Robotics 2025

Ismael González Loro, Samuel Corrionero Fernández, José Pulido Delgado
Ingeniería Informática en Ingeniería de Computadores
extitUniversidad de Extremadura
Cáceres, Spain
Email: {igonzaleoa, scorrion, jopulidod}@alumnos.unex.es

https://github.com/CoferSamuel/ODFE_robotica

***Abstract*—This paper is a summary of the Robotics 2025 subject. It presents theory lessons given in class and the development and implementation of a reactive control system for the OmniRobot platform. The system was implemented using a Component-Based Software Engineering (CBSE) paradigm, leveraging the open-source RoboComp framework. The architecture, written in C++, includes a Qt-based graphical user interface for high-level control and monitoring. The system's effectiveness was validated through a series of tasks, e.g., "Sweeping a room" and "Coordinate system based on global coordinates". The results demonstrate that the CBSE approach significantly simplified development and debugging, leading to a robust and maintainable robotic system. This paper provides a practical validation of the RoboComp framework for building complex, reactive applications.**

***Index Terms*—Mobile robotics, Component-Based Software Engineering, RoboComp, Webots, C++, Simulation, Reactive systems**

I. Introduction

The field of mobile robotics studies machines capable of autonomous motion through dynamic environments. According to Corke [1], mobile robots range from simple reactive platforms such as Elsie the tortoise to complex deliberative systems like Shakey, the first robot capable of reasoning about its environment. These contrasting paradigms—reactive and deliberative—have shaped modern robot control architectures [2], [3]. Murphy [11] highlights this dichotomy as one of the defining distinctions between artificial intelligence (AI) and engineering approaches to robotics: the former focuses on adaptability and autonomy, while the latter emphasizes precision and repeatability.

Historically, robotics evolved along two separate trajectories: industrial manipulators and AI-driven autonomous systems. Industrial robots prioritized accuracy and control theory for repetitive tasks in structured environments, while AI robotics pursued intelligent perception and decision-making for unstructured contexts such as space exploration [11]. The convergence of these paths—industrial precision with AI adaptability—has become the foundation of modern mobile robotic design.

In recent years, the convergence of Component-Based Software Engineering (CBSE) and robotic simulation has transformed how robotic systems are developed and tested. Frameworks like RoboComp [4] allow developers to design modular components that encapsulate specific functionalities such as perception, actuation, or control. This modularity accelerates development, enhances code reuse, and facilitates integration between heterogeneous systems. As Murphy [11] observes, this modular approach echoes the AI field's long-standing goal of creating autonomous agents composed of interacting intelligent subsystems (e.g., perception, planning, reasoning, and learning).

The project described in this paper was carried out within an academic environment aimed at providing students with practical experience in mobile robot software engineering. Using the RoboComp framework, a reactive control system was implemented for the OmniRobot platform, a simulated differential-drive robot in Webots. The robot is equipped with sensors (e.g., LiDAR) and actuators managed through interconnected software components. This aligns with Murphy's concept of *teleoperation transitioning to autonomy*, where systems evolve from human-supervised control toward self-governing behavior through improved perception and decision mechanisms.

Following the principles discussed by Corke [1] and Murphy [11], this project adopts a reactive navigation approach, where the robot continuously perceives and responds to its surroundings. The main objective is to demonstrate the effectiveness of CBSE methodologies in building modular, maintainable, and extensible robotic systems. The rest of this paper describes the technical setup, tools, and methodologies used to achieve these goals.

A. Types of Robots

Robots can be broadly categorized based on their intended environment and method of locomotion. While hybrid designs exist, this subsection outlines four primary types that form the basis for most robotic systems.

1) Mobile Robots (Wheeled): These robots navigate the ground using wheels, as seen in Fig. 1. They are the most common type in industrial and domestic settings due to their high energy efficiency, stability, and mechanical simplicity. They are foundational to logistics and manufacturing, excelling on flat or semi-structured surfaces for tasks like material transport. Their primary limitation is a reliance on prepared environments, as they struggle to traverse unstructured terrain, stairs, or significant obstacles.

- **Representative Example:** Warehouse Robot (e.g., Amazon Kiva [13]).

Fig. 1: A typical wheeled mobile robot [13].

2) Aerial Robots (Drones): Commonly known as drones, these robots operate in the air, typically using a multi-rotor configuration (like the quadcopter in Fig. 2) for high maneuverability and vertical takeoff. Their key advantage is the ability to access a three-dimensional workspace, providing an aerial perspective for tasks like infrastructure inspection, precision agriculture, search and rescue, and cinematography. Key engineering challenges for this class include limited flight time due to battery constraints, payload capacity, and navigating complex, often regulated, airspace.

- **Representative Example:** Quadcopter (e.g., DJI Phantom [14]).

Fig. 2: A quadcopter, a common type of aerial robot [14].

3) Legged Robots: These robots use articulated limbs (legs) for movement, mimicking biological locomotion. This design, seen in quadrupeds (Fig. 3) or bipeds, allows them to traverse complex, unstructured, and human-centric environments (like stairs) that are inaccessible to wheels. They offer high mobility and adaptability by stepping over obstacles and adjusting their gait. Primary applications include remote inspection in hazardous areas, disaster response, and research. However, they face significant challenges in dynamic stability, control complexity, and lower energy efficiency compared to wheeled systems.

- **Representative Example:** Boston Dynamics' Spot [15].

Fig. 3: The 'Spot' quadruped legged robot [15].

4) Aquatic Robots: These robots are designed for operation in or on water (Fig. 4). They are broadly classified into Autonomous Underwater Vehicles (AUVs), which are untethered and perform pre-programmed missions, and Remotely Operated Vehicles (ROVs), which are tethered to a human operator for real-time control. Applications include oceanographic research, seafloor mapping, offshore infrastructure maintenance, and underwater archaeology. Key challenges involve waterproofing, resistance to high pressure, and the need for alternative navigation and communication systems (e.g., acoustic modems), as GPS and radio waves are ineffective underwater.

- **Representative Example:** Underwater ROV [16].

Fig. 4: An aquatic robot (ROV) for underwater tasks [16].

II. Materials and Methods

A. Component-Based Software Engineering (CBSE)

Component-Based Software Engineering (CBSE) is a paradigm that promotes the development of software systems through the integration of reusable, self-contained components. Each component exposes defined interfaces and encapsulates its internal functionality [5]. In robotics, CBSE has become essential due to the inherent complexity and multidisciplinary nature of robotic systems [6].

RoboComp is an open-source framework designed to support CBSE in robotics. It provides communication middleware based on Ice (Internet Communications Engine), enabling distributed components to interact seamlessly [4]. This architecture promotes scalability, allowing different robotic subsystems (e.g., navigation, perception, planning) to run independently while sharing data through well-defined interfaces. Such modularity reduces integration costs and facilitates code reuse across different robotic projects.

The CBSE paradigm also aligns with Model-Driven Engineering (MDE) practices, emphasizing the use of high-level abstractions for system design and implementation. In educational contexts, CBSE provides a valuable learning platform, enabling students to experiment with distributed architectures and system integration within a controlled simulation environment.

B. Development Environment and Technical Resources

The development process was carried out in a simulated environment using Webots [7], a professional mobile robot simulator that allows realistic testing of control algorithms before deployment. The programming languages used were C++ [8], with build automation handled by CMake [10]. Source code management was performed using GitHub, fostering collaborative development and version control [9].

The use of simulation and modular software engineering reflects the pedagogical goals discussed by Murphy [11], who emphasized the importance of bridging teleoperation principles with autonomy in educational robotics. Students are encouraged to iterate between manual control and autonomous modes, observing how layered control architectures respond to dynamic environments—paralleling the historical evolution from telemanipulators to intelligent mobile robots.

- **Operating system:** The project was developed on Linux, specifically Ubuntu-based distributions, due to their strong support for open-source robotics software. Linux provides native compatibility with essential libraries such as ROS, RoboComp, and CMake. Its terminal-based workflow facilitates dependency management and compilation processes. Furthermore, Linux's permission and process control systems enhance robustness and security in real-time applications [12].
- **Class PC (Sala Beta):** Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz; Kingston 16GB $x2$ (lab machine used for in-class development and tests).
- **Programming language:** C++23. C++ is a general-purpose language designed by Bjarne Stroustrup as an enhancement of the C language. It supports multiple paradigms (procedural, object-oriented, and generic programming), offering both high-level abstraction and low-level hardware control [8]. Over time, C++ has evolved through international standardization, maintaining backward compatibility while introducing features that enhance expressiveness and safety. Its large community ensures extensive documentation and continuous innovation. In robotics, C++ is preferred due to its efficiency, strong typing, and deterministic performance in real-time systems.
- **Framework:** Qt6 was used to develop graphical user interfaces (GUIs) and handle event-driven programming. Qt6 offers cross-platform support, an extensive widget library, and native integration with CMake, facilitating visualization and control panels for robotics. In this project, Qt6 supported the interface components of RoboComp and simplified interaction with robot control modules.
- **Version control platform:** GitHub was used for distributed version control and team collaboration. Through branching and merging mechanisms, it allowed contributors to develop features concurrently while maintaining synchronization. Continuous Integration (CI) workflows were configured to automate compilation and testing. GitHub also supports code reviews and issue tracking, facilitating agile and traceable development practices [9].
- **Build automation system:** CMake was used as the build automation and configuration tool. CMake simplifies the process of building multi-platform projects by generating native build systems such as Makefiles. It also allows dependency management through configuration files (`CMakeLists.txt`). In robotics, CMake's modularity aligns with CBSE principles, enabling separate component compilation and linking [10].
- **Integrated development environment:** Visual Studio Code (VSCode) and CLion were used. VSCode is a lightweight, cross-platform IDE developed by Microsoft, offering extensions for C++, Git integration, and debugging tools. Its open-source nature and extensibility make it ideal for collaborative robotics development. CLion, developed by JetBrains, provides intelligent code analysis, CMake integration, and refactoring tools. It enables in-depth static code analysis and real-time syntax inspection, which reduce debugging time and improve code quality. The IDEs enhanced productivity by providing syntax highlighting, linting, and build management integrated with RoboComp's component structure.
- **Simulation software:** Webots is a professional-grade robot simulator that allows the modeling and control of robots in a 3D virtual environment [7]. It supports sensors, actuators, and physics simulation, enabling rapid prototyping without the need for physical robots. Webots provides interfaces for C++, Python, and ROS, making it suitable for educational and research applications. In

this project, Webots was used to simulate the OmniRobot platform and its environment, providing realistic testing conditions for perception and control algorithms.

- **Robotic components utilized:**
  - OmniRobot: The sweeping robot, named OmniRobot, is modeled as a circular platform equipped with distributed sensors. It features a minimalist white design and a tall structure supported by four small omnidirectional wheels capable of linear and rotational motion. At the top of the robot, there is a tray to transport objects. It was provided by the professor and developed by the RoboComp team.
  - Lidar3D: This component simulates a LiDAR sensor capable of measuring both the distance and angular position of nearby obstacles. It provides 360° environmental awareness, essential for obstacle avoidance and mapping. It was provided by the professor and developed by the RoboComp team.
  - Chocachoca: Software responsible for controlling OmniRobot's motion behavior, acting as the robot's decision-making brain. It defines navigation and collision-avoidance behaviors based on sensor inputs. This is the only component created by the authors, following the professor's guidelines.
  - Webots-bridge: Serves as the intermediary communication layer between OmniRobot, Lidar3D, and Chocachoca, allowing seamless data exchange within the simulation environment.
  - JoystickPublish: Connects a real joystick to OmniRobot, enabling manual control during testing phases. It was provided by the professor and developed by the RoboComp team.
  - Aspirator: A program with a GUI that evaluates the efficiency of the robot in terms of surface coverage and time employed.
  - rcnode: This tool provides a publish-subscribe messaging infrastructure for RoboComp components to communicate asynchronously. Components can publish messages to topics and subscribe to topics of interest, enabling decoupled, event-driven communication in robotic systems.

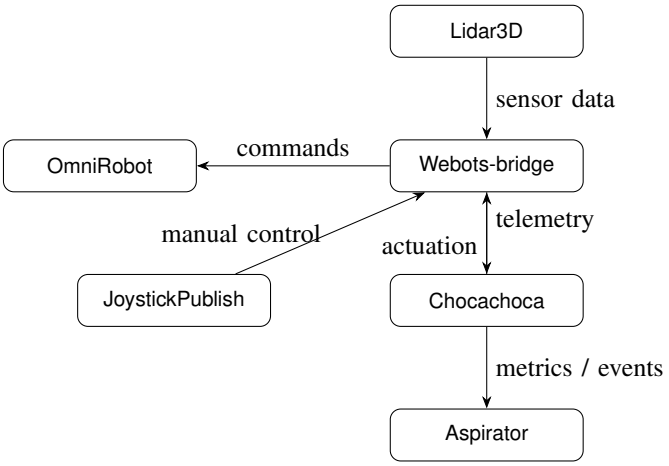


Fig. 5: Component diagram

- **Documentation tool:** Overleaf was used for collaborative documentation and paper writing. Overleaf is a cloud-based LaTeX editor that simplifies academic and technical writing through real-time synchronization, version history, and multi-user editing. Its compatibility with IEEE templates facilitates the production of professional papers and ensures reproducible document formatting.

III. Activity 1: Sweeping Robot

A. Objective

The main objective of this activity is to design and implement a program (Chocachoca) that guides an autonomous robot capable of covering the largest possible area within a confined space in the shortest time. The system relies on sensor feedback (Lidar3D) to detect obstacles and dynamically adjust its trajectory to achieve efficient coverage while avoiding collisions. Performance is evaluated in a test lasting up to 3 minutes using the previously described Aspirator component.

B. Graphical User Interface (GUI)

The GUI provides real-time visualization of the robot's perception and localization state, enabling monitoring and debugging of the control system during operation.

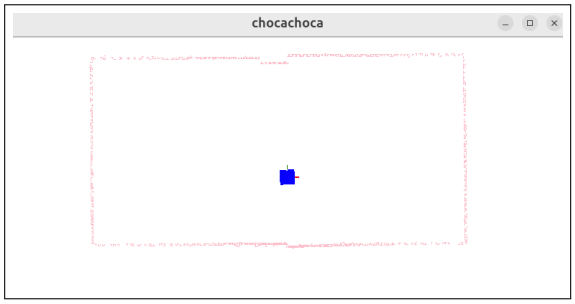


Fig. 6: Real-time GUI showing LiDAR point cloud visualization and robot localization.

The interface displays two key components:

- **LiDAR Point Cloud Visualization:** The pink/red dotted outline represents the filtered LiDAR 3D point cloud, which shows the detected obstacles and environment geometry in real-time. The raw sensor data undergoes two filtering stages to reduce noise and improve computational efficiency: (1) angle-based grouping reduces the point cloud by selecting, for each azimuth direction, only the closest detected point; and (2) isolation filtering removes sparse, noise-prone points that lack neighboring detections within a 200 mm radius. This two-stage filtering preserves the geometric structure of the environment while eliminating spurious measurements. The maximum sensing range is limited to 12 meters to balance perception coverage with real-time processing requirements.
- **Robot Representation:** The blue square in the center of the visualization represents the robot's current position and orientation within the environment. Its location is updated continuously from the robot's odometry system, which tracks the robot's x and z Cartesian coordinates (in millimeters) and heading angle alpha (in radians). This visualization provides immediate feedback on the robot's self-reported localization state, enabling real-time monitoring of whether the robot is executing the intended trajectory and responding correctly to obstacle avoidance commands.

C. Operational Algorithms and Behavioral States

As previously explained, the robot's navigation behavior is defined by a finite set of operational states that govern its decision-making process and movement patterns, as shown in Figure 7.

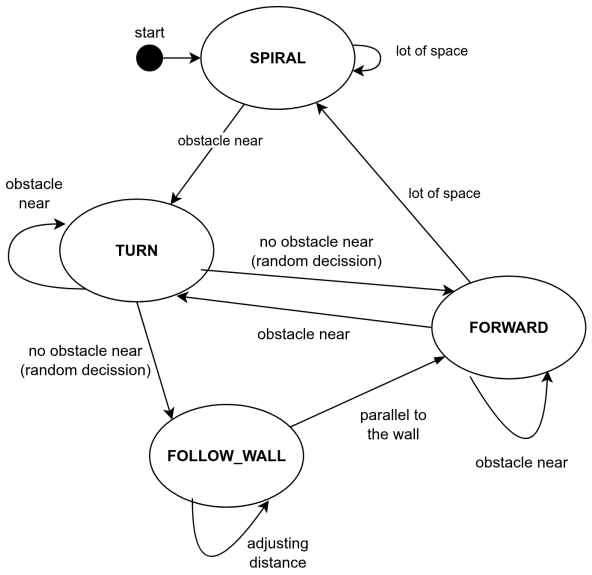


Fig. 7: State machine for the Chocachoca controller.

The initial state is *SPIRAL*, in which the robot performs a spiral movement until it approaches an obstacle. It then transitions to *TURN* to avoid the obstacle. After turning, the robot randomly switches to either *FORWARD* or *FOLLOW_WALL*.

If it switches to *FORWARD*, it will continue moving straight until it encounters another obstacle, at which point it transitions back to *TURN*. Otherwise, it changes to *FOLLOW_WALL*, where the robot moves parallel to the wall.

All states are detailed below:

TABLE I: Variable and Constant Legend

Variable	Description
D_T	Min. distance to obstacle to trigger *TURN*.
v_s	Robot's forward speed during *SPIRAL*.
ω_s	Robot's rotational speed during *SPIRAL*.
$V_{s,max}$	Max. allowed forward speed in *SPIRAL*.
ω_{min}	Min. allowed rotational speed (0 rad/s).
V_{max}	Max. general forward speed (*FORWARD*/*TURN*).
c	Counter for "stuck" iterations in *TURN*.
d_{min}	Data point for the closest detected obstacle.
D_S	Distance threshold to switch to *SPIRAL*.
v	Calculated forward speed in *FORWARD*.
d	Specific distance value of the closest obstacle.
T_{min}	Min. safety threshold for speed calculations.
v_b	Calculated braking speed.
B_{max}	Max. allowed braking value.
ω	Calculated rotational speed (*TURN*/*FORWARD*).

1) SPIRAL: Executes an Archimedean spiral trajectory to maximize surface coverage, gradually increasing the radius of rotation while avoiding overlaps in path. The transition between spiral and turn states is determined by the following conditions:

- 1.1. Check for nearby obstacles: if any obstacle is within D_T, abort spiral and change to *TURN*.

$$D_T = 600.0 \text{ mm} \tag{1}$$

- 1.2. If no obstacles, slowly increase advance speed (v_s) and reduce rotation (ω_s) to expand the spiral.

$$v_s = 500.0 \text{ mm/s} \tag{2}$$
$$\omega_s = 3.0 \text{ rad/s} \tag{3}$$

These values change as follows:

$$v_s \mathrel{+}= 2.5 \text{ mm/s} \tag{4}$$
$$\omega_s \mathrel{-}= 0.01 \text{ rad/s} \tag{5}$$

Respecting the thresholds:

$$V_{s,max} = 1000.0 \text{ mm/s} \tag{6}$$
$$\omega_{min} = 0.0 \text{ rad} \tag{7}$$

2) TURN: Rotate the robot in place to avoid obstacles that are blocking its path. The robot keeps turning until the frontal area is clear, then decides whether to go forward or follow a wall.

- 1. Filter frontal points
  - Select only points in $\pm 45°$ frontal cone
  - Determines if path ahead is clear
- 2. Find nearest frontal obstacle

  - Find the closest point in front of the robot
- 3. Decision tree: Clear path vs Still blocked
  - Path A: Obstacle cleared
    * Check if nearest obstacle > 700 mm
    * Randomly set rotation speed:
      - 0.6 rad/s with a 70%
      - 3.0 rad/s with a 30%
    * Randomly choose next state: *FOLLOW_WALL* or *FORWARD* with V_{max} speed.

$$V_{max} = 800 \text{ mm/s} \tag{8}$$

    * Randomness avoids deterministic loops
  - Path B: Still blocked
    * Increment c. This variable measure the amount of times that the robot executes *TURN* in the Path B.
    * If $c \geq 100$, force positive rotation (escape hatch)
    * Otherwise, choose rotation direction based on obstacle side.
    * This path is usually seen in the corners and this approach achieves to escape from them when stuck.

3) FORWARD: Commands linear motion until an obstacle is detected by the front sensors.

- 1. Collect frontal points: keep points with ϕ between $-45°$ and $+45°$.
- 2. Find the closest frontal point.
- 3.1. If $d_{min} > D_S$ (very far):

$$D_S = 2000 \text{ mm} \tag{9}$$

  - Switch to *SPIRAL* state.
- 3.2. Else: d_{min} is closer
  - 3.2.1. If the distance to d_{min} from the robot is greater than D_T (1), keep *FORWARD* with a new value of v and zero rotation.
    v will increase until the distance to the point is equal or smaller than D_T (1). It has to increase gradually, because the virtual robot is simulating a real robot with real physics.

$$v = (d_{min} - T_{min}) \cdot \frac{V_{max}}{D_T - T_{min}} \tag{10}$$

  - 3.2.2. Else, transition to *TURN* with linear 0 and a new value of rotation ω. Compute brake term v_b proportional to how much closer than D_T (1) we are:

$$v_b = (D_T - d_{min}) \cdot \frac{B_{max}}{D_T - T_{min}} \tag{11}$$

$$\text{Let } x := D_T - d_{min}, \qquad y := v_b$$

$$y = \frac{B_{max}}{D_T - T_{min}} x \;=\; m\,x + n \tag{12}$$

    Where:
    * $m = \dfrac{B_{max}}{D_T - T_{min}}$ (slope / proportional gain),
    * $n = 0$ (no intercept when using $x = D_T - d_{min}$).

    The first line (Eq. 11) shows the original brake expression. Using $x = D_T - d_{min}$ yields the linear form (Eq. 12). Here x is the distance-to-threshold scalar and y is the brake term v_b (both in consistent units). These are control variables, not spatial robot coordinates.

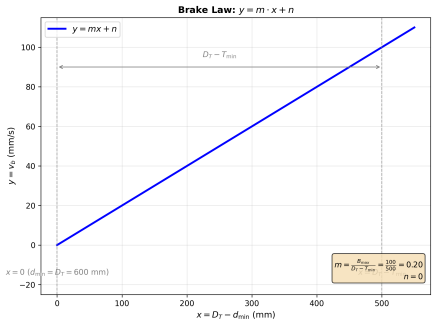


Fig. 8: Brake law: linear relation $y = mx + n$ where $x = D_T - d_{min}$, $y = v_b$, $m = B_{max}/(D_T - T_{min})$, and $n = 0$. Thresholds at $x = 0$ (obstacle trigger $d_{min} = D_T = 600$ mm) and $x = D_T - T_{min}$ (safety threshold) are shown as dashed gray lines.

Finally, assign the angular speed:

$$\omega = v_b/2 \tag{13}$$

The /2 is just a tuning factor to make the rotation speed reasonable. Without it, the robot might spin too fast.

4) FOLLOW_WALL: Implements wall-following using proportional control to maintain a constant distance from the wall.

- 1. Compute the absolute closest point among all points (not only frontal).
  - d_{min} = min_element(points, by p.r); this is interpreted as the wall (closest obstacle).
- 2. Handle cases depending on the distance to the closest point: d_{min}:
  - If $D_T \leq r < D_T + 150$:
    * Continue in `FOLLOW_WALL` with `adv = 150.0f` and `rot = 0.0f` (move forward slowly).
    * This is the "ideal following band".
  - If $r > D_T + 150$ (too far from the wall):
    * If the angle to d_{min} $\phi < 0$ (wall to the left): continue in `FOLLOW_WALL` with speed of `150 mm/s` and angular speed of `-0.2 rad/s` (small rotation toward the wall; negative rotation means one direction).
    * Else (wall to the right): continue in *FOLLOW_WALL* with speed of `150 mm/s` and angular speed of `+0.2 rad/s` (small rotation to the other side).
  - If $D_T - 100 < r < D_T + 10$ (too close to the wall):

    - If $\phi < 0$ (wall is left and near): continue in `FOLLOW_WALL` with speed of `150 mm/s` and angular speed of `+0.2 rad/s` (steer away).
    - Else steer away the other side, with same linear speed and the same value of angular speed but with negative rotation.
  - Else (no special condition matched):
    - Fall back to *FORWARD* with linear speed of 600.0 mm/s (fast forward).
- 3. The function attempts to maintain the robot in a narrow corridor around D_T (1).

These are all the detailed explanations of the states. State transitions are triggered based on sensor input and internal logic thresholds, ensuring adaptive behavior in dynamic environments. The modular implementation enables easy adjustment of thresholds and parameters for different testing scenarios.

D. Experimental Results

The robot was evaluated in multiple simulated environments in Webots. These environments were provided by Robocomp and are used to test the robot in different contexts. The performance metrics focused on surface coverage efficiency.

- **Test 1**
  This is a simple map with no obstacles, just 4 walls composing a rectangle room (Figure 9). After several attempts, the best result achieved was 70 % of the room covered.

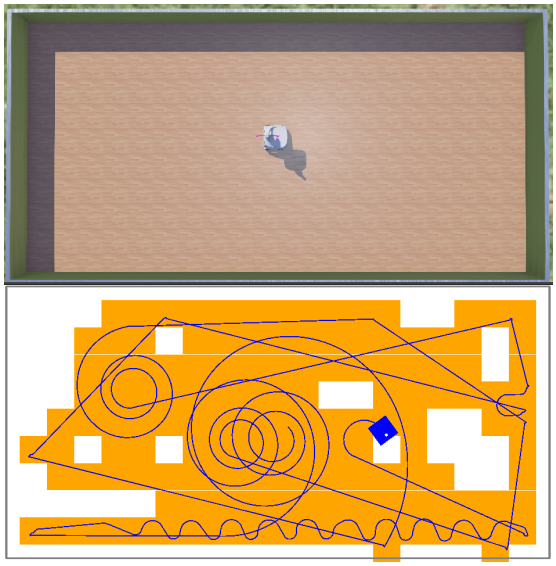


Fig. 9: SimpleWorld map and results

- **Test 2**
  This map is more complex than the previous one. A box is added, increasing the difficulty for the robot to cover the entire floor (Figure 10). The best result accomplished was the 67 % of the floor covered, worse than the previous one (Figure 10).

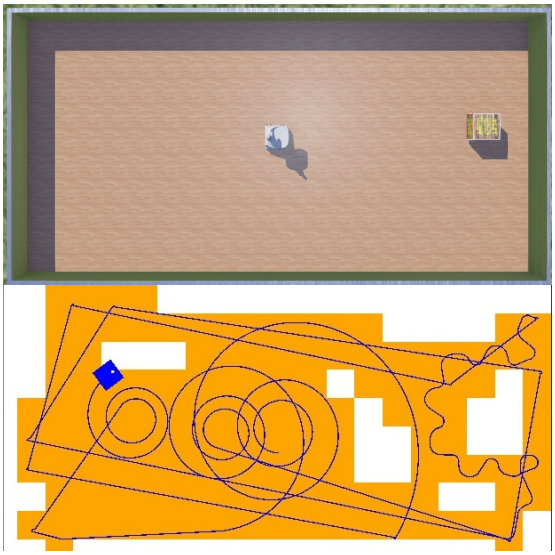


Fig. 10: SimpleWorld map with one box and results

- **Test 3**
  The increasing difficulty continues and now the room has two boxes (Figure 11). In this occassion, the best percentage achieved was 59 % (Figure 11). As can be observed, the robot performs a bit worse.

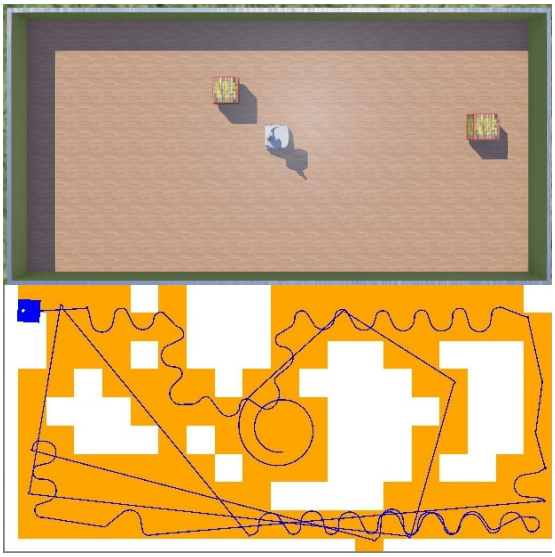


Fig. 11: SimpleWorld map with two boxes and results

E. Conclusions

As shown in the tests, the robot was unable to achieve complete floor coverage in any trial. The best result was obtained in Test 1, where there were no obstacles, while the worst performance occurred in Test 3, which has more obstacles.

The conclusions drawn from this activity are that the robot, with the Chocachoca software, struggles to cover complex rooms with obstacles, and performs better in simpler environments with unobstructed layouts.

IV. Activity 2: Robot Self-Localization in the Room

A. Objective

The primary objective of Activity 2 is to design and implement a geometry-based Localiser for the OmniRobot. This component enables the robot to estimate and incrementally correct its pose within a known rectangular room, effectively compensating for odometry drift.

The system operates by detecting salient corners from 2D LiDAR scans and associating them with the room's nominal corners using the Hungarian assignment algorithm. Based on these correspondences, it computes small incremental translation and rotation corrections by solving a linearized least-squares system.

Integrated into the RoboComp architecture, the Localiser operates in two distinct modes:

- **Estimation-only:** The system runs with actuators disabled to evaluate localization accuracy in isolation.
- **Closed-loop:** The system provides corrected pose estimates to the reactive controller developed in Activity 1, enabling more accurate navigation.

Quantitative evaluation is conducted in the Webots simulation environment through multiple randomized trials. To assess system performance, we measure the accuracy of the estimated pose (deviation of positional and angular errors), the speed of correction and the robustness of the algorithm.

Figure 12 visualizes the localization problem. The lines represent where the robot expects the walls to be, while the dots show where the LiDAR sensor actually detects them. The misalignment between the two indicates that the robot's estimated position is incorrect due to accumulated movement errors. The Localiser works to align these two views, correcting the robot's position.

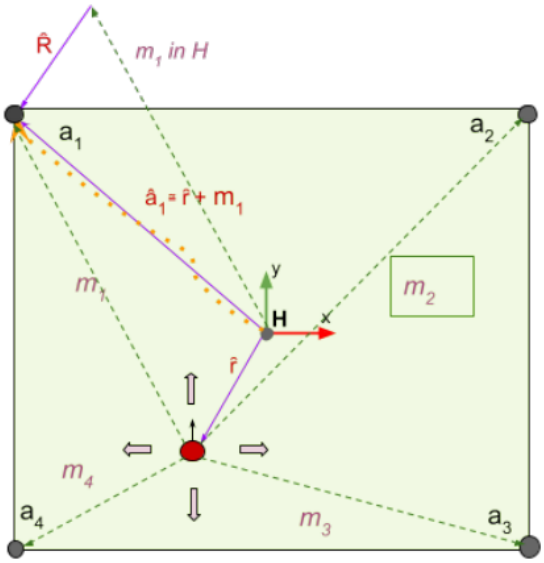


Fig. 12: Visualization of the localization problem.

B. Proposed Solution and Methodology

1) Proposed Solution: The Localiser is a modular system that corrects the robot's position using 2D LiDAR data. In every control cycle, it calculates how much the robot has drifted from its expected position in the room and applies a correction.

The process follows these steps:

1) **Preprocess scans:** We filter the LiDAR data to keep only the closest points and remove noise. This ensures we work with clean data.
2) **Extract lines (RANSAC):** We use the RANSAC algorithm to find straight lines in the point cloud. These lines correspond to the walls of the room.

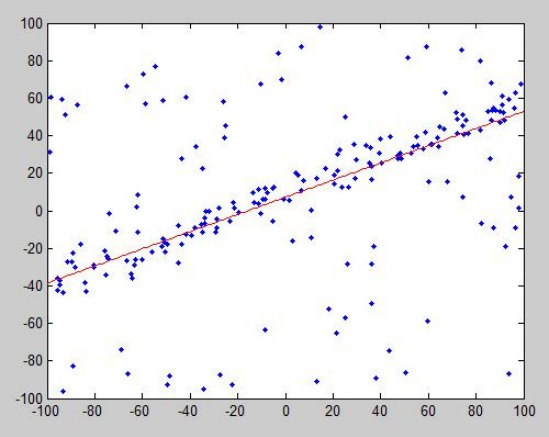


Fig. 13: RANSAC Line Extraction

3) **Find corners:** We look for intersections between perpendicular lines to find the corners of the room. We verify these corners and remove duplicates.
4) **Transform model corners:** We need to compare the known map of the room with what the robot is currently seeing. Since the map is fixed and the robot is moving, we mathematically transform the map's corners to the robot's point of view.
   If the robot is at position T and rotated by θ, we calculate the local position of a map corner C_{map} as:

$$C_{local} = R(\theta)^T (C_{map} - T)$$

   This formula subtracts the robot's position and rotates the point, allowing us to directly match the map against the LiDAR data.
5) **Match corners (Hungarian):** We use the Hungarian algorithm to match the detected corners with the known map corners based on distance. This tells us which detected corner corresponds to which map corner.

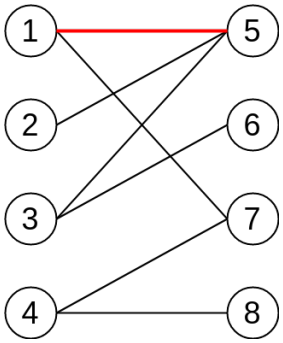


Fig. 14: Hungarian Matching

6) **Optional consensus filter:** To avoid errors, we can use a secondary check to remove matches that don't fit well with the others.
7) **Compute and apply pose update:** Using the matched corners, we calculate the exact error in the robot's position (x, y, and angle) and apply a correction to fix it.

2) Measurement Protocol and Metrics: **Experimental Setup:**

- **Environments:** We use the same Webots simulation environments as in Activity 1.
- **Trials:** We run at least 10 randomized trials for each scenario. In each trial, the robot starts at a random position.
- **Modes:** We test the system in two modes:
  - *Estimation-only:* The robot does not move, to test localization accuracy.
  - *Closed-loop:* The robot moves using the corrected position, to test navigation performance.

Data Logging: We record key data at every step to analyze performance, including the true position, estimated position, number of detected corners, and calculation time.

Primary Metrics:

- **Positional Error:** The distance between the estimated position and the true position.
- **Angular Error:** The difference between the estimated angle and the true angle.
- **Convergence Time:** How long it takes for the robot to find its correct position (error < 0.5 m).
- **Valid-solve Rate:** How often the system successfully calculates a position correction.
- **Solver Latency:** The time it takes to compute the solution.

C. Mathematical Techniques

1) Rigid Transforms in SE(2): To compare the map with the sensor data, we need to express them in the same coordinate system. A point C in the world frame can be transformed to the robot's frame using the robot's position T and rotation matrix $R(\theta)$:

$$\tilde{C} = R(\theta)^\top (C - T)$$

This equation essentially "moves" the map points so they are relative to the robot.

2) Small-Angle Linearization: The relationship between the robot's position and the sensor measurements is non-linear (due to sine and cosine functions), which makes it hard to solve directly. However, since we correct the position frequently, the errors are small. This allows us to use a mathematical trick called linearization ($\cos\Delta\theta \approx 1, \sin\Delta\theta \approx \Delta\theta$) to turn the problem into a system of linear equations:

$$\tilde{C}_x \approx T_x + m_x - \Delta\theta\, m_y$$
$$\tilde{C}_y \approx T_y + m_y + \Delta\theta\, m_x$$

Now we can solve for the corrections using standard linear algebra.

3) Solving the Linear System: We stack the equations from all matched corners into a large system $Wr = b$. To find the best correction r (change in x, y, and angle), we use the Singular Value Decomposition (SVD) method. SVD is robust and helps us find the optimal solution that minimizes the overall error between the expected and actual corner positions.

4) Robust Line Extraction (RANSAC): RANSAC is an algorithm used to find lines in noisy data. It works by:

1) Randomly picking two points to define a line.
2) Counting how many other points fit this line (inliers).
3) Repeating this many times and keeping the line with the most inliers.

This ensures we find the actual walls and ignore scattered noise points.

5) Corner Detection: Corners are found by intersecting the lines detected by RANSAC. An intersection is considered a valid corner if:

- The lines are perpendicular (forming a 90-degree angle).
- The intersection point actually lies within the wall segments.

We also filter out duplicate corners to ensure a clean set of features.

6) Data Association (Hungarian Algorithm): Once we have the corners from the map and the corners from the LiDAR, we need to match them. We use the Hungarian algorithm, which finds the optimal pairing that minimizes the total distance between matched corners. This ensures we are comparing the right corner in the map to the right corner in the sensor data.

7) Outlier Rejection: Even with good matching, mistakes can happen. We use a consensus filter (similar to RANSAC) to double-check the matches. It looks for a group of matches that all agree on the same robot position and discards any "outliers" that don't fit this consensus. This prevents a single bad match from ruining the position correction.

8) Outlier Rejection: Even with good matching, mistakes can happen. We use a consensus filter (similar to RANSAC) to double-check the matches. It looks for a group of matches that all agree on the same robot position and discards any "outliers" that don't fit this consensus. This prevents a single bad match from ruining the position correction.

D. Experimental Results

The robot was evaluated in multiple simulated environments in Webots provided by RoboComp.

Test Case 1: SimpleWorld The first test environment (Figure 15) consists of a simple rectangular room with no obstacles.

Figure 16 presents the results of the localization process. The image shows two perspectives:

- On the left, the **robot's internal view**, where the detected corners (from LiDAR) are successfully matched with the map corners. The visual alignment confirms that the algorithm has correctly estimated the robot's pose.
- On the right, the **external simulation view**, showing the actual position of the robot in the room.

The successful matching of corners in the internal view corresponds to the accurate positioning seen in the external view, validating the system's performance.

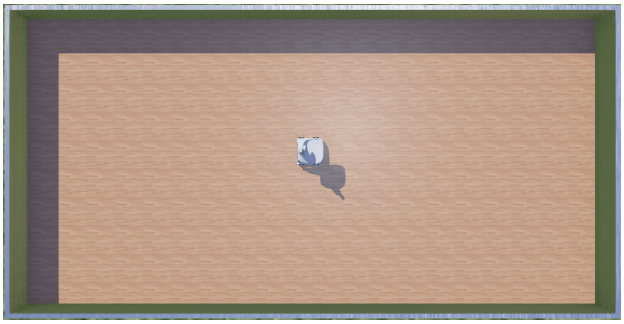


Fig. 15: SimpleWorld Map

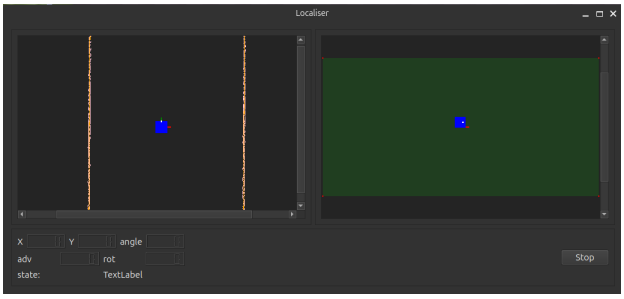


Fig. 16: SimpleWorld Test Results

A video demonstration of the localiser in action can be viewed at:

https://drive.google.com/file/d/1f_QV6La_op_kgW51XC4HKma_aQV0oxyi/view?usp=sharing

E. Conclusions

This activity successfully integrates a geometry-based Localiser into the RoboComp architecture. The developed pipeline (using RANSAC, Hungarian algorithm, and SVD) effectively corrects the robot's position in rectangular environments.

While the system performs well in simple scenarios, it requires robust filtering to handle complex environments with obstacles. Future improvements could include more advanced optimization techniques or combining data from multiple sensors to further increase reliability.

V. Activity 3: Visually-guided navigation (II). Finding doors

A. Objective

The primary objective of this task is to extend the robot's navigation capabilities to include autonomous door detection and traversal, enabling transition between distinct rooms. The system must implement a robust feature extraction algorithm based on 2D LiDAR range-bearing vectors to identify closed doors within the environment. A Finite State Machine (FSM) will orchestrate the navigation tasks. The sequence commences at an assumed target in the room's geometric center, where the Localiser developed in Activity 2 is deployed to refine the robot's pose. Subsequently, the control logic involves: detecting the door's patch, navigating to a 1-meter standoff position, aligning the robot's heading with the opening, and cross at the other room.

B. Behavioral States

As explained previously, robot navigation behavior is defined by a finite set of operational states that govern its decision-making process and movement patterns, as shown in Figure 17.

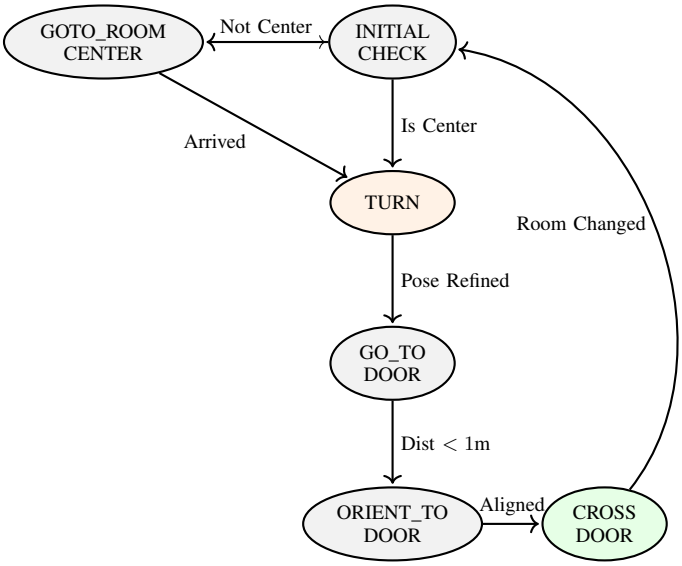


Fig. 17: Finite State Machine (FSM) logic for the door crossing task.

The process initiates in the *INITIAL_CHECK* state, where the system verifies if the robot is positioned at the room's geometric center. If the robot is not centered, it transitions to *GOTO_ROOM_CENTER* to navigate to the optimal starting position. Once the center is reached—or if the robot was already positioned there—the system changes to *TURN*.

In this state, the robot performs an in-place rotation until the LiDAR detects the target patch. Upon successful detection, it transitions to *GOTO_DOOR*, navigating towards the door located behind the identified patch. When the robot approaches within a 1-meter threshold of the entrance, it switches to

ORIENT_TO_DOOR to align its heading with the door frame. Finally, once alignment is confirmed, the state changes to *CROSS_DOOR* to traverse the opening, after which the cycle restarts for the new room. All states are detailed below:

1) GO_TO_ROOM_CENTER: Navigation behavior is encapsulated in the `goto_room_center` method, which operates as a closed-loop controller. The routine requires a valid target coordinate, representing the room's geometric center relative to the robot's current frame.

The control logic proceeds as follows:

1) **Distance Computation:** At each time step, the Euclidean distance d to the target is computed as the norm of the relative position vector $\mathbf{p}_{target}$:

$$d = \|\mathbf{p}_{target}\| \tag{14}$$

2) **Termination Condition:** A proximity threshold δ is defined at 100 mm. If the distance to the target falls below this threshold ($d < \delta$), the maneuver is considered complete. The robot sets its velocities to zero, resets the target variable, and triggers a state transition to `TURN`.
3) **Velocity Command:** If the threshold is not met, the target vector is passed to the `robot_controller` [24], which calculates the required linear (v) and angular (ω) velocities. These commands are then sent to the omnidirectional base platform to drive the robot towards the centroid.

2) TURN: The `turn` state implements an active perception strategy designed to locate specific visual landmarks (colored panels) within the environment. This method executes a scanning maneuver in which the robot performs a constant-velocity rotation at the place ($\omega = 0.2$ rad/s) while simultaneously processing visual data from the RGB camera.

The logic operates as follows:

1) **Target Selection:** The routine checks the Boolean flag `search_green` to determine the target hue (Green or Red). This allows the same state logic to be reused for different stages of the mission.
2) **Image Processing:** At each iteration, the system queries the `ImageProcessor::check_color_patch_in_image` static method. This function performs color segmentation and blob detection on the camera stream, returning a Boolean status indicating if the target patch is centered within the field of view.
3) **Termination Condition:** If the target is detected (‘detected == true‘), the robot immediately arrests its motion by setting all velocity components to zero. The internal flag `badge_found` is asserted, and the FSM transitions to the `GOTO_DOOR` state, signaling that the orientation phase is complete.
4) **Continuous Rotation:** As long as the target remains undetected, the method maintains the rotational velocity command. The kinematic control law governing this state is defined as:

$$S_{next} = \begin{cases} \texttt{GOTO\_DOOR} & \text{if detected} = \text{true} \\ \texttt{TURN} & \text{otherwise} \end{cases} \tag{15}$$

where S_{next} represents the state at the next instant in time.

3) GO_TO_DOOR: The `Go_To_Door` state orchestrates the approach maneuver towards the detected exit while simultaneously computing the precise orientation required for the subsequent alignment phase.

The control logic is divided into two operational stages:

1) **Target Acquisition and Approach** The system continuously queries the `door_detector` module. If no valid door candidates are identified within the LiDAR scan, the robot executes a recovery behavior by rotating at a fixed angular velocity (0.2 rad/s) to sweep the environment. Upon detection, a navigation target $\mathbf{p}_{target}$ is generated at a defined standoff distance ($\delta = 1000$ mm) perpendicular to the door's centroid. This ensures the robot stops with sufficient clearance to perform the final alignment maneuver.
2) **Alignment Vector Computation [23]** When the robot stops in front of the door ($v \approx 0$), it calculates the correct angle to face the opening. Since a door is detected simply as a line segment, there is a risk that the robot might calculate the angle backwards (facing the room instead of the exit). To fix this 180° ambiguity, the system uses the known center of the room as a reference. It calculates a vector pointing away from the room's center to ensure the robot faces the correct direction before trying to cross. To resolve this, the system uses the room's geometric center ($\mathbf{c}_{room}$) as a reference anchor. It compares the door's normal vector $\mathbf{n}$ with the vector pointing towards the room center ($\mathbf{v}_{inward}$) using the dot product. If the calculation indicates that the vector points into the room ($\mathbf{n} \cdot \mathbf{v}_{inward} > 0$), the target orientation is flipped to ensure the robot faces the exit:

$$\mathbf{v}_{target} = \begin{cases} -\mathbf{v}_{door} & \text{if } \mathbf{n} \cdot \mathbf{v}_{inward} > 0 \\ \mathbf{v}_{door} & \text{otherwise} \end{cases} \tag{16}$$

where:

$\mathbf{v}_{target}$ represents the final corrected heading.
$\mathbf{v}_{door}$ is the initial vector aligned with the door frame.
$\mathbf{n}$ denotes the door's normal vector.
$\mathbf{v}_{inward}$ is the reference vector pointing from the robot's position to the room's geometric center.

Finally, this local vector is transformed into a global world angle θ_{global}, which is stored as a "sticky" target. This prevents orientation jitter in the subsequent state. The system then transitions to `ORIENT_TO_DOOR`.

4) ORIENT_TO_DOOR: The `orient_to_door` state executes a precision rotation maneuver to align the robot's heading with the previously computed trajectory vector. This decoupling of target computation (performed in

GO_TO_DOOR) and execution ensures stability, as the target angle remains fixed ("sticky") regardless of sensor noise during the rotation.

The control loop functions as follows:

1) **State Validation:** The system first validates the existence of the stored orient_target_angle. This acts as a safety interlock; if the target is undefined, the system aborts to an safe state.
2) **Angular Error Computation:** The current global heading θ_{curr} is extracted from the robot's rotation matrix. The error θ_{err} is computed and, crucially, normalized to the range $(-\pi, \pi]$ to correctly handle the trigonometric discontinuity at $\pm 180°$:

$$\theta_{err} = \text{atan2}(\sin(\theta_{target} - \theta_{curr}), \cos(\theta_{target} - \theta_{curr})) \tag{17}$$

   θ_{err} represents the normalized angular error bounded within $(-\pi, \pi]$
   θ_{target} is the desired orientation (the "sticky" target) computed in the previous state
   θ_{curr} denotes the robot's current instantaneous heading in the world frame.
3) **Bang-Bang Control Law [22]:** Unlike the approach phase, the alignment uses a constant-velocity controller to ensure consistent turning behavior. The control output ω_z is determined by the sign of the error:

$$\omega_z = -\text{sgn}(\theta_{err}) \cdot \omega_{fixed} \tag{18}$$

   ω_{fixed} is set to 0.2 rad/s.
4) **Convergence:** The maneuver terminates when the absolute heading error falls within a tolerance window $\epsilon = 0.2$ rad ($\approx 11.5°$). Upon convergence, the robot halts ($\omega_z = 0$), clears the sticky target, and transitions to the subsequent state (e.g., CROSS_DOOR).

5) CROSS_DOOR: The final stage of the sequence, Cross_Door, executes a ballistic (open-loop) maneuver to traverse the physical threshold of the doorway. Unlike previous states that rely on continuous sensor feedback, this state assumes the heading alignment performed in ORIENT_TO_DOOR is sufficiently precise to allow a straight-line traversal. This approach minimizes the risk of oscillatory behavior caused by sensor noise within the narrow door frame. The execution logic is time-based rather than geometric:

1) **Temporal Initialization:** Upon entering the state, a monotonic clock timestamp t_{start} is captured. This ensures the maneuver has a fixed temporal reference frame.
2) **Constant Velocity Command:** The robot applies a constant longitudinal velocity v_x while maintaining zero lateral and rotational velocities. This enforces a straight trajectory:

$$\mathbf{v}_{lineal} = 400 \text{ mm/s} \tag{19}$$

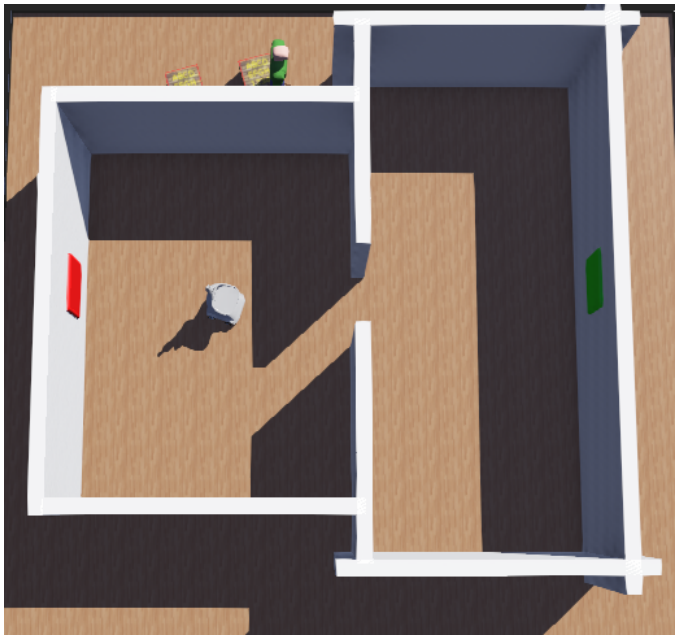
3) **Termination and Context Switch:** The system calculates the elapsed time $\Delta t = t_{now} - t_{start}$ at each control cycle. The maneuver terminates when Δt exceeds a pre-defined safety duration ($T_{limit} = 5.0$ s).
4) **Room Transition:** Upon timeout, the robot halts, resets the timer, and executes the switch_room() routine. This function updates the robot's topological belief (switching the active map or reference frame) before transitioning the FSM to GO_TO_ROOM_CENTER to begin operation in the new environment.

C. Experimental Results

The proposed navigation system was evaluated within a simulated environment developed in Webots. The testing arena consists of two adjacent rectangular rooms connected by a single, centrally located doorway. Both rooms are devoid of obstacles to isolate the door traversal performance.

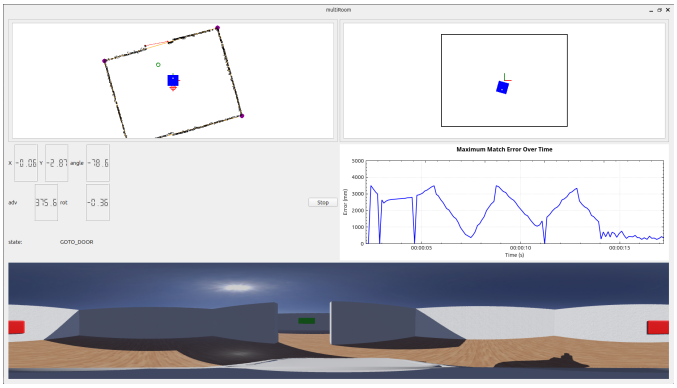
To facilitate the visual recognition task, colored fiducial markers (patches) are aligned with the door frame to guide the robot's approach. The initialization room is characterized by a red marker, while the destination room features a green marker.

Regarding the simulation interface, a real-time Heads-Up Display (HUD) provides critical telemetry data, including the robot's instantaneous linear and angular velocities, as well as the current active state of the Finite State Machine. Additionally, a dynamic plot visualizes the evolution of the minimization error metric throughout the trial.



(a) Initial Map Configuration

Fig. 18: Visual comparison of the simulation environment (Part 1).



(b) Simulation Result

Fig. 18: Simulation result. A full video of the execution is available at:
https://drive.google.com/file/d/10n_1iG7BUhLd8JUsoaLkpJoZF9h5ERZR/view?usp=sharing

D. Conclusions

The proposed system successfully demonstrates the efficacy of a modular Finite State Machine (FSM) for autonomous door traversal. Key contributions include the resolution of the 180° directional ambiguity through a geometric dot-product approach referencing the room's centroid, and the implementation of a hybrid control strategy. By decoupling the precise alignment phase (closed-loop) from the traversal phase (open-loop), the system effectively mitigated sensor noise within the narrow doorway.
These results validate the architecture's robustness for reliable topological transitions in multi-room environments.

References

[1] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, Springer, 2011.
[2] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.
[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, MIT Press, 2005.
[4] L. Rodríguez, P. Bustos, and P. Bachiller, "RoboComp: A Tool-based Robotics Framework," *Simulation Modelling Practice and Theory*, vol. 19, no. 9, pp. 1815–1825, 2011.
[5] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 2002.
[6] D. Brugali, "Software engineering for robotics," *IEEE Robotics and Automation Magazine*, vol. 22, no. 3, pp. 24–32, 2015.
[7] O. Michel, "Cyberbotics Ltd. Webots: Professional mobile robot simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, 2004.
[8] B. Stroustrup, *The C++ Programming Language*, 4th ed., Addison-Wesley, 2013.
[9] C. Bird et al., "The promises and perils of mining GitHub," in *Proc. MSR*, pp. 1–10, 2011.
[10] K. Hoffman, "CMake: Cross Platform Make," *Software—Practice and Experience*, vol. 33, no. 10, 2003.
[11] R. R. Murphy, *Introduction to AI Robotics*, MIT Press, 2000.
[12] M. Quigley et al., "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.
[13] Amazon Robotics. (2025) "Mobile Robotic Fulfillment Systems." [Online]. Available: https://www.amazonrobotics.com
[14] SZ DJI Technology Co., Ltd. (2025) "DJI Phantom Series." [Online]. Available: https://www.dji.com/phantom
[15] Boston Dynamics. (2025) "Spot - The Agile Mobile Robot." [Online]. Available: https://www.bostondynamics.com/spot
[16] VideoRay LLC. (2025) "Remotely Operated Vehicles (ROVs)." [Online]. Available: https://videoray.com
[17] P. Bustos, "Robot localisation in a rectangular room," Robolab Technical Report RL0020, 29 Oct. 2025.
[18] M. A. Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.
[19] J. M. Martínez-Otzeta, I. Rodríguez-Moreno, I. Mendialdua, and B. Sierra, "RANSAC for Robotic Applications: A Survey," *Sensors*, vol. 23, no. 1, p. 327, 2023.
[20] H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.
[21] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.
[22] S. N. S. Al-Humairi and A. H. K. Al-Jiboori, "Design and Simulation of Bang-Bang Controller for DC Motor Speed Control," in *2019 International Conference on Advanced Science and Engineering (ICOASE)*, Zakho - Duhok, Iraq, 2019, pp. 138–143. [Online]. Available: https://ieeexplore.ieee.org/document/8980070
[23] G. Sanderson, "Dot products and duality | Chapter 9, Essence of linear algebra," *3Blue1Brown*, Aug. 2016. [Online Video]. Available: https://youtu.be/LyGKycYT2v0
[24] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, "Mobile Robot Kinematics," in *Introduction to Autonomous Mobile Robots*, 2nd ed. Cambridge, MA: MIT Press, 2011, ch. 3, pp. 47–88.

Ismael González Loro is an undergraduate student in Computer Engineering at the Polytechnic School of Cáceres, University of Extremadura. His email address is igonzaleoa@alumnos.unex.es.

Samuel Corrionero Fernández is an undergraduate student in Computer Engineering at the Polytechnic School of Cáceres, University of Extremadura. His email address is scorrion@alumnos.unex.es.

José Pulido Delgado is an undergraduate student in Computer Engineering at the Polytechnic School of Cáceres, University of Extremadura. His email address is jopulidod@alumnos.unex.es.