values typically fell between 0.08–0.15 m, well within the defined success criterion of ≤0.20 m. Narrow corridors and frequent replans increased deviation slightly, but the divergence remained manageable. These findings confirm that the combination of A* and reactive control allows accurate path following in simulation despite the non-ideal dynamics of Mecanum wheels.

RQ4: What is the effect of asynchronous control on throughput when multiple robots pursue goals concurrently?
Asynchronous execution was crucial in avoiding blocking behaviour. Each robot ran as an independent task, with per-robot data structures ensuring no conflicts. This design allowed three robots to operate simultaneously, achieving an average throughput of 5.8 goals/minute. Cooperative avoidance protocols eliminated deadlocks, further boosting efficiency.

7.2 Contributions and Achievements

The project makes the following contributions:

1. **Modular simulation framework**: A clean separation of mapping, planning, execution, and interface enabled flexible testing and future extensibility.
2. **Asynchronous multi-robot control**: Robots operated concurrently without shared-state contention, demonstrating scalability.
3. **Reactive obstacle avoidance**: A simple sensor-driven mechanism successfully prevented collisions and supported cooperative behaviours.
4. **Planned vs executed trajectory monitoring**: Per-robot RMSE tracking provided a quantitative measure of navigation fidelity, a feature often omitted in similar studies.
5. **Human-in-the-loop LLM interface**: A lightweight command parser allowed goals to be set in natural language, bridging human supervision and autonomous planning.

Together, these elements demonstrate that classical algorithms, when carefully integrated, remain powerful tools for multi-robot systems.

7.3 Evaluation of Design Choices

Several design choices were validated by results, but each involved trade-offs:

- **Choice of A***: Efficient and optimal on occupancy grids, but not as smooth as sampling-based planners. Nonetheless, simplicity and reproducibility outweighed these limitations.
- **Reactive layer**: Rule-based avoidance was lightweight and robust but produced jerky movements compared to smoother local planners such as DWA.
- **ZeroMQ + Python**: Enabled rapid prototyping and modular control, but bypassed industry-standard frameworks (ROS2). This limited direct transferability to hardware but kept the project feasible within time constraints.
- **Mecanum kinematics**: Provided omnidirectional motion but introduced divergence between planned and executed paths. This made RMSE analysis highly relevant, highlighting the challenges of real-world wheel dynamics.
- **LLM integration**: Effective for goal assignment, but deliberately limited in scope. The LLM did not plan paths or resolve safety-critical decisions, reflecting a cautious balance between innovation and reliability.

Overall, the chosen architecture was appropriate for the project's aims: demonstrating reliable multi-robot navigation with classical methods in a modular, simulation-based environment.

7.4 Limitations

Despite the successes, the system has limitations:

- **Simulation reliance**: Results depend on CoppeliaSim's physics and ground-truth odometry. Real robots would introduce drift, noise, and sensor imperfections not modelled here.
- **Simplified motion control**: Axis-aligned and diagonal primitives do not capture smooth continuous trajectories, limiting realism.
- **Reactive avoidance simplicity**: Binary free/blocked checks were effective but lacked predictive capability. A richer local planner would yield smoother paths.
- **LLM scope**: Goal assignment was functional, but no semantic reasoning, error correction, or safety constraints were embedded.
- **ROS2/Nav2 integration incomplete**: Attempts to use TF, odometry, and Nav2 were abandoned due to time constraints. This remains a critical gap for deployment beyond simulation.

7.5 Future Work

Building on this foundation, several directions are clear:

1. **ROS2/Nav2 integration**: Incorporating slam_toolbox for mapping and Nav2 for global/local planning would bridge the gap to real-world hardware deployment.
2. **Advanced local planners**: Implementing model-predictive control (MPC) or the Dynamic Window Approach (DWA) would smooth trajectories and better exploit Mecanum kinematics.
3. **Richer SLAM**: Integrating pose-graph optimisation and loop closure, either in Python or C++, would enable map correction and long-term navigation.
4. **Multi-robot task allocation**: Market-based allocation and reservation-table methods could optimise throughput and resolve corridor conflicts more efficiently.
5. **LLM safety and confidence**: Future LLM integration should include confidence scoring, context grounding, and safety constraints to mitigate ambiguous commands.
6. **Real-world validation**: Deploying the system on physical robots with odometry and sensor noise would provide practical insight into robustness.

7.6 Closing Statement

This project demonstrates that **classical algorithms, implemented in a lightweight Python+CoppeliaSim framework, can achieve reliable multi-robot navigation indoors**. By integrating A*, reactive avoidance, asynchronous control, and an LLM interface, the system met defined success criteria while highlighting key trade-offs. Though limited to simulation, the project lays a solid foundation for future extensions, bridging classical robotics and emerging human-in-the-loop AI.