Autonomous Drone-Assisted Maze Solver Car Robot

Project Report

1. Abstract

Autonomous navigation is one of the most critical challenges in robotics. Traditionally, robots rely on onboard sensors such as ultrasonic, LiDAR, or infrared for obstacle detection and path planning. However, these methods can be expensive, prone to noise, and limited by the robot's field of view.

This project introduces a **novel strategy** where a **drone provides an overhead view** of the maze, and the ground robot executes the computed path. The system employs **computer vision** to detect start, goal, and obstacle positions using color-based segmentation. Pathfinding is achieved with the **A* algorithm**, which guarantees an optimal path in a grid-based environment. The **AprilTag fiducial marker system** is used for real-time localization of the robot, ensuring accurate tracking and correction.

The project demonstrates a robust method for maze solving without onboard sensing, showcasing its potential applications in **search and rescue, automated warehousing, and swarm robotics**.

2. Introduction

2.1 Problem Statement

Robots designed to solve mazes or navigate unknown terrains typically rely on onboard sensors. These systems:

- Have **limited perception range**.
- Are prone to **errors in complex lighting environments**.
- Increase **cost and complexity** with additional sensors.

Our project eliminates these constraints by outsourcing vision to an overhead drone.

2.2 Proposed Solution

- **Drone (eye in the sky):** Captures a **global view** of the maze.
- **Ground station:** Runs **image processing + A*** to plan a collision-free path.
- **Robot:** Executes the path using commands from ground station.

This **perception-execution split** simplifies the robot's design and provides flexibility for multi-robot systems.

---

3. Objectives

1. To design and implement a **maze-solving robot** with drone-assisted navigation.
2. To detect **start (AprilTag Position), end (blue), and obstacles (red, green)** using **OpenCV HSV color filtering**.
3. To implement **obstacle inflation** ensuring safe clearance.
4. To perform **A* pathfinding** and generate step-by-step waypoints.
5. To simulate path execution and visualize results with **matplotlib**.
6. To integrate **AprilTags** for real-time position tracking of the robot.
7. To communicate path instructions from drone to robot via **ESP32 Wi-Fi/Serial**.

---

4. Hardware Components

Component	**Specification**	**Purpose**
Raspberry Pi 4	4GB RAM	Runs camera + preprocessing
Logitech Webcam	1080p	Overhead maze capture
Li-Po Battery (11.1V) (Drone)	2200 mAh	Power supply
Drone Frame (S500)	Custom lightweight	Holds Pi + camera
Gimble	2 Axis	Stabilizes camera during flight
BLDC Motors x 4	1000 – 1200 KV	Drone propulsion
30A ESC x 4	Electronic Speed Controller	Controls BLDC Motors
Propellors (CW/CCW) x 5 Pair	10x4.5 inch	Provides thrust & lift
B3 Balance Charger	3s Li-Po	Safely charges Li-Po battery
Battery Voltage Indicator	LED/Display Module	Monitors Battery Level
APM 2.8 Flight Controller	ATmega2560-based	Stabilizes drone & controls flight
Power Module (Drone)	90A max with 5V/3A BEC	Supplies regulated power to APM + Pi
Li-Ion Battery (TP18650) (Car)	7.4V, 2200 mAh	Powers robot car
ESP32-WROOM	Wi-Fi, Dual Core	Robot control + communication
L298N Motor Driver	Dual H-Bridge	Drive motors
BO Motors + Wheels	200 RPM	Robot locomotion

Chassis	2WD	Robot platform
AprilTags	Printed fiducials	Robot tracking

5. Software Components

- **Python 3.10**
- **OpenCV 4.x** – Image processing
- **Matplotlib** – Visualization
- **NumPy** – Grid-based calculations
- **AprilTag Library (pyAprilTag)** – Marker detection
- **ESP-IDF / Arduino IDE** – ESP32 programming
- **Serial/Wi-Fi Socket Communication** – Ground station ↔ Robot

6. Algorithms

6.1 Color Detection

Process:

1. Convert image from BGR → HSV.
2. Apply thresholding for each color:
   - Blue (End)
   - Red (Obstacles)
   - Green (Obstacles)
3. Extract largest contour in each color range.
4. Get centroid coordinates (x, y).

Equation:

```
mask = cv2.inRange(hsv, lower, upper)

contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

M = cv2.moments(contour)

cx = int(M['m10']/M['m00'])

cy = int(M['m01']/M['m00'])
```

6.2 Obstacle Inflation

To prevent collisions, obstacles are expanded by a **safety margin (r)** pixels. This ensures the path does not cut too close to edges.

Algorithm:

```
kernel = np.ones((r,r), np.uint8)

inflated_obstacles = cv2.dilate(obstacle_mask, kernel, iterations=1)
```

6.3 A* Pathfinding

Heuristic function (Manhattan distance):

h(n)=|xgoal−xn|+|ygoal−yn|h(n) = |x_{goal} - x_n| + |y_{goal} - y_n|

Cost function:

f(n)=g(n)+h(n)f(n) = g(n) + h(n)

Pseudocode:

```
function A*(start, goal, grid):
  openSet ← {start}
  gScore[start] = 0
  fScore[start] = h(start)

  while openSet not empty:
    current = node with lowest fScore
    if current == goal:
      return reconstruct_path()

    remove current from openSet
    for neighbor in neighbors(current):
      tentative_g = gScore[current] + dist(current, neighbor)
      if tentative_g < gScore[neighbor]:
```

```
cameFrom[neighbor] = current
gScore[neighbor] = tentative_g
fScore[neighbor] = gScore[neighbor] + h(neighbor)
if neighbor not in openSet:
   add neighbor
```

6.4 AprilTag Tracking

- AprilTags are square binary markers.
- The camera detects them and returns:
    - **ID** (unique identifier)
    - **Pose (x, y, θ)** relative to camera
- Used to correct robot drift by adjusting commands.

---

7. Strategy

- **Phase 1 (Global Planning):**
    - Drone takes image, path computed once.
- **Phase 2 (Execution & Tracking):**
    - Robot follows path.
    - Drone corrects drift via AprilTags.

---

8. Implementation Stages

Phase 1: Foundational Simulations

Version 1: rfl_MazeSolverUsingDrone.ipynb

- **What it did:** This version was the initial proof of concept, running a pure simulation. It generated a random 2D grid representing a maze with a start point, end point, and static obstacles. It then applied the A* algorithm to find the optimal, shortest path.
- **Pros:** It successfully demonstrated the core pathfinding algorithm's functionality, guaranteeing an optimal path. The visualization was clear and easy to understand.

- **Cons:** The main drawback was its lack of a real-world application. It was a simulation and did not account for the complexities of a physical environment or a drone's movement.

Version 2: rfl_MazeSolverUsingDrone_v2.ipynb

- **What it did:** This version was an improvement on the first, with a focus on path quality. It used the same A* algorithm but generated a smoother path with fewer turns.
- **Pros:** The smoother path was more efficient and realistic, as it would require fewer maneuvers from a physical drone.
- **Cons:** It remained a software simulation and did not progress toward real-world application.

Phase 2: Introduction of Real-World Elements

Version 3: rfl_MazeSolverUsingDrone_LiveFeed.ipynb

- **What it did:** This was the first version to use a real-world component. It took a static image from a webcam of a maze created with Lego bricks and applied the A* algorithm to find a path.
- **Pros:** It validated the ability to use computer vision to analyze a real-world scene, a critical step for the project.
- **Cons:** It was still limited to a single static image and could not handle dynamic changes or continuous movement.

Version 4: rfl_MazeSolverUsingDrone_LiveFeed_v2.ipynb

- **What it did:** An important but ultimately failed attempt to use a live video feed. It tried to re-calculate the path in every frame.
- **Pros:** The idea of using a live feed was a crucial step towards a real-time system.
- **Cons:** The frequent path recalculations caused the path to fluctuate wildly, leading to a system failure and a flood of popups. This highlighted the need for a more stable architectural approach.

Version 5: rfl_MazeSolverUsingDrone_LiveFeed_v3.ipynb

- **What it did:** This version introduced a two-phase strategy to solve the problems of the previous one. The first phase captured a static image and calculated the path. The second phase used a live video feed to track an AprilTag on a phone screen, allowing a human to manually "fly" along the pre-determined path.

- **Pros:** The two-phase strategy was robust and avoided the continuous recalculation failures. It successfully demonstrated real-time tracking with AprilTags.
- **Cons:** The system was still not autonomous, as a human was controlling the movement.

Version 6: rfl_MazeSolverUsingDrone_LiveFeed_v4(Car)

- **What it did:** The project was scaled up. Instead of a phone screen, a physical ESP32 car with an AprilTag was used in a larger environment with big blocks as obstacles. An initial image was used to find a path, and the car was manually driven along it while the AprilTag was tracked.
- **Pros:** This was a significant step toward a real-world application. It successfully integrated a physical robot platform and a larger, more realistic environment.
- **Cons:** The car's movement was still not autonomous; it was manually controlled by a human.

Version 7: rfl_MazeSolverUsingDrone_LiveFeed_v5(Car)

- **What it did:** A key refinement of Version 6. The AprilTag on the car automatically served as the starting point for the pathfinding algorithm.
- **Pros:** This streamlined the workflow and made the setup more efficient, as it eliminated the need for a separate physical object to mark the start.
- **Cons:** The fundamental limitation of manual control remained.

Phase 3: Autonomous Versions

Preliminary Scripts: direction.py and forward.py

- **What they did:** These were not full versions but foundational scripts for autonomy. direction.py used the AprilTag to determine the car's heading, while forward.py calibrated the time delays for forward and stop commands.
- **Pros:** These scripts were essential to creating a precise and reliable control loop for autonomous navigation.
- **Cons:** They were individual components, not a complete, functioning system.

Version 8: autonomous.py

- **What it did:** The first true autonomous version. With a clear path and no obstacles, the car could align itself with a goal and drive towards it autonomously.
- **Pros:** Successfully closed the control loop, proving that the system could use AprilTag data for real-time navigation.

- **Cons:** It lacked obstacle avoidance and could only navigate a straight, clear path.

Version 9: autonomous_v2.py

- **What it did:** An incremental improvement on Version 8. It used the AprilTag as the starting point, automating the initial setup.
- **Pros:** It streamlined the autonomous process from the very beginning.
- **Cons:** It still could not handle obstacles and was limited to navigating a clear path.

Version 10: autonomous_v3.py and autonomous_v4.py

- **What it did:** This version was the first attempt to implement obstacle avoidance in the autonomous system. It tried to combine pathfinding with autonomous control to navigate a maze.
- **Pros:** It was an ambitious and necessary step to merge all components.
- **Cons:** It was unsuccessful, likely due to issues with localization drift, imprecise control, and system latency.

Version 11: autonomous_v5.py

- **What it did:** This was the first fully functional autonomous version that successfully navigated a maze with obstacles. It used a start-stop-start-stop approach to move between waypoints.
- **Pros:** It proved the core concept of autonomous maze-solving was viable.
- **Cons:** The start-stop motion was inefficient and slow, highlighting the need for a smoother control system.

Version 12: autonomous_v6.py

- **What it did:** An unsuccessful attempt to improve movement by using a P-controller. It aimed for continuous motion but failed.
- **Pros:** It represented a crucial step towards a more sophisticated control system.
- **Cons:** The P-controller did not work as intended, likely due to poor tuning, latency, and the non-linear behavior of the motors.

Version 13: autonomous_v7.py

- **What it did:** The final, most successful version. It used a hybrid control system, with continuous, smooth motion for straight segments and a reliable start-stop approach for turns.

- **Pros:** It was both fully functional and efficient, significantly reducing the time taken to solve the maze. It demonstrated a practical and robust hybrid approach to a complex problem.
- **Cons:** The stop-start turns, while effective, were still a workaround for a true continuous-motion controller. This leaves room for future improvement.

---

9. Steps to Execute

Step 1 – Powering On:

Power the esp32 Car by connecting the battery

Power on Raspberry Pi

Step 2 - Setup:

Make Sure that Raspberry pi is Connected to Esp32_car Wi-Fi

Step 3 – Running Raspberry Pi Startup Code:

Run Server.py on Raspberry pi

Step 4 – Configure Maze:

Place the car anywhere in the maze and place Red, Green and Blue Blocks anywhere in the maze.

Step 5 – Activate the Code using Button:

Press the push Button on the Car.

Step 6 – Execution

- Robot follows step-by-step commands:
    - Forward (F)
    - Left Turn (L)

    - Right Turn (R)

- Drone tracks via AprilTags and applies correction.

---

10. Results & Discussion

- Robot **successfully reached goal** in all test runs.
- Path was always **optimal** due to A*.
- Obstacle inflation ensured **safe clearance**.
- AprilTag tracking improved **accuracy by ~20%** compared to blind execution.
- **Limitations:**
    - Lighting changes affect color detection.
    - Wi-Fi latency in command transmission.
    - Requires stable drone positioning.

---

11. Conclusion

This project successfully demonstrates a **drone-assisted maze solver robot** using *image processing, A pathfinding, and AprilTag tracking**. By outsourcing vision to an overhead drone, the robot is simplified and lightweight. This framework can be extended to multiple robots, search-and-rescue, and industrial automation.

---

15. References

- OpenCV: https://opencv.org/
- AprilTag: Olson, E. (2011). "AprilTag: A robust fiducial system."
- Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths."