ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Developing ROS-based software for the e-puck

An experiment on exploration and mapping

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique

Miquel Kegeleirs

Directeur
Professeur Mauro Birattari

Co-Promoteur
Gianpiero Francesca

Superviseur
David Garzon Ramos

Service
IRIDIA

Année académique
2017 - 2018

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme: DEMIURGE Project, grant agreement No 681872

Acknowledgment

I would first like to thank my thesis director, Mauro Birattari, for encouraging me to go further in my work and in my objectives.

I would also like to acknowledge my supervisor, David Garzon, for all the help and the good spirit he provided throughout this work, for the invaluable advise and comments he gave me and for being so enthusiastic about my work.

I want then to thank the researchers of IRIDIA laboratory, especially Ken Hasselmann, Antoine Ligot and Harry Weixu, for the friendly atmosphere at the lab and their punctual but crucial help when I was struggling with my work.

Finally, I must express my profound gratitude to my parents and to Angélique for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. In particular, I give a special thanks to my grandmother for reading my entire thesis four or five times, searching for mistakes. This accomplishment would not have been possible without them. Thank you.

Miquel Kegeleirs

Résumé

Developing ROS-based software for the e-puck

Les recherches sur les systèmes multi-robot, et en particulier sur la robotique en essaim, ont largement prouvé que des tâches complexes peuvent être accomplies par l'interaction de robots simples et peu puissants. Le robot e-puck est un bon exemple de cette notion. Malgré sa simplicité, l'e-puck a été utilisé dans des études portant sur des expériences de conception automatique d'essaims de robots, de comportements d'organisation spatiale et de processus de décision collective ; à la fois en simulation et avec de véritables robots.

Cependant, plusieurs années se sont écoulées depuis la création de l'e-puck et son architecture logicielle est maintenant obsolète. De plus, de nouveaux outils puissants tels que ceux fournis par ROS, une plate-forme de développement logiciel pour robots largement utilisée, ne pouvaient pas être facilement intégrés facilement au système embarqué de l'e-puck. Par conséquent, des tâches intéressantes telles que l'exploration et la cartographie de scénarios inconnus ont été à peine traitées en utilisant l'e-puck.

Ce mémoire a pour but d'étendre les capacités de l'e-puck en intégrant ROS dans son architecture logicielle. À cette fin, j'ai développé des outils et des méthodologies qui permettent désormais aux chercheurs de créer des logiciels basés sur ROS pour l'e-puck. En particulier, j'ai conçu et évalué des contrôleurs utilisant les fonctionnalités de ROS qui accordent à l'e-puck, ou à un groupe d'entre eux, la possibilité de cartographier des scénarios inconnus, en simulation ou en réalité. Différentes stratégies d'exploration ont été évaluées et les résultats ont démontré que les nouvelles capacités ajoutées à l'e-puck sont prometteuses pour aborder de nouveaux paradigmes de recherche dans le domaine des systèmes multi-robot et de la robotique en essaim.

Mots-clés : e-puck, exploration, cartographie, multi-robot, ROS, robotique en essaim.

Mémoire présenté en vue de l'obtention du diplôme d'Ingénieur Civil en informatique
Miquel Kegeleirs, 2017-2018

Abstract

Research on multi-robot systems, and particularly on swarm robotics, has largely proven that complex tasks can be solved by the interaction of simple robots. The e-puck robot is a good example of this notion. Despite its simplicity, the e-puck has been used in studies that encompasses experiments on the automatic design of robot swarms, spatially organizing behaviours and collective decision processes; both in simulation and reality.

However, many years have passed from the e-puck release and its software architecture is now outdated. Moreover, new and powerful tools such as the ones provided by the widely used ROS framework could not be easily ported to the e-puck. Therefore, interesting tasks like the exploration and mapping of unknown scenarios have been barely addressed by using the e-puck.

This master thesis aims to extend the capabilities of the e-puck by integrating ROS in its software architecture. With this purpose, I developed tools and methodologies that now allow researchers to build ROS-based software for the e-puck. In particular, I conceived and evaluated ROS-based controllers that grant the e-puck, or a group of them, the ability of mapping unknown scenarios either in simulation or reality. Different exploration behaviours were evaluated and results demonstrated how the new capabilities added to the e-puck are promising for addressing new research paradigms in the domain of multi-robot systems and swarm robotics.

Keywords : e-puck, exploration, mapping, multi-robot, ROS, swarm robotics.

Contents

List of Figures

Chapter 1

Introduction

Swarm robotics is a thrilling field of robotics that exploits redundancy of many simple agents to achieve complex tasks. Although the research has notably evolved in past years, most swarm robotics studies today are still conducted on simulators as real robot experiments are complex and time-consuming, even more when considering the low reliability of swarm dedicated robots.

The e-puck is a robot that has been designed to be used on swarm robotics research. However and although it has been largely used in the past, the robot has not been updated to fit the requirements of new paradigms of swarm robotics research.

This work aims to provide and evaluate tools that help to overcome those limitations and to extend the capabilities of the e-puck, both being achieved by integrating the widely used Robot Operating System (ROS). In order to asses the extended capabilities of the robot, they are evaluated in a task that has recently brought noticeable interest from the robotics community: robot mapping.

The rest of this Chapter is structured as follows : Section 1.1 defines the main objectives of the thesis as well as the challenges to achieve; Section 1.2 presents the main contributions of this thesis; Section 1.3 details the content of the other Chapters of this document.

1.1 Objective of the thesis

The aim of this work is to conceive, develop and evaluate ROS-based software for the e-puck robot.

ROS provides indeed a lightweight development framework working similarly as a true operating system : hardware abstraction, package management, inter-process communication management, etc. This can be exploited in various applications, from standardized communication between robots to mapping.

By using ROS, researchers will benefit from two main advantages :

- ROS is a high level programming framework and allows then to hide most of the complexity of some applications like mapping. Hence, the difficulty on designing and implementing

the software is reduced.

- ROS is widely used in the robotics community and several new tools have been integrated and intensively assessed by other researchers (control development, communication protocols, development standards, etc).

  Producing ROS-based software for the e-puck allows to integrate software modules implemented by other developers and share with them those developed at IRIDIA.

However, in order to fulfill this goal, ROS first needs to be installed on the e-puck. In consequence, the first part of the thesis addresses the integration of ROS on the e-puck, and the design of methodologies for developing ROS-based software for the robot. This objective present itself as a real challenge as the default distribution installed on the e-puck, Ångström, does not allow to integrate and use ROS easily.

The second part of the thesis is devoted to experiment with one of the new functionalities brought by ROS : mapping. The e-puck has short range sensors that theoretically allow the robot to perform this task. However, due to the intrinsic simplicity of the e-puck, the mapping task can be barely considered as easy as with other robotics platforms.

Even though experiments limited themselves in some cases to single robot or multi-robot systems, an important objective of this thesis is also to provide the tools required for swarm experiments involving ROS and e-puck robots. All choices done to tackle the different issues encountered throughout this work carefully took possible swarm applications into account.

1.2 Main contributions of the thesis

In this thesis, I propose some contributions to multi-robot and swarm research.

First, I designed a new Linux distribution for the e-puck extension board (Gumstix Overo COM) and installed it successfully on multiple e-pucks. This more recent distribution replaces the old and deprecated Ångström distribution and assures compatibility with recent software such as ROS.

Then, I provided a methodology and tools to develop new ROS-based e-puck controllers usable on the new distribution. This covers the whole process, from the controller design to the cross-compiling.

Finally, I exploited those new capabilities to analyze different exploration behaviours for mapping, both in simulation and in reality. I identified the most important parameters to influence mapping quality and speed. I also succeeded to accomplish multi-robot mapping in simulation.

1.3 Structure of the thesis

The current first introduction Chapter exposes the context, the objective and the contributions of this thesis.

Chapter 2 presents the state of the art of the topics addressed by this work.

Chapter 3 describes the e-puck robot architecture along with its strengths and limitations.

Chapter 4 addresses the integration of ROS and elaborates on the changes done to the e-puck software architecture.

Chapter 5 focuses on the mapping application as a working example of ROS on the e-puck robot.

Chapter 6 details experiments about exploration behaviours in mapping that have been conducted in simulation.

Chapter 7 sketches experiments realized on the real e-puck robot to validate simulation results.

Finally, chapter 8 concludes this thesis with a short summary of achieved work and some concluding remarks.

Chapter 2

Related work

Robotics field has been subject to many changes and innovation in the last years. As artificial intelligence (AI) has improved considerably, new possibilities and complex behaviours are considered for robots. Some of them can run or stand up after falling, others can have discussions with humans, some more can cut vegetables to prepare food.

However, more simple robots such as those used in swarm robotics follow a different evolution. Of course they also benefit from AI growth : more efficient optimization algorithms, better image recognition, etc. But AI is not the main factor as those small robots use simple embedded computers unable to run powerful algorithms. Hence, hardware improvement leading to new capabilities for the robots and optimization of simple essential algorithms such as exploration behaviours have the biggest impact on this field.

The rest of this Chapter is structured as follows : Section 2.1 defines swarm robotics and its main properties; Section 2.2 explores the different robots and simulators frequently used in swarm experiments; Section 2.3 describes ROS and its interest in swarm applications; Section 2.4 presents mapping and its current state of research in robotics; Section 2.5 focuses on exploration behaviours in robotics.

2.1 Swarm robotics

Among the different fields of artificial intelligence, swarm intelligence is probably not the most well known by the general public. However, swarm intelligence is an important tool in many applications. The Ant Colony Optimization (ACO)[25] algorithms for example are widely used to solve complex optimization problems such as the Quadratic Assignment Problem (QAP)[93]. Swarm robotics applies the concepts of swarm intelligence to classical robotics and constitutes a completely distinct field. The core idea of swarm robotics is to create complex global behaviours from interactions between many simple agents. Additionally, the system should present some properties that may slightly vary from one author to another. However, the following properties are commonly agreed to be desirable for a system to be considered as a swarm[15][91][94] :

- Robustness : the system should continue to operate (possibly at lower performances) in the case of the loss/dysfunction of an individual or perturbations in the environment.
- Flexibility : the system should be able to tackle different tasks.

- Scalability : the system should be able to work with a wide range of group sizes.
- Robots are autonomous and can act on their environment.
- Robots sensing and communication capabilities are local.
- Robots do not have access to a centralized control/global knowledge.
- Robots cooperate to achieve the tasks.

Swarm systems should not be mistaken with multi-robot systems. Usually, the difference stands in some centralized communication system established between the robots or the use of some localization tool such as a GPS.

2.2 Robots in swarm robotics

Swarm robotics involves a large number of robots working together and finds its strengths in this number as well as in the interaction between so many robots. Hence, robots in swarm robotics are usually small, cheap and not computationally powerful. Dozens of robotics platforms dedicated to swarm robotics exist : Kobot[102], marXbot[12], eSwarBot[20], etc.

However, most of swarm experiments are still done on simulators as performing experiments with tens or hundreds of robots is expensive, time consuming and complex and real experiments are then usually performed with robotics platforms available on renowned simulators. A good simulator is therefore a powerful tool that provides many advantages :

- Quickly reproduce experiments for analysis
- Simulate sensors/actuators to evaluate their utility or their performance
- Test scenarios dangerous for the robot or impossible to reproduce in a lab.
- Simulate more robots than we physically possess

If Gazebo[37] is a well-known free simulator providing a huge set of tools and models for many robot applications, other simulators exist like Webots[21], a paying counterpart.

Among those simulators, ARGoS is worth discussing. ARGoS is a free open-source multi-physics simulator for swarm robotics experiments mainly developed by Carlo Pinciroli[80][81]. It supports different robot types but its main feature is that it allows efficient large-scale swarms simulation contrary to most simulators that focus on accuracy. It can also use multiple physics engines as the engine is completely separated from the simulation space.

ARGoS can be easily customized through plugins, in particular the "e-puck for ARGoS" plugin developed at IRIDIA[54] models the e-puck robot in ARGoS.

The e-puck[73] is a small wheeled robot widely used in education and research for its relatively low cost and its basic yet complete set of sensors and actuators. Moreover, the robot can be extended with an extension board (Gumstix Overo COM) providing a Linux running embedded

computer and a few additional sensors/actuators. A more detailed presentation of the e-puck can be found in Chapter 3. Many swarm experiments use e-pucks such as AutoMoDe from Francesca et al.[29], an experiment in automatic design of robot swarms. The experiment shows that constrained robots can achieve tasks better than unconstrained ones and hence an automatic design method based on modules can be created. The paper has been followed by AutoMoDe-chocolate[34] and AutoMoDe has then been included in ARGoS[3]. Chen et al.[18] experimented a segregation algorithm using e-pucks mimicking disks of different sizes and Marcolino et al.[67] developed a coordination method to allow robots to spread along complex shapes and tested it with e-puck robots.

The Kilobot[71] is a very small, low-cost robot especially designed for large-scale swarm experiments. Contrary to the e-puck, it is not wheeled and moves thanks to vibrations of 3 metal legs. The Kilobot is a firmware-only robot and does not have any embedded computer but benefits from an integration in ARGoS to perform simulations[106]. They are often used in large-scale experiments for different tasks such as self-assembly[90] or collective transport of complex objects[89].

The foot-bot is a small wheeled robot especially remarkable for its high modularity, from the hardware point of view as well as from the software point of view. The robot is indeed structured in many modules with specific purposes that are as independent as possible from each other, sharing only battery power and some common connections. The robot was designed for the Swarmanoid project[30] and is well integrated in ARGoS as it is the first robot model developed for the simulator[80].

The foot-bot is much more similar to the e-puck than the Kilobot, both from software and hardware perspective. The foot-bot seems even better than the e-puck as they both share approximately the same sensors and actuators but the foot-bot wheels are tougher and able to reach 2 times the speed of the e-puck and the proximity sensors of the foot-bot are more precise and less noise-sensitive. However, the e-puck possesses a more powerful embedded computer running a more recent Linux distribution.
A noticeable work using ARGoS as simulator is the Swarmanoid Project from Dorigo et al.[30] in which an heterogeneous swarm of robots collaborates to retrieve an object. Actually, the foot-bot has been designed specifically for the Swarmanoid project and has not been commercialized. Hence no other experiment involving the foot-bot has been conducted yet.

Another important component of robots are embedded computers and the distributions installed on it. Even with the simple robots used in swarm robotics, embedded computers are often installed on the robot to enhance its capabilities such as the e-puck and its Gumstix extension.

However, developing operating system for embedded computers has always been tricky and difficult. Embedded systems have indeed needs and limitations very different from those of a classical computer : most embedded computers are rather simple and not computationally powerful, additional pieces of hardware are often needed to acquire a new functionality, memory is limited, etc. Therefore, such systems are usually optimized for their own purpose, resulting in systems very different from each other.

This leads to a major problem in embedded systems field : there are as many different systems as there are devices : Rasperry Pi has Raspbian and Noobs[79], GCtronics Overo extension for e-pucks[40] runs Ångström, etc. As there is no common architecture, each system needs its own cross-compiler and to customize a system to achieve a particular purpose can become

really complicated. The main obstacles are the documentation that does not necessarily exist for every existing system and the maintenance of the system as an update can suddenly make an application incompatible with the system.

In an attempt to unify as much as possible this huge variety of systems, Yocto Project has been launched in March 2011. Yocto Project is an open-source set of tools, processes and frameworks that facilitate the creation of custom Linux distribution. Thanks to this, projects designed for different machines and different distributions can share a common architecture with common naming conventions which improves collaboration between people and reusability of the different works. Most of all, it allows to easily handle changes, either for hardware or software update.

If Yocto Project was not the first framework to provide system building tools, its wide community and the numerous capabilities it offers pushed it in front of all other similar projects. Many of those projects have merged with Yocto since then. Even OpenEmbedded[76], another prominent system builder project, has now merged with Yocto Project and has been adopted as its build system. Since then, many projects have been powered by Yocto Project in various domains.

Leppakoski et al.[8] have selected Yocto Project as framework to build software systems for industrial machines embedded systems. Yocto was compared to other open-source frameworks such as Buildroot[16] and was selected because it supports a wide range of hardware and processor architectures and it minimizes development and maintenance costs.

Geréb et al.[42] used a Yocto based embedded computer to handle the sensors of their noise monitoring device as those sensors could not be added to the main system.

Finally, some Arduino boards and extensions such as Intel® Galileo Gen2[6] and Intel® Edison[5] use a Yocto distribution as software system. The first one was used by Ryser et al.[35] to control a wearable robot hand orthosis designed for rehabilitation and the second was used by Dabran et al.[23] to build a mobile hot spot relay, a small car able to crawl in disaster areas to provide Wi-fi support.

2.3 ROS in swarm robotics

Communication is an essential aspect of robotics, especially swarm robotics where all the interest of the technique resides in the interaction between the robots. However, communication is usually handled locally, usually depending on the available hardware. For example, a robot like the e-puck can communicate through a range-and-bearing device but also via color signals using LEDs and a camera.

This has an important downside : to use a controller developed for an e-puck on any other robot, we need to adapt the whole communication layer to this particular robot capabilities. While this is also true for other sensors and actuators, communication is used in nearly any robotics project. A standardized but still efficient development framework is then a huge benefit for collaborative work. This is exactly what proposes ROS : various robotics applications such as mapping or steering connected together thanks to a common development framework based on message publishing. ROS works similarly as forums on the Internet. ROS applications, called nodes, can publish (send) messages to topics where those messages will be seen by any

node subscribed to (listening to) this topic. Data can hence be exchanged between applications in a broadcasting way.

Numerous projects involving ROS exist in the literature. Some papers present new robot systems for educational or research purposes designed to work together with ROS such as the ReMoRo platform of Karimi et al.[72] or the Arduino based robot of Araújo et al.[4]. Some works also exist on the e-puck, which is also a robot first designed for education and research purposes. GCtronics provide a C++ driver to run ROS on the e-puck[41] and Florea et al.[33] experimented on e-pucks a new generic controller based on ROS to handle multi-robot interactions.

Besides its features, ROS popularity comes also from its wide and active community of users throughout the world. Applications using ROS are now countless and it might seem at first glance that there is no other viable alternative. This is actually almost true because most similar frameworks are designed for a very specific use contrary to ROS that is generic and provides a lot of packages for various applications. For example, CARMEN[95] is a collection of software dedicated to mobile robot navigation and does not support at all image processing applications.

Actually, no other framework possesses a community as big as the ROS one. However, another framework is sufficiently versatile and widely used to be fairly compared to ROS : Mobile Robot Programming Toolkit (MRPT)[96], originally developed at MAPIR lab in the University of Málaga.

MRPT provides many applications and library for robotics research, focusing especially on mapping, motion planning and computer vision. Similarly to ROS, MRPT provides C++ libraries that can be easily included in robot controllers. It has a fairly broad community and has been used in a few academic papers[52][11][44].
Yet, even MRPT can not stand the comparison with ROS. Even though, compared to most other frameworks, MRPT capabilities are varied and its community is broad, ROS provides much more different packages and its community is much wider judging by the huge number of papers referencing ROS. Comparing the efficiency of similar packages of both frameworks is not even relevant as an official wrapper providing access to MRPT packages in ROS exists.

If the use of another framework such as CARMEN or MRPT can be justified in some particular contexts, ROS is definitely the most interesting choice for the capabilities and the support it provides. Moreover, as ROS community keeps growing, it might become a necessity to use it in the future so there is a strong incentive to start using it now.

Even though ROS is commonly used in many robotics fields, very few swarm robotics applications using ROS can be found in the literature. We can indeed realize that most of the works involving ROS concern only a single robot, or a small multi-robot system, never a swarm.

The reason for that is quite simple : the usual way of using ROS communication is through a network. However, according to Brambilla et al.[15], robots in swarm robotics must be autonomous and cannot be dependant of a single global application, which means that they cannot be slaves of a centralized control nor have access to a global knowledge. This is also related to the resiliency principle stating that a swarm must be self-healing, i.e. removing an element of the swarm should not disturb it. In order to achieve this, a network is indeed problematic as it violates both the autonomy principle by adding a global control (and potentially a global source of knowledge) over the robots and the resiliency principle by introducing a single point

of failure in the system.

The few existing swarm experiments using ROS are related to very specific swarm applications : navigation in the air (drones)[61][62] or underwater[10]. This is easy to explain because 3D navigation is very complex, if not impossible, without a GPS. However, such system, such as a network, is in conflict with the swarm principles; a GPS depends indeed of a satellite. In terms of swarm, a GPS could be seen as a necessary evil and therefore it opens the door for ROS as a network is not a big conceptual problem once a GPS is accepted.

I think however that including ROS in swarm applications (other than 3D navigation), if done correctly, could bring new possibilities. There exists actually ways to include ROS in a swarm without violating (or at least not excessively) its principles. The main issue concerns the network. The network cannot be simply thrown out of the problem. It is indeed very restrictive not to use it when working with ROS. However, there is no need to rely on it neither exclusively nor all the time. It actually depends on the limitations we want to impose to the use of the network and on the application itself.

The most important observation is that using ROS should not be a mean to add multiple functions to the robot. A robot designed for a swarm is indeed simple and some works such as AutoMoDe[29] have demonstrated that a constrained robot acts more efficiently. ROS should then be a way to improve existing capabilities of the robot or bring new ways to use them such as mapping.

Then, we need to define how ROS and the network will be used and what limits we will impose to them. Here are a few examples :

- If we consider that the network offers too much liberty and power regarding inter-robots communications (long range and possibility for the robots to communicate without seeing each other), we can use a ROS based communication in which a robot will filtrate messages if their senders are not sensed by it, i.e. if they are not within acceptable range and visibility.
- If we consider than relying on a network constitutes a risky single point of failure, we can define a communication protocol based on ROS but that can switch to traditional communication medias such as range-and-bearing if the network is shut down.
- If we consider that the network allows the robots to have access to a global knowledge they are not supposed to acquire, it suffices to forbid the robots to subscribe to topics and to allow them only to publish the local data they gather.

Another interesting scenario can be depicted in an experimental context. If simulating a huge number of robots is no more a problem thanks to simulators such as ARGoS, porting the experiment to real robots can be difficult in a lab as it requires a lot of real robots that the lab either does not possess or can not handle (because of space for example). Even if is not ideal, an interesting possibility offered by ROS is to enable communication between real robots and simulated ones through topics. Hence, experiments can be performed with a smaller number of robots. This is of course not ideal but it can give a good idea of the result of the experiment.

Finally, some useful tools of ROS can bring value to a swarm even without any use of the network. The best example is the *rosbag* tool[99] that allows to record any messages published within a defined period. This process creates bag files that can be replayed at will to study

the data or to simulate the behaviour of the robot. As ROS can work with the local network of a computer (for example with the loopback address 127.0.0.1), we can imagine a scenario where the robots will explore an area, record their sensors data thanks to *rosbag* and messages published on their respective local networks, and finally return back to their nest where the bags of each robot will be gathered and replayed to create a map of the area. For example, this could be helpful in case of an earthquake as robots could crawl in disaster areas to map them or to report the presence of victims thanks to some thermal sensors.

There is a second problem related to ROS that occurs also in multi-robot applications : the standard way of starting ROS is to create a core node, the master, unique in the whole network that will handle all other ROS applications. It is impossible to run 2 masters at the same time and masters on different machines are not able to communicate. However, this apparently very restrictive problem can be solved very easily thanks to the *multimaster_fkie* package[98]. As its name says, this package allows multiple masters to communicate. It even allows to select the topics to share. Even better, the package works with a discovery/synchronization system that allows new masters to be added or discarded on the fly. Robots can then easily access to the shared topics and a robot failure will not impact the swarm, respecting thus the resiliency principle.

Even if it needs a careful adaptation, I believe that ROS can be a valuable tool for swarm applications development.

2.4 Mapping

Mapping is a complex process consisting of creating a graphic representation of the significant features of the environment. This task is usually done by a single powerful robot equipped with multiple sensors and even a laptop such as the Pioneer III used by Peiliang et al.[109] to build a map of a home and help the robot navigate in it. Self-localization, studied for example by Costa et al.[19], is indeed often coupled with mapping to develop delivery, home caring or surveillance applications. While those tasks are usually performed inside a building, outdoor experiment have also been conducted like the street-mapping robot designed by Irie et al.[51].

Many software exist to build maps and most of them have been designed for a specific robot or a specific application but some generic, configurable tools or commonly used such as ROS packages like *GMapping*. A lot of mapping experiments are done with ROS such as the review on real-world mapping with ROS-based multi-robot system realized by Garzon et al.[36] or the distributed multi-robot map fusion algorithm developped by Zhang et al.[112]. ROS provides indeed a standardized interface for sending and receiving data as well as many tools to handle mapping related files and devices that simplify the design of mapping experiments and robots controllers.

Some experiments with more than one robot can be found such as those conducted by Lee et al.[58][59] with 3 Pionneer III robots. However, those experiments can not be considered as swarm robotics considering the complexity of the robots and their access to centralized global knowledge (Internet, GPS, etc). Those experiments actually focus on map merging algorithm development rather than multi-robot application. Other experiments feature different types of robots to achieve mapping such as Nam et al.[74] that use a ground robot and a drone to build a pseudo 3D map of en indoor environment. However, if a few multi-robot experiment can be found, no swarm experiment on mapping have been conducted so far (to my best knowledge).

This is probably due to benefits of complex centralized systems such as GPS in this kind of applications that are not allowed by swarm robotics philosophy.

Hence, while mapping can currently be achieved with high quality by super-equipped complex robots, few experiments involving a significant number of robots have been conducted. Despite the availability of very decent map merging algorithms, multi-robot and moreover swarm mapping are probably restrained by technical limitations (especially for real robots applications) and/or conceptual barriers.

2.5 Exploration behaviours

Building a map requires first that the robot explores the environment and gather data about it. While a lot of mapping experiments use a single human driven robot, autonomous robots should follow some exploration behaviour in order to move in the experiment space. Many exploration behaviours are based on randomness and are inspired by biology or physics. Hence, bacterial chemotaxis[31] is based on bacteria reaction to chemical stimuli, Brownian motion[32] was first used to describe the movement of particles in a fluid, etc. One of the most well known strategy is called Lévy walk[111] and is frequently used as basis for more advanced strategies such as the Yuragi based technique mixing Lévy walk and bacterial chemotaxis[75].

While many experiments have been performed on a simulator with a single robot, some of them study those behaviours on real robots, such as Emery et al.[27] that tested an improved version of Lévy walk on a Khepera IV robot, or with a big number of robots such as Yang et al.[110] that evaluated a bacterial chemotaxis algorithm with 30 simulated robots. The closest experiment to swarm applications is the experiment of Dimidov et al.[24] that evaluated common exploration behaviours with 100 kilobots.

Although very important in real exploration situations, obstacles are not often considered in random walk experiments. Obstacle avoidance is however a key parameter in environment mapping and many different algorithms exist. However, most techniques are based on simple algorithms such as the vector field histogram[13][14] that represents obstacles as repulsive forces for the robot or the curvature-velocity method[92] that treats obstacle avoidance as an optimization problem. It is interesting to notice that contrary to most exploration experiments, obstacle avoidance algorithms are usually tested on real robots.

Chapter 3

The e-puck

An e-puck is a small differential wheeled robot originally designed by Michael Bonani and Francesco Mondada at EPFL in Lausanne for education and research purposes[28][73]. It is now developed by GCtronics[39], a spin-off of the Autonomous System Lab at the EPFL. It is fairly inexpensive[38] compared to similar robots and gather therefore a wide community of users throughout the world. The e-puck robot is then an interesting platform for many applications.

The rest of this Chapter is structured as follows : Section 3.1 details the hardware architecture of the e-puck robot; Section 3.2 presents the software architecture of the e-puck robot; Section 3.3 discusses the benefits as well as the limitation of the e-puck robot.

3.1 Hardware architecture

The basic version of the robot features a restricted set of sensors and actuators but e-pucks used at IRIDIA are upgraded with some extensions[54]. They are equipped with an extension board provided by GCtronics[40]. This extension board includes, in addition to some additional sensors/actuators such as a camera, a Wi-fi dongle and a Gumstix Overo COM (Computer-On-Module). The usually used model is EarthSTORM[48], a quite powerful embedded computer.

The COM runs Linux and features in particular an ARM processor (ARM Cortex-A8[7]) clocked at 800 MHz and a slot for microSD (uSD) card. A uSD card can then be used to increase storage and install a custom Linux distribution, Ångström in this case. The board uses an Edimax Wi-fi dongle[26] to access to the internet.

Here is a list of the most noticeable sensors and actuators of the upgraded e-puck. The list is hence not exhaustive, some devices not being used at IRIDIA.

- Sensors :
  - 8 infrared proximity sensors displayed around the circular body of the robot. They can detect obstacles within a small range (a few cm) as well as light intensity.
  - A ground sensor that detects gray-scale color variations on the ground.
  - An omni-directional camera on top of the robot that provides 360°view (extension).

Figure 3.1 – Upgraded e-puck used at IRIDIA[54]

- Actuators :
  - The motors of the 2 wheels that can produce a speed from 0 to 18 cm/s independently of each other.
  - A ring of 8 LED's around the robot.
  - A range-and-bearing device for local communication between robots (extension).

3.2 Software architecture

The distribution currently used at IRIDIA on the Overo COM is Ångström. Ångström is a Linux distribution designed for embedded systems. It is the result of the work of developers from other projects such as OpenEmbedded[76]. This distribution has been used in many embedded systems projects but is not very common today. Hence, finding information about Ångström has become difficult as the official website is not maintained any longer. The most complete source of information is the Ångström Manual written by Luca Merciadri[69].

The distribution pre-installed on e-puck robots features a 2.6.32 Linux kernel providing a wide range of useful functionalities such as *ssh* connection. ARGoS can be installed on Ångström and interfaces with the robot sensors and actuators have been designed. ARGoS controllers developed for simulations with e-pucks can then be reused on the real robots with no or few modifications.

3.3 System analysis : strengths and limitations

E-pucks are a very useful tool for experimental or educational purposes. They are cheap, easy to handle because of their small size but still equipped with many different sensors and actuators that allow a wide range of different applications. They are hence especially fitted for swarm experiments and their integration in ARGoS swarm simulator make them ideal candidates. Many experiments in this field are indeed conducted on e-puck robots, a major one being AutoMoDe.

However, e-pucks present some weaknesses. First they are quite fragile and should be manipulated with precaution to avoid damaging them. The bottom battery connectors are especially easy to break.

Then, the autonomy of the e-pucks is quite short, about 15 minutes will fully charged batteries. But this autonomy is highly affected by the external temperature. In a hot room, the autonomy can decrease below 10 minutes. Moreover, the performances of the robot tends to decrease when the batteries reach a low level (and, at the same time, a high temperature). Performances seem also to be negatively affected by the heat.

Finally, the e-pucks sensors sometimes lack precision. Contrary to a simulation robot, the sensors vary greatly in their sensitivity and it is frequent that some of them are barely sensitive compared to the others, or on the contrary way more sensitive. This should be taken into account in the experiments as this can greatly impact the results if they rely a lot on those sensors.

Nevertheless, all those inconveniences are manageable. Manipulating e-pucks with precaution, defining adapted experimental duration or calibrating the robots sensors are simple examples of solutions to those issues. Moreover, those problems are quite common when working with small simple robots. In the context of swarm robotics, the individual robots are not important and defects in one e-puck will be globally corrected by the other robots of the swarm.

There is however another concern about e-pucks that is more problematic : the Ångström distribution installed on the robot embedded computer is old and hard to work with. Major obstacles to development under Ångström are the critical lack of documentation and the absence of regular maintenance and recent updates of the distribution.

The lack of documentation is a major obstacle when working with any software or hardware component. Often, unclear documentation leads to unexpected errors and time wasting. However, the issue is particularly severe with Ångström as finding documentation about it is a challenge in itself. Indeed, the official website seems completely abandoned[115] since September 2015 (date of the last post) and its content has been wiped. The community does not seem very active neither as very few topics about Ångström can be found on online forums.

The only accurate documentation that is available is the Ångström Manual[69] written by Luca Merciadri. While quite complete regarding the installation and the setup of an Ångström system, this manual does not tell much about adding new packages (such as ROS). The manual also dates back from 2009 and does not include the last known updates such as the compatibility with Yocto Project. Most of all, a lot of links in the manual are broken or lead to the official website that does not contain anymore information.

This dramatic lack of documentation makes developing on Ångström really difficult. On top of that, finding help is becoming more and more difficult as it seems that the community of Ångström users is decreasing, most of the latest topics on the subject date back to 2012 (or 2014 if Ångström distribution built with Yocto is considered).

Besides the lack of documentation and the vanishing of the community, another problem shows up regarding the fact that the latest information dates back to 2015. Indeed, when looking at Ångström latest specifications it can be seen that some packages and component of the distribution are quite old. The best example is the kernel, its version being 2.6.32. This is quite old considering that Yocto Project can build embedded system distributions with a 3.X or even a 4.X kernel.

However, as embedded systems do not always require the newest version for every package, this can be acceptable and some projects continue to use Ångström like Toradex COM[100].

The real problem is that Ångström core project does not seem to be updated anymore, which makes it very difficult to use new tools requiring new versions of some components. Hence, Ångström might not be the best option for embedded systems that could benefit from the development of new tools such as robots.

A quick look at the GitHub account associated to Ångström[114] suffices to convince oneself that the distribution is nearly abandoned :

- Most repositories have been abandoned since 2014
- When looking at the core build repository, we can see that some work has been done in early 2017. However, when considering the details, we can figure out that only packages have been updated, the core image actually dates back to 2015.
- The most recent activity is related to the Yocto layer of Ångström. However, looking closer to the manifest reveals that the README, hence the compilation instructions, dates back to 2016. These instructions are actually quite incomplete as they only explain how to initialize the repository, not how to compile everything. Moreover the Yocto release used for Ångström is a custom release, forked from the official repository which confirms that Ångström is not officially supported (yet somehow compatible).

I will now compare the activity on the meta-angstrom repository, the most recently updated software, to the activity on the meta-gumstix repository, a Yocto layer for Gumstix embedded computers.

Figure 3.2 shows 3 different activity comparisons between the meta-angstrom layer and the meta-gumstix and meta-gumstix-extras layers for Yocto project. The data come from their respective GitHub pages[113][49][50]. A first important remark is that Gumstix uses actually 2 layers in Yocto (meta-gumstix and meta-gumstix-extras) both being required for the compilation. For this reason, graphs for the 2 layers are displayed here and their combined data will be compared to those of the meta-angstrom layer. I am aware that a combined graph of the 2 Gumstix layers would have been better for the comprehension but unfortunately, GitHub does not provide the tools to do it, nor even the raw data to do it on my side. In the following, I will hence speak about the Gumstix layers as if they were a single combined set of data.

Figures 3.2a and 3.2b plot the contributions to the master branch of the repository from its creation until today. It is important to notice that the scale is different for the 3 graphs. However, if we compare the number of contributions at the creation of both Ångström and Gumstix layers (respectively around 2011 and around 2013) we can observe a similar amount. The period from the creation until 2017 is also quite similar for both and looks typical for such repository. Indeed, few contributions are registered along the years and a peak appears when a new version is released. However, Gumstix registers a similar amount of contributions to Ångström during this period while having been created 2 years after the meta-angstrom layer. Finally, if we look at the data around 2018, we can clearly see that the Ångström layer has nearly been abandoned since 2015-2016 while the Gumstix layers still receive regular contributions.

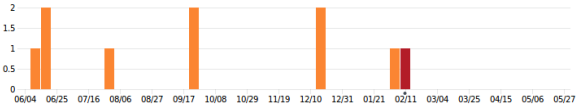
Figures 3.2c and 3.2d present approximately the same data but on the basis of additions and deletions (grouped in the following under the term "modifications") in the code.

(a) Contributions to master repository over the 8 last years for meta-angstrom

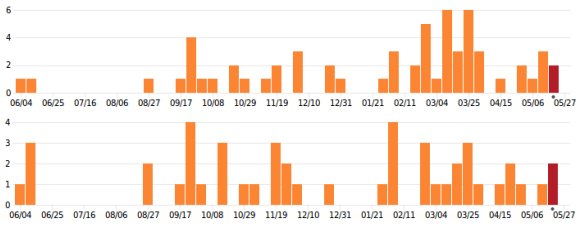
(b) Contributions to master repository over the 6 last years for meta-gumstix (up) and meta-gumstix-extras (down)

(c) Additions (green) and deletions (red) over the last 8 years for meta-angstrom

(d) Additions (green) and deletions (red) over the last 6 years for meta-gumstix (up) and meta-gumstix-extras (down)



(e) Commits over the last year for meta-angstrom



(f) Commits over the last year for meta-gumstix (up) and meta-gumstix-extras (down)

Figure 3.2 – Comparison between the development activity of the Ångström layer (left) and the Gumstix layers (right)

This is however interesting as a contribution in the previous graphs could be a single line of code modification or a thousand lines modification. And again, the scales are very different : the Ångström graph uses a 5k unit while the upper Gumstix graph uses a 100k unit. We can hence clearly see that very few real modifications have been performed on meta-angstrom since its creation in 2011 while the Gumstix layers still receive consequent updates. As an example, the last big modification on the Ångström layer (15k deletions in mid 2015) is approximately as important as the regular updates meta-gumstix receives in 2018 (around 10k modifications).

Finally, Figures 3.2e and 3.2f plot the number of commits per week on the past year (from June 2017 to May 2018). These graphs clearly show that the combined Gumstix layers receive, in average, 4 times the number of commits received by meta-angstrom the same week. Besides, commits are posted much more frequently on the Gumstix layers (nearly every week) than on the Ångström layer (about every 2 months and it seems to decrease).

From all those graphs, we can conclude that if the Ångström layer is not completely abandoned, the latest contributions are minor (cf Figure 3.2c), infrequent (Figures 3.2a and 3.2e) and seem to be less and less frequent. As this layer is supposed to be the active part of Ångström, the rest of the distribution dating back from 2015, this does not incite me to look further for a major update of Ångström.

The lack of documentation brings a major issue : adding new packages to the distribution without breaking the system because of a wrong configuration is tricky.
Without good documentation, it is indeed really hard to perform modifications on the system without risking to update the wrong file or setting a bad value to a parameter. This is a huge waste of time when trying to improve the system.

Finally, the absence of support highlights the problem of outdated content. As Ångström core image will probably not be updated anymore, inevitably the time will come when nothing new will be possible on the system because it will be based on an aged structure too old to handle it. And this time is not far away as I already experienced some limitations with the difficulty of installing ROS on the system.

Those issues lead to a conclusion : Ångström is deprecated. If not yet, it will be in a near future. This conclusion tends to be confirmed by Gumstix which states in the tutorial to set up Overo COM that : "Ångström images are no longer in development, as Gumstix Software Development is currently transitioning from the classic OpenEmbedded build system to the Yocto Project build system."[47].

A final technical problem is related to the Edimax Wi-fi dongle. Without entering too much into the details, the major issue with the Edimax EW-7811Un dongle[26] comes from the driver of the Realtek[87] RTL8192CU EW-7811Un it uses. This driver is indeed known for many compatibility issues with Linux, especially with kernel version 3.X and 4.X. As the driver is not referenced anymore on Realtek website, no official solution exists. A good patch can be found on GitHub[105] but it is not fitted for embedded systems as there are no explanations related to cross-compiling. Ultimately, the most viable solution is to compile a custom generic Realtek driver. This operation is complex and troublesome and is specific to the distribution. Hence, as this could still cause problems in the future, some replacement dongles should be considered.

However, most Wi-fi adapters provide a proper Linux support only for old kernel versions (2.X in general), if they provide any. It is hence quite difficult to find dongles that will work without too much troubles. The safest approach is probably to seek for adapters using drivers

already supported in Yocto. Through some research, 2 drivers supported by Yocto seem to be frequently used in adapters known to work on embedded computers such as Raspberry Pi : the Ralink RL3070 and the Atheros AR9170.

The following list enumerates a few adapters implementing one of those drivers.

- AWUS036NEH and AWUS036NH from ALFA Network Inc. (RL3070)
- F6D4050 V1/V2 from Belkin (RL3070)
- TL-WN822N v1.1 from TP-LINK (AR9170)

As it is highly difficult to find working examples for a specific architecture such as the Overo board used on the e-puck, this selection is based on a list of dongles working on Raspberry Pi[107]. The aim here is only to give some leads in case of future Wi-fi issues. Until now, the Edimax dongle works well.

Chapter 4

ROS for the e-puck

Developing ROS-based software on the e-puck robot is the main objective of this thesis. However, to develop ROS-based software for the e-puck, ROS must be first integrated in the controller of the robot. This should be done both for the simulator and for the real robot. From many aspects, this is a complex process that requires thoughtful design choices and manual adaptations.

While integrating ROS in ARGoS controller is not difficult, installing it on the real e-puck is complicated and maybe impossible on the current system, mainly because of Ångström deprecation. Alternatives must then be considered, for the operating system of course but also for the embedded system itself (the Gumstix Overo board) or even for the control software, ARGoS, if there is no other choice.

The rest of this Chapter is structured as follows : Section 4.1 discusses the different problems encountered with Ångström and the solution found to replace it; Section 4.2 presents available alternatives for the Gumstix Overo board; Section 4.3 describes procedure to integrate ROS in the e-puck software system; Section 4.4 evaluates other possible controllers for the e-puck and explains how ROS can be integrated in ARGoS.

4.1 A new embedded distribution

Installing ROS on the e-puck is not a simple task. A first attempt to install ROS would be to boot the robot, connect it to the internet and download directly ROS. However, this is not realistic because :

1. even if the package manager of Ångström is powerful enough to download ROS, it will take a lot of time that will largely exceed the duration of the battery;
2. we will have to compile ROS on any new robot we want to setup instead of adding it to the image.

In order to install ROS on Ångström, it is then necessary to cross-compile it so it can be read by the Ångström kernel. Cross-compiling is the process of compiling code for a target distribution

while being on another, different distribution. In this particular case, the objective is to compile ROS for Ångström while working on a computer running Ubuntu.

After 2 months attempting to achieve this task, I realized that its complexity was a lot higher than expected. Cross-compiling is indeed a difficult process as it requires a lot of configuration to get the desired outcome. This is especially true with embedded distributions as there are many of them and from the point of view of cross-compiling, those distributions are all completely different.

There are actually numerous issues related to cross-compiling in Ångström. The most important ones are discussed below.

The most common problem is that there is no specific procedure for Ångström. Most of the time, installation tutorials found on the internet does not describe a procedure for a machine running Ångström. There is then no other option than trying to adapt a procedure dedicated to another distribution or even to a specific machine such as BeagleBone Black[9]. At best, some paragraph explains briefly how to adapt it to Ångström but it is barely enough.

Adapting such low level processes is highly difficult and does not even work in general. The absence of a procedure dedicated to Ångström for ROS installation is then a major obstacle. An illustration of this problem can be seen on the ROS for BeagleBone page[63].

Another frequent problem is that most of the tutorials that can be found are outdated. First, many procedures explain how to cross-compile ROS Hydro but this version of ROS is no more maintained since 2015. One does not even know if newer ROS distributions are even compatible with Ångström.

A more important problem with outdated procedures is that they usually do not work anymore. For example, the procedure described in ROS official wiki[64] is full of bugs and despite I was able to solve some of them, a successful completion of the whole procedure was not possible.

While it is sometimes possible to adapt old procedures to current systems, when they are too old it is usually pointless. In the particular case of cross-compiling ROS for Ångström, most procedures are from 2012 or earlier which is definitely too old.

A last common problem with ROS cross-compiling procedures for Ångström is the lack of explanations. Some procedures found in the documentation seem promising but unfortunately they barely describe the different steps and it becomes tricky to guess the missing information. This is especially true with a complex process such as cross-compiling where there are a lot of parameters to adjust carefully in order to achieve even a simple task. An example of this problem can be seen in the instructions to install ROS Hydro on Ångström[65].

The 3 problems sketched above are separately real obstacles to the cross-compiling of ROS on an Ångström distribution as, to my best knowledge, no procedure without any of those failures exists. But in practice, it is even worse because the majority of the documentation about ROS cross-compiling suffers from 2 or even the 3 problems at the same time. Actually, the example procedures presented for each problem above suffer from at least 1 other failure.

This brings us to a situation where it is almost impossible to cross-compile ROS on an Ångström distribution. Moreover, Ångström suffers some other weaknesses that were presented in Section 3.3. It is then necessary to replace Ångström by another distribution. A report from the Technical Univeristy of Munich[78] also goes in the sense of deprecation and encourages the use

of a Yocto based distribution.

Selecting a new distribution is a complicated process as a wide variety of embedded software exist and comparing them all is long and difficult. Yocto Project is the reference tool to build an embedded system distribution and it was hence an interesting solution. Indeed, Yocto allows to build custom distributions and provides installation support of ROS.

I stopped then my choice on the Poky distribution with a 3.5.7 kernel built with the 1.8.2 release of Yocto. Poky[82] is the reference embedded distribution for Yocto Project. Although it is not usable as such for most possible applications, it provides a strong base to create custom distributions. As the architecture of Yocto is very flexible and modular, components can be easily added to Poky to satisfy the need of a specific application. This modularity motivated my choice for this distribution. As no particular distribution was specifically required, a simple, customizable distribution seemed the best choice to me. Such a distribution would allow me to easily integrate the other ingredients that I needed, ARGoS, and ROS and could be as easily modified and improved for further work. Moreover, if a particular distribution is required in the future, moving from a generic distribution such as Poky to the new distribution would be a lot easier than changing between 2 specific distributions.

Additionally, the meta-gumstix layer installation instructions[97] that I used as starting point for my work are based on the Poky distribution.

The kernel choice was motivated by 2 observations. First, images including ROS and built with Yocto are available on the Overo extension web page [40] as a solution to use ROS on the Overo COM. Those images were using a 3.5.7 kernel. As this building process is very difficult and time-consuming, even with Yocto, I thought that it was more relevant to base my work on a working example before trying more recent kernels.

The other observation is that the OTG port of the Overo COM which hosts the Wi-fi dongle does not seem to work on the 3.18 kernel that was also available with the used Yocto release.

As no one knows if the OTG port problem would be solved on newer versions such as 4.X and considering the importance of the Wi-fi dongle for ROS, I selected the 3.5.7 version as it is already a big improvement from the 2.6.32 kernel used by Ångström.

The Yocto Project version used for this work is the 1.8.2 Fido release[84] (and hence Poky 13.0). As the latest stable version of Yocto Project is 2.4.2 Rocko, this choice needs some motivation.

First, Fido is the version used in the Gumstix Yocto manifest[97] and the files downloaded on Overo extension web page[40], both used as references for the installation procedure define in this thesis. The first objective being to make it work, it was logical to stick with the version used in those procedures before trying to modify it (as long as Fido is not deprecated which is not yet the case).

Then, features brought by newer versions of Yocto Project[83] are not useful or necessary for the objective of this work. They could then lead to irrelevant additional bugs in the installation procedure.

Finally, a more pragmatic argument is that newer versions of Yocto have moved to 4.X kernels. As I chose a 3.5.7 kernel, it was necessary to use a Yocto release that still provides it without additional manipulations.

	No Wireless Communication	WiFi 802.11b/g/n Bluetooth 3.0	WiFi 802.11b/g/n Bluetooth 4.1 + BLE
Base Model	EarthSTORM	AirSTORM-P	AirSTORM-Y
PowerVR SGX C64 + Digital Signal Processor	WaterSTORM	FireSTORM-P	FireSTORM-Y
PowerVR SGX C64 + Digital Signal Processor Extended Temperature Components	IceSTORM	IronSTORM-P	IronSTORM-Y

Figure 4.1 – General comparison of the Overo series COMs[48]

4.2 Extension boards for the e-puck robot

The embedded computer used on the e-puck is a Gumstix Overo EarthSTORM COM. While complex at first glance, changing the embedded computer of the e-puck is a worth considering option to make ROS available on the e-puck. A comparison with other potential candidates is then important. I will discuss 2 different alternatives : comparing the Overo EarthSTORM to other COMs of the Gumstix suite and comparing the whole board to totally different boards. In these comparisons, we should keep in mind that changing the board COM is additional work that adds to the complexity of the ROS installation process, which is already quite complex on an embedded system. Changing the board has then an interest only if the new board can bring crucial features to the system.

The Gumstix COMs are divided into 3 different series : Overo, DuoVero and Verdex Pro. As Gumstix COMs are similar from the electronic point of view, changing the EarthSTORM COM for another one should not be a too complex procedure, especially if we change it for a COM of the Overo series. It is then worth discussing it but, as stated above, a change will still require additional work and will hence be approved only if it brings real benefits.

According to the Gumstix official website[48] “the processor and memory components on the verdex pro product line are imminently approaching EOL”. Hence, the Verdex Pro COMs are then out of consideration.

The Gumstix Overo series is organized in 3 families : the xxxSTORM family, the xxxSTORM-P family and the xxxSTORM-Y family. Each family comprises 3 different COMs with different level of sophistication. The EarthSTORM COM that is used with the e-puck is the most simple of all 9 COMs.

As it can be seen on Figure4.1, the only differences with the 2 other COMs of the same family, WaterSTORM and IceSTORM, are a digital signal processor, graphics acceleration and more temperature resistant materials. As these additional features are not particularly useful for the use of the e-puck, the EarthSTORM COM capabilities suffice.

The 2 other families, P and Y, are basically the same 3 COMs but with respectively Wi-fi and Bluetooth 3.0 integrated for the P family and Wi-fi, Bluetooth 4.1 and Bluetooth Low Energy

Performance	DuoVero™ Crystal COM Up to 2,500 Dhrystone MIPS	Overo® EarthSTORM COM Up to 2,000 Dhrystone MIPS	Overo® WaterSTORM N/A	Overo® IceSTORM COM Up to 2,000 Dhrystone MIPS
Power	Powered via expansion board (DuoVero series or custom) connected to dual 70-pin connector	Powered via expansion board (Overo series or custom) connected to dual 70-pin connectors	Powered via expansion board (Overo series or custom) connected to dual 70-pin connectors	No
Power Management	SmartReflex™ 2 technology	Texas Instruments TPS65950	Texas Instruments TPS65950	Texas Instruments TPS65950
Power via USB	N/A	No	N/A	No
Processor	Dual-core Texas Instruments OMAP4430 Digital Video Processor	Texas Instruments Sitara AM3703	Texas Instruments DaVinci DM3730	Texas Instruments DaVinci DM3730 Applications Processor
Processor Architecture	ARM Cortex-A9	ARM Cortex-A8	ARM Cortex-A8	ARM Cortex-A8
Processor Base Clock	1 GHz	1 GHz (Recommended @ 800MHz)	Up to 1 GHz (recommended @ 800MHz)	1 GHz (Recommended @ 800 MHz)

Figure 4.2 – Technical comparison between the DuoVero Crystal COM and basic Overo COMs[48]

(BLE) technology for the Y family. Again, these features are not necessary as the Bluetooth is not used by the e-puck and the Wi-fi is already provided by a dongle included in the board.

Finally, we can also compare to the EarthSTORM the COMs of the DuoVero series. However, the DuoVero COMs are basically improved versions of the 3 best COMs of the Overo series (the standard IceSTORM and the versions improved with Wi-fi and Bluetooth). As the additional Wi-fi/Bluetooth are not interesting in this particular case, I will only discuss the DuoVero Crystal COM. As it can be seen on Figure 4.2, the biggest difference between Overo COMs and the DuoVero Crystal is the dual-core processor that the latter features. Moreover, the DuoVero series COMs are shipped with a Yocto distribution installed on a uSD card instead of the Ångström distribution pre-installed on the Overo series COMs.

However, despite the advantages of the DuoVero Crystal, I chose to keep the EarthSTORM COM already equipping the e-pucks. Indeed, the dual-core processor is for sure interesting but definitely not required for a small robot like the e-puck, at least for the current missions assigned to it. Besides, the provided Yocto installation is more a convenience than a real benefit as a custom version of the distribution will anyway be needed to install ARGoS.

That being said, the DuoVero COMs are good embedded computers and if more computational power is required by the e-puck in the future, it might be interesting to consider the change with a DuoVero Crystal.

Contrary to the previous discussion, choosing a COM not manufactured by Gumstix implies to deeply modify the board itself to make it compatible with the new COM. For example, using another popular embedded computer such as BeagleBone Black[9] (which has a similar processor as the Overo EarthSTORM) would require to connect it to the e-puck base, to establish connections with the omnidirectional camera, etc which reduces basically to build a brand new board. As it represents an important overhead in terms of time and work, a more

interesting approach would be to use a completely different board providing at least the same features as the Overo extension board designed by GCtronics.

Actually, only 2 other boards have been reported : the BRL Linux extension board developed by Bristol Robotics Laboratory[60] and the recent Pi-puck extension[1] developed at the York Robotics Laboratory.

The BRL Linux extension is clearly outdated as it only features a ARM9 processor clocked at 180MHz and runs a 2.6.26 Linux kernel compared to the 800 MHz ARM Cortex-A8 of the Overo board running a 2.6.32 kernel. Besides, it is no longer maintained and the EmDebian Linux distribution that the board uses also stopped its support in 2014.

The Pi-puck extension on the other hand is more recent (2017) and is based on a Raspberry Pi Zero W with computational power similar to the Overo board. However the board runs Raspbian Jessie on kernel version 4.9 and already includes ARGoS and ROS. As this board could be seen as an obvious choice, 2 major drawbacks actually exist.

First, to my best knowledge, the board is still a prototype and is then not available on the market yet. IRIDIA also contacted GCtronics, the official e-puck supplier, and this is not producing the Pi-puck extension. Moreover, while the hardware design and the related software are supposed to be open-source, very few documents can be found on the York Robotics Laboratory website[55] and the related GitHub[56].

In addition to the availability problem and also due to its recent development, the Pi-puck extension has not been tested in experiments yet. The only publication referencing the Pi-puck extension is a one-page paper from the York Robotics Laboratory[2] that only presents the general idea of a neural network based controller for an e-puck equipped with the Pi-puck extension. On the other hand, the Overo extension board for the e-puck has been tested in numerous experiments and studies[29][34][70][68][103].

Secondly, as the Raspbian distribution used by the board can be seen as an advantage as it runs a recent kernel and it is frequently updated, it loses all the advantages brought by a Yocto distribution. Yocto allows indeed to customize and tune the distribution to fit the specific needs of the robot. This provides also a great flexibility and adaptability to add new pieces of software or update existing ones. Moreover, the agility of a Yocto based software architecture gives the possibility to change the distribution at will or adapt it to a new hardware in a single line of code. Raspbian is dedicated to the Raspberry Pi and does not provide such adaptability. I remind also that, even though the 4.9 kernel of the Pi-puck extension seems a better choice, I have chosen a 3.5.7 kernel version for the Overo to ease the transition from Ångström and because there existed working examples on this version (see Section 4.1 for more details). Further work can hence be performed in order to upgrade my solution to a more recent kernel version.

A solution could then be to use the Pi-puck extension by replacing the Raspbian distribution by a custom Yocto based distribution. However, the result is unpredictable as there is no official Raspberry Pi support in Yocto Project and hence no official BitBake recipes for such distribution. All Overo COMs, as well as the DuoVero series COMs, are officially supported by Yocto Project.

The best choice, at least for now, seems finally to keep the Overo board as it is easily available, its performances have been validated through many experiments and it is fully compatible with Yocto Project, allowing a great flexibility. The Pi-puck board seems nevertheless promising

and might be considered as an option if the concept is successfully proved in the future.

4.3 Integrating ROS in the software architecture

In order to install ROS on the e-puck, I decided to replace the aging Ångström distribution by a custom version of Poky, the reference distribution of Yocto Project.

Installing a Yocto Project distribution on the e-puck is an important part of this thesis work. To my best knowledge, it is indeed the first time that a working Yocto distribution including both ARGoS and ROS is installed on an e-puck. Therefore, some parts of the installation procedure could not rely on existing documentation and solving the new bugs and problems required a consequent amount of work. Configuring properly everything after the installation took nearly the same time.

In order to prevent other people that might want to use a similar system from the many troubles encountered during this procedure, I describe hereby the methodology I used to achieve my objective : building a Yocto distribution including both ROS and ARGoS, making it run on an e-puck and configure everything properly to meet at least the same performances and behaviours as with the Ångström distribution.

I will present here the guidelines of my method in an abstracted fashion to allow a better understanding of my work and to provide the interested reader the general workflow of this installation. Finding a full installation procedure specific to a particular architecture is quite difficult in the world of embedded systems as there are nearly as many different architectures as there are different projects. However, some issues such as a non working driver or a wrong Wi-fi configuration are commonly encountered. A general description of my work can then give some leads in case of a similar issue even if the details differ. However, a detailed description of the installation and configuration procedures can be found in Appendix E.

Building a custom distribution including all desired software is the first step to create a working robot system on the e-puck. This step was actually already documented even if I had to adapt a bit the described procedures.

Two different approaches existed : either download an already compiled image on the Overo extension webpage[40] or build everything from scratch according to the Gumstix manifests for the Yocto Project build system[97] and the ROS on Gumstix page [57]. In the context of this thesis, it is obviously the second approach that was ultimately used. However, I began by trying the first approach in order to have a working example at my disposal to help me during the custom build of the distribution but also during the configuration of the system.

Despite the apparent simplicity of using an already compiled image, this first method already contained some flaws in its procedure. Indeed, in order to bring the image on the e-puck, this image should be mounted on a microSD (uSD) card. However, the script provided by Gumstix to prepare the uSD card[46] was deprecated (and still is to a lesser extent). This illustrate a frequent issue with embedded systems : as most software in this field are open-source, updates are quite frequent and keeping everything up to date can become quickly tricky. It is then very common to find deprecated instructions in the documentation. There is then no other choice than seeking for the cause of the error and try to correct it. Hopefully, Linux users community is quite broad and we can find at least leads to solve most problems.

I was ultimately able to prepare the uSD card and this working example provided me a very useful comparison tool to build the custom distribution. Even though the Yocto recipe of this image would have been more valuable, I was still able to compare this system to the system I was trying to build in order to identify missing or useless files. This constituted a huge gain of time in the process of identifying an error and seeking for a solution

Building a custom distribution with Yocto Project is actually fairly simple. Yocto features indeed a powerful compilation tool, BitBake, that allows to customize at will the build process and to include quite easily almost any additional software directly in the installation process. In particular, installing ROS on Yocto Project requires very few operations. Things become tricky when trying to add a package for which there exists no official recipe.

Yocto provides indeed BitBake recipes for many common software such as *git* or *Python*, but of course many other packages are left without any support. While installing ROS was very simple as all required recipes were already provided and packages just needed to be explicitly added to the image, adding ARGoS was much more complicated. To my best knowledge, ARGoS has indeed never been built with Yocto and I had therefore to create myself the recipe for both *argos3* and *argos3-epuck* packages.

Creating a BitBake recipe is not an easy process without any experience of Yocto. It basically requires to specify the location of the source file, the compilation tool to use (CMake, catkin, etc) and the specific actions to perform for the compilation and the installation of the package. Three particular points require special attention :

- The path to a software dependencies should be specified with care as the Yocto architecture is a bit sprawling and finding a particular file location can be tricky without any experience.
- The compilation process in Yocto, while customizable, is quite strict and treats some warnings as errors. When building the *argos3-epuck* plugin, I discovered errors in the source code that were ignored by a classic compilation (or treated as warnings) but were pointed by BitBake as errors, preventing the compilation.
- BitBake does not tolerate debugging files outside their debug directory and will treat them as errors. In particular, this issue appears with both *argos3* and *argos3-epuck* and the faulty files need to be deleted to be able to complete the compilation process.

Using existing recipes as models to create a new one is a good way to understand the mechanisms of this unusual architecture. I heavily based my work on comparison with working examples to help me creating and debugging the recipes. Other valuable allies are error logs as their standardized format helps identifying the problem and the official documentation of Yocto which is quite complete.

Finally, it is useful to verify that the package compiles correctly outside of the Yocto build environment before trying to build it with BitBake. This allows to differentiate standard compilation errors from BitBake related errors.

The next step is to configure properly parameters related to the Linux distribution : choosing the kernel version, adding required drivers and firmware. This part is not particularly complicated but requires some time to find the location of the parameters. If most of them can be

found in the configuration (.conf) files, some of them are directly specified in kernel recipes and the search tools of Linux (*grep*, *locate*, *find*, etc) prove themselves to be efficient tools.

Once the recipes are written and debugged and Yocto is properly configured, the rest of the procedure should run without any problem. Packages just need to be added as dependencies of the image recipe and the BitBake tool will compile everything when building the image.

To summarize, the general steps of the building process are :

1. Download and initialize the Yocto build environment
2. Create (and debug) missing recipes for all desired packages
3. Configure the build system : kernel version, target architecture, packages to add, etc
4. Build everything with BitBake

Once the distribution is built, mounted on an uSD card and run on the e-puck, some more work is still required in order to reach the same capabilities as with the previous Ångström distribution. This configuration step was another tricky part of this method. Indeed, if ROS and ARGoS setups were quite straightforward, the Wi-fi configuration was one of the most difficult tasks of this thesis.

As stated in Section 3.3, the Wi-fi dongle caused a lot of problems before working as expected but it was not the only source of issues.

A first issue was actually related to the OTG (On-The-Go) port of the Overo extension board. The OTG is indeed disabled by default and requires an action to enable it. However, it was not possible to enable it on the Overo COM in the same way as on a desktop Linux computer.

This is another type of problem quite specific to embedded systems. Even though most of those systems are running a Linux distribution, their configuration slightly differs from desktop computers and laptops. In particular, power consumption is much more critical on those systems than it is on a classic computer. As the Wi-fi dongle of the Overo board consumes a lot of power (a warning message about power consumption of the OTG port was even showing up when enabling the port , I suspect some sort of power management on the system that forbid it to enable this port by default, forcing the user to do it. The user hence becomes acquainted with the risk thanks to the warning message. I was finally able to solve this limitation by tricking the system and forcing to enable the port, but the exact cause of this problem remains unknown.

Solving this kind of issues usually requires a careful reading of the different logs available in Linux. This is indeed the most efficient way to identify the problem which is crucial before being able to solve it.

The Wi-fi dongle caused problems because of its driver. The Realtek RTL8192CU driver is indeed known for having many compatibility issues on Linux. Worse, this driver seems not supported any longer as there is no entry for it on the Realtek website[87]. However, as the driver was not presenting any evident malfunction, I thought for a long time that the problem was coming from the Wi-fi configuration.

Even worse, this issue was made more complicated because of a bug in the pre-compiled image provided by GCtronics that I was using as a working example. Indeed, the Wi-fi on this image was working sporadically, making me think in a first time that it was behaving well and that it was hence a good comparison tool. However, I discovered later that the connection was shutting down randomly. The only reliable tool that I had to help me setting up the system unexpectedly appeared to be unreliable.

I had then to search for similar issues reported online and try the corresponding solution, if any. However, finding a case corresponding to a particular system is quite difficult in the wide embedded systems world when the official support is not reliable.

The procedure described on the Overo extension page[40] is indeed deprecated as it is suited for the 2.6.32 kernel of the Ångström distribution and the instructions provided by Gumstix specifically for Yocto[45] images were not working at that time[1]. Most other procedures found online were either also deprecated or contradicting each other which increased even more my confusion.

This again is a common problem with embedded systems. When the official directives are deprecated or confusing, which happens quite often because of the frequent updates I mentioned earlier, people have no other choice than to rely on patchwork, temporary solutions that are very system specific and that do not always solve completely the original issue. In this case, help from other skilled people is highly valuable.

The only solution that worked was compiling with BitBake a custom generic driver. By doing this, I was able to modify the firmware of this driver and to remove the power save system that was installed on it. This power save mode was indeed cutting the network connection arbitrarily when few data were exchanged trough it.

To summarize, the general steps of the configuration process are :

1. Configure applications such as ARGoS and ROS
2. Configure the OTG port
3. Configure the network interface (typically, the *interfaces* and *wpa_supplicant* files)
4. Try to connect to the network : if it does not work, verify that the driver of the adapter is correctly loaded and installed
5. If the driver is faulty, find/compile an appropriate substitute : in general, generic versions of the faulty drivers work fine

4.4 Designing ROS controllers for the e-puck

The last point to be considered in this discussion about ROS integration is the interaction between ROS and the e-puck controller.

The controller is a piece of code that describes the actions of a robot depending on the readings of its sensors. At IRIDIA, e-pucks are controlled by an ARGoS controller. As briefly described

[1]Those instructions have recently been updated and might work properly now.

in Section 2.2, ARGoS is a robot simulator designed especially for swarm experiments. A very interesting feature of ARGoS is that controllers developed for the simulated robots can be directly used on the real robots such as the e-pucks. This allows to quickly prototype controllers on the simulator before testing them on the real robot.

Besides, ARGoS controllers are high-level description in C++ of the robot behaviour. All the complexity related to sensors reading and actuators manipulation is hidden in the structure of the code, allowing to design very clear controllers easier to debug. Moreover, as controllers need to be compiled to be run on the robot, ARGoS uses XML files to define the implementations of the sensors and actuators, which allows to test different implementations without recompiling the controller. Other parameters can also be set thanks to this file such as the maximum velocity.

Fortunately, including ROS is straightforward as ARGoS controllers are C++ code. ROS can then be integrated directly in the controller thanks to the *roscpp* package that translates ROS functionalities in simple C++ code. The base setup of ROS only consists in a few lines of code and other functionalities are really easy to use. AppendixB.5 presents a simple code skeleton illustrating how ROS is included.

Concerning the compilation of the controller, ARGoS controllers are usually compiled using CMake[53] as compilation tool. ROS projects are usually compiled using catkin[101], a CMake-like tool optimized for ROS that also creates ROS specific files. However, only few statements related to ROS need to be added in the CMake file to make it work, once again nothing complicated has to be done. An example of a typical CMake file including ROS packages can be seen in AppendixB.5.

While ARGoS may seem an obvious choice, other solutions exist to control the e-puck and it is important to evaluate the relevance of those alternatives

The list of potential candidates is however very restricted and includes only 2 alternative choices : Webots[22] controllers and the ROS driver for e-puck designed by GCtronics[41]. Indeed, using another controller means that if this controller does not provide an abstraction level comparable to the one ARGoS provides, all sensors/actuators readings and writings will need to be handled by hand. As such work is quite technical and complicated, the probability of error is high and the gain would be negligible as ARGoS controllers are known to work very well (such as in the Swarmanoid project[30] or AutoMoDe[29]).

A first interesting alternative would be to use the Webots simulator as it already implements the e-puck as well as its extended version with the Overo board. It also provides a cross-compiling tool to port the code on the real robot and the controllers are written in C++ or Python (or Java) making them compatible with the ROS C++/Python API. However, Webots is a proprietary software contrary to ARGoS which is open-source and custom plugins can hence not be developed for the Webots simulator. Using Webots makes hence the system less flexible.

Moreover, Webots is accuracy-oriented and therefore it does not provide a good scalability in the simulation. As this thesis targets in particular multi-robot/swarm applications, ARGoS seems a better choice for its swarm-oriented features. Indeed, even though this thesis describes a methodology to improve real e-pucks, simulation is still important to prototype controllers.

The second available option is to use the controller developed by GCtronics to include ROS functionalities to the e-puck. Although this work was very useful to design the controller used in experiments, the driver of GCtronics is hard to use as such. Indeed, it is a stand-alone driver

that is not related to any simulator. It is hence not possible to test it with ARGoS which means new controllers based on it will need to be tested directly on the e-pucks.

Moreover, the driver does not implement ARGoS structure and can hence not benefit from the abstraction it provides, making the controller very hard to understand (also considering that it is not well documented). Additional sensors or actuators would also need to be implemented by hand. Finally, ARGoS XML configuration files that are very useful to quickly test different values of parameters and different sensors/actuators implementations are not usable with the driver.

This controller is hence a good working example of ROS on an e-puck but definitely lacks of flexibility, reusability and modularity.

A few words can be added about the ARGoS-bridge plugin[104]. This plugin for the simulator aims to connect it to ROS. However, it is designed for foot-bot (according to the sensors/actuators names) and it is not frequently maintained as only 4 commits have been performed since its creation 2 years ago. Finally, this plugin seems to be designed for the simulator only which is not very useful for the purpose of this thesis.

ARGoS is then the easiest and most interesting choice from performances point of view to control the e-puck while using ROS. A brief motivation about the ARGoS release version used in this thesis needs to be given. The version installed on the e-puck is 3.0.0-beta48. However, the 3.0.0-beta50 was just released on May 18th 2018. This choice is actually motivated by an incompatibility between the e-puck plugin (*argos3-epuck*) and the last versions of ARGoS. The last supported version being, 3.0.0-beta48, this specific release was selected.

The only remaining task is to define a new procedure to cross-compile the ARGoS controllers for the e-puck. The previous procedure for Ångström (detailed in Appendix A.2.1) basically consisted in a specific compilation process specifying the cross-compiling tool. The executable controller had just then to be transferred to the real robot via ssh.

This procedure needed to be adapted to the new distribution as the old cross-compiler was designed for Ångström. However, thanks to Yocto the new cross-compiling process is fairly easy. Controllers can indeed be treated as packages and compiled with a BitBake recipe. The executable can then be send to the robot via ssh, as before.

A particular point needs however some attention : to compile an ARGoS controller including ROS implementations, catkin must be specified as compilation tool in the BitBake recipe. For a pure ARGoS controller, either catkin or CMake can be used, depending on how the CMake file is written.

To summarize, the general steps to build a controller are :

1. Write a recipe for the controller

2. Verify that the right compilation tool is specified

3. Build the executable with BitBake

4.4.1 ROS interface for the e-puck

As an example of ARGoS controller using ROS, I present here an interface providing a full communication with the e-puck through the ROS development framework.

The interface consists in a set of ROS publishers and subscribers, each connected to a particular sensor or actuator. Hence, 4 sets of publishers and 2 subscribers are created :

- 2 sets of 8 publishers for the 8 proximity sensors;
- a set of 8 publishers for the 8 light sensors;
- a set of 3 publishers for the 3 ground sensors;
- a subscriber for the wheels;
- a subscriber for the RGB LEDs.

With such configuration, the data of each sensor can be directly published on the appropriate topic : /proximity, /light and /ground. The readings are embedded in a standard ROS message (such as Range or LaserScan) together with a header. The sensors of a same group can be differentiated thanks to their ID in the header of the ROS messages. For example, the light sensor 5 will send its readings to the /light topic in a ROS message with a header containing the string "base_light5".

An additional topic, /scan, is linked to the proximity sensors : while the raw readings of the proximity sensors with values from 0 to 1 are published on the /proximity topic, those values are then remapped to real distances in meters between the minimum and maximum ranges of the sensor. Those scans are very useful for navigation purpose.

Concerning the actuators, each subscriber is linked to a specific handler that will perform an action each time a message is received.

The wheels are subscribed to the /cmd_vel topic on which velocity messages are published. The associated handler will set the velocity of the e-puck wheels according to the received linear and angular speeds.

The LEDs are subscribed to the /color topic on which RGBA values are published. The handler will simply set the color of the LEDs according to received values.

It should finally be noticed that both /cmd_vel and /color topics use also standardized ROS messages (respectively Twist and ColorRGBA).

The test consists then in verifying that sensors readings are correctly published on the corresponding topics, that the robot can be remotely steered and that the color of the LEDs can be set to any possible color.

The interest of such interface is to show that it is possible to abstract the communication with an e-puck to the level of ROS. Hence, it is possible for an external application to get any sensor value by subscribing to the corresponding topic or to send instructions to the robot by publishing on the appropriate topic. Moreover, as all messages used with this interface are standardized, it is possible to make the e-puck communicate directly with other standard ROS

applications : it is for example possible to remotely control the e-puck with the steering tool of the *rqt* package.

To test the interface, all 3 sets of sensors (proximity, light and ground) were calibrated. The sensors readings were then checked one by one : obstacles were placed near the robot for the proximity sensors, light conditions were changed for the light sensors and the e-puck was placed on different colored floors to test the ground sensors.

Then the robot was remotely controlled with steering tool of the *rqt* package and finally some color messages were send to test the e-puck LEDs.

All performed tests actually succeeded. As the readings collected for the different sensors were coherent, it means that the sensors were well behaving. However, as ROS was not interfering with the sensors readings, the fact that messages were published on the correct topics was sufficient to verify that the interface was working as intended. The only exception concerns the LaserScan messages as in this case, values were remapped before being published. However, the results were still coherent.

Concerning the actuators, the robot was well responding to the steering and colored its LEDs adequately to the values received.

This test was then very successful as every part of the ROS interface behaved as intended. While this result was expected considering the relative triviality of the tasks performed by the robot (nothing different from usual experiments), this proves however that ROS is well integrated in the ARGoS controller and offers new communication options. As the sensors and actuators used were also quite different from each other, this gives the intuition that adding new sensors/actuators to the robot and to the interface should not be a complicated task.

Actually, there is no direct application of such demonstration. However, its successful result opens the doors for many other applications :

- By adding an odometry support, we can do mapping.
- By recording all the messages thanks to *rosbag*, we can collect information about the robot perception and reproduce it easily by replaying the bag file.
- By adding a thermal sensor, the e-puck could detect the presence of a living being and report it immediately to an operator subscribed to the sensor topic.
- etc

Of course, all those examples are just general ideas and do not target a particular implementation.

Chapter 5

The e-puck robot for mapping

ROS brings to the e-puck a wide variety of new capabilities. Those new features provide to the e-puck the necessary tools for performing better in some tasks but also to achieve tasks that it was not able to fulfill before. Mapping is a complex task, hard to handle without ROS for a small robot like the e-puck. The mapping process consists in exploring an environment and creating a 2- or 3-dimensional view of this environment. This view, the map, can then be used by the robot to locate itself in the environment, by a human to retrieve information or by both. Though it is a widely studied field in robotics, most research rely on big complex robots and drones for this task.

The rest of this Chapter is structured as follows : Section 5.1 presents the mapping task and its interest for the e-puck robot; Section 5.2 details the different exploration behaviours that will be studied in this work; Section 5.3 describes the design of e-puck controllers to perform mapping; Section 5.4 discusses the benefits as well as the limitation of the mapping process on the e-puck.

5.1 A new capability for the e-puck: mapping

Creating a map requires two elements : data gathered by the robot sensors during exploration and a tool that will build the map using the data. The data used for mapping are measurements of the distance between the robot and surrounding obstacles called scans and measurements of the robot movement called odometry.
Scans can be obtained using the readings of the robot sensors such as proximity sensors or long-range scanner. The e-puck are equipped with multiple sensors that could produce scans but for this work, the proximity sensors were selected because of their simplicity. Such sensors are indeed quite common in robotics and if an e-puck is able to produce a map using those simple sensors, any better sensors could be used to improve further the results.
The odometry is often calculated by a dedicated feature usually located in the motor. However, the e-pucks used at IRIDIA do not possess this capability and the odometry was then calculated in the controller. At each time-step, modifications to the robot position and orientation were approximated based on the robot wheels velocities. This approximation has an arbitrary position and the impact of this parameter is analyzed in Section 6.4.1.

Even though e-pucks can produce the required data, they possess no tool to create the map. A

solution could be to record the data and create the map on a computer after the exploration. An e-puck running an ARGoS controller can record experiment measurements using ARGoS functionalities but creating a good quality map requires a huge amount of data that is hard to handle this way. Moreover, even if it was possible to record efficiently the data, building the map on another platform decreases the interest of mapping with the a-puck.

ROS provides the e-puck those 2 missing functionalities with rosbag and *GMapping*. The rosbag tool is a data recorder and player that uses a specific file format called a bag to store data. It can be used to record all messages published on some chosen topics along with their time-step. The bag is then a very useful experimental tool that can be used to reproduce quickly a run or analyze the data.
The *GMapping* package is a ROS node that provides map building tools. It can be used directly on the e-puck to create the map while the robot is exploring. In order to produce a map, *GMapping* requires the following data :

- laser scans
- odometry
- *tf* transforms

A *tf* transform basically defines the translations and rotations required to reach a particular position from the center of the robot. Those transformations are then needed for the odometry but also to specify the position of each sensor.

With all those data, *GMapping* will publish the resulting map on the /map topic and we can then visualize it with a ROS visualization tool such as *rviz*. A more detailed explanation of the mapping process can be found in Appendix 5.

Hence, using ROS tools and simple sensors readings, a small robot like an e-puck has everything necessary to perform mapping. However, to perform experiments, an exploration behaviour should be designed to define the e-puck movements..

5.2 Exploration behaviours

In order to map its environment, a robot needs to explore it efficiently. However, as e-pucks are small simple robots, complex exploration behaviours should be avoided. Among many exploration behaviours, two approaches were selected for their simplicity : random walks and ballistic motion.

Random walks are exploration algorithms in which most of the directions to take are picked at random. A huge number of variants exist some strategies are often studied in robotics :

- Brownian motion
- Correlated random walk
- Lévy walk

- Lévy taxis

Brownian motion is described as the random motion of particles suspended in a fluid resulting only from their collision with the particles of the fluid[32]. Contrary to the other three strategies, Brownian motions are uncorrelated random walks and have therefore no particular tendency for exploration, making the movement truly unpredictable. The Brownian motion is actually a special case of the other strategies.

Correlated random walk[17] is a specific random motion where the next direction to take is biased towards the previous one. Hence, a correlated random walk tends to exploit an area rather than quickly exploring the largest area possible. This can be interesting for mapping as such behaviour could improve the precision of obstacles mapping by wandering around them for a long time.

Lévy walk[111] is a fractal-like random motion mixing long trajectories and short random moves. It is then especially useful for exploration as it promotes a quick exploration of the environment rather than the exploitation of a small area.

Finally, Lévy taxis[77] is a mixture of correlated random walk and Lévy walk. It is hence the more balanced category, combining exploration and exploitation behaviours. While a more efficient adaptive variant of Lévy taxis exists, the non adaptive variant was selected for the experiments for its simplicity and to allow a fair comparison with the other algorithms that are non-adaptive. Moreover, using the adaptive version would also increase the complexity of the e-puck controller and keeping it as simple as possible is an important choice of this work.

It should also be mentioned that all those random walks can be biased to improved their behaviour. However, introducing a bias in mapping does not have much sense as no part of the arena is really more important than the other, the robot should map the entire space.

Mathematically, all four strategies share the same governing laws and can be described by 2 equations describing the calculation of the step length (equation 5.1) and turning angle (equation 5.2) :

$$S_l = L_{min}.r^{\frac{1}{1-\mu}} \tag{5.1}$$

$$T_a = \left[2.\arctan\left(\frac{1-\gamma}{1+\gamma}.\tan(\pi.(r-0.5))\right)\right] + \text{bias} \tag{5.2}$$

The step length defines the number of time-steps before the next change of direction. In equation 5.1, L_{min} defines the minimum step length and this parameter is analyzed in Section 6.4.5. The turning angle defines the new direction to take. The bias in equation 5.2 is considered equal to zero in this work. Hence, at each change of direction, a random number r is picked and a new step length and a new turning angle are calculated using the equations.
The distinction between the four random walks is made by modifying the μ and γ parameters. Table 5.1 resumes this and present the different values possible for the parameters.

Optimal values for those parameters have been found by Pasternak et al.[77] : $\gamma = 0.05$ for the correlated random walk and $\mu = 2.8$ for the Lévy walk. As a mixture of those strategies, the Lévy taxis finds its optimal behaviour with $\gamma = 0.05$ and $\mu = 2.8$.

Name	**Step length**	**Turning angle**
Brownian Motion	Asymptotically Gaussian-like ($\mu = 3$)	Uniform ($\gamma = 0$)
Correlated Random Walk	Asymptotically Gaussian-like ($\mu = 3$)	Wrapped Cauchy ($0 \leq \gamma \leq 1$)
Lévy Walk	Power Law ($1 < \mu \leq 3$)	Uniform ($\gamma = 0$)
Lévy Taxis	Power Law ($1 < \mu \leq 3$)	Wrapped Cauchy ($0 \leq \gamma \leq 1$)

Table 5.1 – Governing laws and key parameters of random walks[27]

The other approach selected for the experiments is the ballistic motion. It is simply a straight line motion, without any change of direction. It should be noticed that this strategy is deterministic. This strategy is obviously useless as it is because the e-puck would be blocked by the first wall encountered. It is then necessary to introduce another feature in the controller : obstacle avoidance.

5.2.1 Obstacle avoidance

Obstacle avoidance is a feature provided to a robot to allow it to avoid bumping into obstacles and other robots. Depending on its implementation and its sensitivity, obstacle avoidance can be used to prevent a robot to be blocked against a wall or to follow the contour of an obstacle. In the context of mapping, obstacle avoidance might appear as counterproductive as the robot should actually approach the obstacles to map them correctly instead of running away from them. However, a robot blocked against an obstacle can not map at all and might also fail its odometry evaluation. A compromise should then be found between approaching enough to the obstacles and not being blocked by them. Soft turns and rotating in place are then the best options.

In order to keep the global complexity of the controller as low as possible, 3 simple implementations were selected :

- Without obstacle avoidance
- Force field (soft turns)
- Random rotations (in place rotation)

The first implementation is trivially the absence of any obstacle avoidance behaviour and will be only studied with the random walks as the ballistic motion can not work without it.

The force field algorithm is a simple deterministic vector based technique to avoid obstacles. When something is detected, a vector pointing perpendicularly in the opposition direction of the obstacle is calculated for each object detected. Those vectors are summed and the result is added to the vector describing the current direction taken by the robot. Hence, without stopping the current behaviour of the robot, this algorithm modifies its trajectory to go away from obstacles and prevent collision. This implementation will be tested with random walks as well as with the ballistic motion.

Finally, the random rotations algorithm is an adaptation to this work of the obstacle avoidance algorithm used in AutoMoDe[3]. When obstacles are detected, the robot will stop its current

behaviour and start turning on itself. The robot will turn for a certain amount of time-steps that is picked at random between 1 and a maximum number set as a parameter. The influence of this parameter is studied in Section 6.4.6. Considering how this algorithm work, it was quickly observed that it worked poorly with the random walks as too much time-steps were wasted. The random rotations will hence be tested only with the ballistic motion. Moreover, a deterministic version of this algorithm by setting the maximum number of rotation steps to 1, called fixed rotation, will also be tested, again with the ballistic motion.

5.3 Controller design

During experiments, the e-puck follows a simple routine : processing its sensors readings, sending data to ROS topics for mapping and defining the speed of its wheels taking into account surrounding obstacles. The e-puck controller for mapping experiments has then been divided into 3 parts : the mapping process, the exploration and the obstacle avoidance. As those 3 parts are totally independent, exploration behaviours can be easily interchanged and the mapping process could even be replaced by any other process to study other behaviours.

The mapping process consists basically in processing the readings of the e-puck sensors to produce the necessary data, scans and odometry, and publish them as ROS messages on appropriate topics.

The exploration part defines the strategy that will set the robot velocity and direction. This part has been designed with parameters allowing to change easily the exploration behaviour to use for the experiment.

The obstacle avoidance part follows the same model and through parameters defines the algorithm to use to avoid obstacles.

It should be noticed that the GMapping node is launched on the robot but outside of the controller. The main reason is to be able to easily modify the parameters of the GMapping node to adjust the map creation process but this also avoids slowing down the controller. For the same reason, the eventual bags are recorded outside of the controller. This is however a choice and those ROS functionalities could be started inside the controller.

5.4 Strengths and limitations

Of course, a simple robot such as the e-pucks can not perform as well as a more complex robot. Its battery discharges quickly and is accompanied with a loss of performances that degrades the quality of the map. The range and moreover the precision of its sensors is limited, reducing the quality of the obstacles on the map. Its computing power is also limited and increasing the complexity of the calculations done during the mapping process could lead to a loss of performances.

Additionally, mapping in robotics is often studied in the context of Simultaneous Localization And Mapping (SLAM) in which a robot produces a map and is also able to locate itself on it. In this work, only the mapping part was considered and achieving localization with an e-puck seems difficult regarding the complexity of the task and the capabilities of the robot.

However, the simplicity of all components of this mapping experiment provides multiple benefits. First, it is a strong base to build more complex behaviours and experimental setups. Is is indeed possible to optimize each part of the mapping process described here to improve the quality of the final map and hence have a functioning mapping robot at a relatively low cost. Such utilization could be useful for performing mapping experiments presenting risks for the robot that would be too risky with a more expensive robot.

It is also possible to go beyond the technical limitations of the e-puck. A single e-puck is not very efficient but many of them working together are. The second benefit of the simplicity is then to allow multi-robot or even swarm mapping. Indeed, multiple e-pucks released together in the same area could map it faster and better than a single e-puck. While it is still difficult to say if such multi-mapping process is more efficient than with a single complex robot, one can easily think of applications where many small robots would be necessary. A simple example is a disaster area. In such environment, mapping would be difficult or impossible for a single robot, especially if it is large but not for a swarm of small robots mapping together.
Simplicity is here a key factor, especially for a swarm-oriented application. Indeed, individual complex behaviours often perform poorly when expended to a swarm and do not cross well the reality gap. The complex behaviour should rather appear at a swarm level, with simple individual behaviours.

In this work however, multi-robot experiments are considered rather than swarm experiments because the inter-robots communication, essential to a swarm, is not addressed.

It is also important to precise that ROS provides also a tool to merge maps created by different robot into a unique complete map through the *multirobot_map_merge* package. Multi-robot (and swarm) mapping with e-pucks running ROS is hence possible. That all robots build the entire map or that only one of them does it is then a matter of choice.

Chapter 6

Simulation Experiments

In order to evaluate the mapping efficiency of the exploration behaviours described in Section 5.2, I first conducted experiments with simulated e-pucks. These experiments aimed to provide information about the influence of different robot characteristics and environmental factors on the performances of the robot. In this sense, it was also possible to determine the strengths and weaknesses of the exploration behaviours when they face different scenarios.

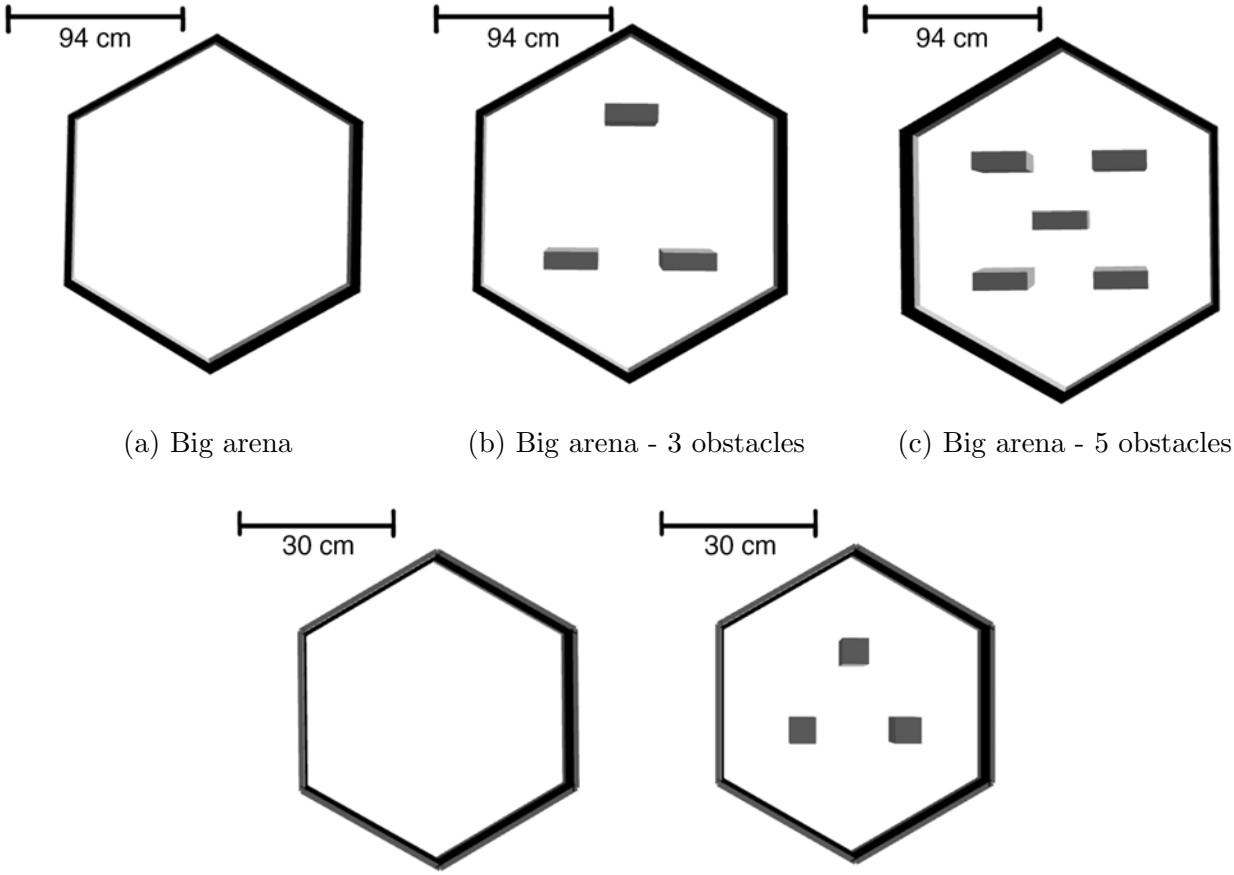
By conducting experiments in simulated environments I could test the controllers without exposing the robots to any risk. The simulator provided reproducible conditions for comparing the exploration behaviours; and it also allowed me to run faster a large number of experiments. Besides, the experiments on simulation were the basis from which I conceived and analyzed the experiments performed in reality.

The rest of this Chapter is structured as follows : Section 6.1 presents the simulation environment and the different scenarios conceived for the experiments; Section 6.2 describes the experiments and their main considerations; Section 6.3 details the metrics used to analyze the results; and Section 6.4 discusses the results and draws preliminary conclusions.

6.1 Environment

The simulator used in the experiments is ARGoS, a multi-physics simulator designed for swarm robotics. ARGoS already provides a model for the e-puck robot through a dedicated plug-in, model that has been assessed within a large number of research projects[29][34][3]. In addition, ARGoS is particularly interesting for conducting simulated experiments thanks to the easy parametrization of the simulator. Each experiment is described by a specific XML file (.argos) that holds all the specific parameters of both the controller and the scenario. Besides, ARGoS controllers can be easily adapted to include ROS functionalities since both are developed with C++. ARGoS is more detailed in AppendixB.

In the experiments, the robots execute the mapping tasks within closed spaces referred to as arenas. All the arenas are hexagonal spaces delimited by walls, but they differ in aspects like their size and number of enclosed obstacles. The first type of arena (A1) comprises a space of $22956.61cm^2$ with walls of $94cm$ in length. It has three configurations by including three, five or none rectangular obstacles of $200cm^2$. The second type of arena (A2) is a smaller version



(a) Big arena (b) Big arena - 3 obstacles (c) Big arena - 5 obstacles

(d) Small arena (e) Small arena - 3 obstacles.

Figure 6.1 – Arenas used in the experiments

of A1 with an area of $2338.27cm^2$ and walls of $30cm$ in length. Due to its reduced area, A2 only has the configurations that include three or none rectangular obstacles. Those are square obstacles and cover an area of $25cm^2$. Finally, both the walls and the obstacles are sufficiently high ($20cm$) to ensure their detection by the sensors of the robot. The set of arenas used in the experiment are shown in Figure 6.1

I conceived the experiments to assess only the ability of the e-puck for mapping closed spaces. Open space problems are very different and should be addressed with specific experiments that are not part of the thesis.

Experiments consider either one single e-puck or a group of ten e-pucks mapping the arenas. Though real e-pucks are equipped with many sensors and actuators that can be simulated by ARGoS, the robots modeled in these experiments are only equipped with proximity sensors and wheels; other sensors and actuators are not necessary for mapping and hence not relevant for the evaluation. In addition, the odometry of the robot is estimated by its own controller in open loop, as the e-puck has no functionality to determine whether the commands sent by the controller are properly executed. It is worth mentioning that the robot controllers have been designed to be the most simple possible. In the past, projects such as AutoMoDe[29] have indeed shown that simple controllers often cross more easily the reality gap, especially when this controllers are evaluated in robot swarms.

The present study do not completely consider what is expected from a robot swarm, but rather

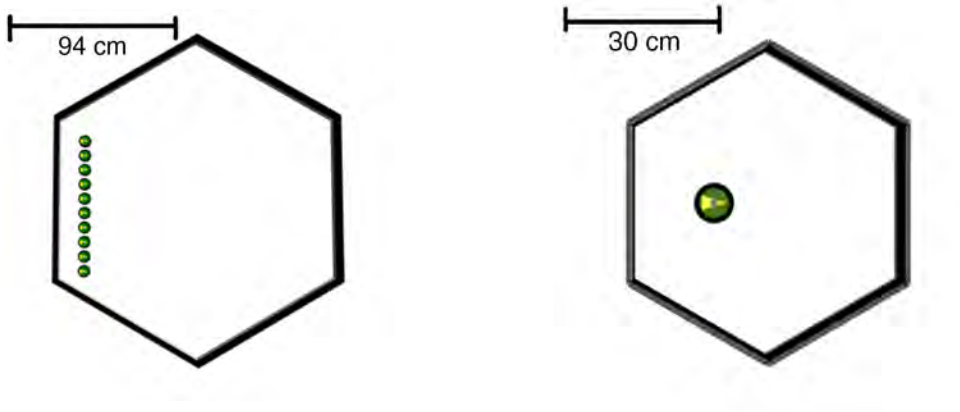
a more generalized multi-robot system. On one hand, the robots somehow share a common reference to an unique coordinate system defined by the arena. On the other hand, even though the mapping process occur separately for each robot, if multiple robots are used in the experiment the maps are centrally collected and merged at the end of the experiment. Both situations could be considered to some extent out of the desirable properties of robot swarms. However, it is still realistic to think of groups of robots that are deployed in a known position and from which the gathered information can be collected at the end of the task.

6.2 Experimental setup

The experiments consider two groups of exploration behaviours : random walks and ballistic motions. The first group comprises Brownian motion, correlated random walk, Lévy walk and Lévy taxis. Each of these behaviours are studied both without obstacle avoidance and with the force field obstacle avoidance algorithm. The second group consists only in the ballistic motion behaviour but studied in combination with three obstacle avoidance algorithms : fixed rotation, random rotation and force field obstacle avoidance. All those behaviours and obstacle avoidance algorithms are described in details in Section 5.2.

The research is organized in a set of experiments that each time aim to evaluate different characteristics of the exploration behaviours named above. In other words, each set of experiments study the influence on the mapping task of one parameter across all the exploration behaviours. Since a single different parameter is modified for each set, the other parameters are set to their default values. The following list presents the different studied parameters along with their default values.

- Precision of the odometry : the precision is defined by two constant values mapping the movement calculated within the controller to the actual movement in the simulator/in reality. The precision of the odometry hence depends directly on the precision of those values (default : high precision).
- Importance of obstacle avoidance : presence/absence of obstacle avoidance algorithm (default : with obstacle avoidance).
- Velocity of the robot : can be set to any reasonable values, that is between 0 and 20cm/s (default : 5cm/s).
- Minimum length of the random walk step : express the minimum number of time-steps before changing direction, can be set to any value between 1 and 100 (or any reasonably big number depending on experiment length) (default : 10 time-steps).
- Maximum number of rotation steps for the random rotations algorithm : can be set to any value between 1 and 25 (or any reasonably big number depending on experiment length) (default : 25 time-steps).
- Size of the arena (default : A2, see Figure 6.1).
- Number of robots (default : 1).
- Number of obstacles (default : 0).



(a) 10 e-pucks in the big arena

(b) 1 e-puck in the small arena

Figure 6.2 – E-pucks initial position

This thesis is mostly focused on assessing the capabilities of the e-puck for performing mapping tasks. Therefore, most experiments have hence be conceived with a single robot. Nevertheless, on top of those experiments, I conducted experiments to analyze the joint mapping capabilities of groups of 10 e-pucks. In the later, the experiments are performed in the arena A1 6.1, A2 being too small.

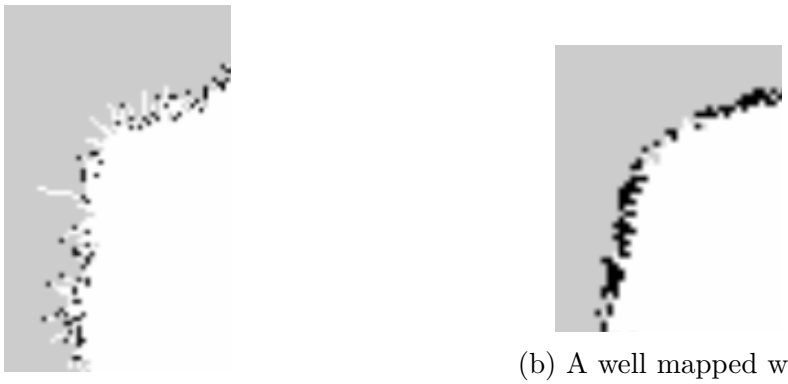
Finally, all experiments share some common setup :

- An experiment is composed of 30 runs with identical configuration. Nevertheless, the random seed of the simulator varies from 1 to 30 on each run.
- The experiments last for 3 minutes, hence 1800 time-steps in ARGoS.
- The starting position of the robots is always the same : they are aligned along the west wall of the arena with 10cm between them and the wall, and 9cm between each other as shown on Figure 6.2a. In the case of a single robot, it occupies the central position of such placement (Figure 6.2b). This is required as the mapping process needs to know the initial position of the robots, especially for merging the individual maps for the 10 e-pucks experiments.

6.3 Metrics

The correctness of a map can already be estimated by visually comparing it to a reference model. Such observations would allow to see directly if a map presents obvious mistakes but do not provide clear and precise comparison. In this thesis, rather simple but quantifiable metrics have been defined to evaluate the correctness of the maps produced by the e-puck. Hence, the quality of the map is described based on set of features that describe the obtained maps: mapped area (coverage), and number of walls and obstacles correctly mapped.

First, the coverage of the map is evaluated using the minimum bounding rectangle area[108]. This is an image processing technique that detects objects in an image and find the smallest rectangle that enclose the whole object, and then calculates its area. A measure of the coverage can be obtained by comparing those values to the reference map. Besides, since the arenas



(a) A poorly mapped wall

(b) A well mapped wall

Figure 6.3 – Different mapping precision

considered in the experiments are convex, the exceeding area covered by the rectangle will not false the results.

Then, the precision is evaluated by counting recognizable walls and obstacles in the map. It should be noticed that this choice was made for simplicity as it does not take into account subtle differences in the quality. Indeed, 2 walls could be perfectly recognizable but one of them could be more precisely mapped. Figure 6.3 illustrates this difference : the right wall is better mapped than the left one, yet both are recognizable as walls. In this work, as soon as a wall is recognizable and at a correct position, it is counted. In other words, both walls on Figure 6.3 would be counted. This distinction will however be discussed in the analysis.

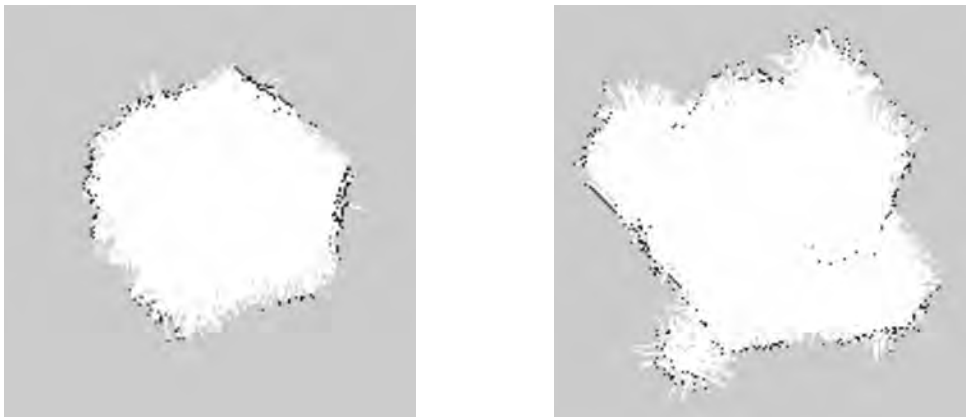
Evaluating this metric is indeed a complex task. Hough transform can be used to detect lines and hence, a priori, walls. However, maps are composed of many small segments and even with pre-processing of the image, the Hough transform often detects too many lines. For the same reason, counting obstacles is even more complicated as e-pucks tend to map better the walls than the obstacles as they spend more time near the borders. A manual counting, while not as precise as an automated technique was here the better and simpler solution and gave very plausible results.

Finally, it is important to precise that a correct mapping of the walls and the obstacles was considered to have more importance. To be useful, the map should show features rather than a big void area. Moreover, it could be stated that the desirable results would minimize the errors in all the metrics at the same maps. It could happen that maps with good coverage, and right number of walls and obstacles as the reference can still be completely wrong.

6.4 Results and analysis

As mentioned before, the experiments aimed to compare different exploration behaviours for mapping as well as the influence of their parameters. The following section present the different experiments and their results. First, the importance of each parameter and their respective impact on the general performances are discussed. Then, a global conclusion compares the results of the different exploration behaviours.

A display of the mapping process at 6 different time can be seen in Appendix F.



(a) Ballistic motion with random rotations and high odometry precision

(b) Ballistic motion with random rotations and low odometry precision

Figure 6.4 – Impact of odometry precision

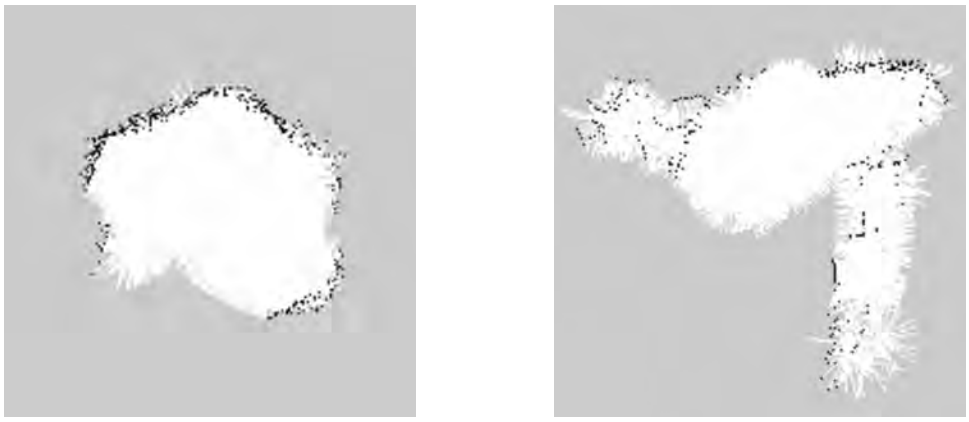
6.4.1 Precision of the odometry

The first set of experiments performed with the e-pucks clearly highlighted a crucial parameter while mapping scenarios : the precision of the odometry. Since the odometry is estimated in open loop by the controller of the e-pucks, a calibration process was required in order to approximate those estimations to the position and orientation simulated by ARGoS. With this purpose, I designed a calibration methodology by adding and tuning two parameters in the controller, the step calibration for the position and the angle calibration for the orientation. Both being two constants that adapt the estimations of the odometry to make it fit with the simulator.

I studied two sets of values while searching for correct values for the odometry parameters. The first one provided correct position and orientation up to the fourth decimal, the second one provided values only up to the first decimal.

The results showed a clear distinction regarding the precision used to define the parameters. While the first set of values leads to correct maps (with still different qualities depending on the exploration behaviour), the second set of values provided totally wrong maps with a lot of outgrowths and walls placed at a wrong location. Figure 6.4 shows an example of the same run with the first set (left) and with the second set (right). The second map is bigger from what is expected, and some walls are incorrectly placed.

While on each time step the error of the odometry estimation was still small in both cases, the open loop nature of the odometry grew the error by summing them up with each movement. Hence, the second set of values added more error over the time, and therefore it produced worst maps. In this case, the huge number of rotations and changes of direction performed by the robot during a run rapidly made the total error to become significant. In comparison, this problem does not clearly showed up with the first set of values, as the individual errors were too small considering the 3 minutes of the run. However, this level of precision might not be sufficient for longer experiments.



(a) Brownian motion with force field

(b) Brownian motion without obstacle avoidance

Figure 6.5 – Impact of obstacle avoidance on mapping

6.4.2 Importance of obstacle avoidance

This experiment focuses on the importance of including an obstacle avoidance behaviour in the mapping process. Results are not discussed for the ballistic motion behaviour as it would trivially fail without obstacle avoidance since the robot would try to go in straight line forever.

By experimenting I was able to determine that the absence of any obstacle avoidance algorithm leads to totally wrong maps. Indeed, the problem lies again on the estimation of the odometry in an open loop. If no obstacle avoidance is implemented, the robot can be stuck against walls and obstacles but the simulated odometry is not able to determine that the e-puck is not actually moving. Hence, the controller keeps updating erroneously the position and orientation of the robot. As result, the map will contain outgrowths and walls wrongly placed.

Figure 6.5 shows an example of the same run with (left) and without (right) obstacle avoidance.

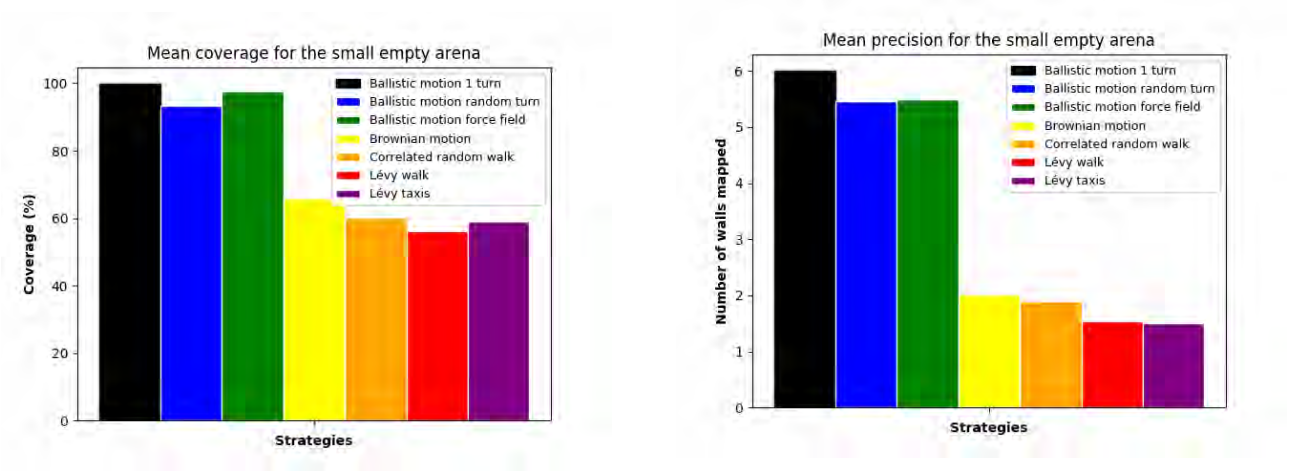
Since it was found that obstacle avoidance is then mandatory in order to produce correct maps. In the following, the random walk behaviours will hence always be considered along with the force field obstacle avoidance.

6.4.3 Reference experiment

This experiment aims to provide a comparison basis from which to analyze the variation of parameters in the other experimental setups. I studied the maps created by a single robot in the small empty arena (A2) with the default configuration parameters (6.2). Figure 6.6 displays results for the coverage and precision metrics. Figure 6.7 displays maps obtained in this reference experiment.

The results showed a different in performance regarding two groups : the ballistic motions and the random walks, the first being highly better than the second. Indeed, while ballistic motions cover the whole map (or very close to) and successfully map the walls, the random walks usually fail to cover more than half of the arena.

Besides, distinctions can be made inside the ballistic motions group. The fixed rotation and



(a) Coverage results

(b) Precision results

Figure 6.6 – Results of the reference experiment

(a) Ballistic motion with fixed rotation

(b) Ballistic motion with random rotations

(c) Brownian motion

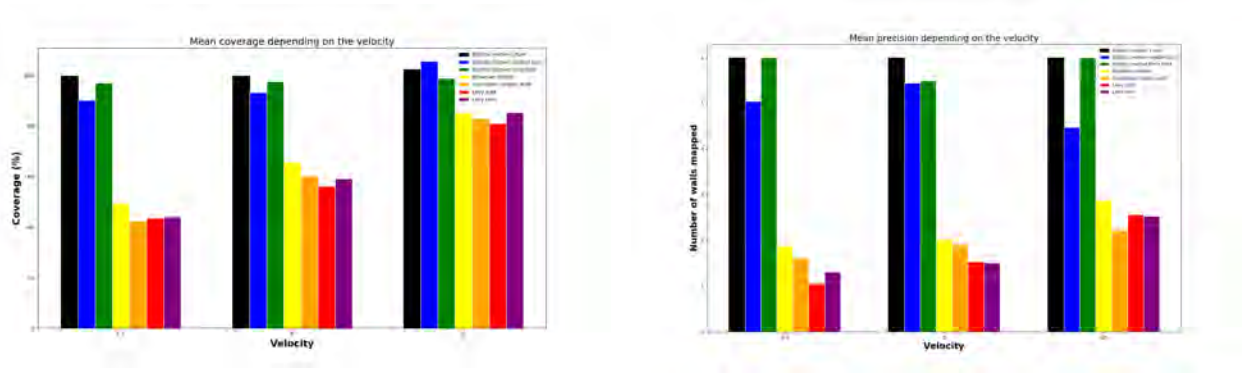
Figure 6.7 – Maps of the reference experiment

force field variants, being both deterministic, provide the best maps from all strategies, with a slightly better result for the fixed rotation However, contrary to the random rotations variant, they do not cover the center of the arena. For this experiment, avoiding to map the center did not have impact on the metrics since the arena was empty, however it might become troublesome when obstacles are added.

Regarding the random walks group, it was difficult to point differences between the four behaviours as they showed similar results, both for the coverage and the precision. The Brownian motion and the correlated random walk seem still slightly better but nothing that could lead to conclusive statements. Actually, the random walks suffer here from their randomness that prevent them most of the time to cover the entire arena before the end of the experiment.

6.4.4 Velocity of the robots

From preliminary experiences I found the velocity of the robots clearly affecting the quality of the maps. Indeed, a lower speed will allow the robot to spend more time near walls and



(a) Coverage results

(b) Precision results

Figure 6.8 – Results of the velocity experiments

obstacles and hence to map them more precisely. On the other side, a higher speed will allow it to cover more distance and hence more area. Hence this parameter was evaluated to identify the values that fit more the experimental setup proposed before. For this, I conducted experiments with two values of velocity of 2.5 and 10cm/s. Figure 6.8 displays the results for the coverage and precision metrics.

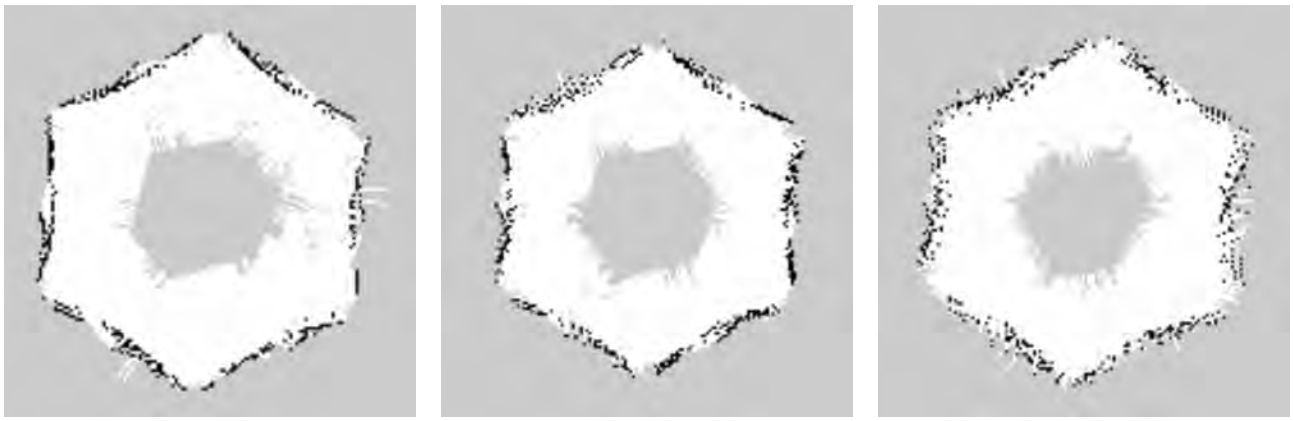
The results exposed the coverage/precision trade-off in an interesting way and highlighted the limitations of the metrics. The analysis showed up that there exists a distinction between the random walks and the ballistic motions. Similarly to the reference experiment, the second outperforms the first.

In the case of the ballistic motions, it could be observed that the coverage is barely influenced by the change of speed : the robot still has enough time to cover the entire surface. However, the precision of the walls differs for the version using the random rotations obstacle avoidance : when the velocity increases, the precision tends to decrease and less walls are correctly mapped. Then, the number of walls correctly mapped is also smaller for the low velocity experiment than for the default reference experiment. Indeed, even if the coverage remains approximately the same, with a lower speed the robot has not always the time to map all the walls.

Nevertheless, when looking at the maps, the precision of the walls is slightly better with a lower speed. It should be clarified that this specific criterion was not easy to measure and was then not represented by one of the metrics. It should also be noticed that as long as the robot have enough time to complete a full round trip of the map, results do not change much for the fixed rotation and the force field versions of the ballistic motion as those variants are fully deterministic. Figure 6.9 illustrates the difference of quality on the same run with different velocities. The difference is subtle but walls are more dense with low speed and more sparse with high speed.

Concerning the random walks, I found a clear benefit of implementing controllers with a higher velocity as it increased considerably the coverage of the environment. This was actually the most noticeable impact of the speed for the random walks. Indeed, the walls count results are here highly correlated to the coverage results. With a high speed, the robot covered a more area and hence there were more chances to map the walls, even in detriment of the precision. On the other hand, with a low speed, the e-puck usually covered less than half of the entire area, decreasing the chances to map walls.

Results of this experiment highlighted two important points :

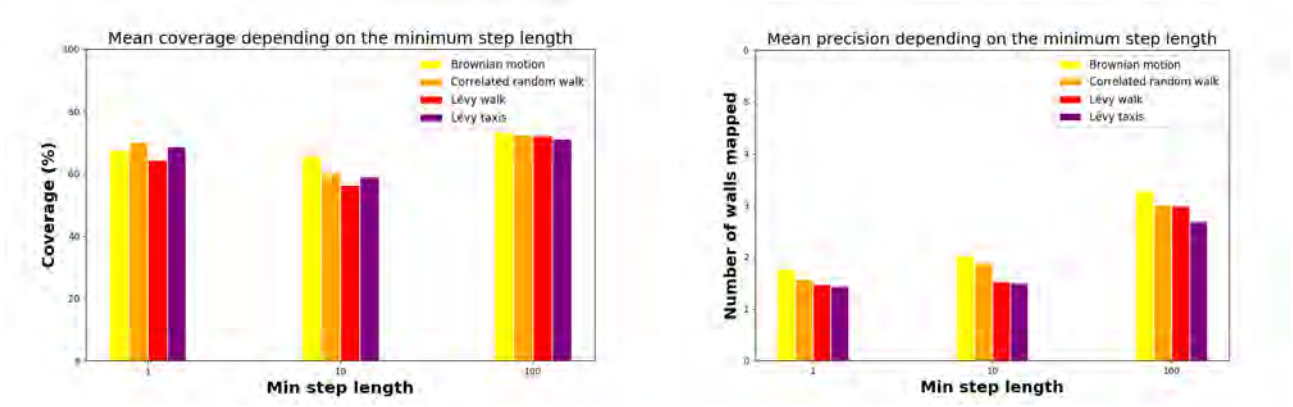


(a) Ballistic motion with fixed rotation and velocity of 2.5

(b) Ballistic motion with fixed rotation and velocity of 5

(c) Ballistic motion with fixed rotation and velocity of 10

Figure 6.9 – Difference of quality depending of the velocity



(a) Coverage results

(b) Precision results

Figure 6.10 – Results of the minimum step length experiments

- The simple metrics used to evaluate the maps are not fitted for measuring subtle improvements of the map quality such as the precision of the walls.
- The velocity is an important parameter that needs to be balanced to cover enough area but with a reasonable speed to map efficiently the surroundings. While other values might be tested, the intermediary value of 5 seems to produce good results and will be kept for the following experiments.

6.4.5 Minimum length of the random walk step

This parameter concerns only random walks as it defines the minimum number of steps (and indirectly also the maximum number) before the robot picks a new direction at random. Its impact on the walk is however barely discussed in the papers treating of random walks. At best, it is explained that a low value will induce more changes of direction while a high value will lead to long straight lines between each change of direction.

I conducted experiments with a low value (1) and a high value (100) of this parameter. Afterwards, I compared the results with those obtained with the default value (10). Figure 6.10 displays the results for the coverage and precision metrics.

(a) Lévy taxis with a minimum step length of 1

(b) Lévy taxis with a minimum step length of 10

(c) Lévy taxis with a minimum step length of 100

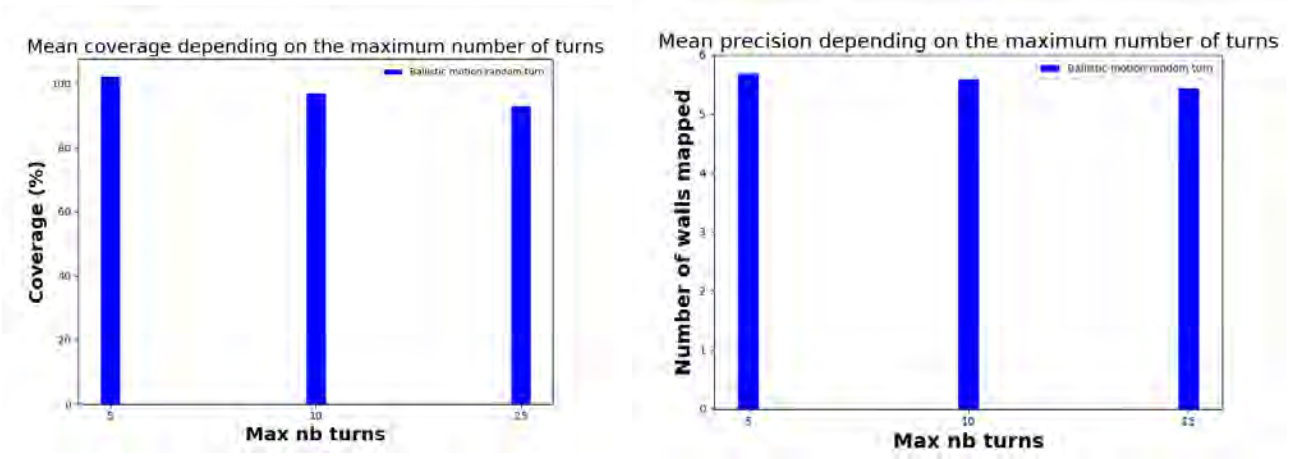
Figure 6.11 – Difference of quality depending of the minimum step length

The results showed interesting properties while this parameter is varied. With a high value, as expected, the robot tend to cover a larger area. However, unlike results with a high speed, the relatively slow speed helped the robot to map the walls correctly and present hence a better precision. Yet, this precision is slightly lower than the one obtained with the default value. Similarly to the high velocity experiment, the increase on performance of the walls count is mainly due to a coverage improvement.

On the other hand, the experiment with a low value showed results that are far from what could be expected : the coverage was indeed slightly increased, except for the Brownian motion. It seems that changing direction more often benefits the random walk behaviours. An explanation could be that if the robot takes an uninteresting direction, it keeps it for a shorter time. While this is also true for interesting directions, the short length of the runs makes the result very sensitive to a long periods of poor mapping. It is also worth discussing that the coverage improvement does not affect the Brownian motion, which is the only of the four behaviours that does not have any strong tendency on its walk. This could be interpreted as the four behaviours actually behave differently although it was not clearly observable in the experiment. Finally, a low step length value has however a negative impact on the precision of the mapping process as less walls are correctly mapped, even with the coverage improvement.

Figure 6.11 shows an example illustrating the subtle difference in the density of the walls points when changing the minimum step length.

To conclude, the minimum step length is an interesting parameter that deserve further studies. Its effects on the mapping process are not as significant as the velocity in terms of coverage but it has a greater impact in terms of precision. The default value will be kept for the following experiments as it leads to better results for the precision metric, which is preferred than coverage metric. Also, the apparently good results of the experiment with the high value should be nuanced as the graphs showed an average over the 30 runs. Some run were considerably bad as the robot in some cases was stuck in the same area for nearly the entire run, only driven in some cases by the obstacle avoidance.



(a) Coverage results

(b) Precision results

Figure 6.12 – Results of the maximum number of rotation steps experiments

6.4.6 Maximum number of rotation steps

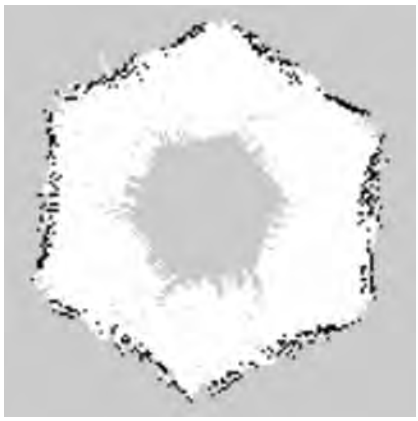
This parameter is only relevant for one of the exploration behaviours : the ballistic motion with random rotations obstacle avoidance. The parameter defines the maximum number of steps during which the robot will turn on itself when detecting an obstacle. So far, only two values have been tested for the parameter : the default value of 25, and the value of 1 that corresponds to the deterministic fixed rotation variant studied before. In all previous experiments, the fixed rotation variant was always better than the random rotations since it leaded the robot to map the entire contour of the arena, which was not always the case for the random rotations. However, the fixed rotation variant never mapped the center of the arena. Then, one could think that by tuning the number of rotation steps, one could help the random rotations based behaviour to map more efficiently the walls and obstacles at the same time. I conducted experiments with a maximum number of rotation steps of 5 and 10. Figure 6.12 displays the results for coverage and precision metrics.

The results, while showed a small improvement in the precision, do not present highly accurate maps for each run. Indeed, an unexplored area often appears in the center of the arena similarly to the fixed rotation variant. This unexplored area is usually smaller with 10 steps than with 5 steps. Figure 6.13 illustrates this phenomenon for the same run.

There exists hence a trade-off between a good contour mapping and a full coverage of the arena. It could be though that an optimal value exists between 10 and 25 steps. It should however been noticed that this optimal value would only make sense for this particular arena configuration, i.e. without obstacles. The default value of 25 steps will hence be kept for the rest of the experiments.

6.4.7 Size of the arena

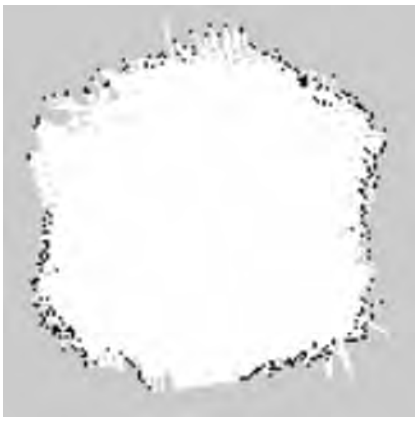
In this experiment I let one single robot to map the big empty arena (A1). The results, as expected, showed a poor performance on mapping since the arena is too big for one robot and such short time. However, it was interesting to find that the ballistic motions outperformed again the random walks in terms of coverage and precision. The maps obtained with the



(a) Ballistic motion with fixed rotation (1 step)

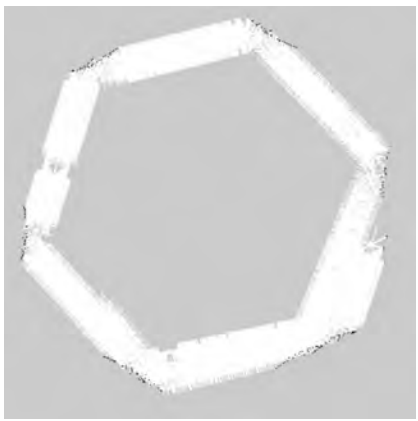
(b) Ballistic motion with random rotations (maximum 5 steps)

(c) Ballistic motion with random rotations (maximum 10 steps)

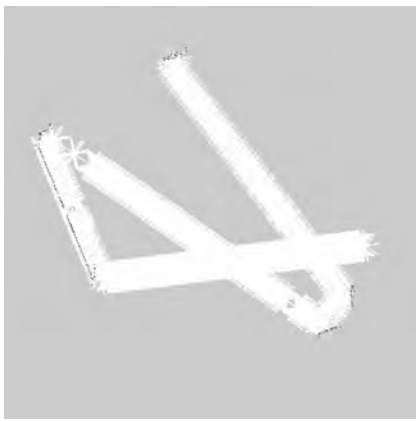


(d) Ballistic motion with random rotations (maximum 25 steps)

Figure 6.13 – Difference of quality depending of the minimum step length



(a) Ballistic motion with fixed rotation



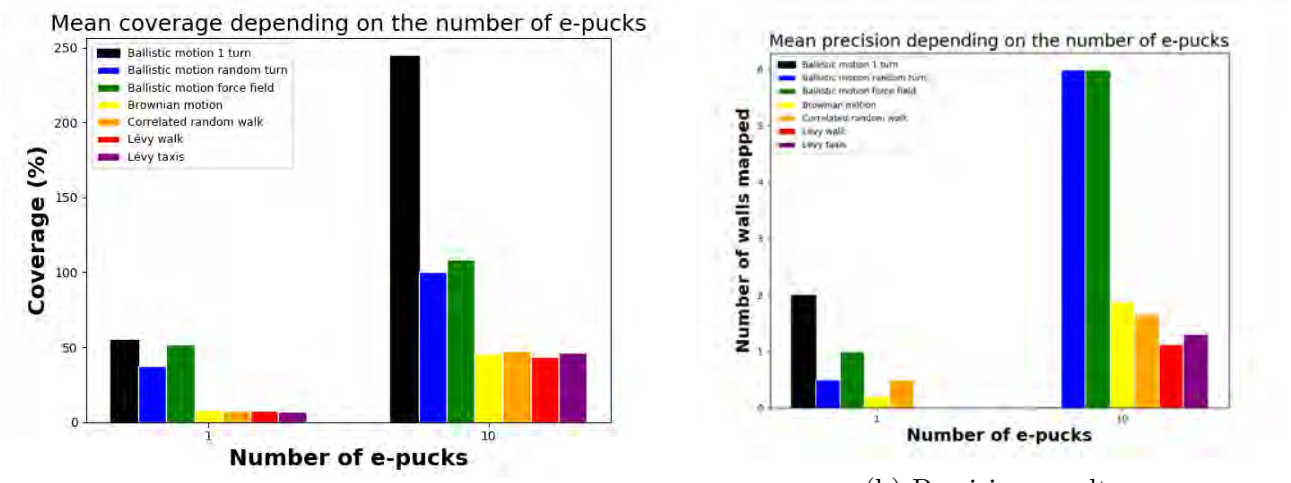
(b) Ballistic motion with random rotations

(c) Brownian motion

Figure 6.14 – Maps obtained in the arena size experiment

ballistic motion with fixed rotation obstacle avoidance were even relatively good compared to the others. Nevertheless, none of those maps are truly useful as in the best cases they still did not show many walls. Figure 6.14 displays maps obtained in the experiment.

This experiment illustrates well the limitations of attempting to map relatively big areas with a



(a) Coverage results

(b) Precision results

Figure 6.15 – Results of multi-robot experiments

single robot. Most of literature on mapping refer to experiments where the sensors of the robot are able to map in a range of several times the size of the robot. Experiments of mapping with short range sensors are not often described, and therefore their limitations are not discussed either.

6.4.8 Number of robots

In this experiment, I conducted a comparative study between the results obtained by mapping the big empty arena (A1) with a single robot, with results obtained in the same arena by mapping with a group of 10 robots. It should be noticed that only the mapping efficiency was evaluated in these experiments, neither cooperative strategies nor distributed mapping were considered.

Quantitatively evaluating the precision of the composed maps was harder than in previous experiments. In order to keep the same metrics and fairly compare the previous experiments, maps that presented too many or too important errors were counted as a complete failure. Hence they counted as 0 for the metrics. This is the reason for which the score obtained by the ballistic motion with fixed rotation is at the minimum, since all the maps produced during the experiment were considerably wrong. Figure 6.15 displays the results for the coverage and precision metrics.

Without surprise, the 10 robots perform better that the lone robot. This could be easily noticed by comparing the results with those obtained in the previous experiment. Moreover, the ballistic motions perform better than the random walks with results very similar to those obtained for the reference experiment (Section 6.4.3). However, with the inclusion of multiple robots in the mapping process, new problems appeared as well.

The first and most important problem was the interference among robots. Those interference appeared in 2 different ways : robots blocking each other and artifacts on the map. Robots blocking each other is a phenomenon that can be easily seen in the results obtained for the ballistic motion with fixed rotation obstacle avoidance. Contrary to all other behaviours, this one completely failed to build a correct map because most of the robots were blocked together

against a wall. The determinism of the strategy prevented the robots to free themselves, only 1 of them was able to escape and map the contour of the arena. The deterministic behaviour is then not viable with multiple robots. This could be also observed in the other deterministic behaviour, the ballistic motion with force field obstacle avoidance. While the errors were not as problematic as with the fixed rotation obstacle avoidance, small precision errors could be seen around the maps. Similar errors were also occasionally present in the maps created by other behaviours, but this was very rare.

The artifacts are small black dots and trails appearing in the center of the map where nothing should be found. Those artifacts came from e-pucks mapping each other as obstacles. Indeed, they could not differentiate a wall and another robot and hence they mapped them. However, most of those artifacts were corrected during the mapping process by robots passing over the same spot multiple times. Some artifacts remained though but were easily identifiable and did not lead to confusions in the map. One could think that by introducing communication among robots, the impact of this phenomenon could be reduce.

The second problem consisted in map superposition mistakes. As multiple robots created their own map during a run, a map merging tool must be used to fuse the individual maps into a single complete one. However, some superposition errors could be seen on many maps. It is unclear if those errors came from a lack of precision in the merging tool, the heterogeneity of the individual maps created by the e-pucks, or due to positioning errors in the odometry estimation. However, as the simulated robots were all the same and shared an already calibrated odometry, it is more likely that the merging tool was faulty and leaded to the errors.

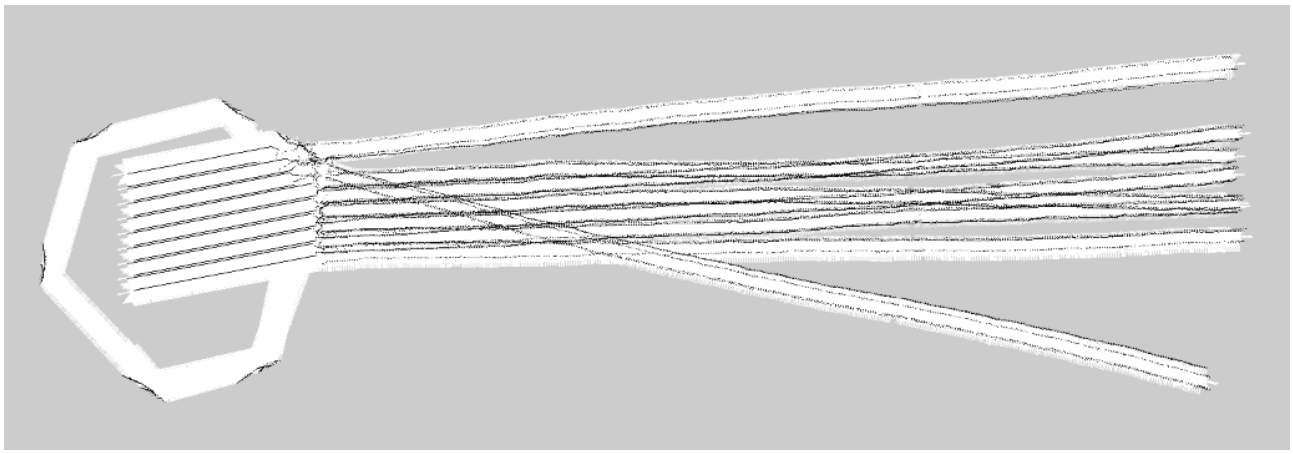
In overall, the best results were then obtained by the ballistic motion with random rotations that produced very good and understandable maps. This experiment illustrated then well the importance of randomness in the mapping process, especially when robots are not able to differentiate the nature of the sensed obstacles.

Figure 6.16 illustrates the different errors that may appear.

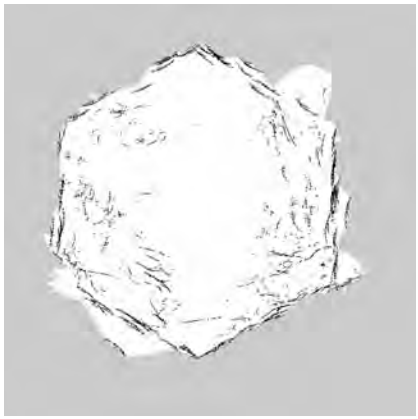
6.4.9 Number of obstacles

The final set of experiments was aimed to study the robustness of the different behaviours when adding obstacles in the arena. Three experiments were done with this purpose : one robot mapping the small arena (A2) in the presence of three obstacles, and ten robots mapping the big arena (A1) in the presence of three and five obstacles. Figure 6.17 displays the results for the coverage and precision metrics.

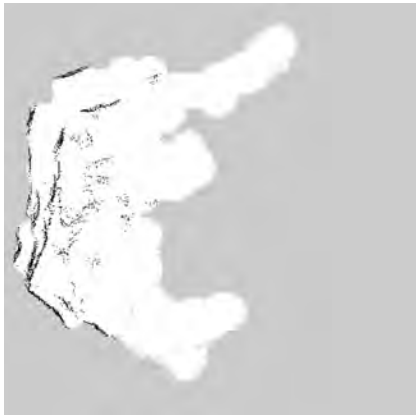
The results were coherent with the previous observations : in all cases, the ballistic motion with random rotations strategy gave the best results, with a full coverage, a correct mapping of obstacles and walls and very few errors. The other (deterministic) ballistic motions failed to reach the same results for the reasons already pointed out in the previous experiments. In the small arena (A1) with one robot, they did not allow the e-puck to map the center of the arena and hence the obstacles. In the big arena (A1), the interference between robots prevented them to achieve a correct map, especially for the fixed rotation obstacle avoidance. Concerning the random walks, it should be noticed that despite they did not reach the performances of the ballistic motion, their results were coherent with those obtained in an empty arena. This means that despite their low performances, they showed robustness and could adapt to the presence of obstacles where deterministic strategies cannot.



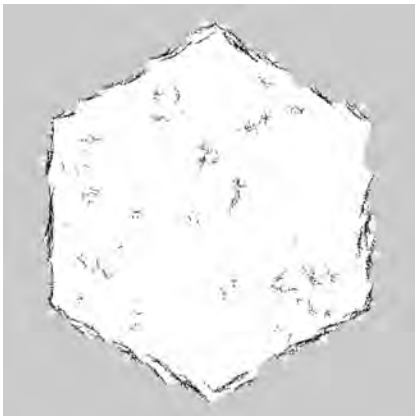
(a) Robots blocking each other (ballistic motion with fixed rotation)



(b) Robots blocking each other occasionally (ballistic motion with force field)



(c) Superposition error (Brownian motion)



(d) Artifacts only (ballistic motion with random rotations)

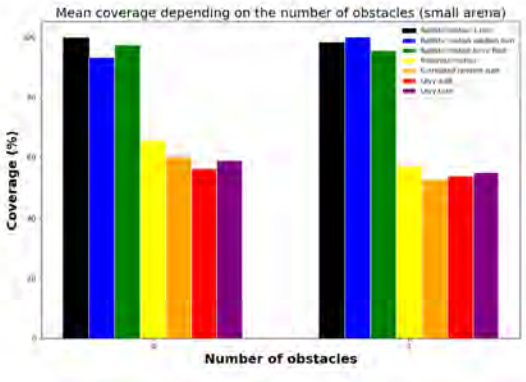
Figure 6.16 – Errors encountered in multi-robot experiments

A last observation is that obstacles in the small arena seemed harder to map as they were smaller. Indeed, the e-puck often stayed few times near them and was not able to get enough readings to position the object in the map.

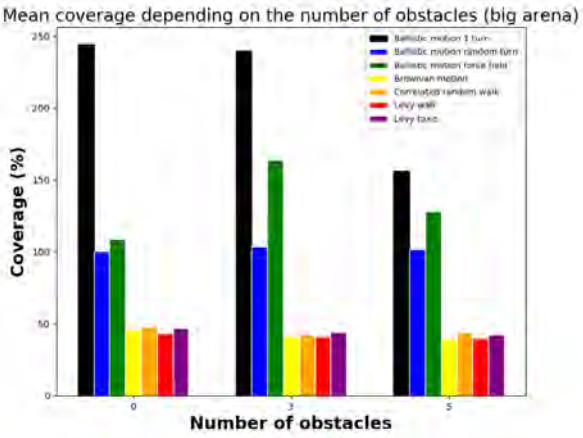
Figure 6.18 presents the best results for each experiment, all obtained with the ballistic motion with random rotations.

6.4.10 Conclusion

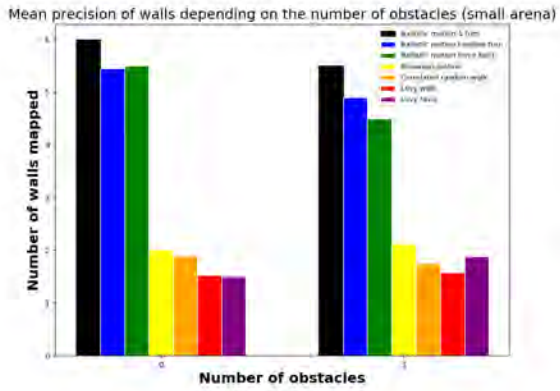
A first important conclusion is that random walk behaviours are not very efficient for mapping tasks. Random walks are rather interesting for general exploration purposes such as in target retrieval situations. In the case of mapping, where every wall and object could be seen as a target, they tend to be slower, and therefore, less efficient than simpler techniques such as the ballistic motion. The random walk behaviours are actually not bad for mapping as they do not create erroneous maps and adapt well to different configurations, but they are too slow compared to ballistic motions. In the literature, the random walks are often biased when used to retrieve a target to improve their efficiency, this is hard to accomplish while mapping unknown scenarios as no part of the arena is really more interesting than another.



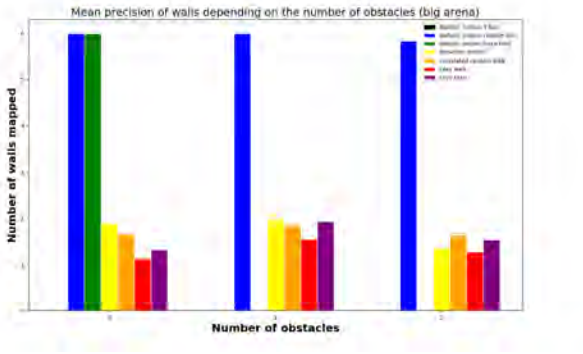
(a) Coverage results for the small arena



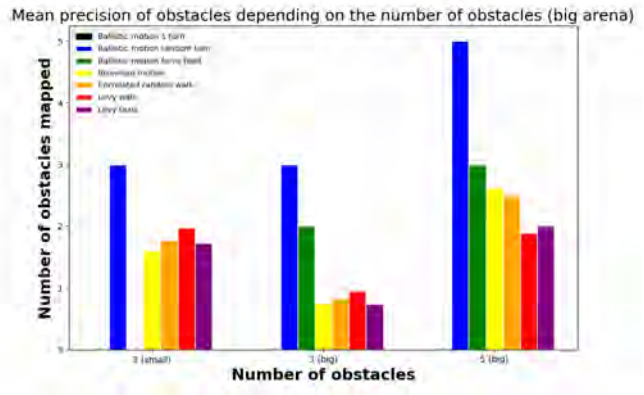
(b) Coverage results for the big arena



(c) Precision of walls results for the small arena



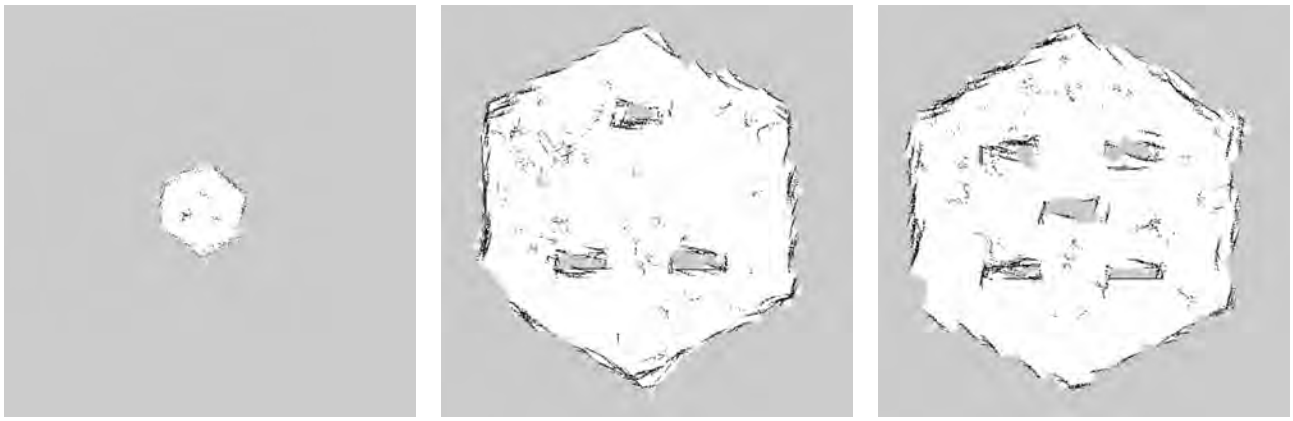
(d) Precision of walls results for the big arena



(e) Precision of obstacles results

Figure 6.17 – Results of the obstacles experiments

Many parameters have also been studied through those experiments. A good precision of the odometry estimation and the presence of an obstacle avoidance behaviour clearly proved to be the most important factors, and to some extent, they are mandatory to achieve good mapping. Then it is also clear that multiple robots mapping together are more efficient to cover quickly a big surface than a lone robot. Moreover, multi-robot experiments have demonstrated that



(a) Ballistic motion with random rotations, small arena with 3 obstacles

(b) Ballistic motion with random rotations, big arena with 3 obstacles

(c) Ballistic motion with random rotations, big arena with 5 obstacles

Figure 6.18 – Some maps of the arena size experiment

deterministic behaviours, while efficient in some particular configurations, are not robust and adapt poorly to the presence of other robots and obstacles.

Some other parameters such as the velocity or the step length have a less significant impact on the mapping process, especially on what I found to be the best behaviour, the ballistic motion with random rotations obstacle avoidance. Their impact is greater on the random walks where they propose some trade-off between coverage and precision.

All of those experiments finally tend to confirm that even for a task like mapping, simple solutions are often better than complex ones. This is however true in the context of closed space experiments. It is easy to convince oneself that the ballistic motion cannot work in an open space without a high risk of loosing the robot. In an open space scenario, random walks may be the more efficient exploration behaviour.

Chapter 7

Real robot validation

In order to validate the results obtained with the simulations, I conducted experiments by using the controllers on a real e-puck. The experiments were not exhaustive and mainly aimed to compare, at least for single robot experiments, the results obtained on simulation and reality. The comparison then served as a basis for analyzing the reality gap that has to be crossed. Multi-robot experiments are much more complex to setup and will be performed in a future work.

The rest of this Chapter is structured as follows : Section 7.1 presents the experimental environment and the different arenas used in the experiments; Section 7.2 describes the experiments and the important parameters studied; Section 7.3 details the metrics used to analyze the results; Section 7.4 discusses the results and draws some conclusions.

Figure 7.1 shows the real arena with five obstacles from the point of view of the e-puck (i.e. from the left side of the arena).

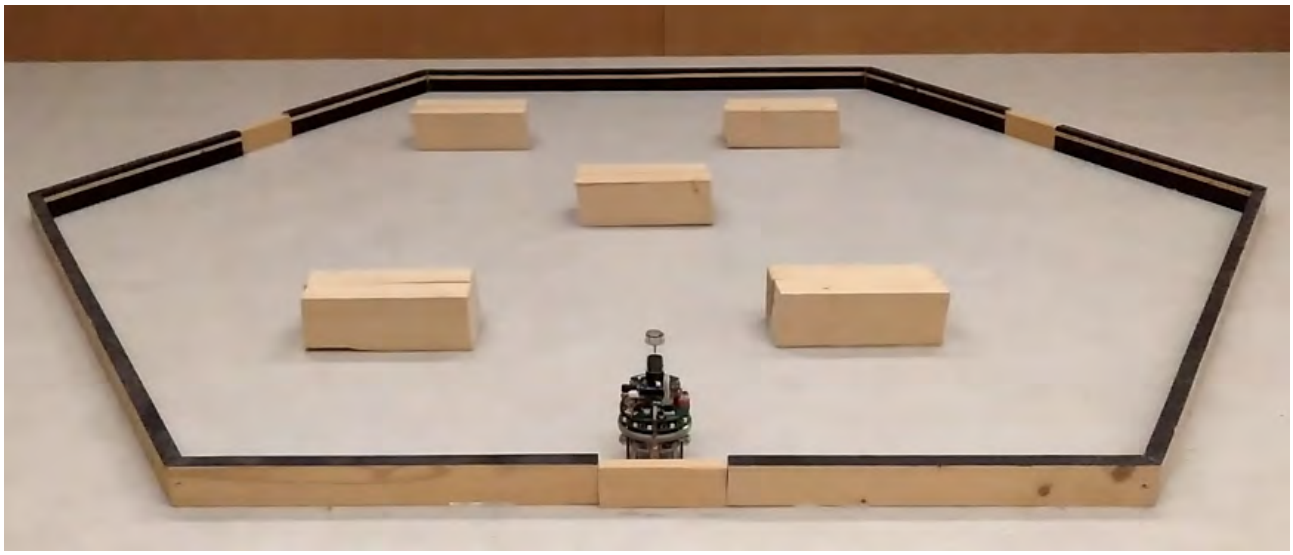


Figure 7.1 – Real arena with five obstacles and the e-puck (seen from left side)

7.1 Environment

The experiments took place in the experiment room at IRIDIA research laboratory. An arena with $94cm$ edges built in wood was placed on a uniform flat floor. Two different configurations have been tested : the arena without obstacles and the arena with five obstacles, also made of wood and each covering an area of $200cm^2$. The obstacles were placed at the same position as in the simulated version to allow a fair comparison.

The e-puck robot used for the experiments was running the custom Yocto distribution that I developed and it used the exact same controller that was used in the simulation experiments.

Finally, it should be noticed that the e-puck created itself the map using *GMapping* and recorded all data with *rosbag*.

7.2 Experimental setup

The same exploration behaviours as in the simulation experiments were evaluated. Brownian motion, correlated random walk, Lévy walk, Lévy taxis and ballistic motion. The four random walks were tested only with the force field obstacle avoidance as it was clearly demonstrated by the simulation that mapping cannot work without obstacle avoidance. The ballistic motion was tested with the same three obstacle avoidance behaviours: fixed rotation, random rotations and force field.

I conceived two sets of experiments for the real e-puck : one in the empty arena and one in the arena populated by five obstacles. The experiments consisted in five runs for each behaviour, each run lasting for three minutes. In the experiments, the seeds used by ARGoS to add randomness in the behaviour were five seeds already used for the simulation.

Parameters were set to the default values defined in Section 6.2. Hence, the real experiments should be conducted in most similar possible conditions as for the simulation to allow a fair comparison. The only difference was that the robot sensors were calibrated before the experiments to reach the best performances possible (simulated e-pucks have ideal performances).

7.3 Metrics

The metrics used to evaluate those experiments were similar to the metrics used for the simulation. The coverage was indeed measured in the exact same way, i.e. with the minimum bounding rectangle technique[108]. However, the precision was not measured by counting walls and obstacles but rather by comparing directly the maps between simulation and real robot. Indeed, results presented in Section 6.4.7 clearly showed that a single e-puck could not map the big arena (the same as the real one) in 3 minutes. Only few portions of the walls and the obstacles will hence be mapped and it is therefore more interesting to search for differences between the simulated map and the real one.

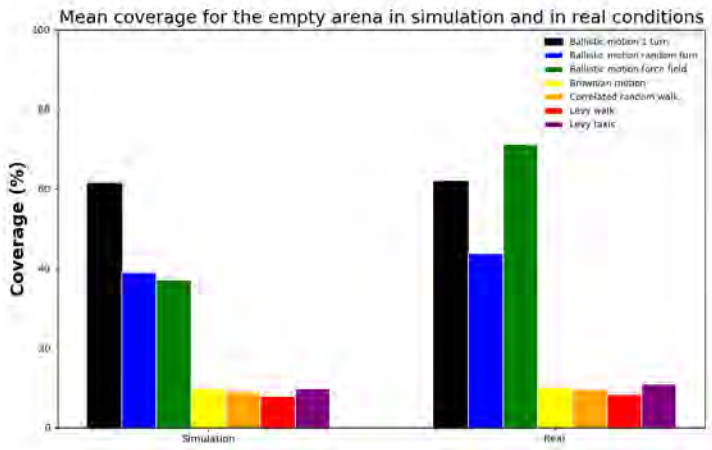


Figure 7.2 – Comparison of coverage results between real robot and simulation (empty arena)

7.4 Results and analysis

The following Sections present the different experiments and their results. First, the general performances and differences with the simulation are discussed. Then, a global conclusion compares the results of the different exploration behaviours in real experiments

A display of the mapping process at 6 different times can be seen in Appendix F.

7.4.1 Empty arena

In this experiment, a single real e-puck was released in the empty arena. The results were compared to the equivalent experiment in simulation and are displayed on Figure 7.2.

A few observations can already be made. First, the results for most of the exploration behaviours were similar in simulation and in reality. Regarding the coverage metric, the results in reality tended however to be slightly better because the e-puck moves a bit faster in reality than in ARGoS simulator.

New phenomena were found while experimenting with the real e-puck. For instance, the inertia of the robot took a more important role while mapping with ballistic motion and force field obstacle avoidance. While the result in reality seems better than in the simulation, this was actually due to errors caused by the e-puck being stuck against a wall. Indeed, the real robot was slightly quicker than the simulated model and because of its inertia, it reacted slightly too late and did not completely avoided the wall. Moreover, as the e-puck sensors encounter sometimes difficulties to detect obstacles touching the robot, it happened that the e-puck was not seeing the wall anymore. It continued in straight motion against the wall, making the odometry fail and introducing errors in the resulting map. Figure 7.3 illustrates this issue.

However, this error did not happen in every experiment. It usually happened just once per run for the ballistic motion. In comparison, the frequent changes of direction of the random walk behaviours helped them to avoid this problem in most cases, or at least minimize it. The issue never happened for the other ballistic motion as the fixed rotation and random rotation obstacle

(a) Ballistic motion with force field, simulation

(b) Ballistic motion with force field, reality

Figure 7.3 – Error in the real map with the ballistic motion with force field

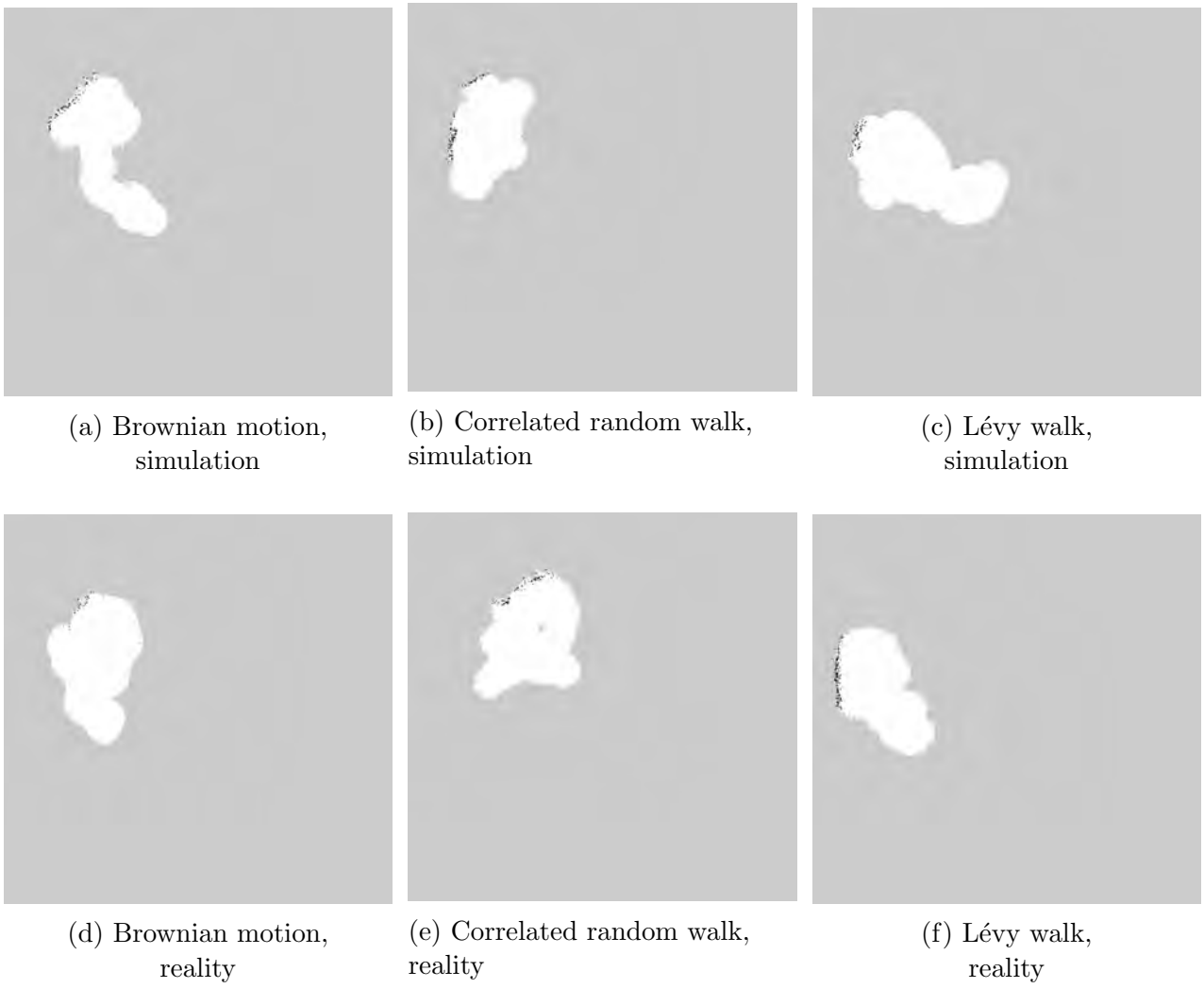
avoidance make the robot stop and turn on itself. Hence, the only maps being significantly affected were those created by the ballistic motion with force field obstacle avoidance.

This problem highlights the differences between a simulation, idealistic and ignoring such factors, and the imperfect reality. However, except for this particular case, all other maps looked coherent with reality and error-free.

Another observation that can be made is that the ballistic motion with fixed rotation and with force field obstacle avoidance, that were supposed to be deterministic, did not produce the same result at each run. One could think that this could be easily explained by little differences in the orientation of the e-puck at the beginning of each run. Indeed, despite all the care taken for the precise positioning of the e-puck, small differences can appear and change slightly the trajectory of the robot.

Moreover, to this phenomenon is added the heterogeneity in the e-puck sensors precision. As explained in Section 3.3, the sensors of the e-puck do not all have the same sensitivity contrary to the simulation. This could be also observed in other behaviours as the maps were not always similar between reality and simulation. Hence, small trajectory differences appeared due to the real e-puck nature and as those differences sum up, they leaded to a totally different result. Moreover, the precision for walls mapping was a bit lower in reality than in the simulation.

Figures 7.4 and 7.5 display maps built during the experiments and their equivalent in the simulation. Through visual inspection it could be noticed that although in some cases there are notable differences, on the whole the maps fall within the expectations.



(a) Brownian motion, simulation

(b) Correlated random walk, simulation

(c) Lévy walk, simulation

(d) Brownian motion, reality

(e) Correlated random walk, reality

(f) Lévy walk, reality

Figure 7.4 – Comparison of maps between simulation and reality

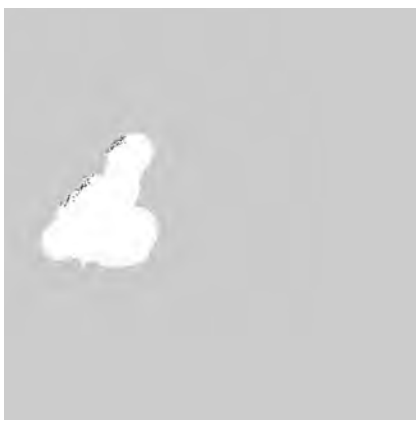
7.4.2 Arena with five obstacles

In this experiment, a single real e-puck was released in the arena containing five obstacles. The results were compared to the equivalent experiment in simulation and are displayed on Figure 7.6.

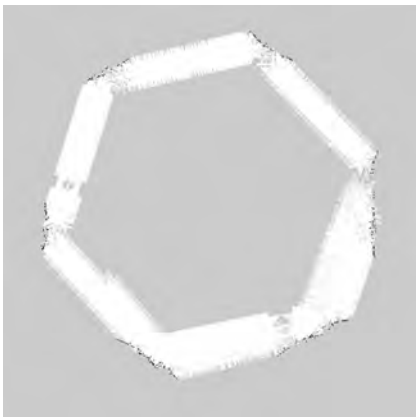
This experiment presented similar results as the previous one. The same observations can be made regarding the comparison between the simulated environment and reality, those were then confirmed by this experiment. However, due to the presence of additional perturbations, the obstacles, the differences were slightly higher than for the empty arena.

The same issue as in the empty arena also appeared here with the ballistic motion with force field obstacle avoidance: the robot sometimes got stuck against the wall. Figure 7.7 illustrates this issue.

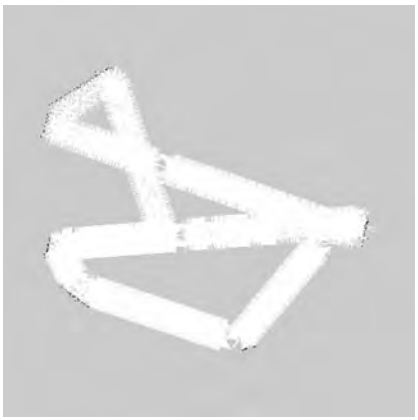
Additionally, it could be seen that the real e-puck was also able to map obstacles. Nevertheless, the robot often encounters and maps only a piece of the obstacle before moving in another direction. Therefore, it was only able to map small portions of the obstacles in such a short time. In a future work, experiments with multiple robots will be held in order to obtain better results for this scenario.



(a) Lévy taxis, simulation

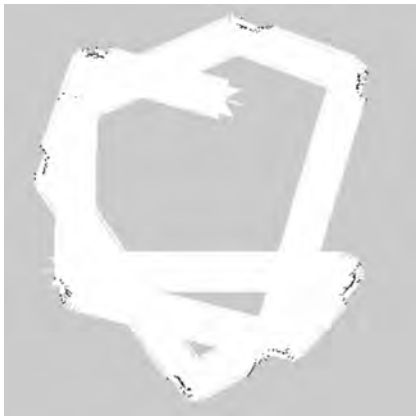


(b) Ballistic motion with fixed rotation, simulation

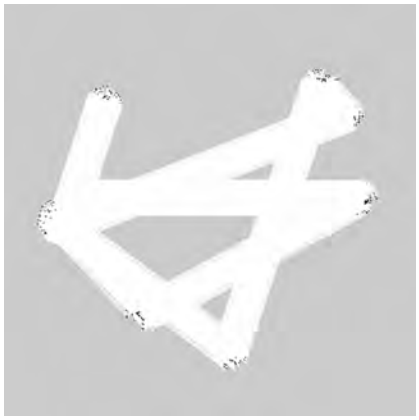


(c) Ballistic motion with random rotations, simulation

(d) Lévy taxis, reality



(e) Ballistic motion with fixed rotation, reality



(f) Ballistic motion with random rotations, reality

Figure 7.5 – Comparison of maps between simulation and reality

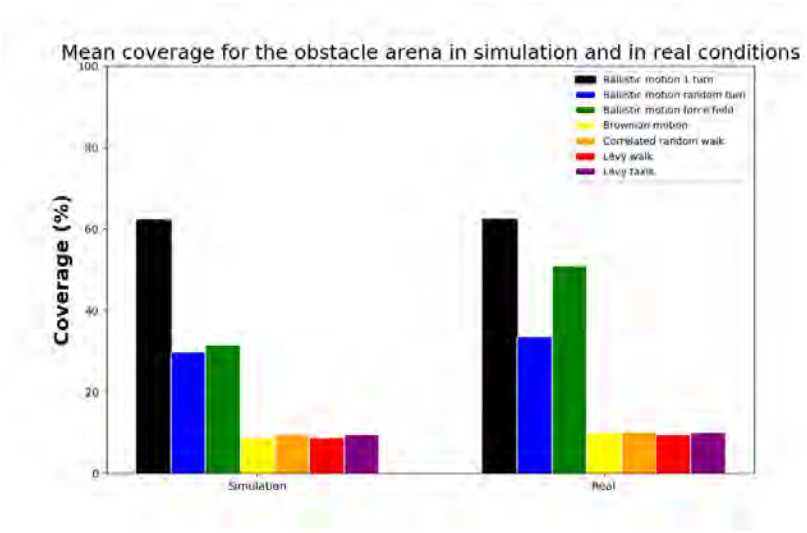
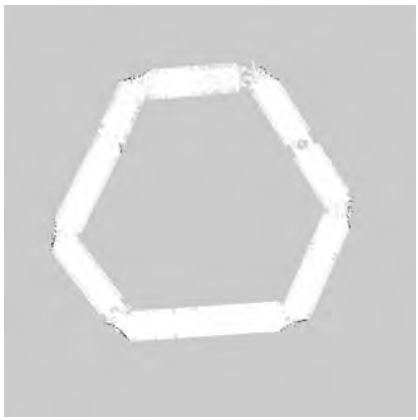


Figure 7.6 – Comparison of coverage results between real robot and simulation (obstacle arena)

Figures 7.8 and 7.9 display a few maps built during the experiments and their equivalent in the simulation. Similarly as the previous experiment, through visual inspection it could be noticed that on the whole the maps fall within the expectations.

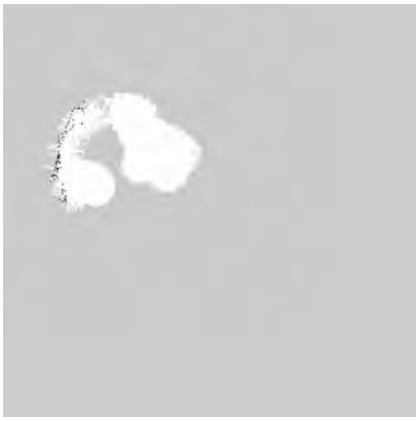


(a) Ballistic motion with force field, simulation

(b) Ballistic motion with force field, reality

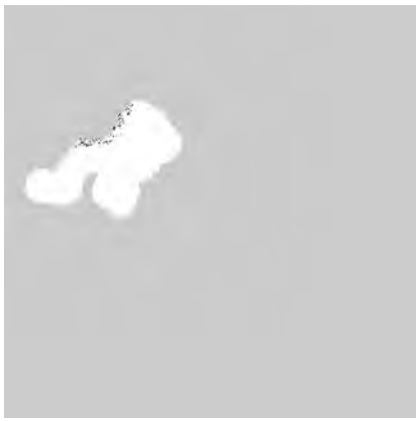
Figure 7.7 – Error in the real map with the ballistic motion with force field

(a) Brownian motion, simulation

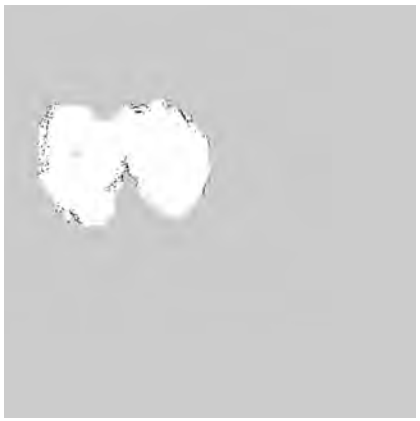


(b) Correlated random walk, simulation

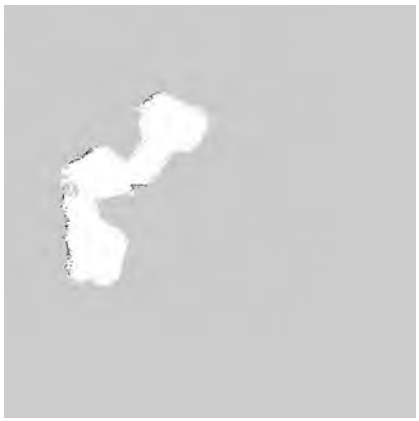
(c) Lévy walk, simulation



(d) Brownian motion, reality



(e) Correlated random walk, reality

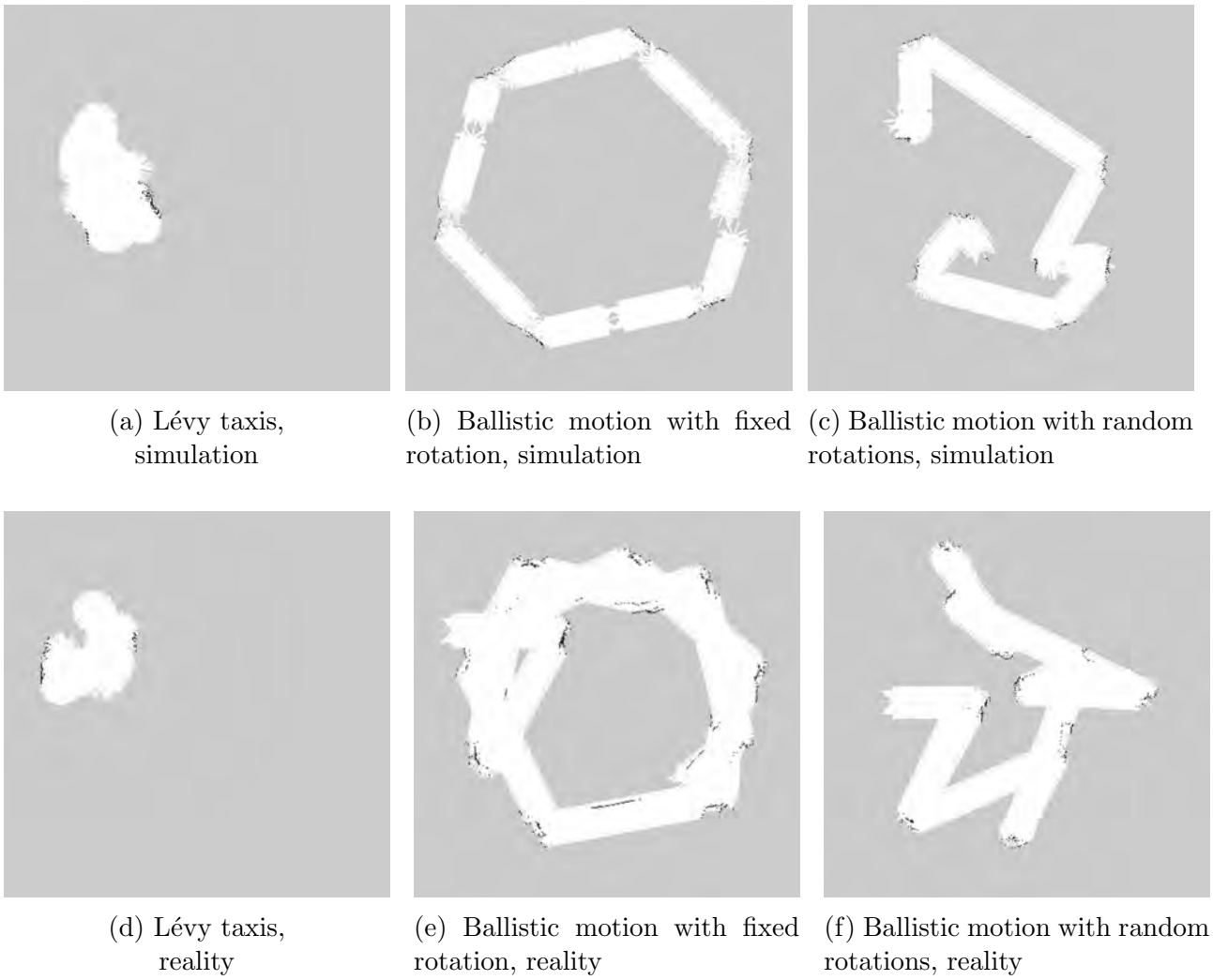


(f) Lévy walk, reality

Figure 7.8 – Comparison of maps between simulation and reality

7.4.3 Conclusion

The experimentation with real e-pucks added interesting results and observations to the research. First, the results confirmed those obtained in simulation which demonstrates that a



(a) Lévy taxis, simulation

(b) Ballistic motion with fixed rotation, simulation

(c) Ballistic motion with random rotations, simulation

(d) Lévy taxis, reality

(e) Ballistic motion with fixed rotation, reality

(f) Ballistic motion with random rotations, reality

Figure 7.9 – Comparison of maps between simulation and reality

real e-puck is able to map. Even though the maps produced have a lower quality, this was expected as the real e-puck introduces new and undefined perturbations in the system. The results were however positive and the reality gap showed to be surpassed to some extent.

The results are very promising regarding multi-robot mapping as the ROS merging tool used in the simulation would work similarly with the maps created by the robots. One could think that since the maps created here are very similar to the simulation results, fusing the maps of several e-pucks should behave similarly.

Chapter 8

Conclusion

In this work I have presented a set of improvements for the e-puck platform and the ARGoS simulator. They provide new capabilities to the e-puck as the development framework ROS can now be used together with ARGoS controllers to conduct experiments on ARGoS simulator and on real e-pucks. The interest of integrating ARGoS and ROS is here highlighted as the same controllers can be used for both simulation and real applications.

To achieve this result, I reviewed multiple configurations used in the literature including other robots, simulators, controllers, embedded computers and Linux distributions. It appeared however that the Ångström distribution used on the e-puck had to be changed and I selected Yocto Project to build a custom distribution that is more recent, better documented and easily customizable. I took also great care to keep this distribution as simple as possible to fit the e-puck technical limitations.

To easily develop e-puck material on this new distribution, I designed methodologies for developing ROS-based software covering the design of controllers, their cross-compiling and the integration of new packages.

Thanks to this new software architecture, I was able to conduct experiments on a new capability brought by ROS : mapping. The aim of those experiments was double : providing a first working application of ROS functionalities and performing an actual review of exploration behaviours in the context of mapping. E-pucks were then given the task to map closed hexagonal arenas containing different numbers of obstacles.

The results obtained in the simulation lead to interesting conclusions. Indeed, creating a map requires precision, often obtained by a good odometry, and sufficient area coverage, obtained by a quick exploration behaviour. It appeared then that a simple ballistic motion with obstacle avoidance outperformed more complex random walks behaviours. Nevertheless, the results pointed out that at least a few randomness was required in the exploration as fully deterministic strategies were not robust to a change of environment represented by different arena configurations.
The results also demonstrated that good quality multi-robot mapping was possible with robots as simple and short ranged as the e-pucks.

Real world experiments were also conducted to verify the results obtained by simulation. Except for small differences that were expected considering imperfections of real sensors, the maps created by a real e-puck were very close to those obtained in the ARGoS simulator. While

multi-robot experiments were not conducted yet because of their huge time investment and their complex setup, the results of such experiments seem promising.

Beyond multi-robot experiments, swarm experiments could also be performed on basis of this work. The exploration controllers used for the mapping experiments could indeed be improved by inter-robot communication and finally lead to interesting real world applications such as disaster area mapping. The only remaining obstacle is ROS dependency to a network, prohibited by swarm principles. Many solutions exist though in order to adapt ROS to the swarm philosophy and I believe that the standardization brought by this framework is already an interesting contribution in that extent.

Appendix A

Working with e-puck

This Appendix is structured as follow : SectionA.1 details the procedures to connect a computer to an e-puck or to the epucks network used at IRIDIA; Section A.2 presents the procedure to cross-compile and upload a controller on an e-puck running Ångström.

A.1 Interaction with the e-puck

Here are described the procedures to connect a computer to the epucks network and then to the e-pucks.

A.1.1 Connection to an e-puck

Connection via ssh
To be able to connect to an e-puck, a computer must be on the same network as the e-puck. At IRIDIA, the standard network used with the e-pucks is simply called epucks.

When connected, a connection can be established with any powered e-puck via ssh with the following command:

```
$ ssh root@10.0.1.EpuckID
```

In order to avoid the terminal to freeze, the ssh connection should be ended using :

```
$ halt -p && exit
```

Connection via cable
A connection with an e-puck can also be done using a mini-usb cable.
Many existing softwares allow then to connect to the robot. Here, picocom was chosen for its simplicity to use and its stability. The command line with picocom is :

```
$ sudo picocom -b 115200 /path/to/device
```

Example : $ sudo picocom -b 115200 /dev/ttyUSB0

By default, the login to an e-puck is 'root' and and the password is empty.

A.1.2 Connection to the epucks network

The epucks network available at IRIDIA needs a password to be accessed. If the password is unreachable for any reason, it can be acquired with the following procedure by connecting to a working e-puck :

1. Connect to the network with an ethernet cable;
2. Create a new static ip address for this network :

   $ sudo ifconfig NameOfMainAddress:0 10.0.1.XXX

   where NameOfMainAddress is the name of the ethernet address of the computer and XXX is any 0-255 number except those already attributed to the e-pucks. The ethernet address can be seen using ifconfig and should begin by enp or eth.

   Example : $ sudo ifconfig enp0s31f6:0 10.0.1.123
3. Connect to the e-puck via ssh (see Section A.1.1);
4. The password of the network is located in */etc/wpa_supplicant/wpa_IRIDIA.conf*;
5. The mask and the gateway of the network are located in */etc/network/interfaces*. The current mask is 255.255.255.0 and the current gateway is 10.0.1.1;

It should be noticed that the epucks network is not connected to the internet.

A.2 Working with e-pucks

E-pucks are used at IRIDIA to run swarm robotics experiments. Those experiments are performed using ARGoS simulator. ARGoS allows to use controllers designed for the simulation to be used without modifications on the real robots. It is however essential to cross-compile the controllers so they can be run by the e-pucks.

A.2.1 Cross-compilation of a controller

As embedded Linux architectures are different from architectures used on a standard computer, a file compiled on the computer will not work if uploaded as it on the robot. In order to be understandable for the e-puck, the controllers have to be cross-compiled using the arm tool (current used tool for the Ångström distribution is arm-unknown-linux-gnueabi). The cross-compilation is indeed the process of compiling files for a different architecture than the one of

the computer[43].

This Section presents the different steps necessary to use a custom controller on a real e-puck robot. As defined Section B.2, controllers are projects using CMake as a compilation tool. Such a CMake project can contain multiple controllers but the specific actions required for such a process are the same whatever the number of controllers we want to cross-compile. Hence, for the sake of clarity, it will be assumed in the following that we are compiling a CMake project containing an arbitrary number of controllers. An ARGoS experiment configuration file is also needed to launch the desired controller. However, this file does not need to be compiled and can be directly uploaded on the robot.

Configure the main CMake file
The cross-compilation process requires to configure properly the CMake file of the project. The following lines should be added to the main CMake file :

```
include(ARGoSBuildOptions.cmake)
include(ARGoSBuildFlags.cmake)
include(ARGoSBuildChecks.cmake)
include(ARGoSPackaging.cmake)
include(FindPthreads.cmake)
include(TargetEpuck.cmake)
[...]
add˙executable(exec˙name1 header1.h code1.cpp)
add˙executable(exec˙name2 header2.h code2.cpp)
[...]
```

The 5 first include statements aim to include the different ARGoS specific CMake files.
The last include statement targets a cross-compilation specific file designed by Ken Hasselmann and is required.
Finally, the last lines aim to create an executable for the controllers (1 line per controller with the corresponding .h and .cpp files).

All paths in the CMake should of course be adapted to the local file system and should point to the desired cross-compiling tool (arm-unknown-linux-gnueabi).

Cross-compile
In the project directory :

```
$ mkdir build
$ cd build
$ CMake -DCMAKE_BUILD_STYLE=Release
-DCMAKE_TOOLCHAIN_FILE=/path/to/cross-compiling/cmake/file ../src
```

The last line should of course be adapted to target the TargetEPuck.cmake file.

Example : $ CMake -DCMAKE_BUILD_STYLE=Release
-DCMAKE_TOOLCHAIN_FILE= /project/src/cmake/TargetEPuck.cmake ../src

A.2.2 Upload a controller on the e-puck

To transfer a controller executable and an experiment configuration file to the e-puck, a connection should first be established with the robot (see Section A.1.1). Then, the upload is done using the upload.sh script (found in *argos3-epuck/scripts*) with the following command lines :

```
$ upload.sh file/path/to/your/controller/ destination/path/on/robot/controller EpuckID
$ upload.sh file/path/to/your/config/ destination/path/on/robot/config EpuckID
```

It should be noticed that paths in the script may need to be modified to fit local file system.
Also, the destination path in the above command lines is usually /home/root/your_directory.

Launch an experiment

When desired controllers and configuration files are uploaded on the e-puck, an experiment can be launched with :

```
$ ./controller -c configuration -i controller_ID
```

The controller ID is usually the same as the controller name.

Example : $ ./epuck_ros -c epuck_ros.argos -i epuck_ros

Appendix B

ARGoS

This Appendix is structured as follow : Section B.1 highlights the main specifications of ARGoS; Section B.2 defines the ARGoS controllers; Section B.3 details the ARGoS XML configuration files; Section B.4 presents CMake, the compilation tool used in ARGoS projects; Section B.5 describes the procedure to include ROS in a C++ controller.

B.1 Overview

ARGoS is a widely used multi-physics simulator for swarm robotics experiments mainly developed by Carlo Pinciroli[80][81]. It supports different robot types and allows large-scale swarm simulations contrary to most other simulators. It also allows to use multiple physics engines for the simulation.

ARGoS can be easily customized trough plugins, in particular the "e-puck for ARGoS" plugin developed at IRIDIA[54] that is used to model the e-puck robot in ARGoS. The version used for this thesis is 3.0.0-beta48.

Creating an experiment in ARGoS is easy. It only requires :

- An XML configuration file (.argos);
- User defined codes including the robots controller and optionally other functions.
- Optionally, a loop function can be used to gather statistics or handle other behaviours such as a random element in the simulation.

B.2 Controller

A controller is a piece of software that describes the behaviour of a robot. Defining custom controllers for the robot is a key component of this work. In this work, the controllers are ARGoS controllers and are therefore C++ pieces of code gathered in a project.
The general structure of an ARGoS controller is fairly simple and can be resumed as follows :

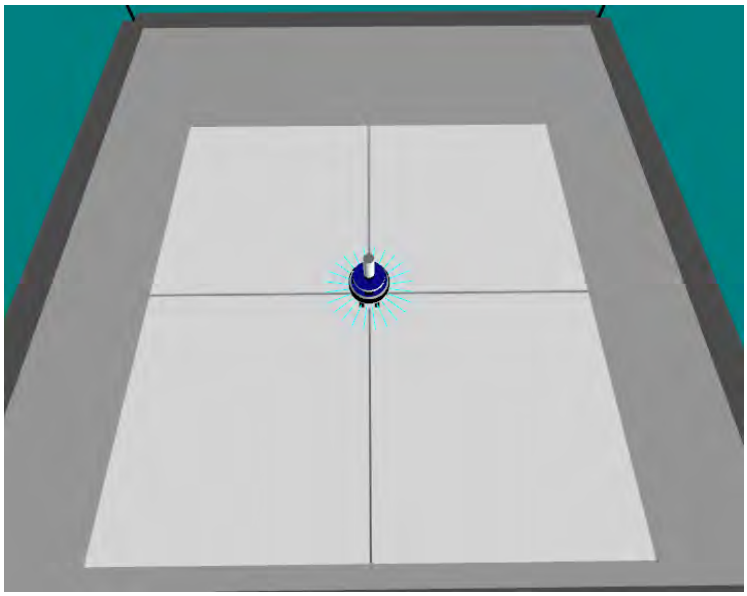


Figure B.1 – Arena example

- An init process that initialize all relevant variables for the task the controller is designed for. In particular, models of all relevant sensors and actuators are initialized here.
- A control step that describes the behaviour of the e-puck at each time step. In particular, values of the sensors are read and actuators are configured accordingly.

It can finally be mentioned that ARGoS supports controllers implemented in the LUA language[86].

B.3 Configuration file

The configuration file is an XML-like file with a .argos extension. It sets up the arena (and the potential obstacles) in which the experiment takes place, the number of robots and their type, the physics engine and which controller to use for the robots, as well as various other options. The configuration file also defines which sensors and actuators are available as well as their implementation.
While most of the parameters are used essentially for simulation, this file is still necessary to run experiments on real robots.
Figure B.1 shows an example of an arena in ARGoS.

B.4 CMake

CMake is an open-source system managing the build process in an operating system. It is compiler-independent and is meant to be used together with the native build environment. Hence, the same CMake configuration can be used almost without any modification to build a project on a Linux distribution or on Windows.

CMake uses simple configuration files (CMakeLists.txt) placed in each directory and subdirectory of the source we want to compile. When running the build command, those files will

generate the standard build files (like makefiles in Unix, etc) of the native build environment. A CMake file consists in 1 or more commands of the form COMMAND(arguments). Various commands are pre-defined by CMake but new ones can be defined by the user. Those commands define build options, directories to place built libraries and executable and even which compiler to use. CMake is then fully compatible with cross-compilation (see Section A.2.1).

Once each required CMake file has been placed in the corresponding directory, the build process can be initiated by a simple command :

```
$ CMake path/to/first/src/directory [-OPTIONS]
```

Options can be used, for example, to specify a particular variable or a path to some files.
The usual way to build a project using CMake is to create a build directory and once inside, use something similar as :

```
$ CMake ../src
```

B.5 ROS in ARGoS

To allow the e-puck to use ROS, its controller has to have access to the ROS library (ros.h in C++), to launch the ROS **ros::init** function giving a unique name to the node representing the controller and finally it must create an instance of the **NodeHandle** class. This object will be used to subscribe and publish to topics and receive or send messages like steering commands or sensors data.

Any number of subscriber (**ros::Subscribe**r) can be created through the **NodeHandle::subscribe** function by specifying the topic name, the size of the message queue and a handler function that will use the message content. The handler function must be defined by the user and must take as argument the receive message, hence the argument type must be the same as the topic message type. As a general remark, a message type can be one of the many built-in ROS message types but also a custom one (see Section C.1). Also notice that once the subscriber object has been created, no more interaction is needed with it, the handler function will do the job.

Similarly, any number of publisher (**ros::Publisher**) can be created through the family of functions **NodeHandle::advertise<msgType>**, where msgType must specify the type of message published. The topic where to publish data and the message queue size must also be provided as arguments. Then, a message of the appropriate type can be published whenever one wants through the **Publisher::publish** function by giving it a correct message object.

Services work in a similar way : a **ros::ServiceClient** object must request a service through the **NodeHandle::serviceClient<srvType>(srvName)** function and a **ros::ServiceServer** object must provide it through the **NodeHandle::advertiseService(srvName, handler)** function, where the handler is a Boolean function that takes special objects **srvType::Request &req** and **srvType::Response &res** as argument and should return true if nothing went wrong. Attributes of **req** and **res** should be defined in the corresponding service file (see section C.1).

Finally, for all used messages and services types, the corresponding header files must be in-

cluded otherwise compilation will fail.
Besides, in order to send and receive messages, an instance of **ros::spin** function must be called to pump messages from buffers. In the case of ARGoS controllers, **ros::spinOnce** inside the **ControlStep** function works fine. It should also be noticed that usually, ROS periodic actions are called within a **while(ros::ok())** loop but as the **ControlStep** function is already called periodically, a simple **if(ros::ok())** statement suffices.

The following code skeleton illustrates the minimal code to include in a standard ARGoS controller to use ROS :

```
1   #include "ros/ros.h" //include ROS library
2   [...]
3   void CEPuckRos::Init(TConfigurationNode& t_node) {
4       //init e-puck
5       [...]
6       char** argv = NULL;
7       int argc = 0;
8       ros::init(argc, argv, "epuck"); //creation of e-puck node
9       ros::NodeHandle rosNode; //access point to ROS features
10  }
11  void CEPuckRos::ControlStep() {
12      //do stuff
13      [...]
14      ros::spinOnce(); // pump/send messages
15  }
16  [...]
17
```

Appendix C

ROS

This Appendix is structured as follow : Section C.1 presents an overview of ROS architecture; Section C.2 introduces a few useful ARGoS tools.

C.1 Architecture

C.1.1 Overview

Robot Operating System (ROS) is a development framework that provides lots of libraries and tools for robot applications development[66] such as map generation, steering, messages sending, etc. One of the main goal of ROS is to be as thin and modular as possible to make reusing of code easier.

ROS uses a network of distributed processes called **nodes** that are loosely coupled and interact with each other using ROS communication infrastructure. In order to use ROS features, the main node, roscore, must be called in a terminal and has to run as long as ROS is used. From then, any node can be called in a terminal or in a program.
Each node serves a particular purpose : path planning, control of an actuator, visualization, etc. Some nodes can be grouped into a package to easily provide a more complex feature. ROS also allows to define custom messages structures inside packages as well as services, a kind of request/reply structure. This whole system provides a very flexible and robust interface to perform various tasks useful in robotics.

Messages
Messages works with a many-to-many publish/subscribe system on topics : a node can publish messages on topics and receive messages from the topics it has subscribed to, each topic being dedicated to a specific type of message (built-in or custom).

ROS messages consist in simple .msg text files that must be added in a msg directory within the project directory. They can contain any number of data fields, each of them defined by a data type and a name. Common data types are boolean, string and all sizes of integers and floating point numbers as well as time and duration. The type of the message will be simply

the name of the file (without .msg extension).
For example, a steering message for the wheels of an e-puck could consist of 2 float32 numbers named leftSpeed and rightSpeed contained in a steering.msg file.
A huge variety of messages can then be defined. However, the use of such messages should only be used for internal communication. External communication (with other modules or robots) should indeed be handled by standardized messages to increase reusability of the code and compatibility with new modules.

Services
Services work similarly as messages but provide only 1-to-1 communication : a node provides a service and another one can request it via its name, in a way similar to client-server applications. Services consist in simple .srv text files that must be added in a srv directory within the project directory. They are defined in a similar way as messages except that they are composed of 2 series of data fields separated by "—" :

- The first series defines the inputs of the service
- The second series defines the outputs of the service

ROS services are only interfaces defining ins and outs : the actual implementation of a service should be done in the node that provides it. Again, the name of the service is the file name (without .srv extension).

Compilation
Another feature of ROS is that it is fully compatible with CMake compilation tools (see Section B.4), which offers comfortable building methods for projects. This is provided by the catkin tool[101]. Catkin is a collection of CMake macros and associated code used to build packages in ROS. All needed ROS libraries as well as custom defined messages and services must be added into the CMake file and can be easily built using the catkin_make command line. .

C.2 ROS tools

ROS provides a wide range of tools working as individual nodes. Here is a description of some interesting ones.

C.2.1 rqt

rqt is a multi-function tool that implements various GUI tools as plugins. It is useful to display all needed tools in the same interface, though it is still possible to use it for a single tool. In particular, it provides a steering controller that controls the robot speed (linear and angular).

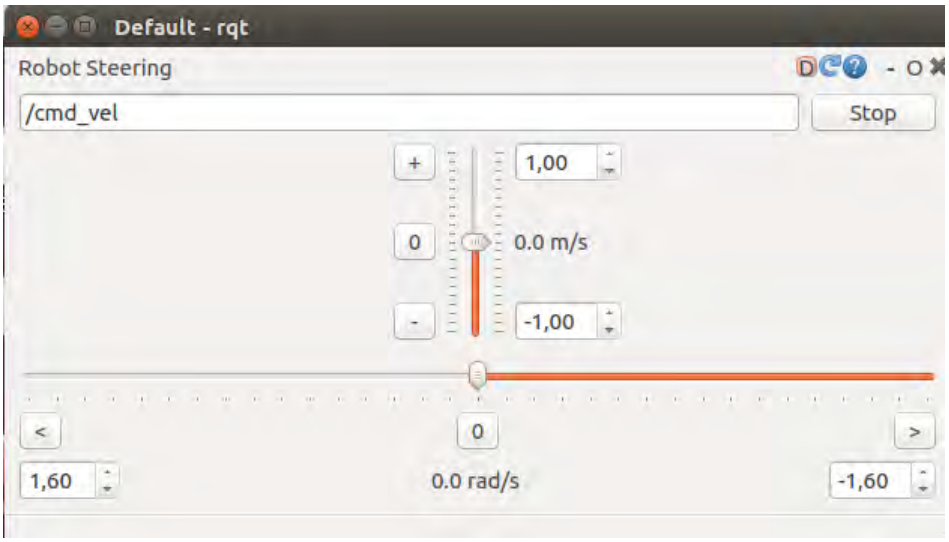


Figure C.1 – rqt steering

C.2.2 rviz

rviz is a visualization tool that can display diverse information such as maps, laser scans, odometry, etc. The desired topics can be selected through a list and added to the view window. As *rviz* uses ROS internal clock, some desynchronization may happen and *rviz* has to be restarted.

C.2.3 rosbag

Thanks to the *rosbag* package, ROS can play and record bag files, special files that contains all messages published on specified topics. The bags can then replay those messages in the same order, which is useful to analyze data, to reproduce an experiment or even in collaborative debugging as errors can be reproduced without caring about the simulator used for the experiment.

C.2.4 gmapping

gmapping is a package that manages map creation from data provided by a robot. It requires laser scans, odometry and tf transforms to output a map on the */map* topic. Laser scan data are proximity sensors readings remapped to real distances in meters and odometry represents the robot movements.

Tf (transform frame) data are combinations of translations and rotations called frames that describe the position of robot components with respect to a reference point, the base_link frame, that corresponds for e-pucks to the center of the robot at ground level. Tf frames describe then the needed translation and rotation that must be performed from this reference point to reach the position and orientation of the component.
A frame is required for :

- each of the 8 proximity sensors (base_prox0 to base_prox7)

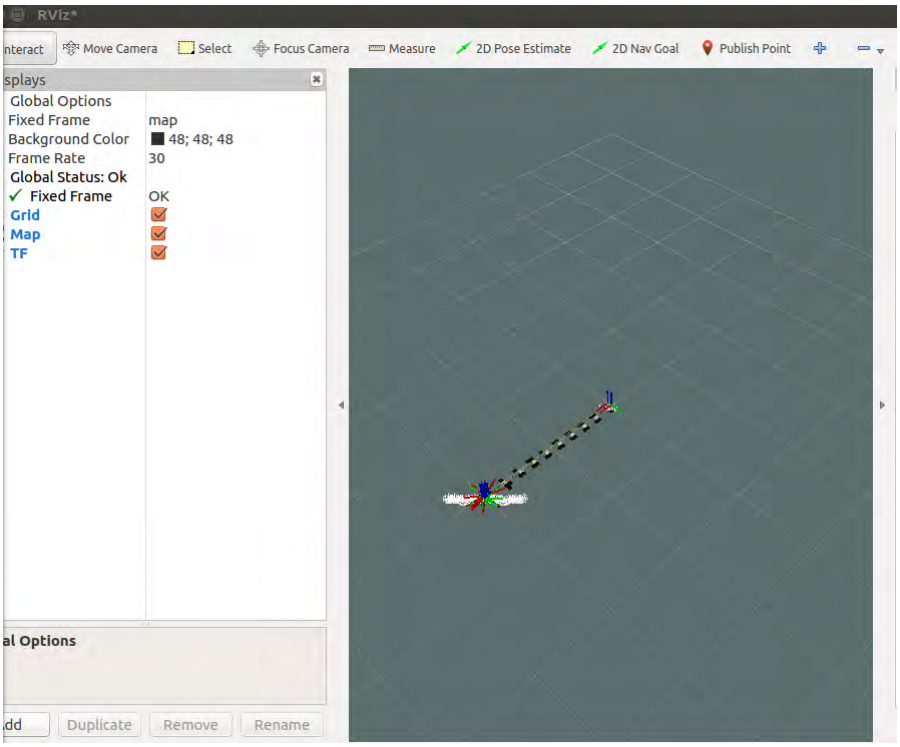


Figure C.2 – rviz

- the laser scanner (base_laser)
- the odometry (odom)

The base_laser frame is located in the center of the robot, at sensors level. The odom frame is particular as it represents the position of the robot and does not necessarily represent a physical device.
All frames need to send at each step:

- a tf::Transform object[1] comprising :
  - xyz coordinates
  - rpy orientation
- a timestamp
- the frames ids of the parent and the child frame

As tf transforms can be seen as a tree, it is important to explicitly tell which is the parent and which is the child so that the transforms can apply in the right order. For the e-puck, the child is always the base_link and the parent, the considered frame. The tf tree can be generated using the view_frames service of the tf ROS node, allowing a better understanding of the system.

In order to publish through ROS the tf data, a tf::TransformBroadcatser object from tf library is needed. It will send all four information stated above to the tf topic. This object must be a static variable in the code, otherwise successive frames will not be correlated.

[1]Odometry needs special geometry_msgs::TransformStamped that basically does the exact same thing as the Transform but integrates frames id and timestamp and can thus be sent alone

Launch files

ROS also allows to run multiple nodes at the same time using special XML files called launch files. Those files are simple ways of launching required nodes and setting their parameters. They can be used with the roslaunch command in a terminal by specifying the name of the launch file. Besides, if the roscore node is not running, roslaunch will create an instance of it.

multirobot_map_merge

This package provide a simple tool to merge multiple maps into a single complete map. It only requires to receive the individual map topics as /<ROBOT_NAME>/map.
Moreover, the initial position (x, y, z) and orientation (yaw) should be set as parameters in the robot namespace.

Appendix D

Yocto Project

Yocto Project[85] is an open source project helping developers to create custom Linux distributions for embedded systems. It provides for that a wide set of tools to build, test and maintain systems regardless of the hardware. On top of those tools, Yocto provides a powerful build system co-maintained with the OpenEmbedded Project and a reference distribution, Poky.

The rest of this Appendix is structured as follow : Section D.1 describes the architecture of Yocto Project; Section D.2 presents BitBake and the Poky distribution, 2 important tools provided by Yocto.

D.1 Architecture

The structure of a project using Yocto is based on a model called Layer Model. A layer in Yocto is a repository containing related packages and files along with the instructions to build them. A project is then composed of multiple layers that form together a complete Linux distribution for an embedded system.
The building configuration of a layer is specific to it and can then override general settings if necessary. Parameters specific to a layer can be added to a .conf file that will set the general configuration of the entire layer.
Layers name are of the form "meta-layerName".

Each layer contains multiple components that need to be built. To each component is associated a recipe, a text file that contains instructions of how to build it (more information in Section D.2.1. Components are usually gathered by dependencies in directories named as "recipes-compName".

This whole architecture is highly modular as new recipes or layers can be added without any modification of the existing ones.

D.2 Tools

D.2.1 BitBake

BitBake[88] is the build tool of the OpenEmbedded build system and it has been integrated in Yocto Project. BitBake is essentially a generic task execution engine that provides a very easy way to build systems defined by the Layer Model. The BitBake tool uses metadata stored in recipe (.bb), configuration (.conf) and class (.bbclass) files to build up and execute tasks.

Recipe
Recipes are files containing compilation information about a specific component such as dependencies, compilation flags or installation instructions.
A recipe typically contains (among many other parameters) :

- Information about the component, its creator(s) and possibly license data
- A way to fetch the source files of the component, either an URL link to some repository (on Git for example) or a specific location on the computer. It is possible to specify a specific release or commit of the repository as well as indicating the location of patches that will be applied on related files automatically.
- Compilation and possibly installation commands (do_compile and do_install) that can be configured at will, to specify compilation flags, target directories for the compiled binaries, etc.

Recipes can also be easily reused using .bbappend files. This allows to use an existing recipe (even from another layer) as a base for a new recipe, only adding additional parameters.

Class
Class files are a way to abstract common settings and share them between multiple recipes. A class can hence contain anything that can be found in a recipe.
Class files are included in a recipe through the *inherit* statement.
An important example of the usefulness of classes is that it allows to specify which build tool to use for the compilation of a recipe. It is indeed possible to use CMake (see Section B.4) or even catkin (see Section C.1) to build a specific component by adding a simple *inherit* statement in the recipe.

Configuration
Those files are mostly used to define general parameters related to the target machine, the distribution, the compiler or any option that should be apply to most of the recipes. Typically, each layer contains a configuration file that provides settings common to the whole layer.

After getting all necessary compilation parameters, BitBake creates a dependency tree, schedules the compilation and finally builds everything.

The bitbake tool can be used to build a single component, an entire layer or the complete system image. If more than 1 component has to be built, bitbake will seek the required component in the configuration file of each layer (layer.conf) it has to build. Hence it is possible to specify which layers/component have to be taken into account during the compilation.
The command line is :

```
$ bitbake <TARGET>
```

where <TARGET> is the name of a component, a layer or the system image.

D.2.2 Poky

Poky[82] is a reference embedded distribution for Yocto Project. Although it is not usable as it for most of possible applications, it provides a strong base to create custom distributions. As the architecture of Yocto (see Section D.1) is very flexible and modular, components can be easily added to Poky to satisfy the need of a specific application.

Appendix E

Installation of Poky on the e-puck

This Appendix provides a detailed procedure to build a Yocto image including ARGoS and ROS and to configure properly the created system, including the Wi-fi configuration.

The rest of this Appendix is structured as follow : Section E.1 describes the base installation of a Yocto Project distribution; Section E.2 details the additional operations required to install ROS on the Yocto image; Section E.3 defines the ARGoS specific steps necessary to install it on the distribution; Section E.4 presents the complete procedure to enable the Wi-fi on the system; Section E.5 proposes a summary of the whole procedure.

E.1 Bare-bone installation

This Section describes the base installation of Poky on an a-puck. Two installation procedures are detailed : a first installation from scratch and a second based on a pre-built image.

E.1.1 Installation from scratch

Configuring and installing Yocto from scratch is a long process so it should be done only for major updates of the distribution. It allows however to completely customize the installation of Yocto.

The following procedure is based on the Gumstix Yocto manifest[97] and ROS on Gumstix[57] but a few modifications have been applied to fit the specific requirements of this work.

1. Install Repo (a script used to configure the build environment of Yocto) :

   - Download the Repo script:

     $ curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > repo

- Make it executable:

  ```
  $ chmod a+x repo
  ```

- Move it on to your system path:

  ```
  $ sudo mv repo /usr/local/bin/
  ```

2. Initialize a Repo client :

   - Create an empty directory to hold your working files:

     ```
     $ mkdir yocto
     $ cd yocto
     ```

   - Make it executable:

     ```
     $ chmod a+x repo
     ```

   - Tell Repo where to find the manifest:

     ```
     $ repo init -u git://github.com/gumstix/yocto-manifest.git -b fido
     ```

     The version used here, Fido, can be replaced by any stable version of Yocto if needed. However, this work has been performed only on Fido so there is no guarantee it works with other versions, although versions close to Fido should work with no or few modifications.

   - Fetch all the repositories (this may take some time) :

     ```
     $ repo sync
     ```

3. Initialise the Yocto Project Build Environment :

   - Configure correct Linux kernel version by uncommenting (remove "#") the following line in *yocto/poky/meta-gumstix-extras/conf/local.conf.sample* :

     PREFERRED_VERSION_linux-gumstix = "3.5.7"

   - Copy default configuration information and set up environment variables

     ```
     $ export TEMPLATECONF=meta-gumstix-extras/conf
     $ source ./poky/oe-init-build-env
     ```

     Configuration files (.conf) can be customized at will, however it was not needed here. Also, poky is the name of the latest version of the Yocto Project OS, hence this should be adapted if needed.

4. Add additional packages :

   - Yocto includes by default a few packages such as wget or vim utilities but any additional package, such as ROS or custom packages needs to be added to the recipe.

- Add each package to the UTILITIES_INSTALL variable of *poky/meta-gumstix-extras/recipes-images/gumstix/gumstix-console-image.bb*
  The syntax of each line is as follow :

  ```
  package˙name  \
  ```

5. Build an image :

   - Create an image in *build/tmp/deploy/images/overo*

     $ bitbake gumstix-console-image

     This is the longest part of the procedure. Exact time may vary but it will for sure last at leasta few hours. This is only true for the first compilation. Building a single package can be done in a similar way :

     $ bitbake <PACKAGE_NAME>

6. Create a bootable micro SD card :

   - Verify that path to sdimage-gumstix.wks is correct
     The kickstart file (.wks) describes the layout of the storage. Its location should be modified from *poky/meta-gumstix-extras/scripts/lib/wic/canned-wks/sdimage-gumstix.wks* to *poky/meta-gumstix-extras/scripts/lib/image/canned-wks/sdimage-gumstix.wks*
   - Modify the name of uImage file
     For some reason, last versions of Yocto Project changed the denomination of some configuration files names: uImage was replaced by zImage.
     However, the bitbake operation still creates files using the old uImage denomination. It can be fixed simply by renaming the link file uImage to zImage (renaming the other files containing uImage in their name is not needed as the link file is the only one accessed by other scripts).
   - Create a .direct file for the uSD card

     $ wic create sdimage-gumstix -e gumstix-console-image

     The .direct file is a DD-able image that fits into a 2GB uSD card.
     It will be created in */var/tmp/wic/build/* and should be named similarly as *sdimage-gumstix-201506231742-mmcblk.direct*

7. Flash the image :

   - Install bmap-tools :

     $ sudo apt-get install bmap-tools

   - Create a bmap file :

     $ bmaptool create /var/tmp/wic/build/imageName.direct ¿ image.bmap

     where imageName is the name of the .direct file created step 6.

- Flash uSD :

  $ sudo bmaptool copy –bmap image.bmap /var/tmp/wic/build/imageName.direct /dev/<DEVICE NAME>

  where imageName is the name of the .direct file created step 6 and <DEVICE NAME> is the name of the uSD card (for example : mmcblk0).

E.1.2 Installation from pre-built image

The following procedure is based on pre-built image that can be downloaded from the Overo extension page[40]. This allows to skip most of the previous described procedure but not to include additional packages. This procedure is then explained only for the sake of completeness and in the following Sections, all references to the base procedure refer hence to the first procedure (installation from scratch).

As this pre-built image is composed of MLO and u-boot.img files, the wic tool of Yocto Project (cf step 6 of Section E.1.1) is no longer usable. It is then required to prepare the uSD card "manually" by following the procedure described on the Gumstix website[46]. The following steps are then based on this procedure but contains some modifications to adapt it to the present work. Note that ROS packages are already included in the image.

1. Download yocto-ros-3.5.7-FAT.tar.gz and yocto-ros-3.5.7-rootfs.tar.gz from the GCtronic website[40] :

   Those archives contain the MLO and u-boot.img files required for the OS to work.

2. Prepare the uSD card :

   - Partition the uSD using mk2partsd script :

     $ sudo ./mk2partsd /dev/<DEVICE NAME>

     where <DEVICE NAME> is the name of the uSD card.

     Example : $ sudo ./mk2partsd /dev/mmcblk0

     However, the script that can be found on the Gumstix website[46] needs to be modified : every 'ext4' in the script should be changed to 'ext3'.
     Also, other versions of the script can be found on the internet and may have a sfdisk statement followed by many options (-D -S etc). For this statement to work, it should look like :

     sfdisk –force $DRIVE << EOF

   - Mount the uSD card partitions :

     $ sudo mkdir /media/boot,rootfs

```
$ sudo mount -t vfat /dev/<DEVICE NAME>p1 /media/boot
$ sudo mount -t ext3 /dev/<DEVICE NAME>p2 /media/rootfs
```

where <DEVICE NAME> is the name of the uSD card.
The first line is only needed if those directories don't exist already.

Example :

```
$ sudo mount -t vfat /dev/mmcblk0p1 /media/boot
$ sudo mount -t ext3 /dev/mmcblk0p2 /media/rootfs
```

- Copy the files on the uSD card :

```
$ sudo cp MLO /media/boot/MLO
$ sudo cp u-boot.img /media/boot/u-boot.img
$ sudo tar -xzvf yocto-ros-3.5.7-rootfs.tar.gz -C /media/rootfs
```

The MLO and the u-boot.img files can be found in the yocto-ros-3.5.7-FAT.tar.gz archive.

- Unmount the uSD card :

```
$ sudo umount /media/boot
$ sudo umount /media/rootfs
```

E.2 ROS setup

This Section describes the different steps to install ROS on the Poky distribution as well as the additional configuration required to make it work properly.

E.2.1 Installation

ROS installation is pretty straightforward as it only requires to add ROS packages to the image. As mentioned at step 4 of the base installation (section E.1.1), additional packages must be appended to the UTILITIES_INSTALL variable of the *poky/meta-gumstix-extras/recipes-images/gumstix-console-image.bb* file.
A base installation of ROS requires the following packages (following the syntax of the .bb file) :

```
packagegroup-ros-comm \
python-wstool \
python-email \
python-distutils \
git \
git-perltools \
```

```
python-rosinstall \
rospy-tutorials \
roscpp-tutorials \
screen \
```

Any additional package can be added in a similar way. Typical packages are :

```
roscpp \
std-msgs \
geometry-msgs \
nav-msgs \
tf \
```

E.2.2 Configuration

Although ROS should now be installed on the e-puck, it is not completely ready to work. The following lines[1] should be added to /etc/profile :

```
export ROS_ROOT=/opt/ros/indigo
export PATH=$PATH:/opt/ros/indigo/bin
export LD_LIBRARY_PATH=/opt/ros/indigo/lib
export PYTHONPATH=/opt/ros/indigo/lib/python2.7/site-packages
export ROS_MASTER_URI=http://localhost:11311
export CMAKE_PREFIX_PATH=/opt/ros/indigo
export ROS_PACKAGE_PATH=/opt/ros/indigo/share
touch /opt/ros/indigo/.catkin
```

The ROS_MASTER_URI environment variable is here set to point to localhost but it can be replaced by another IP address to connect to a master over the network.
It should also be noticed that *indigo* refers to the ROS distribution and should be adapted if another distribution is used.

E.3 ARGoS setup

This Section describes the different steps to install ARGoS and its e-puck plugin on the Poky distribution. Once installed, everything should work with only a small modification in the configuration.

E.3.1 Installation

Installing ARGoS on the Poky distribution is conceptually very simple : it "suffices" to write a BitBake recipe for it, add it to the image packages and build everything as usual. However,

[1] Partially coming from ROS on Gumstix[57].

writing the recipe was actually a really tricky task and many unexpected bugs showed up during the process.

Below is described the procedure to configure Yocto environment and the most important parts of the recipes for both ARGoS and its e-puck plugin. In the following, it is assumed that the 3 first steps of the base installation (section E.1.1) have been performed. The */home/<USERNAME>/* directory should then contain a *yocto/* folder containing itself both *poky/* and *build/* folders.

Configuring Yocto
In order to successfully compile ARGoS, some modifications have to be done on a few files from Yocto Project :

- *poky/meta/classes/cmake.bbclass* file :

  1. Modify the following line :

     ```
     set( CMAKE_FIND_ROOT_PATH ${STAGING_DIR_HOST} ${STAGING_DIR_NATIVE}
     ${CROSS_DIR} ${OECMAKE_PERLNATIVE_DIR} ${OECMAKE_EXTRA_ROOT_PATH}
     ${EXTERNAL_TOOLCHAIN})
     ```

     to make it look like :

     ```
     set( CMAKE_FIND_ROOT_PATH /home/<USERNAME>/yocto/build/tmp/
     sysroots/overo/usr/include /home/<USERNAME>/yocto/build/
     tmp/sysroots/overo/usr/lib
     ${STAGING_DIR_HOST}${STAGING_DIR_NATIVE} ${CROSS_DIR}
     ${OECMAKE_PERLNATIVE_DIR} ${OECMAKE_EXTRA_ROOT_PATH}
     ${EXTERNAL_TOOLCHAIN})
     ```

     where <USERNAME> should be replaced by the actual user name.
     This is required as it seems that BitBake encounters difficulties to find freshly built library. In this case, compiling the e-puck plugin of ARGoS requires some libraries from ARGoS core.

  2. Add (at the end of the file for example) :

     ```
     # By default, install stuff in the toolchain tree
     set(CMAKE_INSTALL_PREFIX /home/panda-cool/yocto/build/tmp/
     sysroots/overo/usr CACHE STRING "Install path prefix,
     prepended onto install directories.")

     # Compile with optimizations
     set(CMAKE_BUILD_TYPE Release CACHE STRING "Choose the
     type of build")

     # Configure ARGoS flags
     set(ARGOS_BUILD_FOR epuck)
     set(ARGOS_DOCUMENTATION OFF)
     set(ARGOS_DYNAMIC_LIBRARY_LOADING OFF)
     set(ARGOS_THREADSAFE_LOG OFF)
     set(ARGOS_USE_DOUBLE OFF)
     set(ARGOS_INCLUDE_DIRS /home/<USERNAME>/yocto/build/tmp/
     ```

```
sysroots/overo/usr/include)
set(ARGOS_LIBRARY_DIRS /home/<USERNAME>/yocto/build/tmp/
sysroots/overo/usr/lib/argos3)

set(ARGOS_DYNAMIC_LIBRARY_LOADING OFF)
```

where, again, <USERNAME> should be replaced by the actual user name.
Those lines set up the ARGoS variables for the compilation and are adapted from the TargetEPuck.cmake file that was previously used to cross-compile ARGoS file for Ångström. Those lines could have been patched to the existing ARGoS files but as they do not interfere with any other build process, this is simpler and without any issue.
It is worth noticing that the last line is crucial as it avoids an **undefined function error** at line 496 of the *real_epuck.cpp* file in the ARGoS for e-puck plugin.

- *poky/meta/conf/distro/include/as-needed.inc* file :

  1. Modify the following line :

     ASNEEDED = "-Wl,--as-needed"

     to make it look like :

     ASNEEDED = "-Wl,--no-as-needed"

     This parameter seems to lead to linker errors when not set properly. The –no-as-needed option was already set in ARGoS CMake files but does not seem to have any effect when compiling with BitBake, this modification is then required.

ARGoS recipes

In order to be compiled by BitBake, recipes are required for both ARGoS and the e-puck plugin. The structure of those files is the following :
Figure E.1 shows the directory tree of a portion of the project. The recipes files (.bb) can be

Figure E.1 – Directory tree of the project (truncated)

seen in the tree as well as some appendices (.bbappend) that solve a bug with the linker. A

folder containing a patch (.patch) file can also be seen.
As the ARGoS source files are downloaded directly from Github, a patch is required to perform some necessary modifications on the files.
Some important points are discussed below :

- In both recipes, the SRCREV variable is set to a long hexadecimal number representing a particular commit tag of the Git repository. This commit correspond to ARGoS 3.0.0-beta48 (and the equivalent for the e-puck plugin). This is important as the last version of ARGoS, 3.0.0-beta52, is not yet fully compatible with the e-puck plugin.
- Both patches modify the flags for the C and the C++ compilers (in *ARGoSBuildFlags.cmake*). In particular, they add the -ldl flag which is required in order to avoid linking error of the base libraries (ldsym, lderror, etc).

It is finally worth mentioning that if a future patch needs to be created, it must satisfy those two points :

- The patch must obviously be done on the same commit as the one specified by the SRCREV variable in the recipe.
- When creating the patch, Git will add some header lines containing paths similar to *a/src/...* and *b/src/....* Due to the directory tree created by BitBake, the *src/* part of all those paths must be removed.

Installation
This is actually the simplest part. Indeed, it only requires to add the 2 packages to the image. Similarly as what was done for ROS (section E.2.1), it suffices to append the following lines to the UTILITIES_INSTALL variable of the *poky/meta-gumstix-extras/recipes-images/gumstix-console-image.bb* file :

```
argos \
argos-epuck \
```

There is however a bug related to the compilation of those 2 packages. For some unknown reasons, a hidden folder, *.debug*, will be created in *build/tmp/work/cortexa8hf-vfp-neon-poky-linux-gnueabi/argos/1.0-r0/packages-split/argos/usr/lib/argos3/.debug* (similarly for the plugin in *build/tmp/work/cortexa8hf-vfp-neon-poky-linux-gnueabi/argos-epuck/1.0-r0/packages-split/argos-epuck/usr/lib/argos3-epuck/.debug*. This folder will make the compilation fail. However, it suffices to delete this folder and compile again to solve the problem.
As the compilation of the image takes a long time (as specified in step 5 of Section E.1.1) it is recommended to compile argos and argos-epuck packages first and solve their respective error before compiling the complete image.

E.3.2 ARGoS controllers

The previous cross-compiling procedureA.2.1 can not be used anymore as it was designed for Ångström. However, BitBake can also be used to cross-compile ARGoS controllers for Poky.
The procedure is quite simple as it suffices to consider the controller as a package. A folder

containing the BitBake recipe and the source files should be added in *poky/meta-gumstix/argos-recipes* (or anywhere else). It is even possible to add directly a controller to the image by adding its name together with the other packages in *poky/meta-gumstix-extras/recipes-images/gumstix-console-image.bb*.
The only unusual point is that the CMake file must contain a **install** statement, otherwise the compilation will fail. While the reason of this error is not clear, it suffices to use a line similar to the one of the template to solve the problem.

E.3.3 Configuration

ARGoS should normally work without additional configuration steps except for a small change related to the change of kernel version. As mentioned in Section 4.1, Poky uses 3.X Linux kernels while Ångström was using 2.6.32. However, in 3.X kernels, the port interface */dev/ttyS0* has been renamed */dev/ttyO0*. As the ARGoS for e-puck plugin was initially developed to work with Ångström, the Linux board uses the old terminology (*/dev/ttyS0*) to communicate with the base.
Adding this simple line to */etc/profile* solves the problem :

ln -s /dev/ttyO0 /dev/ttyS0

E.4 Wi-fi setup

The Wi-fi will not work directly on the e-puck with the default installation of Yocto Project Poky. This Section describes hence the different steps to install Wi-fi drivers on the Poky distribution as well as the additional configuration required to make the wireless connection work properly.
Until it is properly configured, the communication with the robot should be done by cable (see Section A.1.1).

E.4.1 Installation

The base configuration of Yocto does not include the necessary driver and firmware for the Edimax Wi-fi dongle. Hence, some modifications need to be done before building the image :

- *poky/meta-gumstix-extras/conf/local.conf.sample* file :
  1. Uncomment (remove the '#') the following line :

     # PREFERRED_VERSION_linux-gumstix = "3.5.7"
     This is required as otherwise Yocto will build an image with the 3.18 version of the Linux kernel, and the OTG port that the Wi-fi dongle uses does not seem compatible with it. This is however not a big restriction as the Ångström distribution was using a 2.6 kernel, the 3.5 is then already an improvement.

2. Add the following line at the end of the file :

   IMAGE_INSTALL_append += "rtl8192cu"
   This is required to install the Wi-fi driver of the dongle. The official driver (rtl8192cu) does not work well and despite some efforts to make it work correctly, the results were not satisfying. A custom driver has then be built (base on Edimax) and works well

E.4.2 Configuration

Some files need to be configured in order to enable the Wi-fi :

1. Activate the Wi-fi dongle[2] :

   ```
   $ echo host > /sys/bus/platform/devices/musb-hdrc/mode
   $ echo -n 1 > /sys/bus/usb/devices/2-1/bConfigurationValue
   ```

   Those 2 lines should be added in ... in order to activate the dongle at boot time. The dongle is indeed connected to the OTG port that is not enabled by default.

2. Set up network related files :

   - Modify /etc/hosts

     This is optional but the file should look like :

     ```
     127.0.0.1   localhost.localdomain   localhost
     127.0.1.1   overo
     ```

   - Create the /etc/network/interfaces file :

     Here is what it should look like :

     ```
     auto lo
     iface lo inet loopback

     auto wlan0
     iface wlan0 inet static
     #address 10.0.1.49
     #netmask 255.255.255.0
     #network 10.0.1.0
     #gateway 10.0.1.1

     address 192.168.1.7
     netmask 255.255.255.0
     network 192.168.1.1
     gateway 192.168.1.1
     ```

[2] According to Overo Section of GCtronics website[40], Section 4.6.

```
wpa-conf /etc/wpa_supplicant/wpa_supplicant-wlan0.conf
up echo search ulb.ac.be >> /etc/resolv.conf
up echo domain ulb.ac.be >> /etc/resolv.conf
up echo nameserver 164.15.59.200 >> /etc/resolv.conf
up echo nameserver 164.15.59.201 >> /etc/resolv.conf
up echo dns-nameservers 8.8.8.8 8.8.4.4 >> /etc/resolv.conf
```

Here, two different setup for the IP address are proposed. The first one is to use with the epucks network (see Section A.1.2) and the second one is to use with a generic Wi-fi network. The selected configuration should be uncommented.

- Modify the /etc/wpa_supplicant/wpa_supplicant-wlan0.conf file

  The file should look like :

```
# Adjust this file for your network. See [1] for
# more details.
# [1] http://linux.die.net/man/5/wpa_supplicant.conf
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
#update_config=1
ap_scan=1

# Connect to a WPA2 protected network
network={
ssid="ssid"
proto=WPA2
key_mgmt=WPA-PSK
pairwise=CCMP TKIP
group=CCMP TKIP
scan_ssid=1
psk="password"
priority=10
}
```

  Of course, ssid and password should be replaced by the actual name and password of the network.

E.5 Full installation summary

This Section presents an overall summary of the installation procedure including all additional steps for installing ROS, ARGoS and the Wi-fi drivers :

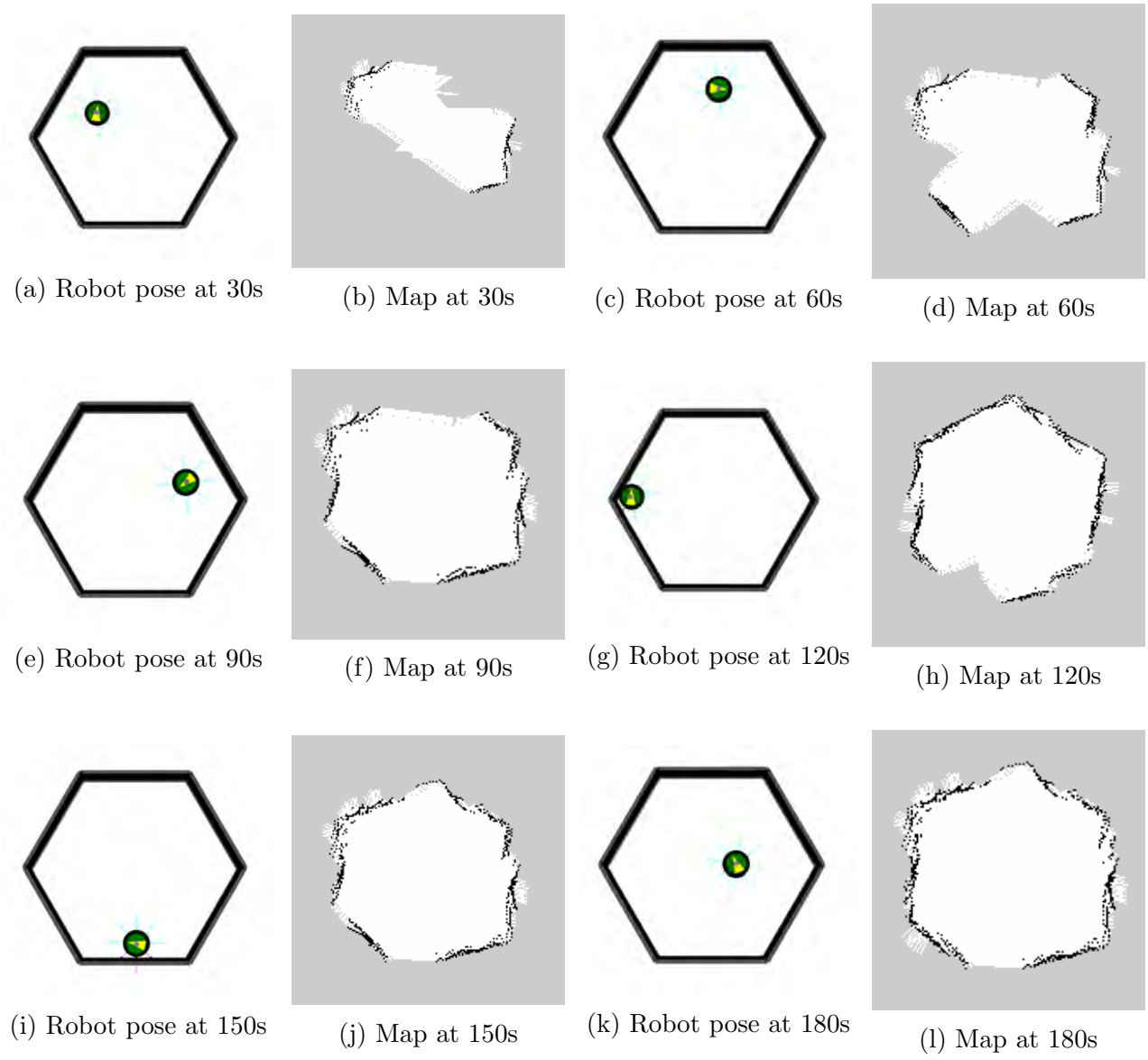
1. Perform steps 1 to 3 of Section E.1.1.
2. During step 4, do the modifications listed in Sections E.2.1 (ROS), E.3.1 (ARGoS) and E.4.1 (Wi-fi).
3. Perform steps 5 to 7 of Section E.1.1.
4. Once everything is installed, configure everything according to Sections E.2.2 (ROS), E.3.3 (ARGoS) and E.4.2 (Wi-fi).

Appendix F

Step by step mapping

Figures F.1 and F.2 illustrate the mapping process both in simulated and real scenarios. The demonstration has been conducted in the small and empty arena for the simulated mapping, and the real arena with five obstacles in the case of the real mapping.

F.1 Simulated e-puck



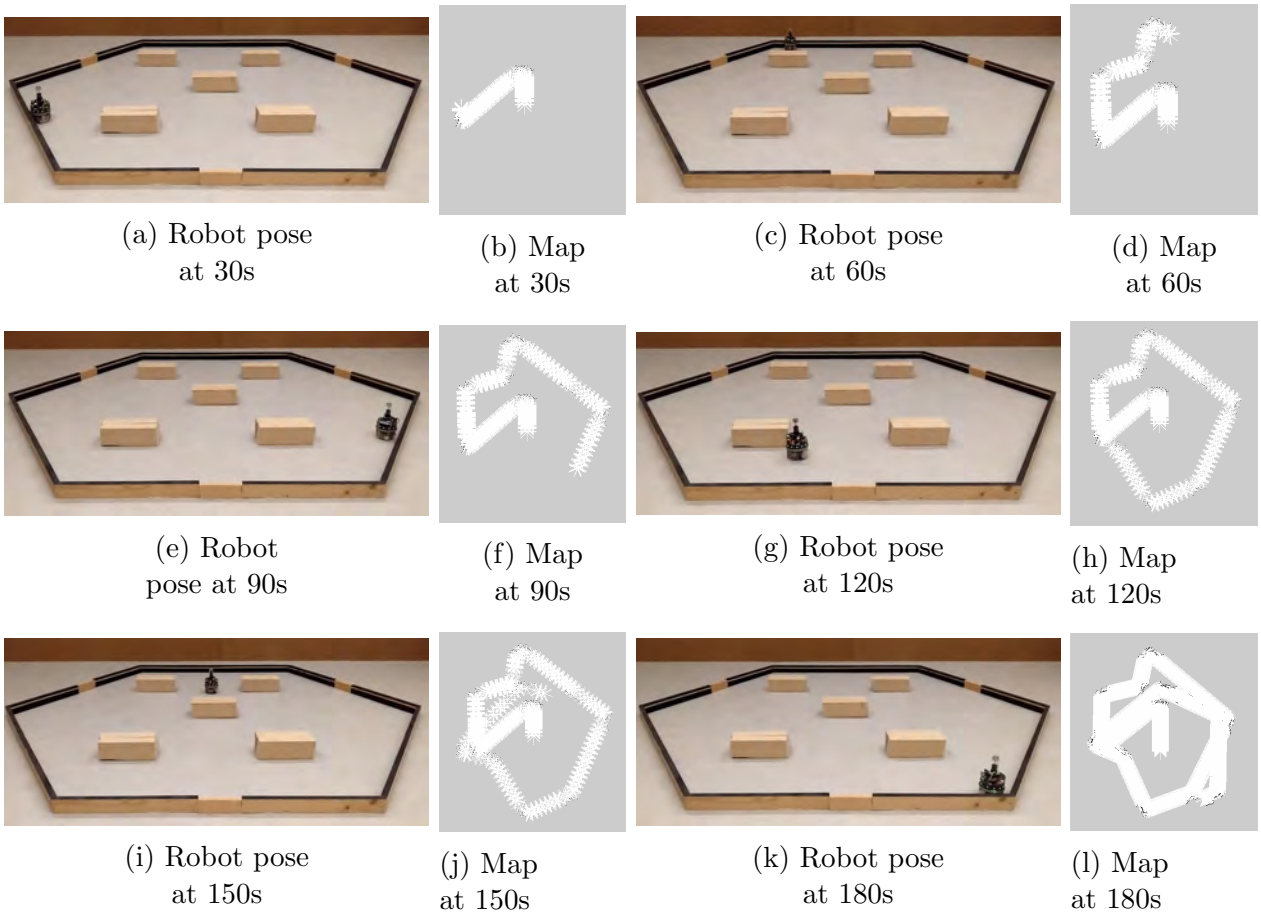
(a) Robot pose at 30s (b) Map at 30s (c) Robot pose at 60s (d) Map at 60s

(e) Robot pose at 90s (f) Map at 90s (g) Robot pose at 120s (h) Map at 120s

(i) Robot pose at 150s (j) Map at 150s (k) Robot pose at 180s (l) Map at 180s

Figure F.1 – Mapping a simulated scenario with ballistic motion and random rotation obstacle avoidance

F.2 Real e-puck



(a) Robot pose at 30s

(b) Map at 30s

(c) Robot pose at 60s

(d) Map at 60s

(e) Robot pose at 90s

(f) Map at 90s

(g) Robot pose at 120s

(h) Map at 120s

(i) Robot pose at 150s

(j) Map at 150s

(k) Robot pose at 180s

(l) Map at 180s

Figure F.2 – Mapping a real scenario with ballistic motion and fixed rotation obstacle avoidance

It is interesting to notice that *GMapping* improves the final quality of the map by splatting the sparse readings of the robot.

Bibliography

[1] James A. Hilder Cristian Fleseriu Leonard Newbrook Wei Li Liam J. McDaid Alan G. Millard, Russell Joyce and David M. Halliday. The pi-puck extension board: a raspberry pi interface for the e-puck robot platform. *York Robotics Laboratory*, 2017.

[2] James A. Hilder David M. Halliday Andy M. Tyrrell Jon Timmis Junxiu Liu Shvan Karim Jim Harkin Alan G. Millard, Anju P. Johnson and Liam J. McDaid. Self-repairing spiking neural network controller for an autonomous robot. *York Robotics Laboratory*, 2018.

[3] B.Delhaisse L.Garattoni G.Francesca A.Ligot, K.Hasselmann and M.Birattari. Automode, neat, and evostick: Implementations for the e-puck robot in argos3. *IRIDIA Technical Report*, 2018.

[4] Micael S. Couceiro André Araújo, David Portugal and Rui P. Rocha. Integrating arduino-based educational mobile robots in ros. *Intell Robot Syst*, 2015.

[5] Arduino. Edison. https://www.arduino.cc/en/ArduinoCertified/IntelEdison, last visited on May 10, 2018.

[6] Arduino. Galileo gen2. https://www.arduino.cc/en/ArduinoCertified/IntelGalileoGen2, last visited on May 10, 2018.

[7] ARM. Arm website. http://www.arm.com, last visited on May 10, 2018.

[8] Erno Salminen Arttu Leppakoski and Timo D. Hamalainen. Framework for industrial embedded system product development and management. *System on Chip (SoC), 2013 International Symposium on*, 2013.

[9] BeagleBoard. Beaglebone black. https://beagleboard.org/black, last edited an April 11, 2018.

[10] Cedric Isokeit Erik Maehle Benjamin Meyer, Kristian Ehlers. The development of the modular hard- and software architecture of the autonomous underwater vehicle monsun. *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*, 2014.

[11] Gonzalez-Jimenez Javier Blanco José-Luis and Fernández-Madrigal Juan-Antonio. An alternative to the mahalanobis distance for determining optimal correspondences in data association. *IEEE Transactions on Robotics (T-RO)*, 28(4), aug 2012.

[12] Michael Bonani, Valentin Longchamp, Stéphane Magnenat, Philippe Rétornaz, Daniel Burnier, Gilles Roulet, Florian Vaussard, Hannes Bleuler, and Francesco Mondada. The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4187–4193. IEEE, 2010.

[13] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots in cluttered environments. pages 572–577. IEEE Comput. Soc. Press, 1990.

[14] J. Borenstein and Y. Koren. The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, June 1991.

[15] Ferrante-E.-Birattari M. et al. Brambilla, M. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intell*, 7: 1, 2013.

[16] Buildroot. Buildroot website. https://buildroot.org/, last visited on May 10, 2018.

[17] M Cao, Q Meng, B Luo, and M Zeng. Experimental comparison of random search strategies for multi-robot based odour finding without wind information. *Austrian Contributions to Veterinary Epidemiology*, 8:43–50, 2015.

[18] Jianing Chen, Melvin Gauci, Michael J Price, and Roderich Groß. Segregation in swarms of e-puck robots based on the brazil nut effect. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 163–170. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[19] Carlos M. Costa, Héber M. Sobreira, Armando J. Sousa, and Germano M. Veiga. Robust 3/6 dof self-localization system with selective map update for mobile robot platforms. *Robotics and Autonomous Systems*, 76:113–140, February 2016.

[20] Micael S Couceiro, Carlos M Figueiredo, J Miguel A Luz, Nuno MF Ferreira, and Rui P Rocha. A low-cost educational platform for swarm robotics. *International Journal of Robots, Education & Art*, 2(1), 2012.

[21] Cyberbotics. Webots. https://www.cyberbotics.com/, last visited on May 10, 2018.

[22] Cyberbotics. Webots user guide : Gctronic' e-puck. https://cyberbotics.com/doc/guide/epuck, last visited on May 10, 2018.

[23] Itai Dabran and Tom Palny. A robotic mobile hot spot relay (mhsr) for disaster areas. *SYSTOR '16 Proceedings of the 9th ACM International on Systems and Storage Conference*, 2016.

[24] Cristina Dimidov, Giuseppe Oriolo, and Vito Trianni. Random Walks in Swarm Robotics: An Experiment with Kilobots. In Marco Dorigo, Mauro Birattari, Xiaodong Li, Manuel López-Ibáñez, Kazuhiro Ohkura, Carlo Pinciroli, and Thomas Stützle, editors, *Swarm Intelligence*, volume 9882, pages 185–196. Springer International Publishing, Cham, 2016.

[25] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2011.

[26] Edimax. Edimax wi-fi adapter. https://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/fr/wireless_adapters_n150/ew-7811un/, last visited on May 10, 2018.

[27] R. Emery, F. Rahbar, A. Marjovi, and A. Martinoli. Adaptive L #x00e9;vy Taxis for odor source localization in realistic environmental conditions. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3552–3559, May 2017.

[28] EPFL. E-puck website. http://www.e-puck.org/, last edited on February 23, 2018.

[29] Francesca G. et al. An experiment in automatic design of robot swarms. *Swarm Intelligence. ANTS 2014. Lecture Notes in Computer Science*, 8667, 2014.

[30] M. Dorigo et al. Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine*, 20(4):60–71, 2013.

[31] Joseph J Falke, Randal B Bass, Scott L Butler, Stephen A Chervitz, and Mark A Danielson. The two-component signaling pathway of bacterial chemotaxis: a molecular view of signal transduction by receptors, kinases, and adaptation enzymes. *Annual review of cell and developmental biology*, 13(1):457–512, 1997.

[32] Richard Phillips Feynman, Robert B. Leighton, and Matthew L. Sands. *The Feynman lectures on physics. volume 1: Mainly mechanics, radiation, and heat.* Basic Books, New York, NY, paberback first published edition, 2011. OCLC: 1039713911.

[33] Andrei George Florea and Cătălin Buiu. Modelling multi-robot interactions using a generic controller based on numerical p systems and ros. *Electronics, Computers and Artificial Intelligence (ECAI), 2017 9th International Conference on*, 2017.

[34] Brutschy A. Garattoni L. Miletitch R. Podevijn G. Reina A. Soleymani T. Salvaro M. Pinciroli C. Mascia F. Trianni V. Francesca G., Brambilla M. and Birattari M. Automode-chocolate: automatic design of control software for robot swarms. *Swarm Intelligence*, 2015.

[35] Tobias Bützer Franziska Ryser and Jeremia P. Held. Fully embedded myoelectric control for a wearable robotic hand orthosis. *Rehabilitation Robotics (ICORR), 2017 International Conference on*, 2017.

[36] Mario Garzón, João Valente, Juan Jesús Roldán, David Garzón-Ramos, Jorge de León, Antonio Barrientos, and Jaime del Cerro. *Using ROS in Multi-robot Systems: Experiences and Lessons Learned from Real-World Field Tests*, pages 449–483. Springer International Publishing, Cham, 2017.

[37] Gazebo. Gazebo website. http://gazebosim.org/, last visited on May 10, 2018.

[38] GCtronic. Gctronic shop. https://www.gctronic.com/shop.php, last visited on May 10, 2018.

[39] GCtronic. Gctronic website. https://www.gctronic.com/index.php, last visited on May 10, 2018.

[40] GCtronic. Overo extension. http://www.gctronic.com/doc/index.php/Overo_Extension, last edited on April 17, 2018.

[41] GCtronics. Ros driver for e-puck robot. https://github.com/gctronic/epuck_driver_cpp, last edited on December 21, 2016.

[42] Gábor Geréb and Fülöp Augusztinovicz. Source-selective noise monitoring (resono) pilot project in birmingham city. *INTER-NOISE and NOISE-CON Congress and Conference Proceedings*, 2016.

[43] GNU. Cross-compilation. http://www.gnu.org/software/automake/manual/html_node/Cross_002dCompilation.html, last visited on April 28, 2018.

[44] Javier Gonzalez-Jimenez, José-Luis Blanco, Cipriano Galindo, Antonio Ortiz-de Galisteo, Juan-Antonio Fernández-Madrigal, Francisco-Angel Moreno, and Jorge Martínez. Mobile robot localization based on ultra-wide-band ranging: A particle filter approach. *Robotics and Autonomous Systems*, 57(5):496–507, 2009.

[45] Gumstix. Bring up wifi on a yocto image. https://www.gumstix.com/support/faq/yocto-wifi/, last visited on May 24, 2018.

[46] Gumstix. Create bootable microsd card. https://www.gumstix.com/support/getting-started/create-bootable-microsd-card/, last visited on April 17, 2018.

[47] Gumstix. Getting started with gumstix. https://www.gumstix.com/support/getting-started/get-image/, last visited on May 18, 2018.

[48] Gumstix. Gumstix overo com. https://www.gumstix.com/support/hardware/overo/, last visited on April 22, 2018.

[49] Gumstix. meta-gumstix. https://github.com/gumstix/meta-gumstix, last edited on May 24, 2018.

[50] Gumstix. meta-gumstix-extras. https://github.com/gumstix/meta-gumstix-extras, last edited on May 26, 2018.

[51] Kiyoshi Irie, Masashi Sugiyama, and Masahiro Tomono. Dependence maximization localization: a novel approach to 2d street-map-based robot localization. *Advanced Robotics*, 30(22):1431–1445, November 2016.

[52] Javier Gonzalez Jose-Luis Blanco and Juan-Antonio Fernandez-Madrigal. Extending obstacle avoidance methods through multiple parameter-space transformations. *Autonomous Robots*, 2008.

[53] Kitware. Cmake website. https://cmake.org/, last visited on April 29, 2018.

[54] A. Brutschy C. Pinciroli M. Birattari L. Garattoni, G. Francesca. *Software Infrastructure for E-puck (and TAM)*. IRIDIA, Université Libre de Bruxelles, 2015.

[55] York Robotics Laboratory. Pi-puck extension board. https://www.york.ac.uk/robot-lab/pi-puck/, last visited on May 18, 2018.

[56] York Robotics Laboratory. Pi-puck github repository. https://github.com/yorkrobotlab/pi-puck, last edited on September 25, 2017.

[57] Adam YH Lee. Ros on gumstix. https://github.com/gumstix/yocto-manifest/wiki/ROS-on-Gumstix, last edited on July 30, 2015.

[58] Heon-Cheol Lee and Beom-Hee Lee. Enhanced-spectrum-based map merging for multi-robot systems. *Advanced Robotics*, pages 1–16, January 2013.

[59] Heon-Cheol Lee, Seung-Hwan Lee, Myoung Hwan Choi, and Beom-Hee Lee. Probabilistic map merging for multi-robot rbpf-slam with unknown initial poses. *Robotica*, 30(2):205–220, March 2012.

[60] W. Liu and A.F.T. Winfield. Open-hardware e-puck linux extension board for experimental swarm robotics research. *Microprocessors and Microsystems*, 35(1):60–67, 2011.

[61] M. Kholid Arrofi Adi Wibowo Petrus Mursanto Wisnu Jatmiko M. Anwar Ma'sum, Grafika Jati. Autonomous quadcopter swarm robots for object localization and tracking. *Micro-NanoMechatronics and Human Science (MHS), 2013 International Symposium on*, 2014.

[62] Z. Ahmad Hafizh M. Anwar Ma'sum Grafika Jati Wisnu Jatmiko Petrus Mursanto M. Sakti Alvissalim, Big Zaman. Swarm quadrotor robots for telecommunication network coverage area expansion in disaster area. *SICE Annual Conference (SICE), 2012 Proceedings of*, 2012.

[63] ROS maintain team. Beaglebone. http://wiki.ros.org/BeagleBone, last edited on December 19, 2014.

[64] ROS maintain team. Gumros. http://wiki.ros.org/gumros, last edited on February 15, 2012.

[65] ROS maintain team. Installation instructions for hydro medusa in Ångström. http://wiki.ros.org/hydro/Installation/Angstrom, last edited on October 8, 2014.

[66] ROS maintain team. Ros website. http://www.ros.org/, last visited on May 10, 2018.

[67] Leandro Soriano Marcolino and Luiz Chaimowicz. No robot left behind: Coordination to overcome local minima in swarm navigation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 1904–1909. IEEE, 2008.

[68] Johannes Feldmaier Martin Knopp, Can Aykın and Hao Shen. Formation control using gq(λ) reinforcement learning. *Robot and Human Interactive Communication (RO-MAN), 2017 26th IEEE International Symposium on*, 2017.

[69] Koën Kooi Merciadri Luca. Ångström manual. http://www.student.montefiore.ulg.ac.be/~merciadri/angstrom/files/angstrom-manual.pdf June, 2009.

[70] Patricia A.Vargas Micael S.Couceiro and Rui P.Rocha. Bridging the reality gap between the webots simulator and e-puck robots. *Robotics and Autonomous Systems*, 62(10):1549–1567, 2014.

[71] Christian Ahler Michael Rubenstein and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2012.

[72] Navid Khazaee Korghond Edwin Babaians Mojtaba Karimi, Alireza Ahmadi and Saeed Shiry Ghidary. Remoro; a mobile robot platform based on distributed i/o modules for research and education. *Robotics and Mechatronics (ICROM), 2015 3rd RSI International Conference on*, 2016.

[73] Bonani M. Raemy X. Pugh J. Cianci C. Klaptocz A. Magnenat S. Zufferey J.-C. Floreano D. Mondada, F. and A. Martinoli. The e-puck, a robot designed for education in engineering. proceedings of the 9th conference on autonomous robot systems and competitions, 2009.

[74] T.H. Nam, J.H. Shim, and Y.I. Cho. A 2.5d map-based mobile robot localization via cooperation of aerial and ground robots. *Sensors (Switzerland)*, 17(12), December 2017.

[75] S. G. Nurzaman, Y. Matsumoto, Y. Nakamura, S. Koizumi, and H. Ishiguro. Yuragi-based adaptive searching behavior in mobile robot: From bacterial chemotaxis to Levy walk. In *2008 IEEE International Conference on Robotics and Biomimetics*, pages 806–811, February 2009.

[76] OpenEmbedded. Openembedded website. http://www.openembedded.org/wiki/Main_Page, last edited on May 3, 2017.

[77] Zohar Pasternak, Frederic Bartumeus, and Frank W Grasso. Lévy-taxis: a novel search strategy for finding odor plumes in turbulent flow-dominated environments. *Journal of Physics A: Mathematical and Theoretical*, 42(43):434010, October 2009.

[78] Knopp Martin Meyer Dominik Phanuel Hieber, Feldmaier Johannes. Yocto project on the gumstix overo board. *Technische Universität München,*, 2015.

[79] Raspberry Pi. Os images. https://www.raspberrypi.org/downloads/, last visited on May 10, 2018.

[80] Carlo Pinciroli. Argos website. http://www.argos-sim.info/index.php, last edited on May 2, 2018.

[81] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[82] Yocto Project. Poky. https://www.yoctoproject.org/software-item/poky/, last visited on May 10, 2018.

[83] Yocto Project. Release feature table. https://wiki.yoctoproject.org/wiki/Release_Feature_Table, last edited on April 20, 2017.

[84] Yocto Project. Releases. https://wiki.yoctoproject.org/wiki/Releases, last edited on April 2, 2018.

[85] Yocto Project. Yocto project website. https://www.yoctoproject.org/, last visited on May 10, 2018.

[86] PUC-Rio. Lua website. https://www.lua.org/, last edited on October 17, 2017.

[87] Realtek. Realtek website. www.realtek.com, last visited on May 10, 2018.

[88] Chris Larson Richard Purdie and Phil Blundell. Bitbake user manual. https://www.yoctoproject.org/docs/1.6/bitbake-user-manual/bitbake-user-manual.html, 2014.

[89] Michael Rubenstein, Adrian Cabrera, Justin Werfel, Golnaz Habibi, James McLurkin, and Radhika Nagpal. Collective transport of complex objects by simple robots: theory and experiments. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 47–54. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[90] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[91] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[92] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 3375–3382 vol.4, April 1996.

[93] Thomas Stützle and Marco Dorigo. *New Ideas in Optimization*. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[94] Ying Tan and Zhong-yang Zheng. Research advance in swarm robotics. *Defence Technology*, 9(1):18–39, 2013.

[95] CARMEN Team. Carmen website. http://carmen.sourceforge.net/intro.html, last visited on May 18, 2018.

[96] MRPT team. Mrpt website. https://www.mrpt.org/, last visited on May 18, 2018.

[97] Yocto Project team. Gumstix repo manifests for the yocto project build system. https://github.com/gumstix/yocto-manifest, last edited on April 9, 2018.

[98] Alexander Tiderko. multimaster_fkie package. http://wiki.ros.org/multimaster_fkie, last edited on June 7, 2015.

[99] James Bowman Tim Field, Jeremy Leibs. rosbag package. http://wiki.ros.org/rosbag, last edited on June 16, 2015.

[100] Toradex. Toradex website. https://www.toradex.com/ last visited on May 10, 2018.

[101] Brian Gerkey Dirk Thomas Troy Straszheim, Morten Kjaergaard. Catkin. http://docs.ros.org/api/catkin/html/, last visited on May 10, 2018.

[102] Ali E Turgut, F Gokce, Hande Celikkanat, L Bayindir, and Erol Sahin. Kobot: A mobile robot designed specifically for swarm robotics research. *Middle East Technical University, Ankara, Turkey, METU-CENG-TR Tech. Rep*, 5(2007), 2007.

[103] Hamann H. Dorigo M Valentini G., Brambilla D. Collective perception of environmental features in a robot swarm. *Swarm Intelligence. ANTS 2016. Lecture Notes in Computer Science*, 9882, 2016.

[104] Andrew Vardy. Argos bridge. https://github.com/BOTSlab/argos_bridge, last edited on June 20, 2017.

[105] P. Varet. Rtl8192cu-fixes. https://github.com/pvaret/rtl8192cu-fixes, last edited on April 18, 2018.

[106] Carlo Pinciroli Vito Trianni. Argos-kilobot. https://github.com/ilpincy/argos3-kilobot, last edited on May 6, 2018.

[107] Linux wiki. Rpi usb wi-fi adapters. https://elinux.org/RPi_USB_Wi-Fi_Adapters, last edited on August 23, 2017.

[108] Jordan Wood. Minimum bounding rectangle. In *Encyclopedia of GIS*, pages 660–661. Springer, 2008.

[109] Peiliang Wu, Lingfu Kong, and Shengnan Gao. Holography map for home robot: an object-oriented approach. *Intelligent Service Robotics*, 5(3):147–157, July 2012.

[110] Bin Yang, Yongsheng Ding, Yaochu Jin, and Kuangrong Hao. Self-organized swarm robot for target search and trapping inspired by bacterial chemotaxis. *Robotics and Autonomous Systems*, 72:83–92, October 2015.

[111] V. Zaburdaev, S. Denisov, and J. Klafter. L\'evy walks. *Reviews of Modern Physics*, 87(2):483–530, June 2015.

[112] L. Zhang, P.Y. Shen, J. Song, L.B. Dong, Y.Z. Zhang, X.X. Zhang, and J.Q. Zhang. A distributed multi-robot map fusion algorithm. volume 536-537, pages 917–924. Trans Tech Publications, 2014.

[113] Ångström. meta-angstrom. https://github.com/Angstrom-distribution/meta-angstrom last edited on February 16, 2018.

[114] Ångström. The Ångström distribution. https://github.com/Angstrom-distribution last visited on May 25, 2018.

[115] Ångström. Ångström website. http://wp.angstrom-distribution.org/ last edited on September 29, 2015.