Hybrid Remote-Autonomous Robot Using SLAM and April Tag Detection

Nargess Seifi nargessseifi@outlook.com
Sanaz Motie s.motie@gmail.com

Supervisor: prof. Javad Vahidi

September 2024

Abstract

This project develops a hybrid robot using SLAM and April Tag detection for navigation. The robot, controlled by an ESP32 and equipped with motors and a smartphone camera, connects to a laptop for real-time video processing. SLAM is managed by the stella_vslam library, and April Tag detection is by Pupil-apriltags. A Python script handles tag detection and communicates with the ESP32 for movement control. The robot can operate remotely via a web interface or autonomously navigate to tags using a PID controller. The April Tags' locations are identified on the SLAM-generated map, and after reaching the target, the robot switches back to remote control. This system integrates human control with automated navigation and localization.

Contents

1 Introduction

This project aims to design and implement a hybrid remote-autonomous robot that leverages SLAM (Simultaneous Localization and Mapping) and April Tag detection for real-time navigation and environment mapping. This system combines manual control via a web interface with autonomous navigation triggered by the detection of April Tags, creating a versatile and efficient robotic platform.

The robot is equipped with essential components such as motors and an ESP32 microcontroller, while a smartphone serves as the camera. The phone streams live video to a laptop via Droidcam, enabling real-time video processing. The Stella_vslam library is responsible for tracking the robot's movement, performing localization, and constructing a map of the environment. On the other hand, the Pupil-apriltag library detects April Tags in the environment, which serve as reference points for autonomous navigation.

The Python script, a key component of the system, plays a pivotal role in managing the April Tag detection and communication with the ESP32. When an April Tag is detected during the robot's movement, the script sends an HTTP request containing the tag's attributes (distance and horizontal position) to the ESP32 via the robot's dedicated WiFi network. Upon receiving this data, the robot transitions from manual control to autonomous operation. During this mode, the robot uses a nonlinear transformation of the tag's attributes and a PID controller to adjust its movement and align itself with the tag.

Once the robot reaches the tag and centres it within the camera's view (where calculation errors are minimal), the laptop calculates the tag's precise location relative to the robot's position. This information is saved, and the robot receives a "seen" message, signalling it to revert to manual control.

This hybrid system allows the robot to operate efficiently in dynamic environments, combining the flexibility of human control with the precision and speed of autonomous navigation. By integrating SLAM and April Tag detection, the robot is able to map its surroundings while accurately approaching specific reference points, making it suitable for tasks that require both navigation and object interaction.

2 System Overview

2.1 General Architecture

The system consists of an ESP32-controlled robot with motors and a smartphone camera. The camera sends live video to a laptop via wifi. Two software processes run on the laptop simultaneously. Firstly, a SLAM executable, created using the Stella_vslam library, is responsible for localization and mapping in C++. Secondly, a Python script handles April Tag detection and communicates with the ESP32 for movement control.

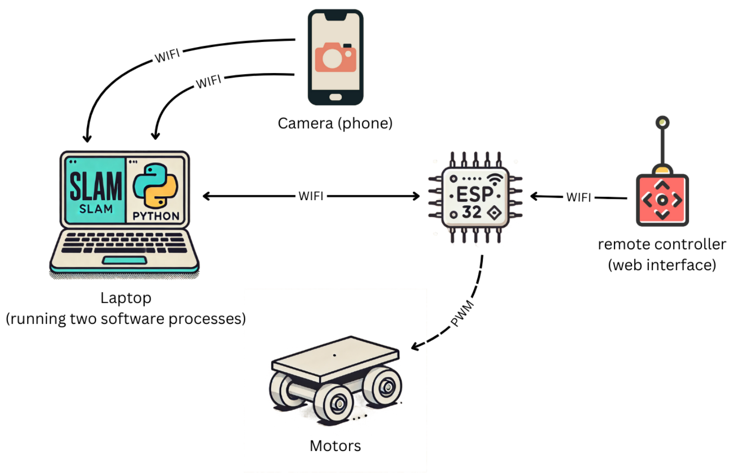


Figure 1: General Architecture of the System

2.2 Control Flow

The control flow between the camera, ESP32, and motors follows a well-defined sequence, ensuring seamless transitions between manual and autonomous modes:

1. **Camera to Laptop (SLAM and April Tag Detection):**

   The smartphone's video feed is streamed to the laptop via Droidcam. The Stella_vslam library processes this feed to localize the robot and map its environment in real-time. Simultaneously, the Pupil-apriltag library scans the video frames for April Tags, which serve as navigational targets.

2. **Tag Detection to ESP32 (Transition to Autonomous Mode):**

   When an April Tag is detected, the Python script extracts its attributes, including the relative distances to the robot in cm. Then, an HTTP request containing these tag attributes is sent to the ESP32 over the robot's wifi network.

   Upon receiving the request, the ESP32 updates its global state, triggering a transition from remote to autonomous control. This shift disables manual input from the web interface, allowing the robot to focus on reaching the detected tag autonomously.

3. **Autonomous Movement (ESP32 and Motors):**

   Using a nonlinear transformation of the tag's attributes, the ESP32 calculates the necessary movements for the robot to align itself with the April Tag. Based on these calculations, a PID controller is applied to fine-tune the movement, ensuring smooth and accurate navigation.

   The ESP32 adjusts the motor speeds accordingly, enabling the robot to move toward the tag. As the robot approaches the tag, it continuously receives updated tag attributes from the laptop to refine its movement.

4. **Robot Reaches the Tag (Feedback to Laptop):**

   Once the robot centres the April Tag in its camera view, it sends a confirmation back to the laptop, signalling that it has reached the appropriate distance from the tag.

   At this stage, the laptop computes the absolute coordinates of the April Tag based on the robot's current position and orientation and the relative tag distance. The tag's location is saved, and a "seen" message is returned to the ESP32.

5. **Return to Remote Control:**

   Upon receiving the "seen" message, the ESP32 resets its global state, re-enabling manual control from the web interface. The robot is now back in its primitive state, awaiting further commands from the remote controller.

   This structured control flow ensures efficient and reliable transitions between remote and autonomous operation, enabling the robot to navigate its environment accurately while interacting with specific objects like April Tags.

3 Hardware Design

The final structure of the robot (Figure 2) consists of the components mentioned below:

3.1 ESP32

The ESP32-wroom [3] is a highly versatile microcontroller from Espressif Systems, known for its strong processing power, integrated Wi-Fi and Bluetooth, and extensive GPIO support, making it ideal for IoT and smart control applications. It features a dual-core Xtensa 32-bit LX6 processor (up to 240 MHz), 520 KB SRAM, 4 MB Flash, and supports multiple communication protocols like I2C, SPI, and UART. Additionally, it offers 18 ADC channels, 2 DAC channels, and PWM signal generation for controlling devices such as motors or LEDs.

3.2 Power Supply

A 12V lithium-polymer battery is used as the robot's power source.

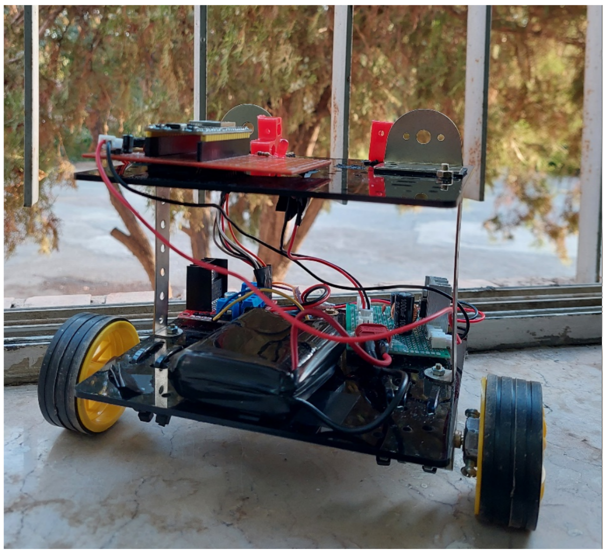


Figure 2: Final structure of the robot

3.3 Voltage Regulator

The voltage regulator employed in this project provides two output voltages: 5V and 12V. The 12V output powers the DC motors, while the 5V output is used for the ESP32 board. This regulator ensures that a single power source can provide the required voltages for both the motor and control circuit while protecting the circuit components from inappropriate voltage levels.

3.4 DC Motors

This project uses two 25GA model DC motors. These motors are ideal for the project due to their low weight, adequate torque, simple control over speed and direction, and affordability.

3.5 Motor Driver

An L298N motor driver module controls the DC motors. This dual-channel H-Bridge module allows for control over the speed and direction of two DC motors. It includes an L298N driver, protective diodes, and a 5V voltage regulator. A 12V power supply is provided to the driver module. The ENA and ENB pins enable or disable the motors, while speed is controlled via PWM signals.

3.6 Camera

A smartphone is mounted on top of the robot, acting as the camera for visual input. In our tests, we mainly used the Samsung Galaxy A73.

4 Software design

The software system consists of several components working together seamlessly. The ESP32 runs C++ code to control the motors and communicates with the Python script via HTTP requests. The SLAM process, also written in C++, uses the stella_vslam library for localization and mapping, with real-time visualization. The smartphone camera is connected to the laptop using the Droidcam app over a Wi-Fi network, where it functions as a webcam for video processing. The Python script detects April Tags, sends movement instructions to the ESP32, and reads from the shared "map.msg" file, which is updated by the SLAM process with the robot's current localization data. This shared file allows the Python script to calculate the April Tag positions relative to the robot. (Figure 3)

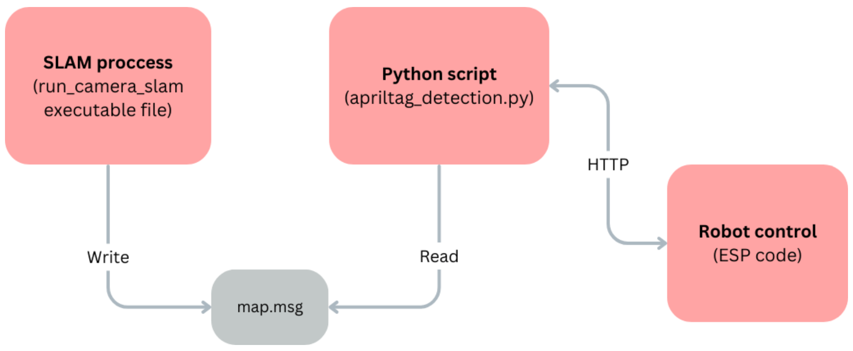


Figure 3: Software components and their communication

5 April tag detection

April Tags are visual markers designed for efficient real-time localization and identification. They are similar to QR codes but optimized for better detection and pose estimation. Each April Tag has a unique binary pattern, allowing it to be identified precisely in various lighting conditions or from different angles. These markers are handy for robotics applications requiring precise navigation and object tracking.

5.1 Detection Process and Communication

This project uses April Tags as reference points for autonomous navigation. The detection is handled using the Pupil-apriltag [2] library, integrated into a Python script running on the laptop. The library's Detector class is configured to detect the tag family tag36h11, which provides the best balance between accuracy and performance for this application. The camera parameters, including focal length and centre coordinates, are calibrated based on the specific smartphone model used for video capture.

When a frame is captured from the camera, the get_tags function processes it and extracts key information about any detected tags, including:

- **Tag ID:** The unique identifier for each tag.
- **Tag Pose:** This is represented by a transformation matrix (translation) and a rotation matrix. From these matrices, the relative position of the tag in 3D space (x, y, z) is determined, as well as the angle of the tag relative to the robot. These relative coordinates and angles are critical for guiding the robot's movements.

The Python script communicates with the ESP32 over the robot's dedicated WiFi network, sending HTTP requests that contain the detected tag's attributes: the distance in two dimensions. (The Z axis is ignored because the robot is always at the same height.) These attributes are continuously updated, allowing the ESP32 to adjust the robot's movement in real-time. This ensures precise alignment with the tag as the robot approaches.

Additionally, the system verifies the validity of the detected tag attributes, such as checking if the tag is within a feasible distance range before taking further actions. This helps to minimize both sensor and calculation errors. Once the robot has aligned itself and reached the tag, the absolute position of the tag is computed, taking into account the robot's current location and rotation. These computed locations are stored in a pickle file for future use, ensuring the environment is gradually mapped accurately.

5.2 April Tag Location Calculation

The absolute location of an April Tag can be determined by transforming its relative coordinates from the robot's local frame to the global coordinate system, which requires both the robot's position and its orientation (angle θ).

At the beginning of each run, the robot is positioned at the origin $(0, 0)$ and faces the X-axis. As the robot moves and rotates, its orientation changes, which must be accounted for when determining the tag's absolute position.

Let the vector **R2** represent the relative coordinates of the April Tag in the robot's local frame. These coordinates are calculated as if the robot were aligned with the global X-axis. However, since the robot is rotated by an angle θ, we must rotate this vector to align it with the global frame.

The transformation from **R2** to **R1** is performed using the following 2D rotation matrix:

$$\mathbf{R1} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \mathbf{R2}$$

Where **R1** represents the relative position of the April Tag in the global coordinate system, after considering the robot's rotation.

Once **R1** is obtained, the absolute position of the April Tag, denoted as $\mathbf{P_{tag}}$, is calculated by adding **R1** to the robot's current global position, $\mathbf{P_{robot}}$:

$$\mathbf{P_{tag}} = \mathbf{P_{robot}} + \mathbf{R1}$$

In this equation $\mathbf{P_{robot}}$ is the robot's global coordinates, and **R1** is the transformed vector that describes the distance and direction from the robot to the tag in the global coordinate system.

The image below illustrates this process. (Figure 4)

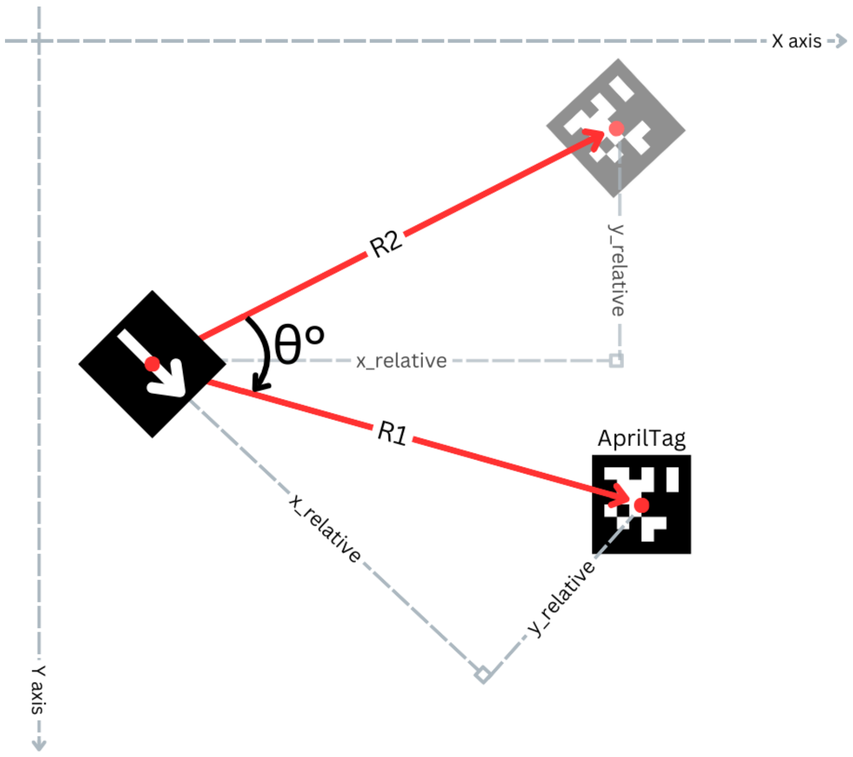


Figure 4: April Tag Location Calculation

6 Robot Control

6.1 Motor Control

The robot's motor control is based on a code sourced from a GitHub repository (Ujwal Nandanwar, 2022 [4]), which we adapted for this project. This code includes functions to control two motors (left and right) using GPIO pins to set the direction and speed of the robot. The "moveCar" function manages various movements, including forward, backward, left, right, and stop. Speed control is handled through PWM signals using a dedicated channel.

6.2 Web Interface

The web interface is also derived from the GitHub code. It provides a user-friendly platform for remote robot control through a web page hosted on the ESP32. The interface includes directional arrows and a speed slider, allowing real-time control via WebSocket communication (wsCarInput). Commands sent from the browser are processed by the ESP32, triggering the respective motor control functions. This setup enables easy interaction with the robot over a Wi-Fi network.

6.3 PD controller

Our robot control system heavily relies on the PD controller [?], a critical component that guides the robot to move towards and align with a detected tag. A crucial aspect of this process is the calculation of an error value to feed the PD controller. The primary challenge is to ensure the robot makes the correct rotational and forward adjustments, irrespective of the tag's distance. To address this, we have developed the formula $\sqrt{25 \cdot \frac{x}{d}}$ for error calculation, which considers both the horizontal offset (x) of the tag and its distance (d) from the robot. Let's delve into the rationale behind this formula by examining two example test cases.

Consider the following two scenarios where the tag is positioned at the same angle relative to the robot's camera:

Test Case 1: The tag is at a distance (d) of 100 cm with a horizontal offset (x) of 50 cm.

Test Case 2: The tag is much closer at a distance (d) of 25 cm, and the horizontal offset (x) is now 12.5 cm.

In both cases, the angle between the camera and the tag is the same, so the robot should ideally rotate by the same amount to face the tag. However, the range of possible x values increases proportionally when the tag is farther away (see Figure 5), causing potential over-rotation when the tag is farther and under-rotation when it is closer. To normalize the error, we make the error inversely proportional to d, reducing the error as distance increases. Through testing, we found that the relationship between x and the error resembles a square root curve, leading to the final formula:

$$error = \sqrt{25 \cdot \frac{x}{d}} \tag{1}$$

The coefficient 25 was fine-tuned through multiple test cases to ensure optimal performance.

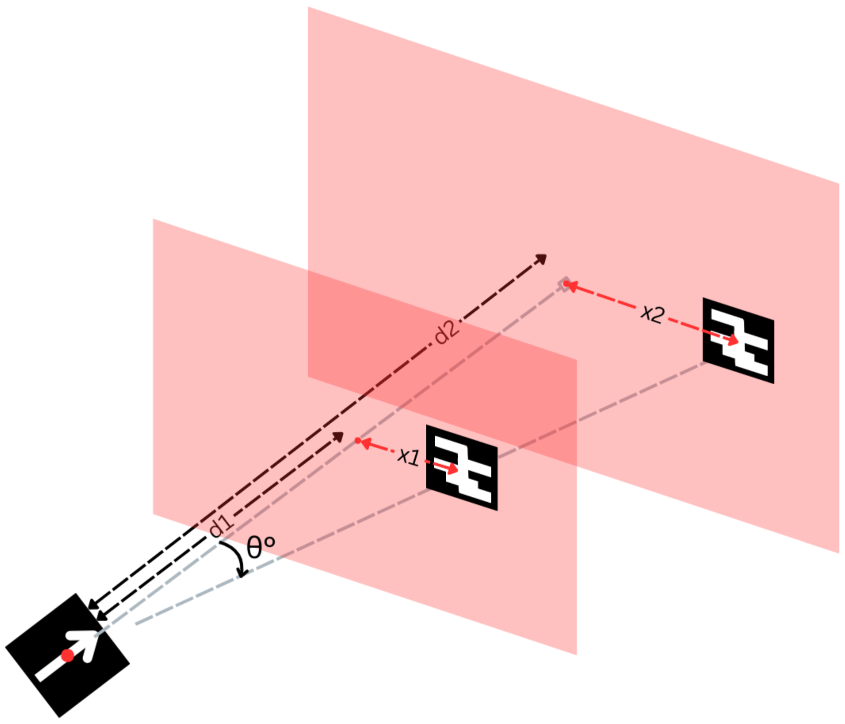


Figure 5: two tags at the same angle relative to the robot's camera, with different d and x

6.3.1 Moving Toward the Tag

When moving toward the tag, the PD controller computes the output as:

$$\text{output} = p \times \text{error} + d \times \Delta(\text{error}), \tag{2}$$

where error is calculated by (1) and p and d are the proportional and derivative coefficients, respectively. The robot adjusts its motor speeds to steer toward the tag by setting the left motor speed to offset − output and the right motor speed to offset + output. Where This differential speed control ensures that the robot remains directed toward the tag while moving forward.

6.3.2 Aligning with the Tag (Rotating in Place)

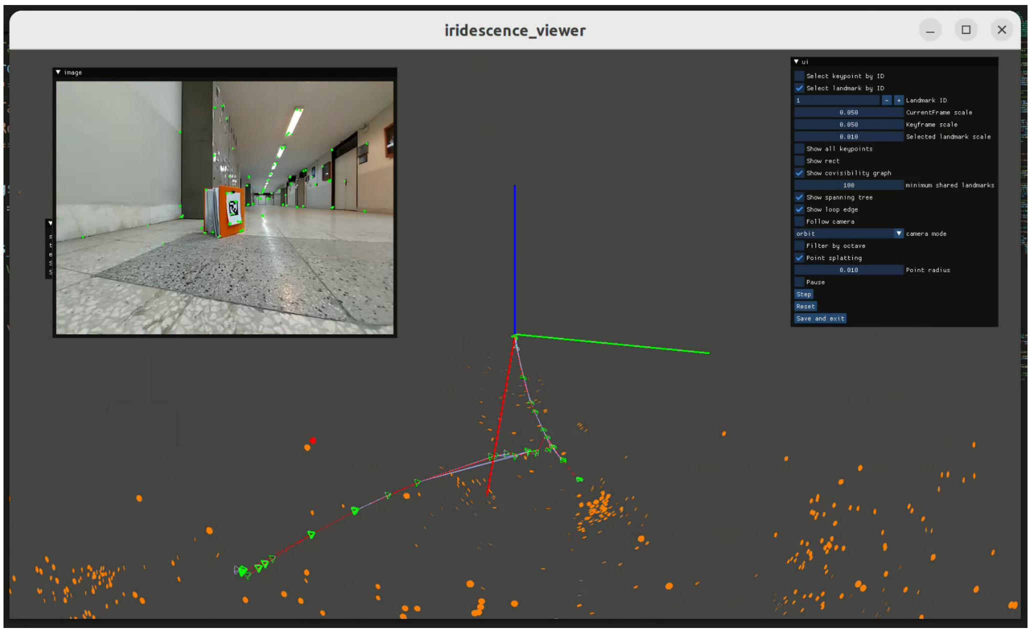
After moving toward the tag, there are cases where the robot gets too close to the tag and stops moving forward because it cannot proceed any further. However, at this point, the error may not be zero, indicating that the tag is not perfectly centered in the camera's frame. To address this, we use the `face_the_tag` function.

The `face_the_tag` function employs the same PD controller to adjust the robot's rotation. In this function, the motor speeds are set to −output for the left motor and +output for the right motor, causing the robot to rotate in place. This rotation aligns the tag's center with the camera's frame. Once the tag is centered, the robot stops rotating, ensuring that even when the robot can no longer move forward, it achieves precise alignment with the tag.

7 SLAM Integration

Simultaneous Localization and Mapping (SLAM) is a critical component of autonomous robotic systems, enabling a robot to navigate and map its environment simultaneously. SLAM works by using sensors to observe the surroundings and employing algorithms to build a map while estimating the robot's location within that map. This dual function of mapping and localization is essential for any robot to navigate dynamically without prior knowledge of the environment.

For this project, we chose the stella_vslam [1] library to handle SLAM. Stella_vslam is an efficient, versatile, and open-source library that supports various camera types, including monocular, stereo, and RGB-D. It provides real-time performance and is optimized for both desktop and embedded systems, making it an ideal choice for integrating SLAM with the robot's control system. Its robust mapping and localization algorithms allow for consistent and accurate mapping even in complex environments.



```
[ ] Select keypoint by ID
[✓] Select landmark by ID
1  - +  Landmark ID
0.050   CurrentFrame scale
0.050   Keyframe scale
0.010   Selected landmark scale
[ ] Show all keypoints
[ ] Show rect
[✓] Show covisibility graph
100     minimum shared landmarks
[✓] Show spanning tree
[✓] Show loop edge
[ ] Follow camera
orbit   camera mode
[ ] Filter by octave
[✓] Point splatting
0.010   Point radius
[ ] Pause
```

Figure 6: Example map generated by stella_vslam and viewed by iridescence Viewer in real-time

7.1 Why stella_vslam?

- **Real-Time Mapping:** Stella_vslam offers real-time mapping capabilities, crucial for our robot to update its map as it navigates. This allows the robot to adapt to dynamic environmental changes, ensuring effective navigation.

- **Flexibility:** The library is camera-agnostic, which means it supports a wide range of cameras, making it easier to integrate with the smartphone-based camera system used in this project.
- **Customizability:** Stella_vslam's open-source nature allows easy modification to suit specific project needs. We leveraged this flexibility to fine-tune the mapping process, ensuring the robot's location and map are consistently updated.

7.2 Implementation

In this project, we initially used the unmodified code from stella_vslam's example scripts to perform the SLAM tasks. However, to meet the requirement of real-time location retrieval for April Tag detection, we made a small yet significant modification: we added a thread that saves the map every 0.5 seconds.

This adjustment was implemented by introducing a secondary thread that periodically triggers the save_map() function in the SLAM code. By doing so, a snapshot of the current map is saved at regular intervals, allowing the Python script to retrieve the robot's current location from the map file. This information is currently used when the robot reaches an April Tag to calculate the tag's absolute position relative to the robot.

While this real-time map-saving is essential for accurate tag positioning at the moment, it also holds potential for future applications. For instance, it could be used to enhance autonomous navigation by enabling the robot to constantly refer to its updated map for path planning and decision-making. Without this periodic map-saving, the Python script would lack real-time access to the robot's position, limiting its ability to process the data effectively for both current and future developments.

To further streamline our process, we have developed a few Python scripts to retrieve information from the saved map.msg file and convert it into precise location data (in centimetres), used in our calculations. Additionally, we created scripts to visualize the robot's path and the detected April Tag locations, providing a clearer understanding of the robot's movement and interaction with its environment.

By using stella_vslam and implementing these modifications, we ensure that the system remains flexible and ready for future enhancements while efficiently supporting the project's current requirements.

8 Testing, Challenges, and Results

One of the significant challenges we faced in this project was the use of a smartphone camera for video input. While it provided a cost-effective solution, it was a monocular setup, not stereo, which led to notable distortion and noise in the images. This affected the accuracy of the VSLAM map generated during the robot's navigation. To mitigate this issue, we conducted multiple camera calibrations using a chessboard pattern to obtain accurate intrinsic and extrinsic matrices. Despite the calibrations, due to the camera's limitations, we still encountered challenges with the accuracy of the map, requiring several recalibrations during the testing phase.

Another challenge arose from the robot's low height, which meant that nearly half of the camera's input image consisted of the ground surface. When the ground had no significant texture or pattern, such as in environments with ceramic tiles or asphalt, there were too few landmarks for the VSLAM algorithm to track. This resulted in a reduced mapping accuracy, where the robot's actual movement of 50 cm could be perceived as only 10 cm by the VSLAM system. To address this, we conducted tests using patterned cloth beneath the robot, which significantly improved the accuracy, yielding much closer estimates to the robot's actual movement distances. These adjustments highlight the importance of proper calibration and environmental factors in achieving accurate SLAM performance.

8.1 Final Map Generation

After executing SLAM and detecting the positions of the tags along the path, a final map is generated, as shown in Figure 7. The blue line represents the robot's trajectory, with light blue indicating the starting point and dark blue showing the end. The red points mark the positions of the tags in the real-world environment. The video of this test is included in the project's GitHub repository.

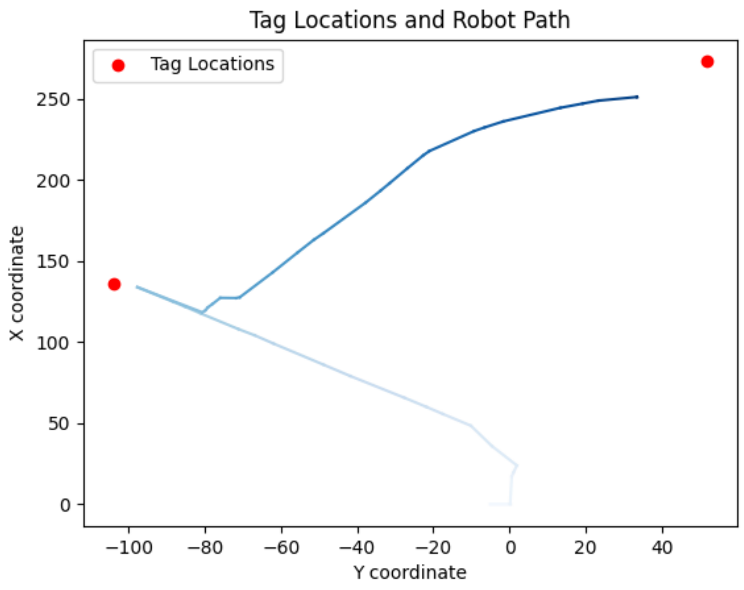


Figure 7: An example of final map showing the robot's trajectory and tag locations

9 Conclusion and Future Work

In this project, we successfully developed a hybrid robot capable of remote and autonomous navigation, utilizing SLAM and AprilTag detection for real-time localization and mapping. The integration of the ESP32 microcontroller, smartphone camera, and Python-based AprilTag detection allowed the robot to navigate efficiently in dynamic environments. By leveraging the Stella_vslam library for SLAM, we generated accurate maps of the robot's surroundings and effectively positioned AprilTags within the environment.

Our testing demonstrated that the system is reliable under various lighting conditions, with an acceptable margin of error for both SLAM-based localization and AprilTag position estimation. The experiments showed that natural lighting produced more accurate results than fluorescent lighting, and that the ultrawide camera outperformed the wide camera in lower average error. The PID-controlled navigation system also ensured the robot could align with and reach specific AprilTags autonomously.

Future work could focus on optimizing the PID controller for smoother transitions and more precise tag alignment, especially over longer distances. Additionally, incorporating other sensor modalities, such as LiDAR, could enhance the robot's mapping and localization accuracy. Extending the system to handle more complex environments with dynamic obstacles would also be a valuable improvement. This project lays a solid foundation for further developing hybrid remote-autonomous robotic systems.

References

[1] Yamashita, M., and Furukawa, Y., "Stella-VSLAM: A Versatile and Accurate SLAM Library for Monocular, Stereo, and RGB-D Cameras," *GitHub Repository*, 2020. Available at: `https://github.com/stella-cv/stella_vslam`.

[2] Kaess, M., and Sommerlade, E., "Pupil-Apriltags: Robust Fiducial Markers for Computer Vision," *GitHub Repository*, 2016. Available at: `https://github.com/pupil-labs/apriltags`.

[3] Espressif Systems, "ESP32-WROOM: A Wi-Fi and Bluetooth-enabled System-on-Chip," Espressif Technical Documentation, 2018. Available at: `https://www.espressif.com/en/products/socs/esp32`.

[4] Ujwal Nandanwar, "WifiCar" *GitHub Repository*, 2021. Available at: `https://github.com/un0038998/WiFiCar`.