Personal Project

Unknown Environment Mapping with multiple robots

Author:
Aurélien Werenne

Advisor:
Bernard Boigelot

Montefiore Institute
Faculty of Applied Sciences
University of Liège
Liège, Belgium

Academic Year 2018 - 2019

Foreword

This work was performed in the context of the optional course *Personal Project* at the University of Liege. The objective is to design and develop a realistic interdisciplinary project from the conceptual to the operational phase. Moreover, the project must be chosen by the student, approved by the advisor and carried out remotely.

The choice was made to construct two robots cooperating to explore and map an unknown environment.

To conclude, I would like to express my sincere gratitude to my advisor Prof. Bernard Boigelot at the University of Liège. The door to Prof. Boigelot office was always open whenever I ran into a trouble spot or had a question about my project. He consistently allowed this project to be my own work, but steered me in the right direction whenever he thought I needed it. Thank you.

Contents

Chapter 1

Introduction

1 Motivation

Constructing a map of environments using robots can be of great interest for rescue teams, especially for environments situated in war zones or affected by natural disasters. In those cases the rapidity component is of particular importance. In an attempt to improve the speed of the exploration process researchers investigated the use of multiple robots to cooperate during the mapping process [1, 2].

In this work we decided to develop a simple algorithm for a dual robotic system whose goal is to map an environment like the one shown in Figure 1.1. The algorithm was then validated in a self-constructed simulator. Once the results were satisfying, the robots were constructed and tested on physical environments.

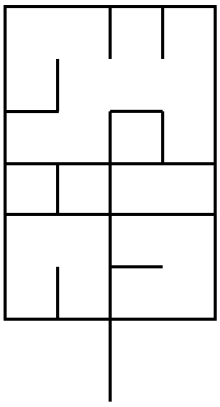


Figure 1.1: *Example of an environment. The robots/slaves start their exploration at the bottom edge. In the physical world, the scale is such that the smallest edge displayed above corresponds to 20 cm.*

2 Problem Statement & Hypothesis

The objective is to construct two robots that cooperate to map an unknown environment as quickly as possible. The environments considered are exclusively composed of straight paths[1] intersecting each other with right angles, where the paths are modelled by black lines on a white background (see Fig. 1.1). The robots are only allowed to move along the black lines. Furthermore, each robot is equipped with sensors enabling it to i) distinguish the black line from the white background ii) detect and classify an intersection iii) detect the other robot if it is in a very close range.

3 Proposed Solution

The proposed solution is implemented in the form of a master-slave architecture, where the master will be running locally in our computer and the slaves correspond to the robots (see Figure 1.2). The slaves move through the environment while incrementing a distance counter until an intersection is encountered. In this case, the slave stops and sends an information message to the master, containing the travelled distance and the type of encountered intersection. The master receives this information, and is used to further construct an internal representation of the map. Next, the master computes the direction the slave needs to move towards to. This instruction is then send to the slave. It is worth noticing that the slave stays at rest during the decision process of the master. The code for this work can be found at this[2] repository.

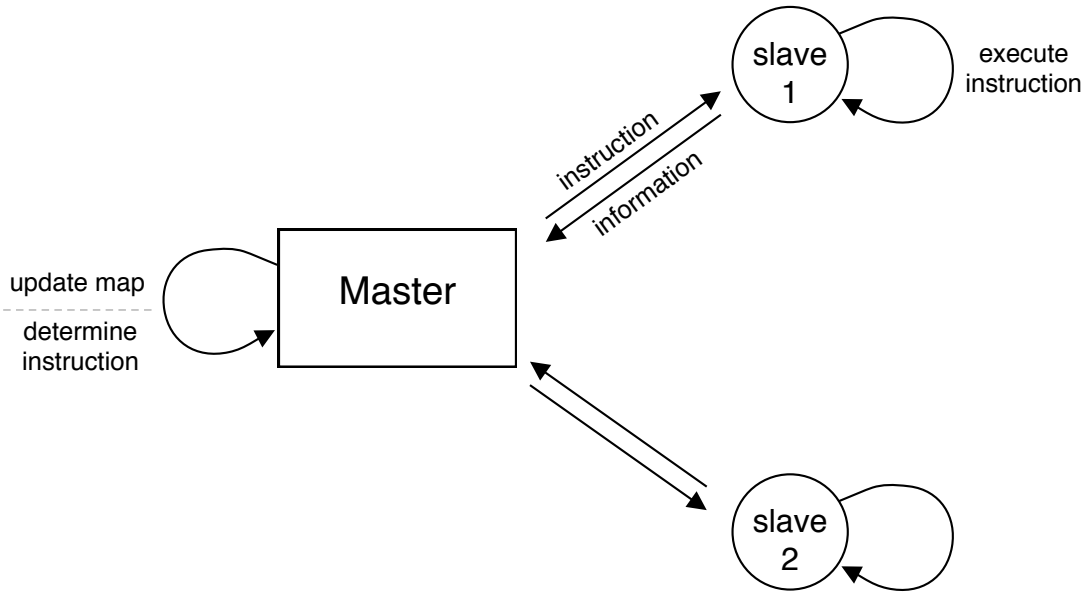


Figure 1.2: *Master-slave architecture.*

[1] Also called *edges* and *roads* in this work.
[2] Url: https://github.com/Werenne/multi-agent-mapping

The next chapter describes the implementation of the master and simulator. Chapter 3 then details the construction/implementation of the physical robot.

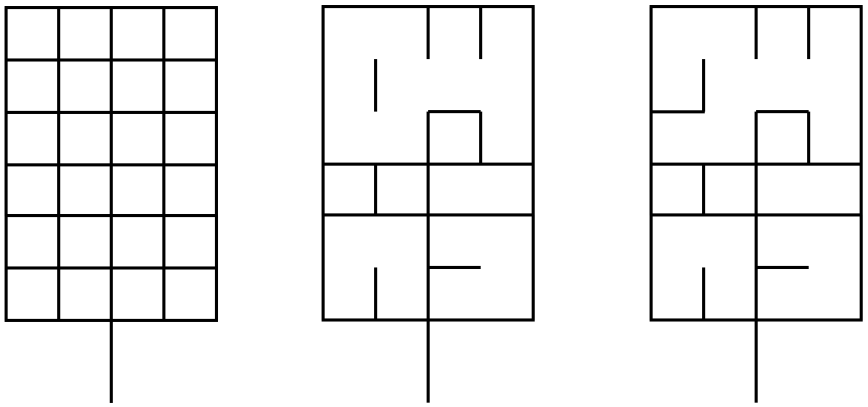
Chapter 2

Master

1 Random environment generation

The structure of the environments on which our system is tested (in the simulator and physical world) is generated randomly. The algorithm for this task had to be written from scratch as no existent one was found. The procedure consists in three main steps,

1. *Initialization*: initialize the set of edges as shown in Figure 2.1a.
2. *Destruction*: remove randomly 40%[1] of the interior edges.
3. *Fixing*: adds an edge randomly on nodes disconnected from the rest of the graph (relative to the *root node*[2]). This process is repeated until all remaining nodes can be accessed from the *root node*.



(a) *Initialization* **(b)** *Destruction* **(c)** *Fixing*

Figure 2.1: *Resulting environments of three steps of the generation.*

[1] Determined arbitrary by experimenting.

[2] The root node refers to the node at the bottom, and is also the starting point of the robots.

2 Simulator

It would be costly in time to design, debug and evaluate the performance of the master algorithm if ran in combination with the physical robots. Thus, a simulator was constructed emulating environment and robots. In particular, the simulated robots are constrained to be only able to follow roads (edges) in the environment. Furthermore, their sensory inputs were limited to a one-step lookahead in the front, left, and right directions. Each robot increments their distance and once they encounter an intersection, an information message is send to the master.

3 Communication protocol

Messages are of the following form: <*identifier*, *sequence number*, *message*>. The *message* can correspond to *information* if the sender is a robot, or to an *instruction* if it comes from the master. The *identifier* is used so that the master is able to distinguish the robots, but also to prevent communication between the robots. The *sequence number* permits to trigger a recovery mechanism for unreceived messages and to ignore redundant messages. Delimiters (<, >) are used to verify that the entire message is received.

4 Graphical User Interface

Simulations are visualized in a Graphical User Interface (GUI), which was made with the framework `PyGame`[3]. The GUI enables us, on the one hand to visualize the internal map representation of the master, and on the other hand the groundtruth environment and robots (see Figure 2.2). In order to keep the simulations smooth, the master, the simulator and the GUI are executed as separate threads. Let us emphasize that the same GUI is used for the physical robot exploration, but obviously does not display the groundtruth since it is not known.

5 Architecture & Decision Algorithm

Architecture

The architecture of the master is composed of three main classes: `map`, `messenger` and `master`. When a message is send by a robot, it is received, parsed and verified by the `messenger`. The information is then passed to the `map` which updates the internal representation of the environment and the corresponding

[3]More information about the `Pygame` can be found here.

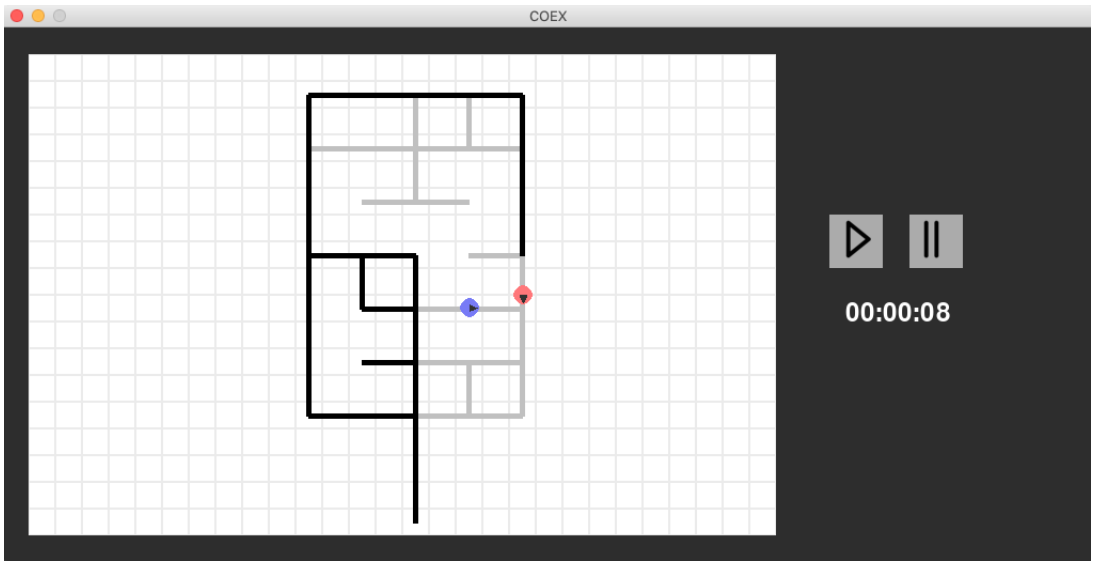


Figure 2.2: *A snapshot of the Graphical User Interface (grey lines correspond to the groundtruth, black lines to the current internal map representation, coloured circles to the robots).*

robot pose. Then, the `master` computes and sends the optimal direction that must be taken by the robot.

Decision Algorithm

When a robot arrives at an intersection and sends the information, it waits for a new instruction. This instruction is determined by the decision algorithm described in the next paragraph. Beforehand, the concept of *frontier*, although commonly used, must be specified: a frontier is referred to as a known but unexplored edge, i.e. the robot has sensed that there is a path but did not explore it.

The decision algorithm distinguishes three cases when a robot arrives at an intersection. In the first case, the intersection is a frontier, and thus the new direction is simply chosen by using the left-hand-rule on the unexplored edges. In the second case, the intersection is not a frontier and their are are no other frontiers in the current map. It is then concluded that the exploration is completed and the robots are send back to their start position (root note). For the last case the intersection is not a frontier, however, at least one frontier is still remaining elsewhere in the current map. The robot is then assigned to a frontier. The chose frontier is the one with the minimum Manhattan distance to the current position of the robot. Once the frontier assigned, the master computes the path to take for the robot via the A* algorithm[4]. The robot then moves towards the assigned frontier following the determined path. However,

[4]The Dijkstra algorithm improved with the heuristic that a path to a goal is always greater or equal to the Manhattan distance to that goal. This permits to avoid unnecessary computations. This heuristic is valid since all the angles are right.

the path is recomputed at each intersection since the explored environment expands, and thus new optimal paths could potentially be found.

Simulations showed that using two robots with the aforementioned algorithm increased the rapidity by 61% compared to only using one robot. Although the results seemed sufficient, four problems appeared during the simulations. These problems are described below with a proposed solution.

Avoiding robot encounters

Since rapidity is the main objective, caution must be taken to avoid robot encounters as much as possible, since the recovery process (described in the next sub-section) would lead to a loss of time. The method applied to avoid encounters is by masking the neighbouring edges in the map around the other robot when computing a path. More specifically, the edges are masked by assigning them an infinite weight in the A* algorithm. Two cases are distinguished: if the other robot is on an intersection, the neighbourhood is defined as the set of edges directly connected to that intersection. On the other hand, if the other robot is moving on an edge, the neighbourhood corresponds to the neighbourhood of the next intersection the other robot will encounter.

Handling robot encounters

Masking the neighbourhood will not guarantee to completely avoid robot encounters. For example, two robots moving from opposite sides on an unexplored edge can not be planned/predicted. When such an encounter occurs, one of the two robots is send away, and when out of reach, the second robot updates its path and moves forward.

Updating assigned frontiers

During the time a robot moves towards its assigned frontier, the other robot could potentially explore it, coming from the other side of the frontier. This issue is solved by verifying for each newly explored frontier if it was assigned.

Cooperative strategy

An attempt is made to induce a cooperative behaviour. Indeed, we suggest that the exploration may be more efficient if the robots are enforced to a certain degree to explore separate distant regions. To do this, the cost function to chose the frontier is adapted as follows:

$$\text{cost}(\text{frontier}) = D(\text{current}, \text{frontier}) - \lambda D(\text{frontier}, \text{other robot}) \qquad (2.1)$$

where the function D represents the manhattan distance between two points, and λ is the importance parameter controlling the trade-off between the two terms. The first term suggests to choose the most nearby frontier, whereas the second term induces the robot to chose the frontier which will be the most distant form the other robot.

Different values of λ are tested, for each λ 20 environments were generated and explored. Figure 2.3 shows the mean elapsed time[5] of the explorations. The optimal trade-off seems to be for $\lambda = 0.875$.

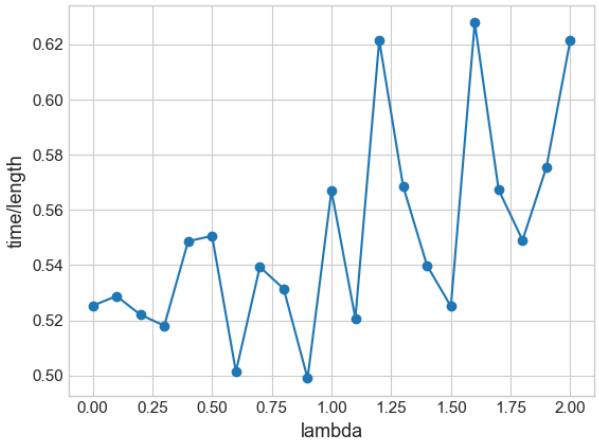


Figure 2.3: *Mean elapsed time for different λ's.*

[5]The elapsed time rescaled by the covered length of the corresponding environment.

Chapter 3

Robot (slave)

The first part of this chapter (Section 1 and 2) discusses physical aspects of the robot such as its components and how these are assembled. The second part then describes the software and control aspect.

1 Components & Functions

We identified seven different requirements the robots needs to fulfil, they must be able to

- perform basic computations.
- move and rotate.
- localize black lines on white background.
- avoid collisions with the other robot.
- measure the travelled distance.
- communicate with the master.
- supply power to its components.

Computational unit

The `Arduino Nano` is used as the micro-controller. The main advantages are its simplicity of use and its relatively small dimensions. On the other hand, the `Arduino Nano` only contains 14 digital pins and 8 analog pins. Moreover, some capabilities (e.g. interrupt, PWM) are only available on specific pins. As a consequence of these constraints, the connectivity was a puzzle-work and the final result is shown in Figure 3.2.

Line localization

Straightforwardly, in order to perform efficient line-following it is important to be able to localize the line with respect to the center of the robot. This problem is solved by using an infra-red (IR) reflectance sensor, which is composed of an infra-red transmitter and a photo-diode receptor. The idea is that the reflectivity of infra-red light varies with color and distance of the reflecting surface. The transmitter emits an infra-red beam onto a surface. The surface reflects the beam which is received back at the diode. For a light-coloured surface (e.g., white background), the intensity of the reflected signal will be high, resulting in a flow of current in the diode. On the contrary, if it is dark-coloured (e.g., black line), then the intensity of the reflected signal will be low, so that the diode blocks the current. By placing an array of those sensors side-by-side, the robot becomes able to detect where the black line is positioned on the white background.

Instead of constructing such an array from separate sensors, we use the `QTRX-MD-06A Reflectance Sensor Array`, consisting of six IR sensors (see Figure B.3 in Appendix B). In addition, it has a high-level interface permitting the robot to obtain a continuous value for the position of the line (0 if line at left extremity, 5000 if line at right extremity). A continuous value is a significant advantage since it improves the smoothness of the control system (see Section 5).

In practice, from this continuous value we will compute the *error line*. The error line is defined as the deviation from the center, such that the center of the robot corresponds to an error of zero. Figure 3.1 shows then error for a moving black line. As can be seen, the measures do not contain significant noise, due to pre-processing steps of the `QTRX` sensor. However, it was found experimentally that shadow differences in luminous environments could affect the performance. To solve this issue a shield in carton was constructed and placed around the sensor (see Figure B.3).

Obstacle detection

Since multiple robots are moving in the environment, collisions could occur. To avoid this, a `Sharp` sensor is placed in front of the robot to detect their peers. A Sharp sensor is also an infra-red sensor but with an additional feature, it can measure the distance to an object and return a corresponding analog value. The desired behaviour is to trigger a signal quickly for an object in a range close to collision. In order to do this, we chose the `Sharp GP2Y0D805Z0F`, which detects objects between 0.5 cm and 5 cm away, and has a quick response time.

It is commonly recommended to add a bypass capacitor because the sensor draws current in large and short bursts. A bypass capacitor is a low-pass filter placed across the power and ground close to the sensor, to diminish the

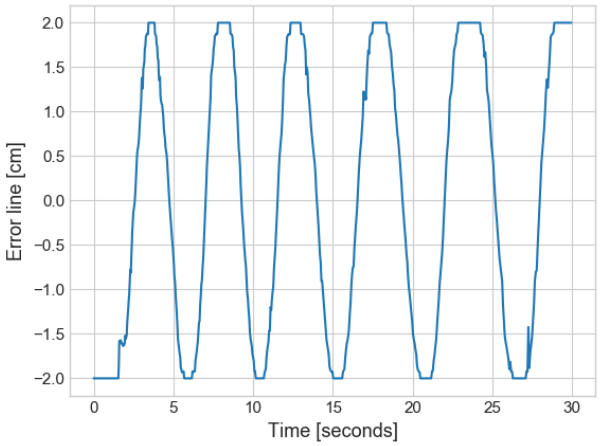


Figure 3.1: *Experiment where a black line is moved back-and-forth between the left and right side of the robot.*

possible interferences resulting from these bursts.

Moving

The robot is designed as a differentially steered two-wheel. It is differential in the sense that both wheels can have different rates of rotations but unlike a differential gearing system as in automobiles, a differentially steered system will have both wheels powered. In other words, to go to the left (resp. right) the speed of the right (resp. left) motor is increased. Additionally, a free rotating wheel is placed at the front of the robot to keep the body in balance and be able to rotate in both directions. Notice that the robot will rotate around the center between the two wheels. The wheels are powered by brushed DC gearmotors with a 150:1 ratio (see Figure B.3 in Appendix B), controlled by sending a special form a digital signal, namely a pulse width modulation (PWM). The higher the absolute value of the PWM, the faster the revolutions of the motor (a negative PWM corresponding to rotations in the opposite direction). In addition, a dual H-bridge `L298n` is used in order to decouple the power supply to the motors and the arduino.

Distance Measurement

The robot computes the travelled distance by multiplying speed and time, at short regular intervals. For this purpose, two quadrature encoders are soldered to the motors in order to measure the speed. The idea is that the rotating DC motors induce a hall effect in the encoders at each rotation. The used encoders trigger six counts per revolution. The number of revolutions of left and right motor, are denoted respectively n_L and n_R. Thus, the angular velocity of the

left wheel[1] can be computed,

$$w_L = 2\pi f_L \qquad \text{with} \qquad f_L = \frac{n_L}{6\, G_b\, \Delta t} \tag{3.1}$$

with G_b the gearbox ratio, f_L the rotation frequency and Δt the elapsed time. The robot being modelled as differentially steered two-wheel, we obtain the speed of the robot as

$$v = \frac{v_L + v_R}{2} = \frac{R(w_L + w_R)}{2} \tag{3.2}$$

with R the radius of each wheel. Thus, the travelled distance is written,

$$\begin{aligned} \Delta x &= v\Delta t \\ &= \frac{\pi R}{6G_b}(n_L + n_R) \end{aligned}$$

Several factors are not taken into account in this computation such as acceleration, noise in the measurements and the fact that the robot experiences small deviations during the line-following process. With this in mind, the computed distance is discretized in sufficiently large intervals (5 cm) to be considered completely error-free. Lastly, interrupt routines were used, since the counts of the encoders must occur independently of the program flow.

Communication

Communication with the master is performed via a bluetooth module `HC-06`. This specific module is adapted for establishing short range wireless data communication between two systems. Notably, a voltage divider was added to the circuit because of the difference between the logic level voltage of the Arduino ($0 - 5$ Volt) and the bleutooth module ($0 - 3.3$ Volt) (see Figure 3.2).

$$\begin{aligned} V_{arduino} &= \frac{R_4}{R_3 + R_4} V_{bleutooth} \\ &= \frac{2.2}{3.2} V_{bleutooth} \end{aligned}$$

Power supply

Most components have a low consumption. Moreover, no high-performance speed was required. Thus, a rechargeable NiMH Battery of 7.2 Volts with a capacity of 900 mAh seemed sufficient.

[1] The angular velocity of the right wheel is computed in an analogous manner.

2 Physical construction

The robot is constructed following the steps described below:

1. A chassis is cut from wood and used as a base. Additionally, four small pillars are pasted at the corners (see Figure B.4 in Appendix B).
2. The H-bridge is screwed on the chassis.
3. A breadboard is pasted on the pillars.
4. The micro-controller is soldered into header pins which are then fixed in the breadboard.
5. The encoders are soldered to the motors, and the latter are then fixed on the bottom of the chassis.
6. The wheels are placed on the shafts of the motors, and the omni-wheel is placed under the chassis, at the front.
7. The Sharp and QTR sensor along with its shields are fixed at their desired place.
8. Wiring all the components (see Figure 3.2).
9. Connect to the power via a soldered switch.

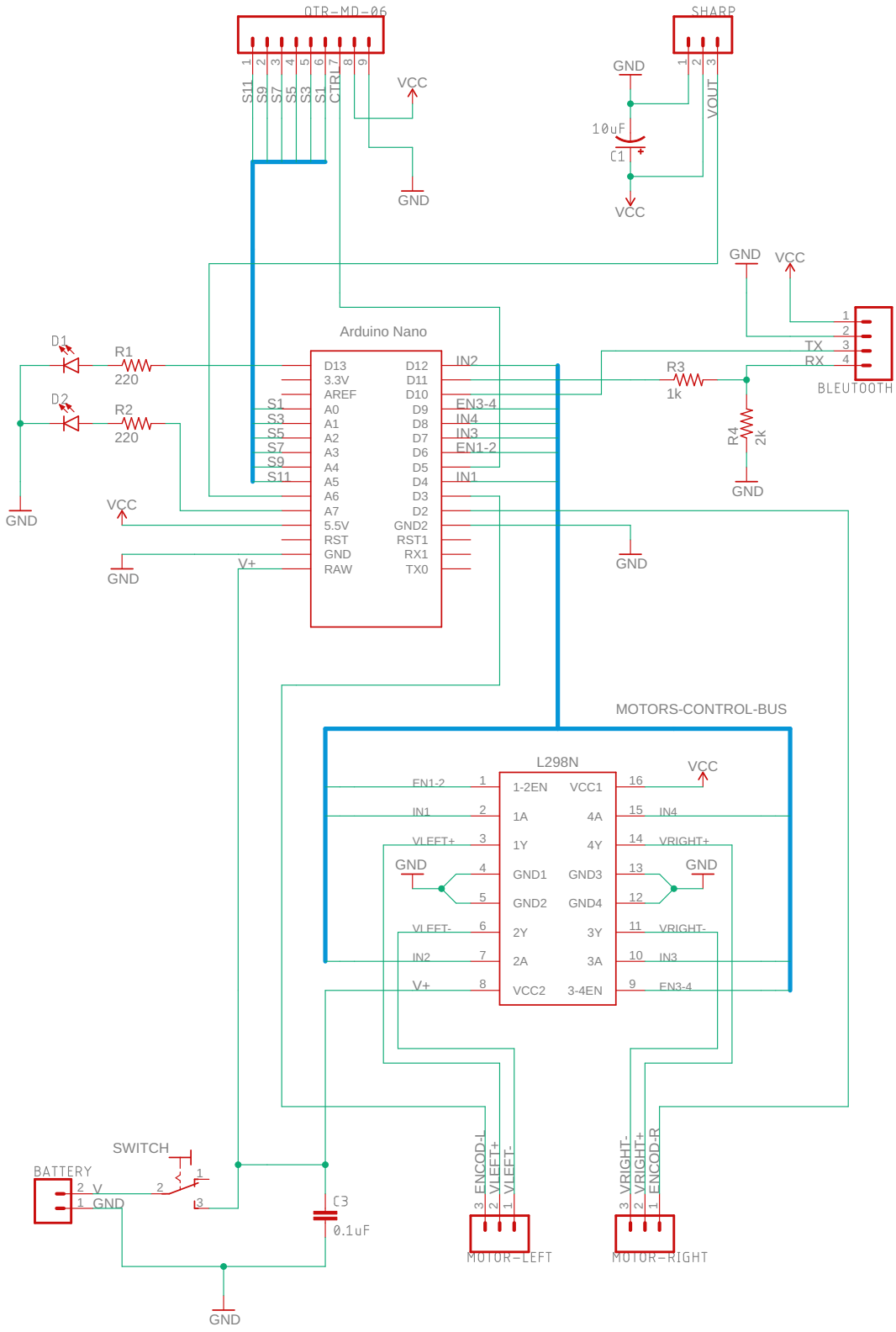


Figure 3.2: *Electric diagram of the robot*

3 High-level algorithm

The main steps of the robot's algorithm are detailed below. As can be observed, the robot executes mostly physical operations whereas the Master does all the 'smart' stuff.

Algorithm 1 High-level view algorithm robot

```
1: procedure MAIN
2:     active ← False
3:     distance ← 0
4:     while True do
5:         if CHECKMESSAGE then
6:             instruction ← READMESSAGE
7:             if instruction == 0 then
8:                 STOPMOTORS
9:                 active ← False
10:            else
11:                EXECINSTRUCTION(instruction)        ▷ turn left, turn right, uturn
12:                active ← True
13:            end if
14:        end if
15:        if active then
16:            FOLLOWLINE
17:            distance ← MEASUREDISTANCE + distance
18:            if ISINTERSECTION then                  ▷ see subsection below
19:                type ← CLASSIFYINTERSECTION         ▷ see subsection below
20:                FORWARDALIGN                        ▷ see subsection below
21:                SEND(distance, type)                ▷ to Master
22:                active ← False
23:            end if
24:        end if
25:        WAIT(20)                                    ▷ milliseconds
26:    end while
27: end procedure
```

Intersection detection & classification

An anomaly is triggered when black is detected at one of the extremities of the `QTR` sensor. Since it could be a noisy measure, the robot continues its execution and checks if the same anomaly pattern occurs consecutively. If this is the case, the anomaly is finally viewed as an intersection. Whenever an intersection is detected, it continues its movement for a few inches in order to classify its type as given by Figure 3.3.

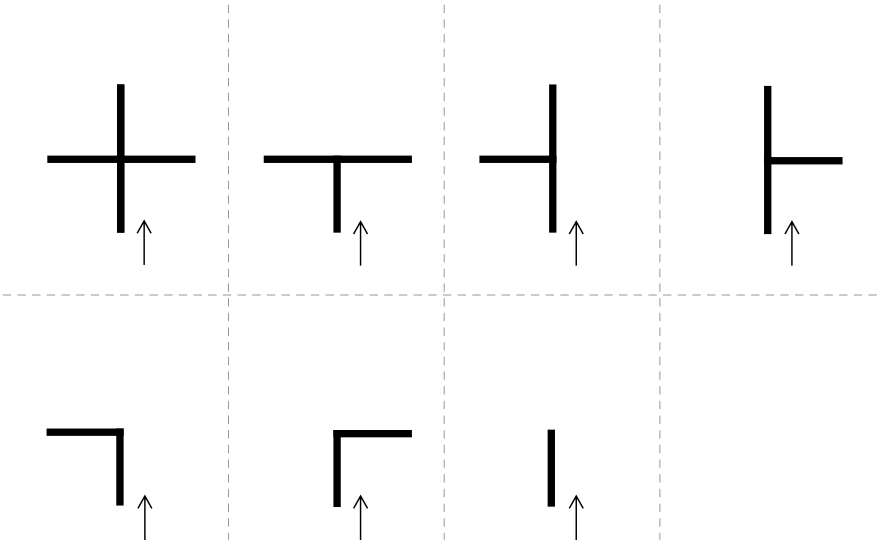


Figure 3.3: *Different types of intersections.)*

Forward align

Since the robot rotates around the central point between the wheels (at the back), the robot must align the wheels with the detected intersection. This permits the robot to still be nicely aligned after a turn, and thus avoid large initial deviations in the line-following. In other words, once an intersection is detected the robot moves forward for a distance equivalent to its own length. As there may be no black line in front, the robot moves without the follow-line procedure but instead controls the speed of the two motors to be equal in order to go in a straight line.

4 Architecture

Figure 3.4 shows the architecture of the implementation in the micro-controller. The *main* corresponds to the algorithm above. The *COEX* class refers to the high-level features related to the cooperative exploration, which is based on low-level features. The advantages offered by this modularity are two-fold: it makes the debugging easier and the low-level features could be re-used for a different purpose.

A particular problem observed experimentally was that the sensors and actuators had optimal performance at different frequencies. The solution implemented assigns a frequency object to each to component/task.

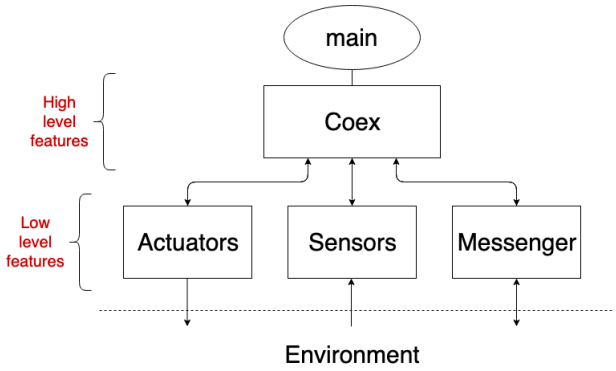


Figure 3.4: *Architecture of the code of the robot.*

5 Control

Line Following & Speed Control

As a beginner roboticist I thought naively that the robot would go straight if the signal (PWM) send to the left and right motor were equal. To verify that this claim does not hold in reality, an experiment was performed were the same PWM signal is send to both motors and their respective speed are measured with the encoders. The result is shown in Figure 3.5. A varying difference between the two motors can clearly be noted. Moreover, we also observe that this relationship varies as the power supply discharges. With this in mind a speed controller was developed in addition to a line-following controller. The two controllers take the form of a PID and are combined as in Figure 3.9. Straightforwardly, the error of the speed controller is the *target speed*[2] v^* minus the *measured speed* v. On the other hand, the error of the direction controller is given by the so-called *error line*, $x^* - x$ (previously discussed in Section 1). The computed PWM values are denoted u_L and u_R for the left and right motor. With anti-windup.

The tuning of the parameters of the speed controller is done with respect to a target speed varying as in Figure 3.6. Notice that the measured speed in this Figure is the one obtained using the tuned parameters. The speed controller seems to work but there is still some room for improvements. However, the results were sufficient for our purpose and thus no further tuning was applied.

To adjust the parameters of the direction controller, the robot was placed on a parcours formed by curved black lines. The heuristic used for tuning was to double the K_p parameter until oscillations occurred, then half it back one time and start to tune the other parameters. Once this was done, the target speed was increased by a small increment and the process repeated (until the desired behaviour).

By experimenting, several improvements were developed as can be viewed

[2]For the demo we fix the target speed to 8 centimeter per second.

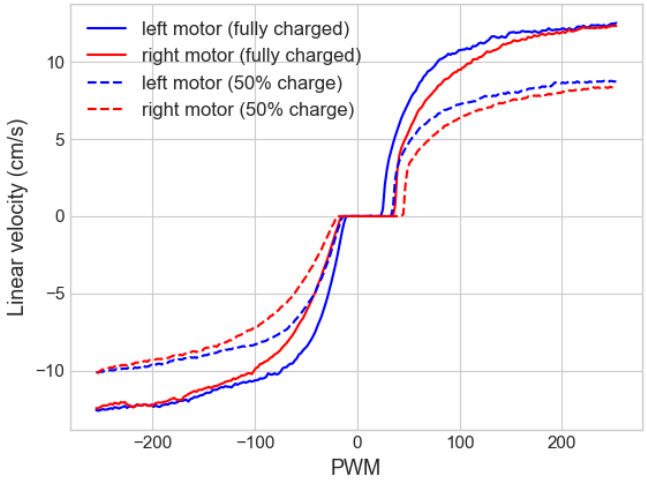
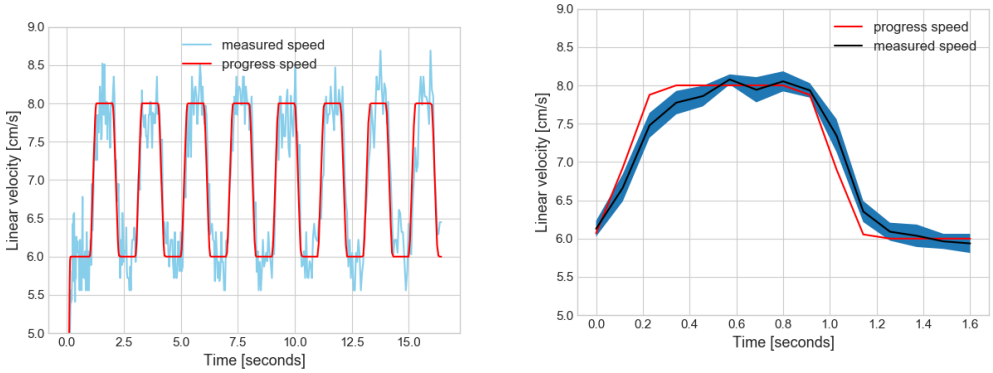


Figure 3.5: *Variations of speed between the motors*



(a) *multiple cycles*

(b) *average trend with bandwith representing the variance*

Figure 3.6: *Experiment speed control with a varying target speed*

in the block diagram of Figure 3.9. Indeed, the first observation to make is that there are a lot starts/stops during the exploration, which caused instabilities. It was suspected that this was due to a high current drainage by the motors for a sudden acceleration. A solution is to design and implement a smoother acceleration profile[3] for a desired target speed to attain in a specific time interval, an intermediate value is computed namely the *progress speed* The progress speed follows an acceleration in trapezoidal form, permitting the robots to have a slower start.

Another challenge is that there can be a misalignment between the black line and the center of the robot after a turn. As a result, a *derivative kick*[4]

[3]See Appendix A.

[4]A derivative kick refers to a very large derivative as a consequence of a significant error. As large corrections are undesired, a common approach is to derive the measurements instead (already our case by default since setpoint remains unchanged) and to fix initial derivation

can occur, which was solved by fixing the initial derivation to zero. A second consequence of the misalignment is that in combination with the smooth acceleration, the correction from the direction controller is potentially too large with respect to the correction of the speed controller. This issue was solved by having a dynamic setpoint weighting, so that the direction correction is weighted with an hyperbolic tangent function converging rapidly with time to unity.

Turning

The COEX-1 is modelled as a two-wheel robot (see figure below). By integrating the angular velocity under the assumption that $v_R = -v_L$, rotational displacement can be deduced

$$\theta(t) = \frac{2vt}{b} + \theta_0 \tag{3.3}$$

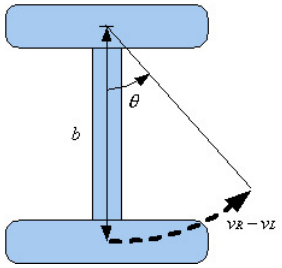


Figure 3.7: *Rotation of COEX-1*

In order to rotate in a specific direction, a naive approach was first tested: rotating at a given speed and duration to obtain the desired angle using Equation 3.3. The results were not catastrophic, but a significant variance was observed, going from a few degrees to a few dozens. This is not acceptable as our goal is to have the center of the robot aligned as closely as possible to the black line.

A second approach tested, was to use `QTR` sensors to detect when the line was in the middle and at that instant, to stop the robot. Notice that if the the robot needs to turn 180 degrees but if edges are at his sides (90 degrees), the robot will stop to soon. To this end, the robot was first rotated with an angle of 130 degrees (using the method of the previous paragraph) and then continued with the `QTR` solution until the robot is aligned. An improvement was observed, however, the robot always stopped a little too late due to drift of the wheels with the surface and a lack of response time from the sensors/actuators.

Ideally, the robot would decrease its rotation speed when approaching the line and sufficiently slow to stop and be aligned perfectly. The implemented

to zero.

solution uses a simple controller so that the speed is decreased with respect to the error line measured. The deceleration was triggered from the moment the black line was in the range of the sensor (i.e., at one of its extremities).

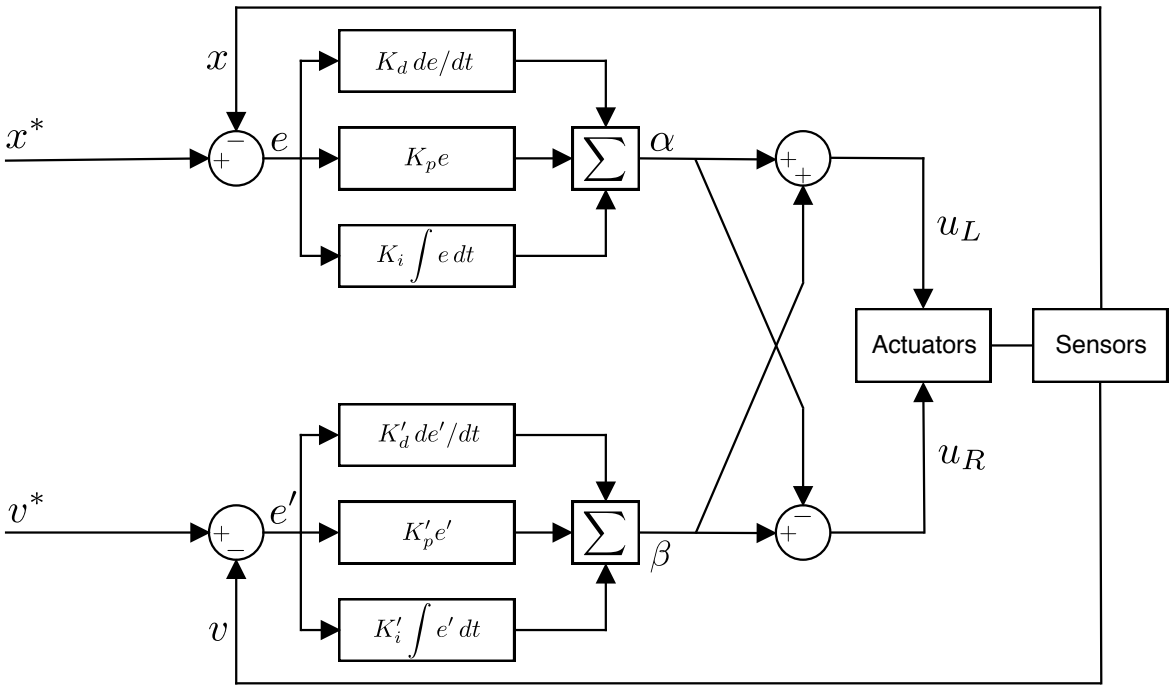


Figure 3.8: *Schematic diagram of the control system (with anti-windup).*

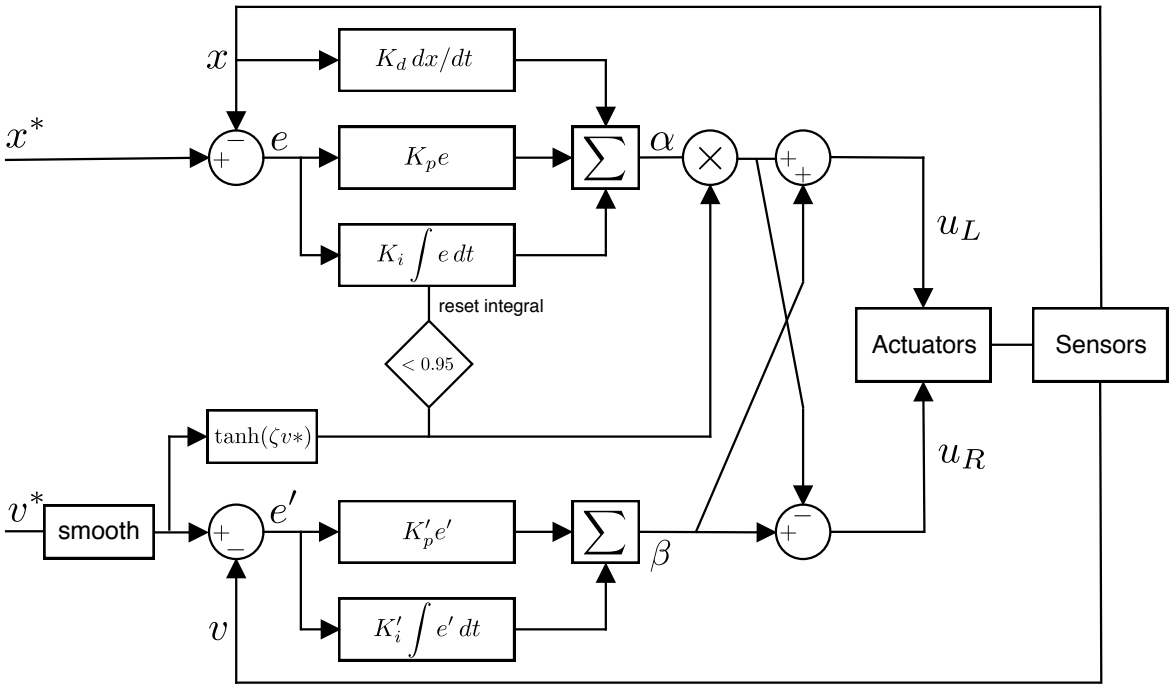


Figure 3.9: *Schematic diagram of the improved control system (with anti-windup).*

Chapter 4

Conclusion

The algorithm of the master that was developed in simulation, showed satisfying results. We verified an improvement from using two robots instead of one. In addition, a slight performance gain was illustrated by inducing a simple cooperative behaviour, enforcing the robots to explore distant regions to a certain degree.

Constructing and implementing the physical robot to be robust and fast was challenging. Nevertheless, an efficient error-free mapping system with one robot could be demonstrated.

Finally, two robots were used to map a physical environment. No entirely error-free solution was found as collisions could not be avoided for certain angles and speeds.

Appendix A

Smooth acceleration

The default acceleration profile is uniform (see Figure A.1a). The update of the progress speed is as follows

$$v_{n+1} \leftarrow v_n + A\,\Delta t_{n:n+1}$$
$$\text{with } A = \frac{v_{target}}{T}$$

For a smoother transition, the robot accelerates following the trapezoidal profile as in Fig.A.1b. As we want the robot to achieve the target speed in the same amount of time T. This constraint is developed,

$$\Leftrightarrow \int_0^T a(t) = \int_0^T a'(t)$$
$$\Leftrightarrow A\,T = \underbrace{d\,B}_{triangles} + \underbrace{(T-2d)B}_{rectangle}$$

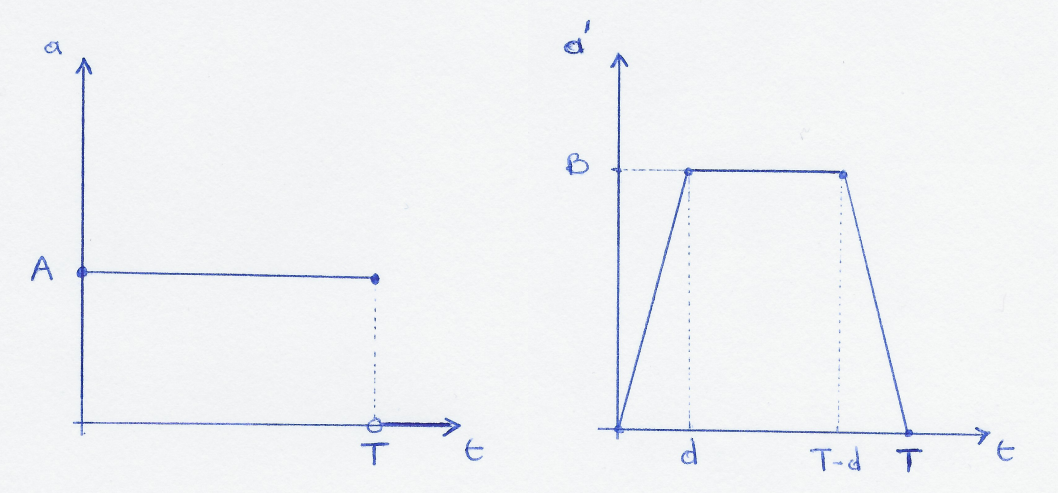
By substituting the parameter $\psi = \frac{d}{T}$, we obtain

$$\boxed{B = \frac{A}{1-\psi}}$$

The update rule for the progress speed becomes

$$v_{n+1} \leftarrow v_n + \int_n^{n+1} a'(t) = v_n + \int_0^{n+1} a'(t) - \int_0^n a'(t)$$

Notice that with the right hand side of the above expression, we can make use of precomputed values of the geometric surfaces.



(a) *uniform* **(b)** *trapezoidal*

Figure A.1: *Acceleration profiles.*

Appendix B

Construction

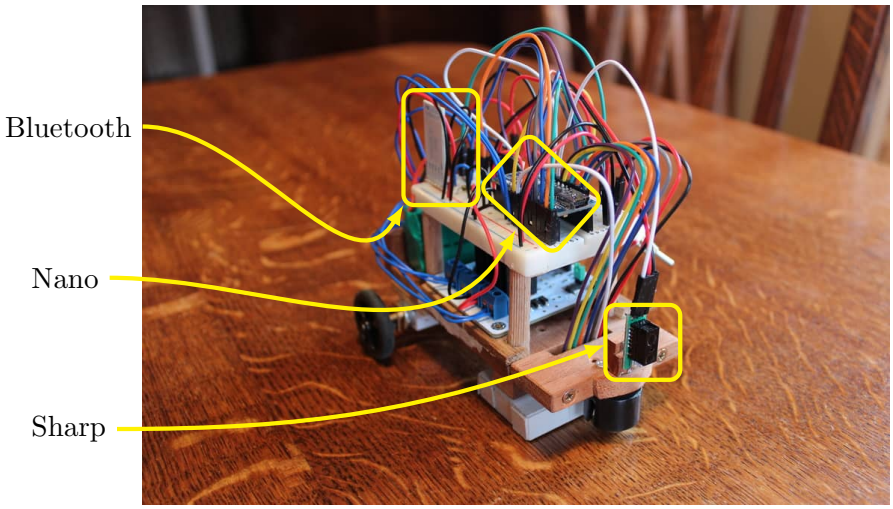


Figure B.1: *Front view of robot*

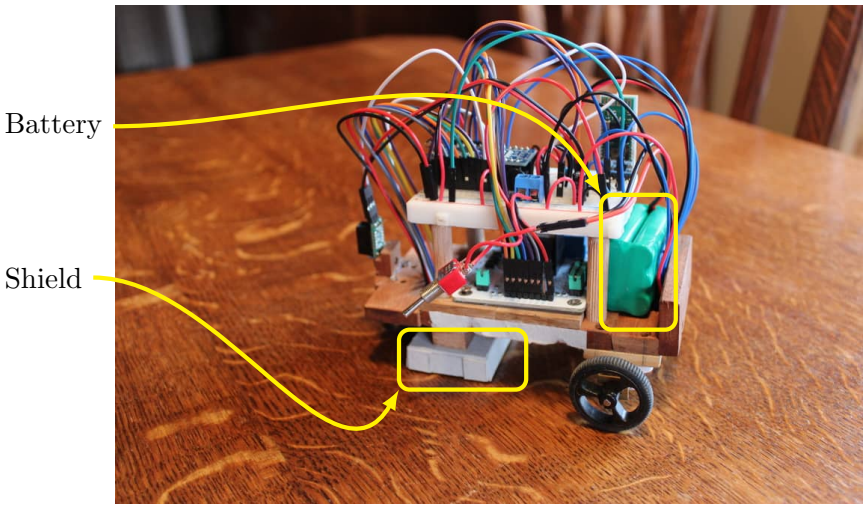


Figure B.2: *Side view of robot*

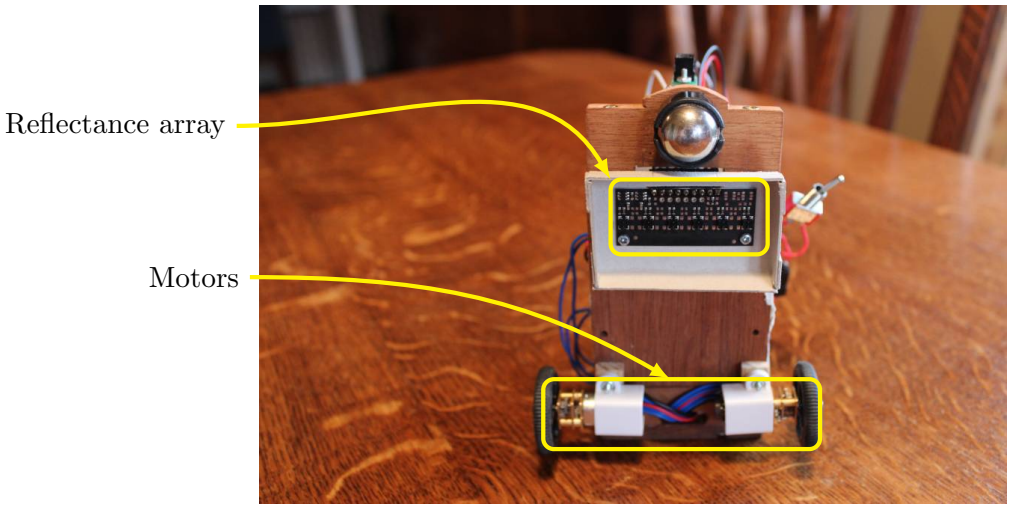


Figure B.3: *Bottom view of robot*

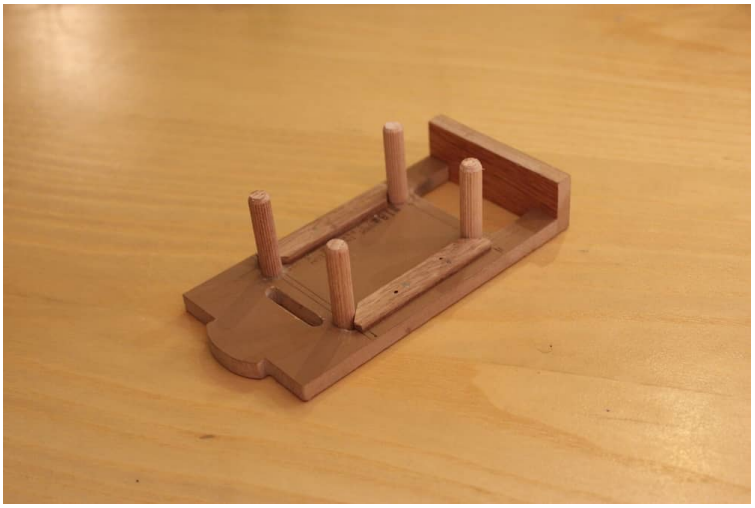


Figure B.4: *Wooden chassis*

Bibliography

[1] W. Burgard, M. Moors, C. Stachniss, and F. E. Schneider. Coordinated multi-robot exploration. *Trans. Rob.*, 21(3):376–386, June 2005.

[2] Kurt Konolige, Didier Guzzoni, and Keith Nicewarner. A multi-agent system for multi-robot mapping and exploration. In Alan C. Schultz and Lynne E. Parker, editors, *Multi-Robot Systems: From Swarms to Intelligent Automata*, pages 11–19, Dordrecht, 2002. Springer Netherlands.