GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN
DRITTES PHYSIKALISCHES INSTITUT

Practical Course on Computer Vision and Robotics

Tommaso Bertola, 25544550
Marco Martinico, 27144566
Roberto Togni, 23340265

Course: *Practical Course on Computer Vision & Robotics*
Lecturers: *Sebastian Ruiz, Lennart Jahn*
Due date: *15.02.2024*

Write 3-6 pages (excluding appendix) for the following sections. Complete the table in **??** and attach interesting pieces of code in **??**.

1 Summary

The project involves the construction of a two-wheeled robot using Raspberry Pi for computations and Lego Mindstorms for sensor and movement control. The robot employs reactive control to explore a track, utilizing an application of Extended Kalman Filter (EKF) to the Simultaneous Localization And Mapping (SLAM) algorithm to create an accurate map of the track. Subsequently, the A* algorithm is applied to compute the optimal path, which the robot then follows. To ensure precision in path following, a PID controller has been implemented, thus preventing the robot from deviating from the calculated path.

Throughout the project implementation, we faced various challenges, particularly in the following areas:

- Managing the transition between different reference systems;
- Developing a robust algorithm for reactive control;
- Implementing the EKF_SLAM algorithm;
- Processing the map effectively for use in the A* algorithm.

2 Method

2.1 Computer Vision

The computer vision task was quite complex, as it involved several aspects. For simplicity, we separated it into several tasks. The code used for detecting discs is detailed in **??**.

2.1.1 ArUco markers detection

To detect the ArUco markers, we used the functions provided by OpenCV. In particular, we used the `estimatePoseSingleMarkers()` method to obtain the traslation and rotation vectors (`tvecs` and `rvecs`, respectively). We then used the values obtained to construct the transformation matrix from the camera to the markers. To obtain the marker positions relative to the robot, we concatenated (i.e. computed the dot product) the transform matrices as shown in **??**:

$$T_{\text{robot,marker}} = T_{\text{robot,camera}} \cdot T_{\text{camera,marker}} \tag{1}$$

2.1.2 Discs detection

To identify red and green discs, we concatenated several image processing techniques within the `find_centroids()` method. First, the input image is converted into the HSV colour space. A colour

mask and bitwise operations are then used to identify the red or green areas of the image (we used a range to account for possible artifacts due to illumination and distance from the discs).

We then converted the resulting image to grayscale, applied Gaussian blur, and used a threshold to create a binary image. Morphological operations (opening and closing) are then applied to refine the binary image. The final step utilises connected components analysis to label distinct regions and computes centroids for each labelled component. The function returns the number of labelled components, centroids (x, y coordinates), and statistics about each labelled region.

Since the coordinates are with respect to the two-dimensional image, the `detect_circle()` function utilises the `img_to_world()` function for transforming image points into corresponding 3D world coordinates.

2.2 Exploration - Reactive Driving

The exploration task entails the robot traversing the path twice, with the inner side followed in the first round and the outer side in the second. This sequential approach contributes to the production of a relatively accurate map by the Extended Kalman Filter (EKF) Simultaneous Localization and Mapping (SLAM) algorithm.

The algorithm we used to follow either side is detailed in **??**. The underlying logic is straightforward. A variable is employed to determine whether the robot should follow the inner or outer side. Based on this determination, the landmark IDs are filtered to exclusively consider either the outer or inner landmarks. If there are at least two filtered landmarks, the robot adjusts its speed and computes the relative position of these landmarks. The latter is utilized to determine the angle (`theta`) between the two landmarks. The robot then sets the `turn` parameter to control its steering. Since the relationship between angle and `turn` is not linear, some conditional statements are used to set the `turn` value. Additional checks are performed to ensure that the distance to the nearest ArUco markers falls within a specified range. This step proved particularly helpful during the early stages of the task.

If the robot fails to detect at least two ArUco markers, a warning message is printed, and the robot starts rotating in the direction of the side being followed. The speed at this stage is reduced slightly to facilitate ArUco recognition.

2.3 Mapping

Since A* takes as input a graph and two points (start and end), we had to process the map obtained by SLAM. In order to do this, we decided to represent the map by means of a matrix, in effect discretizing the environment. The code used to switch between matrix and real-world coordinates is listed in **??**.

First, we separated outer track, inner track, obstacles and starting line into several arrays. We then processed these to join the individual points by means of a polygon. At this point we switched from real coordinates to indices in the array, maintaining a resolution of 5 cm.

We then used the previously obtained polygons to create a binary mask representing the map. Specifically, we assigned a value of 1 to all non-steppable regions (i.e., outside the outer edge and inside the inner edge), and a value of 0 to all steppable regions (i.e., the track). We also performed a dilation on the inner region in order to prevent the pathfinding algorithm from returning a path excessively close to the inner edge.

Finally, we added the starting line and obstacles to the binary mask, assigning a value of 1 to both. The addition of the starting line proved particularly effective in the context of the pathfinding algorithm, effectively making the choice of heuristics much more trivial.

2.4 Pathfinding

The map obtained during exploration is processed in order to use it as input for the A* algorithm (see **??**). We opted for this approach for several reasons. First of all, A* guarantees finding the shortest path (if one exists). Secondly, it is more efficient than Dijkstra's algorithm thanks to the heuristic component.

Since A* requires a starting and an ending point, we had to figure out how to obtain them. To do so, we adopted the following approach: We started by finding the extremes of the start line in the grid; we then converted them to real-world coordinates and computed two circumferences centered in the two points and with a radius equal to three quarters of the distance between them; finally, we found the two points of intersection of the two circumferences and used them as starting and ending points.

2.5 Path Following

After obtaining the optimal path via A*, we implemented the logic to follow it. Due to motor uncertainty and wheel slippage, a blind following approach is not feasible. We therefore implemented a PID controller in order to correct the movement in real time and ensure that the robot follows the planned path.

The main problem we encountered in this context turned out to be the execution time of the code. Since the latter is often around 0.5 seconds, the PID correction suffers from an inherent delay, which results in poor performance. To mitigate this problem, we have rewritten several parts of the code in order to reduce latencies as much as possible. This resulted in improved driving, which allowed us to better fine-tune the PID controller.

For the fine-tuning procedure, we followed the following approach. We set all constants to 0, then increased K_p until we observed oscillations in the output. At this point we halved the value of K_p and increased that of K_i until we observed overshoot. We then started increasing K_d. At the end of the procedure, we set the values of the three constants to 32, 2 and 0 respectively.

2.6 Optimisation

Much of the optimisation was focused on reducing code execution times (see **??** and **??**). In particular, we tried to speed up the matrix products performed in the context of the EKF_SLAM. To do this, we rewrote some functions using the just-in-time compilation service offered by numba. Given the incomplete support for numpy and the conflicts due to the use of OOP, we had to carry out considerable refactoring of the code. However, this approach proved effective, reducing latencies and allowing us to optimise the path-following task.

3 Results

Our robot showed good results in the context of reactive control with respect to the outer side of the track. The discretized map, with a grid element size of 5 cm, provided a reliable basis for the A* pathfinding algorithm to calculate the optimal route.

However, during the exploration phase on the inner side of the track, the robot exhibited difficulties in handling tight bends. This issue likely arose from the limitations of our reactive control approach, which may not have been sufficiently robust for the rapid changes in curvature and direction required in these sections.

4 Discussion

One of the main problems we encountered was the execution time of the code, which introduced a substantial delay in the execution of the various tasks (see **??** for an overview of the execution times). One of the parts most affected by this problem is the EKF SLAM, whose execution time becomes problematic as the number of landmarks increases. To try to mitigate this problem, we decided to perform the SLAM:

- On two-thirds of the landmarks seen during exploration (`fastmode` not active)
- On the (at most) 3 closest bowls during the run (`fastmode` active)

Since this did not prove sufficient, we used numba to boost performance for computationally heavy sections as described in **??**.

A second problem we faced was the ordering of the ArUcos, necessary to perform the transition between a continuous representation of the environment and a discretized one. In particular, we

had to develop a mechanism to order the mapped ArUcos so that they could be used as vertices to draw a polygon. The solution we came up with is conceptually simple: the ArUcos are ordered by considering the nearest marker as next, disregarding any ArUcos that are further away than a threshold. This last detail proved to be of critical importance, partially plugging mapping errors. The algorithm is detailed in **??**.

5 Teamwork

Given the complexity of the task, we decided to tackle every aspect together. We opted for this approach for two reasons:

- Neither of us had ever worked on such a project. As a consequence, we had to work together to find solutions for the problems we had to face;
- Given the complexity of the code, implementing tasks separately would have reduced the awareness each of us had on which section of code did what. This would have resulted in less uniform code as well as taken more time for us to integrate different sections.

However, within the individual working sessions we often divided the tasks, so as to speed up the work and come up with more efficient solutions. This approach worked quite well, allowing us to constantly confront different ideas and work on our teamwork skills. The table in **??** shows the project timestamps.

A Figures

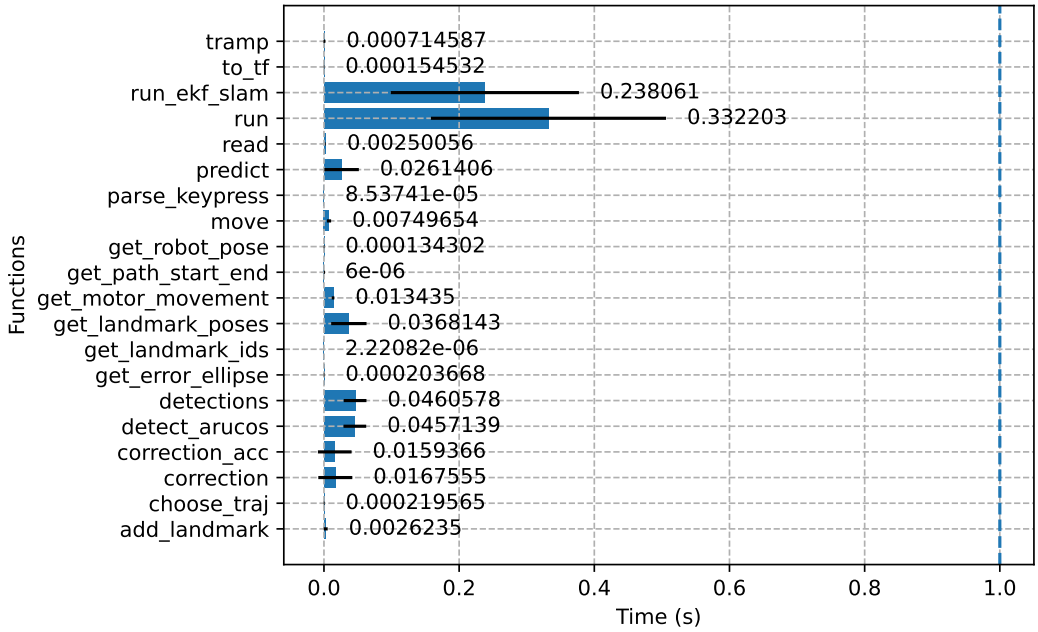

Figure A.1: Execution times of the main functions during the exploration rounds.

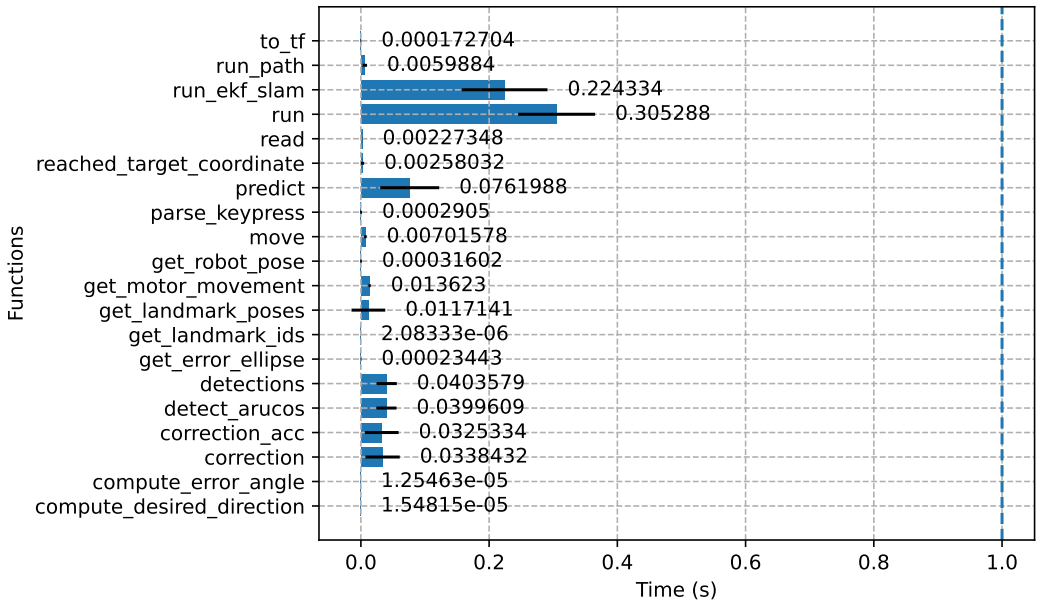

Figure A.2: Execution times of the main functions during the race.

B Time Schedule

Approximate time schedule and task distribution.

Date	Description of Work
10/11/2023	Compiled the OS for the robot and wrote some code to test the motor functions.
10/11/2023	Modified config file for robot.
30/11/2023	Implemented ArUco and circles detection. Also converted ids to integers and use proper numbering for circles.
03/12/2023	Added robot parameters to config. Returned landmark positions with respect to world coordinates. Tested ev3 library.
07/12/2023	Wrote EKF prediction step. Improved vision by drawing rectangles around detected ArUcos and cirles.
10/12/2023	Fixed some bugs, added possibility to save screenshots from camera. Started implementing correction step in EKFSLAM and made first test with reactive control drive.
14/12/2023	Improved reactive control drive.
17/12/2023	Modified reactive control to follow outer track, added some correction maneuvres. Added parameters to allow driving in both directions.
21/12/2023	Bugfix for dimension of recognized outer ArUco. Started working on inner ArUco recognition.
23/12/2023	Improved EKF SLAM.
29/12/2023	Worked on EKF SLAM.
10/01/2024	Added error ellipses to EKF SLAM.
14/01/2024	Fixed EKF SLAM, fixed major error in coordinate system with ArUco detection.
16/01/2024	Started working on A*.
17/01/2024	Implemented map saving for A* and corrected exploration. Started working on polygon creation from landmark positions.
19/01/2024	Finished map conversion for A*.
21/01/2024	Implemented A* with temporary function.
24/01/2024	Improved reactive control on inner track. Worked on heuristic function for A*. Major speed improvements on EKFSLAM.
25/01/2024	Experimented with checkpoints to improve EKF SLAM.
28/01/2024	Added load map functionality and added start line and obstacles to the map.
31/01/2024	Worked on map conversion between grid and real world. Implemented switch to different tasks and fast mode for EKFSLAM.
04/02/2024	Minor corrections on code, started implementing PID for path following. Execution time benchmarking.
07/02/2024	Started fine-tuning PID, improved A*.
09/02/2024	Created notebook for disc recognition demo.
10/02/2024	Optimised code execution time, improved PID controller.
11/02/2024	Rewrote functions using numba, refactored code moving global variables to config file.

C Code

Reactive Control

Computer Vision

Conversion Between Discretized and Real World Coordinates

ArUco Markers Rearrangement

Path Finding Algorithm