# Decision Making for Maze Navigation

*Marklov Decision Process

1st Nádia Gonçalves
*Departamento de Engenharia Mecânica(DEM)*
*Instituto Superior Técnico*
Lisboa, Portugal
nadia.julia.goncalves@tecnico.ulisboa.pt

2nd Filipa Cunha
*Dept. de Eng. Eletrotécnica e de Computadores (DEEC)*
*Instituto Superior Técnico*
Lisboa, Portugal
filipa.cruz.cunha@tecnico.ulisboa.pt

3rd Joel Graça
*Dept. de Eng. Eletrotécnica e de Computadores (DEEC)*
*Instituto Superior Técnico*
Lisboa, Portugal
joelgraca@tecnico.ulisboa.pt

4th João Ferreira
*Dept. de Eng. Eletrotécnica e de Computadores (DEEC)*
*Instituto Superior Técnico*
Lisboa, Portugal
joao.l.n.ferreira@tecnico.ulisboa.pt

***Abstract*—This study presents a ROS-based architecture is designed to enable a robot to navigate mazes autonomously using visual markers and interchangeable planning methods. The navigation strategies are interchangeable, supporting both Value Iteration and Q-learning approaches for path planing, and the software architecture separates perception, planing and control, ensuring robust performance and a straightforward adaptation to new algorithms or environments.**

***Index Terms*—Alphabot2, Value Iteration, Maze navigation, Reinforcement learning, ROS, Fiducial Markers for position perception,**

## I. Introduction

As its known, autonomous navigation in multipath environments remains a fundamental challenge in robotics, demanding for effective perception, planing and control. In this work, it has been addressed the challenge by developing a robotic system capable of solving a maze through the use of visual markers for localization. This study introduces a maze-solving robot system that supports both Markov Decision Process and Reinforcement learning for path planing. Built using ROS, the modular structure of the system enables seamless switching between algorithms, supporting flexible development and comparative analysis.

## II. Methods and Algorithms

### *A. Markov Decision Process (MDP)*

A Markov Decision Process (MDP) is mathematical framework used to model decision problems where the outcome is uncertain [3]. It is defined by:

- **States (s)**: each possible position of the agent (robot). In our project, the states correspond to the cells in the maze.
- **Actions (a)**: the robot can move in four directions (up, down, left, right).
- **Transition Function (P)**: given that each movement has some uncertainty, $P(s'|s,a)$ defines the probability of reaching state $s'$ after taking action $a$ in state $s$.
- **Reward Function**: feedback for each state or state-action pair (e.g., for reaching the end goal/state).
- **Policy** ($\pi$): defines the action that the agent should take in each state. ($\pi : S \rightarrow A$)

MDPs assume additive rewards, meaning that the total reward is the sum of rewards collected over time. Consider the discount factor $\gamma$, which discounts future rewards. If $\gamma \approx 0$, it only values more immediate rewards; whereas with $\gamma \approx 1$, it values future rewards.

There are two main algorithms to find the best movement policy, value iteration [6] and policy [5] iteration, which yield the same strategy through different means.

- **Value Iteration**: works best for large state spaces, and improves policy earlier/quicker.
  1) Starts with random guesses for cell values.
  2) Updates the cell value based on neighbouring cells.
  3) Improves the estimates until they stop changing.
  4) Extracts the best movement direction for each cell.
- **Policy Iteration**: works best for small state/action spaces, and when the discount factor $\gamma \approx 1$.
  1) Starts with a random movement policy.
  2) Evaluates how good the current strategy is, and makes small improvements to the policy.
  3) Repeats until the policy stops improving.
  4) Extracts the best action for each cell.

**Considerations**: Full knowledge of the map and transitions is assumed, thus the robot does not need to explore blindly (since it already knows the environment). Walls are thought of as barriers between states; if an action would move the robot through a wall, the robot remains in the current state. Unique fiducial markers (ArUco) to be placed at key locations (e.g. intersections, turns). This enables the robot to re-align itself (re-localization) and resume from a known state (a necessary condition for the MDP).

## *B. Reinforcement Learning (RL)*

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by interacting with the environment. The agent takes actions in various states to maximize cumulative reward over time. It is defined similarly to MDPs, however, it does not require knowledge of transition probabilities [2], reward model, nor of the environment (which can be unknown or dynamic). Unlike MDPs, RL continuously learns and adapts to environmental changes, and is thus well-suited for real-world scenarios. Its learning cycle is:

1) It selects and performs an action;
2) The environment (maze) provides feedback (through rewards or penalties);
3) The agent updates its strategy to increase future rewards.

**Q-learning** is a type of reinforcement learning algorithm that helps an agent learn the best actions to take in different situations. It estimates a **Q-value**, $Q(s, a)$, which represents the expected cumulative reward of taking action a in state s, and then following the best possible actions. It learns the optimal policy independently of the agent's actions (i.e. it can explore randomly but still learn optimal behaviour).

## *C. Fiducial Marker-based Location*

Fiducial markers (ArUco codes) play a central role in achieving localization throughout this project. Each marker is uniquely identified and strategically placed within the maze [11]. Upon detection, the robot can immediately determine its precise location within the maze grid, which is critical for both MDP-based and reinforcement learning navigation. In implementation, marker detection events initiate state transitions within the finite state machine of the Maze Solver Node. For example, when a marker is detected, the robot pauses, updates its internal state to the corresponding grid cell, and then resumes path execution or planning from that known position. This mechanism not only enables robust navigation but also provides resilience to drift and accumulated error, if the robot loses track of its position, it can recover by searching for the nearest marker [8].

## III. Implementation

The implementation integrates a set of subsystems that operate in real-time to achieve autonomous navigation through marker detection. And its architecture is defined in a modular design pattern as illustrated in Fig.1.

## *A. Motor Control Architecture and Modular Integration*

Our system adopts a modular, distributed ROS architecture comprising three main components: the **Maze Solver Node**, the **Motor Driver Node**, and the **Alphabot Hardware Layer**. High-level navigation and path planning are handled by the MazeSolverNode, which subscribes to camera images and publishes velocity commands. Two interchangeable planning solutions are supported: one based on Markov Decision Processes (MDP) with value iteration, and another using reinforcement learning (Q-learning).

The Motor Driver Node receives velocity commands from the MazeSolverNode, translates them into hardware-specific motor signals, manages smooth acceleration profiles, synchronizes timing, and monitors safety. The Alphabot hardware abstraction layer provides direct motor and sensor interfacing, PWM control, encoder integration, and platform-specific calibration and safety features.

All communication is implemented via ROS topics using a publish-subscribe paradigm , ensuring loose coupling, process isolation, and easy adaptation to other platforms. The CvBridge utility enables seamless conversion between ROS image messages and OpenCV matrices for perception. Motion parameters and timing are precisely tuned for the Alphabot platform, ensuring accurate, robust, and safe movement.

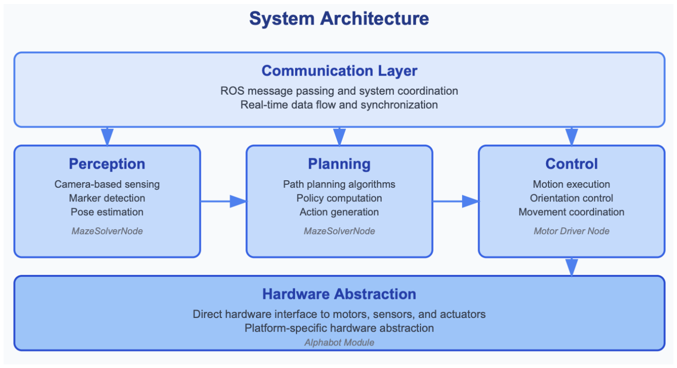


Fig. 1: General Architecture

## *B. System architecture*

For a more detailed explanation about the system implementation it is possible to assume that the system architecture is structures in five distinct layers, where each has particular assignments:

- Perception layer: leverages the Raspberry Pi camera and ROS image topics within the Maze Solver Node for environmental sensing, through marker detection and pose estimation.
- Planning layer: runs path-planing algorithms, the value iteration in the first phase, to determine optimal navigation strategies and generate action sequences, all within the Maze Solver Node.
- Control layer: which was implemented in the motor_driver_node.py code, manages the robot movement by translating the high-level commands into low-level motor control signals through the alphabot interface.
- Hardware abstraction layer: It is provided by the Alphabot.py, this layer abstracts hardware-specific details and supplies standardized control over the motors, sensors, and actuators of the robot.
- Communication layer: Enables real-time message exchange and synchronization between all ROS nodes,

ensuring cohesive and coordinated operation of the entire system.

The system represents the maze as a discrete 4×5 grid, where each cell is directly mapped to a specific physical location in the environment. In this encoding, the free space (value 0) and the goal (value 100) are treated as valid states that the agent can occupy or plan through. Cells assigned a value of -100 represent obstacles or walls and are explicitly excluded from the set of valid states; these locations cannot be entered or traversed by the agent during planning or execution. This approach to discretization supports efficient and reliable path planning, while ensuring that only navigable and goal cells are considered part of the agent's state space.

In this project, there were two MazeSolverNodes, the first one uses the Marklov Decision Process Value Iteration to compute the optimal policy, and the second one uses the Q-learning, a Reinforcement Learning algorithm2.

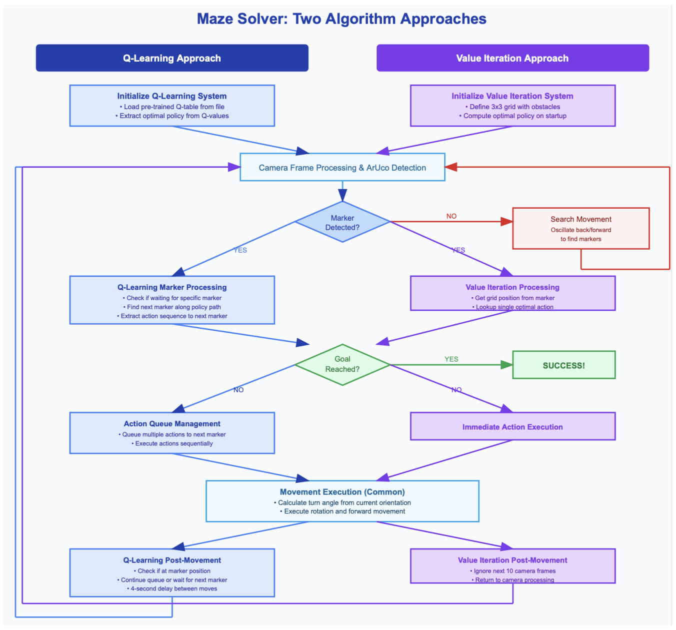

Maze Solver: Two Algorithm Approaches
Q-Learning Approach
Value Iteration Approach
Initialize Q-Learning System
• Load pre-trained Q-table from file
• Extract optimal policy from Q-values
Initialize Value Iteration System
• Define 3x3 grid with obstacles
• Compute optimal policy on startup
Camera Frame Processing & ArUco Detection
Marker Detected?
NO
Search Movement
Oscillate back/forward to find markers
YES
YES
Q-Learning Marker Processing
• Check if waiting for specific marker
• Find next marker along policy path
• Extract action sequence to next marker
Value Iteration Processing
• Get grid position from marker
• Lookup single optimal action
Goal Reached?
YES
SUCCESS!
NO
NO
Action Queue Management
• Queue multiple actions to next marker
• Execute actions sequentially
Immediate Action Execution
Movement Execution (Common)
• Calculate turn angle from current orientation
• Execute rotation and forward movement
Q-Learning Post-Movement
• Check if at marker position
• Continue queue or wait for next marker
• 4-second delay between moves
Value Iteration Post-Movement
• Ignore next 10 camera frames
• Return to camera processing

Fig. 2: Main Algorithm Flowchart

## *C. Computer Vision and Marker-Based Localization*

To stream images from the Raspberry Pi camera on the AlphaBot2, `raspicam_node` package was used. A custom launch file `camerav2_320x240.launch` was created to operate the camera at 320×240 resolution with 30 FPS, providing a smoother frame rate for real-time detection tasks.

**ROS packages used:**

- `raspicam_node`
- `image_transport`
- `compressed_image_transport`
- `cv_bridge`
- `camera_info_manager`
- `rqt_image_view`

To enable OpenCV compatibility, the compressed image stream was republished as raw using:

```
rosrun image_transport republish compressed \
  in:=/raspicam_node/image \
  raw out:=/raspicam_node/image_raw
```

**Camera calibration** was performed using a printed checkerboard and the ROS `camera_calibration` tool. The generated `ost.yaml` file, located in `~/ros/raspicam_calibrations/calib_320x240/`, contains the camera matrix and distortion coefficients. These parameters were essential for accurate pose estimation of the ArUco markers.

Even though there was provided and followed a wiki guide on how to calibrate the camera it was still a slow purpose and changed from alphabot to alphabot which led to the conclusion that the hardware itself was a condition for the calibration. After four different robots we were able to conduct the process and successfully calibrate our robot and extract the corresponding calibration file. Another counter-time was where to launch the camera. When connected directly with the alphabot, the camera speed was very slow and the calibration rarely succeeded, and if so it took more than twenty minutes.

A custom ROS node, `aruco_node.py`, was developed in Python as part of the `aruco_detector` package. This node subscribes to the compressed camera stream, performs ArUco marker detection using OpenCV's `cv2.aruco` module, and publishes both annotated images and detected marker IDs.

*1) ArUco Marker Detection Node:* We implemented a custom ROS node in Python (`aruco_node.py`), part of the `aruco_detector` package. This script subscribes to the compressed camera stream, detects ArUco markers using OpenCV's `cv2.aruco` module, and publishes annotated images and marker IDs.A custom ROS node, `aruco_node.py`, was developed in Python as part of the `aruco_detector` package. This node subscribes to the compressed camera stream, performs ArUco marker detection using OpenCV's `cv2.aruco` module, and publishes both annotated images and detected marker IDs.

**Dependencies:**

- `rospy`
- `cv2.aruco`
- `cv_bridge`
- `sensor_msgs`, `std_msgs`, `image_transport`

**Published topics:**

- `/aruco/image_annotated` – image with detected markers and pose axes
- `/aruco/marker_ids` – list of detected marker IDs

The node loads the camera calibration parameters from the selected `ost.yaml` file and prints the distance and relative position of each detected marker at the terminal.

*2) Rosbag Recording:* For post-processing and evaluation, the key topics were recorded using `rosbag`:

```
rosbag record -O test_3_corridor.bag \
  /raspicam_node/image/compressed \
  /raspicam_node/camera_info \
  /aruco/image_annotated \
  /aruco/marker_ids
```

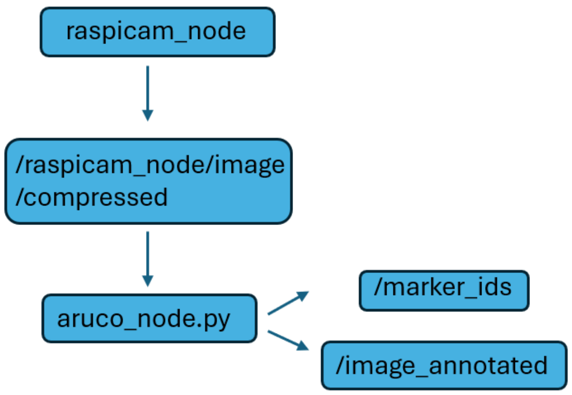


Fig. 3: ROS node diagram showing the flow from camera stream to ArUco detection and visualization.

## D. Path Planning and Decision Making

The navigation system implements two reinforcement learning approaches: model-based value iteration and model-free Q-learning. Both methods utilize a discretized grid world, $4 \times 5$, with shared execution frameworks, but differ in decision-making mechanisms.

*1) Value Iteration Implementation:* The primary approach uses value iteration with dynamic programming:

$$V_{k+1}(s) = \max_a \left[ R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right] \quad (1)$$

Key parameters: discount factor $\gamma = 0.9$, convergence threshold $\theta = 0.01$. The algorithm guarantees optimality through:

- **Policy Extraction**;
- **Path Segmentation**.

*2) Q-Learning Implementation:* The alternative model-free approach employs temporal difference learning:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \quad (2)$$

Implementation features:

- Q-table dimensions: rows $\times$ cols $\times$ $|\mathcal{A}|$
- Persistent storage for cross-episode learning

Listing 1: Q-table Management

```
def load_q_table(filename):
    try:
        Q = np.load(filename)
        return Q
    except Exception as e:
        return np.zeros((rows, cols, len(
            ACTIONS)))
```

| Attribute | Value Iteration | Q-Learning |
|---|---|---|
| Model Needed | Yes | No |
| Convergence | Fast | Slower |
| Optimality | Guaranteed | Asymptotic |
| Adaptability | Static | Dynamic |

TABLE I: Comparison of Value Iteration and Q-Learning Approaches

## E. State Management and Search Strategies

State management is implemented as a hierarchical finite state machine within the `MazeSolverNode`, coordinating high-level navigation and low-level execution using ROS. The main states are the following:

- **Marker Detection:** The node subscribes to `/raspicam_node/image_raw` (`sensor_msgs/Image`) and uses `cv_bridge` and OpenCV ArUco detection to locate the robot. Upon successful marker detection, the system transitions to path execution.
- **Path Execution:** The node publishes velocity commands to `/cmd_vel` (`geometry_msgs/TwistStamped`), executing planned action sequences. At each marker, the robot pauses to verify the location before proceeding.
- **Search:** If no marker is detected, a search routine is activated, alternating small forward/backward moves (`TwistStamped` commands with 0.01 m step) and 10 s dwell times to maximize detection chances.
- **Error Recovery:** The system handles detection or motion errors by re-entering the search state, clearing frame buffers, and reinitializing relevant state variables.

This approach leverages ROS topics, message types, and standard packages (`rospy`, `cv_bridge`, `sensor_msgs`, `geometry_msgs`) to ensure robust and modular coordination between perception, planning, and control.

## F. Real-Time Performance Considerations

To ensure real-time operation, the system integrates several ROS-based optimizations:

- **Efficient Planning:** Value iteration (MDP) and Q-learning policies are precomputed offline, eliminating online planning delays during navigation.
- **Memory Management:** Action sequences are stored using Python `deque` structures, minimizing dynamic memory allocation overhead (`collections.deque`).
- **Frame Rate Optimization:** The image callback in `MazeSolverNode` selectively ignores frames using an `ignore_counter`, reducing computational load from `/raspicam_node/image_raw` (`sensor_msgs/Image`) while maintaining reliable ArUco marker detection via `cv_bridge` and OpenCV.
- **Temporal Consistency:** Multi-frame averaging and outlier rejection are applied to marker detection results, improving localization stability.

These strategies leverage ROS packages such as `rospy`, `cv_bridge`, `sensor_msgs`, and `geometry_msgs`, and ensure that the system maintains low latency and robust performance during autonomous operation.

## IV. Experimental Results

### A. Performance Evaluation and Experimental Conditions

*1) ArUco Detection:* To evaluate the robustness of our ArUco marker detection system, testing was performed under a different conditions. The goal was to analyze how key variables affected detection performance and system responsiveness.

**The main variables tested were:**

- Ambient lighting conditions
- Camera resolution
- Marker size
- Distance between the marker and the camera

The system was tested in both optimal lighting and reduced lighting scenarios — down to approximately 20% of the maximum light intensity. This was done to simulate realistic conditions, as the lighting within the maze is not always ideal or uniform.

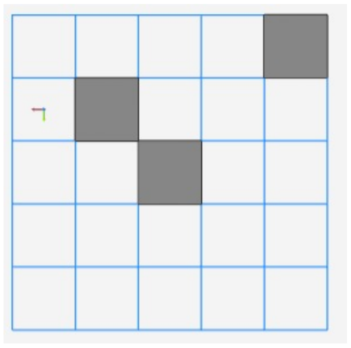
The detection latency, defined as the delay between the marker appearing in the camera frame and its processing by the robot, ranged from approximately 0.5 to 1 seconds.

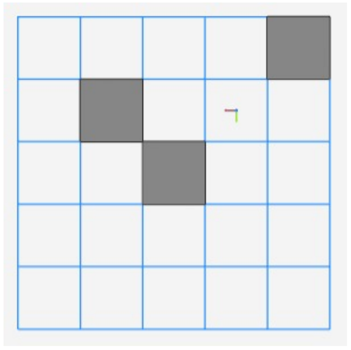
Additional tests involved varying the marker sizes. The system was able to detect ArUco markers ranging from **X mm to 20 mm**. Based on the dimensions of our maze and the practical visibility constraints, we selected markers of **20 mm** side length (ID area), which provided the most reliable detection at different distances.

*2) Simulation:* The robot demonstrated reliable and consistent behavior in the simulated environment. In simulation, parameters such as the step size (e.g., 0.15 m) could be set precisely, resulting in accurate and repeatable motion along planned paths. Marker detection and localization were robust, and the robot was able to follow the computed policy with minimal deviation.

*3) Real life application:* Transferring the system from simulation to the physical Alphabot platform revealed significant practical challenges. Although a step size parameter of 0.0015 m was configured to match the intended movement, the actual displacement of the robot was highly sensitive to the battery level. When the battery was fully charged, even a small step command resulted in the robot moving as much as 0.5 m, far exceeding expectations. This made the parameter-tuning process largely trial and error, as the same settings could produce vastly different behaviours depending on the battery state. Furthermore, the presence of rough or uneven surfaces further interfered with the robot's motion accuracy, compounding the difficulty of achieving consistent performance. These factors highlight the gap between simulated and real-world operation and underscore the importance of iterative testing and adjustment when deploying robotic systems in physical environments.

In addition, real-world conditions such as uneven or rough surfaces further affected the robot's motion accuracy. Variations in surface friction and small obstacles could cause slippage or deviations from the intended path, reducing the reliability observed in simulation. As a result, parameter tuning in the real environment became an iterative process, with adjustments needed to account for both hardware variability and environmental factors. These findings highlight the importance of considering physical constraints and uncertainties when deploying robotic systems outside of simulation.


(a) Initial Location


(b) Middle Location

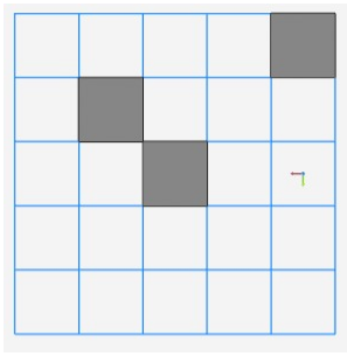

(c) Goal reached

Fig. 4: Maze Solver Simulation on FoxGlove

## V. Conclusion

The project successfully delivered a flexible and extensible ROS-based navigation platform, validated both in simulation and on hardware. The results provide a strong foundation for future research, including the integration of online policy adaptation and robust navigation in partially observable or dynamically changing environments. This work not only demonstrates the practical potential of combining computer vision with advanced planning in robotics, but also highlights the critical importance of real-world experimentation in the development of autonomous systems.

## References

[1] Kons-5. *ROS-for-the-Alphabot2: Setting up, configuring, and programming the Alphabot2 with ROS.* GitHub repository, atualizado em 22 de julho de 2024. Disponível em: https://github.com/Kons-5/ROS-for-the-Alphabot2

[2] Data Science Central. *Reinforcement learning explained: Overview, comparisons, and use cases*. March 15, 2022. Disponível em: https://www.datasciencecentral.com/reinforcement-learning-explained-overview-comparisons-and/

[3] Stuart Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[4] Nguyen Gao. *Markov Decision Process - The Basics*. 2020. Disponível em: https://medium.com/@ngao7/markov-decision-process-basics-3da5144d3348

[5] Nguyen Gao. *Markov Decision Process with Policy Iteration*. 2020. Disponível em: https://medium.com/@ngao7/markov-decision-process-policy-iteration-42d35ee87c82

[6] Nguyen Gao. *Markov Decision Process with Value Iteration*. 2020. Disponível em: https://medium.com/@ngao7/markov-decision-process-value-iteration-2d161d50a6ff

[7] Bischoff, B., Nguyen-Tuong, D., Streichert, F., Ewert, M., e Knoll, A. *Fusing vision and odometry for accurate indoor robot localization*. In: *Proc. of the 2012 12th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, Guangzhou, China, Dez. 2012.

[8] Ubiquity Robotics. *Fiducial Marker Based Localization System - Package Announcement*. 2018. Disponível em: https://discourse.ros.org/t/fiducial-marker-based-localization-system-package-annoucement/4050

[9] Instituto de Sistemas e Robótica – ISR Lisboa. *Autonomous Systems Resources*. Disponível em: https://mediawiki.isr.tecnico.ulisboa.pt/wiki/Autonomous_Systems_resources#Simulated_robots

[10] ROS Wiki. *SLAM with visual markers in ROS*. Disponível em: https://wiki.ros.org/fiducial_slam

[11] ROS Wiki. *ArUco Marker Detection in ROS*. Disponível em: https://wiki.ros.org/aruco_detect

[12] ROS Wiki. *Monocular Camera Calibration Tutorial*. Disponível em: https://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration

[13] Arduino Forum. *Moving and Rotating AlphaBot by Specific Distance and Angle*. Disponível em: https://forum.arduino.cc/t/moving-rotating-alphabot-by-specific-distance-angle/1311231