Autonomous Mapping Robot

Elman Steve Laguna

October 22, 2025

1 Short Report on Problem

1.1 Description of the Project

This project aims to investigate and implement algorithms for autonomous environment mapping and navigation using a simulated agent. The primary goal is to develop a system that can explore an unknown area, construct a map, and calculate efficient routes to a target without human intervention.

1.2 Project Ideas and Goals

1.2.1 Idea 1: 2D Environment Mapping and Pathfinding

The core problem is to enable an agent to automatically move through and map an unknown 2D area. This investigation will be approached through a comparative study using three distinct simulation environments to understand the trade-offs and challenges of each. The main deliverable for this idea is a functional simulation where an agent successfully maps its environment and navigates around obstacles.

1.2.2 Idea 2: 2D Maze Traversal and Optimization (Alternative)

As an alternative or extension, this idea focuses more specifically on pathfinding optimization within a known environment, such as a maze. Using the same three engines (ROS, PyGame, Bevy), the goal would be to implement and compare various search algorithms (e.g., A*, Dijkstra's) to determine the most efficient path to a target—"the fastest to the cheese." This shifts the focus from exploration and mapping (SLAM) to pure algorithmic performance.

1.3 Preliminary Literature Search

Initial research focuses on **Simultaneous Localization and Mapping (SLAM)**, the core problem of concurrently building a map while tracking an agent's position within it. To simulate sensor data for perception, the project will implement **ray casting**, using line-intersection calculations to mimic how a Lidar scanner detects obstacles and builds a map. **Procedural maze generation** will be used to create complex, structured environments for testing the navigation logic. Finally, once a map is generated, **shortest path algorithms** such as Dijkstra's and A* Search will be investigated and implemented to provide the agent with efficient navigation capabilities. Initially, I will start with a simple handmaid map. Them make the tasks incrementally more difficult.

1.4 Ideas on Approach

The proposed approach will begin with the 2D Environment Mapping idea. The project will be executed in three phases:

1. **Phase 1: Foundational Learning with ROS.** I will start by mastering the basics of agent control and sensing within a structured environment using the ROS Turtlesim package. This will establish a baseline understanding of robotic simulation.
2. **Phase 2: Custom Simulation with PyGame.** Next, I will develop a 2D simulation from scratch in Python with PyGame. This will provide the flexibility to implement and visualize a mapping algorithm (such as a basic grid-based or occupancy grid map) and a pathfinding algorithm (like A*).

3. **Phase 3: Advanced Implementation in Bevy.** Finally, I will replicate the simulation in Rust using the Bevy engine. This phase will focus on evaluating the performance, modularity, and scalability benefits of a modern game engine for robotics simulations.

Throughout these phases, the primary challenge will be to process simulated sensor data (e.g., raycasting to simulate Lidar) to build the map and inform the navigation logic.

Project Status and Technology Stack

This project combines multiple technologies to create a simulation of a micromouse competition robot.

Core Technologies

Rust Programming Language The entire simulation is built using the Rust Programming Language.

Bevy Game Engine (v0.16.1) Bevy provides the foundational framework for the simulation.

Physics Simulation - Avian2D Avian2D provides realistic 2D physics simulation.

Supporting Libraries

bevy_ecs_tilemap Specialized tilemap rendering for maze visualization.

Knossos Procedural maze generation library.

Python Ecosystem

Python handles data analysis and visualization.

Architecture Integration

The system follows a modular pipeline architecture:

1. **Rust/Bevy**: Real-time simulation, physics, and sensor processing.
    - Maze generation (Knossos)
    - LiDAR simulation (custom raycast system)
    - Robot control (WORK IN PROGRESS)
    - Physics simulation (Avian2D)
2. **Data Export**: Sensor readings and position data serialized to JSON via `serde`.
3. **Python Analysis**: Post-processing and visualization.
    - Occupancy grid construction from LiDAR scans

Development Tools

UV Package Manager Fast, reliable Python dependency management.

Cargo Rust's build system and package manager, handling all compilation and dependencies.

Design Rationale

This technology stack was chosen to balance several competing requirements:

Performance Rust provides low-level performance with a modern syntax.

Safety Compile-time guarantees prevent logic errors that would be runtime bugs in Python/C++.

Modularity ECS enables easy experimentation with different components.

Analysis Python is super easy to use for data analysis and visualization.

Personal Growth Learning Rust and Bevy is a personal goal.

Small Maze Generation

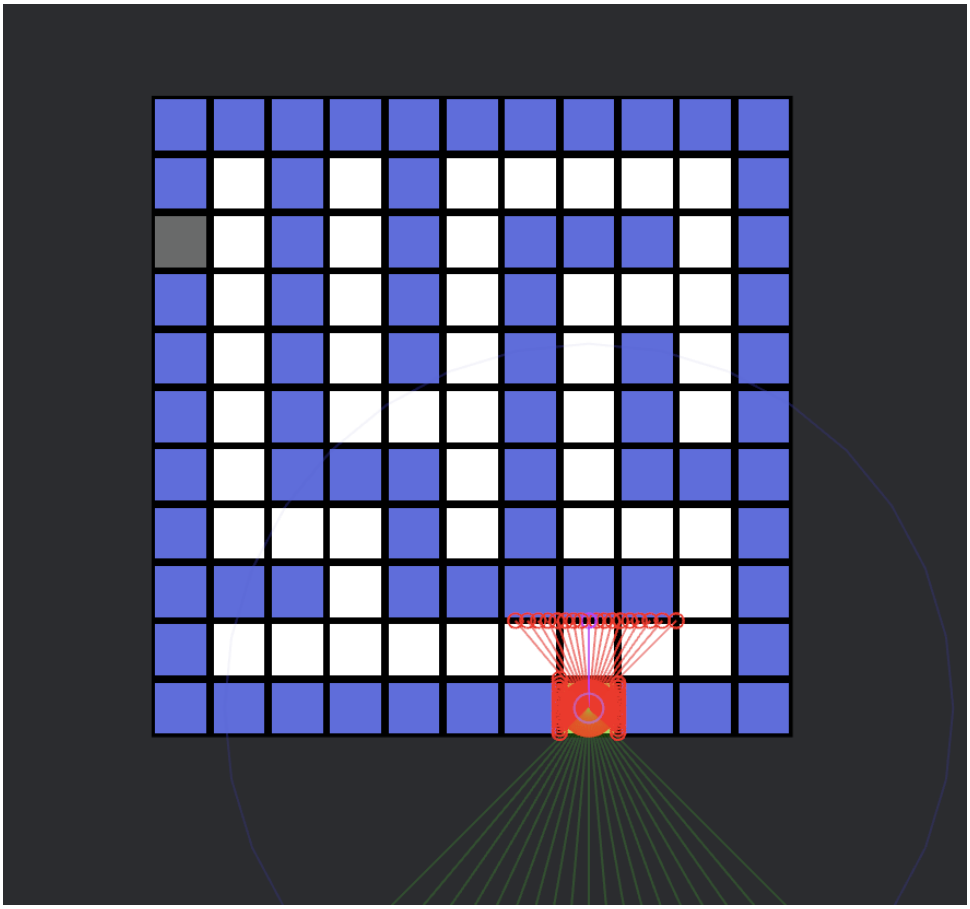

Figure 1: A 5×5 maze configuration generating an 11×11 tile grid.

Medium Maze Generation

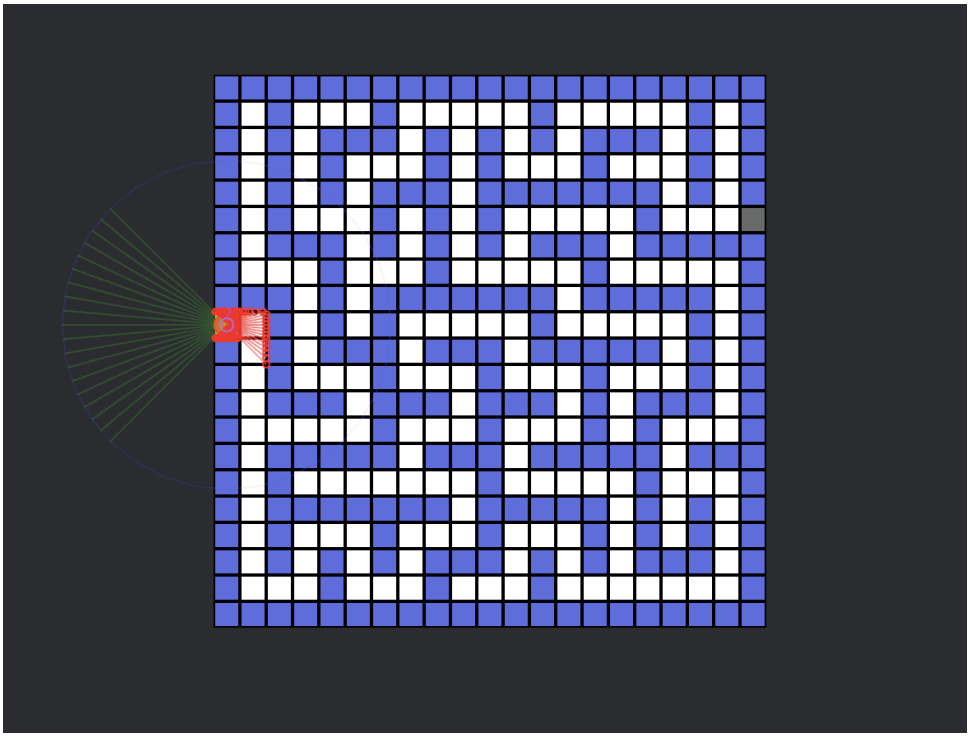

Figure 2: A 10×10 maze configuration generating a 21×21 tile grid.

Large Maze Generation

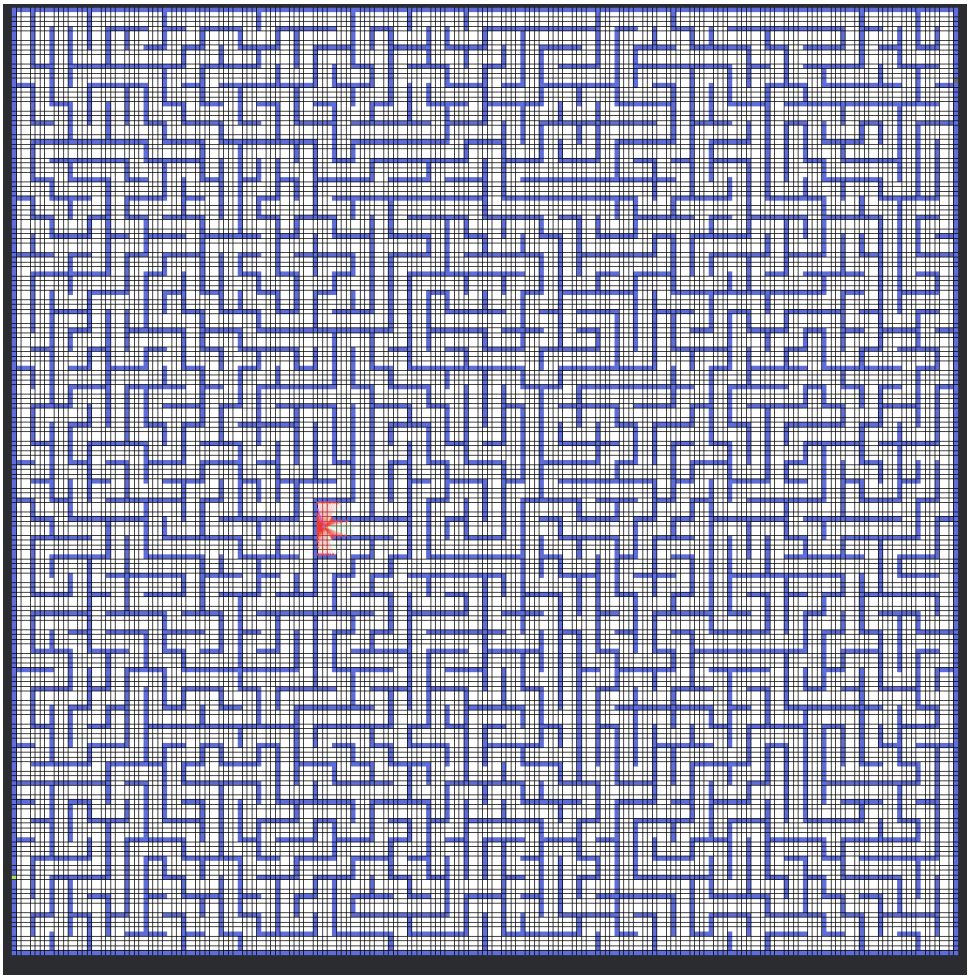

Figure 3: A 50×50 maze configuration generating a 101×101 tile grid.

Maze Generation

The maze generation system uses the `knossos` library with various algorithms to create procedurally generated mazes. The generation process follows these key steps:

1. **Cell Grid Creation**: An initial grid of `MAZE_WIDTH` × `MAZE_HEIGHT` cells is created, representing potential passages in the maze.
2. **Wall Insertion**: The Recursive Backtracking algorithm carves passages through the grid, with walls placed between cells and around the perimeter.
3. **Dimension Calculation**: The final tile count follows the formula:

$$\text{Final Size} = (\text{MAZE\_WIDTH} \times 2 + 1) \times (\text{MAZE\_HEIGHT} \times 2 + 1) \tag{1}$$

   This accounts for the cells, walls between cells, and border walls.
4. **Start and Goal Placement**: The algorithm automatically places a start position and goal position within the generated maze using a configurable seed for reproducibility.

The use of a fixed seed (`SEED = 490`) ensures that the same maze configuration is generated consistently for testing and comparison purposes. The `GAME_MAP_SPAN` parameter controls the spacing between maze elements, set to 1 for standard wall thickness. Turns out that using too thick of a wall makes this a different problem. I have to start using slam techniques.

Each configuration demonstrates the scalability of the generation system, from small 11×11 mazes suitable for rapid testing to large 101×101 mazes for complex pathfinding challenges.

Pathfinding Results

Poor Quality Data (Post Processing)

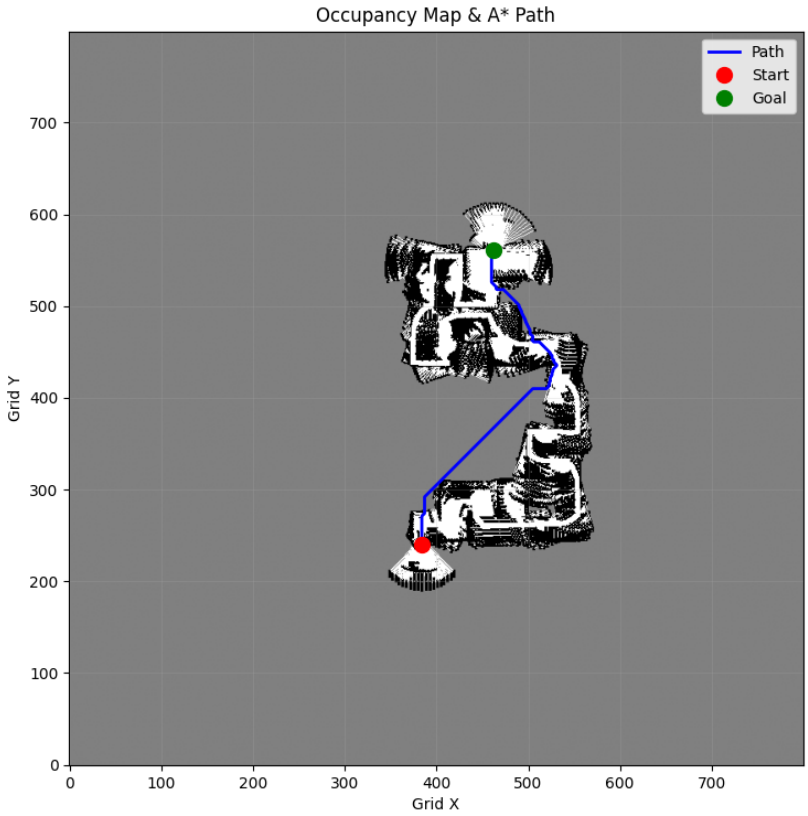

Figure 4: Pathfinding on a 5×5 maze (11×11 tile grid) showing poor reconstruction quality.

Improved Data (Post Processing)

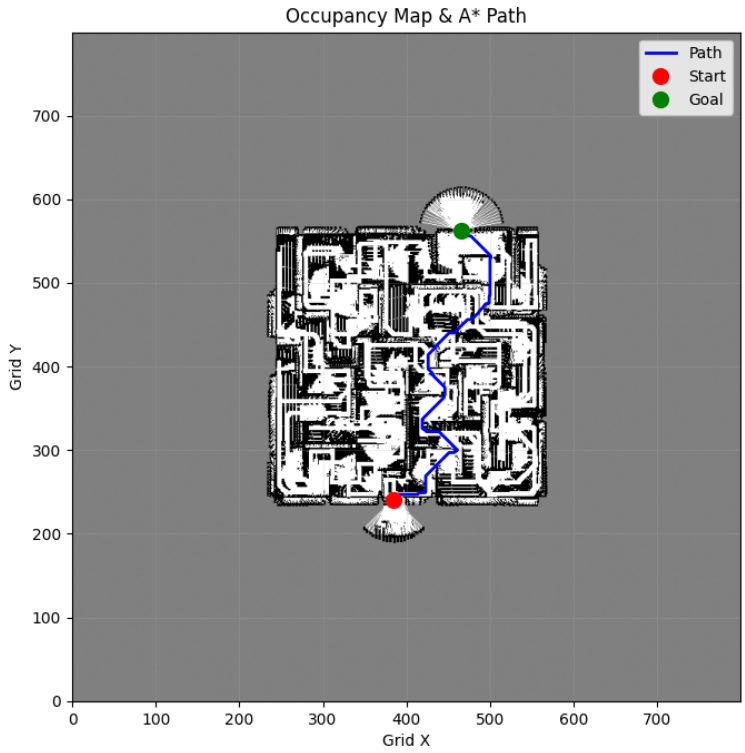

Figure 5: Pathfinding on a 10×10 maze (21×21 tile grid) showing better reconstruction.

Large Maze (Post Processing)

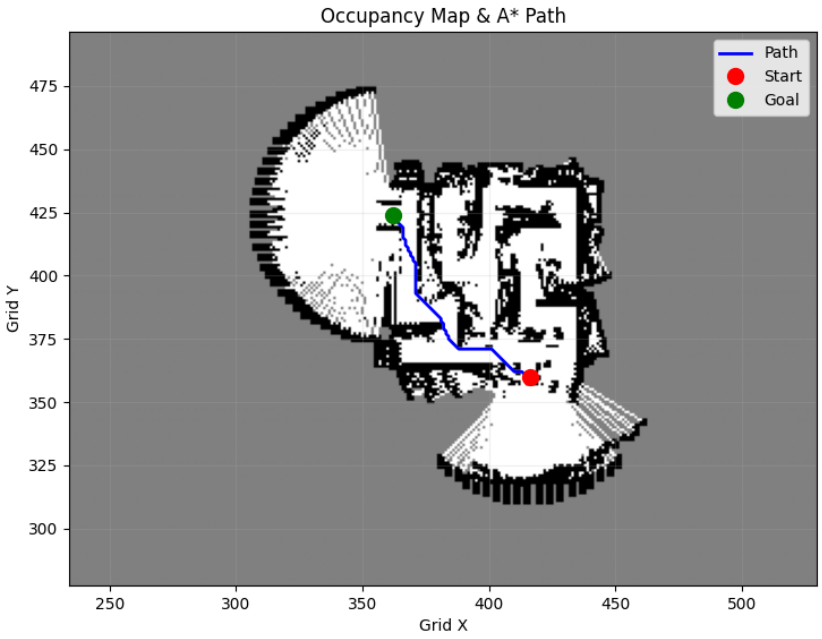

Figure 6: Pathfinding on a 50×50 maze (101×101 tile grid).

Pathfinding Algorithm

The A* algorithm is used for maze solving in Python and Matplotlib to visualize.

Current Challenges and Future Work

Data Collection

- Manual robot control for LiDAR data acquisition is tedious and time-consuming
- Need to implement autonomous navigation

Technical Knowledge Gaps

- SLAM (Simultaneous Localization and Mapping) fundamentals require further study
- Mapping relationship between physical and simulated environments unclear
- Occupancy grid construction from LiDAR data needs research

Abstraction Level

- Struggling to understand and reduce infinite real-world detail to finite usable data
- Should focus on tile-to-tile navigation rather than millimeter precision
- Need to simplify to single-tile-width paths initially

Scope Reduction

- Prioritize discrete tile-based movement over continuous positioning
- Focus on traversal between adjacent tiles as primary navigation unit

References

[1] M. Aborizky, H. A. As'ari, A. Fadlil and R. D. P. W. "Lidar-Based 2D SLAM for Mobile Robot in an Indoor Environment: A Review," *2021 9th International Conference on Information and Communication Technology (ICoICT)*, 2021, pp. 488-493. doi: 10.1109/ICoICT52021.2021.9538731.

[2] L. Vandevenne, "Raycasting," *Lode's Computer Graphics Tutorial*. [Online]. Available: `https://lodev.org/cgtutor/raycasting.html`

[3] NEON Science, "Lidar Basics," *National Ecological Observatory Network*, 2021. [Online]. Available: `https://www.neonscience.org/resources/learning-hub/tutorials/lidar-basics`

[4] Zona Land Education, "Intersection of Two Lines," *Zona Land Education*. [Online]. Available: `http://zonalandeducation.com/mmts/intersections/intersectionOfTwoLines1/intersectionOfTwoLines1.html`

[5] NEON Science, "How Does LiDAR Remote Sensing Work? Light Detection and Ranging," *YouTube*, Nov. 24, 2014. [Video]. Available: `https://www.youtube.com/watch?v=EYbhNSUnIdU`

[6] FinFET, "Raycasting with Pygame in Python! Simple 3D game tutorial Devlog," *YouTube*, Nov. 6, 2021. [Video]. Available: `https://www.youtube.com/watch?v=4gqPv7A_YRY`

[7] WeirdDevers, "Ray casting fully explained. Pseudo 3D game," *YouTube*, Dec. 27, 2020. [Video]. Available: `https://www.youtube.com/watch?v=g8p7nAbDz6Y`

[8] Veritasium, "The Fastest Maze-Solving Competition On Earth," *YouTube*, May 24, 2023. [Video]. Available: `https://www.youtube.com/watch?v=ZMQbHMgK2rw`

[9] mattbatwings, "What School Didn't Tell You About Mazes #SoMEpi," *YouTube*, Jun. 22, 2024. [Video]. Available: `https://www.youtube.com/watch?v=uctN47p_KVk`

[10] Kai Nakamura, "How to Make an Autonomous Mapping Robot Using SLAM," *YouTube*, May 14, 2024. [Video]. Available: `https://www.youtube.com/watch?v=xqjVTE7QvOg`

[11] Articulated Robotics, "How do we add LIDAR to a ROS robot?," *YouTube*, Jun. 2, 2022. [Video]. Available: `https://www.youtube.com/watch?v=eJZXRncGaGM`