Creative Adaptation through Learning

Antoine Cully


Thèse de Doctorat
de l'université Pierre et Marie Curie

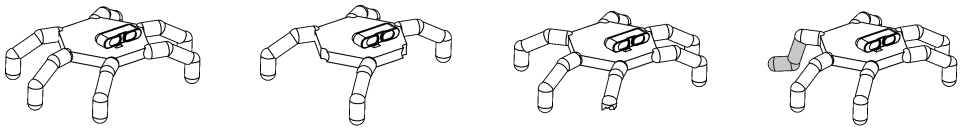
Spécialité : **Informatique (EDITE)**

Présentée par : **M. Antoine Cully**

Soutenue le : **21 Décembre 2015**

Pour obtenir le grade de
Docteur de l'Université Pierre et Marie Curie

Creative Adaptation through Learning



Rapporteurs	Marc Schoenauer	-	INRIA Saclay - Île-de-France
	David Filliat	-	ENSTA - ParisTech
Examinateurs	Raja Chatila	-	Univ. Pierre et Marie Curie - ISIR
	Jan Peters	-	Technische Universität Darmstadt
	Jonas Buchli	-	ETH Zurich
Invitée	Eva Cruck	-	Direction Générale de l'Armement Agence Nationale de la Recherche
Directeur	Stéphane Doncieux	-	Univ. Pierre et Marie Curie - ISIR
Co-encadrant	Jean-Baptiste Mouret	-	Univ. Pierre et Marie Curie - ISIR INRIA Nancy - Grand Est




French Abstract

Les robots ont profondément transformé l'industrie manufacturière et sont susceptibles de délivrer de grands bénéfices pour la société, par exemple en intervenant sur des lieux de catastrophes naturelles, lors de secours à la personne ou dans le cadre de la santé et des transports. Ce sont aussi des outils précieux pour la recherche scientifique, comme pour l'exploration des planètes ou des fonds marins. L'un des obstacles majeurs à leur utilisation en dehors des environnements parfaitement contrôlés des usines ou des laboratoires, est leur fragilité. Alors que les animaux peuvent rapidement s'adapter à des blessures, les robots actuels ont des difficultés à faire preuve de créativité lorsqu'ils doivent surmonter un problème inattendu: ils sont limités aux capteurs qu'ils embarquent et ne peuvent diagnostiquer que les situations qui ont été anticipées par leur concepteurs.

Dans cette thèse, nous proposons une approche différente qui consiste à laisser le robot apprendre de lui-même un comportement palliant la panne. Cependant, les méthodes actuelles d'apprentissage sont lentes même lorsque l'espace de recherche est petit et contraint. Pour surmonter cette limitation et permettre une adaptation rapide et créative, nous combinons la créativité des algorithmes évolutionnistes avec la rapidité des algorithmes de recherche de politique (policy search).

Notre première contribution montre comment les algorithmes évolutionnistes peuvent être utilisés afin de trouver, non pas une solution, mais un large ensemble de solutions à la fois performantes et diverses. Nous appelons ces ensembles de solutions des *répertoires comportementaux* (behavioral repertoires). En découvrant de manière autonome ces répertoires comportementaux, notre robot hexapode a été capable d'apprendre à marcher dans toutes les directions et de trouver des milliers de façons différentes de marcher. A notre sens, ces répertoires comportementaux capturent la créativité des algorithmes évolutionnistes.

La seconde contribution que nous présentons dans ce manuscrit combine ces répertoires comportementaux avec de l'optimisation Bayesienne. Le répertoire comportemental guide l'algorithme d'optimisation afin de permettre au robot endommagé de réaliser des tests intelligents pour trouver rapidement un comportement de compensation qui fonctionne en dépit de la panne. Nos expériences démontrent que le robot est capable de s'adapter à la situation moins de deux minutes malgré l'utilisation d'un grand espace de recherche et l'absence de plan de secours préétablis. L'algorithme a été testé sur un robot hexapode endommagé de 5 manières différentes, comprenant des pattes cassées, déconnectées ou arrachées, ainsi que sur un bras robotisé avec des articulations endommagées de 14 façons différentes.

Pour finir, notre dernière contribution étend cet algorithme d'adaptation afin de pallier trois difficultés qui sont couramment rencontrées en robotique: (1) permettre de transférer les connaissances acquises sur une tâche afin d'apprendre plus rapidement à réaliser les tâches suivantes, (2) être robuste aux solutions qui ne peuvent pas être évaluées sur le robot (pour des raisons de sécurité par exemple), et

qui sont susceptibles de pénaliser les performances d'apprentissage et (3) adapter les informations reçues a priori par le robot, qui peuvent être trompeuses, afin de maximiser leur utilité. Avec ces nouvelles propriétés, notre algorithme d'adaptation a permis à un bras robotisé endommagé d'atteindre successivement 20 cibles en moins de 10 minutes en utilisant uniquement les images provenant d'une caméra placée à une position arbitraire et inconnue du robot.

A travers l'ensemble de cette thèse, nous avons mis un point d'honneur à concevoir des algorithmes qui fonctionnent non seulement en simulation, mais aussi en réalité. C'est pourquoi l'ensemble des contributions présentées dans ce manuscrit ont été testées sur au moins un robot physique, ce qui représente l'un des plus grands défis de cette thèse. D'une manière générale, ces travaux visent à apporter les fondations algorithmiques permettant aux robots physiques d'être plus robustes, performants et autonomes.

English Abstract

Robots have transformed many industries, most notably manufacturing, and have the power to deliver tremendous benefits to society, for example in search and rescue, disaster response, health care, and transportation. They are also invaluable tools for scientific exploration of distant planets or deep oceans. A major obstacle to their widespread adoption in more complex environments and outside of factories is their fragility. While animals can quickly adapt to injuries, current robots cannot "think outside the box" to find a compensatory behavior when they are damaged: they are limited to their pre-specified self-sensing abilities, which can diagnose only anticipated failure modes and strongly increase the overall complexity of the robot.

In this thesis, we propose a different approach that considers having robots *learn* appropriate behaviors in response to damage. However, current learning techniques are slow even with small, constrained search spaces. To allow fast and creative adaptation, we combine the creativity of evolutionary algorithms with the learning speed of policy search algorithms.

In our first contribution, we show how evolutionary algorithms can be used to find, not only one solution, but a set of both high-performing and diverse solutions. We call these sets of solutions *behavioral repertoires* and we used them to allow a legged robot to learn to walk in every direction and to find several thousands ways to walk. In a sense, these repertoires capture a portion of the creativity of evolutionary algorithms.

In our second contribution, we designed an algorithm that combines these behavioral repertoires with Bayesian Optimization, a policy search algorithm. The repertoires guide the learning algorithm to allow damaged robots to conduct intelligent experiments to rapidly discover a compensatory behavior that works in spite of the damage. Experiments reveal successful adaptation in less than two minutes in large search spaces and without requiring self-diagnosis or pre-specified contingency plans. The algorithm has been tested on a legged robot injured in five different ways, including damaged, broken, and missing legs, and on a robotic arm with joints broken in 14 different ways.

Finally, in our last contribution, we extended this algorithm to address three common issues in robotics: (1) transferring knowledge from one task to faster learn the following ones, (2) dealing with solutions that cannot be evaluated on the robot, which may hurt learning algorithms and (3) adapting prior information that may be misleading, in order to maximize their potential utility. All these additional features allow our damaged robotic arm to reach in less than 10 minutes 20 targets by using images provided by a camera placed at an arbitrary and unknown location.

Throughout this thesis, we made a point of designing algorithms that work not only in simulation but also in reality. Therefore, all the contributions presented in this manuscript have been evaluated on at least one physical robot, which represents one of the biggest challenges addressed in this thesis. Globally, this work aims to provide the algorithmic foundations that will allow physical robots to be more robust, effective and autonomous.

Acknowledgement

A PhD thesis is quite an adventure and even if there is a main character, it is, in fact, teamwork. With these few lines, I would like to warmly thank all my teammates for their help and their support that allowed me to enjoy this intense journey.

Like in every journey, some moments have been more difficult than others. Fortunately, my dear Coralie was always present to cheer me on during time of doubt. She has also been extremely patient during the long evenings and nights that I spent to finish a paper, an experiment or simply to find a bug in my code. She has been my highest support during all these 3 years and the manuscript that you are currently reading would not have been the same without her.

It goes without saying that this thesis would also not have been the same without my supervisor Jean-Baptiste Mouret. He taught me everything I know in computer science and in scientific research in general. He believed in me and gave me enough freedom to explore my own ideas, while keeping a close eye on my progresses to prevent me to go on bad paths. He made me face challenges that I never expected to be able to accomplish. The current young scientist that I am today is undoubtedly the fruit of his supervision and I am proud of being one of his students.

The quality of supervision that I had during my thesis is also highly due to Stéphane Doncieux, who gave us the means and freedom to investigate our ideas within a fantastic working environment that we name AMAC team. His pieces of advice have always been extremely valuable and I really enjoyed our fruitful discussions about dreams in robotics.

Within the AMAC team and the ISIR laboratory in general, I had the luck to be in an excellent working environment that both stimulated and encouraged me. In addition to the environment, I had outstanding colleagues. The researchers of the team have always been here to share their own experience, to ask questions or to give pieces of advice. They are the heart of this big family that allows young PhD-students to grow up and I wish to be able, at one point, to help, support, and to advise young students as they did with me.

In particular, I would like to thank the marvelous members of the "famous J01" and affiliates (the list is too large, but they will recognise themselves). Several generations of PhD-students passed, but always with a lot of laughs, jokes and pranks. In addition to be my colleagues, most of them are today my friends.

The only regret that I have about this thesis is that, at the end, I have to leave this exceptional team.

A good preparation is required before every adventure, and for my preparation before the PhD-Thesis, I would like to thank all my teachers from the Polytech'Paris-UPMC engineer school (who are, for most of them, also my colleagues in the ISIR lab). In addition to teach me their knowledge, they also exchange with me their passion about robotics and the profession of researcher.

Even if they do not fully understood what I was doing, I would like to thank my family, who has always encouraged me to follow my desires and gave me the means to accomplish them.

I would like to thank the members of the jury of my PhD defense for accepting to be part of the jury. Their final judgement will validate the fruit of three intense and fascinating years of work and symbolize the end of this fabulous adventure.

Finally, I would like also to thank the DGA and the University Pierre and Marie Curie, who gave me the opportunity to do this fantastic journey.

Thank you for this incredible journey!

Contents

Chapter 1

Introduction

In 1950, Alan Turing proposed to address the question "can machines think?" via an "imitation game", which aims to assess machines' ability to exhibit behaviors that are indistinguishable from those of a human (Turing, 1950). In practice, this game involves a human (the evaluator) that has to distinguish between a human and a machine by dialoguing with them via a keyboard and a screen interface. The objective of Alan Turing was to substitute the initial question "can machines think?" by "Can a machine communicate in a way that makes it indistinguishable from a human". This question and this reasoning had, and still has, a strong impact in artificial intelligence (Warwick and Shah, 2015; Russell et al., 2010). However, a robot is not only a mind in a computer, it is an agent "that has a physical body" (Pfeifer and Bongard, 2007) and "that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors" (Russell et al., 2010). In order to transpose Turing's approach into robotics, his question could be considered from a more general point of view: "Can a robot *act* in a way that makes it indistinguishable from a human?". Without directly considering humans, we can wonder: can robots act more like animals than like current machines (Brooks, 1990)? But, what is acting like an animal? Put differently, what distinguishes a robot from an animal? In this manuscript, we will consider that one of the most striking differences between animals and robots is the impressive adaptation abilities of animals (Meyer, 1996); as a result, we will propose new algorithms to reduce this difference.

After more than 50 years of research in robotics, several breakthroughs have been achieved. For example, a surgeon located in New York operated on a patient in France with a transatlantic robot-assisted telesurgery (Marescaux et al., 2001). The recent advances in 3D printing allowed researchers to automatically design and manufacture robotic lifeforms from their structure to their neural network with an evolutionary algorithm (Lipson and Pollack, 2000). The robots from Boston Dynamics, like the Bigdog (which is robust to uneven terrains, Raibert et al. (2008)), the Cheetah (which runs at 29 mph (46 km/h)) and Atlas (which is an agile anthropomorphic robot) or the Asimo robot from Honda are good examples of very sophisticated systems that show impressive abilities. These noticeable advances suggest that robots are promising tools that can provide large benefits to the society. For example, we can hope that robots will one day be able to substitute humans in the most dangerous tasks they have to perform, like intervening in nuclear plants after a disaster (Nagatani et al., 2013) or in search and rescue missions after earthquakes (Murphy, 2004; Murphy et al., 2008).

While a few robots start to leave the well-controlled environments of factories to get in our homes, we can observe that most of them are relatively simple robots. More advanced robots, that is, more versatile robots, have the power to deliver tremendous benefits to society. Nonetheless, a slight increase in versatility typically results in a large increase in the robots' complexity, which, in turn, makes such versatile robots much more expensive and much more damage-prone than their simpler counterparts (Sanderson, 2010; Carlson and Murphy, 2005). This "exponential complexity" is a major obstacle to the widespread adoption of robots in complex environments and for complex missions.

In more concrete terms, the success of robotic vacuum cleaners and lawn mowers mainly stems from the simplicity of their task, which is reflected in the simplicity of the robots: they can perform only two types of action (moving forward/backward or turning); they perceive only one thing (the presence of obstacles); and they react with predefined behaviors to the encountered situations (Brooks et al., 1986; Brooks, 1990). With more complex robots, the situation is more complicated because they involve more complex behaviors and require a deeper understanding of the context. For example, a robot in a search and rescue mission (Murphy et al., 2008) has to deal with uneven terrains and to locate injured people. In order to be effective, it will have to move obstacles, to repatriate victims and to cooperate with other robots. With all these tasks, these robots may face an almost infinite number of situations, like different types of terrains, different types of disasters or different types of injured people. In other words, while it is possible to foresee most of the situations a simple robot (like a vacuum cleaner) may face, it is unfeasible to predefine how complex robots have to react in front of the quasi infinite set of situations they may encounter, unless having a team of engineers behind each robot[1].

In addition, the high complexity of versatile robots is bound to increase their fragility: each additional actuator or sensor is potentially a different way for a robot to become damaged. For example, many robots sent in search and rescue missions have been damaged and unable to complete their mission. In 2005, two robots were sent to search for victims after a mudslide near Los Angeles (La Conchita). Unfortunately, they became unable to continue their missions after only two and four minutes because of a root in the wet soil and a thick carpet in a house (Murphy et al., 2008). Similarly, after a mine explosion in West Virginia (Sago in 2006): the robot sent to find the victims became stuck on the mine rails after only 700 meters (Murphy et al., 2008). Five years later, the same kind of scenario repeated several times after the nuclear accident at the Fukushima nuclear power plants that happened in 2011. For example, one robot has been lost in 2011 because its communication cable got snagged by the piping of the plant (Nagatani et al., 2013) and a second robot has been lost in 2015 after a fallen object blocked its path. In all these examples, the robots have been abandoned and the missions aborted, which

[1]For example, teams of engineers are involved in the control-loops (DeDonato et al., 2015) of robots that participate to the DARPA Robotics Challenge (Pratt and Manzo, 2013) in order to allow them to react to the situations they have to face.

can lead to dramatic consequences when lives are at stake.

The question of fault tolerance is a classic topic in engineering, as it is a common problem in aviation and aerospace systems. Current damage recovery in deployed systems typically involves two phases: (1) performing self-diagnosis thanks to the embedded sensors and then (2) plan or select the best contingency plan according to the diagnosis (Kluger and Lovell, 2006; Visinsky et al., 1994; Koren and Krishna, 2007). The side effect of this kind of approach is that it requires anticipating all the damage conditions that may occur during the robot's mission. Indeed, each type of damage that should be diagnosed requires having the proper sensor at the right place, similarly to a doctor who uses different tools to diagnose different diseases, electrocardiogram to detect heart diseases and MRI for cancer cells. As a consequence, the robot's abilities to diagnose a damage condition directly depend on its embedded sensors. As the robots and their missions increase in complexity, it becomes almost unfeasible to foresee all the damage conditions that may occur and to design the corresponding diagnostic modules and contingency plans. This is the beginning of a vicious circle that prevents us from keeping the costs reasonable, as each of these modules can be damaged too and may thus require additional sensors.

Animals, and probably humans, respond differently when they face an unexpected situation or when they are injured (Jarvis et al., 2013; Fuchs et al., 2014): they discover by trial and error a new behavior. For example, when a child has a sprain ankle, he is able to discover, without the diagnosis of a doctor, that limping minimizes pain and allows him to go back home.

The objective of the algorithmic foundations presented in this manuscript is to allow robots to do the same: making robots able to learn on their own how to deal with the different situations they may encounter. With improved learning abilities, robots will become able to autonomously discover new behaviors in order to achieve their mission or to cope with a damage. The overall goal of this promising approach is to help develop more robust, effective, autonomous robots by removing the requirement for engineers to anticipate all the situations that robots may encounter. It could, for example, enable the creation of robots that can help rescuers without requiring their continuous attention, or making easier the creation of personal robotic assistants that can continue to be helpful even when a part is broken. Throughout this manuscript, we will mainly consider the challenge of damage recovery, though adapting to other unforeseen situations that do not involve damage would be a natural extension of this work.

In order to effectively discover new behaviors, learning processes have to be both *fast* and *creative*. Being able to learn quickly (in terms of both time and number of evaluations) new behaviors is critical in many situations. The first reason is that most autonomous robots run on batteries, and as a consequence, it is likely that a robot that spends too much time learning new behaviors will run out of battery before achieving its mission. The second reason is that many situations require reacting quickly, typically after a disaster. Therefore, we cannot have robots wasting hours trying to learn new behaviors when people's lives are in danger. Creativity, which " involves the production of novel, useful [solutions]" (Mumford,

2003), is an important property for learning algorithms, as it influences the number of situations in which systems are able to find appropriate solutions. This property requires algorithms to explore large search spaces to discover behaviors that are different from what they already know (Lehman and Stanley, 2011a). Transposed to robotics, the creativity of the algorithm determines the number of unforeseen situations that the robot can handle. For example, riding a bicycle is different from walking or limping. To allow a robot to adapt to a large variety of situations, a learning algorithm has to be able to find all these different types of behaviors, regardless the initial or current behavior of the robot.

Unfortunately, speed and creativity are most of the time antagonistic properties of learning algorithms because exploring a large search space, that is, searching for creative solutions, requires a large amount of time. In the next chapter (chapter 2), we review the different families of learning algorithms and compare their creativity and learning speed. We focus our review on evolutionary algorithms (Eiben and Smith, 2003) and Policy Search methods (Kober et al., 2013) because they are the main families of algorithms used to learn low-level behaviors (motor skills) in robotics.

In chapter 3, we present how the creativity of evolutionary algorithms can be employed to discover large collections of behaviors. For example, we show how a legged robot can autonomously discover several hundred behaviors that allow it to walk in every direction. In other experiments, the same robot also discovered several thousand ways of walking in a straight line. We call these large collections of behaviors *Behavioral Repertoires*.

In the chapter 4, we apply learning algorithms for on-line damage recovery. One of the main challenges is to reduce the required time to adapt while keeping the search space as open as possible to be able to deal with a large variety of situations. In the first part of this chapter, we show how combining simulations and physical experiments allows the algorithm to transfer the majority of the search space exploration in simulation. In the second part of this chapter, we highlight how behavioral repertoires, introduced in the previous chapter, can be combined with a policy search algorithm to guide the exploration process. With these two principles, we propose an algorithm that couples the creativity of evolutionary algorithm and the speed of policy search algorithms to allow damaged robots to conduct intelligent experiments to rapidly discover compensatory behaviors that work in spite of the damage situations. For example, our experiments show a hexapod robot and a robotic arm that recover from many damage conditions in less than 2 minutes, that is, much faster than the state of the art.

The last chapter (chapter 5) extends our damage recovery algorithm to deal with 3 issues that frequently impact robotic experiments: (1) transferring knowledge from one task to learn the following ones faster, (2) dealing with solutions that cannot be evaluated on the robot, which may hurt learning algorithms and (3) adapting prior information that may be misleading, in order to maximize their potential utility. With these extensions, our algorithm aims to become a generic framework that can be used with a large variety of robots to learn and adapt in

numerous situations.

Before the conclusion of this manuscript, we discuss the current limitations of our methods and the different approaches that we plan to investigate in order to circumvent them. In this last chapter (chapter 6), we also highlight the links that may exist between our methods and observations made in neuroscience.

Chapter 2

Background

Contents

2.1 Introduction

Learning algorithms aim to provide systems with the ability to acquire knowledge and skills to solve a task without being explicitly programmed with a solution. For example, Mitchell (1997) proposed this definition: "*The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience.*" Learning algorithms are traditionally divided into three categories of algorithms depending on the quantity of knowledge about the solutions that is provided to the system (Haykin, 1998; Russell et al., 2010): (1) In supervised learning, a "teacher" provides to the system a error signal that reflects the difference between the output value and the expected value. These algorithms

are typically applied on classification problems in which the system has a database of inputs examples and corresponding outputs that the system should reproduce. Based on these examples the goal of the algorithm is to learn a generalization of these examples to deal with unknown examples. Conversely, (2) in unsupervised learning (Hastie et al., 2009), there is no "teacher" and the system has no information about the consequences (or outputs) of its choices. Consequently, the system has to find hidden structures in the obtained data in order to distinguish them. Clustering algorithms (Xu et al., 2005) are a good example of unsupervised learning algorithms. They use the spatial distribution of the data to infer clusters that are likely to correspond to different classes. (3) Reinforcement Learning algorithms (Kober et al., 2013) are in a way situated in between these two families of algorithms. While the correct inputs/outputs pairs are never presented, the "teacher" provides a qualitative feedback after the system has performed a sequence of actions. Most of the time, this feedback is less informative than in supervised learning algorithms. It can be for example a reward value, which does not indicate the actual solution but rather states if the performed action is better or not than the previous ones.

More recently, several other families of learning algorithms have emerged and do not fit well in these three categories. For example, intrinsically motivated learning (Oudeyer et al., 2007; Baldassarre and Mirolli, 2013; Delarboulas et al., 2010), in which the system defines on his own the value of its actions, can be considered as both an unsupervised learning algorithm because there is no teacher, and as a reinforcement learning algorithm because the system tries to maximize the feedback provided by its internal motivation modules. Another example of a recent approach that does not fit in these categories is Transfer learning (Pan and Yang, 2010; Thrun, 1996). This approach consists in transferring knowledge acquired on previous tasks to new ones. It consequently shares similarities with both supervised and unsupervised learning algorithms, as the former task is most of the time learned thanks to a "teacher", while the links with the following tasks have to be autonomously inferred by the system. Imitation learning (Billard et al., 2008; Schaal, 1999) is another family of algorithms that can be situated on the borderlines of the traditional classifications. In this case, the system aims to reproduce a movement showed by a "teacher", for example via kinesthetic teaching. This technique shares similarities with supervised learning algorithms, as there is a "teacher" showing the solution. However, the link between the recorded data and the corresponding motor commands is unknown and the robot has to learn to reproduce the movement by minimizing a cost value, like in reinforcement learning, which corresponds to the differences between the taught movement and the robot's behavior.

Learning a behavior in an unforeseen situation, like after mechanical damage, is typically a reinforcement learning problem, because the robot has to get back its initial performance and no "teacher" can provide examples of good solutions. The robot has to learn on its own how to behave in order to maximize its performance by using its remaining abilities.

The definition of reinforcement learning (RL) from Sutton and Barto (1998a) states that: "Reinforcement learning is learning what to do so as to maximize a

numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them." In other words, a RL algorithms aims to solve a task formulated as a cost function that has to be minimized or as a reward (also called quality or performance) function that has to be maximized. Based on this function (for instance named f), we can define a reinforcement learning algorithm as an algorithm that solves this equation (considering f as a reward function):

$$\mathbf{x}^* = \arg\max_{\mathbf{x}} f(\mathbf{x}) \tag{2.1}$$

The goal of the algorithm is thus to find the politic or behavior x^* that maximizes the reward function (Sutton and Barto, 1998a; Russell et al., 2010).

We can see here that a RL problem can be formulated as a standard optimization problem. However, when learning a behavior, the optimized function as well as its properties and gradient are typically unknown. A typical example is a legged robot learning to walk: it has to learn how to move its legs in order to keep balance and to move. In this context, the reward function can be for example the walking speed and the solutions can be parameters that define leg trajectories or synaptic weights of neural networks that control the motors. The function that maps these parameters to the reward value is unknown. Consequently, in order to acquire information about this function, the system has to try potential solution and to record its performance. This type of function is called black-box functions and makes impossible to use standard optimization algorithms, like linear or quadratic programming (Dantzig, 1998; Papadimitriou and Steiglitz, 1998) or the Newton's optimization method (Avriel, 2003), which require the problem to be linear/quadratic or to have access to the derivative of the function.

Reinforcement learning algorithms have been designed to deal with this kind of function and gather several sub-families of techniques, like Policy Search algorithms (Kober et al., 2013; Deisenroth et al., 2013b) or Differential Dynamic Programming (Bertsekas et al., 1995; Sutton et al., 1992). However, the name *reinforcement learning* refers most of the time to the first family of RL algorithms that have been proposed, making the distinction between the general RL family and these traditional algorithms sometimes difficult and confusing. These algorithms aim to associate to every state of the robot to a reward (it corresponds to the "value function") and then use this information to determine the optimal policy to maximize the reward. This family of algorithms is for example composed by the Q-learning algorithm (Watkins, 1989; Watkins and Dayan, 1992), the State-Action-Reward-State-Action algorithm (SARSA, Rummery and Niranjan (1994)) or the Time Difference Learning algorithm (TD-Learning, Sutton and Barto (1998b)). However, the vast majority of these algorithms are designed to work with small state spaces (for example 2D locations in maze experiments, Thrun (1992)) and with discretized state and action spaces (for example discrete actions can be: move up, move down, move right, move left, etc.).

While these traditional reinforcement algorithms are beneficial to learn discrete high-level tasks (like in path-planning problem (Singh et al., 1994), to learn to

play video games (Mnih et al., 2015)); learning low-level behaviors or motor skills in robotics (like walking gaits or reaching movements) involves most of the time using continuous and high dimensional state and action spaces. These aspects contrast with the constraints of traditional RL algorithms and this is why most of applications of RL in robotics use *Policy Search* algorithms (PS) (Kober et al., 2013), which are known to have better scalability and to work in continuous domains (Sutton et al., 1999; Deisenroth et al., 2013b).

Another family of algorithms widely used to learn behaviors in robotics are the evolutionary algorithms (EAs, Eiben and Smith (2003)). These algorithms are most of the time considered apart from the general family of reinforcement learning algorithms, because they come from different concepts and ideas. However, we can underline that EAs fit perfectly into the definition of RL algorithms presented above. They are used to optimize a fitness function, which is similar to a cost or a reward function, and they test different potential solutions (gathered in a population) to gather information about this unknown function.

In this chapter, we will present in detail both *Evolutionary algorithms* and *Policy Search algorithms*, as they are widely used to learn low-level behaviors in robotics. In the last part of this chapter we provide a concise tutorial on *Bayesian Optimization*, a model-based policy search algorithm that is considered as the state-of-the-art in learning behaviors in robotics. The objective of this overview is to get a global picture of the advantages and the limitations of each of these algorithms, which will be useful to design a learning algorithm that will allow robots to adapt to unforeseen situations. We will in particular focus our attention on their ability to find creative solutions and the number of trials on the robot required to find these solutions. The creativity of an algorithm is related to its ability to perform a global optimization and to the size of the search space they can deal with. Such property is important as a behavior that works in spite of a damage may be substantially different from the robot's initial behaviors. For example, a six-legged robot with a damaged leg will walk differently with its 5 remaining legs than when it was intact. The learning speed of the algorithm is also a very important aspect of learning algorithms as it directly impacts the autonomy of the robot. A robot requiring to test hundreds of behaviors to find a solution is likely to run out of battery or to aggravate its condition.

2.2 Evolutionary algorithms

2.2.1 Principle

Evolutionary Algorithms (EAs) take inspiration from the *synthetic theory of evolution*. This theory, which emerged in the 30's, combines the Darwinian theory of evolution (Darwin, 1859), the Mendelian theory of heredity (Mendel, 1865) and the theory of population genetics (Haldane, 1932). It details the mechanisms that drive the diversity of individuals among a population and those that allow some traits to be transferred from one individual to another though heredity. The synthetic the-

ory of evolution mainly relies on the idea that the DNA encodes the characteristics of the individual and that each DNA sequence is different. During the reproduction process, several sequences are combined and mutated in order to create new individual, which promote the diversity of individuals in the population.

Evolutionary computation algorithms tend to employ an abstract representation of this theory in order to use the adaptation abilities of natural evolution to a large set of problems. This abstraction stands on 4 principles:

- Diversity of individual: Each individual, even from the same specie, is different.
- Struggle for life: Every individual cannot survive because the natural resources are limited. Consequently, the individuals have to compete to survive.
- Natural selection: The individuals survive because they have particular characteristics that make them more adapted.
- Heredity of traits: The surviving individuals share some of their characteristics to their descendants.

EAs are used to solve a large panel of optimization, design and modeling problems (Eiben and Smith, 2015). For example, they can generate structures (of neural networks,Stanley and Miikkulainen (2002), or of objects, Hornby et al. (2011); Lipson and Pollack (2000)), images (Secretan et al., 2008; Sims, 1991), weights of neural networks (Floreano and Mondada, 1994; Whitley et al., 1990; Devert et al., 2008), virtual creatures (Sims, 1994; Lehman and Stanley, 2011b), robotic controller (Zykov et al., 2004; Godzik et al., 2003), soft robots (Cheney et al., 2013) or chemistry reactions (Gutierrez et al., 2014; Dinh et al., 2013). It is also very interesting to mention that, while the EAs come from the synthetic theory of evolution, EAs are also promising tool to study natural evolution (Smith, 1992; Lenski et al., 1999). For example, Clune et al. (2013) used evolutionary algorithms to investigate the evolutionary origins of modularity in biological networks. In other examples, EAs have also been used to explain the emergence of collaborative behaviors like cooperation (Bernard et al., 2015), altruism (Montanier and Bredeche, 2013; Waibel et al., 2011) or communication (Floreano et al., 2007; Mitri et al., 2009; Wischmann et al., 2012).

These application examples, while being incomplete, show that the domain of EAs is very large. However, it is interesting to note that most of them rely on the same principles. In general, one iteration of EAs (called generation) involves 3 steps (Eiben and Smith, 2003). After the initialization (typically random) of the population, (1) all the individuals of the population are evaluated on the problem (see Fig. 2.1B) and are then ranked based on their quality (see Fig. 2.1C). (2) the best individuals survive (see Fig. 2.1 E) while the others are likely to perish (see Fig. 2.1 D). This selection process mimics the fact that each individual has to compete in order to survive. (3) The surviving individuals are crossed and mutated to create offspring individuals, which are then used to form a new population.

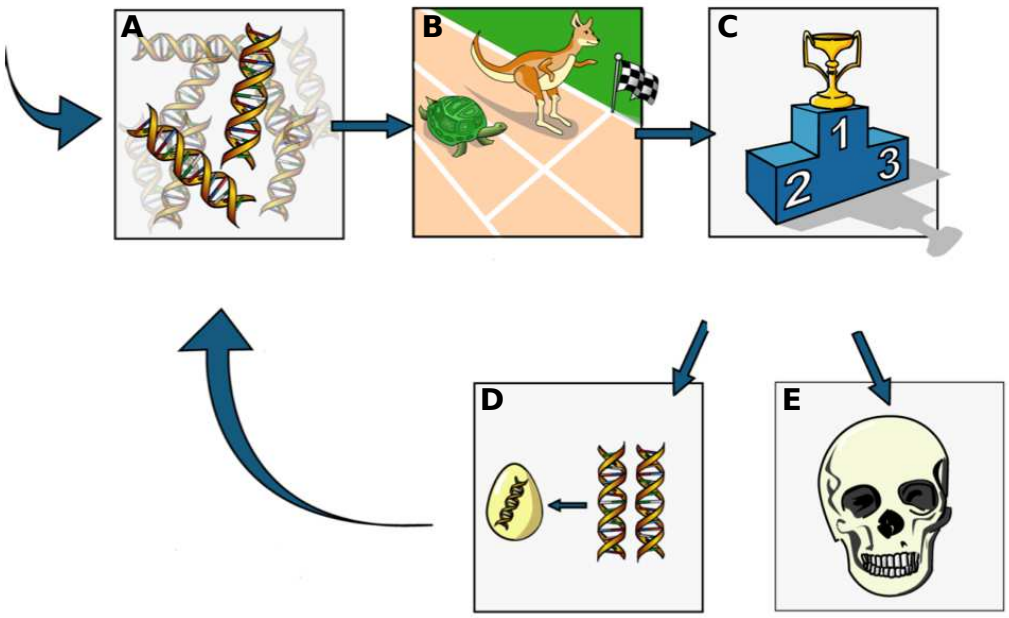


Figure 2.1: The main steps of evolutionary algorithms. All the population (A) is evaluated on the task (B). Based on the recorded performance of the individuals (fitness score), the population is sorted (C). The individuals that are in the bottom of the ordering are removed from the population (E) while the others are mutated and crossed in order to create a new population (D). This new population is then used to repeat the process, which ends when a performance criterion is reached or when a predefined number of generations has been executed.

This 3-step process then repeats with the new population making it progressively more adapted to the problem the algorithm is solving. The algorithm stops after a particular amount of time (CPU time or number of evaluations) or after the population has converged.

In the vast majority of work, EAs are employed to solve optimization problems and they are typically well tailored to deal with black-box functions (Eiben and Smith, 2003). The particularity of such functions is that no information about the structure or gradient of the function is available. The algorithm can only supply a potential solution and gets the corresponding value, making the optimization problem notably difficult. In EAs, the optimized function is called the *fitness function* (Eiben and Smith, 2003):

Definition 1 (Fitness function) *The fitness function represents the requirements the individuals have to adapt to. Mathematically, it is a function that takes as an input a potential solution (also called* an individual*) and outputs a fitness score that represents the quality of the individual on the considered problem.*

The fitness score of an individual can be seen, from a biological point of view, as its ability to survive in its environment. As describe below, this score has an impact on the selection and reproduction processes. The potential solutions are described thanks to a genotype that can be expanded into a phenotype.

Definition 2 (Genotype and Phenotype) *The genotype encodes the information that characterizes the individual. The phenotype is the expression of the information contained in the genotype into an actual potential solution (that can be applied to the problem). It corresponds to the observable and testable traits of the individual. The conversion between the genotype and the phenotype is call the genotype-phenotype mapping (Eiben and Smith, 2003).*

For example, a genotype can be a set of numbers, while the phenotype corresponds to an artificial neural network with synaptic weights parametrized by the values contained in the genotype.

Each potential solution is thus defined by a genotype and its corresponding phenotype, which can be evaluated through the fitness function. Several potential solutions are gathered into a population, which is then evolved by the EA:

Definition 3 (Population) *The population is a set of genotypes that represents potential solutions. Over the numerous generations of the algorithm, the individuals of the population will change and the whole population will adapt to the problem.*

While most of EAs follow this general framework, they differ in the employed genotypes and phenotypes and in the way they implement the selection, mutation and crossover steps.

Definition 4 (Survivor selection) *The Survivor selection mechanism selects among the individuals of the population those that will survive based on their quality (fitness score).*

Typically, some selection operators are based solely on the ranks (e.g., it selects the 10 first individuals), some others can involve random selection (e.g., tournament selection, where two randomly selected individuals of the population compete and only the best one is selected).

The selected individuals are then mutated and crossed in order to create the new population that will be used for the next generation. The crossover and mutation steps affect the information contained in the genotype in order to add variations in the produced individuals:

Definition 5 (Crossover) *The crossover operator merges information from several parent individuals (typically two) into one or two offspring individuals.*

The selection of the parent individuals often involves another selection step (called Parent Selection operator, Eiben and Smith (2003)), which selects the potential parents based on their quality. This mechanism aims to simulate the attractiveness of the individuals in order to allow the better individuals to become parents of the next generation.

Definition 6 (Mutation) *The mutation operator produces variations in the offspring individuals by altering directly the information contained in the genotype.*

These operators are typically stochastic, as the parts of the genomes that are altered or combined and the value of the mutated genes are most of the time randomly determined. We can also note that the use of the crossover operator is not mandatory, for example we can note that most of modern EAs disable this operator and only rely on the mutation operator (Stanley and Miikkulainen, 2002; Clune et al., 2013).

The main role of the cross-over and mutation operator is to make the population explore the genotype space, while the role of the survivors and parents selection operators is to promote the better individuals and to allow the population to progressively improve its quality over the generations. However, even if EAs have shown impressive results and applications (Eiben and Smith, 2003), there is no proof of optimality. Consequently, it is likely that the obtained solution is only a local optimum, but this is a common issue with optimization algorithms (Papadimitriou and Steiglitz, 1998).

From an historical point of view, we find behind the name "Evolutionary Algorithms" or "Evolutionary Computation", several families of algorithms that refer to different combinations or implementations of the operators defined previously. Some of these methods are designed for some genotype/phenotype in particular. The most famous of these families are (De Jong, 2006):

- Genetic Algorithms: This family of algorithms is devoted to the evolution of strings of characters or of numbers, which allows the algorithm to use operators that are independent to the considered problem. The main source of variation of the genotype is the crossover operator that combines parts of individuals in order to create new ones. The usual selection operator of this family consists in changing a constant proportion of the population (for example 50%) with new individuals.

- Evolutionary Strategies: This family of algorithms mainly differs from the first one in its selection operator. In this case, the selection operator considers both the current population and several new individuals. As a consequence, only high performing new individuals are added to the population and the proportion of elitism varies according to the quality of the new individuals. The second difference with genetic algorithms is source of genotype variations, which here mainly consists in random mutations. A famous Evolutionary Strategy is CMA-ES (covariance matrix adaptation - evolutionary strategy, Hansen (2006)).

- Evolutionary and genetic Programming: This family of algorithms aims to evolve computer programs or mathematical functions to solve a predefined task. The generated programs are most of the time finite state machines (Fogel et al., 1966) or tree structures. The structure of the trees is also evolved and each node is a sub-function (for example $\sin, \cos$) or an operator (for example $+, -$) and each leaf is an input variable or a constant (Cramer, 1985; Koza, 1992; Schmidt and Lipson, 2009).

Nowadays, these distinctions become less and less clear and several new categories appeared like, for example, the Estimation of Distribution Algorithms (EDA, Larranaga and Lozano (2002)), the Multi-Objective Evolutionary Algorithms (MOEA, Deb (2001)) or the Ant Colony Optimization algorithms (ACO, Dorigo and Birattari (2010)).

2.2.2 Multi-objective optimization

In some situations, a problem cannot be formalized as a single fitness function but rather as several ones. For example, when designing a car; several features are important like the cost, the consumption, and the maximum speed. One way to solve this problem with EAs is to use a weighted sum of the different features and to use it as the fitness function. Nevertheless, such strategy fixes the trade-off between all the features of the potential produced cars. In other words, the weights of the sum will define the position of the extremum of the fitness function. For example, one configuration of this sum may favor the price of the car while some other configurations may favor its speed or its cost. In such design process, it is likely that the desired trade-off is initially unknown and may require re-launching the optimization process each time the designers want another configuration.

One alternative is to look for all the possible trade-offs or more precisely all the *non-dominated trade-offs*. This concept of dominance states that you cannot improve one of the features of the obtained trade-offs without decreasing the others ones. This notion comes from the definition of *Pareto dominance* (Pareto, 1896; Deb, 2001):

Definition 7 (Pareto dominance) *A solution (or individual)* $\mathbf{x_1}$ *is said to dominate an other solution* $\mathbf{x_2}$*, if and only if both conditions 1 and 2 are true (see Fig. 2.2 A):*

- *(1) The solution* $\mathbf{x_1}$ *is no worse than* $\mathbf{x_2}$ *in all the objectives.*
- *(2) The solution* $\mathbf{x_1}$ *is strictly better than* $\mathbf{x_2}$ *in at least one objective.*

Definition 8 (Pareto front) *Based on this dominance notion, we can define a Pareto front, which is the set of all the non-dominated solutions (see Fig. 2.2 B).*

One of the most used Multi-Objective Evolutionary Algorithm (MOEA) is the *Non-dominated Sorting Genetic Algorithm II* (NSGA-II)[1] introduced by Deb et al. (2002). This algorithm ranks the individuals according to their position in the Pareto front or in the subsequent fronts (see Fig. 2.2B). For example, if the selection step requires selecting 30 individuals, it will first select all the individuals of the Pareto front and then on the subsequent fronts. It may then happen that a front cannot fit entirely in the selection. For example, if the two first fronts contain in total 25 individuals and the third one contains 10 individuals, this last front cannot fit entirely in the 30 selected individuals (see Fig. 2.2C). To deal with this situation, the authors introduced the concept of *crowding distance* that estimates the density of individuals around each of them. This distance is used as a second selection criterion when the algorithm has to compare individuals that are on the same front. The main insight of this distance is to favor individuals that are isolated in the front because they represent solutions that are different from the others.

Based on this selection operator, NSGA-II optimizes its population to have as much as possible individuals on the Pareto front, but with a selection pressure that tends to evenly spread these individuals on the front. The individuals of the population that are on the Pareto front at the end of the evolutionary process represent the solutions provided by the algorithm. It consists in a set of different trade-offs over the objectives. The designers can pick up the most suited trade-off according to the situations.

2.2.2.1 Helper objective

In the previous section, we presented how EAs can be used to optimize several objectives simultaneously. In the previous examples of the car, the objectives were specific to the problem (for example the cost and the speed of a car). However, additional objectives can be used in order to help the algorithm to find the best solutions on one principal objective (Knowles et al., 2001; Jensen, 2005). Optimizing a second objective may lead the population toward better solutions of the first objective. Such objectives are called *helper objectives* (Doncieux and Mouret, 2014; Jensen, 2005) and the algorithm can be written as:

$$\text{maximize} \begin{cases} \text{Fitness objective} \\ \text{Helper objective} \end{cases}$$

[1] NSGA-II is now outdated. However, it is a well known, robust and effective algorithm, which is sufficient for the applications that we will consider in this manuscript.

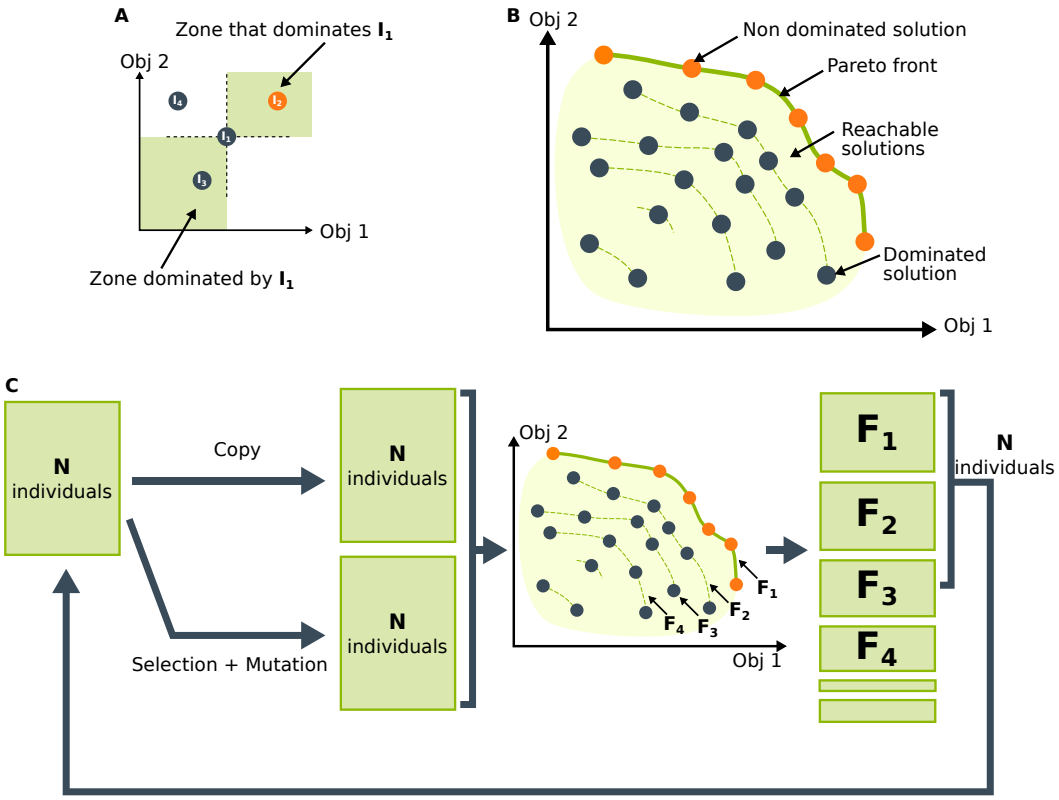


Figure 2.2: The Pareto dominance and the Pareto front. In these schemes, the algorithm maximizes both objective 1 and objective 2. Each dot represents an individual, which is for example contained in the population. Orange dots are non-dominated individuals, while the (dark) blue ones are dominated. (A) $\mathbf{I_1}$ is dominated by $\mathbf{I_2}$ because $\mathbf{I_2}$ is better than $\mathbf{I_1}$ on the two objectives, while $\mathbf{I_1}$ dominates $\mathbf{I_3}$. $\mathbf{I_1}$ and $\mathbf{I_4}$ do not have any dominance relation because none of them is as good as the other on all the objectives ($\mathbf{I_1}$ is better than $\mathbf{I_4}$ on the objective 1, while $\mathbf{I_4}$ is better than $\mathbf{I_1}$ on the objective 2). We can also see that $\mathbf{I_4}$ is dominated by $\mathbf{I_2}$ because $\mathbf{I_2}$ is as good as $\mathbf{I_4}$ on objective 2 but better on objective 1. (B) The solid green line represents an approximation of the Pareto front, which is composed by all the non-dominated individuals of the population. The dashed green lines represent the subsequent fronts. These fronts gather solutions that are only dominated by the solutions of the previous fronts. (C) The stochastic multi-objective optimization algorithm NSGA-II (Deb et al., 2002). Starting with a population of N randomly generated individuals, an offspring population of N new candidate solutions is generated using the best candidate solutions of the current population. The union of the offspring and the current population is then ranked according to Pareto dominance (here represented by having solutions in different ranks connected by lines labeled F_1, F_2, etc.) and the best N candidate solutions form the next generation.

Preventing early convergence of EAs is typically a good example where helper objectives can be useful (Knowles et al., 2001). A promising way to prevent this problem consists in preserving the diversity of the population. Thanks to this diversity, some individuals are likely to get out of local optima and then to populate more promising area of the search space. The diversity of the population can be fostered by using an additional objective that defines a selection pressure to favor individuals that are different from the rest of the population. The diversity of one individual (x_i) can be computed as the average distance from this individual and the others in the population.

$$\text{diversity}(\mathbf{x}_i) = \frac{1}{N} \sum_{j=0}^{j=N} d(\mathbf{x}_i, \mathbf{x}_j) \tag{2.2}$$

where x_i denotes the i-th individual of the population and N the size of the population. The function $d(x, y)$ computes the distance between two individuals.

There exist several ways to compute the distance between two individuals. One way consists in computing a Euclidean or an editing distance between the two genotypes (mainly when they are strings of numbers or characters, Gomez (2009); Mouret and Doncieux (2009)). Another way can be to compute the distance according to the phenotypes (Mouret and Doncieux, 2012), but for these methods there is no proof that two similar (or very different) genotypes or phenotypes will lead to similar solutions (respectively, different solutions). Avoiding premature convergence requires having a diversity of solutions, which is not necessarily provided by a diversity of genotypes (or phenotype). For this reason, the most promising way to compute the distance between two individuals is to consider the behavior of the individual (typically in evolutionary robotics, we can consider the robot's behavior, Mouret and Doncieux (2012)). We call this type of diversity the *behavioral diversity*. Nonetheless, computing a distance between two behaviors is still an open question because it requires finding a descriptor of this behavior that can then be used in the distance computing, but there is not universal procedure to define this descriptor. For example, with a mobile robot it can be the robot's trajectory or its final position.

2.2.3 Novelty Search

The Novelty Search is a concept introduced by Lehman and Stanley (2011a), which proposes to look for novel solutions (or behaviors) instead of focusing on fitness optimization. It goes in a sense further than the behavioral diversity because, instead of only considering the current population and promoting the most different individuals, the Novelty Search considers all the individuals encountered during the evolutionary process (or a representative subset) and rewards the most novel ones. To keep a trace of all these individuals, this algorithm stores the individuals (or a descriptor of them) into an *archive*, which is used afterwards to compute the novelty of the newly generated individuals. The results of this algorithm suggest that,

for some problem families, looking for novel solutions is more interesting than optimizing a performance objectives (Stanley and Lehman, 2015). For example, it has been shown that the Novelty Search algorithm outperforms traditional evolutionary algorithms on some deceptive tasks (like the deceptive maze experiment, Lehman and Stanley (2011a)).

Lehman and Stanley then extended their algorithm to deal with a longstanding challenge in artificial life, which is to craft an algorithm able to discover a wide diversity of interesting artificial creatures. While evolutionary algorithms are good candidates, they usually converge to a single species of creatures. In order to overcome this issue, they recently proposed a method called *Novelty search with local competition* (Lehman and Stanley, 2011b). This method, based on multi-objective evolutionary algorithms, combines the exploration abilities of the Novelty Search algorithm (Lehman and Stanley, 2011a) with a performance competition between similar individuals.

The Novelty Search with Local Competition simultaneously optimizes two objectives for an individual: (1) the novelty objective ($Novelty(\mathbf{x})$), which measures how novel is the individual compared to previously encountered ones, and (2) the local competition objective ($Qrank(\mathbf{x}_i)$), which compares the individual's quality to the performance of individuals in a neighborhood, defined with a morphological distance.

With these two objectives, the algorithm favors individuals that are new, those that are more efficient than their neighbors and those that are optimal trade-offs between novelty and "local quality". Both objectives are evaluated thanks to an *archive*, which records all encountered families of individuals and allows the algorithm to define neighborhoods for each individual. The novelty objective is computed as the average distance between the current individual and its k-nearest neighbors, and the local competition objective is the number of neighbors that the considered individual outperforms according to the quality criterion.

Lehman and Stanley (2011b) successfully applied this method to generate a high number of creatures with different morphologies, all able to walk in a straight line. The algorithm found a heterogeneous population of different creatures, from little hoppers to imposing quadrupeds, all walking at different speeds according to their stature.

The novelty objective fosters the exploration of the reachable space. An individual is deemed as novel when it shows a behavioral descriptor ($\mathscr{B}(\mathbf{x}_i)$) that is different from those that have been previously encountered. The novelty score of an individual $\mathbf{x}_i$ ($Novelty(\mathbf{x}_i)$) is set as the average distance between the behavioral descriptor of the current controller ($\mathscr{D}(\mathbf{x}_i$, which can be for example the ending point of a mobile robot) and the descriptors of the individuals contained in its neighborhood ($\mathscr{N}(\mathbf{x}_i)$):

$$Novelty(\mathbf{x}_i) = \frac{\sum_{\mathbf{x}_j \in \mathscr{N}(\mathbf{x}_i)} d(\mathscr{B}(\mathbf{x}_i) - \mathscr{B}(\mathbf{x}_j))}{card(\mathscr{N}(\mathbf{x}_i))} \tag{2.3}$$

To get high novelty scores, individuals have to show descriptors that are differ-

ent from the rest of the population. Each time a controller with a novelty score exceeds a threshold (ρ), this controller is saved in an *archive*. Given this archive and the current population, a neighborhood is defined for each individual ($\mathscr{N}(\mathbf{x}_i)$). This neighborhood regroups the k nearest individual, according to the behavioral distance. We can see that the definition of the Novelty Search objective is similar to the behavioral diversity, but several main differences exist. The fist difference is that the novelty search considers only the k-nearest individuals instead of the whole population when computing the average distance. The second and main difference remains in the fact that while the behavioral diversity considers only the individuals of the current population, the novelty search considers also individuals that have been stored in the archive. This allows the Novelty Search algorithm to keep a history of what it found.

The local quality rank promotes individuals that show particular properties, like stability or accuracy. These properties are evaluated by the quality score ($quality(\mathbf{x}_i)$), which depends on implementation choices and particularly on the type of phenotype used. In other words, among several individuals that show the same descriptor, the quality score defines which one should be promoted. For an individual $\mathbf{x}_i$, the rank ($Qrank(\mathbf{x}_i)$) is defined as the number of controllers from its neighborhood that outperform its quality score: minimizing this objective allows the algorithm to find individuals with better quality than their neighbors.

$$Qrank(\mathbf{x}_i) = card(\mathbf{x}_j \in \mathscr{N}(\mathbf{x}_i), quality(\mathbf{x}_i) < quality(\mathbf{x}_j)) \tag{2.4}$$

2.2.4 Evolutionary robotics

Evolutionary algorithms have been widely used to learn behaviors in robotics during the two last decades. In the mid 90's, several works used EAs to find neural network parameters to control robots. For example, Lewis et al. (1994) evolved a bit string of 65 bits to tune the synaptic connections between 12 neurons to generate insect-like like walking gaits with a physical six-legged robot. The evolutionary process was divided into two steps: the first step focuses on generating oscillations between pairs of neurons while the second step uses these oscillating bi-neurons and interconnects them to generate walking gaits. This division in the evolutionary process aims to bootstrap the generation of the gait but the total process still required between 200 and 500 physical trials.

Another example comes from Floreano and Mondada (1994) who used an EA to generate a neural network that produces a navigation and obstacle avoidance behavior for a physical Khepera robot (Mondada et al., 1994) (a miniature two-wheeled robot). The main objective of this work was to reproduce "Braitenberg" behaviors (Braitenberg, 1986) with a neural network that has its parameters tuned by an EA. It took between 30 and 60 hours and about 8000 evaluations on the physical robot to generate such behaviors. The neural network has about 20 parameters (the exact number of evolved parameters is not specified in the paper). It is interesting to note that this is one of the first work using EAs on a physical robot

for hours without external human interventions.

Similar procedures have been used to evolve behaviors for various types of robots or various kinds of behaviors (in simulation). For example, Beer and Gallagher (1992) evolved a dynamical neural network that allows a mobile robot to reach predefined positions based on sensory stimulus (Chemotaxis). Similarly, Cliff et al. (1993) evolved a neural network that maps the sensory perceptions (from the robot's camera, its whiskers and bumpers) to the motors. This work is particularly interesting as it proposes to see the robot as a moving set of sensors regardless the way the robot is actuated. Based on this concept, the authors attached the camera and the other sensors on pole beneath a Cartesian gantry. The "robot" is thus actuated by the gantry and moves through an arbitrary scene. Non-mobile robots have also been investigated, for example Whitley et al. (1994) evolved a neural network that control a simulated balancing pole and Moriarty and Miikkulainen (1996) applied an EA on a simulated robotic arm to generate obstacle avoidance behaviors with a neural network.

During the following decade, different concepts have been investigated based on these ideas. For example, EAs have been applied on complex robots like in Zykov et al. (2004) in which a physical nine-legged robot learns dynamic gaits. The EA optimizes the coefficient of a matrix that defines the periodic movement of the 12 pneumatic pistons that constitute the robot (corresponding to 24 valves and a total of 72 parameters). This work needed about 2 hours and between 500 and 750 physical trials to generate a walking gait on the physical robot.

Another research direction focused on the type of genotypes/phenotypes used to encode the controller. For example, Yosinski et al. (2011) used HyperNEAT, a generative encoding that creates large-scale neural networks with the ability to change the network's topology (Stanley et al., 2009), to control a physical quadruped robot. The generative ability of this new type of encoding opens the possibilities of behavior that become, theoretically, infinite. Other approaches used EAs to optimize the parameters of predefined and parametrized leg trajectories (Chernova and Veloso, 2004; Hornby et al., 2005) or cellular automata (Barfoot et al., 2006) to control legged robots. It is worth noting that one of the behaviors found with these methods has been integrated in the consumer version of the quadruped robot named AIBO (Hornby et al., 2005). All these techniques required between 2 and 25 hours to generate walking gaits and between 500 and 4000 evaluations.

Some key concepts also emerged during the last decade and continue to influence the current research in learning for robotics. For example, the notion of embodiment (Pfeifer and Bongard, 2007) proposes to see the cognitive abilities of robots, not only as consequence of the complexity of the robot's brain, but as the combination and the synergies of the interactions of the robot's body and its brain. However, such interactions are challenging to generate with engineering methods and EAs represent a promising approach to evolve the brain and/or the body based on the robot's interactions with its environment.

A very good example of this principle is the Golem project (Lipson and Pollack, 2000) that uses EAs to evolve both the morphology of the robot and the neural

network that controls the robot. Another example is the work of Bongard et al. (2006), which uses an EA and the results (sensory perceptions) of the interactions of the robot with the environment to automatically generate a model (in a physical simulator) of the robot's morphology. Such model can be used afterward to learn new behaviors in simulation.

The domain of evolutionary robotics also provides new tools to explain the biological evolution (Smith, 1992; Lenski et al., 1999). For example, researchers study the principles of distributed evolution thanks to swarm of robots, where each robot represents an individual of the population. This concept, also known as embodied evolution (Watson et al., 2002) offers several promising properties, like the ability to distribute the evaluation of the individual but it may also allow new collective behaviors to emerge, like cooperation (Bernard et al., 2015) or altruism (Montanier and Bredeche, 2013; Waibel et al., 2011).

2.2.5 The Reality Gap problem and the Transferability approach

Most of the previously described methods and EAs in general, typically require to test several thousands of candidate solutions during the optimization process (see Table 2.1). This high number of tests is a major problem when they are performed on a physical robot. An alternative is to perform the learning process in simulation and then apply the result to the robot. Nevertheless, solutions obtained in simulation often do not work well on the real device, because simulation and reality never match perfectly. This phenomenon is called *the Reality Gap* (Jakobi et al., 1995; Koos et al., 2013b).

Several methods have been proposed to deal with this problem. For example Jakobi et al. (1995) proposed to use a minimal simulator and to add noise to this simulator, in order to hide the details of the simulator that the robot may use to artificially improve its fitness. This technique has been successfully applied to a T-maze experiment with a Khepera robot (Mondada et al., 1994) and to a visual discrimination task on a gantry robot Jakobi et al. (1995). It has also been used to allow an octopod robot to learn walking gaits (Jakobi, 1998). However, this method seems to be sensitive to some parameter values, as Koos et al. (2013b) were unable to reproduce the original results of the T-Maze experiment.

Another example of method to deal with the reality gap problem consists in using simultaneously several simulators (Boeing and Braunl, 2012). Each candidate solution is evaluated on all the simulators and its fitness value is defined as a statistical combination of the fitness values coming from each simulator. With this method, a good candidate solution should show a good fitness value with every simulator, which may allow solutions to cross the reality gap, as it is statistically unlikely that the simulation details, used to artificially improve the fitness, are shared across all the simulators. This method has been applied to a wall following experiment with an autonomous underwater robot (Boeing and Braunl, 2012) and to a locomotion task with a bipedal robot (Boeing, 2009). However, this method significantly increase the computational cost of the algorithm, as it requires evaluating

the solutions on several simulators instead of on a single one.

In Lehman et al. (2013), the authors made the hypothesis that behaviors that react to their environment are more likely to be able to cross the reality gap. Indeed, if the individual avoids to over-fit its behavior according to the simulated environment but rather reacts to the situations it is actually facing, it is likely that such behavior is able to handle the differences between the simulation and the reality. This hypothesis has been successfully tested on a maze navigation task using a Khepera III robot. The authors also showed that their approach can be combined with Jakobi's noise-based technique (Jakobi et al., 1995) to further improve the robot's ability to cross the reality gap.

The transferability approach (Koos et al., 2013b,a; Mouret et al., 2012) aims to cross the reality gap by finding behaviors that act similarly in simulation and in reality. During the evolutionary process, a few candidate controllers are transferred to the physical robot to measure the behavioral differences between the simulation and the reality; these differences represent the *transferability value* of the solutions. With these few transfers, a *regression model* is built up to map solution descriptors to an estimated transferability value. The regression model is then used to predict the transferability value of untested solutions. The transferability approach uses a multi-objective optimization algorithm to find solutions that maximize both task-efficiency (e.g. forward speed, stability) and the estimated transferability value.

This mechanism drives the optimization algorithm towards solutions that are both efficient in simulation and transferable (i.e. that act similarly in the simulation and in the reality). It allows the algorithm to exploit the simulation and consequently to reduce the number of tests performed on the physical robot.

This approach was successfully used with an E-puck robot in a T-maze and with a quadruped robot that evolved to walk in a straight line with a minimum of transfers on the physical robots (Koos et al., 2013b). The reality gap phenomenon was particularly apparent in the quadruped experiment: with a controller optimized only in simulation, the virtual robot moved $1.29m$ (in $10s$) but when the same controller was applied on the physical robot, it only moved $0.41m$. With the transferability approach, the obtained solution walked $1.19m$ in simulation and $1.09m$ in reality. These results were found with only 11 tests on the physical robot and $200,000$ evaluations in simulation. This approach has also been applied for humanoid locomotion (Oliveira et al., 2013) and damage recovery on a hexapod robot (Koos et al., 2013a).

2.2.6 Partial conclusion

The previous examples of applications of EAs in robotics and also in the other application domains (like structures, objects, pictures, creatures generation) show the ability of EAs to create and generate a large variety of things, a kind of creativity ability (Nguyen et al., 2015). The creativity is a feature that is likely to be required in learning and adaptation algorithms. For example, a behavior that allows a robot to walk in spite of a broken leg may be significantly different from the behavior

it used when it was intact. Adaptation algorithms thus need to find behaviors that may be far (in the search space) from the initial starting point (global search techniques) and for this aspect, EAs seem particularly well suited.

However one of the main drawbacks of EAs is the number of evaluations required to generate those solutions (between several hundreds to several thousands trials, see section evolutionary algorithms in table 2.1). This is probably the direct and inevitable consequence of the creativity: being creative requires testing a large variety of things and all these tests represent a cost in term of learning time. The balance between creativity and learning time is important for applications like adaptation to unexpected situations. We cannot imagine a robot trying thousand actions when it has to adapt for several reasons: first, because the power resources are often limited on autonomous robots (battery) and second, because of safety reasons (the robot can be damaged during intensive learning session).

2.3 Policy search algorithms

2.3.1 Common principles

Reinforcement learning (RL) is a research field that has been significantly growing during the three last decades and that shows impressive results. A recent illustration of these results is the work from Mnih et al. (2015), in which a reinforcement learning algorithm coupled with a deep learning algorithm is able to learn to play a large variety of Atari games. As mentioned previously in this chapter (see section 2.1), traditional RL algorithms are, in the vast majority, designed to work with small and discrete state and action spaces, while Learning in robotics involves most of the time using continuous and high dimensional state spaces and action spaces. It is why most of applications of RL in robotics use *Policy Search* algorithms (PS), which is known to have a better scalability and to work in continuous domains (Sutton et al., 1999; Deisenroth et al., 2013b).

Most of PS algorithms can be decomposed into one loop of three steps (depicted in Fig: 2.3): (1) The exploration step in which the current policy is somehow perturbed in order to produce variations in robot's movement; (2) the policy evaluation step, which aims to assess the quality of the newly generated behaviors and (3) the policy update step, which changes the policy parameters according to the acquired data. In the following subsections, we will detail how these different steps are achieved in some of the most common PS algorithms.

While policy search and evolutionary algorithms are two different research fields that grew independently, it is interesting to note that they share strong similarities in their overall structure. For instance, the exploration step of PS algorithms has the same goal as the mutation operators in EAs and the update step is close to the selection operators in EAs. Two of the main differences between EAs and PS algorithms are (1) the evaluation step, which considers episodic evaluations in the vast majority of EAs and step-based evaluations in PS algorithms, and (2) the use of a population in EAs while only one policy is kept from one iteration to the

Table 2.1: **How long many previous robot learning algorithms take to run**. While comparisons between these algorithms are difficult because they vary substantially in their objective, the size of the search space, and the robot they were tested on, we nonetheless can see that learning times are rarely below 20 minutes, and often take hours.

approach/article	starting behavior ⋆	evaluations	learning time	robot	DOFs†	param.‡	reward
Evolutionary Algorithm							
Chernova and Veloso (2004)	random	4,000	5 h	quadruped	12	54	external
Zykov et al. (2004)	random	500	2 h	hexapod	12	72	external
Hornby et al. (2005)	non-falling	500	25h	quadruped	19	21	internal
Barfoot et al. (2006)	random	1,150	10 h	hexapod	12	135	external
Yosinski et al. (2011)	random	540	2 h	quadruped	9	5	external
Bongard et al. (2006)[1]	random	16	4 h	hexapod	12(18)	30	internal
Policy Gradient Methods							
Kimura et al. (2001)	n/a	10,000	80 min.	quadruped	8	72	internal
Kohl and Stone (2004)	walking	345*3	3 h	quadruped	12	12	external
Tedrake et al. (2005)	standing	950	20 min.	biped	2	46	internal
Geng et al. (2006)	walking	n/a	4-5 min.	bipedal	4	2	internal
Bayesian optimization							
Lizotte et al. (2007)	center§	120	2h	quadruped	12	15	internal
Calandra et al. (2014)	random	40*3	46 min.	biped	4	8	external
Tesch et al. (2011)	random	25	n/a	snake	16	7	external

⋆Behavior used to initialize the learning algorithm, usually found via imitation algorithms. † DOFs: number of controlled degrees of freedom. ‡ param: number of learned control parameters.§ center: center of the search space.

[1] The authors do not provide time information, reported values come from the implementation made in Koos et al. (2013a).

next in PS algorithms. However, these differences tend to fade and the borderline between EAs and PS algorithms becomes more and more blurry (Stulp and Sigaud, 2013). For example, some of the most recent PS algorithms use episodic evaluations (Daniel et al., 2012; Stulp and Sigaud, 2012) and some EAs (steady state genetic algorithms, Syswerda (1991)) generate only one or two individuals per generation. These similarities are well illustrated by a few algorithms that are considered both as efficient EAs and PS algorithms, like the CMA-ES algorithm (Hansen, 2006), which is a generic optimization algorithm used in both the fields of policy search (Stulp and Sigaud, 2013) and artificial evolution (Igel, 2003).

In the following sections we will present some of the most common concepts in PS algorithms for robot learning and some well-known algorithms. In particular, we will see that two approaches exist to perform the exploration of the search space and the evaluation of the policies: (1) The first approach considers the whole execution of the policy (episodic) while (2) the second approach considers each time-step of the execution. It is common that algorithms that use one of these approaches for the exploration use the same approach for the evaluation step, and vice versa (Deisenroth et al., 2013b). We will also see that the vast majority of PS methods update their policy by following a local estimate of the gradient of the performance function. The more advanced methods mainly aim to mitigate some of the limitations of this gradient based approach, for example to deal with plateaus in the function or with the sensibility of some parameters. This common principle in PS algorithms makes these algorithms more related to local search techniques than to global ones. This aspect is well illustrated by the fact that the vast majority of experiences using PS algorithms start from a policy learned via an imitation learning algorithm and are used to improve its quality. In this section we will deliberately focus our analysis on model-free policy search methods because the last section of this chapter will present in detail the Bayesian Optimization algorithm (see section 2.4), which is a model-based policy search algorithm and which is known to perform more global optimization than model-free approaches (Jones et al., 1998)

In addition to all the details presented in this chapter, we invite the interested readers to refer to Deisenroth et al. (2013b) and Kober et al. (2013). These two reviews cover a large part of work done in this domain and they go in very deep details on a large number of algorithms. Most of the information presented in this section comes from these two reviews.

2.3.1.1 Exploration

In PS algorithms, the policy is, in the majority of work, parametrized by a parameter vector $\boldsymbol{\theta}$ (Kober and Peters, 2011; Kohl and Stone, 2004; Kormushev et al., 2010). During the exploration step, PS algorithms generate several variations of the current policy. The whole execution of a policy is called a *rollout*. Two main exploration strategies are considered in the literature: (1) exploration in the action space and (2) exploration in the parameter space.

The exploration in the action space uses the parameters of the current policy

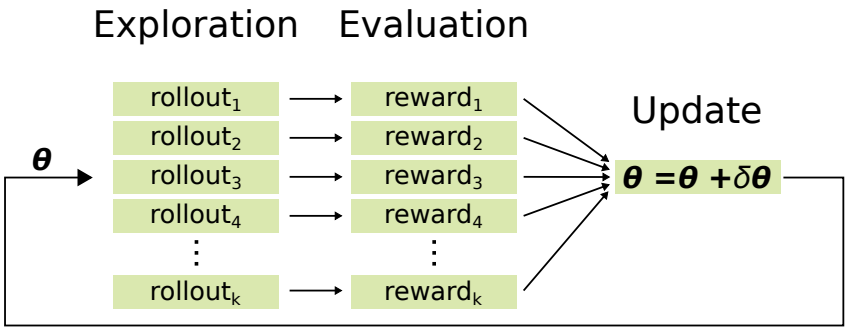


Figure 2.3: Generic Policy Search iteration. The current policy defined with the parameter vector θ is used to generate k rollouts thanks to perturbations on the parameters or on the motor commands. Each rollout is evaluated on the system and is associated to a reward value or several intermediate rewards. Based on these rewards, the current policy is updated and the process repeats.

($\boldsymbol{\theta}$) for all the rollouts and adds a random (most of the time Gaussian) perturbation to the commands that are sent to the motors. This perturbation is most of the time re-sampled at every time step and the corresponding commands are recorded to be used during the update policy step. This approach is one of the first exploration strategies that appeared in the literature (Williams, 1992; Deisenroth et al., 2013b).

The exploration in the parameter space directly perturbs the parameter of the current policy. Thus, the policy parameters of the k^{th} rollout is equal to $\boldsymbol{\theta}_k = \boldsymbol{\theta} + \mathcal{N}(0, \Sigma)$. Note that with this exploration strategy, the motor commands generated by the new policies are no longer altered via a random perturbation because the rollouts use different parameters and are thus already different. Like with the action space exploration, these modifications of the parameters can be done only at the beginning of the evaluation (episodic) or at every time step (Deisenroth et al., 2013b).

In these two exploration strategies, the applied perturbations are most of the time Gaussian using diagonal covariance matrix (Σ). However, non diagonal matrices can also be used to perform correlated exploration For example, CMA-ES uses such correlated exploration strategy: the covariance matrix used for the exploration is updated (or adapted, for consistency with the algorithm's name) according to the previous evaluations. In Deisenroth et al. (2013b), the authors highlight that using a full covariance matrix increases significantly the algorithm learning speed (Stulp and Sigaud, 2012) but also that estimating this matrix can be very difficult when the search space becomes very large as it requires a large number of samples (Rückstiess et al., 2010).

2.3.1.2 Evaluation of the policy

The goal of the evaluation step is to assess the quality of the policy and to gather data about its execution that will be used during the policy update step. The data acquired during the evaluation of each policy are typically the explored states, the

actual motor commands used and the intermediate rewards. As mentioned in the beginning of this section, the evaluation of the policy quality can consider every time steps of its execution or its full execution (episodic evaluation).

In most of PS algorithms using time-step evaluations, the quality of the policy is assessed by the sum of the future rewards that the system will receive at every time step until the end of the episode (at time step T). This quality is computed for every time step, leading to the quality Q_t that corresponds to the state-action value function:

$$Q_t(x_t, u_t) = \mathbb{E}\left[\sum_{h=t}^{T} r_h(x_h, u_h)\right]$$

Such a function is in practice very difficult to compute in large state and action spaces and it is often required using approximation methods, like Monte-Carlo (Metropolis and Ulam, 1949), to estimate its value. However, these approximation algorithms lead most of the time to a large variance in their results or require a very large number of samples. Fortunately, most PS algorithms are robust enough to deal with noisy approximation of the quality. Therefore, most PS algorithms use directly the sum of the immediate reward (Deisenroth et al., 2013b):

$$Q_t(x_t, u_t) = \sum_{h=t}^{T} r_h(x_h, u_h)$$

This quality function is used in the most common PS algorithms like REINFORCE algorithm (Williams, 1992), the POWER algorithm (Kober and Peters, 2011) and PI^2 (Theodorou et al., 2010). A discount factor over time is often used in order to reduce the influence of future time steps (with $\gamma \leq 1$):

$$Q_t(x_t, u_t) = \sum_{h=t}^{T} \gamma^h r_h(x_h, u_h)$$

Episodic evaluations can rely on this quality definition too by taking the quality of the policy at the first time step (Q_0), but many other reward functions can be employed in the context of episodic evaluations. For example, the reward function can directly consider the quality of a whole episode without the constraint of decomposing the reward into intermediate ones. This constraint is important, as a large number of quality functions cannot be decomposed into several intermediate rewards. For example, learning a walking gait on a legged robot and trying to maximize its velocity is typically a task where it is difficult to define intermediate rewards as the robot needs to walk for a few seconds in order to assess its average speed. In general, the episodic return of policy $\boldsymbol{\theta}$ is called $J(\boldsymbol{\theta})$ instead of Q_t. In the following sections, we will mainly consider episodic evaluations and thus use this notation when it is appropriate.

The PS approaches that have initially been designed to deal with step-based or episode-based evaluations have, most of the time, been extended to work with the other type of evaluations. For example the episodic Natural Actor Critic algorithm

(eNAC) that we will present in the following sections has also a step-based variant presented in the same paper (Peters and Schaal, 2008a). The use of one type of evaluation rather than the other depends mainly on the problem. Step-based evaluations are likely to provide more data (one at every time step) and thus being more informative but such evaluation type cannot be applied to all the robotic applications (Deisenroth et al., 2013b).

2.3.1.3 Update of the policy

The update policy step uses the evaluated rollouts and the recorded data (quality, intermediate rewards, states trajectory) and updates the parameters of the current policy ($\boldsymbol{\theta}$) in the direction that will most likely improve its quality. In this section we will present some of the most common update strategies in order to illustrate the different inspirations that are used in PS algorithms. This section does not aim to be exhaustive as there exist a very large variety of update strategies and a deep comparison of these methods would be off of the scope of this manuscript.

Policy Gradient One of the most common approach to update the policy parameter is the local gradient of the quality function (according to the parameters' dimensions):

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where α is the learning rate and $\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\theta)$ is the gradient of $J(\theta)$ according to θ. The gradient of the quality function cannot be analytically computed as the quality function is unknown. Consequently, the methods relying on the gradient actually use an approximation of its value. A very simple and commonly used (Kohl and Stone, 2004; Peters and Schaal, 2008b, 2006) way to approximate the gradient is to use the *finite difference* method. This method uses a first-order Taylor-expansion of the quality function based on the rewards obtained from the rollouts to perform a linear regression that estimates the gradient:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{bmatrix} \Delta\boldsymbol{\theta}_0 & \Delta\boldsymbol{\theta}_1 & \dots & \Delta\boldsymbol{\theta}_k \end{bmatrix}^{+} \begin{bmatrix} \Delta J(\boldsymbol{\theta}_0) \\ \Delta J(\boldsymbol{\theta}_1) \\ \vdots \\ \Delta J(\boldsymbol{\theta}_k) \end{bmatrix}$$

where $\Delta\boldsymbol{\theta}_i$ is a perturbation vector applied on the original parameter vector $\boldsymbol{\theta}$ ($\boldsymbol{\theta}_i - \boldsymbol{\theta}$) and $\Delta J(\boldsymbol{\theta}_i)$ is the quality difference between the rollout i and the current policy quality ($J(\boldsymbol{\theta}_i) - J(\boldsymbol{\theta})$). The pseudo-inversion (symbolized by the "+" exponent) performs a least mean square regression between the applied perturbations and the variations of the quality. This simple approach, while being a local search technique, is very powerful on smooth or not too noisy quality function but it is very sensitive to its parameters like the learning rate α the number of rollouts k and the amplitude of the perturbations (Kober et al., 2013). In order to mitigate these limitations and to increase the convergence speed, other methods to approximate the gradient have

been developed (Peters and Schaal, 2006), like the likelihood ratio methods (also known as the REINFORCE trick, Williams (1992)) or the natural gradient methods (Peters and Schaal, 2008a). We will detail these two approaches in the following paragraphs.

The likelihood ratio methods assumes that each rollout, generated thanks to the current *stochastic* policy $\boldsymbol{\theta}$, has a trajectory τ and an accumulated reward $r(\tau) = \sum_{k=0}^{T} \alpha_k r_k(\tau)$. The quality of the $\boldsymbol{\theta}$ parameters is computed as the expectation of this reward according to potential trajectories:

$$J(\boldsymbol{\theta}) = \mathbb{E}[r(\tau)] = \int p(\tau|\boldsymbol{\theta}) r(\tau) d\tau$$

The gradient is thus equal to:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int \boldsymbol{\nabla}_{\boldsymbol{\theta}} p(\tau|\boldsymbol{\theta}) r(\tau) d\tau$$

The probability of each trajectory given $\boldsymbol{\theta}$ can be analytically computed as (Deisenroth et al., 2013b):

$$p(\tau|\boldsymbol{\theta}) = p(x_0) \prod_{t=0}^{T} p(x_{t+1}|x_t, u_t) \pi_{\boldsymbol{\theta}}(x_t|u_t)$$

The gradient of this expression is in practice very hard to compute, first because of the transition model that defines $p(x_{t+1}|x_t, u_t)$ is most of the time unknown and also because of the derivative of this product may lead to analytic equations that may be not tractable.

In order to deal with this situation, the likelihood ratio can be used to simplify the equation as it states that (Glynn, 1987):

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} p(\tau|\boldsymbol{\theta}) = p(\tau|\boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta}) \tag{2.5}$$

Thanks to this identity, the gradient of the quality can be reduced to:

$$\begin{aligned} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int p(\tau|\boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta}) r(\tau) d\tau \\ &= \mathbb{E}\big[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta}) r(\tau)\big] \\ &= \mathbb{E}\big[\textstyle\sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log(\pi_{\boldsymbol{\theta}}(x_t|u_t)) r(\tau)\big] \end{aligned}$$

This last expression has been obtained thanks to the log operator of the likelihood ratio that transformed the probability product into a sum. Moreover, the $p(x_{t+1}|x_t, u_t)$ term can be removed as it does not depend on $\boldsymbol{\theta}$, which is the case as long as the policy is stochastic (Kober et al., 2013; Deisenroth et al., 2013b). Finally, the gradient can be approximated thanks to a Monte-Carlo sampling to assess the expectation of this last expression. Algorithms using the likelihood ratio to approximate the gradient usually add a *baseline* to the reward function in order to reduce the variance introduced by the Monte-Carlo sampling. For more details, readers can refer to Kober et al. (2013) and to Deisenroth et al. (2013b).

It is worth noting that this approach to approximate the gradient performs the exploration by itself as it estimates the gradient based on the variations generated by the stochastic policy. This approach has been proved to converge to a good estimate of the gradient faster than the finite difference method (Glynn, 1987) (it requires fewer samples to obtain an accurate estimate of gradient than the finite difference method). Unfortunately this approach requires using stochastic policy (otherwise a model of the transition function has to be used), which limits the type of controller that can be used with this method. The REINFORCE algorithm introduced in Williams (1992) uses the likelihood ratio and can be considered as one of the first PS algorithm. It has, for example, been used to find the weights of a neural network (containing stochastic units).

The *natural gradient* (Amari, 1998) is another improvement of the traditional gradient optimization. It has initially been proposed to make it easier to cross plateaus in the quality function (regions with only small variations). The main idea of this method consists in finding the variation of the parameter vector ($\Delta\boldsymbol{\theta}$) that remains as close as possible to the actual gradient ($\boldsymbol{\nabla}_{\boldsymbol{\theta}} J$) but at a fixed distance between the prior distribution $p(\tau|\boldsymbol{\theta})$ and the next one ($p(\tau|\boldsymbol{\theta} + \Delta\boldsymbol{\theta})$). In natural gradient approaches, like the episodic Natural Actor Critic (eNAC, Peters and Schaal (2008a)), this distance is computed thanks to the Kullback-Leibler divergence (d_{KL}, Kullback and Leibler (1951)) defined with the Fisher-information matrix ($\mathbf{F}_{\boldsymbol{\theta}}$) as follow:

$$\mathbf{F}_{\boldsymbol{\theta}} = \mathbb{E}\Big[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta})^{\intercal}\Big]$$

$$d_{KL}\Big(p(\tau|\boldsymbol{\theta}), p(\tau|\boldsymbol{\theta} + \Delta_{\boldsymbol{\theta}})\Big) = \Delta_{\boldsymbol{\theta}}^{\intercal} \mathbf{F}_{\boldsymbol{\theta}} \Delta_{\boldsymbol{\theta}}$$

The perturbation of the parameter vector that is the most similar to the actual gradient but with a predefined distance between the two distributions can be analytically defined as:

$$\Delta\boldsymbol{\theta} \propto \mathbf{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

The distance (included in the proportional operator) is often subsumed into the learning rate of the gradient descent:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \mathbf{F}_{\boldsymbol{\theta}}^{-1} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

This approach has to be performed in addition to one of the previously presented methods as the gradient $\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ still needs to be estimated (for example with the likelihood ratio method, Williams (1992); Peters and Schaal (2008a)). The main purpose of this method is to increase the convergence speed of the learning algorithm, while profiting from most of the advantages of the gradient estimations methods. However, the computational complexity of this method can prevent the application of this method in some situations (Deisenroth et al., 2013b).

Other approaches Several other approaches have been developed to address limitations or drawbacks raised by gradient policy search methods. For example, the learning rate of such methods is a very sensitive parameter that directly impacts the convergence speed and the performance of the algorithms. In order to alleviate this issue, several algorithms based on the Expectation-Maximization (EM) algorithm (Dempster et al., 1977) have been developed and used to learn policies in robotics. This method transforms the reward value of the behaviors into an improper probability distribution of the "reward-event", which is a binary observation of success. This transformation is direct when the reward function is bounded as it can be represented as a ratio between its minimum and maximum value. Rewards that are close to the minimum can be considered as a low probability of the reward event, while high rewards can be seen as a high probability of the reward event. For unbounded reward function, the rewards can be transformed thanks to a soft-max function that will map each value between 0 and 1. The EM algorithm is used to find the policy parameters that maximize the log-marginal likelihood of the reward-event. It first evaluates the expectation of the current parameters, for example thanks to a Monte-Carlo sampling, and then uses these samples to update the parameters in order to maximize the likelihood. The Policy learning by Weighting Exploration with Returns (PoWER) algorithm uses this approach and does not require specifying a learning rate as the update of the policy parameters is directly achieved by the maximization step. Unfortunately, such hidden update phases may lead to unsafe behaviors on the physical robot and do not guarantee the avoidance of local optima (Kober et al., 2013).

The information theory is another source of inspiration for the update policies, for example the Relative Entropy Policy Search (REPS, Peters et al. (2010)) algorithm, in which the Kullback-Leibler divergence is used to compute the distance between the current trajectory distribution and the one obtained after the parameter update (similarly to the natural gradient policy search). This algorithm then uses a weighted average of the likelihood estimate (like in EM methods described above) to update the policy parameters without requiring a learning rate. It thus combines the advantages of both natural gradient and EM methods. An application example of the REPS algorithm is given in the next section.

Policy search algorithms can also take inspiration from optimal control theory to design update policies. For example, in the Policy Improvement with Path Integrals (PI^2, Theodorou et al. (2010)), the problem is expressed into a Hamilton–Jacobi–Bellman equation and then the Feynman-Kac lemma is used to transform this equation into a path integral. Such integral can then be approximated thanks to a Monte-Carlo sampling.

2.3.1.4 Model-based policy search algorithms

In addition to the model-free algorithms presented in this section, there also exist several model-based approaches. The goal of such method is to build a model of the quality function based on the samples acquired on the system and then use this

model to improve the policy. The main challenge of this family of algorithms is to deal with the inaccuracies of the model because the model is often called recursively to make long-term predictions, leading to very large errors. The sources of the inaccuracies are typically inadequate models (linear models applied on non linear systems) and acquisition noise. Two mains approaches have been considered to build the model: (1) Locally Weighted Bayesian Regression for example in Bagnell and Hneider (2001); Ng et al. (2006); Kim et al. (2003) and (2) Gaussian Processes (GPs, Rasmussen and Williams (2006)) in Ko et al. (2007); Deisenroth and Rasmussen (2011); Deisenroth et al. (2011). In the last part of this chapter we will see in details Bayesian Optimization, a model-based optimization algorithm using GPs, as we will use it in the work presented in this manuscript. Consequently, we will not elaborate much more about model-based policy search algorithms in this section.

2.3.2 Application in behavior learning in robotics

Like evolutionary algorithms, policy search algorithms have been applied on a large variety of robotic setups. However, it is important to note that all the experiments presented in this section, like the majority of studies using PS algorithms in robotics, start from an initial policy that is either hand-coded, or learned via a demonstration (imitation learning), and then use the PS algorithm to fine tune the parameters (Deisenroth et al., 2013b).

The eNAC algorithm has been used by Peters and Schaal (2008b) to allow a physical robot to learn how to play baseball. Starting from an initial policy learned by imitation, the robot fine tuned the 70 parameters of its policy and successfully hit the ball (placed on a T-stick) after between 200 and 300 rollouts. The time-step based version of NAC (Peters and Schaal, 2008a) has also been used by Ueno et al. (2006) to control the maintenance phase of a walking gait with a quasi-passive dynamic bipedal robot. The experiments showed that only about 20 trials on the physical robot were required to learn the 4 parameters allowing the robot to walk (starting from initial phase using a hand-tuned controller). In the work of Kormushev et al. (2010), the PoWER algorithm is used to learn to flip pancakes with a physical robotic arm. After 50 rollouts the algorithm found a behavior able to flip pancakes more than 90% of the time.

The ball in the cup is another experiment that has been used several times in the literature to evaluate PS algorithms. Similarly to the previous example, the PoWER algorithm has been used to solve this task with a physical robot after only 75 rollouts (Kober and Peters, 2011). This work has then been extended with a more advanced controller that takes into account the position of the ball (Kober et al., 2008). In this case, the robot needed between 500 and 600 rollouts (in simulation) to find the 91 parameters that allow the robot to achieve the task. The authors report that such long learning time is most probably due to the high variance in the initial state of the ball. The same experiment has been used in Stulp et al. (2014) to assess the performance of a variant of PI^2 (called PI^{BB} for Black-Box functions and which is close to CMA-ES) improved with the ability to discover

"skill options". This algorithm required about 90 trials to discover and distinguish two configurations of the task (two different rope lengths) and then managed to place the ball in the cup.

In Kupcsik et al. (2014), the authors used a variant of the REPS algorithm (named GPREPS, that combines learned probabilistic forward models via GPs and the REPS algorithm, Peters et al. (2010)) to allow a physical robot to play hockey. In this task the robot had to shoot a puck (by hitting it) at a target puck, in order to displace this second puck from a predefined distance. The context of each task is then defined by the x and y position of the target puck and the desired distance of motion. The goal of this experiment is to show that this learning algorithm is able to achieve the task and to generalize over the different contexts (which change between each rollout). After about 80 rollouts, the algorithm inferred the 15 parameters that allow the robot to achieve its task regardless the current context. Like in the previous experiment, the initial policy was obtained through imitation learning.

PS algorithms have also been used to learn walking gaits. For example, Kohl and Stone (2004) used a gradient-based method to optimize the 12 parameters of predefined foot trajectories of an Aibo robot (four-legged robot). The algorithm required 345 rollouts (each being repeated 3 times for accurate evaluations) to perform the optimization. Similarly to the other experiments presented in this section, the algorithm starts from a walking policy and is only used to increase its velocity. With a similar technique, Geng et al. (2006) tuned the two parameters of a artificial neural network, which controls the gait of a physical biped robot. The question of walking gait learning has also been addressed with reinforcement learning methods. For example in Kimura et al. (2001), the authors used an actor critic with a TD learning approach to learn a walking gait for a four-legged robot. About 10,000 robot's steps were required to learn a working gait. A stochastic gradient descent has also been used in Tedrake et al. (2005) to learn a walking gait on a bipedal robot. The 46 parameters of the controller have been optimized in about 950 steps of the bipedal robot (performed in about 20 minutes). In these two last experiments, the difference between episodic-based (rollouts) and time-step based evaluation is very tight as a robot's step is both a full episode and a time-step, making the comparison with the other approaches difficult.

2.3.3 Partial conclusion

In the beginning of this section, we highlighted the similarities between policy search and evolutionary algorithms. We can see through the different examples of robotic applications of PS algorithms that these two families of algorithms are used in similar purposes. However, it is interesting to note that the number of trials required solve the task is significantly lower in experiments using PS algorithms than in those using EAs (see table 2.1). For instance, the previous examples required most of the time about a few hundred rollouts, while evolutionary algorithms require more likely about several thousands of them.

One possible explanation is that all PS applications start from an initial policy obtained thanks to an imitation learning algorithm. This initial policy is thus already very close to a local (potentially the global) extremum and the PS algorithms are only used to perform a local search to fine tune the policy. This represents an important difference between PS algorithms and EAs. In most of EAs experiments, the initial population of solutions is randomly generated. While faster learning speed may be very beneficial when a robot has to face unexpected situations, we cannot suppose that it will always have access to a teacher or a useful initial policy. For example, in the case where a six-legged robot has to adapt to a mechanical damage (like a broken leg), it is likely that the initial walking gait (using its six legs) will be significantly different from a walking gait that works in the damage condition. In this case, the gait using the six legs will not represent a good initial policy, and the local search ability of PS algorithm may not be enough to find a working solution.

2.4 Bayesian Optimization

2.4.1 Principle

Bayesian optimization is a model-based, global, black-box optimization algorithm that is tailored for very expensive objective functions (like reward or cost functions, Lizotte et al. (2007); Brochu et al. (2010b); Mockus (2013); Snoek et al. (2012); Griffiths et al. (2009); Borji and Itti (2013)). As a black-box optimization algorithm, Bayesian optimization searches for the maximum of an unknown objective function from which samples can be obtained (e.g., by measuring the performance of a robot).

Like all model-based optimization algorithms (e.g. surrogate-based algorithms (Booker et al., 1999; Forrester and Keane, 2009; Jin, 2011), kriging (Simpson et al., 1998), or DACE (Jones et al., 1998; Sacks et al., 1989)), Bayesian optimization creates a model of the objective function with a regression method, uses this model to select the next point to acquire, then updates the model and repeats the process.

It is called *Bayesian* because, in its general formulation (Mockus, 2013), this algorithm chooses the next point by computing a posterior distribution of the objective function ($P(M \mid E)$) using the likelihood of the data already acquired ($P(E \mid M)$) and a prior on the type of function ($P(M)$).

$$P(M \mid E) \propto P(E \mid M)P(M)$$

The name Bayesian Optimization refers to a generic framework that follows this equation but several different implementations exist. In the rest of this section we will present the main components of this framework and in particular those that are the most commonly used in robotics.

2.4.2 Gaussian Processes

Using Gaussian Process regression to find a model (Rasmussen and Williams, 2006) is a common choice for Bayesian optimization (Calandra et al., 2014; Griffiths et al.,

2009; Brochu et al., 2010b; Lizotte et al., 2007). Gaussian processes (GPs) are particularly interesting for regression because they not only model the cost function, but also the uncertainty associated with each prediction. For a cost function f, usually unknown, a GP defines the probability distribution of the possible values $f(\mathbf{x})$ for each point $\mathbf{x}$. These probability distributions are Gaussian, and are therefore defined by a mean (μ) and a standard deviation (σ). However, μ and σ can be different for each $\mathbf{x}$; we therefore define a probability distribution *over functions*:

$$P(f(\mathbf{x})|\mathbf{x}) = \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{2.6}$$

where $\mathcal{N}$ denotes the standard normal distribution.

To estimate $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$, we need to fit the Gaussian process to the data. To do so, we assume that each observation $f(\chi)$ is a sample from a normal distribution. If we have a data set made of several observations, that is, $f(\chi_1), f(\chi_2), ..., f(\chi_t)$, then the vector $[f(\chi_1), f(\chi_2), ..., f(\chi_t)]$ is a sample from a *multivariate* normal distribution, which is defined by a mean vector and a covariance matrix. A Gaussian process is therefore a generalization of a n-variate normal distribution, where n is the number of observations. The covariance matrix is what relates one observation to another: two observations that correspond to nearby values of χ_1 and χ_2 are likely to be correlated (this is a prior assumption based on the fact that functions tend to be smooth, and is injected into the algorithm via a prior on the likelihood of functions), two observations that correspond to distant values of χ_1 and χ_2 should not influence each other (i.e. their distributions are not correlated). Put differently, the covariance matrix represents that distant samples are almost uncorrelated and nearby samples are strongly correlated. This covariance matrix is defined via a *kernel function*, called $k(\chi_1, \chi_2)$, which is usually based on the Euclidean distance between χ_1 and χ_2 (see the "kernels function" sub-section below).

Given a set of observations $\mathbf{P}_{1:t} = f(\chi_{1:t})$ and a sampling noise σ^2_{noise} (which is a user-specified parameter), the Gaussian process is computed as follows (Brochu et al., 2010b; Rasmussen and Williams, 2006):

$$\begin{aligned}
&P(f(\mathbf{x})|\mathbf{P}_{1:t}, \mathbf{x}) = \mathcal{N}(\mu_t(\mathbf{x}), \sigma^2_t(\mathbf{x})) \\
&\text{where :} \\
&\mu_t(\mathbf{x}) = \mu_0 + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t} - \mu_0) \\
&\sigma^2_t(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1}\mathbf{k} \\
&\mathbf{K} = \begin{bmatrix} k(\chi_1, \chi_1) & \cdots & k(\chi_1, \chi_t) \\ \vdots & \ddots & \vdots \\ k(\chi_t, \chi_1) & \cdots & k(\chi_t, \chi_t) \end{bmatrix} + \sigma^2_{noise} I \\
&\mathbf{k} = \begin{bmatrix} k(\mathbf{x}, \chi_1) & k(\mathbf{x}, \chi_2) & \cdots & k(\mathbf{x}, \chi_t) \end{bmatrix} \\
&\mathbf{P}_{1:t} = \begin{bmatrix} P_1 \\ \vdots \\ P_t \end{bmatrix}
\end{aligned} \tag{2.7}$$

where μ_t is the mean function of the GP after t iterations (corresponding to t trials),

σ_t is the corresponding standard deviation and $\mathbf{x}$ refers to an untested solution.

Implementations of BO using GP models select the next χ to test by selecting the maximum of the *acquisition function*, which balances exploration – improving the model in the less explored parts of the search space – and exploitation – favoring parts that the model predicts as promising (see the "Acquisition Functions" section below). Once the observation is made, the algorithm updates the GP to take the new data into account. In traditional Bayesian optimization, the GP is initialized with a constant mean because it is assumed that all the points of the search space are equally likely to be good. The model is then progressively refined after each observation (see Fig. 2.4).

2.4.2.1 Kernel Functions

The kernel function is the covariance function of the GP. It defines the influence of a solution's performance on the performance and confidence estimations of not-yet-tested solutions in the search space that are nearby to the tested solution. The covariance function is a positive semi-definite function. The Squared Exponential covariance function and the Matérn kernel are the most common kernels for GPs (Brochu et al., 2010b; Snoek et al., 2012; Rasmussen and Williams, 2006). Both kernels are variants of the "bell curve".

Squared Exponential Kernel The Squared Exponential (SE) kernel, also called Gaussian kernel or "Exponentiated Quadratic" kernel has this form:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{1}{2l^2}\|\mathbf{x}_i - \mathbf{x}_j\|^2) \tag{2.8}$$

Like all kernel functions, the maximum value is reached when the distance between the two samples is null and decreases when the distance increases. The l parameter defines the *characteristic length-scale* of the kernel. In other words, it defines the distance at which effects become nearly zero.

This expression of the kernel is *isotropic*, meaning that the effect is invariant in every direction of the search space. It may happen that the observations are more correlated in particular dimensions of the space than in the others. In order to have an *anisotropic* expression of the SE kernel we can use the vector $\mathbf{l}$ that defines different characteristic length-scales for each dimension:

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{1}{2}(\mathbf{x}_i - \mathbf{x}_j)^T \operatorname{Diag}(\mathbf{l})^{-2}(\mathbf{x}_i - \mathbf{x}_j)) \tag{2.9}$$

The values of the hyper-parameter $\mathbf{l}$ are commonly obtained thanks to Automatic Relevance Determination (ARD, see "hyper-parameters fitting" section below). Such tuning process is also able to autonomously disregard some dimension of the search space based on the acquired observations.

The SE kernel is infinitely differentiable, which allows the GPs that use such kernel functions to be infinitely mean-square differentiable. The notion of mean-square differentiation is the generalization of the notion of differentiation for ordinary functions to random processes (Rasmussen and Williams, 2006). This notion

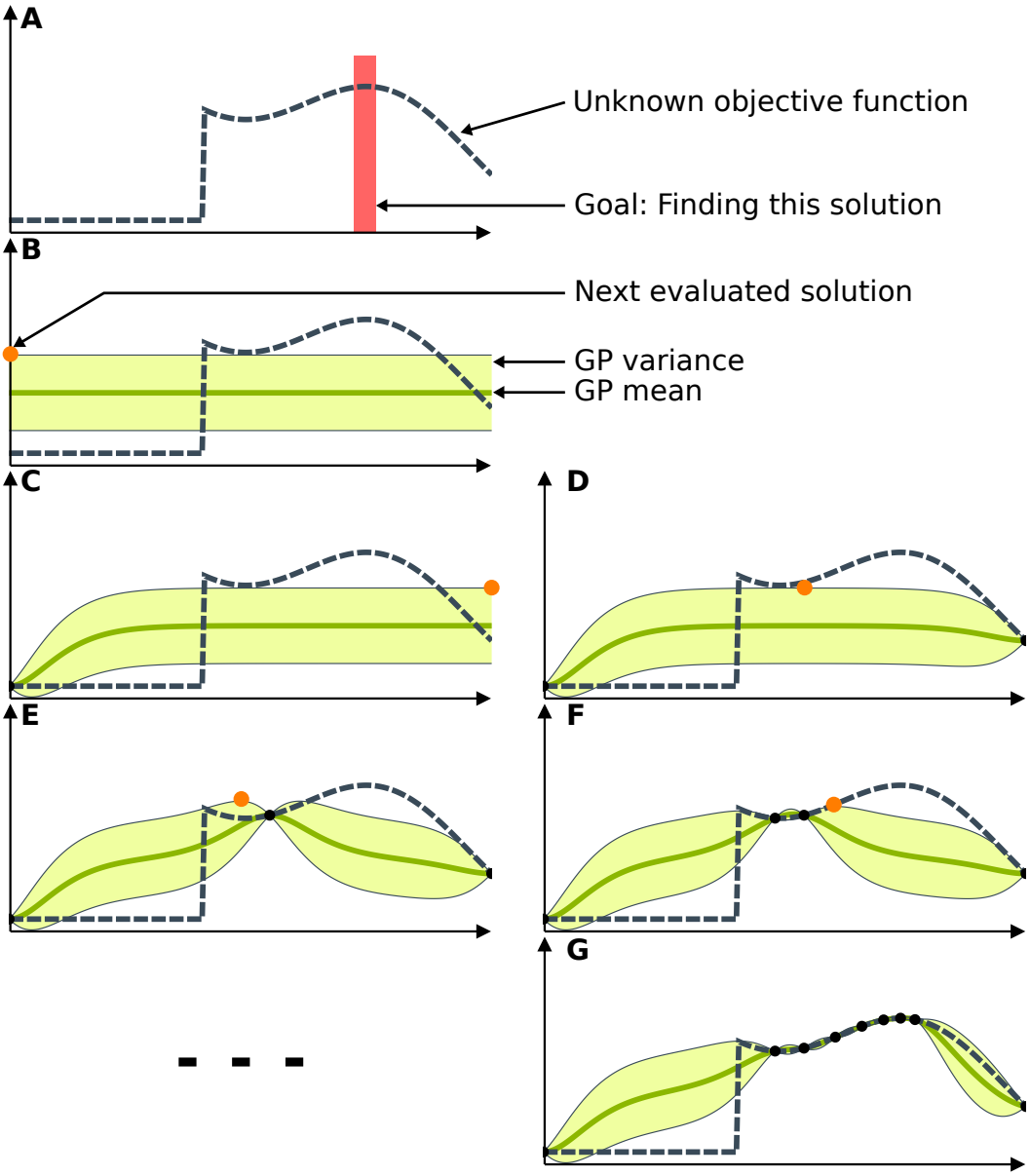


Figure 2.4: Bayesian Optimization of a toy problem. (A) The goal of this toy problem is to find the maximum of the unknown objective function. (B) The Gaussian process is initialized, as it is customary, with a constant mean and a constant variance. (C) The next potential solution is selected and evaluated. The model is then updated according to the acquired data. (D) Based on the new model, another potential solution is selected and evaluated. (E-G) This process repeats until the maximum is reached.

is important when using gradient descent methods to find the maximum of the information acquisition function or to automatically adapt the hyper-parameters of the model.

Matern kernels The Squared Exponential kernel is a very smooth kernel that may be not well suited for modeling some phenomenon like physical processes (Stein, 1999; Rasmussen and Williams, 2006). The Matern class of kernel is more general (it includes the Squared Exponential function as a special case) and allows us to control not only the distance at which effects become nearly zero (as a function of parameter l), but also the rate at which distance effects decrease (as a function of parameter ν). This last parameter allows the user to control the smoothness of the kernel, contrary to the SE kernel.

The Matérn kernel function is computed as follows (Matérn et al., 1960; Stein, 1999):

$$k(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{2^{\nu-1}\Gamma(\nu)}\left(\frac{\sqrt{2\nu}\|\mathbf{x}_i - \mathbf{x}_j\|}{l}\right)^{\nu} \mathbf{H}_{\nu}\left(\frac{\sqrt{2\nu}\|\mathbf{x}_i - \mathbf{x}_j\|}{l}\right) \tag{2.10}$$

where Γ is the gamma function and H_ν is a modified Bessel function (Abramowitz et al. (1966), sec. 9.6). Like the SE kernel, the l parameter controls the characteristic length-scale, while the ν parameter controls the kernel smoothness. Rasmussen and Williams (2006) recommends to use $\nu = 3/2$ or $\nu = 5/2$ for machine learning applications. For these two values, the kernel expression can be simplified as follow:

$$k_{\nu=3/2}(\mathbf{x}_i, \mathbf{x}_j) = \left(1 + \frac{\sqrt{3}\|\mathbf{x}_i - \mathbf{x}_j\|}{l}\right)\exp\left(-\frac{\sqrt{3}\|\mathbf{x}_i - \mathbf{x}_j\|}{l}\right) \tag{2.11}$$

$$k_{\nu=5/2}(\mathbf{x}_i, \mathbf{x}_j) = \left(1 + \frac{\sqrt{5}\|\mathbf{x}_i - \mathbf{x}_j\|}{l} + \frac{5\|\mathbf{x}_i - \mathbf{x}_j\|^2}{3l^2}\right)\exp\left(-\frac{\sqrt{5}\|\mathbf{x}_i - \mathbf{x}_j\|}{l}\right) \tag{2.12}$$

When $\nu \to \infty$, the Matern kernel converges to the SE kernel. In contrast to the SE kernel, the Matern kernels are not infinitely differentiable but only k-times differentiable while $\nu > k$ (Rasmussen and Williams, 2006) and become infinitely differentiable when $\nu \to \infty$.

Several other kernel functions exist, each with different properties and parameters, we encourage interested readers to refer to Rasmussen and Williams (2006) for more details. The selection of the most appropriate kernel function is a question that requires lot of experimentation and engineering, but can also be solved using automatic model selection (Brochu et al., 2010a; MacKay, 1992). The experiments presented in this manuscript that use Bayesian Optimization will be based on Matern kernels or the SE kernel when the hyper-parameters will be automatically determined as presented in the next section.

2.4.2.2 Hyper-parameter fitting

All the proposed kernel functions have parameters that control their different properties. For example, the most common parameter is the length-scale. This parameter is one of the most critical parameters as it controls the impact size of the update after a new observation. A too small length-scale parameter will limit the model's ability to spread the new information over the search space and as a consequence the algorithm will require more samples to update its model over all the search space. Conversely a too large length-scale may propagate knowledge in regions that are irrelevant. For example, if the modeled function is rough, the acquired information only stands for small area and a large length-scale will be not well adapted for a function that varies quickly.

These different examples shows that ideal parameter values that work in any situation do not exist, conversely the parameter values often depend on the considered applications. Resulting from this observation, a promising way to deal with this problem is to allow the algorithm to autonomously adapt these parameters according to the acquired data. In the following sections we will present different methods to automatically tune the hyper-parameters of the model.

2.4.2.3 Marginal log Likelihood

One of the most common methods to automatically tune the model's hyper-parameters is to maximize the marginal log likelihood of the model according to the data (Rasmussen and Williams, 2006):

$$\log p(\mathbf{P}_{1:t}|\chi_{1:t},\boldsymbol{\theta}) = -\frac{1}{2}\mathbf{P}_{1:t}^{\intercal}\mathbf{K}^{-1}\mathbf{P}_{1:t} - \frac{1}{2}\log|\mathbf{K}| - \frac{n}{2}\log(2\pi) \tag{2.13}$$

This expression is composed of three terms. The first term penalizes the likelihood score if the model does not fit the observations ($\mathbf{P}_{1:t}$). The second term, $\frac{1}{2}\log|\mathbf{K}|$, limits the model's complexity and depends only on the kernel function. The last term is a normalization constant. We encourage interested readers to refer to Rasmussen and Williams (2006) for the details about how this expression is obtained.

A very common way to optimize the marginal log likelihood consists in using gradient-based methods. The expression of the marginal log likelihood can be differentiated according to the hyper-parameters using this expression (Rasmussen and Williams, 2006):

$$\begin{aligned}\frac{\partial}{\partial\boldsymbol{\theta}_i}\log p(\mathbf{P}_{1:t}|\chi_{1:t},\boldsymbol{\theta}) &= \frac{1}{2}\mathbf{P}_{1:t}^{\intercal}\mathbf{K}^{-1}\frac{\partial\mathbf{K}}{\partial\boldsymbol{\theta}_i}\mathbf{K}^{-1}\mathbf{P}_{1:t} - \frac{1}{2}\mathrm{tr}(\mathbf{K}^{-1}\frac{\partial\mathbf{K}}{\partial\boldsymbol{\theta}_i}) \\ &= \frac{1}{2}\mathrm{tr}\left((\alpha\alpha^{\intercal}-\mathbf{K}^{-1})\frac{\partial\mathbf{K}}{\partial\boldsymbol{\theta}_i}\right) \quad \text{where} \quad \alpha = \mathbf{K}^{-1}\mathbf{P}_{1:t}\end{aligned} \tag{2.14}$$

The term $\frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_i}$ can be computed with the derivative of the kernel function. Using kernel functions that are at least 1-time differentiable allows us to employ optimization algorithms that use the exact expression of the gradient, like the RPROP algorithm (Riedmiller and Braun, 1993; Blum and Riedmiller, 2013).

One may observe that the equation of the marginal log likelihood (equation 2.13) imposes a trade-off between the model complexity and the ability of the model to fit the data. It is also very surprising to see that this equation is not homogeneous, the first term corresponds to the squared unit of the observation, while the other terms do not have particular units. Consequently, the trade-off obtained by the likelihood optimization can be completely different according to the scale of the observations. For example if the observation are recorded in meters or in centimeters, the first term will be in one case 10,000 more important than in the other case.

To our knowledge, no automatic normalization procedure is used in the main Gaussian Processes library (`http://www.gaussianprocess.org`) and the data from their examples are not normalized (only centered). Consequently, from our point of view, the trade-off imposed by the likelihood expression and the data does not always lead to the model with the best generalization abilities.

2.4.2.4 Cross-Validation

A typical way to improve the generalization abilities of the classifier or models in general is to select the hyper-parameters thanks to *cross-validation*. Such approach is very common in machine learning to prevent over-training of neural networks for example (Kohavi et al., 1995).

The same approach can be used to select the hyper-parameters of Gaussian Processes. For example, the application of the *Leave One Out* cross validation (LOO-CV, Lachenbruch and Mickey (1968); Stone (1974)) for hyper-parameter optimization is discussed in Rasmussen and Williams (2006). This approach consists in training the GP with all the acquired data but one and then computing the log predictive probability, which represents the ability of the model to predict the leaved out sample. This procedure is then repeated for all the samples and all the log probability are summed to compute the LOO log predictive probability (also called *log pseudo-likelihood*). This term can be considered as a performance estimator and can then be optimized.

The most expensive part of this procedure is to compute the inverted kernel matrix for all the leaved out samples, but several mathematical properties allow the complexity to be reduced (like the block-wise matrix inversion theorem, Press et al. (1996)). Moreover, the derivative of the log pseudo-likelihood can be analytically computed with respect to the model's hyper-parameters (Rasmussen and Williams, 2006). Consequently, gradient based optimization methods can be used similarly to the marginal log likelihood optimization. According to Rasmussen and Williams (2006), the computational costs of the LOO cross validation and of the marginal log likelihood optimization are roughly identical and knowing under which

circumstances each method might be preferable is still an open question (Bachoc, 2013).

2.4.3 Acquisition function

The acquisition function selects the next solution that will be evaluated on the system. The selection is made by finding the solution that maximizes the acquisition function. This step is another optimization problem, but does not require testing the solution on the system. In general, for this optimization problem we can derive the exact equation and find a solution with gradient-based optimization (Fiacco and McCormick, 1990).

Several different acquisition functions exist, such as the probability of improvement, the expected improvement, or the Upper Confidence Bound (Brochu et al., 2010b). In the following sections, we will introduce these three acquisition functions and present how the exploitation-exploration trade-off can be handled with their parameters.

2.4.3.1 Probability of improvement

This acquisition function, proposed by Kushner (1964), selects potential solutions that have a high probability of being higher-performing than the highest point already acquired. This probability of improvement (PI) is computed according to the normal cumulative distribution function (Φ) and the difference between the predicted performance of the considered point ($\mathbf{x}$) and the maximum performance obtained up to now. The variance is also taken into account.

$$\begin{aligned} \mathrm{PI}(\mathbf{x}) &= P(f(\mathbf{x}) \geq f(\mathbf{x}^+) \\ &= \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\right) \end{aligned} \tag{2.15}$$

With a high performance difference and a small variance, the probability will be very high. Conversely, if the difference is low (or negative) or if the variance is very high, the probability will be lower. However, this acquisition function is based on pure exploitation of the predicted performance. For example, very small improvements with low variance will have a higher probability of improvement than larger improvements with high variance. As a consequence, the algorithm will select mainly solutions that are close to the best found so far.

One way to mitigate this behavior consists to add a trade-off parameter (ξ) in the equation:

$$\begin{aligned} \mathrm{PI}(\mathbf{x}) &= P(f(\mathbf{x}) \geq f(\mathbf{x}^+) + \xi) \\ &= \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})}\right) \end{aligned} \tag{2.16}$$

The value of this parameter is set by the user to control the exploitation/exploration trade-off of the algorithm. A high value will foster solutions that have a significantly better (predicted) performance that the best one found so far.

This trade-off parameter can also be changed during the optimization process. For example, a high value can be used in the beginning to foster the exploration of the search space and then, near the end of the optimization, a lower value can be set to focus the exploitation of the generated model to fine-tune the obtained solution (Kushner, 1964).

One may notice that this information acquisition function only considers the probability of improvement but the magnitude of the improvement may be of interest too.

2.4.3.2 Expected improvement

As an alternative, Mockus et al. (1978) defined the improvement of the performance observations:

$$I(\mathbf{x}) = \max(0, f_{t+1}(\mathbf{x}) - f(\mathbf{x}^+)) \tag{2.17}$$

The authors propose to consider the expected improvement as an acquisition function. The improvement is equal to zero when the performance of the solution tested at $t+1$ is lower than the high performance found so far. The improvement is positive when the performance difference is also positive.

Based on this definition of the improvement and on the integral over the probability density provided by the GP, the expected improvement (EI) can be analytically computed (see Mockus et al. (1978); Brochu et al. (2010b) for more details):

$$\begin{aligned} \mathrm{EI}(\mathbf{x}) &= \mathbb{E}[\max(0, f_{t+1}(\mathbf{x}) - f(\mathbf{x}^+))|\mathbf{P}_{1:t}, \chi_{1:t}] \\ &= \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+))\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & if \quad \sigma(\mathbf{x}) > 0 \\ 0 & if \quad \sigma(\mathbf{x}) = 0 \end{cases} \\ &\quad \text{where} \qquad Z = \frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})} \end{aligned} \tag{2.18}$$

where ϕ is the probability density function and Φ is the cumulative distribution function of the standard normal distribution.

In the same way as the PI function, the EI acquisition function can be extended with a parameter ($\xi > 0$) that controls the exploitation-exploration trade-off:

$$\begin{aligned} \mathrm{EI}(\mathbf{x}) &= \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi)\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & if \quad \sigma(\mathbf{x}) > 0 \\ 0 & if \quad \sigma(\mathbf{x}) = 0 \end{cases} \\ &\quad \text{where} \qquad Z = \frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})} \end{aligned} \tag{2.19}$$

The influence of this parameter has been discussed in Lizotte (2008). In its PhD-thesis, the author considered a progressively decreasing ξ value according to the number of performed acquisitions, however the experimental results showed no evidence of significant improvement. The author underlines that these experiments are not a proof that such parameter adaptation cannot improve the performance but they show that the improvement is not systematic and such procedure should be used carefully.

2.4.3.3 Upper Confidence Bound

Another information acquisition function that is used commonly is the Upper-Confidence Bound function (UCB, Cox and John (1997)) defined with this equation:

$$\text{UCB}(\mathbf{x}) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}) \tag{2.20}$$

where κ is a user-defined parameter that tunes the trade-off between exploration and exploitation like in the other acquisition functions. In the UCB function, the emphasis on exploitation vs. exploration is explicit and easy to adjust. The UCB function can be seen as the maximum value (argmax) across all solutions of the weighted sum of the expected performance (mean of the Gaussian, $\mu_t(\mathbf{x})$) and of the uncertainty (standard deviation of the Gaussian, $\sigma_t(\mathbf{x})$) of each solution. This sum is weighted by the κ factor. With a low κ, the algorithm will choose solutions that are expected to be high-performing. Conversely, with a high κ, the algorithm will focus its search on unexplored areas of the search space that may have high-performing solutions.

Similarly to the other acquisition functions, several works studied the influence of adapting the κ parameter according to the number of acquisitions. For example, Srinivas et al. (2009) proposed a reduction rate of κ that has a high probability of minimizing the regret of the acquisitions.

2.4.4 Application in behavior learning in robotics

Even if BO has been introduced recently, it has been used for a relatively large variety of tasks in robotics. For example in path planning, BO has been used to select next waypoints that minimize the robot uncertainty about its position (Martinez-Cantin et al., 2007, 2009) or to minimize the vibration on the robot when traveling on an uneven terrain (Souza et al., 2014). This approach has also been used in grasping tasks (Veiga and Bernardino, 2013) and in object recognition tasks (Defretin et al., 2010). This last example employs the Efficient Global Optimization algorithm (EGO, (Jones et al., 1998)[2] in order to select the next view point that will provide the most of information.

The applications of BO to learn behaviors are also numerous. For example, Frean and Boyle (2008) used BO to learn the parameters of a neural network that controls a (simulated) double pole balancing. In this work, the neural network grows in complexity when the performances stagnate by adding additional neurons in the hidden layer. This has been done to avoid bottlenecks coming from simple neural networks and also to bootstrap the learning process. The algorithm required about 200 trials to generate neural networks able to balance the pole, but most of the produced networks used a simple architecture with only one or two hidden neurons (corresponding to 8 or 16 parameters to tune). The authors also compared the performance of their algorithm to the NEAT algorithm on the same task. This

[2]EGO is very close in its application to BO and corresponds to the Kriging algorithm (Simpson et al., 1998) coupled with a GP and the EI acquisition function

evolutionary algorithm introduced by Stanley and Miikkulainen (1996) and designed to augment progressively the topology of neural networks required about 3578 trials on the same task. This is a good illustration of the speed of BO techniques compared to evolutionary algorithms.

As another example, Kuindersma et al. (2011) used BO to optimize the 4 parameters of a controller that moves the arms of a small, physical, wheeled "humanoid" robot (which can be considered as a cart-pole with two arms) that allows the robot to recover after an external shock. The results show that after between 30 and 35 trials the algorithm found controllers that allow the robot to recover its balance and also to reduce the total energy expenditure in this manoeuvre.

In Tesch et al. (2011), the authors used EGO to learn snake gaits that are able to climb up slopes and to deal with obstacles in less than 30 evaluations. In addition to dealing with these environments, the experiments showed that the gaits learned allowed the physical robotic snake to walk faster than with a hand-tuned controller. This work has then been extended to multi-objective optimization in order to simultaneously control both the speed and stability of the robot (Tesch et al., 2013). After 25 evaluations, the algorithm was able to approximate the pareto-front and to generate several behaviors corresponding to speed and stability trades-off.

BO has also been used to generate walking gaits on physical legged robots. For example, Lizotte et al. (2007) employed this algorithm to optimize the 15 parameters of a parametrized gait of the AIBO robot. It required about 120 iterations to produce a gait that is both efficient and smooth. Their work showed that BO is significantly higher performing than gradient-based PS approaches. In a similar way, Calandra et al. (2014) used BO to learn the 4 parameters of the finite state machine controlling a bipedal robot. The experiments required 40 evaluations (each being repeated 3 times for robust observations) to find high performing gaits.

In Kober et al. (2012), the authors do not use BO exactly, but the proposed method employs the same tools as BO and is a good illustration of the potential applications of the BO framework in a more general context. The proposed algorithm learns a mapping between the current state of the system and the meta-parameters that have to be used to achieve a task. This mapping is obtained thanks to modified GP that the authors named Cost-regularized Kernel Regression. The input space of the GP is the current state of the system, the output space represents the distribution over the optimal meta-parameters and the variance of the GP represents the cost of the best solution, found so far, for this state. This contrasts with the standard way of using GP in BO in which the input space corresponds to the controller parameters (sometimes extended to the current state, see Deisenroth and Rasmussen (2011)) and the output space models the distribution of the performance. The BO optimization process aims to find the location in the input space that maximizes the performance of the robot, while in the proposed methods the algorithm aims to find, for a fixed input (state), the output of the GP that minimizes the cost function. The mapping proposed by the authors allows the robot to adjust its policy continuously according to its state. The approach has been

successfully tested on a large variety of tasks (throwing darts and balls to different targets, playing table tennis) and on several physical robots. The performances of this algorithm cannot be directly compared with the previously presented algorithms because it searches, not only for a single solution, but for one solution per state and also because it starts with an initial behavior learned by imitation. Nonetheless, it is worth noting that it manages to find these solutions in only 70 to 250 evaluations, depending on the task.

The PILCO framework is another approach more or less related to BO, introduced by Deisenroth and Rasmussen (2011), that uses a GP to model the dynamic function of the robot and then uses this model to "simulate" the robot's states for several steps ahead and updating its policy according to these predictions. The policy is then executed for one step on the robot and the action's consequences are recorded to update the GP (the dynamic model) and the process repeats until reaching the target. The authors applied this framework on several physical robots, like a cart-pole (Deisenroth and Rasmussen, 2011) and a robotic arm (Deisenroth et al., 2011, 2015). In both of these systems, the algorithm was able to find a behavior solving the task in about a dozen of trials, which can be considered as the state of the art for these relatively simple systems. However, according to the authors (Deisenroth and Rasmussen, 2011), the framework has some limitations in the discovered policy as there is no guarantees of global optimality and that the obtained solutions are mainly locally optimal, but these limitations are commonly shared with most of the policy search methods and EAs. Moreover, applying this framework on more complex systems, like legged robots, seems not trivial. For example, we can wonder if it is possible to build models of complex interactions, like those between the ground and the robot's legs, in only a dozen of trials and to use them when the state space is very large.

2.4.5 Partial conclusion

Bayesian Optimization is a promising model-based policy search method for several reasons: BO is able (1) to deal with noisy observations and (2) shows similar learning speed than model-free PS algorithms (a few hundred trials). More importantly, (3) BO does not require starting from a good initial policy (like those obtained via imitation), which constitute a fundamental difference between BO and PS algorithms. None of the BO applications presented in this section require such initialization procedure. This shows the global optimization ability of the BO algorithm, even if it depends on the initialization of the GP, which is most of the time performed through a random sampling. Nevertheless this ability also highly depends on the size of the search space. In large search spaces, a very large number of random samples will be required to obtain a beneficial initial model of the performance function. This limitation explains why most of the applications presented previously use relatively small search spaces (a maximum of a dozen of parameters).

Unfortunately, this limitation on the size of the search space may also limit the adaptation abilities of the robots, as the search space should contain solutions that

work in spite of the damage. Small search spaces are likely to reduce the number of behavior families that the robot may use and consequently reduce the number of situations in which the robot is able to adapt.

2.5 Conclusion

In this chapter, we presented three families of learning algorithms: Evolutionary algorithms (EAs), Policy Search algorithms (PS) and Bayesian Optimization (BO). We highlighted that each of these families has its own advantages and drawbacks. EAs are creative optimization algorithms that are able to deal with large search spaces and to evolve a large variety of solution types (like parameter vectors, graph or shapes). They are also less affected by local optima than the other approaches thanks to methods like the behavioral diversity. The main downside of EAs is undoubtedly the number of evaluations required by the optimization algorithm, which makes very challenging all EA applications on physical robots.

PS algorithms allow robots to learn behaviors significantly faster than EAs (about one order of magnitude faster), but this learning speed is mainly explained by the fact PS algorithms are mainly local search approaches, needing good initial policy. A good illustration of this aspect of PS algorithms is the fact that most the experiences we are aware of used an initial policy obtained thanks to an imitation learning algorithm and a human demonstration. This kind of initialization procedure cannot be used in every situation. For example, if a robot on a remote planet has to learn a new behavior to adapt to an unexpected situation, a human demonstration is unlikely to be available.

In the last section we saw that BO finds solutions as fast as PS algorithms. However, the fact that this algorithm does not need to be initialized with an imitation algorithm proves that it is less affected by local extrema than PS algorithms. Unfortunately, this ability is tempered by the limited size of the search spaces, which rarely exceeds a dozen dimensions. Such small search spaces are likely to limit the diversity of behaviors that the robot will potentially be able to use to adapt. With large search spaces, the number of different behavior families (given a type of controller) will be larger than with small search spaces. Consequently, to allow robots to face a large variety of situations the search space should be as large as possible.

Taking everything into account, we can conclude that an ideal learning algorithm that might allow robots to adapt will require the creativity of EAs and the learning speed of PS and BO algorithms. The objective of the work presented in this manuscript is thus to combine the advantages of the three algorithm families and alleviate all their drawbacks. The main idea consists in encapsulating the creativity of evolutionary algorithms in a container that we call a behavioral repertoire or a behavioral map, and then use this condensed source of creativity with BO to profit of its learning speed. The behavioral repertoires are autonomously designed in simulation while they are used on physical robots for the adaptation. The links

and differences between the reality and the simulation (called the reality gap problem, Koos et al. (2013b)) play an important role in this manuscript, as we will show that using information provided by the simulation is one of the keys to make fast adaptation possible with physical robots.

In the next chapter we will present in details the concept of behavioral repertoires and we will introduce two new algorithms to create these repertoires. We will then show in the chapter 4 how these repertoires can be combined with BO to allow fast adaptation to mechanical damages. Before the conclusion and the discussion, we will present in the chapter 5, how these results can be extended to deal with usual problems and questions in robotics. More precisely we will see how it is possible to adapt a whole family of behaviors by transferring knowledge from one task to the next ones. We will also deal with the problem of solutions that cannot be evaluated (for safety reasons for instance) or that failed to be assessed. Finally, we will see how to handle behavioral repertoires that contain inconsistency according to the reality and that may be more misleading than useful.

Chapter 3

Behavioral Repertoire

The results and text of this chapter have been partially published the following articles.

Main articles:

- **Cully, A.,** and Mouret, J. B. (2015). Evolving a behavioral repertoire for a walking robot. Evolutionary computation.
- **Cully, A.**, and Mouret, J. B. (2013). Behavioral repertoire learning in robotics. In Proceedings of the 15th annual conference on Genetic and evolutionary computation (pp. 175-182). ACM.

Related articles:

- Maestre, C., **Cully, A.**, Gonzales, C., and Doncieux, S. (2015, August). Bootstrapping interactions with objects from raw sensorimotor data: a Novelty Search based approach. In IEEE International Conference on Developmental and Learning and on Epigenetic Robotics.

Other contributors:

- Jean-Baptiste Mouret, Pierre and Marie Curie University (Thesis supervisor)

Author contributions:

- for the both of the papers: **A.C.** and J.-B.M. designed the study. **A.C.** performed the experiments. **A.C.** and J.-B.M. analyzed the results and wrote the papers.

Contents

3.1 Introduction

As presented in the previous chapter, the creativity of evolutionary algorithms stems from their ability to deal with large search spaces. Nevertheless, while this creativity can allow robot to find solutions in a large number of damage situations, the large number of evaluations often counterbalances their benefits and makes these algorithms unemployable with physical robots. During most of evolutionary processes, we can observe that most of the evaluations lead to degenerated behaviors that are unlikely to work, even on the intact robot. Conversely, some solutions that work on the intact robot can still work once the robot is damaged. For example, behaviors that do not use the damaged part are likely to perform similarly on the intact robot and when it is damaged. Consequently, a promising way to save evaluations could be to avoid such degenerated behaviors and to focus on solutions that work at least on the intact robot.

In the rest of this manuscript, we will hypothesize that such behaviors exist. Focusing the exploration of the search space on behaviors that work on the intact robot implies that our algorithms know how to find these behaviors in advance. However, determining all the behaviors that work on a robot, intact or damaged, represents a challenge that remains open.

Numerous algorithms have been proposed to allow robots to learn behaviors (see chapter 2). However, the vast majority of these algorithms is devised to learn a single behavior, like walking in a straight line or reaching a single goal. In this chapter, we introduce two novel evolutionary algorithms that attempt to discover many possible behaviors for a robot by learning simultaneously and autonomously several hundreds or even thousands of simple behaviors in a single learning process. Walking robots are a good illustration of systems that can perform several types of behaviors. For example, they can walk in every direction or they can walk in very different manners (in a static or dynamic way, by hopping or by using differently their legs). We will use this kind of system as an application example in this chapter and propose methods that will allow legged robots to learn a large variety of behaviors, like walking in every direction, in only one learning process.

3.1.1 Evolving Walking Controllers

Evolving gaits for legged robots has been an important topic in evolutionary computation for the last 25 years (de Garis, 1990; Lewis et al., 1992; Kodjabachian and Meyer, 1998; Hornby et al., 2005; Clune et al., 2011; Yosinski et al., 2011; Samuelsen and Glette, 2014). That legged robots is a classic of evolutionary robotics is not surprising (Bongard, 2013): legged locomotion is a difficult challenge in robotics that evolution by natural selection solved in nature; evolution-inspired algorithms may do the same for artificial systems. As argued in many papers, evolutionary computation could bring many benefits to legged robotics, from making it easier to design walking controllers (e.g., Hornby et al. (2005)), to autonomous damage recovery (e.g., Bongard et al. (2006); Koos et al. (2013a)). In addition, in an embodied cognition perspective (Wilson, 2002; Pfeifer and Bongard, 2007; Pfeifer et al.,

2007), locomotion is one of the most fundamental skills of animals, and therefore it is one of the first skills needed for embodied agents.

It could seem more confusing that evolutionary computation has failed to be central in legged robotics, in spite of the efforts of the evolutionary robotics community (Raibert, 1986; Siciliano and Khatib, 2008). In our opinion, this failure stems from at least two reasons: (1) most evolved controllers are almost useless in robotics because they are limited to walking in a straight line at constant speed (e.g. Hornby et al. (2005); Bongard et al. (2006); Koos et al. (2013a)), whereas a robot that only walks in a straight line is obviously unable to accomplish any mission; (2) evolutionary algorithms typically require evaluating the fitness function thousands of times, which is very hard to achieve with a physical robot.

We call *Walking Controller* the software module that rhythmically drives the motors of the legged robot. We distinguish two categories of controllers: *un-driven controllers* and *inputs-driven controllers*. An un-driven controller always executes the same gait, while an inputs-driven controller can change the robot's movements according to an input (e.g. a speed or a direction reference). Inputs-driven controllers are typically combined with decision or planning algorithms (Russell et al., 2010; Currie and Tate, 1991; Dean and Wellman, 1991; Kuffner and LaValle, 2000) to steer the robot. These two categories contain, without distinctions, both open-loop and closed-loop controllers, and can be designed using various controller and genotype structures. For example, walking gait evolution or learning has been achieved on legged robots using parametrized periodic functions (Koos et al., 2013a; Chernova and Veloso, 2004; Hornby et al., 2005; Tarapore and Mouret, 2014a,b), artificial neural networks with either direct or generative encoding (Clune et al., 2011; Valsalam and Miikkulainen, 2008; Tarapore and Mouret, 2014a,b), Central Pattern Generators (Kohl and Stone, 2004; Ijspeert et al., 2007), or graph-based genetic programming (Filliat et al., 1999; Gruau, 1994).

When dealing with *physical* legged robots, the majority of studies only considers the evolution of un-driven walking controllers and, most of the time, the task consists in finding a controller that maximizes the forward walking speed (Zykov et al., 2004; Chernova and Veloso, 2004; Hornby et al., 2005; Berenson et al., 2005; Yosinski et al., 2011; Mahdavi and Bentley, 2006). Papers on alternatives to evolutionary algorithms, like policy gradients (Kohl and Stone, 2004; Tedrake et al., 2005) or Bayesian optimization (Calandra et al., 2014; Lizotte et al., 2007), are also focused on robot locomotion along a straight line.

Comparatively few articles deal with controllers able to turn or to change the walking speed according to an input, especially with a physical robot. Inputs-driven controllers usually need to be tested on each possible input during the learning process or to be learned with an incremental process, which significantly increases both the learning time and the difficulty compared to learning an un-driven controller. Filliat et al. (1999) proposed such a method that evolves a neural network to control a hexapod robot. Their neural network is evolved in several stages: first, the network is evolved to walk in a straight line; in a second stage, a second neural network is evolved on top of the walking controller to be able to execute turning

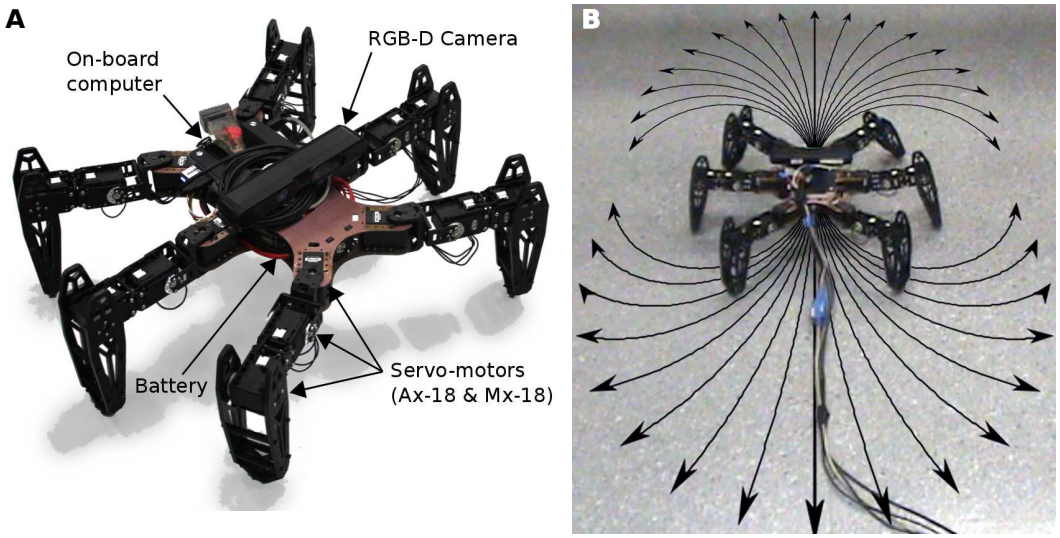


Figure 3.1: (Left) The hexapod robot. It has 18 degrees of freedom (DOF), 3 for each leg. Each DOF is actuated by position-controlled servos (Dynamixel actuators). A RGB-D camera (Asus Xtion) is screwed on the top of the robot. The camera is used to estimate the forward displacement of the robot thanks to a RGB-D Simultaneous Localization And Mapping (SLAM, Durrant-Whyte and Bailey (2006); Angeli et al. (2009)) algorithm (Endres et al., 2012) from the ROS framework (Quigley et al., 2009). (Right) Goal of TBR-Learning. Our algorithm allows the hexapod robot to learn to walk in every direction with a single run of the evolutionary algorithm.

manoeuvres. In a related task (flapping-wing flight), Mouret et al. (2006) used a similar approach, in which an evolutionary algorithm is used to design a neural network that pilots a simulated flapping robot; the network was evaluated by its ability to drive the robot to 8 different targets and the reward function was the sum of the distances to the targets.

Overall, many methods exist to evolve un-driven controllers, while methods for learning inputs-driven controllers are very time-expensive, difficult to apply on a physical robot, and require an extensive amount of expert knowledge. To our knowledge, no current technique is able to make a physical robot learning to walk in multiple directions in less than a dozen hours. In this paper, we sidestep many of the challenges raised by input-driven controllers while being able to drive a robot in every direction: we propose to abandon input-driven controllers, and, instead, search for a large number of simple, un-driven controllers, one for each possible direction.

3.1.2 Evolving behavioral repertoires

In this chapter we propose to see the question of walking in every direction as a problem of learning how to do many different – but related – tasks. Indeed, some

complex problems are easier to solve when they are split into several sub-problems. Thus, instead of using a single and complex solution, it is relevant to search for several simple solutions that solve a part of the problem. This principle is often successfully applied in machine learning: mixtures of experts (Jacobs et al., 1991) or boosting (Schapire, 1990) methods train several weak classifiers on different sub-parts of a problem. Performances of the resulting set of classifiers are better than those of a single classifier trained on the whole problem.

The aim of this chapter is to enable the application of this principle to robotics and, particularly, to legged robots that learn to walk. In this case, an evolutionary algorithm could search for a *repertoire of simple controllers* that would contain a different controller for each possible direction. These simple controllers can then be combined with high level algorithms (e.g. planning algorithms, or neural networks like in (Godzik et al., 2003; **?**)) that successively select controllers to drive the robot. Nevertheless, evolving a controller repertoire typically involves as many evolutionary processes as there are target points in the repertoire. Evolution is consequently slowed down by a factor equal to the number of targets. With existing evolution methods, repertoires of controllers are in effect limited to a few targets, because 20 minutes (Koos et al., 2013a) to dozens of hours (Hornby et al., 2005) are needed to learn how to reach a single target.

In this chapter, we present two algorithms, TBR-Evolution and MAP-Elites, that aim to find such a repertoire of simple controllers, but *in a single run*. They are based on a simple observation: with a classic evolutionary algorithm, when a robot learns to reach a specific target, the learning process explores many different potential solutions, with many different outcomes. Most of these potential solutions are discarded because they are deemed poorly-performing. Nevertheless, while being useless for the considered objective, these inefficient behaviors can be useful for other objectives. For example, a robot learning to walk in a straight line usually encounters many turning gaits during the search process, before converging towards straight-line locomotion.

TBR-Evolution The first algorithm presented here, named the Transferability-based Behavioral Repertoire Evolution algorithm (TBR-Evolution), exploits this idea by taking inspiration from the "Novelty Search" algorithm (Lehman and Stanley, 2011a), and in particular its variant the "Novelty Search with Local Competition" (see section 2.2.3). Instead of rewarding candidate solutions that are the closest to the objective, this recently introduced algorithm explicitly searches for behaviors that are different from those previously seen. The local competition variant adds the notion of a quality criterion, which is optimized within each individual's niche. As shown in the rest of this chapter, searching for many different behaviors during a single execution of the algorithm allows the evolutionary process to efficiently create a repertoire of high-performing walking gaits.

To further reduce the time required to obtain a behavioral repertoire for the robot, TBR-Evolution relies on the *transferability approach* (see section 2.2.5),

which combines simulations and tests on the physical robot to find solutions that perform similarly in simulation and in reality. The advantage of the transferability approach is that evolution occurs in simulation but the evolutionary process is driven towards solutions that are likely to work on the physical robot. In recent experiments, this approach led to the successful evolution of walking controllers for quadruped (Koos et al., 2013b), hexapod (Koos et al., 2013a), and biped robots (Oliveira et al., 2013), with no more than 25 tests on the physical robot.

We evaluate this algorithm on two sets of experiments. The first set aims to show that learning simultaneously all the behaviors of a repertoire is faster than learning each of them separately. We chose to perform these experiments in simulation to gather extensive statistics. The second set of experiments evaluates our method on a physical hexapod robot (Fig. 3.1, left) that has to walk forward, backward, and turn in both directions, all at different speeds (Fig. 3.1, right). We compare our results to learning independently each controller. All our experiments utilize embedded measurements to evaluate the fitness, an aspect of autonomy only considered in a handful of gait discovery experiments (Kimura et al., 2001; Hornby et al., 2005).

MAP-Elites The second algorithm introduced in this chapter is named Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) and has initially been created by Mouret and Clune (2015) to generate graphics that illustrate the fitness landscape of the retina problem (Clune et al., 2013). However, it shows itself to be a simple but high-performing algorithm to learn simultaneously a very large variety of behaviors. This algorithm and more specifically the *Behavior-Performance Maps* (which are similar to behavioral repertoires) that it generates will be used in the next chapter to allow our robots to adapt to a large variety of mechanical damages.

We first evaluate the ability of this algorithm to learn large collection of actions on the same task as the TBR-Evolution algorithm, in which a hexapod robot has to learn to walk in every direction. With this experiment, we can see how MAP-Elites compares to our first algorithm. In a second experiment, we investigate the ability of the algorithm to find a large number of different ways to achieve the same task. In our case we ask our robot to learn a maximum of ways to walk in a straight line as fast as possible. Both of these experiments are performed only in simulation, as we will introduce in the next chapter an algorithm that uses the behavioral-performance maps to allow physical robots to learn or to adapt quickly.

In the two following sections, we introduce the TBR-Evolution algorithm and the MAP-Elites algorithm in details and present the experimental results.

3.2 The TBR-Evolution algorithm

3.2.1 Principle

As mentioned previously, learning independently dozens of controllers is prohibitively expensive, especially with a physical robot. To avoid this issue, the

TBR-Evolution algorithm transforms the problem of learning a repertoire of controllers into a problem of evolving a heterogeneous population of controllers. This problem can be solved with an algorithm derived from novelty search with local competition (Lehman and Stanley, 2011b): instead of generating virtual creatures with various morphologies that execute the same behavior, as presented in section 2.2.3, TBR-Evolution generates a repertoire of controller, each executing a different behavior, working on the same creature. By simultaneously learning all the controllers without the discrimination of a specified goal, the algorithm recycles interesting controllers, which are typically wasted with classical learning methods. This enhances its optimizing abilities compared to classic optimization methods.

Furthermore, our algorithm incorporates the transferability approach (Koos et al., 2013b) to reduce the number of tests on the physical robot during the evolutionary process. The transferability approach and novelty search with local competition can be combined because they are both based on multi-objective optimization algorithms. By combining these two approaches, the behavioral repertoire is generated in simulation with a virtual robot and only a few controller executions are performed on the physical robot. These trials guide the evolutionary process to solutions that work similarly in simulation and in reality (Koos et al., 2013a,b).

The minimization of the number of tests on the physical robot and the simultaneous evolution of many controllers are the two assets that allow the TBR-Evolution algorithm to require significantly less time than classical methods.

More technically, the TBR-Evolution algorithm relies on four principles, detailed in the next sections:

- a stochastic, black box, multi-objective optimization algorithm simultaneously optimizes 3 objectives, all evaluated in simulation: (1) the novelty of the gait, (2) the local rank of quality and (3) the local rank of estimated transferability:

$$\text{maximize} \begin{cases} \text{Novelty}(\mathbf{c}) \\ -\text{Qrank}(\mathbf{c}) \\ -\widehat{\text{Trank}}(\mathbf{c}) \end{cases}$$

- the transferability function is periodically updated with a test on the physical robot;
- when a controller is novel enough, it is saved in the *novelty archive*;
- when a controller has a better quality than the one in the archive that reaches the same endpoint, it substitutes the one in the archive.

Algorithm 1 describes the whole algorithm in pseudo-code.

3.2.1.1 Objectives

The novelty objective fosters the exploration of the reachable space. A controller is deemed as novel when the controlled individual reaches a region where none,

Algorithm 1 TBR-Evolution algorithm (G generations, T transfers' period)

procedure TBR-Evolution
  $pop \leftarrow \{c^1, c^2, \ldots, c^S\}$ (randomly generated)
  $archive \leftarrow \emptyset$
  for $g = 1 \rightarrow G$ **do**
    for all controller $\mathbf{c} \in pop$ **do**
      Execution of $\mathbf{c}$ in simulation
    if $g \equiv 0[T]$ **then**
      Transferability Update($\mathbf{c}$)
    for all controller $\mathbf{c} \in pop$ **do**
      Objective Update($\mathbf{c}$)
      Archive Management($\mathbf{c}$)
    Iteration of NSGA-II on pop
  return $archive$

procedure Transferability Update($\mathbf{c}$)
  Random selection of $\mathbf{c}^* \in pop \cup archive$ and transfer on the robot
  Estimation of the endpoint $\mathscr{E}_{real}(\mathbf{c}^*)$
  Estimation of the exact transferability value $\left|\mathscr{E}_{simu}(\mathbf{c}^*) - \mathscr{E}_{real}(\mathbf{c}^*)\right|$
  Update of the approximated transferability function $\hat{\mathscr{T}}$

procedure Objectives Update($\mathbf{c}$)
  $\mathscr{N}(\mathbf{c}) \leftarrow$ The 15 controllers($\in pop \cup archive$) closest to $\mathscr{E}_{simu}(\mathbf{c})$
  Computation of the novelty objective:
    $Novelty(\mathbf{c}) = \frac{\sum_{\mathbf{j} \in \mathscr{N}(\mathbf{c})} \|\mathscr{E}_{simu}(\mathbf{c}) - \mathscr{E}_{simu}(\mathbf{j})\|}{|\mathscr{N}(\mathbf{c})|}$
  Computation of the local rank objectives:
    $Qrank(\mathbf{c}) = |\mathbf{j} \in \mathscr{N}(\mathbf{c}), quality(\mathbf{c}) < quality(\mathbf{j})|$
    $\widehat{\text{Trank}}(\mathbf{c}) = |\mathbf{j} \in \mathscr{N}(\mathbf{c}), \hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) < \hat{\mathscr{T}}(\mathbf{des}(\mathbf{j})|$

procedure Archive Management($\mathbf{c}$)
  if $Novelty(\mathbf{c}) > \rho$ **then**
    Add the individual to $archive$
  $\mathbf{c}_{nearest} \leftarrow$ The controller $\in archive$ nearest to $\mathscr{E}_{simu}(\mathbf{c})$
  if $\hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) > \tau$ and $quality(\mathbf{c}) > quality(\mathbf{c}_{nearest})$ **then**
    Replace $\mathbf{c}_{nearest}$ by $\mathbf{c}$ in the archive
  else if $\hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) > \hat{\mathscr{T}}(\mathbf{des}(\mathbf{c}_{nearest}))$ **then**
    Replace $\mathbf{c}_{nearest}$ by $\mathbf{c}$ in the archive

or few of the previously encountered gaits were able to go (starting from the same point). The novelty score of a controller $\mathbf{c}$ ($Novelty(\mathbf{c})$) is set as the average distance between the endpoint of the current controller ($\mathscr{E}_c$) and the endpoints of controllers contained in its neighborhood ($\mathscr{N}(\mathbf{c})$):

$$Novelty(\mathbf{c}) = \frac{\sum_{\mathbf{j}\in\mathscr{N}(\mathbf{c})} \|\mathscr{E}_{simu}(\mathbf{c}) - \mathscr{E}_{simu}(\mathbf{j})\|}{card(\mathscr{N}(\mathbf{c}))} \tag{3.1}$$

To get high novelty scores, individuals have to follow trajectories leading to endpoints far from the rest of the population. The population will thus explore all the area reachable by the robot. Each time a controller with a novelty score exceeds a threshold (ρ), this controller is saved in an *archive*. Given this archive and the current population of candidate solutions, a neighborhood is defined for each controller ($\mathscr{N}(\mathbf{c})$). This neighborhood regroups the k controllers that arrive closest to the controller $\mathbf{c}$ (the parameters' values are detailed in appendix C.1).

The local quality rank promotes controllers that show particular properties, like stability or accuracy. These properties are evaluated by the quality score ($quality(\mathbf{c})$), which depends on implementation choices and particularly on the type of controllers used (we will detail its implementation in section 3.2.2.1). In other words, among several controllers that reach the same point, the quality score defines which one should be promoted. For a controller $\mathbf{c}$, the rank ($Qrank(\mathbf{c})$) is defined as the number of controllers from its neighborhood that outperform its quality score: minimizing this objective allows the algorithm to find controllers with better quality than their neighbors.

$$Qrank(\mathbf{c}) = card(\mathbf{j} \in \mathscr{N}(\mathbf{c}), quality(\mathbf{c}) < quality(\mathbf{j})) \tag{3.2}$$

The local transferability rank ($\widehat{Trank}(\mathbf{c})$, equation 3.3) works as a second local competition objective, where the estimation of the transferability score ($\hat{\mathscr{T}}(\mathbf{des}(\mathbf{c}))$) replaces the quality score. Like in (Koos et al., 2013b), this estimation is obtained by periodically repeating three steps: (1) a controller is randomly selected in the current population or in the archive and then downloaded and executed on the physical robot, (2) the displacement of the robot is estimated thanks to an embedded sensor, and (3) the distance between the endpoint reached in reality and the one in simulation is used to feed a regression model ($\hat{\mathscr{T}}$, here a support vector machine (Chang and Lin, 2011)). This distance defines the transferability score of the controller. This model maps a behavioral descriptor of a controller ($\mathbf{des}(\mathbf{c})$), which is obtained in simulation, with an approximation of the transferability score.

Thanks to this descriptor, the regression model predicts the transferability score of each controller in the population and in the archive.

$$\widehat{\mathrm{Trank}}(\mathbf{c}) = card(\mathbf{j} \in \mathscr{N}(\mathbf{c}), \hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) < \hat{\mathscr{T}}(\mathbf{des}(\mathbf{j}))) \tag{3.3}$$

3.2.1.2 Archive management

In the original novelty search with local competition (Lehman and Stanley, 2011b), the archive aims at recording all encountered solutions, but only the first individuals

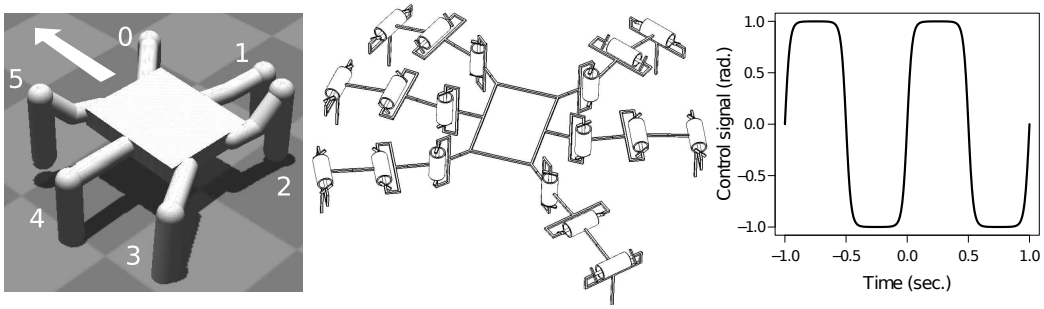


Figure 3.2: (Left) Snapshot of the simulated robot in our ODE-based physics simulator. (Center) Kinematic scheme of the robot. The cylinders represent actuated pivot joints. (Right) Control function $\gamma(t,\ \alpha,\ \phi)$ with $\alpha = 1$ and $\phi = 0$.

that have a new morphology are added to the archive. The next individuals with the same morphology are not saved, even if they have better performances. In the TBR-Evolution algorithm, the novelty archive represents the resulting repertoire of controllers, and thus has to gather only the best controllers for each region of the reachable space.

For this purpose, the archive is differently managed than in the novelty search: during the learning process, if a controller of the population has better scores ($quality(\mathbf{c})$ or $\hat{\mathscr{T}}(\mathbf{c})$) than the closest controller in the archive, the one in the archive is replaced by the better one. These comparisons are made with a priority among the scores to prevent circular permutations. If the transferability score is lower than a threshold (τ), only the transferability scores are compared, otherwise we compare the quality scores. This mechanism allows the algorithm to focus the search on transferable controllers instead of searching efficient, but not transferable, solutions. Such priority is important, as the performances of non-transferable controllers may not be reproducible on the physical robot.

3.2.2 Experimental validation

3.2.2.1 Methods

We evaluate the TBR-Evolution algorithm on two different experiments, which both consist in evolving a repertoire of controllers to access to the entire reachable space of the robot. In the first experiment, the algorithm is applied on a simulated robot (Fig. 3.2, left), consequently the transferability aspect of the algorithm is disabled. The goal of this experiment is to show the benefits of evolving simultaneously all the behaviors of the repertoire instead of evolving them separately. The second experiment applies the algorithm directly on a physical robot (Fig 3.1, left). For this experiment, the transferability aspect of the algorithm is enabled and the experiment shows how the behavioral repertoire can be learned with a few trials on the physical robot.

The pseudo-code of the algorithm is presented in Algorithm 1. The TBR-

Evolution algorithm uses the same variant of NSGA-II (Deb et al., 2002) as the novelty search with local competition (Lehman and Stanley, 2011b), which replaces the diversity mechanism along the non-dominated front with a separate objective explicitly reward the genotypic diversity. The simulation of the robot is based on the *Open Dynamic Engine* (ODE) and the transferability function $\hat{\mathscr{T}}$ uses the ν-Support Vector Regression algorithm with linear kernels implemented in the library *libsvm* (Chang and Lin, 2011) (learning parameters set to default values). All the algorithms are implemented in the Sferes$_{v2}$ framework (Mouret and Doncieux, 2010) (parameters and source code are detailed in appendix). The simulated parts of the algorithms are computed on a cluster of 5 quad-core Xeon-E5520@2.27GHz computers.

The evolved genotype and the corresponding controller used in these experiment are defined in the appendix A.2.1. However, compared to classic evolutionary algorithms, TBR-Evolution only changes the way individuals are selected. As a result, it does not put any constraint on the type of controllers, and many other controllers are conceivable (e.g. bio-inspired central pattern generators (Sproewitz et al., 2008; Ijspeert, 2008), dynamic movement primitives (Schaal, 2003) or evolved neural networks (Yosinski et al., 2011; Clune et al., 2011)).

3.2.2.2 Endpoints of a controller

The endpoint of a controller (in simulation or in reality) is the position of the center of the robot's body projected in the horizontal plane after running the controller for 3 seconds:

$$\mathscr{E}(\mathbf{c}) = \left\{ \begin{array}{l} \text{center}_x(t = 3s) - \text{center}_x(t = 0s) \\ \text{center}_y(t = 3s) - \text{center}_y(t = 0s) \end{array} \right\}$$

When the controller is executed on the physical robot, the location of the robot is assessed thanks to a SLAM algorithm (see Fig.3.1)

3.2.2.3 Quality Score

To be able to sequentially execute saved behaviors, special attention is paid to the final orientation of the robot. Because the endpoint of a trajectory depends on the initial orientation of the robot, we need to know how the robot ends its previous movement when we plan the next one. To facilitate chaining controllers, we encourage behaviors that end their movements with an orientation aligned with their trajectory.

The robot cannot execute arbitrary trajectories with a single controller because controllers are made of simple periodic functions. For example, it cannot begin its movement by a turn and then go straight. With this controller, the robot can only follow trajectories with a constant curvature, but it still can move sideways, or even turn around itself while following an overall straight trajectory. We chose to focus the search on circular trajectories, centered on the lateral axis, with a variable radius (Fig. 3.3A), and for which the robot's body is pointing towards the

tangent of the overall trajectory. Straight, forward (or backward) trajectories are still possible with the particular case of an infinite radius. This kind of trajectory is suitable for motion control as many complex trajectories can be decomposed in a succession of circle portions and lines. An illustration of this principle is pictured on figure 3.3 (D-E-F).

To encourage the population to follow these trajectories, the quality score is set as the angular difference between the arrival orientation and the tangent of the circular trajectory that corresponds to the endpoint (Fig. 3.3B):

$$quality(\mathbf{c}) = -|\theta(\mathbf{c})| = -|\alpha(\mathbf{c}) - \beta(\mathbf{c})| \tag{3.4}$$

3.2.2.4 Transferability score

The transferability score of a tested controller $\mathbf{c}^*$ is computed as the distance between the controller's endpoint reached in simulation and the one reached in reality:

$$\text{transferability}(\mathbf{c}^*) = -|\mathscr{E}_{\text{simu}} - \mathscr{E}_{\text{real}}| \tag{3.5}$$

In order to estimate the transferability score of untested controllers, a regression model is trained with the tested controllers and their recorded transferability score. The regression model used is the ν-Support Vector Regression algorithm with linear kernels implemented in the library *libsvm* (Chang and Lin, 2011) (learning parameters are set to default values), which maps a behavioral descriptor ($\mathbf{des}(\mathbf{c})$) with an estimated transferability score ($\mathscr{T}(\mathbf{des}(\mathbf{c}))$). Each controller is described with a vector of Boolean values that describe, for each time-step and each leg, whether the leg is in contact with the ground (the descriptor is therefore a vector of size $N \times 6$, where N is the number of time-steps). This kind of vector is a classic way to describe gaits in legged animals and robots (Raibert, 1986). During the evolutionary process, the algorithm performs 1 transfer every 50 iterations.

3.2.2.5 Experiments on the Virtual Robot

This first experiment involves a virtual robot that learns a behavioral repertoire to reach every point in its vicinity. The transferability objective is disabled because the goal of this experiment is to show the benefits of learning simultaneously all the behaviors of the repertoire instead of learning them separately. Using only the simulation allows us to perform more replications and to implement a higher number of control experiments. This experiment also shows how the robot is able to *autonomously*:

- discover possible movements;
- cover a high proportion of the reachable space;
- generate a behavioral repertoire.

The TBR-Evolution experiment and the control experiments (described in the next section) are replicated 40 times to gather statistics.

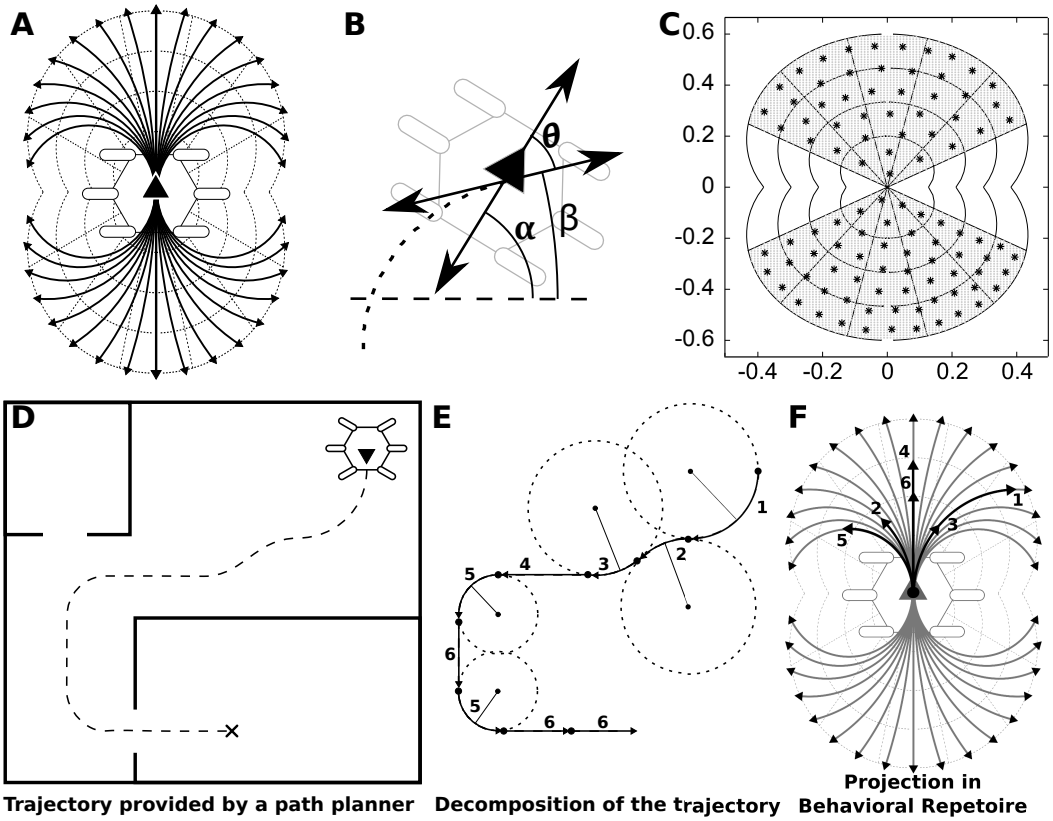


Figure 3.3: (A) Examples of trajectories following a circle centered on the lateral axis with several radii. (B) Definition of the desired orientation. θ represents the orientation error between α, the final orientation of the robot, and β, the tangent of the desired trajectory. These angles are defined according to the actual endpoint of the individual, not the desired one. (C) Reachable area of the robot viewed from top. A region of interest (ROI) is defined to facilitate post-hoc analysis (gray zone). The boundaries of the region are defined by two lines at 60 degrees on each side of the robot. The curved frontier is made of all the reachable points with a curvilinear abscissa lower than 0.6 meters (these values were set thanks to experimental observations of commonly reachable points). Dots correspond to targets selected for the control experiments. (D-E-F) Illustration of how a behavioral repertoire can be used with a hexapod robot. First, a path-planning algorithm computes a trajectory made of lines and portions of circles (LaValle, 2006; Siciliano and Khatib, 2008). Second, to follow this trajectory, the robot sequentially executes the most appropriate behavior in the repertoire (here numbered on E and F). For closed-loop control, the trajectory can be re-computed at each time-step using the actual position of the robot.

Control Experiments To our knowledge, no previous study directly tackles the question of learning simultaneously all the behaviors of a repertoire, thus we cannot compare our approach with an existing method. As a reference point, we implemented a naive method where the desired endpoints are preselected. A different controller will be optimized to reach each different wanted point individually.

We define 100 target points, spread thanks to a K-means algorithm (Seber, 1984) over the defined region of interest (ROI) of the reachable area (see Fig. 3.3C). We then execute several multi-objective evolutionary algorithms (NSGA-II, Deb et al. (2002)), one for each reference point. At the end of each execution of the algorithm, the nearest individual to the target point in the Pareto-front is saved in an archive. This experiment is called "nearest" variant. We also save the controller with the best orientation (quality score described previously) within a radius of 10 cm around the target point and we call this variant "orientation". The objectives used for the optimization are:

$$\text{minimize} \begin{cases} Distance(\mathbf{c}) = \|E_{\mathbf{c}} - E_{Reference}\| \\ Orientation(\mathbf{c}) = |\alpha(\mathbf{c}) - \beta(\mathbf{c})| \end{cases}$$

We also investigate how the archive management added in TBR-Evolution improves the quality of produced behavioral repertoires. To highlight these improvements, we compared our resulting archives with archives issued from the Novelty Search algorithm (Lehman and Stanley, 2011a) and from the Novelty Search with Local Competition algorithm (Lehman and Stanley, 2011b), as the main difference between these algorithms is archive management procedure. We apply these algorithms on the same task and with the same parameters as in the experiment with our method. We call these experiments "Novelty Search"(NS) and "NS with Local Competition". For both of these experiments we will analyze both the produced archives and the resulting populations.

For all the experiments we will study the sparseness and the orientation error of the behavioral repertoires generated by each approach. All these measures are computed within the region of interest previously defined. The sparseness of the archive is computed by discretizing the ROI with a one-centimeter grid ($\mathscr{G}$), and for each point p of that grid the distance from the nearest individual of the archive ($\mathscr{A}$) is recorded. The sparseness of the archive is the average of all the recorded distances:

$$\text{sparseness}(\mathscr{A}) = \frac{\sum_{p \in \mathscr{G}} \min_{i \in \mathscr{A}}(\text{distance}(i,p))}{\text{card}(\mathscr{G})} \tag{3.6}$$

where $\text{card}(\mathscr{G})$ denotes the number of elements in $\mathscr{G}$.

The quality of the archive is defined as the average orientation error for all the individuals inside the ROI:

$$\text{Orientation Error}(\mathscr{A}) = \frac{\sum_{i \in \mathscr{A} \in ROI} \theta(i)}{\text{card}(\mathscr{A} \in \text{ROI})} \tag{3.7}$$

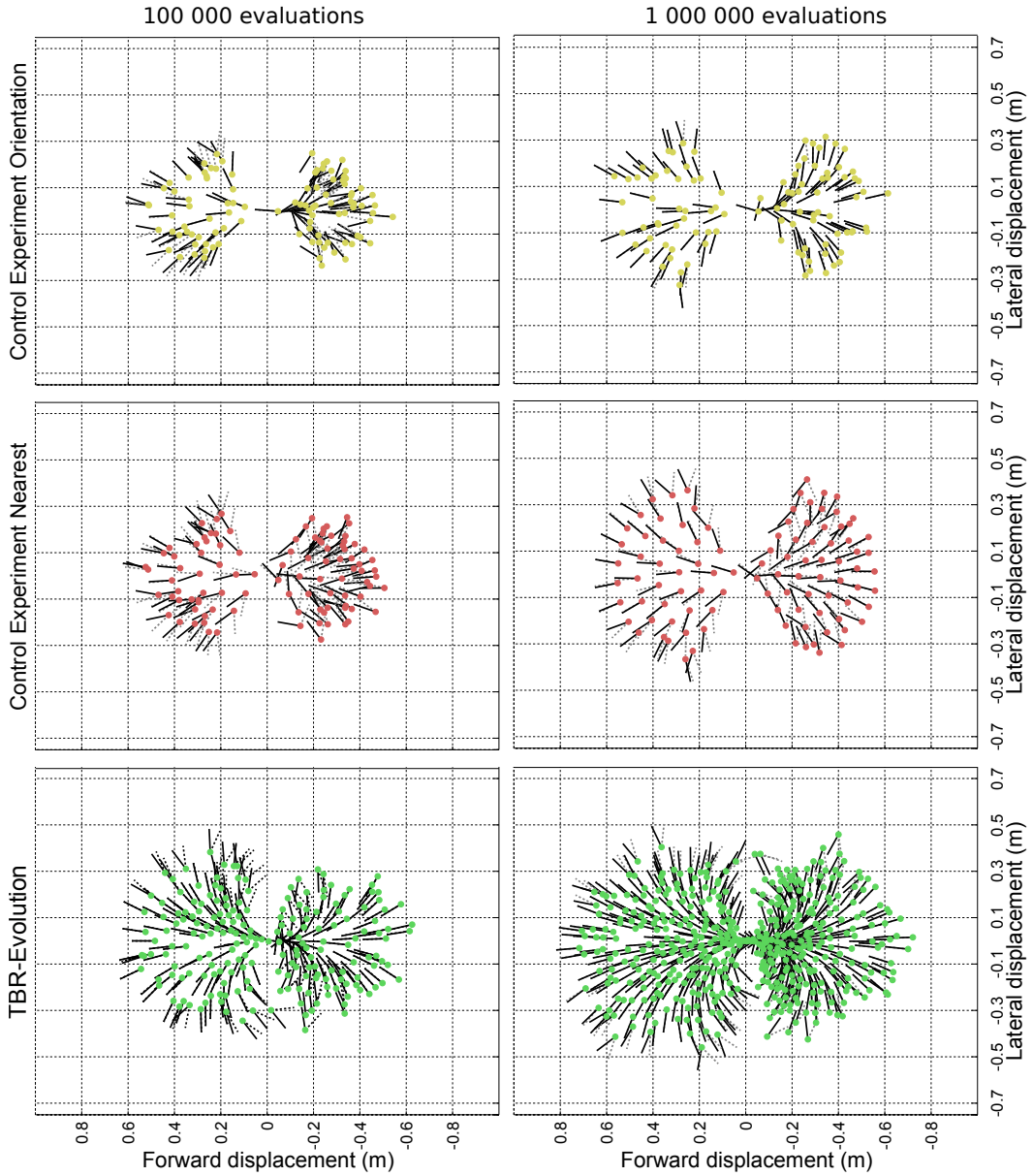


Figure 3.4: Comparison between the typical results of the TBR-Evolution algorithm, the "nearest", and the "orientation" variants. The archives are displayed after 100,000 evaluations (left) and after 1,000,000 evaluations (right). Each dot corresponds to the endpoint of a controller. The solid lines represent the final orientation of the robot for each controller, while the gray dashed lines represent the desired orientation. The orientation errors are the angle between solid and dashed lines.

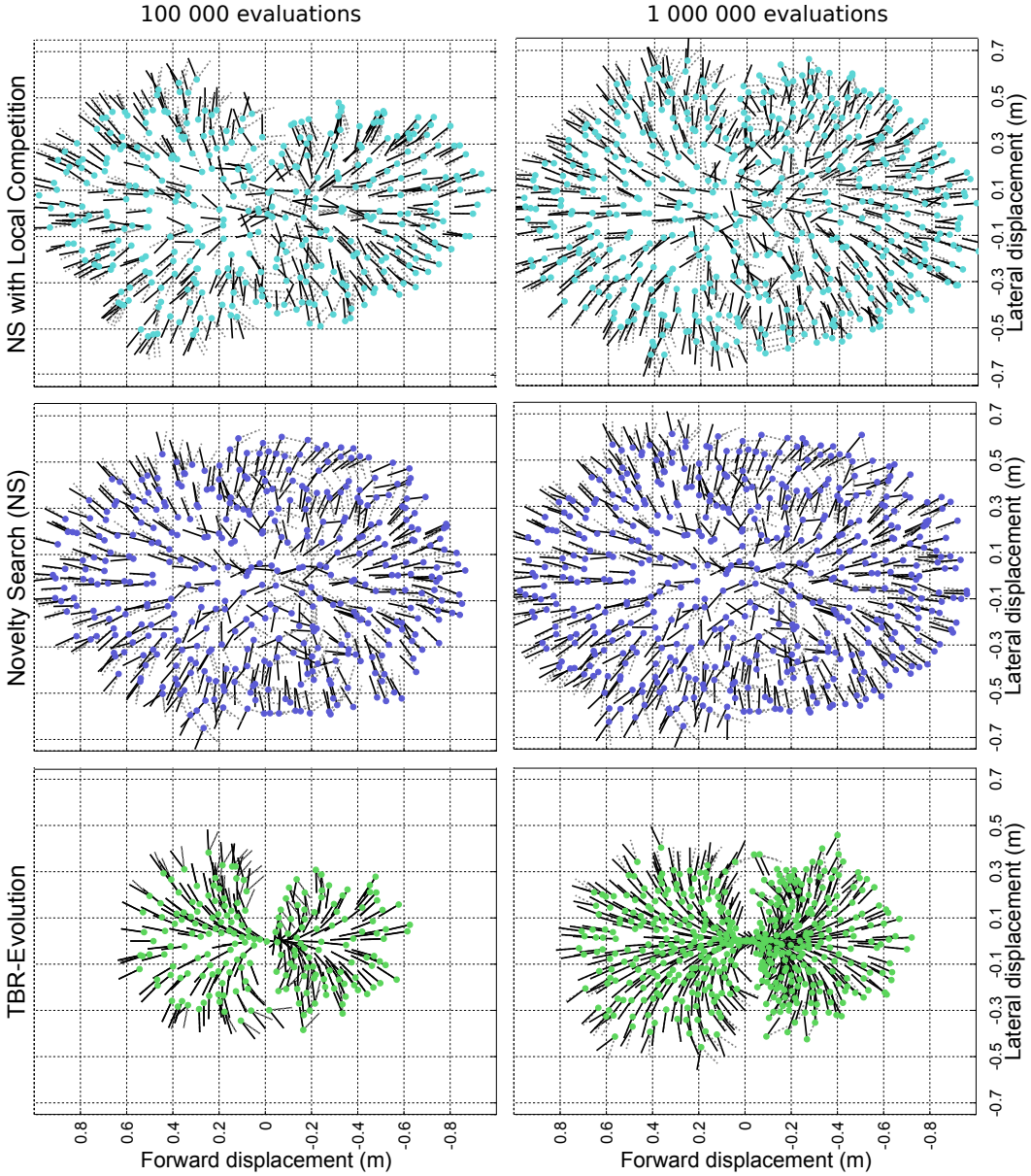


Figure 3.5: Comparison between the typical results of the TBR-Evolution algorithm, the Novelty Search, and the NS with Local Competition algorithm. The archives are displayed after 100,000 evaluations and after 1,000,000 evaluations. Each dot corresponds to the endpoint of a controller. The solid lines represent the final orientation of the robot for each controller, while the gray dashed lines represent the desired orientation. The orientation errors are the angle between solid and dashed lines.

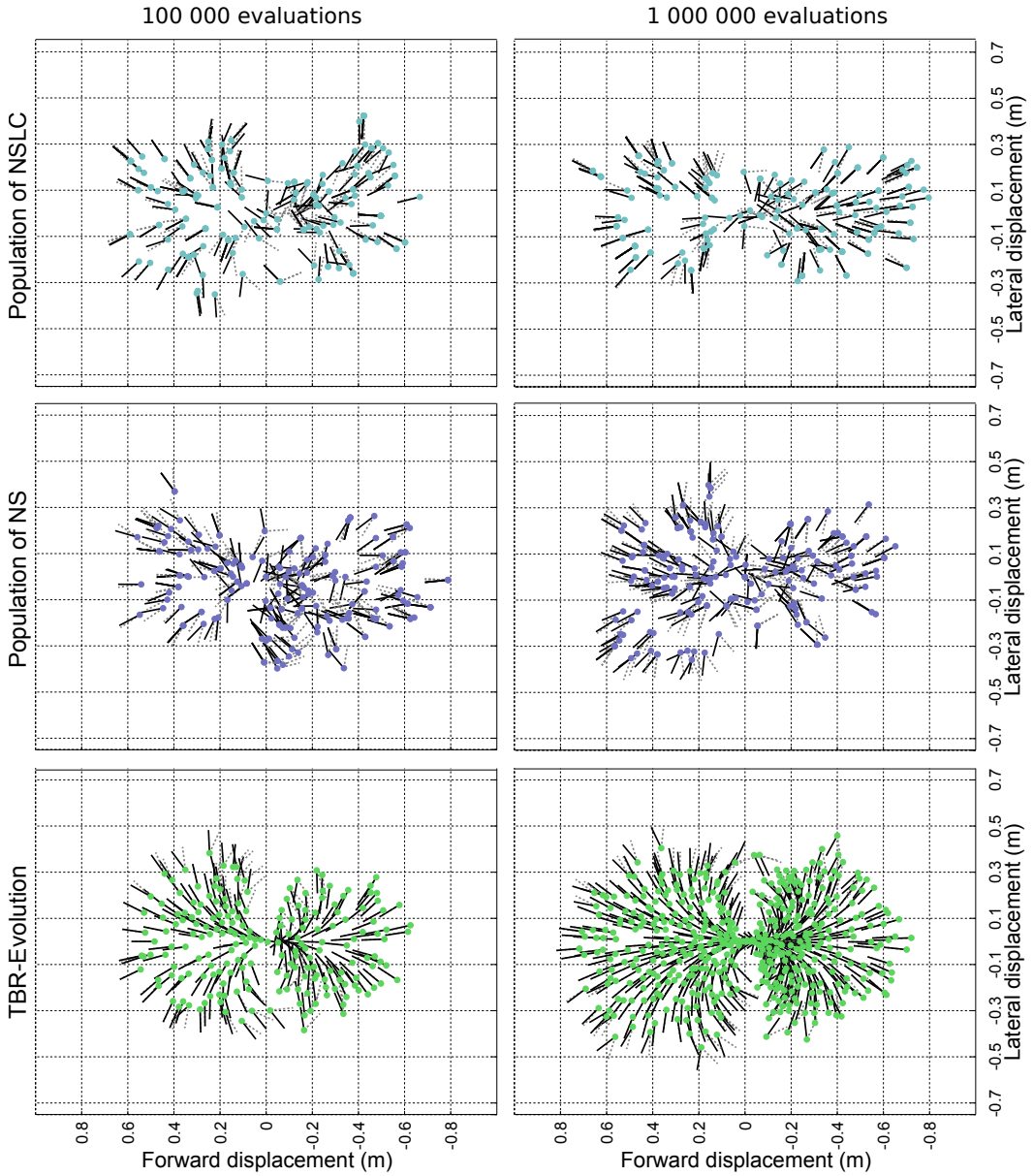


Figure 3.6: Comparison between typical results of the TBR-Evolution algorithm, the population of Novelty Search, and the population of NS with Local Competition. The archives/populations are displayed after 100,000 evaluations and after 1,000,000 evaluations. Each dot corresponds to the endpoint of a controller. The solid lines represent the final orientation of the robot for each controller, while the gray dashed lines represent the desired orientation. The orientation errors are the angle between solid and dashed lines.

Results Resulting behavioral repertoires from a typical run of TBR-Evolution and the control experiments are pictured on figures 3.4, 3.5 and 3.6. The endpoints achieved with each controller of the repertoire are spread over the reachable space in a specific manner: they cover both the front and the back of the robot, but less the lateral sides. These limits are not explicitly defined, but they are autonomously discovered by the algorithm.

For the same number of evaluations, the area is less covered with the control experiments (nearest and orientation) than with TBR-Evolution (Fig. 3.4). With only 100,000 evaluations, this area is about twice larger with TBR-Evolution than with both control experiments. At the end of the evolution (1,000,000 evaluations), the reachable space is denser with our approach. With the "nearest" variant of the control experiment, all target points are reached (see Fig. 3.3C), this is not the case for the "orientation" variant.

The archives produced by Novelty Search and NS with Local Competition both cover a larger space than the TBR-Evolution algorithm (Fig. 3.5). These results are surprising because all these experiments are based on novelty search and differ only in the way the archive is managed. These results show that TBR-Evolution tends to slightly reduce the exploration abilities of NS and focuses more on the quality of the solutions.

We can formulate two hypotheses to explain this difference in exploration. First, the "local competition" objective may have a higher influence in the TBR-Evolution algorithm than in the NS with Local Competition: in NS with local competition, the individuals from the population are competing against those of the archive; since this archive is not updated if an individual with a similar behavior but a higher performance is encountered, the individuals from the population are likely to always compete against low-performing individuals, and therefore always get a similar local competition score; as a result, the local competition objective is likely to not be very distinctive and most of the selective pressure can be expected to come from the novelty objective. This different selective pressure can explain why NS with local competition explores more than TBR-Evolution, and it echoes the observation that the archive obtained with NS and NS with local competition are visually similar (Fig. 3.5). The second hypothesis is that the procedure used to update the archive may erode the borderline of the archive: if a new individual is located close to the archive's borderline, and if this individual has a better performance than its nearest neighbor in the archive, then the archive management procedure of TBR-Evolution will replace the individual from the archive with the new and higher-performing one; as a consequence, an individual from the border can be removed in favor of a higher-performing but less innovative individual. This process is likely to repeatedly "erode" the border of the archive and hence discourage exploration. These two hypotheses will be investigated in future work.

The primary purpose of Novelty Search with Local Competition is to maintain a diverse variety of well-adapted solutions in its population, and not in its archive. For this reason, we also plotted the distribution of the population's individuals for both Novelty Search and NS with Local Competition (Fig. 3.6). After 100,000

evaluations, and at the end of the evolution, the population covers less of the robot's surrounding than TBR-Evolution. The density of the individuals is not homogeneous and they are not arranged in a particular shape, contrary to the results of TBR-Evolution. In particular, the borderline of the population seems to be almost random.

The density of the archive is also different between the algorithms. The density of the archives produced by TBR-Evolution is higher than the other approaches, while the threshold of novelty (ρ) required to add individuals in the archive is the same. This shows that the archive management of the TBR-Evolution algorithm increases the density of the regions where solutions with a good quality are easier to find. This characteristic allows a better resolution of the archive in specific regions.

The orientation error is qualitatively more important in the "nearest" control experiment during all the evolution than with the other experiments. This error is important at the beginning of the "orientation" variant too, but, at the end, the error is negligible for the majority of controllers. The Novelty Search, NS with local competition and the population of the Novelty Search have a larger orientation error, the figures 3.5 and 3.6 show that the orientation of the controllers seems almost random. With such repertoire, chaining behaviors on the robot is more complicated than with the TBR-Evolution's archives, where a vector field is visible. Only the population of the NS with Local Competition seems to show lower orientation error. This illustrates the benefits of the local competition objective on the population's behaviors.

The TBR-Evolution algorithm consistently leads to very small orientation errors (Fig. 3.4 and Fig. 3.9); only few points have a significant error. We find these points in two regions, far from the starting point and directly on its sides. These regions are characterized by their difficulty to be accessed, which stems from two main causes: the large distance to the starting point or the complexity of the required trajectory given the controller and the possible parameters (Appendix 3.2.2.1). For example the close lateral regions require executing a trajectory with a very high curvature, which cannot be executed with the range of parameters of the controller. Moreover, the behaviors obtained in these regions are most of the time degenerated: they take advantages of inaccuracies in the simulator to realize movement that would not be possible in reality. Since accessing these points is difficult, finding better solutions is difficult for the evolutionary algorithm. We also observe a correlation between the density of controllers, the orientation error and the regions difficult to access (Fig. 3.9): the more a region is difficult to access, the less we find controllers, and the less these controllers have a good orientation. For the other regions, the algorithm produces behaviors with various lengths and curvatures, covering all the reachable area of the robot.

In order to get a statistical point of view, we studied the median, over 40 runs, of the sparseness and the quality of controllers inside a region of interest (ROI) (Fig. 3.7, Top). The TBR-Evolution algorithm achieved a low sparseness value with few evaluations. After 100,000 evaluations, it was able to generate behaviors covering the reachable space with an interval distance of about 3 cm. At the end

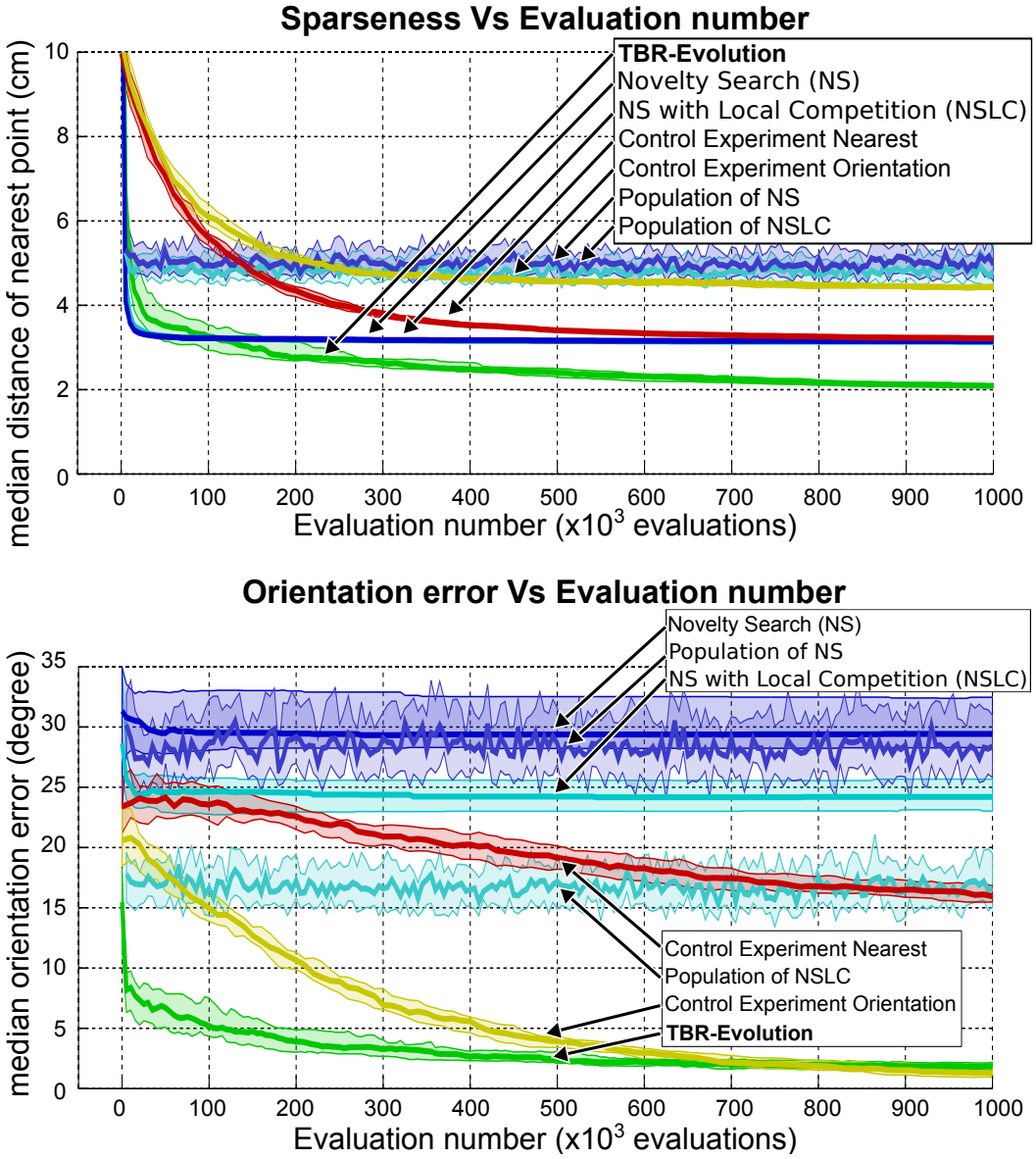


Figure 3.7: (Top) Variation of the sparseness of the controller repertoire. For each point of a one-centimeter grid inside the ROI (Fig. 3.3), the distance from the nearest controller is computed. The sparseness value is the average of these distances. This graph plots the first three quartiles of the sparseness computed with 40 runs for each algorithm. (Bottom) Variation of the median of the orientation error over all the controllers inside the region of interest. This graph also plots the three first quartiles $(25\%, 50\%, 75\%)$ computed with 40 runs for each algorithm.

of the process, the sparseness value is near 2 cm. With the "nearest" and the "orientation" experiments, the variation is slower and reaches a significantly higher level of sparseness (p-values = 1.4×10^{-14} with Wilcoxon rank-sum tests). The "orientation" variant of the control experiment exhibits the worst sparseness value ($> 4cm$). This result is expected because this variant favors behaviors with a good orientation even if they are far from their reference point. This phenomenon leads to a sample of the space less evenly distributed. The "nearest" variant achieves every target points, thus the sparseness value is better than with the "orientation" variant (3 cm vs. 4cm, at the end of the experiment). The Novelty Search and the NS with Local Competition experiments follow the same progression (the two lines are indistinguishable) and reach their final value faster ass the TBR-Evolution algorithm. As our algorithm can increase the density of controller in particular regions, at the end of the evolution, the final value of sparseness of TBR-Evolution is better than all the control experiments. The sparseness of the populations of Novelty Search and NS with Local Competition are indistinguishable too, but also constant over all the evolution and larger than all the tested algorithms, mainly because of the uneven distribution of their individuals (fig. 3.6)

From the orientation point of view (Fig. 3.7, bottom), our approach needs few evaluations to reach a low error value (less than 5 degrees after 100,000 evaluations and less than 1.7 degrees at the end of the evolutionary process). The variation of the "orientation" control experiment is slower and needs 750,000 evaluations to cross the curve of TBR-Evolution. At the end of the experiment this variant reaches a significantly lower error level (p-values = 3.0×10^{-7} with Wilcoxon rank-sum tests), but this corresponds to a difference of the medians of only 0.5 degrees. The "nearest" variant suffers from significantly higher orientation error (greater than 15 degrees, p-values = 1.4×10^{-14} with Wilcoxon rank-sum tests). This is expected because this variant selects behaviors taking into account only the distance from the target point. With this selection, the orientation aspect is neglected. The Novelty Search and the NS with Local Competition experiments lead to orientation errors that are very high and almost constant over all the evolution. These results come from the archive management of these algorithms that do not substitute individuals when a better one is found. The archive of these algorithms only gathers the first encountered behavior of each reached point. The orientation error of the NS with Local Competition is lower than the Novelty Search because the local competition promotes behavior with a good orientation error (compared to their local niche) in the population, which has an indirect impact on the quality of the archive but not enough to reach a low error level. The same conclusion can be drawn with the population of these two algorithms: while the populations of the Novelty Search have a similar orientation error than its archives, the populations of the NS with Local Competition have a lower orientation error than its archives.

With the sets of reference points, we can compute the theoretical minimal sparseness value of the control experiments (Fig. 3.8, Left). For example, changing the number of targets from 100 to 200 will change the sparseness value from 3.14 cm to 2.22 cm. Nonetheless, doubling the number of points will double the required

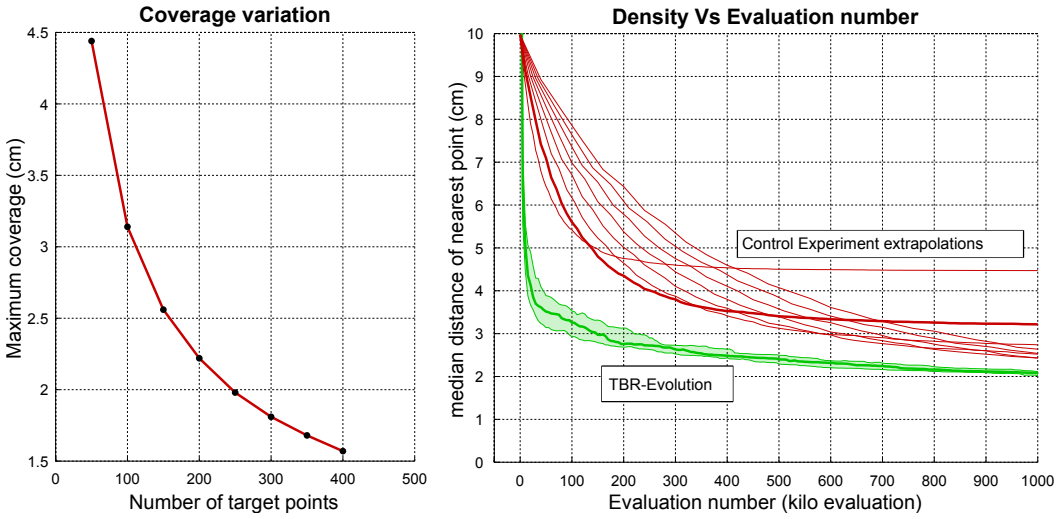


Figure 3.8: (Left) Theoretical sparseness of the control experiments according to the number of points. With more points, the sparseness value will be better (lower). (Right) Extrapolations of the variation of the sparseness for the "nearest" variant of the control experiment according to different number of targets. Each line is an extrapolation of the variation of the sparseness of the "nearest variant", which is based on a number of points starting from 50 to 400, with a 50 points step. The variation of TBR-Evolution is also plotted for comparison.

number of evaluations. Thanks to these values we can extrapolate the variation of the sparseness according to the number of points. For example, with 200 targets, we can predict that the final value of the sparseness will be 2.22 and thus we can scale the graph of our control experiment to fit this prediction. Increasing the number of targets will necessarily increase the number of evaluations, for example using 200 targets will double the number of evaluations. Following this constraint, we can also scale the temporal axis of our control experiment. We can thus extrapolate the sparseness of the archive with regard o the number of target, and compare it to the sparseness of the archive generated with TBR-Evolution.

The extrapolations (Fig. 3.8, right) show higher sparseness values compared to TBR-Evolution within the same execution time. Better values will be achieved with more evaluations. For instance, with 400 targets the sparseness value reaches 1.57 cm, but only after 4 millions of evaluations. This figure shows how our approach is faster than the control experiments regardless the number of reference points.

Figures 3.7 and 3.8 demonstrate how TBR-Evolution is better both in the sparseness and in the orientation aspects compared than the control experiments. Within few evaluations, reachable points are evenly distributed around the robot and corresponding behaviors are mainly well oriented.

(An illustrating video is available on: `http://youtu.be/2aTIL_c-qwA`)

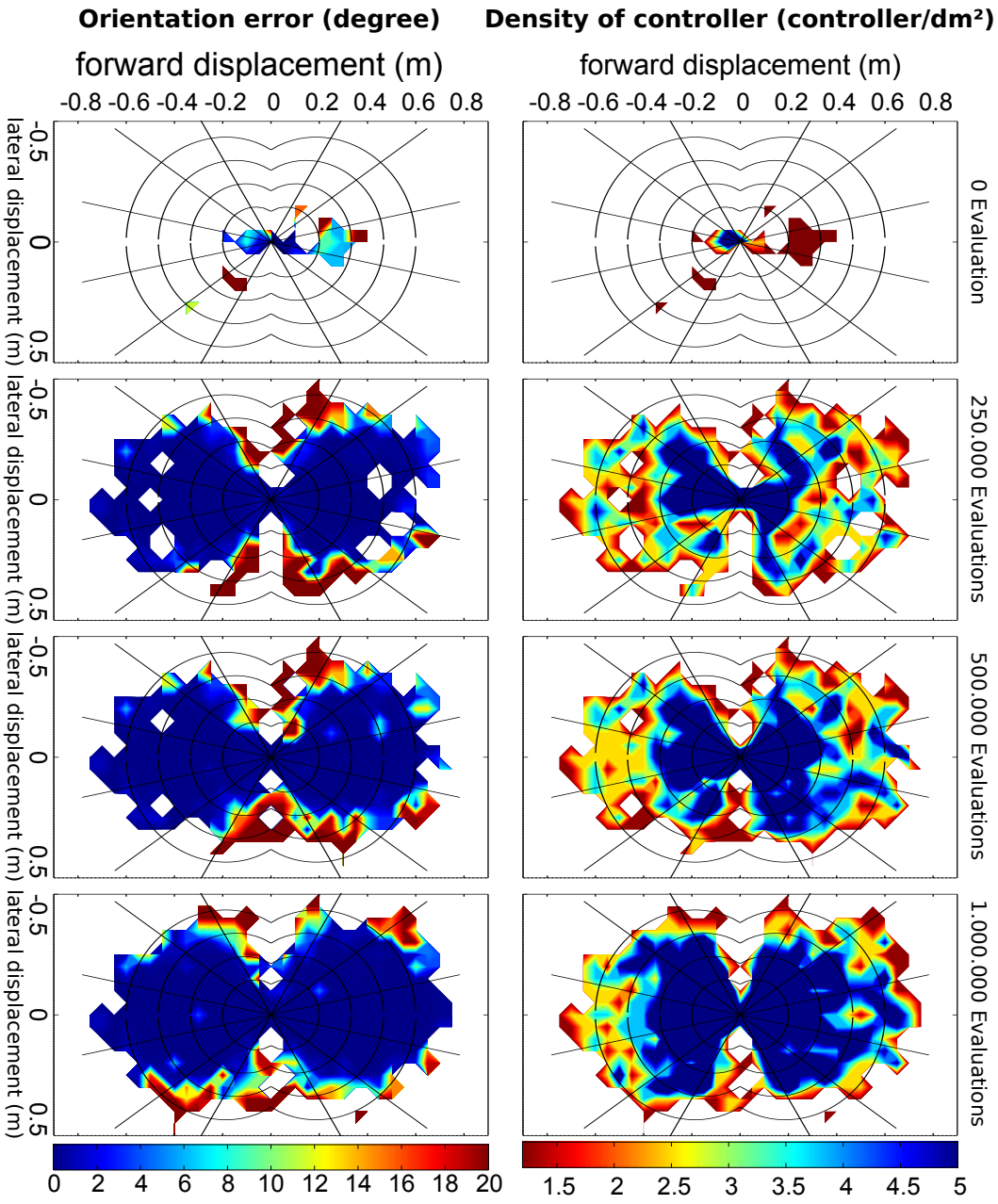


Figure 3.9: (Right) Variation of density of controller (number of controllers per dm^2). (Left) Variation of the orientation error (given by the nearest controller) along a typical run in simulation.

3.2.2.6 Experiments on the Physical Robot

In this second set of experiments, we apply the TBR-Evolution algorithm on a physical hexapod robot (see Fig. 3.1 left). The transferability component of the algorithm allows it to evolve the behavioral repertoire with a minimum of evaluations on the physical robot. For this experiment, $3,000$ generations are performed and we execute a transfer (evaluation of one controller on the physical robot) every 50 generations, leading to a total of 60 transfers. The TBR-Evolution experiments and the reference experiments are replicated 5 times to gather statistics[1].

Reference Experiment In order to compare the learning speed of the TBR-Evolution algorithm, we use a reference experiment where only one controller is learned. For this experiment, we use the NSGA-II algorithm with the transferability approach to learn a controller that reaches a predefined target. The target is situated 0.4m in front and 0.3m to the right: a point not as easy to be accessed as going only straight forward, and not as hard as executing a U-turn. It represents a good difficulty trade-off and thus allows us to extrapolate the performances of this method to more points.

The main objective is the distance (Distance($\mathbf{c}$)) between the endpoint of the considered controller and the target. The algorithm also optimizes the estimated transferability value ($\hat{\mathscr{T}}(\mathbf{des}(\mathbf{c}))$) and the orientation error (perf($\mathbf{c}$)) with the same definition as in the TBR-Evolution algorithm:

$$\text{minimize} \begin{cases} \text{Distance}(\mathbf{c}) \\ \hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) \\ \text{perf}(\mathbf{c}) \end{cases}$$

To update the transferability function, we use the same transfer frequency as in TBR-Evolution experiments (every 50 generations). Among the resulting trade-offs, we select as final controller the one that arrives closest to the target among those with an estimated transferability $\hat{\mathscr{T}}(\mathbf{des}(\mathbf{c})) < 0.10m$. This represents a distance between the endpoint reached in simulation and the one reached in reality lower than 10 cm.

Results After 3,000 iterations and 60 transfers, TBR-Evolution generates a repertoire with a median number of 375 controllers (min = 352, max = 394). This is achieved in approximately 2.5 hours. One of these repertoires is pictured in figure 3.10, left. The distribution of the controllers' endpoints follows the same pattern as in the virtual experiments: they cover both the front and the rear of the robot, but not the lateral sides. Here again, these limits are not explicitly defined, they are autonomously discovered by the algorithm.

[1] Performing statistical analysis with only 5 runs is difficult but it still allows us to understand the main trends. The current set of experiments (5 runs of TBR-Evolution and the control experiment) requires more than 30 hours with the robot and it is materially challenging to use more replications.

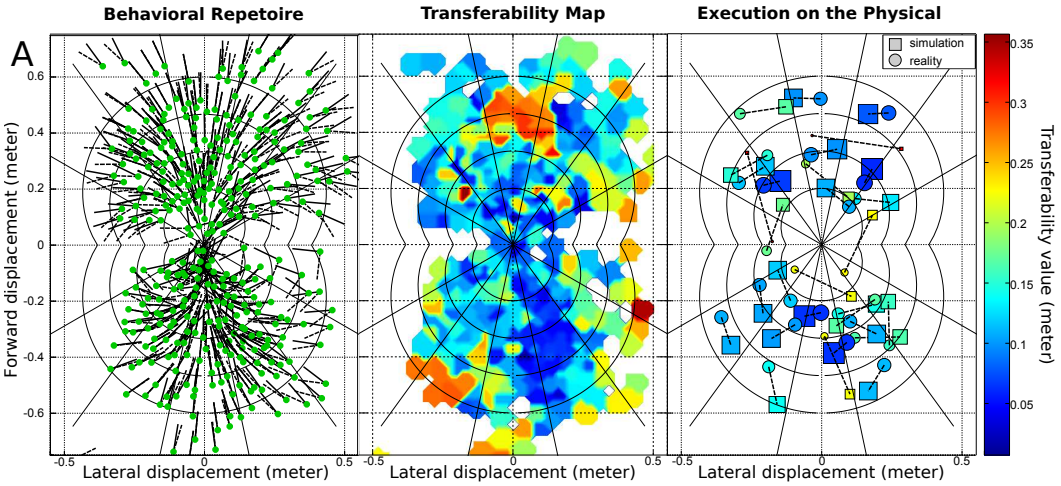


Figure 3.10: Typical repertoire of controllers obtained with the TBR-Evolution algorithm. (Left) The dots represent the endpoints of each controller. The solid lines are the final orientations of the robot while the dashed ones are the desired orientations. The angle between these two lines is the orientation error. (Center) Transferability map. For each point of the reachable space, the estimated transferability of the nearest controller, within a radius of 5cm, is pictured. (Right) Execution on the physical robot. The 30 selected controllers are pictured with square and their actual endpoint with circles. The size and the color of the markers are proportional to their accuracy. To select the tested controllers, the reachable space is split into 30 regions. Their boundaries are defined by two lines at 60 degrees on each side of the robot and by two curved frontiers that regroup all reachable points with a curvi-linear abscissa between 0.2 and 0.6 m. These regions are then segmented into 15 parts for both the front and the rear of the robot. All of these values are set from experimental observations of commonly reachable points.

Similarly to the experiments on the virtual robot, the majority of the controllers have a good final orientation and only the peripheries of the repertoire have a distinguishable orientation error. TBR-Evolution successfully pushes the repertoire of controllers towards controllers with a good quality score and thus following the desired trajectories.

From these results we can draw the same conclusion as with the previous experiment: the difficulty of accessing peripheral regions explains the comparatively poor performances of controllers from these parts of archive. The large distance to the starting point or the complexity of the required trajectory meets the limits of the employed controllers.

The transferability map (Fig. 3.10, center) shows that the majority of the controllers have an estimated value lower than 15cm (dark regions). Nevertheless, some regions are deemed non-transferable (light regions). These regions are situated in the peripheries too, but are also in circumscribed areas inside of the reachable space. Their occurrence in the peripheries have the same reasons as for the orienta-

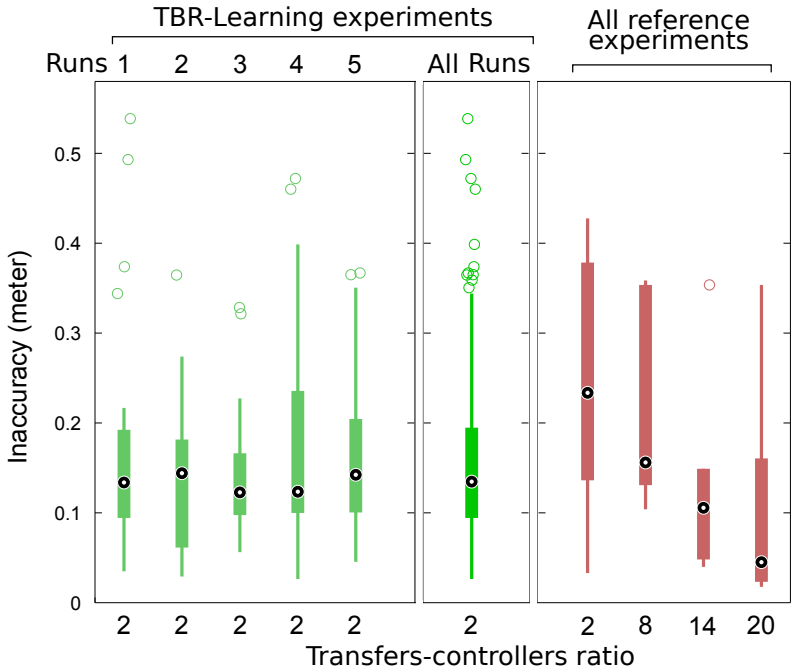


Figure 3.11: Accuracy of the controllers. The accuracy is measured as the distance between the endpoint reached by the *physical robot* and the one reached in simulation (30 points for each run, see text). The results of the TBR-Evolution experiments are, for each run, separately pictured (Left) and also combined for an overall point of view (Center). The performances of the reference experiments are plotted according to the number of transfers performed (Right). In both cases, one transfer is performed every 50 iterations of the algorithm.

tion (section 3.2.2.5), but those inside the reachable space show that the algorithm failed to find transferable controllers in few specific regions. This happens when the performed transfers do not allow the algorithm to infer transferable controllers. To overcome this issue, different selection heuristics and transfer frequencies will be considered in future work.

In order to evaluate the hundreds of behaviors contained in the repertoires on the physical robot, we select 30 controllers in each repertoire of the 5 runs. The selection is made by splitting the space into 30 areas (Fig. 3.10) and selecting the controllers with the best estimated transferability in each area.

Most of these controllers have an actual transferability value lower than 15 cm (Fig. 3.11, left), which is consistent with the observations of the transferability map (Fig. 3.10, center) and not very large once taken into consideration the SLAM precision, the size of the robot and the looseness in the joints. Over all the runs, the median accuracy of the controllers is 13.5 cm (Fig. 3.11, center). Nevertheless, every run presents outliers, i.e. controllers with a very bad actual transferability value, which originate from regions that the transferability function does not correctly approximate.

Figure 3.12: Evolution of the number of controllers deemed transferable. At each transfer (i.e. every 50 iterations), the number of controllers with an estimated transferability lower than 15cm is pictured for all the 5 runs. The bold black line represents the median value, the dark region the first and third quartile and the light one the lower and upper bounds. The variability of the curve is due to the periodic transfers, which update the transferability function and thus the estimated transferability values.

In order to compare the efficiency of our approach to the reference experiment, we use the "transfers-controllers ratio", that is the number of performed transfers divided by the number of produced controllers at the end of the evolutionary process. For instance, if we reduce the produced behavioral repertoires to the 30 tested controllers, this ratio is equal to $60/30 = 2$ for the TBR-Evolution experiments, since we performed 60 transfers.

The performances of the control experiments depend on the number of performed transfers (Fig. 3.11, right) and thus on this ratio. For an equal ratio, the reference experiments are 74% less accurate than TBR-Evolution (13.5 cm vs. 23.4 cm, p-value= 0.12 with the Wilcoxon ranksum test), while the accuracies of both experiments are not statistically different (13.5 cm vs. 15.6 cm and 10.6 cm, p-value= 0.23 and respectively 0.35) if the reference algorithm uses from 8 to 14 transfers to learn one controller (i.e. a process 4 to 7 times longer). The reference experiment only takes advantage of its target specialisation when 20 transfers are performed. With a transfers-controllers ratio equals to 20, the accuracy of the reference controllers outperforms the controllers generated with the TBR-Evolution algorithm (13.5 cm vs. 4.5 cm, p-value= 0.06). Nevertheless, with such a high ratio, the reference experiment only generates 3 controllers, while our approach generates 30 of them with the same running time (60 transfers and 3,000 generations).

We previously only considered the 30 post evaluated controllers, whereas TBR-Evolution generates several hundreds of them. After 60 transfers, the repertoires contain a median number of 217 controllers that have an estimated transferability lower than 0.15 m (Fig. 3.12). The previous results show that more than 50% of the tested controllers have an actual transferability value lower than 0.15 m and 75% lower than 0.20 m. We can consequently extrapolate that between 100 and 150 controllers are exploitable in a typical behavioral repertoire. Once taking into consideration all these controllers, the transfers-controllers ratio of the TBR-Evolution experiments falls between 0.4 and 0.6 and thus our approach is about 25 times faster than the reference experiment, for a similar accuracy.

3.3 The MAP-Elites algorithm

3.3.1 Principle

In this last part of this chapter, we explore the use of a second algorithm, called the multi-dimensional archive of phenotypic elites (MAP-Elites) algorithm, to create behavioral repertoires (Mouret and Clune, 2015). We call the repertoires generated with this algorithm *behavior-performance maps*. MAP-Elites searches for the highest-performing solution for each point in a user-defined space. For example, when designing controllers for legged robot, the user may be interested in seeing the highest-performing solution at each point in a two-dimensional space where one axis is the speed of the robot and the other axis is the direction of the movement. Alternatively, a user may wish to see speed vs. final orientation, or see solutions throughout a 3D space of speed vs. direction vs. orientation. Any dimension that can vary could be chosen by the user. There is no limit on the number of dimensions that can be chosen, although it becomes computationally more expensive to fill the behavior-performance map and store it as the number of dimensions increases. It also becomes more difficult to visualize the results. We refer to this user-defined space as the "behavior space", because usually the dimensions of variation measure behavioral characteristics. Note that the behavioral space can refer to other aspects of the solution, like physical properties of mechanical designs.

If the behavior descriptors and the parameters of the controller are the same (i.e. if there is only one possible solution for each location in the behavioral space), then creating the behavior-performance map is straightforward: one simply needs to simulate the solution at each location in the behavior space and record the performance. However, if it is not known a priori how to produce a controller that will end up in a specific location in the behavior space (i.e. if the parameter space is of higher dimension than the behavioral space: e.g., in our example, if there are many different controllers of a specific speed, direction, and orientation, or if it is unknown how to make a description that will produce a controller with a specific speed, direction, and orientation), then MAP-Elites is beneficial. It will efficiently search for the highest-performing (for example, the most stable) solution at each point of the low-dimensional behavioral space. It is more efficient than a

random sampling of the search space because high-performing solutions are often similar in many ways, such that randomly altering a high-performing solution of one type can produce a high-performing solution of a different type (as already pointed in the previous parts of this chapter). For this reason, searching for high-performing solutions of all types simultaneously is much quicker than separately searching for each type. For example, to generate controllers that make the robot quickly turn left with a high stability, it is often more effective and efficient to modify an existing controller of a fast and stable behavior rather than randomly generate new controllers from scratch or launch a separate search process for each new type of controller.

MAP-Elites begins by generating a set of random candidate solutions. It then evaluates the performance of each solution and records where that solution is located in the behavior space (e.g. if the dimensions of the behavior space are the speed and direction, it records the speed and the direction of each behavior in addition to its performance). For each solution, if its performance is better than the current solution at that location in the behavior-performance map, then it is added to the behavior-performance map, replacing the solution in that location. In other words, it is only kept if it is the best of that type of solution, where "type" is defined as a location in the behavior space. There is thus only one solution kept at each location in the behavior space (keeping more could be beneficial, but for computational reasons we only keep one). If no solution is present in the behavior-performance map at that location, then the newly generated candidate solution is added at that location.

Once this initialization step is finished, MAP-Elites enters a loop that is similar to stochastic, population-based, optimization algorithms, such as evolutionary algorithms (see section 2.2): the solutions that are in the behavior-performance map form a population that is improved by random variation and selection. In each generation, the algorithm picks a solution at random via a uniform distribution, meaning that each solution has an equal chance of being chosen. A copy of the selected solution is then randomly mutated to change it in some way, its performance is evaluated, its location in the behavioral space is determined, and it is kept if it outperforms the current occupant at that point in the behavior space (note that mutated solutions may end up in different behavior space locations than their "parents"). This process is repeated until a stopping criterion is met (e.g. after a fixed amount of time has expired). This main loop is depicted in figure 3.13. Because MAP-Elites is a stochastic search process, each resultant behavior-performance map can be different, both in terms of the number of locations in the behavioral space for which a candidate is found, and in terms of the performance of the candidate in each location.

The pseudo-code of the algorithm is defined in Algorithm 2.

MAP-Elites can be seen as an evolutionary algorithm working on multiple niches, in which competition only occurs between individuals from the same niche. Individuals from one niche can invade other niches only if, after being mutated, the new individuals can survive in a new niche and that their fitness overrun the former

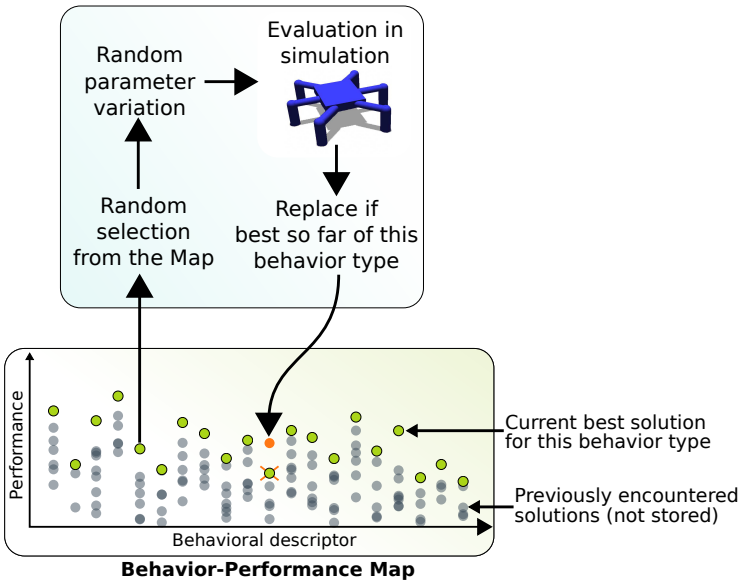


Figure 3.13: The MAP-Elites main loop. After initializing the map with a few random behaviors, MAP-Elites randomly selects, via an uniform distribution, a behavior in the map and generates a mutated copy of this behavior. During the evaluation of this new behavior in simulation, its performance and behavior descriptor are recorded. If the cell corresponding to this new behavior descriptor is empty, then the new behavior is added to the map, otherwise, the behavior with the highest performance is kept in the cell. This simple loop, which can easily be performed in parallel, is repeated several millions of time, which progressively increases both the coverage of the map and the quality of the behaviors.

Algorithm 2 MAP-Elites Algorithm (I evaluations, N random evaluations)

$(\mathscr{P} \leftarrow \emptyset, \mathscr{C} \leftarrow \emptyset)$ ▷ *Creation of an empty behavior-performance map (empty N-dimensional grid).*
for iter $= 1 \rightarrow I$ **do** ▷ *Repeat during I iterations (here we choose I).*
  if iter $< N$ **then**
    $\mathbf{c}' \leftarrow$ random_controller() ▷ *The first N controllers are generated randomly.*
  else ▷ The next controllers are generated using the map.
    $\mathbf{c} \leftarrow$ random_selection($\mathscr{C}$) ▷ *Randomly select a controller c in the map.*
    $\mathbf{c}' \leftarrow$ random_variation($\mathbf{c}$) ▷ *Create a randomly modified copy of c.*
  $\mathbf{x}' \leftarrow$behavioral_descriptor(simu($\mathbf{c}'$)) ▷ *Simulate the controller and record its behavioral descriptor.*
  $p' \leftarrow$performance(simu($\mathbf{c}'$)) ▷ *Record its performance.*
  if $\mathscr{P}(\mathbf{x}') = \emptyset$ or $\mathscr{P}(\mathbf{x}') < p'$ **then** ▷ *If the cell is empty or if p' is better than the current stored performance.*
    $\mathscr{P}(\mathbf{x}') \leftarrow p'$ ▷ *Store the performance of $\mathbf{c}'$ in the map according to its behavioral descriptor $\mathbf{x}'$.*
    $\mathscr{C}(\mathbf{x}') \leftarrow \mathbf{c}'$ ▷ *Associate the controller with its behavioral descriptor.*
return behavior-performance map ($\mathscr{P}$ and $\mathscr{C}$)

individuals of this niche. In MAP-Elites, each cell of the map is a different niche and its main loop (Fig. 3.13) tends to mimic this natural process.

While MAP-Elites and TBR-Evolution (and thus Novelty Search with Local Competition) are based on two different ideas, a few parallels can be drawn. First, both of these algorithms do not optimize a single objective but rather search for a set of diverse and high performing solutions. This new family of evolutionary algorithms, which has been named *Quality Diversity* algorithms by Pugh et al. (2015), contains currently only MAP-Elites and Novelty Search with Local Competition (including TBR-Evolution). A second parallel is that they both use an archive or a map that keeps a kind of history of the individuals that have been encountered during the evolutionary process. This information is then used in the selection process of the algorithms. In TBR-Evolution, the archive is used to maintain a population containing only novel and locally high-performing behaviors, while in MAP-Elites the selection is randomly performed among the whole map. The selection process is crucial in evolutionary algorithms, as it influences the stepping stones that the algorithms may encounter, which can lead to new types of solution. The Novelty Search algorithm makes the hypothesis that to find these stepping stones, the algorithm can focus on individuals that are novel. Nonetheless, this hypothesis actually adds a bias in exploration strategy and stepping stones may come from individual that are neither novel nor high-performing. MAP-Elites removes this bias and the following experimental results suggest that focusing on novel behaviors may be counterproductive.

Pugh et al. (2015) also investigated the differences between MAP-Elites and Novelty-based approaches. In particular, they used a variant of the novelty search algorithm that fills maps similar to those of MAP-Elites. This variant is thus analogous to the TBR-Evolution algorithm (without the Transferability objective). The authors analyzed the quality of the maps (percentage of filled cells and their performance) produced with the algorithms according to different behavioral descriptors. Their experiments revealed that the algorithm that should be employed depends on the alignment of the behavioral descriptor with the performance criteria. Indeed, behavioral descriptor can be directly related to the performance function. For example, one of the behavioral descriptor presented in the "QD-Maze" experiment of Pugh et al. (2015) is the end-point of the robot's trajectory, while the performance is the distance to the exit of the maze. In this case, the behavioral descriptor and the performance function are perfectly aligned because the performance can be directly computed from the behavioral descriptor. Conversely, the final orientation of the robot is behavioral descriptor orthogonal to the performance function, as the orientation is unrelated to the robot's position. The conclusion of the authors is that if the behavioral descriptor is aligned with the performance function then novelty-based approaches produced the best maps, while when the behavioral descriptor and the performance function are orthogonal then MAP-Elites is more instrumental than the other approaches.

3.3.2 Experimental validation

We evaluated the quality of the behavior-performance maps generated by MAP-Elites on two tasks. The first one is similar to the experiment presented in the previous section. Our hexapod robot has to learn to walk in every direction and at various speeds. The goal of this experiment is to produce behavior-performance maps that are similar to the behavioral repertoires generated with the TBR-Evolution algorithms, in order to compare these two algorithms. The second experiment consists in finding all the possible ways to walk on a straight line with our hexapod robot. In order words, the robot has to discover how to walk by using only some of its legs (for example with only 5 or 4 legs) and also many other ways like moving forward by hopping. For each of these walking manners, the robot has to find the fastest behaviors. For example, the fastest walking gait that uses the 6 legs, the fastest hopping behaviors etc.

These two experiments are only conducted in simulation, as MAP-Elites requires a large number of evaluations (like most EAs). Nonetheless, we will present in the next chapter how to use the produced behavior-performance maps to allow a physical robot to learn to walk and to adapt to damage situations in only a handful of evaluations.

3.3.2.1 Learning to walk in every direction

Methods In this first experiment, in which our hexapod robot (fig 3.2 left) learns to walk in every direction, the behavioral descriptor is defined as the end-point of the robot's trajectory after running the controller during 3 seconds. The behavioral space is a 2-dimensional rectangle (2 meters by 1.4 meters). The frontal axis is discretized in 51 chunks while the frontal axis is discretized in 35 chunks. This leads to a total of 1785 cells in the map and each cell is a 4 x 4cm location in the robot's vicinity. This cell size has been chosen to produce maps with the same density of behaviors as the behavioral repertoires from the previous experiments.

Like in the previous experiment, the performance criterion is the orientation error (see section 3.2.2.3) and it is used to foster behaviors to follow circular trajectories (see Fig. 3.3A). MAP-Elites is allowed to perform 1,000,000 evaluations during the map generation process (like in the TBR-Evolution experiments). The type of controller evolved in this experiment is the same as in the TBR-Evolution's experiments. Like TBR-Evolution, MAP-Elites can be used with any type of controller. Here again, we choose to use a simple controller in order to show that the abilities of our robot come from our learning algorithms and not from a particular controller type. This experiment has been replicated 40 times.

Results The results show that MAP-Elites is significantly better than TBR-Evolution in many aspects. First qualitatively, we can see that the maps produced by MAP-Elites cover a larger space around the robot than the behavioral repertoire from TBR-Evolution (see Fig. 3.14). This difference appears very soon in the

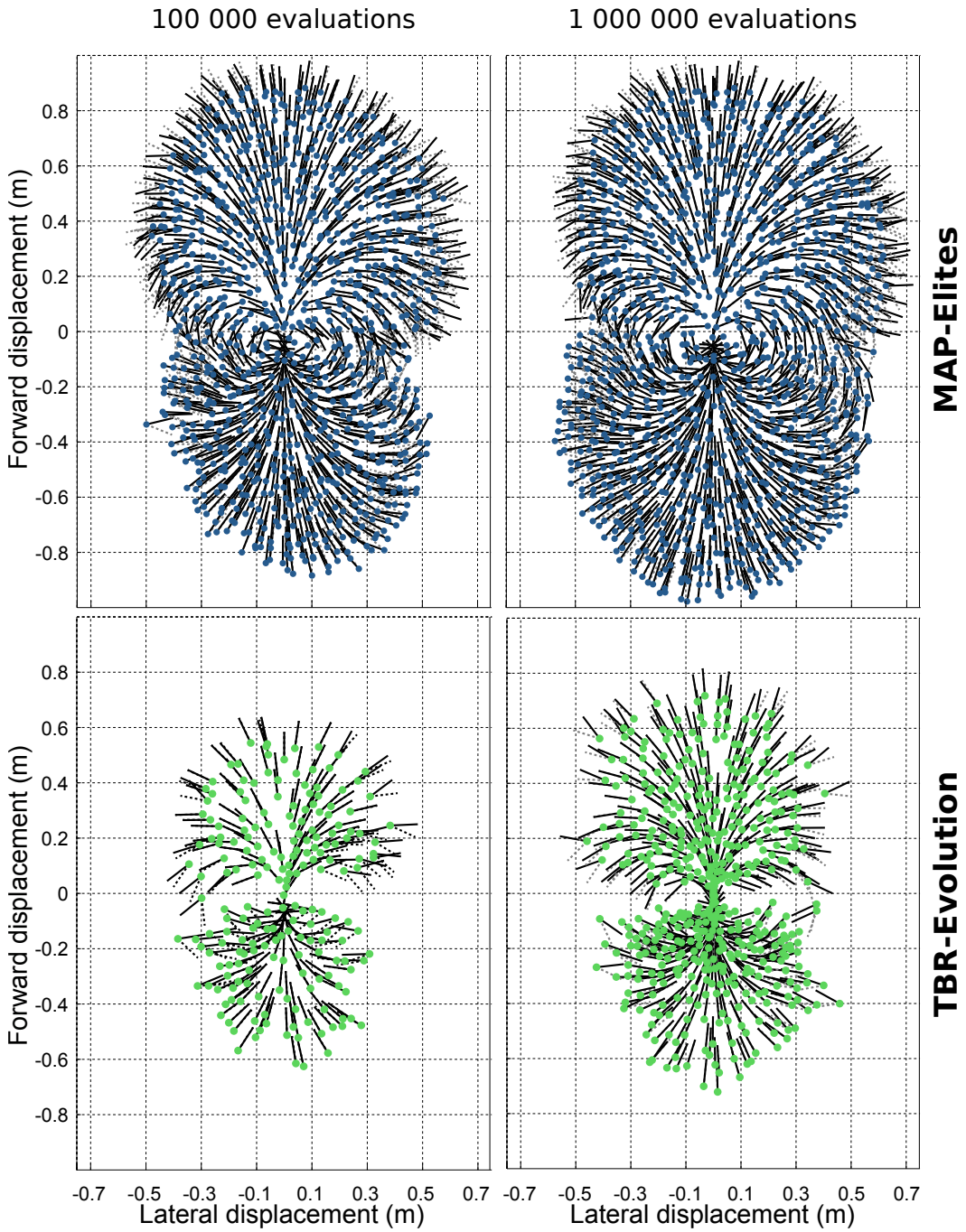


Figure 3.14: Comparison between a typical behavior-performance map obtained with MAP-Elites and a behavioral repertoire generated with the TBR-Evolution algorithm. The archives are displayed after 100,000 evaluations (left) and after 1,000,000 evaluations (right). Each dot corresponds to the endpoint of a controller. The solid lines represent the final orientation of the robot for each controller, while the gray dashed lines represent the desired orientation. The orientation error is the angle between solid and dashed lines.

generation process, as the map and the behavior repertoire are already different after only 100,000 evaluations. After 1,000,000 evaluations, the maps generated with MAP-Elites contain about 1270 controllers (median: 1271 controllers, 5^{th} and 95^{th} percentiles: [1159.5; 1325] controllers) corresponding to about 70% of the map cells, while the behavioral repertoires from TBR-Evolution only contain about 470 behaviors (468.5 [429.3; 502.45] evaluations). The controllers in the map show a good orientation error, as a clear vector field can be seen in the figure that suggest that the behaviors follow the desired circular trajectories.

From a quantitative point of view, we can see in figure 3.15 that MAP-Elites has always a better sparseness than TBR-Evolution during all the experiment (median: 1.74 vs. 2.10; p-value: 10^{-13}). In particular, we can observe that TBR-Evolution reaches a sparseness value similar to MAP-Elites only at the end of the experiment. The same conclusion can be drawn with the orientation error. MAP-Elites becomes rapidly better than TBR-Evolution and converges to an error close to zero (median 0.06 degrees vs. 1.83; p-values: 10^{-13}).

While the final sparseness values are influenced by some algorithm's parameters (for example, the ρ value of TBR-Evolution or the discretization of the grid in MAP-Elites), we can observe that MAP-Elites has an higher convergence speed than TBR-Evolution. In less than 200,000 evaluations, both the sparseness and the orientation error converged to a stable value, which is at least 5 times faster than TBR-Evolution. The sparseness decreases rapidly and converges in less than 50,000 evaluations. Once most of the map's cells are filled, their quality progressively increases before converging after about 200,000 evaluations. This shows that MAP-Elites benefits from using its whole map as a current population and not only the current most novel individuals like in TBR-Evolution. Furthermore, it may suggest that the random selection of MAP-Elites over its whole map is a more efficient way to find stepping stones than focusing on novel behaviors.

With this experiment we saw that, while MAP-Elites is a very simple algorithm, it shows results that outperform the TBR-Evolution algorithm. For this reasons, we will use MAP-Elites to generate the maps that will be used in the following chapters.

3.3.2.2 Discovering large number of ways to walk straight in a line

Methods In this experiment, the "mission" of the robot is to go forward as fast as possible and also to find many different ways to do it. Each of these different ways is a different cell of the behavioral-performance map, which has a different behavioral descriptor, and the goal of this experiment is thus to find a behavior in each cell of the map and to maximize their performance.

The behavioral descriptor is a 6-dimensional vector that corresponds to the proportion of time that each leg is in contact with the ground (also called duty factor). When a controller is simulated, the algorithm records at each time step (every 30 ms) whether each leg is in contact with the ground (1: contact, 0: no contact). The result is 6 Boolean time series (C_i for the i^{th} leg). The behavioral

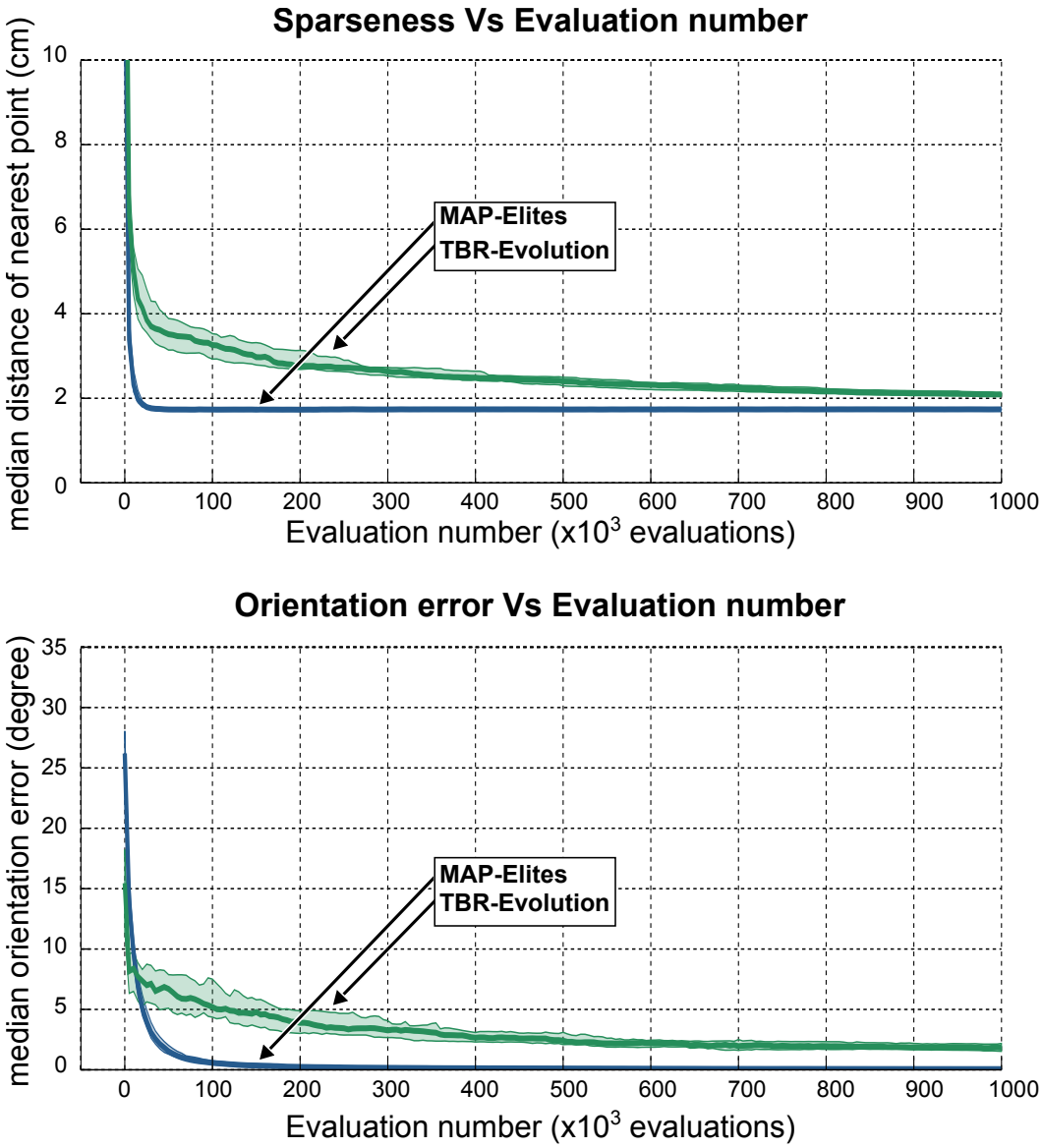


Figure 3.15: (Top) Variation of the sparseness of the behavioral repertoires or maps. For each point of a one-centimeter grid inside the ROI (Fig. 3.3), the distance from the nearest controller is computed. The sparseness value is the average of these distances. This graph plots the first three quartiles of the sparseness computed with 40 runs for each algorithm. (Bottom) Variation of the median of the orientation error over all the controllers inside the region of interest. This graph also plots the three first quartiles (25%, 50%, 75%) computed with 40 runs for each algorithm.

descriptor is then computed with the average of each time series:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t C_1(t)}{\text{numTimesteps}(C_1)} \\ \vdots \\ \frac{\sum_t C_6(t)}{\text{numTimesteps}(C_6)} \end{bmatrix} \tag{3.8}$$

During the generation of the behavior-performance map, the behaviors are stored in the maps's cells by discretizing each dimension of the behavioral descriptor space with these five values: $\{0, 0.25, 0.5, 0.75, 1\}$. With this behavioral descriptor, the robot will try to find all the ways to walk while using its six legs differently. Some behavior will use evenly the 6 legs, some of them will use only 5 or 4. Some other areas of the behavioral space require being more creative, for example it may seem impossible to walk with few or no legs. We call this behavioral descriptor the *duty-factor* descriptor. In the next chapter we will see that many other behavioral descriptors can be use to generate behavior-performance maps.

The performance of a controller is defined as how far the robot moves in a pre-specified direction in 5 seconds. The controller used in this experiment is different from the one used in the previous experiment. This new controller, detailed in appendix A.2.2, has additional parameters that define the duty cycle of each joint's periodic motion. The duty cycle is the proportion of one period in which the joint is in its higher position. In total, this controller is defined by 36 parameters (6 per leg).

Control Experiment The MAP-Elites algorithm is a stochastic search algorithm that attempts to fill a discretized map with the highest-performing solution at each point in the map. As explained previously, each cell of the map represents a different type of behavior, as defined by the behavioral dimension of the map. MAP-Elites generates new candidate points by randomly selecting a location in the map, changing the parameters of the controller that is stored there, and then saving that controller in the appropriate map location if it is better than the current occupant at that location. Intuitively, generating new candidate solutions from the best solutions found so far should be better than generating a multitude of controllers randomly and then keeping the best one found for each location in the map. In this section we report on experiments that confirm that intuition.

To understand the advantages of MAP-Elites over random sampling, we compared the two algorithms by generating data with the simulated hexapod. The experiments have the same virtual robot, environment, controller, performance function, and behavioral descriptors as in the main experiments. We analyzed the number of cells for which a solution is found (an indication of the diversity of behavior types the algorithms generate), the average performance of behaviors in the map, and the maximum performance discovered.

We replicated each experiment 8 times, each of which included 20 million evaluations on the simulated robot. Note that the 8 replications of MAP-Elites are

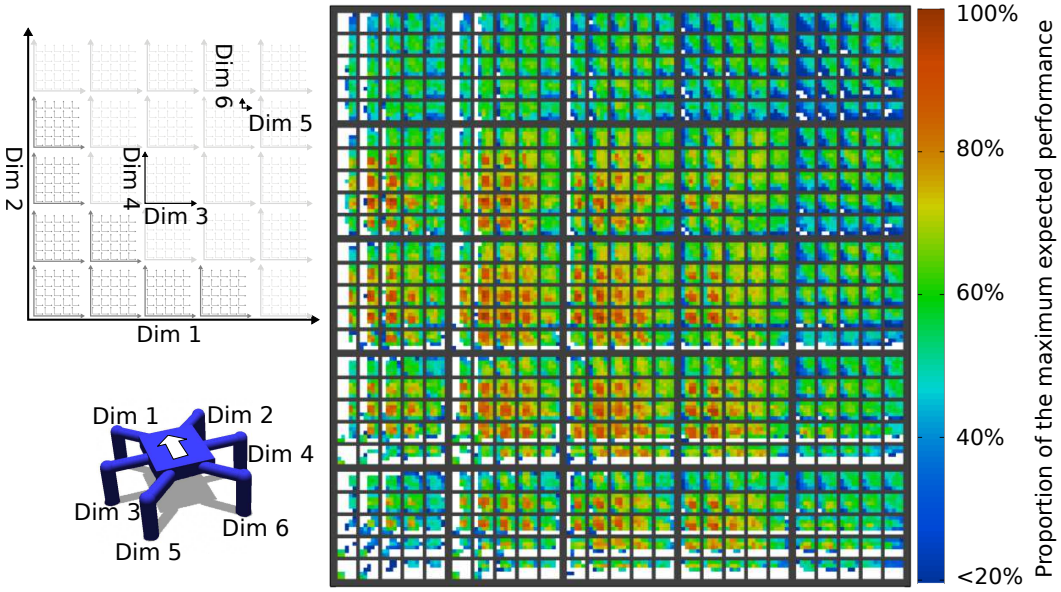


Figure 3.16: An example behavior-performance map. This map stores high-performing behaviors at each point in a six-dimensional behavior space. Each dimension is the portion of time that each leg is in contact with the ground. The behavioral space is discretized at five values for each dimension (0; 0.25; 0.5; 0.75; 1). Each colored pixel represents the highest-performing behavior discovered during map creation at that point in the behavior space. The matrices visualize the six-dimensional behavioral space in two dimensions according to the legend in the top-left. The behavior-performance map is created with a simulated robot (bottom left) in the Open Dynamics Engine physics simulator (http://www.ode.org). The matrix is a map produced by the MAP-Elites algorithm.

then allowed to continue for a total of 40 millions of evaluations to ensure a complete convergence of the algorithm, while only 20 million evaluations are enough to compare MAP-Elites and a random sampling.

Results One of the eight maps produced with MAP-Elites after 40 million evaluations can be seen in figure 3.16. We can see that a very large portion of the map is filled, as only a few cells remain in white. The maps contain about 13,000 different behaviors. A video that illustrates some of the different types of behavior that can be found in the map can be seen here: `https://youtu.be/IHQgnpSphEI` Moreover, some regions of the map are filled while they are supposed to be impossible or very challenging. For example, the most extreme bottom left cell represents the behavioral descriptor with all its values set to zero, meaning that the robot has to find a way to walk without touching the ground. The results show that the algorithm was able to find a solution even in this seemingly impossible case. After looking to the corresponding behaviors, it appears that the robot found a way to flip over and to walk on its knees. As the contact with the ground, used to compute the

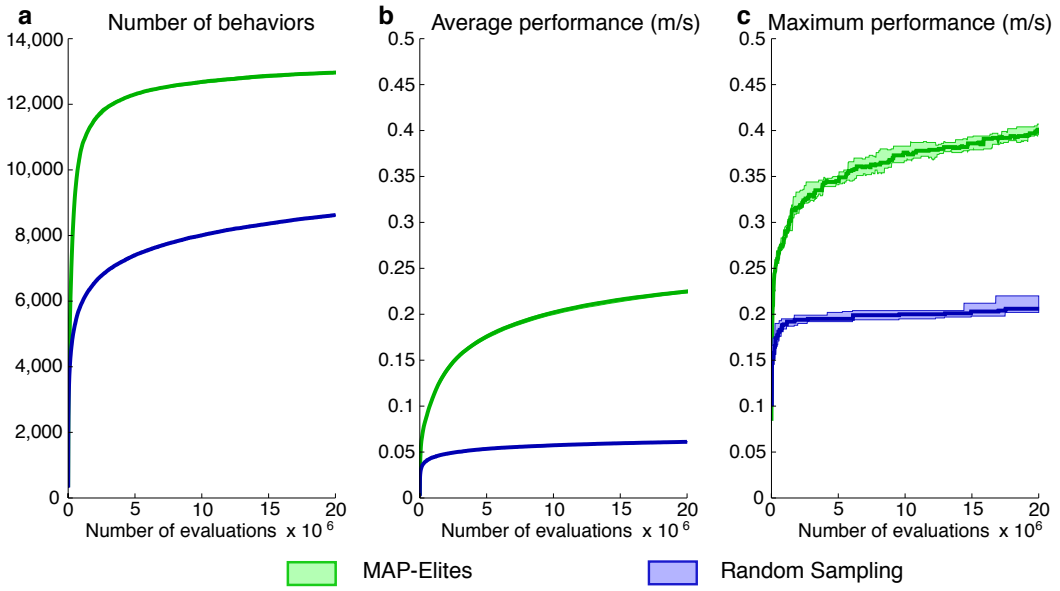


Figure 3.17: **Comparing MAP-Elites and random sampling for generating behavior-performance maps. (a) The number of points in the map for which a behavior is discovered. (b) The mean performance of the behaviors in the map. (c) The maximum performance of the behaviors in the map.** For all these figures, the middle lines represent medians over 8 independently generated maps and the shaded regions extend to the 25^{th} and 75^{th} percentiles, even for (a) and (b), where the variance of the distribution is so small that it is difficult to see. See Supplementary Experiment S4 for methods and analysis.

behavioral descriptor, are only recorded on the tips of each leg, walking on its knees is considered by the algorithm as not touching the ground. This type of behavior typically illustrates the creativity of MAP-Elites and how it is able to find solutions to various problems. This behavior can be seen in the previous video.

When comparing MAP-Elites with a random sampling, the results show that the MAP-Elites algorithm outperforms random sampling on each of the proposed measures (Fig. 3.17). After 20 million evaluations, about 13,000 cells (median: 12968, 5^{th} & 95^{th} percentiles: [12892; 13018]) are filled by MAP-Elites (about 83% percent of the map), whereas random sampling only filled approximately 8600 (8624 [8566; 8641]) cells (about 55% percent of the map) (Fig. 3.17a). The difference between the two algorithms is large and appears early (Fig. 3.17a); after only 1 million evaluations, MAP-Elites filled 10670 [10511; 10775] cells (68% of the map), whereas random sampling filled 5928 [5882; 5966] cells (38% of the map).

The solutions discovered by MAP-Elites are not only more numerous, but also outperform those found by random sampling (Supplementary Fig. 3.17b): with MAP-Elites, after 20 million evaluations the average performance of filled cells is 0.22 [0.22; 0.23] m/s, whereas it is 0.06 [0.06; 0.06] m/s with random sampling, which is similar to the performance obtained with the reference controller on a

damaged robot (Fig. 3). These two results demonstrate that MAP-Elites is a much better algorithm than random sampling to find a map of the diverse, "elite" performers in a search space.

In addition, MAP-Elites is a better optimization algorithm, as measured by the performance of the best single solution produced. The performance of the best solution in the map after 20 million evaluations is 0.40 [0.39;0.41] m/s with MAP-Elites, compared to 0.21 [0.20; 0.22] m/s with random sampling.

With this second experiment, we showed that MAP-Elites can be used not only to learn to achieve several tasks (like walking in every directions) but also to learn many different ways to solve the same task (here moving forward as fast as possible). Our experiments demonstrated that MAP-Elites is higher performing than TBR-Evolution and consequently than other algorithms of quality-diversity search (like Novelty Search or Novelty Search with Local Competition). They also proved that one of the main advantage of MAP-Elites comes from the fact that MAP-Elites can rely on its whole map as a sort of population, while TBR-Evolution (and Novelty Search) only uses the current most novel behaviors.

3.4 Conclusion

In this chapter, we introduced two algorithms to generate behavioral repertoires or maps: TBR-Evolution and MAP-Elites. These algorithms generate a large number of efficient behaviors without requiring to learn each of them separately (with independent learning processes), or to test complex controllers for each condition. Behavioral repertoires or behavior-performance maps are an interesting way to encapsulate the creativity of evolutionary algorithms (EAs), as they project the large search spaces of EAs into small behavioral spaces containing only high performing solutions. If the behavioral descriptors are well chosen, the diversity of solution contained in these containers can be enough to deal with a large variety of tasks or situations. In the footsteps of Novelty Search, these new algorithms thus highlight that evolutionary robotics can be more than black-box optimization (Doncieux and Mouret, 2014): evolution can simultaneously optimize in many niches, each of them corresponding to a different, but high-performing, behavior.

We first investigated how TBR-Evolution can be used to allow a physical robot to learn to walk in every direction. With this experiment, we showed that, thanks to its ability to recycle solutions usually wasted by standard evolutionary algorithm, TBR-Evolution generates behavioral repertoires faster than by evolving each solution separately. We also showed that its archive management allows it to generate behavioral repertoires with a significantly higher quality than the Novelty Search algorithm (Lehman and Stanley, 2011a). With the TBR-Evolution algorithm, our physical hexapod robot was able to learn several hundreds of controllers with only 60 transfers of 3 seconds on the robot, which was achieved in 2.5 hours (including computation time for evolution and the SLAM algorithm). The repartition of these controllers over all the reachable space has been autonomously inferred by the al-

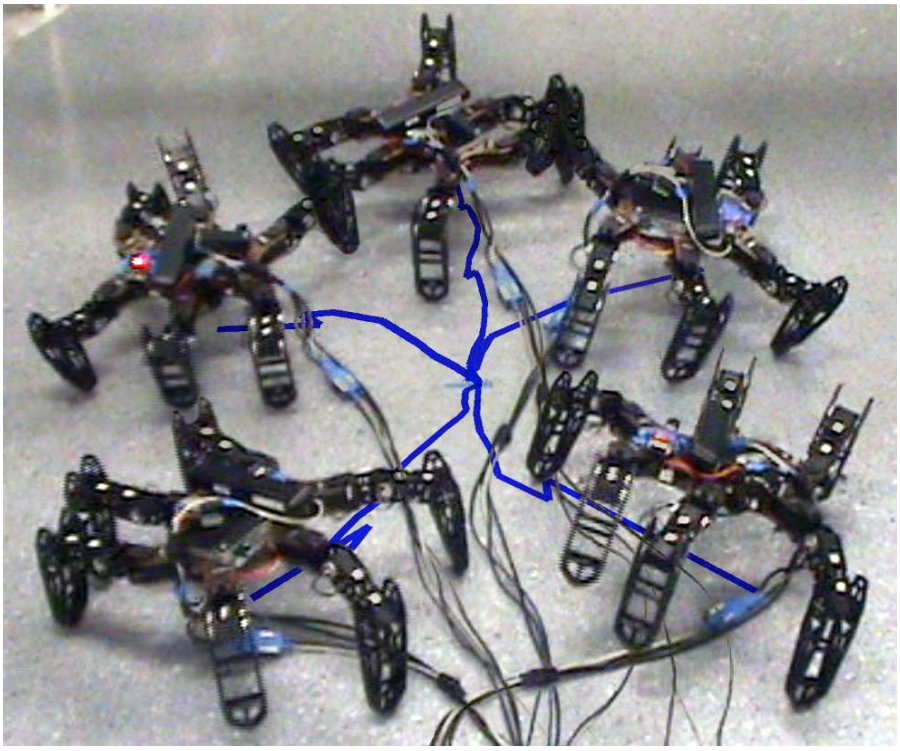


Figure 3.18: Illustration of the results obtained with TBR-Evolution. These 5 typical trajectories correspond to controllers obtained with TBR-Evolution as recorded by the SLAM algorithm. The supplementary video shows a few other examples of controllers.

gorithm according to the abilities of the robot. Our experiments also showed that our method is about 25 times faster than learning each controller separately. These experiments show that *the TBR-Evolution algorithm is a powerful method for learning multiple tasks with only several dozens of tests on the physical robot* by taking advantage of the transferability approach. Figure 3.18 and the supplementary video illustrate the resulting ability of the robot to walk in every direction.

In the second part of this chapter, we introduced MAP-Elites. We compared its performance to those of TBR-Evolution and random sampling. The results showed that this simple algorithm provides impressive results. On the same task, MAP-Elites generates a map 5 times faster than TBR-Evolution and with a quality, in terms of orientation error, 30 times better. We tested MAP-Elites in a second experiment in which the robot has to find different ways for walking straight in a line as fast as possible. This experiment highlighted the creativity of this algorithm, as some of the produced behaviors showed non-trivial strategies that solved seemingly impossible tasks. We will show in the next chapter how these maps, and the creativity they contain, can be used to allow our physical robots to learn and adapt in less than a dozen trials.

Chapter 4

Damage Recovery

The results and text of this chapter have been partially published the following articles.

Main articles:

- **Cully, A.**, Clune, J., Tarapore, D., and Mouret, J. B. (2015). Robots that can adapt like animals. Nature, 521(7553), 503-507.
- Koos, S., **Cully, A.**, and Mouret, J. B. (2013). Fast damage recovery in robotics with the t-resilience algorithm. The International Journal of Robotics Research, 32(14), 1700-1723.

Other contributors:

- Jean-Baptiste Mouret, Pierre and Marie Curie University (Thesis supervisor)
- Danesh Tarapore, Pierre and Marie Curie University (Post-doc)
- Sylvain Koos, Pierre and Marie Curie University (Post-doc)
- Jeff Clune, University of Wyoming (Assistant Prof.)

Author contributions:

- for the paper in Nature: **A.C.** and J.-B.M. designed the study. **A.C.** and D.T. performed the experiments. **A.C.**, J.-B.M., D.T. and J.C. analyzed the results and wrote the paper.
- for the paper in IJRR: S.K. and J.-B.M. designed the study. **A.C.** and S.K. performed the experiments. S.K., **A.C.**, and J.-B.M. analyzed the results and wrote the paper.

Contents

4.1 Introduction

Autonomous robots are inherently complex machines that have to cope with a dynamic and often hostile environment. They face an even more demanding context when they operate for a long time without any assistance, whether when exploring remote places (Bellingham and Rajan, 2007) or, more prosaically, in a house without any robotics expert (Prassler and Kosuge, 2008). As famously pointed out by Corbato (2007), when designing such complex systems, "[we should not] wonder *if* some mishap may happen, but rather ask *what* one will do about it when it occurs". In autonomous robotics, this remark means that robots must be able to pursue their mission in situations that have not been anticipated by their designers. Legged robots clearly illustrate this need to handle the unexpected: to be as versatile as possible, they involve many moving parts, many actuators and many sensors (Kajita and Espiau, 2008); but they may be damaged in numerous different ways. These robots would therefore greatly benefit from being able to autonomously find a new behavior if some legs are ripped off, if a leg is broken or if one motor is inadvertently disconnected.

Fault tolerance and resilience are classic topics in robotics and engineering. Current damage recovery methods in deployed robots typically involves two phases: (1) performing a self-diagnosis thanks to the embedded sensors and then (2) selecting the best, pre-designed contingency plan available (Verma et al., 2004; Blanke and Schröder, 2006; Bongard et al., 2006; Kluger and Lovell, 2006; Visinsky et al., 1994; Koren and Krishna, 2007; Görner and Hirzinger, 2010; Jakimovski and Maehle, 2010; Mostafa et al., 2010; Schleyer and Russell, 2010). For instance, if a hexapod robot diagnoses that one of its legs is not reacting as expected, it can drop it and adapt the position of the other legs accordingly (Jakimovski and Maehle, 2010; Mostafa et al., 2010).

These methods undoubtedly proved their usefulness in space, aeronautics and numerous complex systems. However, such self-diagnosing robots are expensive, because self-monitoring sensors are expensive, and are difficult to design, because robot engineers cannot foresee every possible situation. Moreover, this approach often fails either because the diagnosis is incorrect (Bongard et al., 2006; Verma

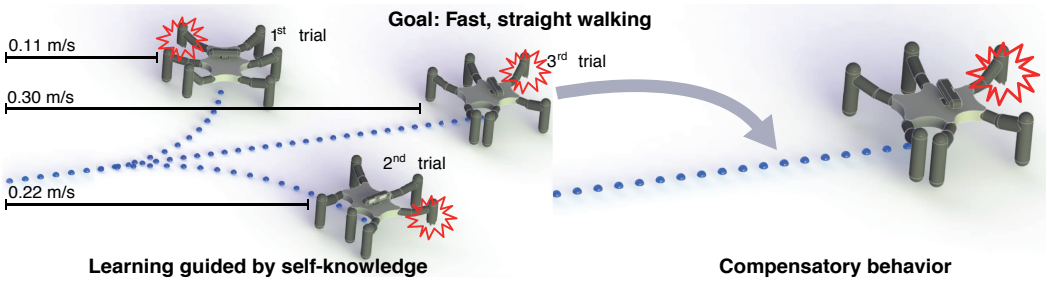


Figure 4.1: **With learning algorithms, robots, like animals, can quickly adapt to recover from damage.** After damage occurs, in this case making the robot unable to walk straight, damage recovery via Intelligent Trial and Error begins. The robot tests different types of behaviors from an automatically generated map of the behavior-performance space. After each test, the robot updates its predictions of which behaviors will perform well despite the damage. This way, the robot rapidly discovers an effective compensatory behavior.

et al., 2004) or because an appropriate contingency plan is not provided (Kluger and Lovell, 2006).

Another classic approach to fault tolerance is to employ robust controllers that can work in spite of damaged sensors or hardware inefficiencies (Goldberg and Chen, 2001; Caccavale and Villani, 2002; Qu et al., 2003; Lin and Chen, 2007). Such controllers usually do not require diagnosing the damage, but this advantage is tempered by the need to integrate the reaction to all faults in a single controller.

4.1.1 Learning for resilience

Injured animals respond differently: they learn by trial and error how to compensate for damage (e.g. learning which limp minimizes pain, Jarvis et al. (2013); Fuchs et al. (2014)). Based on this observation, an alternative to traditional recovery methods and a promising line of thought is to *let the robot learn on its own* the best behavior for the current situation (see figure 4.1). If the learning process is open enough, then the robot should be able to creatively discover compensatory behaviors without being limited to their designers' assumptions about how damage may occur and how to compensate for each damage type.

As presented previously in this manuscript, numerous learning algorithms, like policy search or evolutionary algorithms, have been experimented in robotics (see chapter 2), with different levels of openness and various a priori constraints. Nonetheless, only a few of these learning methods have been explicitly tested in situations in which a robot needs to adapt itself to unexpected situations (see table 4.1). One of the reasons that explain this lack of interest is that current state-of-the-art learning algorithms are impractical because of the "curse of dimensionality" (Kober et al., 2013): the fastest algorithms constrain the search to a few behaviors (e.g. tuning only 2 parameters, requiring 10-25 minutes) or require human demon-

strations. Algorithms without these limitations take several hours (see tables 2.1 and 4.1). Damage recovery would be much more practical and effective if robots adapted as creatively and quickly as animals (e.g. in minutes) in larger search spaces and without expensive, self-diagnosing sensors.

It is interesting to note that all policy gradient and evolutionary algorithms spend most of their running time in evaluating the quality of controllers by testing them on the target robot. Since, contrary to simulation, reality cannot be sped up, their running time can only be improved by finding strategies to evaluate fewer candidate solutions on the physical robot. In their "starfish robot" project, Bongard et al. (2006) designed a general approach for resilience that makes an important step in this direction. The authors highlighted how an internal simulation and physical tests can be combined to allow the algorithms to reduce the number of evaluations on the physical robot. The simulation of the robot is called a *self-model*[1] (Metzinger, 2004, 2007; Vogeley et al., 1999; Bongard et al., 2006; Holland and Goodman, 2003; Hoffmann et al., 2010). In effect, this algorithm transfers most of the learning time to a computer simulation, which makes it increasingly faster when computers are improved (Moore, 1975).

4.1.2 Resilience with a self-model

Taking advantage of a self-model while learning behaviors is a promising way to reduce the number of evaluation on the physical robot. However, simulations are never perfect and most of behaviors learned in simulation do not work as expected in reality. This well known problem, called reality gap (Koos et al., 2013b), is even worse when considering resilient robotics because the simulation do not model the undergone damage. In this situation, a first idea consists in updating the self-model to include the damage, which corresponds to performing a diagnosis. As mentioned previously, such diagnosis is expensive and may fail, as the number of situation that can be correctly diagnosed depends on the sensors embedded on the robot. Updating the model via a diagnosis leads consequently to the same shortcomings as traditional damage recovery algorithms.

As an alternative to the traditional ways of performing a diagnosis, Bongard et al. (2006) propose to infer the damage and to update the self-model by choosing motor actions and measuring their consequences on the behavior of the robot. The algorithm then relies on the updated model of the robot to learn a new behavior in simulation. This approach has been successfully tested on a starfish-like quadrupedal robot, which managed to update its self-model and to discover a new walking gait after the loss of one of its legs (Bongard et al., 2006; Zykov, 2008).

In Bongard's algorithm, the identification of the self-model is based on an active

[1] Following the literature in psychology (Metzinger, 2004, 2007; Vogeley et al., 1999) and artificial intelligence (Bongard et al., 2006; Holland and Goodman, 2003), we define a self-model as a forward, internal model of the *whole body* that is accessible to introspection and instantiated in a model of the environment. In the present manuscript, we only consider a minimal model of the environment (a horizontal plane).

Table 4.1: **How long many previous robot damage recovery algorithms take to run.** While comparisons between these algorithms are difficult because they vary substantially in their objective, the size of the search space, and the robot they were tested on, we nonetheless can see that damage recovery times are rarely below 20 minutes, and often take hours.

approach/article	starting behavior ⋆	evaluations	learning time	robot	DOFs†	param.‡	reward
Policy Gradient Methods							
Christensen et al. (2013)	n/a	180	10 min	quadruped	8	8	external
Evolutionary Algorithm							
Berenson et al. (2005)	random	200	2 h	quadruped	8	36	external
Mahdavi and Bentley (2006)	random	600	10 h	snake	12	1152	external
Bongard et al. (2006)[1]	random	15	4 h	hexapod	12(18)	30	internal
Reinforcement learning							
Erden and Leblebicioğlu (2008) [2]	standing	150	15-25min.	hexapod	18	n/a	internal
Our approaches							
Koos et al. (2013a) T-Resilience	random	25	20 min.	hexapod	12(18)	24	internal
Cully et al. (2015) Intelligent Trial and Error	random	8	1 min.	hexapod	12 (18)	36	internal

⋆Behavior used to initialize the learning algorithm.
† DOFs: number of controlled degrees of freedom.
‡ param: number of learned control parameters.
[1] The original authors do not provide time information, reported values come from the implementation of Koos et al. (2013)Koos et al. (2013a).
[2] Free-State generation with reinforcement learning.

learning loop that is itself divided into an *action selection loop* and a *model selection loop*. The action selection loop aims at selecting the action that will best distinguish the models of a population of candidate models, while the model selection loop looks for the models that best predict the outcomes of the actions as measured on the robot. In the "starfish" experiment (Bongard et al., 2006), the following steps are repeated:

1.1. action selection (*exploration*): Each of 36 possible actions is tested on each of the 16 candidate models to observe the orientation of robot's body predicted by the model. The action for which models of the population disagree at most is selected. This action is tested on the robot and the corresponding exact orientation of robot's body is recorded by an external camera;

1.2. model selection loop (*estimation*): An evolutionary algorithm is used to optimize the population of models so that they accurately predict what was measured with the robot, for each tested action;

Once this loop has been repeated 15 times, the best model found so far is used to learn a new behavior using an EA:

2. controller optimization (*exploitation*): An evolutionary algorithm is used to optimize a population of controllers so that they maximize forward displacement within the simulation of the self-model. The best controller found in the simulation is transferred to the robot, making it the new controller.

With these two loops, only 15 tests on the physical robot are required to update the self-model in their experiments, as most of the evaluation are executed in simulation. The speed of this algorithm directly depends on the performance of the employed computers. Nonetheless, significant computing times are still required for the optimization of the population of models.

In the results reported by Bongard et al. (2006), only half of the runs led to correct self-models. As Bongard's. approach implies identifying a full model of the robot, it would arguably require many more tests to be reasonably certain to find the correct self-model. For comparison, results obtained by the same authors but in a simulated experiment required from 600 to 1500 tests to consistently identify the model (Bongard and Lipson, 2005). It should also be noted that these authors did not measure the orientation of robot's body with internal sensors, whereas noisy internal measurements could significantly impair the identification of the model. Other authors experimented with self-modeling process similar to the one of Bongard et al., but with a humanoid robot (Zagal et al., 2009). Preliminary results suggest that thousands of evaluations on the robot would be necessary to correctly identify 8 parameters of the global self-model. Alternative methods have been proposed to build self-models for robots and all of them require numerous tests, e.g. on a manipulator arm with about 400 real tests (Sturm et al., 2008) or on a hexapod robot with about 240 real tests (Parker, 2009). Overall, experimental costs

for building self-models appear expensive in the context of resilience applications in both the number of tests on the real robot and in computing time.

Moreover, the approach has a few important shortcomings. First, actions and models are undirected: the algorithm can "waste" a lot of time to improve parts of the self-model that are irrelevant for the task. Second, it is computationally expensive because it includes a full learning algorithm (the second stage, in simulation) and an expensive process to select each action that is tested on the robot. Third, there is often a "reality gap" between a behavior learned in simulation and the same behavior on the target robot (Jakobi et al., 1995; Zagal et al., 2004; Koos et al., 2013b), but nothing is included in Bongard's algorithm to prevent such gap to happen: the controller learned in the simulation stage may not work well on the real robot, even if the self-model is accurate.

4.1.3 Dealing with imperfect simulators to make robots more robust

Instead of updating or improving the simulation, we propose in this chapter that a damaged robot discovers new original behaviors *using the initial, hand-designed self-model*. The model of the undamaged robot is obviously not accurate because it does not model the damage. Nonetheless, since damage cannot radically change the overall morphology of the robot, this "undamaged" self-model can still be viewed as a reasonably accurate model of the damaged robot. Most of the degrees of freedom are indeed correctly positioned, the mass of components should not change much and the body plan is most probably not radically altered.

Based on this observation, we hypothesis that there exist some behaviors in the search space that work similarly both with the virtual, intact robot and with the physical, damaged robot. For instance, if a hexapod robot breaks a leg, then gaits that do not critically rely on this leg should lead to similar trajectories in the self-model and on the damaged robot. Such gaits are numerous: those that make the simulated robot lift the broken leg so that it never hits the ground; those that make the robot walk on its "knees"; those that are robust to leg damage because they are closer to crawling than walking. Similar ideas can be found for most robots and for most mechanical and electrical damage, provided that there are different ways to achieve the mission. For example, any redundant robotic manipulator with a blocked joint should be able to follow a less efficient but working trajectory that does not use this joint.

The algorithms proposed in this chapter aim to find behaviors that work similarly in simulation and in reality, in spite of the damage. However, finding such behaviors is not a question restricted to damage recovery: it is a well known and an unavoidable issue when robotic controllers are first optimized in simulation then transferred to a real robot, even when the robot is intact. Evolutionary robotics is probably one of the most affected fields, because of his emphasis on opening the search space as much as possible and as a consequence, behaviors found within the simulation are often not anticipated by the designer of the simulator. Therefore, it's

not surprising that they are often wrongly simulated. Researchers in evolutionary robotics explored three main ideas to cross this "reality gap": (1) automatically improving simulators (Bongard et al., 2006; Pretorius et al., 2012; Klaus et al., 2012), (2) trying to prevent optimized controllers from relying on the unreliable parts of the simulation (in particular, by adding noise) (Jakobi et al., 1995), and (3) model the difference between simulation and reality (Hartland and Bredeche, 2006; Koos et al., 2013b).

Translated to resilient robotics, the first idea is equivalent to improving or adapting the self-model, with the aforementioned shortcomings. The second idea corresponds to encouraging the robustness of controllers so that they can deal with an imperfect simulation. It could lead to improvements in resilient robotics but it requires that the designer anticipates most of the potential damage situations. The third idea is more interesting for resilient robotics because it acknowledges that simulations are never perfect and mixes reality and simulation during the optimization. Among the algorithms of this family, the recently-proposed transferability approach (see chapter 2.2.5 or Koos et al. (2013b)) explicitly searches for high-performing controllers that work similarly in both simulation and reality. It led to successful solutions for quadruped robot (2 parameters to optimize) and for a Khepera-like robot in a T-maze (weights of a feed-forward neural networks to optimize) (Koos et al., 2013b; Koos and Mouret, 2011).

In this chapter, we present two algorithms that follow this idea of modeling the difference between the simulation and reality to allow robots to adapt to unanticipated situations. The first algorithm, named T-Resilience, is based on the transferability approach, which uses a machine-learning algorithm (regression algorithm) to guide an evolutionary algorithm towards solutions that are transferable. In our second algorithm, named the Intelligent Trial and Error algorithm, we do the opposite: a behavioral repertoire (see chapter 3) generated by an evolutionary algorithm is used to guide a machine learning algorithm (policy search algorithm) toward solutions that are both high-performing and diverse.

In both of these algorithms, the vast majority of controller evaluations is performed in simulation and only a few physical trials are required to allow our robots to adapt to the damage situations. We evaluate both of the algorithms on our hexapod robot injured in several ways, including damaged, broken, and missing legs. Results show that only 25 trials and 20 minutes are required for the robot to adapt with the T-Resilience algorithm while less than 2 minutes are enough with the Intelligent Trial and Error algorithms. For all these experiments, the performance of the behaviors are assessed on-board thanks to a RGB-D sensor coupled with state-of-the-art Simultaneous Localization And Mapping (SLAM, Durrant-Whyte and Bailey (2006); Angeli et al. (2009)) algorithms (Endres et al., 2012; Dryanovski et al., 2013). These two algorithms can work with many different types of robot, provided that they have several ways to achieve their tasks. To highlight this property, we also tested the Intelligent Trial and Error algorithm on a robotic arm with joints broken in 14 different ways.

4.2 The T-Resilience algorithm

4.2.1 Motivations and principle

As mentioned in section 2.2.5, the transferability approach captures the differences between the self-model and reality through the *transferability function* (Mouret et al., 2012; Koos et al., 2013b):

Definition 9 (transferability function) *A* transferability function $\mathscr{T}$ *is a function that maps a vector* $\mathbf{b} \in \mathbb{R}^m$ *of* m *solution descriptors (e.g. control parameters or behavior descriptors) to a* transferability score $\mathscr{T}(\mathbf{b})$ *that represents how well the simulation matches the reality for this solution (e.g. performance variation):*

$$\begin{array}{rccc} \mathscr{T}: & \mathbb{R}^m & \mapsto & \mathbb{R} \\ & \mathbf{b} & \mapsto & \mathscr{T}(\mathbf{b}) \end{array}$$

This function is usually not accessible because this would require testing every solution both in reality and in simulation (see Mouret et al. (2012) and Koos et al. (2013b) for an example of exhaustive mapping). The transferability function can, however, be approximated with a regression algorithm (neural networks, support vector machines, etc.) by recording the behavior of a few controllers in reality an in simulation.

To cross the reality gap, the transferability approach essentially proposes optimizing both the approximated transferability and the performance of controllers with a stochastic multi-objective optimization algorithm. This approach can be adapted to make a robot resilient by seeing the original, "un-damaged" self-model as an inaccurate simulation of the damaged robot, and if the robot only uses internal measurements to evaluate the discrepancies between predictions of the self-model and measures on the real robot. Resilient robotics is thus a related, yet new application of the transferability concept. We call this new approach to resilient robotics "T-Resilience" (for Transferability-based Resilience).

4.2.2 Method description

The T-Resilience algorithm relies on three main principles (Fig. 4.3 and Algorithm 3):

- the self-model of the robot is not updated;
- the approximated transferability function is learned "on the fly" thanks to a few periodic tests conducted on the robot and a regression algorithm;
- three objectives are optimized simultaneously:

$$\text{maximize} \begin{cases} \mathscr{F}_{self}(\mathbf{c}) \\ \hat{\mathscr{T}}(\mathbf{b}_{self}(\mathbf{c})) \\ diversity(\mathbf{c}) \end{cases}$$

where $\mathscr{F}_{self}(\mathbf{c})$ denotes the performance of the candidate solution $\mathbf{c}$ that is predicted by the self-model (e.g. the forward displacement in the simulation); $\mathbf{b}_{self}(\mathbf{c})$ denotes the behavior descriptor of $\mathbf{c}$, extracted by recording the behavior of $\mathbf{c}$ in the self-model; $\hat{\mathscr{T}}(\mathbf{b}_{self}(\mathbf{c}))$ denotes the approximated transferability function between the self-model and the damaged robot, which is separately learned using a regression algorithm; and $diversity(\mathbf{c})$ is a application-dependent helper-objective that helps the optimization algorithm to mitigate premature convergence (see chapter 2.2.2.1 or Toffolo and Benini (2003); Mouret and Doncieux (2012)).

Evaluating these three objectives for a particular controller does not require any real test: the behavior of each controller and the corresponding performance are predicted by the self-model; the approximated transferability value is computed thanks to the regression model of the transferability function. *The update of the approximated transferability function is therefore the only step of the algorithm that requires a real test on the robot.* Since this update is only performed every N iterations of the optimization algorithm, only a handful of tests on the real robot have to be done.

At a given iteration, the T-Resilience algorithm does not need to predict the transferability of the whole search space, it only needs these values for the candidate solutions of the current population. Since the population, on average, moves towards better solutions, the algorithm has to periodically update the approximation of the transferability function. To make this update simple and unbiased, we chose to select the solution to be tested on the robot by picking a random individual from the population. We experimented with other selection schemes in preliminary experiments, but we did not observe any significant improvement.

The three objectives are simultaneously optimized thank to the NSGA-II algorithm (Deb et al., 2002; Deb, 2001), one of the most widely used multi-objective optimization algorithm (see section 2.2.2); however, any Pareto-based multi-objective algorithm can replace this specific EA in the T-Resilience algorithm.

At the end of the optimization algorithm, the MOEA discards diversity values and returns a set of non-dominated solutions based on performance and transferability. We then need to choose the final controller. Let us define the "transferable non-dominated set" as the set of non-dominated solutions whose transferability values are greater than a user-defined threshold. To determine the best solution of a run, the solution of the transferable non-dominated set with the highest performance in simulation is transferred onto the robot and its performance in reality is assessed. The final solution of the run is the controller that leads to the highest performance on the robot among all the transferred controllers (Fig. 4.2).

Three choices depend on the application:

- the performance measure $\mathscr{F}_{self}$ (i.e. the reward function);
- the diversity measure;
- the regression technique used to learn the transferability function and, in particular, the inputs and outputs of this function.

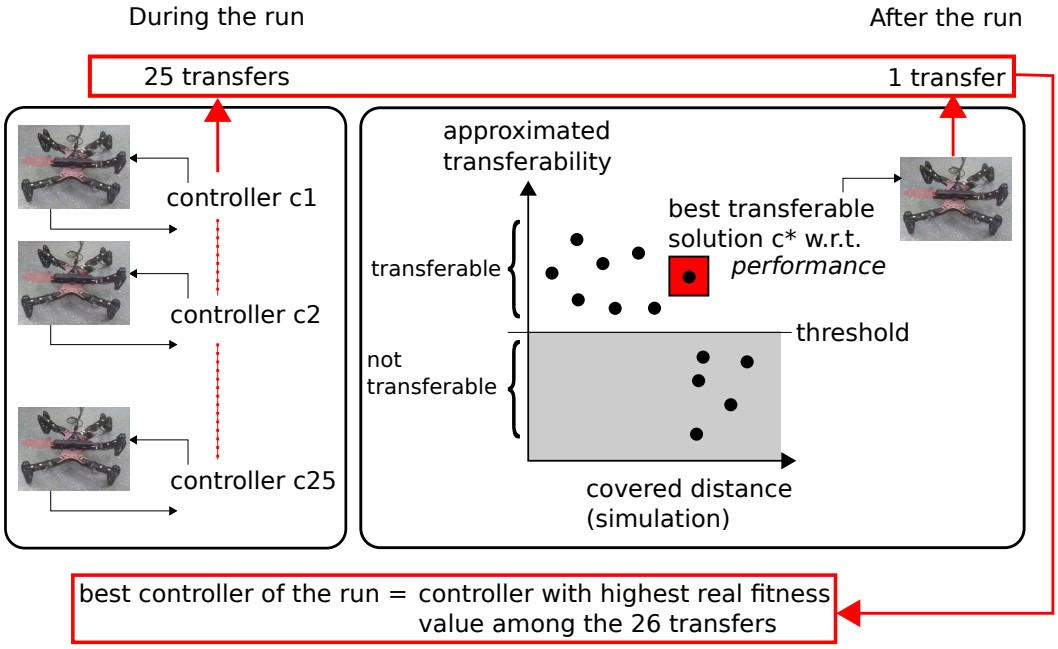


Figure 4.2: Choice of the final solution at the end of the T-Resilience algorithm.

We will discuss and describe each of these choices for our resilient hexapod robot in the next section.

Algorithm 3 T-Resilience (T real tests)

$pop \leftarrow \{c^1, c^2, \ldots, c^S\}$ (randomly generated)
$data \leftarrow \emptyset$

for $i = 1 \rightarrow T$ **do**

random selection of c^* in pop	
computation of $\mathbf{b}_{self}(c^*)$, vector of m values describing c^* in the self-model	
transfer of c^* on the robot	
estimation of performance $\mathscr{F}_{real}(c^*)$ using internal measurements	(B)
estimation of transferability score $\mathscr{T}(\mathbf{b}_{self}(c^*)) = \|\|\mathscr{F}_{self}(c^*) - \mathscr{F}_{real}(c^*)\|\|$	
$data \leftarrow data \cup \{[\mathbf{b}_{self}(c^*), \mathscr{T}(\mathbf{b}_{self}(c^*))]\}$	
learning of new approximated transferability function $\hat{\mathscr{T}}$, based on $data$	
N iterations of MOEA on pop by maximizing $\mathscr{F}_{self}(c)$, $\hat{\mathscr{T}}(\mathbf{b}_{self}(c))$, $diversity(c)$	(A)

selection of the new controller

4.2.3 Experimental validation

We evaluate the performance of our algorithm on the same hexapod robot and the same type of controller as in the previous chapter. Like in the previous experiments, the performance of the robot is assessed thanks to an embedded SLAM algorithm, which autonomously infers the walking speed of the robot. The self-model of the robot is also identical to the simulation used in the previous chapter. The details

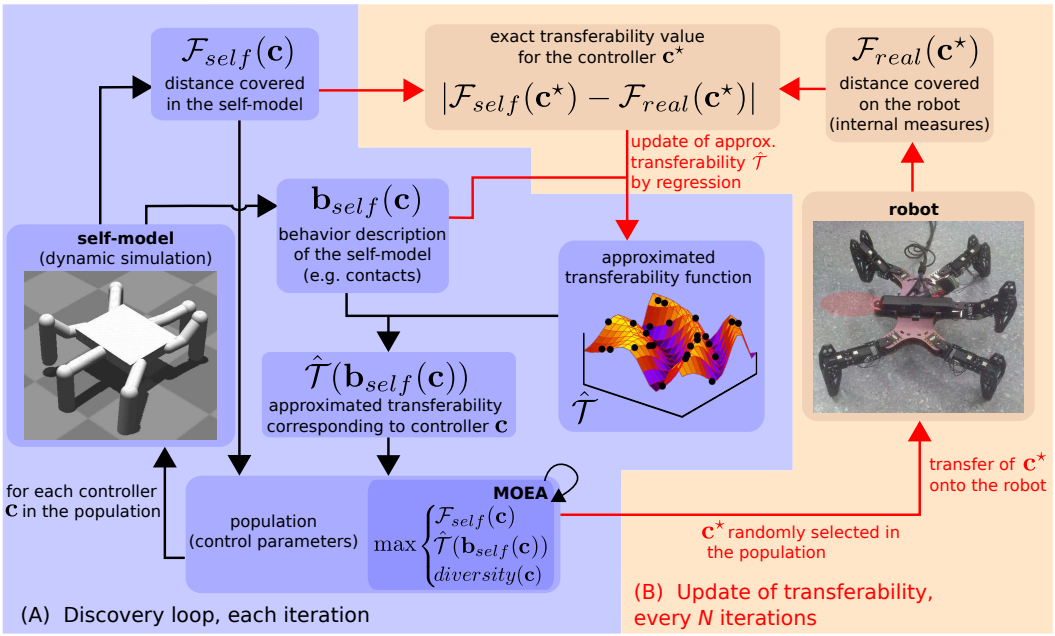


Figure 4.3: Schematic view of the T-Resilience algorithm (see algorithm 3 for an algorithmic view). (A) Discovery loop: each controller of the population is evaluated with the self-model. Its transferability score is approximated according to the current model $\hat{\mathscr{T}}$ of the exact transferability function $\mathscr{T}$. (B) Transferability update: every N iterations, a controller of the population is randomly selected and transferred onto the real robot. The model of the transferability function is next updated with the data generated during the transfer.

about the robots and the employed encoding for the controller and the self-model can be found in the appendix A.1 and A.2.1.

A classic tripod gait (Wilson, 1966; Saranli et al., 2001; Schmitz et al., 2001; Ding et al., 2010; Steingrube et al., 2010), designed by using the same controller, is used as a reference point to compare the performance of the proposed algorithm compared to hand-coded behaviors. The description and the parameters of this behavior are available in appendix A.2.1.

4.2.3.1 Implementation choices for T-Resilience

Performance function. The mission of our hexapod robot in this experiment is to go forward as fast as possible, regardless of its current state and of the undergoing damage. The performance function to be optimized is the forward displacement of the robot predicted by its self-model. Such a high-level function does not constrain the features of the optimized behaviors, so that the search remains as open as possible, possibly leading to original gaits (Nelson et al., 2009):

$$\mathscr{F}_{self}(\mathbf{c}) = p_x^{t=E,SELF}(\mathbf{c}) - p_x^{t=0,SELF}(\mathbf{c}) \tag{4.1}$$

where $p_x^{t=0,SELF}(\mathbf{c})$ denotes the x-position of the robot's center at the beginning of the simulation when the parameters $\mathbf{c}$ are used and $p_x^{t=E,SELF}(\mathbf{c})$ its x-position at the end of the simulation.

Because each trial lasts only a few seconds, this performance function does not strongly penalize gaits that do not lead to straight trajectories. Using longer experiments would penalize these trajectories more, but it would increase the total experimental time too much to perform comparisons between approaches. Other performance functions are possible and will be tested in future work.

Diversity function. The diversity score of each individual is the average Euclidean distance to all the other candidate solutions of the current population. Such a parameter-based diversity objective enhances the exploration of the control space by the population (Toffolo and Benini, 2003; Mouret and Doncieux, 2012) and allows the algorithm to avoid many local optima. This diversity objective is straightforward to implement and does not depend on the task.

$$diversity(\mathbf{c}) = \frac{1}{N} \sum_{y \in P_n} \sqrt{\sum_{j=1}^{24} (\mathbf{c}_j - \mathbf{y}_j)^2} \tag{4.2}$$

where P_n is the population at generation n, N the size of P and $\mathbf{c}_j$ the j^{th} parameter of the candidate solution $\mathbf{c}$. Other diversity measures (e.g. behavioral measures, like in (Mouret and Doncieux, 2012)) led to similar results in preliminary experiments.

Regression model. When a controller $\mathbf{c}$ is tested on the real robot, the corresponding exact transferability score $\mathscr{T}$ is computed as the absolute difference between the forward performance predicted by the self-model and the performance estimated on the robot based on the SLAM algorithm.

$$\mathscr{T}(\mathbf{c}) = \left| p_{t=E}^{SELF}(\mathbf{c}) - p_{t=0}^{REAL}(\mathbf{c}) \right| \tag{4.3}$$

The transferability function is approximated by training a SVM model $\hat{\mathscr{T}}$ using the ν-Support Vector Regression algorithm with linear kernels implemented in the library *libsvm*[2] (Chang and Lin, 2011) (learning parameters are set to default values).

$$\hat{\mathscr{T}}(\mathbf{b}_{self}(\mathbf{c})) = \text{SVM}(b_{t=0}^{(1)}, \cdots, b_{t=E}^{(1)}, \cdots, b_{t=0}^{(6)}, \cdots, b_{t=E}^{(6)}) \tag{4.4}$$

where E is the number of time-steps of the control function (equation A.1) and:

$$b_t^{(n)} = \begin{cases} 1 & \text{if leg } n \text{ touches the ground at that time-step} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

[2] http://www.csie.ntu.edu.tw/~cjlin/libsvm

We chose to describe gaits using contacts[3], because it is a classic representation of robotic and animal gaits (e.g. Delcomyn (1971)).

We chose SVMs to approximate the transferability score because of the high number of inputs of the model and because there are many available implementations. Contrary to other classic regression models (neural networks, Kriging, ...), SVMs are indeed not critically dependent on the size of the input space (Smola and Vapnik, 1997; Smola and Schölkopf, 2004). They also provide fast learning and fast prediction when large input spaces are used.

4.2.3.2 Test cases and compared algorithms

To assess the ability of T-Resilience to cope with many different failures, we consider the six following test cases (Fig. 4.4):

- A. the hexapod robot is not damaged;
- B. the left middle leg is no longer powered;
- C. the terminal part of the front right leg is shortened by half;
- D. the right hind leg is lost;
- E. the middle right leg is lost;
- F. both the middle right leg and the front left leg are lost.

We compare the T-Resilience algorithm to three representative algorithms from the literature (see appendix D.1.1 for the exact implementations of each algorithm and appendix D.1.2 for the validation of the implementations):

- a stochastic local search (Hoos and Stützle, 2005), because of its simplicity;
- a policy gradient method inspired from Kohl and Stone (2004), because this algorithm has been successfully applied to learn quadruped locomotion;
- a self-modeling process inspired from Bongard et al. (2006).

To make the comparisons as fair as possible, we designed our experiments to compare algorithms after the same amount of running time or after the same number of real tests (see appendix D.1.3 for their median durations and their median numbers of real tests). In all the test cases, the T-Resilience algorithm required about 19 minutes and 25 tests on the robot (1000 generations of 100 individuals). Consequently, two key values are recorded for each algorithm (see Appendix D.1.1 for exact procedures):

[3]When choosing the input of a predictor, there is a large difference between using the control parameters and using high-level descriptors of the behavior (Mouret and Doncieux, 2012). Intuitively, most humans can predict that a behavior will work on a real robot by watching a simulation, but their task is much harder if they can only see the parameters. More technically, predicting features of a complex dynamical system usually requires simulating it. By starting with the output of a simulator, the predictor avoids the need to re-invent physical simulation and can focus on discrimination.

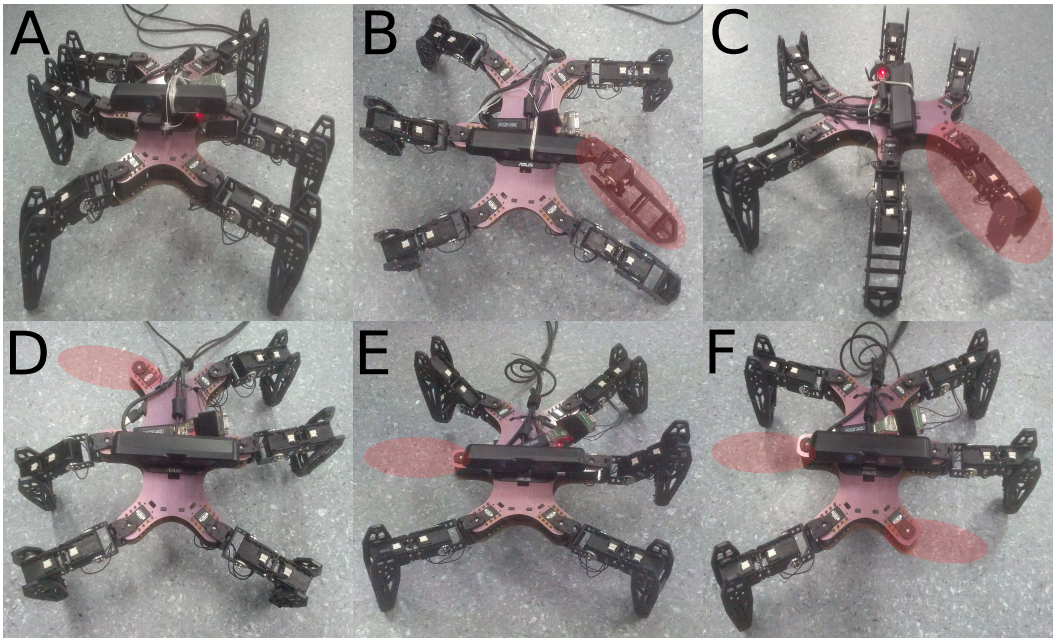


Figure 4.4: Test cases considered in our experiments. (A) The hexapod robot is not damaged. (B) The left middle leg is no longer powered. (C) The terminal part of the front right leg is shortened by half. (D) The right hind leg is lost. (E) The middle right leg is lost. (F) Both the middle right leg and the front left leg are lost.

- the performance of the best controller obtained after about 25 real tests[4];
- the performance of the best controller obtained after about 19 minutes.

The experiments for the four first cases (A, B, C and D) showed that only the stochastic local search is competitive with the T-Resilience. To keep experimental time reasonable, we therefore chose to only compare T-Resilience with the local search algorithm for the two last failures (E and F).

Preliminary experiments with each algorithm showed that initializing them with the parameters of the reference controller did not improve their performance. We interpret these preliminary experiments as indicating that the robot needs to use a qualitatively different gait, which requires substantial changes in the parameters. This observation is consistent with the gaits we tried to design for the damaged robot. As a consequence, we chose to initialize each of the compared algorithms with random parameters instead of initializing them with the parameters of the reference controller. By thus starting with random parameters, we do not rely on any a priori about the gaits for the damaged robot: we start with the assumption that anything could have happened.

We replicate each experiment 5 times to obtain statistics. Overall, this comparison requires the evaluation of about 4000 different controllers on the real robot. The

[4]Depending on the algorithm, it is sometimes impossible to perform exactly 25 tests (for instance, if two tests are performed for each iteration).

Test cases	A	B	C	D	E	F
Perf. (CODA)	0.78	0.26	0.25	0.00	0.15	0.10
Perf. (SLAM)	0.75	0.17	0.26	0.00	0.04	0.16

Table 4.2: Performances in meters obtained on the robot with the reference gait in all the considered test cases. Each test lasts 3 seconds. The CODA line corresponds to the distance covered by the robot according to the external motion capture system. The SLAM line corresponds to the performance of the same behaviors but reported by the SLAM algorithm. When internal measures are used (SLAM line), the robot can easily detects that a damage occurred because the difference in performance is very significant (column A versus the other columns).

different values of parameters used in these experiments are defined in appendix C.2

We use 4 Intel(R) Xeon(R) CPU E31230 3.20GHz, each of them including 4 cores. Each algorithm is programmed in the Sferes$_{v2}$ framework (Mouret and Doncieux, 2010). The MOEA used in Bongard's algorithm and in the T-Resilience algorithm is distributed on 16 cores using MPI.

Final performance values are recorded with a CODA cx1 motion capture system (Charnwood Dynamics Ltd, UK) so that reported results do not depend on inaccuracies of the internal measurements. However, all the tested algorithms have only access to the internal measurements.

4.2.4 Results

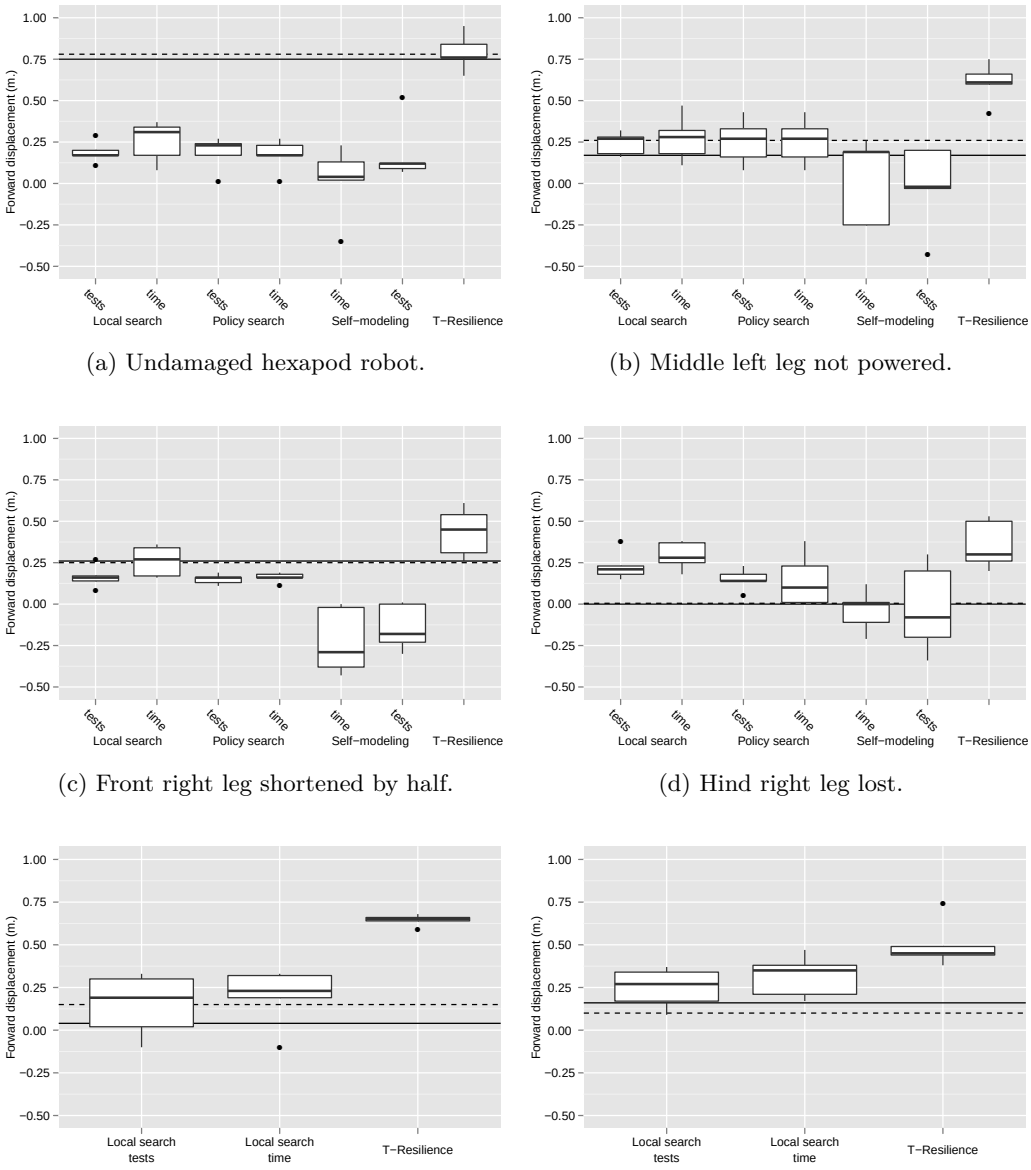
4.2.4.1 Reference controller

Table 4.2 reports the performances of the reference controller for each tested failure, measured with both the CODA scanner and the on-board SLAM algorithm. At best, the damaged robot covered 35% of the distance covered by the undamaged robot (0.78 m with the undamaged robot, at best 0.26 m after a failure). In cases B, C and E, the robot also performs about a quarter turn; in case D, it falls over; in case F, it alternates forward locomotion and backward locomotion.

This performance loss of the reference controller clearly shows that an adaptation algorithm is required to allow the robot to pursue its mission. Although not perfect, the distances reported by the on-board RGB-D SLAM are sufficiently accurate to easily detect when the adaptation algorithm must be launched.

4.2.4.2 Comparison of performances

Fig. 4.5 shows the performance obtained for all test cases and all the investigated algorithms. Table 4.3 reports the improvements between median performance values. P-values are computed with the Wilcoxon rank-sum tests (appendix D.1.4). The horizontal lines in Figure 4.5 show the efficiency of the reference gait in each case. Videos of the typical behaviors obtained with the T-Resilience on every test case are available here: `https://youtu.be/MSwdmC0dZ74`



(a) Undamaged hexapod robot.

(b) Middle left leg not powered.

(c) Front right leg shortened by half.

(d) Hind right leg lost.

(e) Middle right leg lost.

(f) Middle right leg and front left leg lost.

Figure 4.5: Performances obtained in each test case (distance covered in 3 seconds). On each box, the central mark is the median; the edges of the box are the lower hinge (defined as the 25th percentile) and the upper hinge (the 75th percentile). The whiskers extend to the most extreme data point, which is no more than 1.5 times the length of the box away from the box. Each algorithm has been run 5 times and distances are measured using the external motion capture system. Except for the T-Resilience, the performance of the controllers found after about 25 transfers (*tests*) and after about 20 minutes (*time*) are depicted (all T-Resilience experiments last about 20 minutes and use 25 transfers). The horizontal lines denote the performances of the reference gait, according to the CODA scanner (dashed line) and according to the SLAM algorithm (solid line).

	Local search		Policy search		Self-modeling		reference gait
	tests	time	tests	time	time	tests	
A	4.5	2.5	3.3	4.5	6.3	19.0	1.0
B	2.3	2.2	2.3	2.3	+++	3.2	2.3
C	2.8	1.7	2.8	2.8	+++	+++	1.8
D	1.4	1.1	2.1	3.0	+++	+++	+++
E	3.4	2.8					4.3
F	1.7	1.3					4.5
global median	2.8	2.0	2.6	2.9	+++	+++	3.3

(a) Ratios between median performance values.

	Local search		Policy search		Self-modeling		reference gait
	tests	time	tests	time	time	tests	
A	+59	+45	+53	+59	+64	+72	- 2
B	+34	+33	+34	+34	+63	+42	+35
C	+29	+18	+29	+29	+63	+74	+20
D	+ 9	+ 2	+16	+20	+38	+30	+30
E	+46	+42					+50
F	+18	+10					+35
global median	+32	+26	+32	+32	+63	+57	+33

(b) Differences between median performance values (cm).

Table 4.3: Performance improvements of the T-Resilience compared to other algorithms. For ratios, the symbol +++ indicates that the compared algorithm led to a negative or null median value.

and https://youtu.be/dncuBUnfkA4

Performance with the undamaged robot (case A). When the robot is not damaged, the T-Resilience algorithm discovered controllers with the same level of performance as the reference hexapod gait (p-value = 1). The obtained controllers are from 2.5 to 19 times more efficient than controllers obtained with other algorithms (Table 4.3).

The poor performance of the other algorithms may appear surprising at first sight. Local search is mostly impaired by the very low number of tests that are allowed on the robot, as suggested by the better performance of the "time" variant (20 minutes / 50 tests) versus the "tests" variant (10 minutes / 25 tests). Surprisingly, we did not observe any significant difference when we initialized the control parameters with those of the reference controller (data not shown). The policy gradient method suffers even more than local search from the low number of tests because a lot of tests are required to estimate the gradient. As a consequence, we were able to perform only 2 to 4 iterations of the algorithm. Overall, these results are consistent with those of the literature because previous experiments used longer experiments and often simpler systems. Similar observations have been reported previously by other authors (Yosinski et al., 2011).

Bongard's algorithm mostly fails because of the reality gap between the self-model and the real robot. Optimizing the behavior only in simulation leads – as

expected – to controllers that perform well with the self-model but that do not work on the real robot. This performance loss is sometimes high because the controllers make the robot flip over or go backward.

Resilience performance (cases B to F). When the robot is damaged, gaits found with the T-Resilience algorithm are always faster than the reference gait (p-value = 0.0625, one-sample Wilcoxon signed rank test).

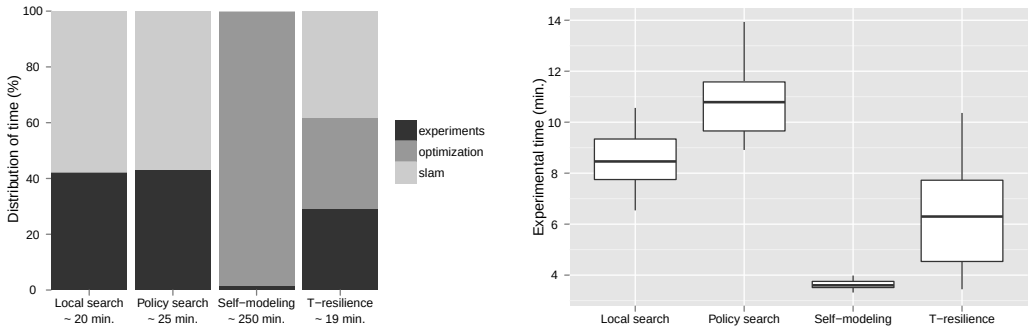
After the same number of tests (variant *tests* of each algorithm), gaits obtained with T-Resilience are at least 1.4 times faster than those obtained with the other algorithms (median of 3.0 times) with median performance values from 30 to 65 cm in 3 seconds. These improvements are all stastically significant (p-values ≤ 0.016) except for the local search in the case D (loss of a hind leg; p-value = 0.1508).

After the same running time (variant *time* of each algorithm), gaits obtained with T-Resilience are also significantly faster (at least 1.3 times; median of 2.8 times; p-values ≤ 0.016) than those obtained with the other algorithms in cases B, E and F. In cases C (shortened leg) and D (loss of a hind leg), T-Resilience is not statistically different from local search (shortened leg: p-value = 0.1508; loss of a hind leg: p-value = 0.5476). Nevertheless, these high p-values may stem from the low number of replications (only 5 replications for each algorithm). Moreover, as section 4.2.4.3 will show, the execution time of the T-Resilience can be compressed because a large part of the running time is spent in computer simulations. Consequently, depending on the hardware, better performances could be achieved in smaller amounts of time.

For all the tested cases, Bongard's self-modeling algorithm doesn't find any working controllers. We observed that it suffers from two difficulties: the optimized models do not always capture the actual morphology of the robot, and reality gaps between the self-model and the reality (see the comments about the undamaged robot). In the first case, more time and more actions could improve the result. In the second time, a better simulation model could make things better but it is unlikely to fully remove the effect of the reality gap.

Loss of a leg (case D and E). When the hind leg is lost (case D), the T-Resilience yields controllers that perform much better than the reference controller. Nevertheless, the performances of the controllers obtained with the T-Resilience are not statistically different from those obtained with the local search. This unexpected result stems from the fact that many of the transfers made the robot tilt down (fast six-legged behaviors optimized on the self-model of the undamaged robot are often unstable without one of the hind legs): in this case, the SLAM algorithm is unreliable (the algorithm often crashed) and we have to discard the distance measurements. In effect, only a dozen of transfers are usable in case D, making the estimation of the transferability function especially difficult. Using more transfers could accentuate the difference between T-Resilience and local search.

If the robot loses a less critical leg (middle leg in case E), it is more stable and the algorithm can conduct informative tests on the robot. The T-Resilience is then



(a) Distribution of duration (median duration indicated below the graph).

(b) Experimental time (experiments with the robot).

Figure 4.6: Distribution of duration and experimental time for each algorithm (median values on 5 runs of test cases A, B, C, D). All the differences between experimental times are statistically significant (p-values $< 2.5 \times 10^{-4}$ with Wilcoxon rank-sum tests).

able to find fast gaits (about 3 times faster than with the local search).

4.2.4.3 Comparison of durations and experimental time

The running time of each algorithm is divided into experimental time (actual experiments on the robot), sensor processing time (computing the robot's trajectory using RGB-D slam) and optimization time (generating new potential solutions to test on the robot). The median proportion of time allocated to each of this part of the algorithms is pictured for each algorithm on figure 4.6[5].

The durations of the SLAM algorithm and of the optimization processes both only depend on the hardware specifications and can therefore be substantially reduced by using faster computers or by parallelizing computation. Only experimental time cannot easily be reduced. The median proportion of experimental time is 29% for the T-Resilience, whereas both the policy search and the local search leads to median proportions higher than 40% for a similar median duration by run (about 20 minutes). The proportion of experimental time for the self-modeling process is much lower (median value equals to 1%) because it requires much more time for each run (about 250 minutes for each run, in our experiments).

The median experimental time of T-Resilience (6.3 minutes) is significantly lower than those of local search and of policy search (resp. 8.5 and 10.8 minutes, p-values $< 2.5 \times 10^{-4}$). With the expected increases of computational power, this difference will increase each year. The self-modeling process requires significantly lower experimental time (median at 3.6 minutes, p-values $< 1.5 \times 10^{-11}$) because

[5]Only test cases A, B, C and D are considered to compute these proportions (5 runs for each algorithm) because the policy search and the self-modeling process are not tested in test cases E and F

it only tests actions that involve a single leg, which is faster than testing a full gait (3 seconds).

4.2.5 Partial conclusion

All our experiments show that T-Resilience is a fast and efficient learning approach to discover new behaviors after mechanical and electrical damage (less than 20 minutes with only 6 minutes of irreducible experimental time). Most of the time, T-Resilience leads to gaits that are several times faster than those obtained with direct policy search, local search and Bongard's algorithm; T-Resilience never obtained worse results. Overall, T-Resilience appears to be a versatile algorithm for damage recovery, as demonstrated by the successful experiments with many different types of damage. These results validate the combination of the principles that underlie our algorithm: (1) using a self-model to transform experimental time with the robot into computational time inside a simulation, (2) learning a transferability function that predicts performance differences between reality and the self-model (instead of learning a new self-model) and, (3) optimizing both the transferability and performance to learn behaviors in simulation that will work well on the real robot, even if the robot is damaged. These principles can be implemented with alternative learning algorithms and alternative regression models.

During our experiments, we observed that the T-Resilience algorithm was less sensitive to the quality of the SLAM than the other investigated learning algorithms (policy gradient and local search). Our preliminary analysis shows that the sensitivity of these classic learning algorithms mostly stems from the fact that they optimize the SLAM measurements and not the real performance. For instance, in several of our experiments, the local search algorithm found gaits that make the SLAM algorithm greatly over-estimate the forward displacement of the robot. The T-Resilience algorithm relies only on internal sensors as well. However, these measures are not used to estimate the performance but to compute the transferability values. Gaits that lead to over-estimations of the covered distance have low transferability scores because the measurement greatly differs from the value predicted by the self-model. As a consequence, they are avoided like all the behaviors for which the prediction of the self-model does not match the measurement. In other words, the self-model acts as a "credibility check" of the SLAM measurements, which makes T-Resilience especially robust to sensor inaccuracies. If sensors were redundant, this credibility check could also be used by the robot to continue its mission when a sensor is unavailable.

4.3 The Intelligent Trial and Error algorithm

4.3.1 Motivations and principle

In the previous section, our experiments demonstrated that using a simulation of the intact robot allows the physical robot to quickly adapt to unforeseen situations. In

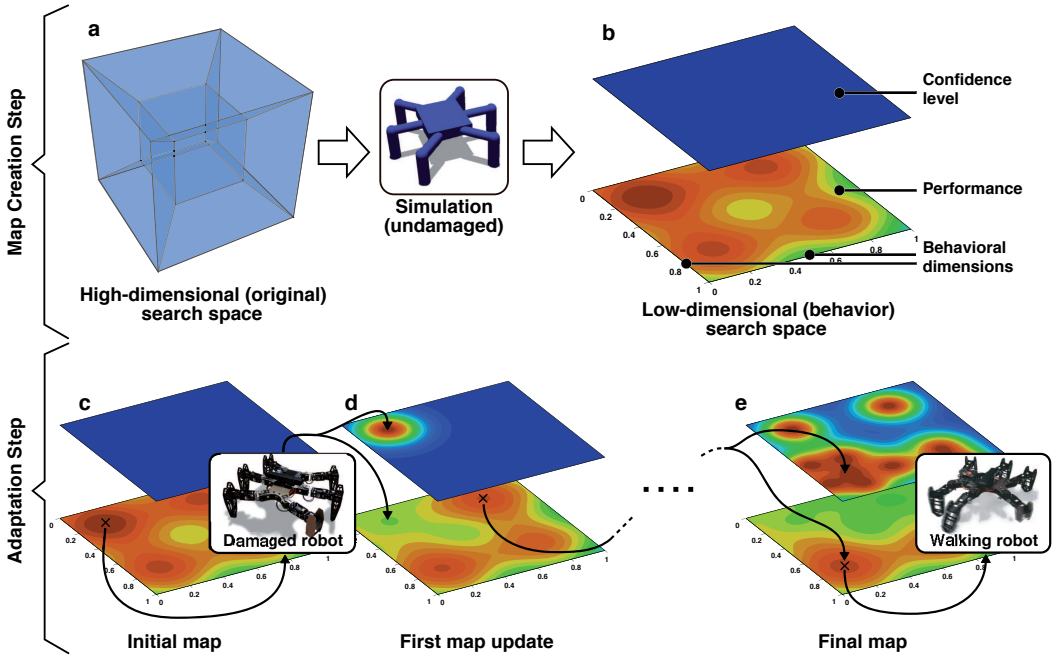


Figure 4.7: **(A & B). Creating the behavior-performance map:** A user reduces a high-dimensional search space to a low-dimensional behavior space by defining dimensions along which behaviors vary. In simulation, the high-dimensional space is then automatically searched to find a high-performing behavior at each point in the low-dimensional behavior space, creating a "behavior-performance" map of the performance potential of each location in the low-dimensional space. In our hexapod robot experiments, the behavior space is six-dimensional: the portion of time that each leg is in contact with the ground. The confidence regarding the accuracy of the predicted performance for each behavior in the behavior-performance map is initially low because no tests on the physical robot have been conducted. **(C & D) Adaptation Step:** After damage, the robot selects a promising behavior, tests it, updates the predicted performance of that behavior in the behavior-performance map, and sets a high confidence on this performance prediction. The predicted performances of nearby behaviors–and confidence in those predictions–are likely to be similar to the tested behavior and are thus updated accordingly. This select/test/update loop is repeated until a tested behavior on the physical robot performs better than 90% of the best predicted performance in the behavior-performance map, a value that can decrease with each test (Fig. 4.8). The algorithm that selects which behavior to test next balances between choosing the behavior with the highest predicted performance and behaviors that are different from those tested so far. Overall, the Intelligent Trial and Error approach presented here rapidly locates which types of behaviors are least affected by the damage to find an effective, compensatory behavior.

the T-Resilience algorithm, this speed improvement requires to be able to perform a significant amount of simulations between each trial, which significantly slows down the learning process. In the previous experiments, the robot was actually moving only 6 or 7 minutes over the 20 minutes taken by the learning process. A common way to reduce the time spent in simulation is to reduce the number of iterations in simulation. However, such approach often results in a decrease of the quality in the obtained solutions.

In this section, we introduce a second algorithm, based on the same principles as the T-Resilience algorithm, but in which all the learning time is spent in testing behaviors on the physical robot. With this algorithm, we show that rapid adaptation can be achieved by guiding a learning algorithm with an automatically generated, *pre-computed*, behavioral repertoire that predicts the performance of thousands of different behaviors. We call this behavioral repertoire a *behavior-performance map*. By pre-computing the behavior-performance map, the algorithm is no longer limited by the trade-off between time spent in simulation and quality of the solutions like with the T-Resilience algorithm, and can spend as much time as needed to improve the behaviors in simulation.

As mentioned previously, current learning algorithms either start with no knowledge of the search space (Kober et al., 2013) or with minimal knowledge from a few human demonstrations (Kober et al., 2013; Argall et al., 2009). Our intuition is that animals understand the space of possible behaviors and their value from previous experience (Wolpert et al., 2001), which they use to adapt by intelligently selecting tests that validate or invalidate whole families of promising compensatory behaviors. The key insight of our approach is that robots could do the same.

We propose to have robots that store knowledge from previous experience in the form of a map of the behavior-performance space. Guided by this map, a damaged robot tries different types of behaviors that are predicted to perform well and, as tests are conducted, updates its estimates of the performance of those types of behaviors. The process ends when the robot predicts that the most effective behavior has already been tested. We call this algorithm "Intelligent Trial and Error". This approach relies on the same hypothesis than the rest of this chapter: information about many different behaviors on the undamaged robot will still be useful after damage, because some of these behaviors will still be functional despite the damage. In addition to the results from the previous section, the results of the experiments presented here support this assumption for all types of damage we tested, revealing that a robot can quickly discover a way to compensate for damage without a detailed mechanistic understanding of its cause, as occurs with animals.

The behavior-performance map is created with the MAP-Elites algorithm presented in chapter 3.3 and a self-model of the robot, which either can be a standard physics simulator or can be automatically discovered (Bongard et al., 2006). The robot's designers only have to describe the dimensions of the space of possible behaviors and a performance measure. For instance, walking gaits could be described by how much each leg touches the ground (a behavioral measure) and speed (a performance measure). An alternate gait behavioral measure could be the percent

of time a robot's torso has positive pitch, roll, and yaw angles. For grasping, performance could be the amount of surface contact, and it has been demonstrated that 90% of effective poses for the 21-degree-of-freedom human hand can be captured by a 3-dimensional behavioral space describing the principle components of ways that hand-poses commonly vary (Santello, 1998). To fill in the behavior-performance map, the MAP-Elites algorithm simultaneously searches for a high-performing solution at each point in the behavioral space (Fig. 4.7a,b and Fig. 4.8). This step requires simulating millions of behaviors, but needs to be performed only once per robot design before deployment.

A low confidence is assigned to the predicted performance of behaviors stored in this behavior-performance map because they have not been tried in reality (Fig. 4.7b and Fig. 4.8). During the robot's mission, if performance drops below a user-defined threshold (either due to damage or a different environment), the robot selects the most promising behavior from the behavior-performance map, tests it, and measures its performance. The robot subsequently updates its prediction for that behavior and nearby behaviors, assigns high confidence to these predictions (Fig. 4.7c and Fig. 4.8), and continues the select-test-update process until it finds a satisfactory compensatory behavior (Fig. 4.7d and Fig. 4.8).

All of these ideas are technically captured via a Gaussian process model (Rasmussen and Williams, 2006), which approximates the performance function with already acquired data, and a Bayesian optimization procedure (Mockus, 2013; Borji and Itti, 2013), which exploits this model to search for the maximum of the performance function (see chapter 2.4). The robot selects which behaviors to test by maximizing the acquisition function that balances exploration (selecting points whose performance is uncertain) and exploitation (selecting points whose performance is expected to be high). The selected behavior is tested on the physical robot and the actual performance is recorded. The algorithm updates the expected performance of the tested behavior and lowers the uncertainty about it. These updates are propagated to neighboring solutions in the behavioral space by updating the Gaussian process. These updated performance and confidence distributions affect which behavior is tested next. This select-test-update loop repeats until the robot finds a behavior whose measured performance is greater than some user-defined percentage (here, 90%) of the best performance predicted for any behavior in the behavior-performance map (see appendix D.2.2).

4.3.2 Method description

The Intelligent Trial and Error Algorithm consists of two major steps (Fig. 4.8): the behavior-performance map creation step and the adaptation step (while here we focus on damage recovery, Intelligent Trial and Error can search for any type of required adaptation, such as learning an initial gait for an undamaged robot, adapting to new environments, etc.). The behavior-performance map creation step is accomplished via the MAP-Elites algorithm, while the adaptation step is accomplished via a new algorithm called the map-based Bayesian optimization algorithm

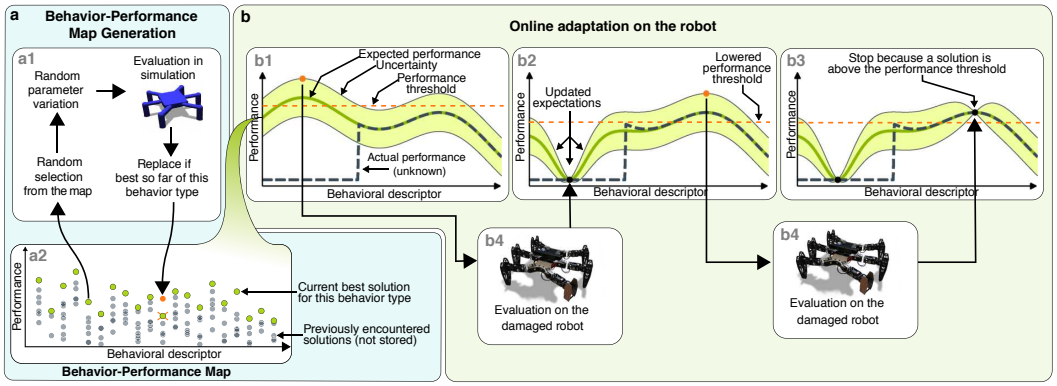


Figure 4.8: **An overview of the Intelligent Trial and Error Algorithm. (A) Behavior-performance map creation.** After being initialized with random controllers, the behavioral map (A2), which stores the highest-performing controller found so far of each behavior type, is improved by repeating the process depicted in (A1) until newly generated controllers are rarely good enough to be added to the map (here, after 40 million evaluations). This step, which occurs in simulation, is computationally expensive, but only needs to be performed once per robot (or robot design) prior to deployment. In our experiments, creating one map involved 40 million iterations of (A1), which lasted roughly two weeks on one multi-core computer (see appendix D.2.5). **(B) Adaptation.** (B1) Each behavior from the behavior-performance map has an expected performance based on its performance in simulation (dark green line) and an estimate of uncertainty regarding this predicted performance (light green band). The actual performance on the now-damaged robot (black dashed line) is unknown to the algorithm. A behavior is selected to try on the damaged robot. This selection is made by balancing exploitation—trying behaviors expected to perform well—and exploration—trying behaviors whose performance is uncertain (see section 2.4.3). Because all points initially have equal, maximal uncertainty, the first point chosen is that with the highest expected performance. Once this behavior is tested on the physical robot (B4), the performance predicted for that behavior is set to its actual performance, the uncertainty regarding that prediction is lowered, and the predictions for, and uncertainties about, nearby controllers are also updated (according to a Gaussian process model, see section 2.4.2.1, section "kernel function"), the results of which can be seen in (B2). The process is then repeated until performance on the damaged robot is 90% or greater of the maximum expected performance for any behavior (B3). This performance threshold (orange dashed line) lowers as the maximum expected performance (the highest point on the dark green line) is lowered, which occurs when physical tests on the robot underperform expectations, as occurred in (B2).

(M-BOA).

The key concept of M-BOA is to use the output of MAP-Elites as a prior for the Bayesian Optimization algorithm: thanks to the simulations, we expect some behaviors to perform better than others on the robot. In our implementation of BO, we use Gaussian Process (GP) regression (Rasmussen and Williams, 2006) to model the unknown function, which is a common choice for Bayesian optimization (Calandra et al., 2014; Griffiths et al., 2009; Brochu et al., 2010b; Lizotte et al., 2007). Here is a reminder of the traditional formulation of a GP (see chapter 2.4):

$$
\begin{aligned}
& P(f(\mathbf{x})|\mathbf{P}_{1:t}, \mathbf{x}) = \mathcal{N}(\mu_t(\mathbf{x}), \sigma_t^2(\mathbf{x})) \\
& \text{where :} \\
& \mu_t(\mathbf{x}) = \mu_0 + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t} - \mu_0) \\
& \sigma_t^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1} \mathbf{k} \\
& \mathbf{K} = \begin{bmatrix} k(\chi_1, \chi_1) & \cdots & k(\chi_1, \chi_t) \\ \vdots & \ddots & \vdots \\ k(\chi_t, \chi_1) & \cdots & k(\chi_t, \chi_t) \end{bmatrix} + \sigma_{noise}^2 I \\
& \mathbf{k} = \begin{bmatrix} k(\mathbf{x}, \chi_1) & k(\mathbf{x}, \chi_2) & \cdots & k(\mathbf{x}, \chi_t) \end{bmatrix} \\
& \mathbf{P}_{1:t} = \begin{bmatrix} P_1 \\ \vdots \\ P_t \end{bmatrix}
\end{aligned}
\tag{4.6}
$$

To incorporate the information provided by the behavior-performance map into the Bayesian optimization, M-BOA models the *difference* between the prediction of the map and the actual performance on the real robot, instead of directly modeling the objective function. This idea is incorporated into the Gaussian process by modifying the update equation for the mean function ($\mu_t(\mathbf{x})$, equation 4.6):

$$
\mu_t(\mathbf{x}) = \mathscr{P}(\mathbf{x}) + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t} - \mathscr{P}(\chi_{1:t})) \tag{4.7}
$$

where $\mathscr{P}(\mathbf{x})$ is the performance of $\mathbf{x}$ according to the simulation and $\mathscr{P}(\chi_{1:t})$ is the performance of all the previous observations, also according to the simulation. Replacing $\mathbf{P}_{1:t}$ (eq. 4.6) by $\mathbf{P}_{1:t} - \mathscr{P}(\chi_{1:t})$ (eq. 4.7) means that the Gaussian process models the difference between the actual performance $\mathbf{P}_{1:t}$ and the performance from the behavior-performance map $\mathscr{P}(\chi_{1:t})$. The term $\mathscr{P}(\mathbf{x})$ is the prediction of the behavior-performance map. M-BOA therefore starts with the prediction from the behavior-performance map and corrects it with the Gaussian process.

In our implementation, we decided to use the Matérn kernel function for the GP (with $\nu = 5/2$, see section 2.4.2.1) because it allows us to control not only the distance at which effects become nearly zero (as a function of parameter ρ, see appendix figure D.3a), but also the rate at which distance effects decrease (as a function of parameter ν). Because the model update step directly depends on ρ, it is one of the most critical parameters of the Intelligent Trial and Error Algorithm. We selected its value after extensive experiments in simulation detailed in appendix D.2.4.

For the acquisition function of the BO algorithm, we chose the Upper Confidence Bound because it provided the best results in several previous studies (Brochu et al., 2010b; Calandra et al., 2014). To perform the "inner-optimization" of this function, instead of using a gradient descent or another optimization algorithm, we exhaustively compute the acquisition value of each solution of the behavior-performance map and then choose the maximum value. Indeed, for the specific behavior space in the example problem in this chapter, the discretized search space of the behavior-performance map is small enough that an exhaustive exploration is computationally tractable. We describe how we chose the value of the parameter that controls the exploitation/exploration trade-off (κ) value in appendix D.2.4.

The pseudo-code of the entire algorithm is available in the algorithm 4, while more details about BO can be found in chapter 2.4.

4.3.3 Experimental validation

We test the Intelligent Trial and Error algorithm on five different experimental setups in order to evaluate, not only its overall performance over different situations, but also to determine which of its components contribute to the observed performances.

In our two first experiments, we evaluate the performance of our algorithm on two physical systems: (1) our hexapod robot (see appendix A.1) and (2) on a robotic arm with 8 degrees of freedom (see appendix B.1). The first objective of these experiments is to show that the algorithm allows these robots to cope with a large variety of damage situations. Among these situations, the hexapod robot suffers from a damaged, broken, or missing legs, and the robotic arm has joints broken in 14 different ways. The second objective of these experiments is to illustrate that our algorithm can be applied to many different types of robot, provided that they can execute the same task in different ways.

The three other experiments are performed in simulation and aim to gather extensive data to analyse the different properties of the Intelligent Trial and Error algorithm. We first investigate how the performance of the algorithm is altered when we deactivate one of its subcomponents or replaced it with an alternative algorithm from the literature. The goal of this experiment is to highlight the components of the algorithm that contribute the most to its performance. In particular, this experiment shows how our algorithm competes against state-of-the-art algorithms. In a second experiment, we consider another major challenge in robotics: adapting to new environments. In particular, we demonstrate how the Intelligent Trial and Error algorithm can help our robot to learn to walk on a slopped terrain. In our last experiment, we show that the choice of the behavioral descriptor, which may seem to be a critical parameter of the algorithm, does not affect the performance of the algorithm. Concretely, we test our algorithm with 12 different behavioral characterizations, including randomly choosing the 6 dimensions of the description among 63 possibilities and show that the performance of the algorithm remains unchanged.

Algorithm 4 The Intelligent Trial and Error Algorithm. Notations are described in the appendix D.2.1

procedure INTELLIGENT TRIAL AND ERROR ALGORITHM

Before the mission:

CREATE BEHAVIOR-PERFORMANCE MAP ▷ (via the MAP-Elites algorithm in simulation)

while In mission **do**

if Significant performance fall **then**

ADAPTATION STEP (VIA M-BOA ALGORITHM)

procedure MAP-ELITES ALGORITHM

$(\mathscr{P} \leftarrow \emptyset, \mathscr{C} \leftarrow \emptyset)$ ▷ *Creation of an empty behavior-performance map (empty N-dimensional grid).*

for iter $= 1 \rightarrow I$ **do** ▷ *Repeat during I iterations (here we choose I = 40 million iterations).*

if iter < 400 **then**

$\mathbf{c}' \leftarrow$ random_controller() ▷ *The first 400 controllers are generated randomly.*

else ▷ The next controllers are generated using the map.

$\mathbf{c} \leftarrow$ random_selection($\mathscr{C}$) ▷ *Randomly select a controller c in the map.*

$\mathbf{c}' \leftarrow$ random_variation($\mathbf{c}$) ▷ *Create a randomly modified copy of c.*

$\mathbf{x}' \leftarrow$behavioral_descriptor(simu($\mathbf{c}'$)) ▷ *Simulate the controller and record its behavioral descriptor.*

$p' \leftarrow$performance(simu($\mathbf{c}'$)) ▷ *Record its performance.*

if $\mathscr{P}(\mathbf{x}') = \emptyset$ or $\mathscr{P}(\mathbf{x}') < p'$ **then** ▷ *If the cell is empty or if p' is better than the current stored performance.*

$\mathscr{P}(\mathbf{x}') \leftarrow p'$ ▷ *Store the performance of $\mathbf{c}'$ in the behavior-performance map according to its behavioral descriptor $\mathbf{x}'$.*

$\mathscr{C}(\mathbf{x}') \leftarrow \mathbf{c}'$ ▷ *Associate the controller with its behavioral descriptor.*

return behavior-performance map ($\mathscr{P}$ and $\mathscr{C}$)

procedure M-BOA (MAP-BASED BAYESIAN OPTIMIZATION ALGORITHM)

$\forall \mathbf{x} \in$ map: ▷ *Initialisation.*

$P(f(\mathbf{x})|\mathbf{x}) = \mathcal{N}(\mu_0(\mathbf{x}), \sigma_0^2(\mathbf{x}))$ ▷ *Definition of the Gaussian Process.*

where

$\mu_0(\mathbf{x}) = \mathscr{P}(\mathbf{x})$ ▷ *Initialize the mean prior from the map.*

$\sigma_0^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) + \sigma_{noise}^2$ ▷ *Initialize the variance prior (in the common case, $k(\mathbf{x}, \mathbf{x}) = 1$).*

while $\max(\mathbf{P_{1:t}}) < \alpha \max(\mu_t(\mathbf{x}))$ **do** ▷ *Iteration loop.*

$\chi_{t+1} \leftarrow \arg\max_{\mathbf{x}}(\mu_t(\mathbf{x}) + \kappa\sigma_t(\mathbf{x}))$ ▷ *Select next test (argmax of acquisition function).*

$P_{t+1} \leftarrow$ performance(physical_robot($\mathscr{C}(\chi_{t+1})$)). ▷ *Evaluation of $\mathbf{x_{t+1}}$ on the physical robot.*

$P(f(\mathbf{x})|\mathbf{P}_{1:t+1}, \mathbf{x}) = \mathcal{N}(\mu_{t+1}(\mathbf{x}), \sigma_{t+1}^2(\mathbf{x}))$ ▷ *Update the Gaussian Process.*

where

$\mu_{t+1}(\mathbf{x}) = \mathscr{P}(\mathbf{x}) + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t+1} - \mathscr{P}(\chi_{1:t+1}))$ ▷ *Update the mean.*

$\sigma_{t+1}^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) + \sigma_{noise}^2 - \mathbf{k}^\intercal \mathbf{K}^{-1} \mathbf{k}$ ▷ *Update the variance.*

$\mathbf{K} = \begin{bmatrix} k(\chi_1, \chi_1) & \cdots & k(\chi_1, \chi_{t+1}) \\ \vdots & \ddots & \vdots \\ k(\chi_{t+1}, \chi_1) & \cdots & k(\chi_{t+1}, \chi_{t+1}) \end{bmatrix} + \sigma_{noise}^2 I$ ▷ *Compute the observations' correlation matrix.*

$\mathbf{k} = \begin{bmatrix} k(\mathbf{x}, \chi_1) & k(\mathbf{x}, \chi_2) & \cdots & k(\mathbf{x}, \chi_{t+1}) \end{bmatrix}$ ▷ *Compute the $\mathbf{x}$ vs. observation correlation vector.*

4.3.3.1 Experiments on the physical hexapod robot

Methods In this first experiment, we evaluate the Intelligent Trial and Error algorithm on our hexapod robot that needs to walk as fast as possible (see appendix A.1). The robot's performance is recorded thanks to a visual odometry algorithm, which records the position of the robot in real time and average its velocity after 5 seconds of execution (see appendix D.2.2). The behavior-performance maps used in this experiment are the ones generated in the previous chapter (see section 3.3.2.2), which contain approximately 13,000 different gaits that are governed by a controller with 36 real-values (see appendix A.2.2). The behavior-performance maps have 6 dimensions, where each dimension is the proportion of time the i^{th} leg spends in contact with the ground (i.e. the duty factor) (Siciliano and Khatib, 2008) (see section 3.3.2.2).

We test the adaptation abilities of our robot in six different conditions: undamaged (Fig. 4.9a:C1), four different structural failures (Fig. 4.9a:C2-C5), and a temporary leg repair (Fig. 4.9a:C6). Like in the previous experiments with the T-Resilience algorithm, we compare the walking speed of resultant gaits with a widely-used, classic, hand-designed tripod gait (Siciliano and Khatib, 2008) (see appendix A.2.2). For each of the 6 damage conditions, we ran our adaptation step 5 times for each of 8 independently generated behavior-performance maps (with the default "duty factor" behavioral description), leading to $6 \times 5 \times 8 = 240$ experiments in total. We also ran our adaptation step 5 times on 8 independently generated behavior-performance maps defined by an alternate behavioral description which consider the orientation of the body (see appendix D.2.2) on two damage conditions (Fig. 4.9b-c), leading to $2 \times 5 \times 8 = 80$ additional experiments. The implementation details of the behavioral descriptors, the performance function and the stopping criterion can be found in appendix D.2.2. A video of the typical results obtained with the Intelligent Trial and Error algorithm is available here: `https://youtu.be/T-c17RKh3uE`

Results When the robot is undamaged (Fig. 4.9a:C1), our approach yields dynamic gaits that are 30% faster than the classic reference gait (Fig. 4.9b, median 0.32 m/s, 5^{th} and 95^{th} percentiles [0.26; 0.36] vs. $0.24 m/s$), suggesting that Intelligent Trial and Error is a good search algorithm for automatically producing successful robot behaviors, putting aside damage recovery. In all the damage scenarios, the reference gait is no longer effective ($\sim$0.04 m/s for the four damage conditions, Fig. 4.9b:C2-C5). After Intelligent Trial and Error, the compensatory gaits achieve a reasonably fast speed ($> 0.15 m/s$) and are between 3 and 7 times more efficient than the reference gait for that damage condition (in m/s, C2: 0.24 [0.18; 0.31] vs. 0.04; C3: 0.22 [0.18; 0.26] vs. 0.03; C4: 0.21 [0.17; 0.26] vs. 0.04; C5: 0.17 [0.12; 0.24] vs. 0.05; C6: 0.3 [0.21; 0.33] vs. 0.12).

These experiments demonstrate that Intelligent Trial and Error allows the robot to both initially learn fast gaits and to reliably recover after physical damage. On the undamaged or repaired robot (Fig. 4.9: C6), Intelligent Trial and Error learns

a walking gait in less than 30 seconds (Fig. 4.9c, undamaged: 24 [16; 41] seconds, 3 [2; 5] physical trials, repaired: 29 [16; 82] seconds, 3.5 [2; 10] trials). For the five damage scenarios, the robot adapts in approximately one minute (66 [24; 134] seconds, 8 [3; 16] trials). It is possible that for certain types of damage the prior information from the undamaged robot does not help, and could even hurt, in learning a compensatory behavior (e.g. if the map does not contain a compensatory behavior). We did not find such a case in our experiments with the hexapod robot, but we did find a case with our robotic arm in which the prior information provided little benefit (see next experiment).

We investigated how the behavior-performance map is updated when the robot loses a leg (Fig. 4.10 and 4.11)). Initially the map predicts large areas of high performance. During adaptation, these areas disappear because the behaviors do not work well on the damaged robot. Intelligent Trial and Error quickly identifies one of the few, remaining, high-performing behaviors (Fig. 4.10 and 4.11).

Figure 4.9: **(A) Conditions tested on the physical hexapod robot.** C1: The undamaged robot. C2: One leg is shortened by half. C3: One leg is unpowered. C4: One leg is missing. C5: Two legs are missing. C6: A temporary, makeshift repair to the tip of one leg. **(B) Performance after adaptation.** Box plots represent Intelligent Trial and Error. The central mark is the median, the edges of the box are the 25^{th} and 75^{th} percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually. Yellow stars represent the performance of the handmade reference tripod gait (see appendix A.2.2). Conditions C1-C6 are tested 5 times each for 8 independently created behavior-performance maps with the "duty factor" behavior description (i.e. 40 experiments per damage condition). Damage conditions C1 and C3 are also tested 5 times each for 8 independently created behavior-performance maps with the "body orientation" behavior description. **(C) Time and number of trials required to adapt.** Box plots represent Intelligent Trial and Error. **(D) Robotic arm experiment.** The 8-joint, planar robot has to drop a ball into a bin. **(E) Example conditions tested on the physical robotic arm.** C1: One joint is stuck at 45 degrees. C2: One joint has a permanent 45-degree offset. C3: One broken and one offset joint. A total of 14 conditions were tested (Fig. 4.12). **(F) Time and number of trials required to reach within 5 cm of the bin center.** Each condition is tested with 15 independently created behavior-performance maps.

⇒

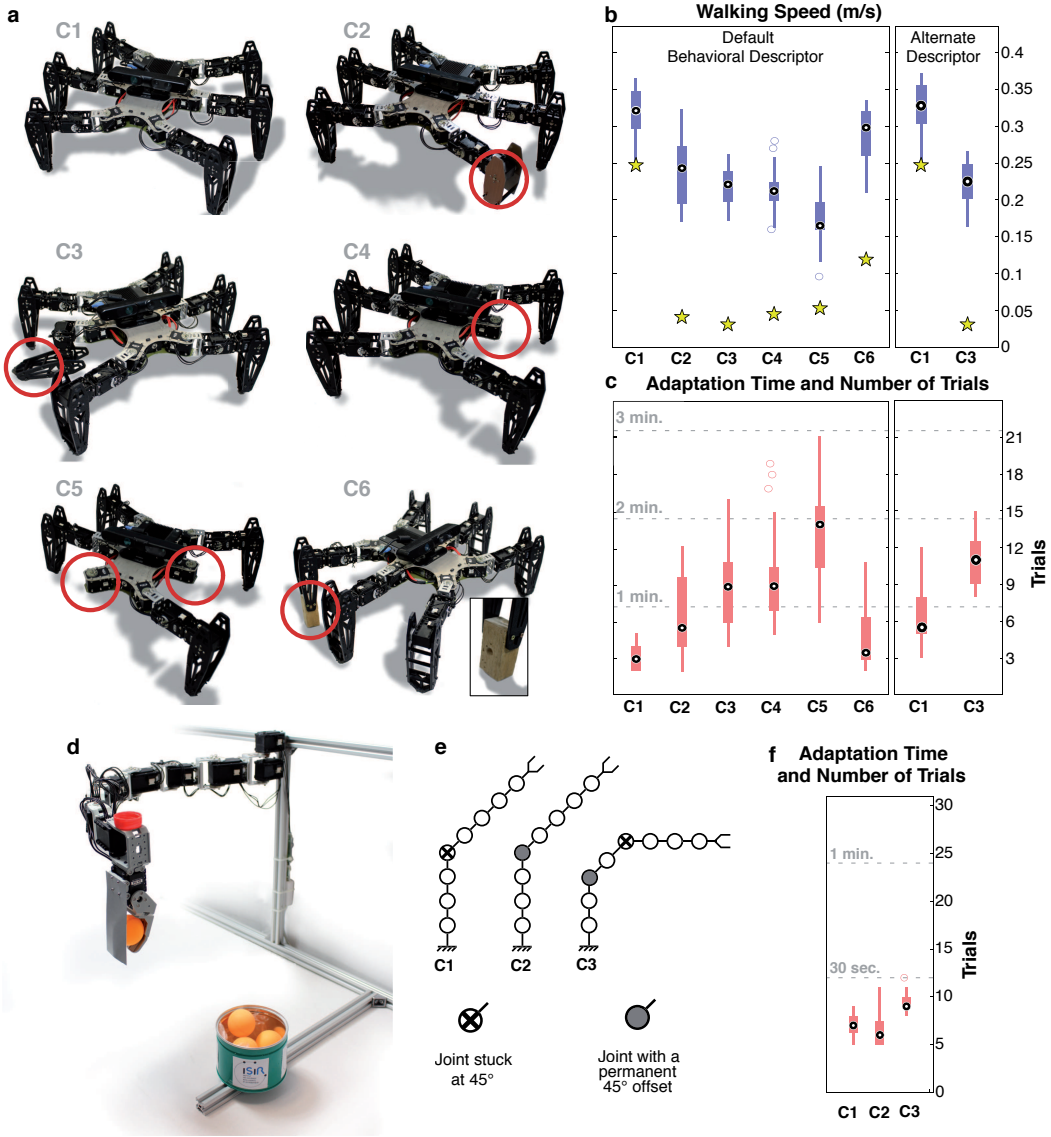

a
C1
C2
C3
C4
C5
C6
b
Walking Speed (m/s)
Default
Behavioral Descriptor
Alternate
Descriptor
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
C1 C2 C3 C4 C5 C6 C1 C3
c
Adaptation Time and Number of Trials
3 min.
2 min.
1 min.
21
18
15
12
9
6
3
Trials
C1 C2 C3 C4 C5 C6 C1 C3
d
e
C1
C2
C3
Joint stuck
at 45°
Joint with a
permanent
45° offset
f
Adaptation Time
and Number of Trials
1 min.
30 sec.
30
25
20
15
10
5
0
Trials
C1 C2 C3

Figure 4.9

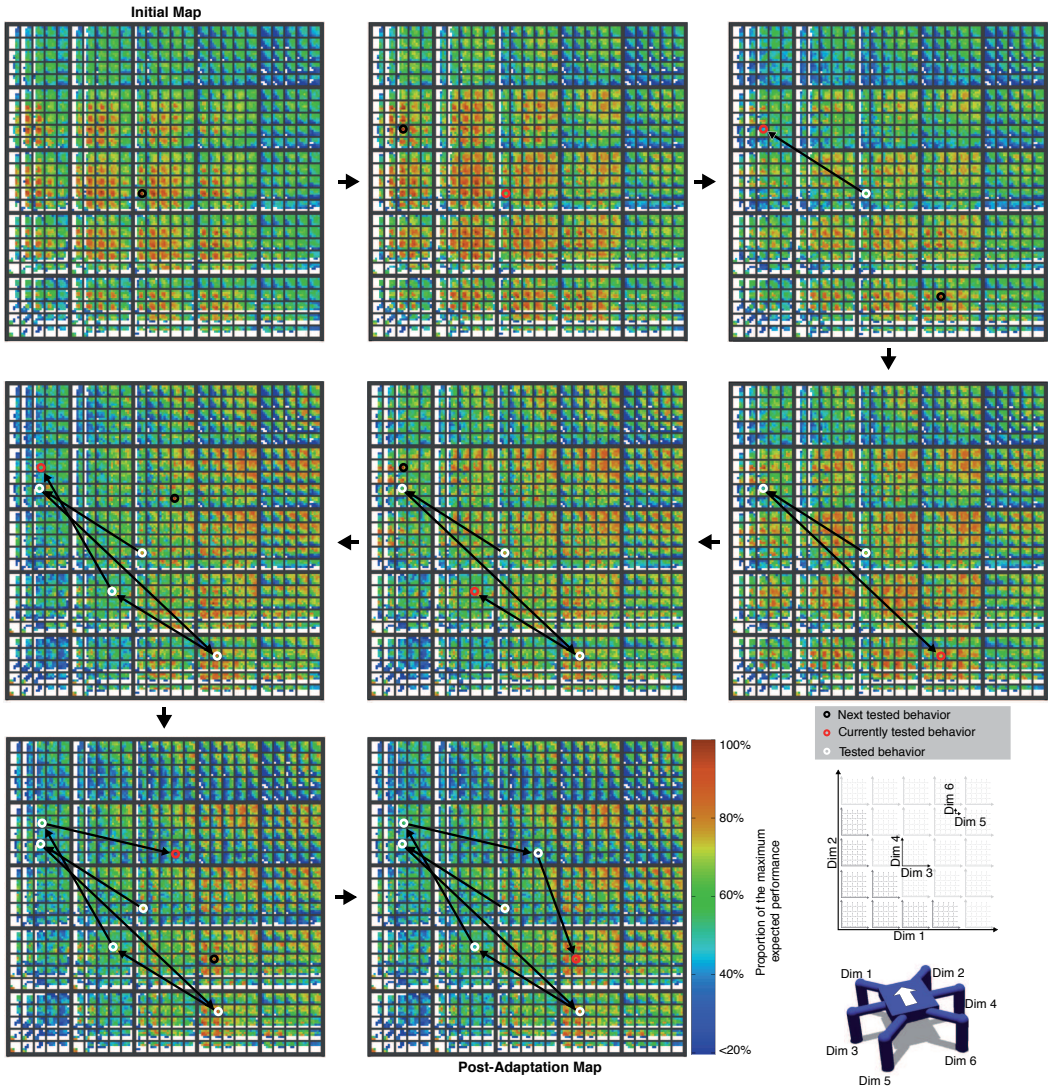


Figure 4.10: **How the behavior performance map is explored to discover a compensatory behavior (normalized each iteration to highlight the *range* of remaining performance predictions).** Colors represent the performance prediction for each point in the map relative to the highest performing prediction in the map at that step of the process. A black circle indicates the next behavior to be tested on the physical robot. A red circle indicates the behavior that was just tested (note that the performance predictions surrounding it have changed versus the previous panel). Arrows reveal the order that points have been explored. The red circle in the last map is the final, selected, compensatory behavior. In this scenario, the robot loses leg number 3. The six dimensional space is visualized according to the inset legend.

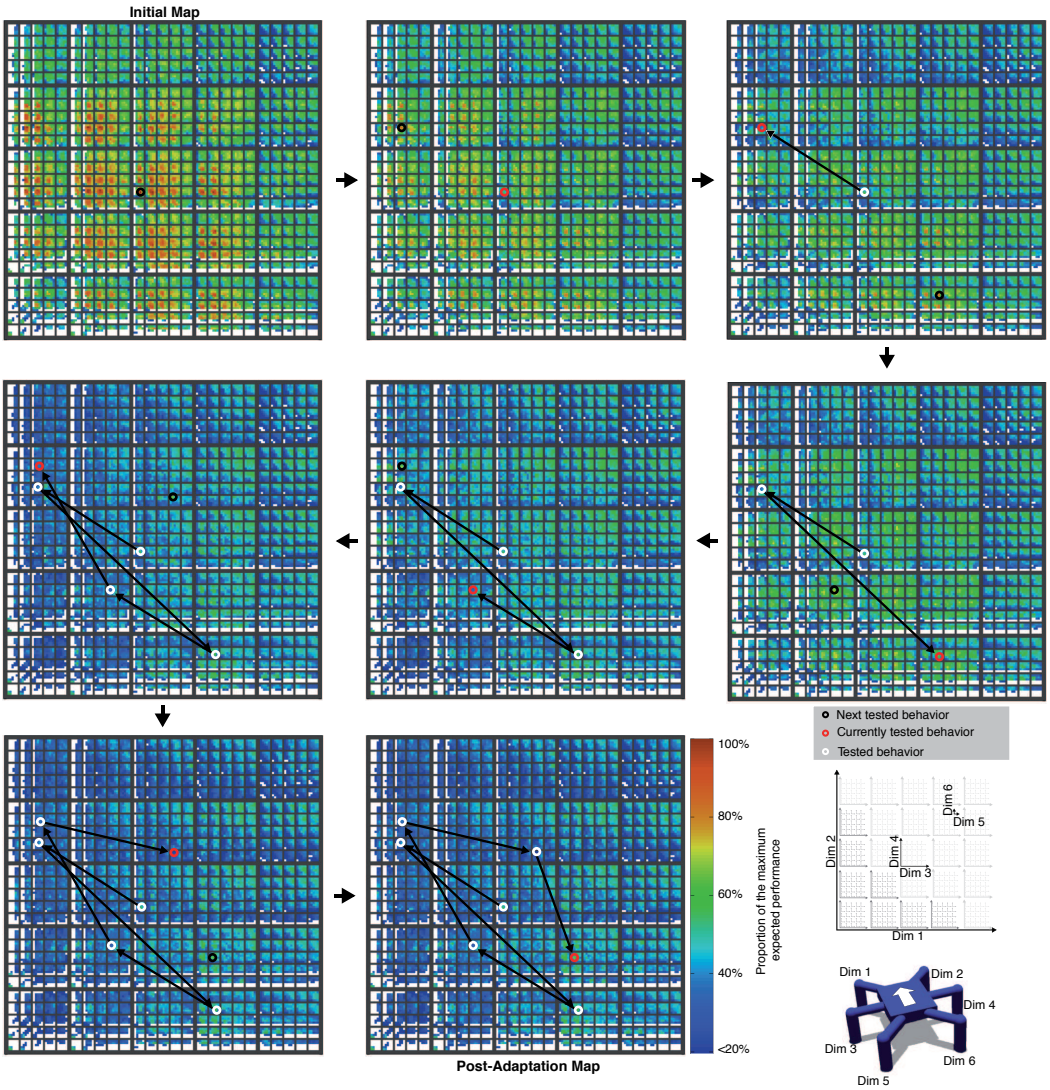


Figure 4.11: **How the behavior performance map is explored to discover a compensatory behavior (non-normalized to highlight that performance predictions decrease as it is discovered that predictions from the simulated, undamaged robot do not work well on the damaged robot).** Colors represent the performance prediction for each point in the map relative to the highest performing prediction in the *first* map. A black circle indicates the next behavior to be tested on the physical robot. A red circle indicates the behavior that was just tested (note that the performance predictions surrounding it have changed versus the previous panel). Arrows reveal the order that points have been explored. The red circle in the last map in the sequence is the final, selected, compensatory behavior. In this scenario, the robot loses leg number 3. The six dimensional space is visualized according to the inset legend. The data visualized in this figure are identical to those in the previous figure: the difference is simply whether the data are renormalized for each new map in the sequence.

4.3.3.2 Experiments on the physical robotic arm

Methods The same damage recovery approach can be applied to any robot, such as a robotic arm. We tested 14 different damage conditions with a planar, 8-joint robotic arm, which has to release a ball into a target bin (Fig. 4.9d-f and Fig. 4.12). The behavior-performance map's behavioral dimensions are the x, y position of the end-effector. To show that the map-generating performance measure can be different from the ultimate performance measure, and to encourage smooth arm movements, the performance measure during map-generation is minimizing the variance of the 8 specified motor angles (see appendix B.1). During adaptation, performance is measured as distance to the target.
For each of the 14 damage scenarios, we replicated experiments on the physical robot with 15 independently generated behavior-performance maps (210 runs in total). We also replicated control experiments, which consist of traditional Bayesian optimization directly in the original parameter space (i.e. without behavior-performance maps), 15 times for each of the 14 damage conditions (210 runs in total). For both the experimental and control treatments, each experiment involved 30 evaluations on the physical robot (31 if the initial trial is counted). In many cases, not all 30 evaluations were required to reach the target, so we report only the number of trials required to reach that goal. The implementation details of the behavioral descriptors, the performance function and the stopping criterion can be found in appendix D.2.3.

Results After running the MAP-Elites algorithm for 20 million evaluations, each of the 15 generated maps contain more than $11,000$ behaviors (11,209 [1,1206; 1,1217] behaviors, Fig. 4.12c).
In all the generated maps, the regions of different performance values for behaviors are arranged in concentric shapes resembling cardioids (inverted, heart-shaped curves) that cover the places the robot can reach (Fig. 4.12c). The black line drawn over the shown map corresponds to all the positions of the end-effector for which all the degrees of freedom are set to the same angle (from $-\pi/4$ to $+\pi/4$), that is, for the theoretically highest achievable performance (i.e. the lowest possible variance in servo angles). The performance of the behaviors tends to decrease the further they are from this optimal line. This result illustrates how MAP-Elites is able to find both a large variety of solutions but also the optimal ones.
Like with the hexapod robot, our approach discovers a compensatory behavior in less than 2 minutes, usually in less than 30 seconds, and with fewer than 10 trials (Fig. 4.9f and Fig. 4.12). The adaptation results (Fig. 4.12e) show that the Intelligent Trial and Error algorithm manages to reach the goal of being less than 5 cm from the center of the bin for all the runs in all the tested scenarios except two (scenarios 11 & 12). For these two scenarios, the algorithm still reaches the target 60% and 80% of the time, respectively. For all the damage conditions, the Intelligent Trial and Error algorithm reaches the target significantly more often than the Bayesian optimization algorithm ($p < 10^{-24}$). Specifically, the median number

of iterations to reach the target (Fig. 4.12f) is below 11 iterations (27.5 seconds) for all scenarios except 11 and 12, for which 31 and 20 iterations are required, respectively. When the robot is not able to reach the target, the recorded number of iterations is set to 31, which explains why the median number of iterations for the Bayesian optimization algorithm is equal to 31 for most damage conditions. For all the damage conditions except one (scenario 11), the Intelligent Trial and Error algorithm used fewer trials to reach the target than the traditional Bayesian optimization algorithm.

If the robot is allowed to continue its experiment after reaching the 5 cm radius tolerance, for a total of 31 iterations (Fig. 4.12g), it reaches an accuracy around 1 cm for all the damage conditions except the two difficult ones (scenarios 11 and 12). This level of accuracy is never achieved with the classic Bayesian optimization algorithm, whose lowest median accuracy is $2.6cm$.

Scenarios 11 and 12 appear to challenge the Intelligent Trial and Error algorithm. While in both cases the success rate is improved, though not substantially, in case 11 the median accuracy is actually lower. These results stem from the fact that the difference between the successful pre-damage and post-damage behaviors is so large that the post-damage solutions for both scenarios lie outside of the map. This illustrates a limit of the proposed approach: if the map does not contain a behavior able to cope with the damage, the robot will not be able to adapt. This limit mainly comes from the behavioral descriptor choice: we chose it because of its simplicity, but it does not capture all of the important dimensions of variation of the robot. More sophisticated descriptors are likely to allow the algorithm to cope with such situations. On the other hand, this experiment shows that with a very simple behavioral descriptor, using only the final position of the end-effector, our approach is able to deal with a large variety of different target positions and is significantly faster than the traditional Bayesian optimization approach (Fig. 4.12d, maximum p-value over each time step $< 10^{-16}$), which is the current state of the art technique for direct policy search in robotics (Lizotte et al., 2007; Tesch et al., 2011; Calandra et al., 2014; Kober et al., 2013).

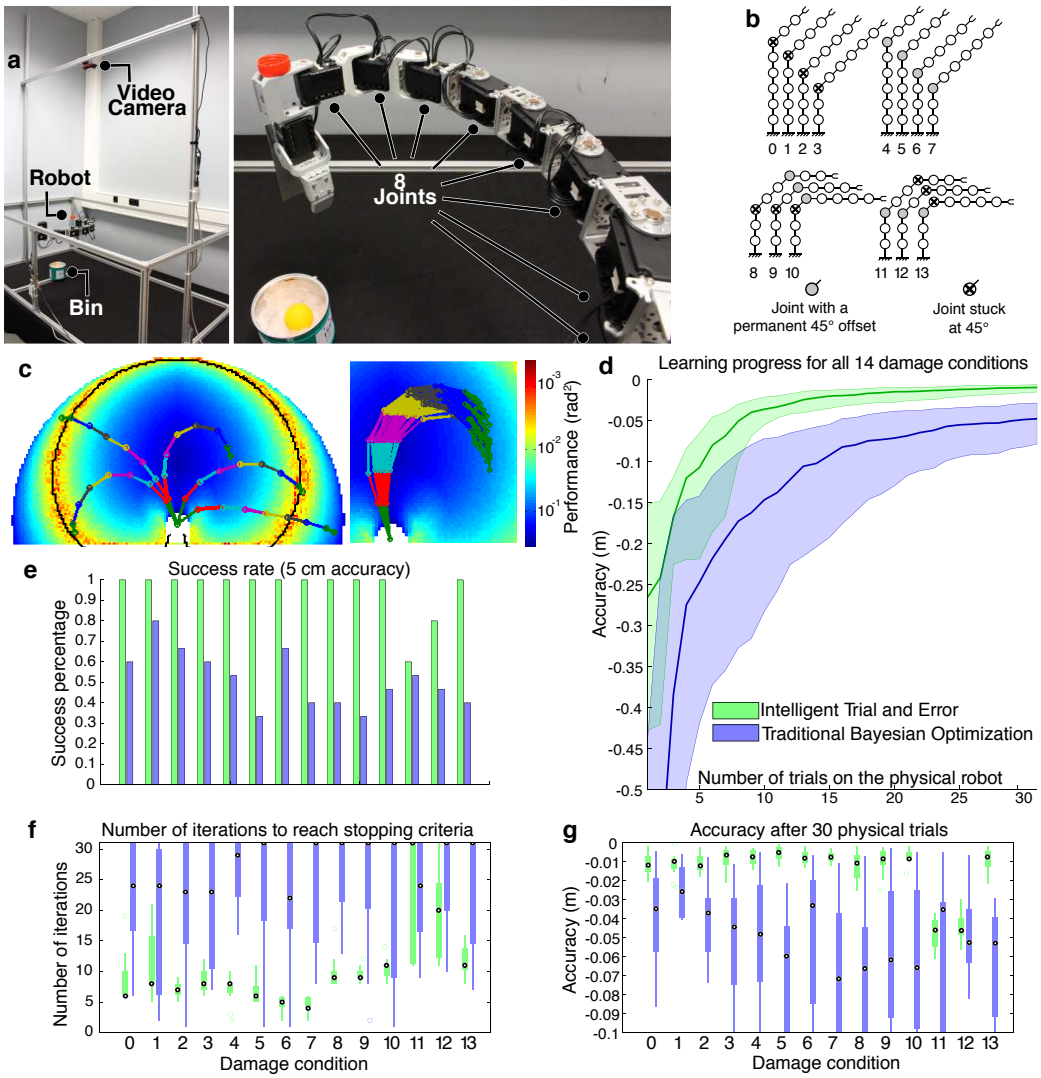

a
Video Camera
Robot
Bin
8 Joints
b
0 1 2 3
4 5 6 7
8 9 10
11 12 13
Joint with a permanent 45° offset
Joint stuck at 45°
c
Performance (rad²)
d
Learning progress for all 14 damage conditions
Accuracy (m)
Intelligent Trial and Error
Traditional Bayesian Optimization
Number of trials on the physical robot
e
Success rate (5 cm accuracy)
Success percentage
f
Number of iterations to reach stopping criteria
Number of iterations
Damage condition
g
Accuracy after 30 physical trials
Accuracy (m)
Damage condition

Figure 4.12

Figure 4.12: **Intelligent Trial and Error works on a completely different type of robot: detailed data from the robotic arm experiment. (A) The robotic arm experimental setup. (B) Tested damage conditions. (C) Example of behavior performance maps (colormaps) and behaviors (overlaid arm configurations) obtained with MAP-Elites.** Left: A typical behavior-performance map produced by MAP-Elites with 5 example behaviors, where a behavior is described by the angle of each of the 8 joints. The color of each point is a function of its performance, which is defined as having low variance in the joint angles (i.e. a zigzag arm is lower performing than a straighter arm that reaches the same point). Right: Neighboring points in the map tend to have similar behaviors, thanks to the performance function, which would penalize more jagged ways of reaching those points. That neighbors have similar behaviors justifies updating predictions about the performance of nearby behaviors after a testing a single behavior on the real (damaged) robot. **(D) Performance vs. trial number for Intelligent Trial and Error and traditional Bayesian optimization.** The experiment was conducted on the physical robot, with 15 independent replications for each of the 14 damage conditions. Performance is pooled from all of these $14 \times 15 = 210$ experiments. **(E) Success for each damage condition.** Shown is the success rate for the 15 replications for each damage condition, defined as the percentage of replicates in which the robot reaches within 5 cm of the bin center. **(F) Trials required to adapt.** Shown is the number of iterations required to reach within 5 cm of the basket center. **(G) Accuracy after 30 physical trials.** Performance after 30 physical trials for each damage condition (with the stopping criterion disabled).

⇐

4.3.3.3 The contribution of each subcomponent of the Intelligent Trial and Error Algorithm

Methods The Intelligent Trial and Error Algorithm relies on three main concepts: (1) the creation of a behavior-performance map in simulation via the MAP-Elites algorithm, (2) searching this map with a Bayesian optimization algorithm to find behaviors that perform well on the physical robot, and (3) initializing this Bayesian optimization search with the performance predictions obtained via the MAP-Elites algorithm: note that the second step could be performed without the third step by searching through the MAP-Elites-generated behavior-performance map with Bayesian optimization, but having the initial priors uniformly set to the same value. We investigated the contribution of each of these subcomponents by testing five variants of our algorithm : in each of them, we deactivated one of these three subcomponents or replaced it with an alternative algorithm from the literature. We then tested these variants on the hexapod robot. The variants are as follows:

- Variant 1 (MAP-Elites in 6 dimensions + random search): evaluates the benefit of searching the map via Bayesian optimization by searching that map with random search instead. Each iteration, a behavior is randomly selected

from the map and tested on the robot. The best one is kept.

- Variant 2 (MAP-Elites in 6 dimensions + Bayesian optimization, no use of priors): evaluates the contribution of initializing the Gaussian process with the performance predictions of the behavior-performance map. In this variant, the Gaussian process is initialized with a constant mean (the average performance of the map: 0.24 m/s) at each location in the behavior space and a constant variance (the average variance of the map's performance: 0.005 m^2/s^2). As is customary, the first few trials (here, 5) of the Bayesian optimization process are selected randomly instead of letting the algorithm choose those points, which is known to improve performance (Calandra et al., 2014).

- Variant 3 (MAP-Elites in 6 dimensions + policy gradient): evaluates the benefit of Bayesian optimization compared to a more classic, local search algorithm (Kober et al., 2013; Kohl and Stone, 2004); there is no obvious way to use priors in policy gradient algorithms.

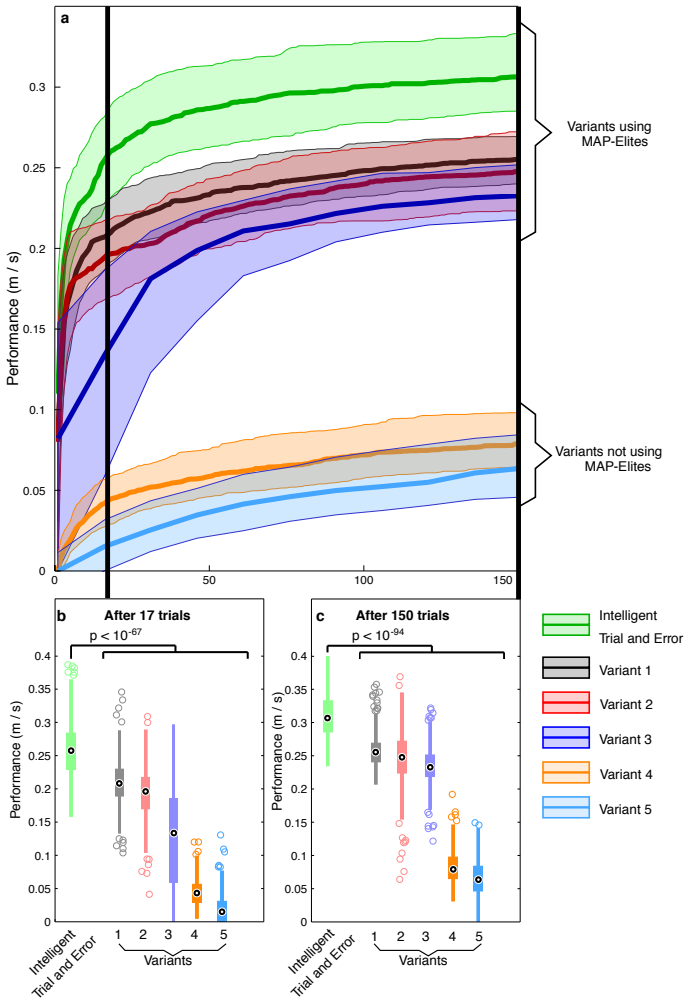
- Variant 4 (Bayesian optimization in the original parameter space of 36 dimensions): evaluates the contribution of using a map in a lower-dimensional behavioral space. This variant searches directly in the original 36-dimensional parameter space instead of reducing that space to the lower-dimensional (six-dimensional) behavior space. Thus, in this variant no map of behaviors is produced ahead of time: the algorithm searches directly in the original, high-dimensional space. This variant corresponds to one of the best algorithms known to learn locomotion patterns (Lizotte et al., 2007; Calandra et al., 2014). In this variant, the Gaussian process is initialized with a constant mean set to zero and with a constant variance ($0.002m^2/s^2$). As described above, the five first trials are selected from pure random search to prime the Bayesian optimization algorithm (Calandra et al., 2014).

- Variant 5 (Policy gradient in the original parameter space of 36 dimensions): a stochastic gradient descent in the original parameter space (Kohl and Stone, 2004). This approach is a classic reinforcement learning algorithm for locomotion (Kober et al., 2013) and it is a baseline in many papers (e.g. (Lizotte et al., 2007)).

It was necessary to compare these variants in simulation because doing so on the physical robot would have required months of experiments and would have repeatedly worn out or broken the robot. We modified the simulator from the previous experiments (section A.1: Simulator) to emulate 6 different possible damage conditions, each of which involved removing a different leg. For variants in which MAP-Elites creates a map (variants 1, 2 and 3), we used the same maps from the previous experiments (the eight independently generated maps, which were all generated with a simulation of the undamaged robot): In these cases, we launched ten replicates of each variant for each of the eight maps and each of the six damage conditions. There are therefore $10 \times 8 \times 6 = 480$ replicates for each of those variants.

For the other variants (4 and 5), we replicated each experiment 80 times for each of the six damage conditions, which also led to $80 \times 6 = 480$ replicates per variant. In all these simulated experiments, to roughly simulate the distribution of noisy odometry measurements on the real robot, the simulated performance values were randomly perturbed with a multiplicative Gaussian noise centered on 0.95 with a standard deviation of 0.1.
We analyze the fastest walking speed achieved with each variant after two different numbers of trials: the first case is after 17 trials, which was the maximum number of iterations used by the Intelligent Trial and Error Algorithm (Fig. 4.9, maximum 17 trials, median 6 trials), and the second case is after 150 trials, which is approximately the number of trials used in previous work (Kohl and Stone, 2004; Lizotte et al., 2007; Calandra et al., 2014).

Results After 17 trials on the robot, Intelligent Trial and Error significantly outperforms all the variants (Fig. 4.13b, $p < 10^{-67}$, median Intelligent Trial and Error performance: 0.26 [0.20; 0.33] m/s), demonstrating that the three main components of the algorithm are needed to quickly find high-performing behaviors. Among the investigated variants, the random search in the map performs the best (Variant 1: 0.21 [0.16; 0.27] m/s), followed by Bayesian optimization in the map without prior (Variant 2: 0.20 [0.13; 0.25] m/s), and policy gradient in the map (Variant 3: 0.13 [0; 0.23] m/s). Variants that search directly in the parameter space did not find any working behavior (Variant 4, Bayesian optimization: 0.04 [0.01; 0.09] m/s; Variant 5, policy gradient: 0.02 [0; 0.06] m/s).
There are two reasons that random search performs better than one might expect. First, the map only contains high-performing solutions, which are the result of the intense search of the MAP-Elites algorithm (40 million evaluations in simulation). The map thus already contains high-performing gaits of nearly every possible type. Therefore, this variant is not testing random controllers, but is randomly selecting high-performing solutions. Second, Bayesian optimization and policy gradient are not designed for such a low number of trials: without the priors on performance predictions introduced in the Intelligent Trial and Error Algorithm, the Bayesian optimization process needs to learn the overall shape of the search space to model it with a Gaussian process. 17 trials is too low a number to effectively sample six dimensions (for a uniform sampling with only two possible values in each dimension, $2^6 = 64$ trials are needed; for five possible values, $5^6 = 15,625$ samples are needed). As a consequence, with this low number of trials, the Gaussian process that models the performance function is not informed enough to effectively guide the search. For the policy gradient algorithm, a gradient is estimated by empirically measuring the partial derivative of the performance function in each dimension. To do so, following (Kohl and Stone, 2004), the policy gradient algorithm performs 15 trials at each iteration. Consequently, when only 17 trials are allowed, it iterates only once. In addition, policy gradient is a local optimization algorithm that highly depends on the starting point (which is here chosen randomly), as illustrated by



Variant	Behavior-performance map creation	Priors on performance	Search algorithm	equivalent approach
Intelligent Trial and Error	MAP-Elites	yes	Bayesian Optimization	-
Variant 1	MAP-Elites	none	Random Search	-
Variant 2	MAP-Elites	none	Bayesian Optimization	-
Variant 3	MAP-Elites	none	Policy Gradient	-
Variant 4	none	none	Bayesian Optimization	Lizotte et al. (2007)
Variant 5	none	none	Policy Gradient	Kohl et al. (2004)

Figure 4.13: **The contribution of each subcomponent of the Intelligent Trial and Error Algorithm. (A) Adaptation progress versus the number of robot trials.** The walking speed achieved with Intelligent Trial and Error and several "knockout" variants that are missing one of the algorithm's key components. Some variants (4 and 5) correspond to state-of-the-art learning algorithms (policy gradient: Kohl and Stone (2004); Bayesian optimization: Lizotte (2008); Tesch et al. (2011); Calandra et al. (2014)). The bold lines represent the medians and the colored areas extend to the 25^{th} and 75^{th} percentiles. **(B, C) Adaptation performance after 17 and 150 trials.** Shown is the the speed of the compensatory behavior discovered by each algorithm after 17 and 150 evaluations on the robot, respectively. For all panels, data are pooled across six damage conditions (the removal of each of the 6 legs in turn).

the high variability in the performance achieved with this variant (Fig. 4.13b).
The issues faced by Bayesian optimization and policy gradient are exacerbated when the algorithms search directly in the original, 36-dimensional parameter space instead of the lower-dimensional (six-dimensional) behavior space of the map. As mentioned previously, no working controller was found in the two variants directly searching in this high-dimensional space.
Overall, the analysis after 17 trials shows that:

- The most critical component of the Intelligent Trial and Error Algorithm is the MAP-Elites algorithm, which reduces the search space and produces a map of high-performing behaviors in that space: $p < 5 \times 10^{-50}$ when comparing variants searching in the behavior-performance map space vs. variants that search in the original, higher-dimensional space of motor parameters.

- Bayesian optimization critically improves the search, but only when it is initialized with the performance obtained in simulation during the behavior-performance map creation step (with initialization: 0.26 [0.20; 0.33] m/s, without initialization: 0.20 [0.13; 0.25] m/s, $p = 10^{-96}$).

To check whether these variants might perform better if allowed the number of evaluations typically given to previous state-of-the-art algorithms (Kohl and Stone, 2004; Lizotte et al., 2007; Calandra et al., 2014), we continued the experiments until 150 trials on the robot were conducted (Fig. 4.13c). Although the results for all the variants improved, Intelligent Trial and Error still outperforms all them ($p < 10^{-94}$; Intelligent Trial and Error: 0.31 [0.26; 0.37] m/s, random search: 0.26 [0.22; 0.30] m/s, Bayesian optimization: 0.25 [0.18; 0.31] m/s, policy search: 0.23 [0.19, 0.29] m/s). These results are consistent with the previously published results (Kohl and Stone, 2004; Lizotte et al., 2007; Calandra et al., 2014; Kober et al., 2013), which optimize in 4 to 10 dimensions in a few hundred trials. Nevertheless, when MAP-Elites is not used, i.e. when we run these algorithms in the original 36 dimensions for 150 evaluations, Bayesian optimization and policy gradient both perform much worse (Bayesian optimization: 0.08 [0.05; 0.12]; policy gradient: 0.06 [0.01; 0.12] m/s). These results shows that MAP-Elites is a powerful method to reduce the dimensionality of a search space for learning algorithms, in addition to providing helpful priors about the search space that speed up Bayesian optimization.
Overall, these experiments demonstrate that each of the three main components of the Intelligent Trial and Error Algorithm substantially improves performance. The results also indicate that Intelligent Trial and Error significantly outperforms previous algorithms for both damage recovery (Erden and Leblebicioğlu, 2008; Bongard et al., 2006; Christensen et al., 2013; Mahdavi and Bentley, 2006; Koos et al., 2013a) and gait learning (Hornby et al., 2005; Kohl and Stone, 2004; Barfoot et al., 2006; Sproewitz et al., 2008; Koos et al., 2013c; Lizotte et al., 2007; Tesch et al., 2011; Calandra et al., 2014; Kober et al., 2013; Yosinski et al., 2011).

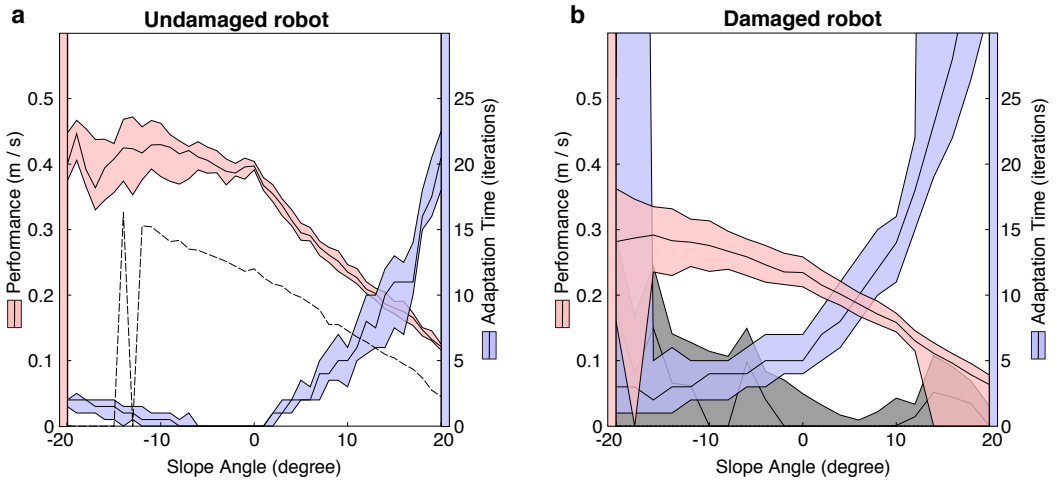


Figure 4.14: **The Intelligent Trial and Error Algorithm is robust to environmental changes.** Each plot shows both the performance and required adaptation time for Intelligent Trial and Error when the robot must adapt to walk on terrains of different slopes. **(A) Adaptation performance on an undamaged robot.** On all slope angles, with very few physical trials, the Intelligent Trial and Error Algorithm (pink shaded region) finds fast gaits that outperform the reference gait (black dotted line). **(B) Adaptation performance on a damaged robot.** The robot is damaged by having each of the six legs removed in six different damage scenarios. Data are pooled from all six of these damage conditions. The median compensatory behavior found via Intelligent Trial and Error outperforms the median reference controller on all slope angles. The middle, black lines represent medians, while the colored areas extend to the 25^{th} and 75^{th} percentiles. In (A), the black dashed line is the performance of a classic tripod gait for reference. In (B), the reference gait is tried in all six damage conditions and its median (black line) and 25^{th} and 75^{th} percentiles (black colored area) are shown.

4.3.3.4 Robustness to environmental changes

Methods The map creation algorithm (MAP-Elites) uses an undamaged robot on flat terrain. The previous experiments show that this algorithm provides useful priors for damage recovery on a flat terrain. In this experiment, we evaluated, in simulation, if the map created on flat terrain also provides a useful starting point for discovering gaits for sloped terrains.

We first evaluated the effect slopes have on undamaged robots (Fig. 4.14a). We launched 10 replicates for each of the eight maps and each one-degree increment between $-20°$ and $+20°$, for a total of $10 \times 8 \times 41 = 3280$ experiments. As in the previous experiment, to roughly simulate the distribution of noisy odometry measurements on the real robot, we perturbed performance values with a multiplicative Gaussian noise centered on 0.95 with a standard deviation of 0.1.

Results The results show that, when the slope is negative (descending), the Intelligent Trial and Error approach finds fast gaits in fewer than than 3 trials. For reference, a hand-designed, classic, tripod gait (see appendix A.2.2) falls on slopes below $-15°$ degrees. When the slope is positive (ascent), Intelligent Trial and Error finds slower behaviors, as is expected, but even above $10°$ the gait learned by Intelligent Trial and Error outperforms the reference gait on flat ground. Overall, for every slope angle, the controller found by Intelligent Trial and Error is faster than the hand-designed reference controller.

We further evaluated damage recovery performance for these same slopes with the same setup as Experiments S1 (6 damage conditions). We launched 10 replicates for each damage condition, for 8 independently generated behavior-performance maps, and each two-degree increment between $-20°$ and $+20°$ degrees. There are therefore 480 replicates for each degree between $-20°$ and $+20°$, for a total of $480 \times 21 = 10080$ experiments.

Intelligent Trial and Error is not critically affected by variations of slope between $-10°$ and $+10°$ (Fig. 4.14b): for these slopes, and for all 6 damage conditions, Intelligent Trial and Error finds fast gaits (above 0.2 m/s) in less than 15 tests on the robot despite the slope. As expected, it finds faster gaits for negative slopes (descent) and slower gaits for positive slopes (ascent). For slopes below $-10°$ and above $10°$, the algorithm performs worse and requires more trials. We can suppose that these results likely are caused by the constraints placed on the controller and the limited sensors on the robot, rather than the inabilities of the algorithm. Specifically, the controller was kept simple to make the experiments clearer, more intuitive, and more reproducible. Those constraints, of course, prevent it from performing the more complex behaviors necessary to deal with highly sloped terrain. For example, the constraints prevent the robot from keeping its legs vertical on sloped ground, which would substantially reduce slippage. Nevertheless, the median Intelligent Trial and Error compensatory gait still outperforms the median performance of the reference gait on all slope angles.

4.3.3.5 Alternate behavioral descriptors

Methods To create a map with MAP-Elites, one has to define the dimensions of the behavioral space, i.e. the behavioral descriptors. The previous experiments show that using a predefined behavioral descriptor (the proportion of time that each leg of a hexapod robot is in contact with the ground, i.e. the duty factor) creates a map that provides useful priors for damage recovery.

This section describes how we tested (in simulation) how performance is affected by alternate behavioral descriptors, including descriptors that have a different number of dimensions. We also evaluated how performance is affected if the behavioral descriptors are randomly selected from a large list of potential descriptors. This test simulates the algorithm's performance if the behavioral descriptors are chosen without insight into the problem domain.

The behavioral descriptors we tested are as follows:

1. Duty factor (6-dimensional): This descriptor is the default one from the previous experiment. It corresponds to the proportion of time each leg is in contact with the ground:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t C_1(t)}{\text{numTimesteps}} \\ \vdots \\ \frac{\sum_t C_6(t)}{\text{numTimesteps}} \end{bmatrix} \tag{4.8}$$

   where $C_i(t)$ denotes the Boolean value of whether leg i is in contact with the ground at time t (1: contact, 0: no contact).

2. Orientation (6-dimensional): This behavioral descriptor characterizes changes in the angular position of the robot during walking, measured as the proportion of 15ms intervals that each of the pitch, roll and yaw angles of the robot frame are positive (three dimensions) and negative (three additional dimensions):

$$\mathbf{x} = \begin{bmatrix} \frac{1}{K}\sum_k U(\Theta^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Theta^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(\Psi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Psi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(\Phi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Phi^T(k) - 0.005\pi) \end{bmatrix} \tag{4.9}$$

   where $\Theta^T(k)$, $\Psi^T(k)$ and $\Phi^T(k)$ denote the pitch, roll and yaw angles, respectively, of the robot torso (hence T) at the end of interval k, and K denotes the number of 15ms intervals during the 5 seconds of simulated movement (here, $K = 5s/0.015s \approx 334$). The unit step function $U(\cdot)$ returns 1 if its argument exceeds 0, and returns 0 otherwise. To discount for insignificant motion around 0 rad, orientation angles are only defined as positive if they exceed 0.5% of π rad. Similarly, orientation angles are only defined as negative if they are less than -0.5% of π rad.

3. Displacement (6-dimensional): This behavioral descriptor characterizes changes in the position of the robot during walking. It is measured as the proportion of 15ms intervals that the robot is positively or negatively displaced along each of the x, y, and z axes:

$$\mathbf{x} = \begin{bmatrix} \frac{1}{K}\sum_k U(\Delta x(k) - 0.001) \\ \frac{1}{K}\sum_k U(-\Delta x(k) - 0.001) \\ \frac{1}{K}\sum_k U(\Delta y(k) - 0.001) \\ \frac{1}{K}\sum_k U(-\Delta y(k) - 0.001) \\ \frac{1}{K}\sum_k U(\Delta z(k) - 0.001) \\ \frac{1}{K}\sum_k U(-\Delta z(k) - 0.001) \end{bmatrix} \tag{4.10}$$

where $[\Delta x(k), \Delta y(k), \Delta z(k)]$ denote the linear displacement in meters of the robot during interval k, and K denotes the number of 15ms intervals during 5 seconds of simulated movement (here, $K = 5s/0.015s \approx 334$). The unit step function $U(\cdot)$ returns a value of 1 if its argument exceeds 0, and returns a value of 0 otherwise. To ignore insignificant motion, linear displacements are defined to be positive if they exceed 1mm, and are defined to be negative if they are less than -1mm.

4. Total energy expended per leg (6-dimensional): This behavioral descriptor captures the total amount of energy expended to move each leg during 5 seconds of movement:

$$\mathbf{x} = \begin{bmatrix} \frac{E_1}{M_E} \\ \vdots \\ \frac{E_6}{M_E} \end{bmatrix} \tag{4.11}$$

where E_i denotes the energy utilized by leg i of the robot during 5 seconds of simulated movement, measured in N.m.rad. M_E is the maximum amount of energy available according to the servo model of the simulator, which for 5 seconds is 100 N.m.rad.

5. Relative energy expended per leg (6-dimensional): This behavioral descriptor captures the amount of energy expended to move each leg relative to the energy expended by all the legs during 5 seconds of simulated movement:

$$\mathbf{x} = \begin{bmatrix} \frac{E_1}{\sum_{i=1..6} E_i} \\ \vdots \\ \frac{E_6}{\sum_{i=1..6} E_i} \end{bmatrix} \tag{4.12}$$

where E_i denotes the energy utilized by leg i of the robot during 5 seconds of simulated movement, measured in N.m.rad.

6. Deviation (3-dimensional): This descriptor captures the range of deviation of the center of the robot frame versus the expected location of the robot if it traveled in a straight line at a constant speed.

$$\mathbf{x} = \begin{bmatrix} \frac{0.95\left(\max\limits_t(x(t)) - \min\limits_t(x(t))\right)}{0.2} \\ \frac{0.95\left(\max\limits_t(y(t) - \frac{y_{\text{final}}}{5} \times t) - \min\limits_t(y(t) - \frac{y_{\text{final}}}{5} \times t)\right)}{0.2} \\ \frac{0.95\left(\max\limits_t(z(t)) - \min\limits_t(z(t))\right)}{0.2} \end{bmatrix} \tag{4.13}$$

where $[x(t), y(t), z(t)]$ denote the position of robot's center at time t, and $[x_{\text{final}}, y_{\text{final}}, z_{\text{final}}]$ denote its final position after 5 seconds.

The robot's task is to move along the y-axis. Its starting position is (0,0,0). The deviation along the x and z axes is computed as the maximum difference in the robot's position in those dimensions at any point during 5 seconds. For the y axis, $\frac{y_{\text{final}}}{5}$ corresponds to the average speed of the robot (the distance covered divided by total time), therefore $\frac{y_{\text{final}}}{5} \times t$ is the expected position at timestep t if the robot was moving at constant speed. The deviation from the y axis is computed with respect to this "theoretical" position.

To obtain values in the range [0,1], the final behavioral descriptors are multiplied by 0.95 and then divided by 20 cm (these values were determined empirically).

7. Total ground reaction force per leg (6-dimensional): This behavioral descriptor corresponds to the amount of force each leg applies to the ground, measured as a fraction the total possible amount of force that a leg could apply to the ground. Specifically, the measurement is

$$\mathbf{x} = \begin{bmatrix} \frac{F_1}{M_F} \\ \vdots \\ \frac{F_6}{M_F} \end{bmatrix} \tag{4.14}$$

where F_i denotes the ground reaction force (GRF) each leg i generates, averaged over 5 seconds of simulated movement. M_F is the maximum such force that each leg can apply, which is 10N.

8. Relative ground reaction force per leg (6-dimensional): This behavioral descriptor corresponds to the amount of force each leg applies to the ground, relative to that of all the legs:

$$\mathbf{x} = \begin{bmatrix} \frac{F_1}{\sum_{i=1..6} F_i} \\ \vdots \\ \frac{F_6}{\sum_{i=1..6} F_i} \end{bmatrix} \tag{4.15}$$

where F_i denotes the ground reaction force (GRF) each leg i generates, averaged over 5 seconds of simulated movement.

9. Lower-leg pitch angle (6-dimensional): This descriptor captures the pitch angle for the lower-leg with respect to the ground (in a global coordinate frame), averaged over 5 seconds:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t \Theta_1^L(t)}{\pi \times N_1} \\ \vdots \\ \frac{\sum_t \Theta_6^L(t)}{\pi \times N_6} \end{bmatrix} \tag{4.16}$$

where $\Theta_i^L(t)$ is the pitch angle of lower-leg i (hence the L in Θ_i^L) when it is in contact with the ground at time t, and N_i is the number of time-steps for which lower-leg i touches the ground. The foot pitch angles are in range $[0, \pi]$ (as the leg can not penetrate the ground) and normalized to $[0, 1]$.

10. Lower-leg roll angle (6-dimensional): This descriptor captures the roll angle for the lower-leg with respect to the ground (in a global coordinate frame), averaged over 5 seconds:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t \Psi_1^L(t)}{\pi \times N_1} \\ \vdots \\ \frac{\sum_t \Psi_6^L(t)}{\pi \times N_6} \end{bmatrix} \tag{4.17}$$

where $\Psi_i^L(t)$ is the roll angle of lower-leg i (hence L in Ψ_i^L) when it is in contact with the ground at time t, and N_i is the number of time-steps for which lower-leg i touches the ground. The foot roll angles are in range $[0, \pi]$ (as the leg can not penetrate the ground) and normalized to $[0, 1]$.

11. Lower-leg yaw angle (6-dimensional): This descriptor captures the yaw angle for the lower-leg with respect to the ground (in a global coordinate frame), averaged over 5 seconds:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t \Phi_1^L(t)+\pi}{2\pi \times N_1} \\ \vdots \\ \frac{\sum_t \Phi_6^L(t)+\pi}{2\pi \times N_6} \end{bmatrix} \tag{4.18}$$

where $\Phi_i^L(t)$ is the yaw angle of lower-leg i (hence L in Φ_i^L) when it is in contact with the ground at time t,and N_i is the number of time-steps for which lower-leg i touches the ground. The foot yaw angles are in range $[-\pi, \pi]$ and are normalized to $[0, 1]$.

12. Random (6-dimensional): The random behavioral descriptor differs from the other intentionally chosen descriptors in that it does not consist of one type of knowledge, but is instead randomly selected as a subset of variables from the previously described 11 behavioral descriptors. This descriptor is intended to simulate a situation in which one has little expectation for which behavioral descriptor will perform well, so one quickly picks a few different descriptor dimensions without consideration or experimentation. Instead of generating one such list in this fashion, we randomly sample from a large set to find the average performance of this approach over many different possible choices.

    For the random descriptor, each of the 6-dimensions is selected at random (without replacement) from the $1 \times 3 + 10 \times 6 = 63$ available behavior descriptor

dimensions described in the previous 11 descriptors (1 of the above descriptors is three-dimensional and the other 10 are six-dimensional):

$$\mathbf{x} = \begin{bmatrix} R_1 \\ \vdots \\ R_6 \end{bmatrix} \tag{4.19}$$

where R_i denotes the i^{th} dimension of the descriptor, randomly selected uniformly and without replacement from the 63 available dimensions in behavior descriptors 1-11.

It was necessary to compare these behavioral descriptors in simulation because doing so on the physical robot would have required months of experiments and would have repeatedly worn out or broken the robot. We modified the simulator from the previous experiments (section A.1) to emulate 6 different possible damage conditions, each of which involved removing a different leg. The MAP-Elites algorithm, run for 3 million iterations, was used to create the behavior-performance maps for each of the behavioral descriptors (using a simulation of the undamaged robot). During the generation of the behavior-performance maps, the behaviors were stored in the map's cells by discretizing each dimension of the behavioral descriptor space with these five values: $\{0, 0.25, 0.5, 0.75, 1\}$ for the 6-dimensional behavioral descriptors, and with twenty equidistant values between $[0, 1]$ for the 3-dimensional behavioral descriptor. During the adaptation phase and like in the previous experiments, the behaviors were used with their actual values and thus not discretized.
We independently generated eight maps for each of the 11 intentionally chosen behavioral descriptors. Twenty independently generated maps were generated for the random behavioral descriptor. We launched ten replicates of each descriptor for each of the maps (eight for intentionally chosen behavioral descriptors and twenty for random behavioral descriptor) and each of the six damage conditions. There are therefore $10 \times 8 \times 6 = 480$ replicates for each of the intentionally chosen descriptors, and $10 \times 20 \times 6 = 1200$ replicates for the random descriptor. In all these simulated experiments, to roughly simulate the distribution of noisy odometry measurements on the real robot, the simulated performance values were randomly perturbed with a multiplicative Gaussian noise centered on 0.95 with a standard deviation of 0.1.
We analyze the fastest walking speed achieved with each behavioral descriptor after two different numbers of trials: the first case is after 17 trials, and the second case is after 150 trials.

Results The following results include 17 trials on the simulated robot, which was the maximum number of trials required for Intelligent Trial and Error to find a compensatory gait in the previous experiment. The post-adaptation performance achieved with our alternate, intentionally chosen behavioral descriptors (numbers 2-11) was similar to the original duty factor behavioral descriptor (number 1) (Fig. 4.15a).

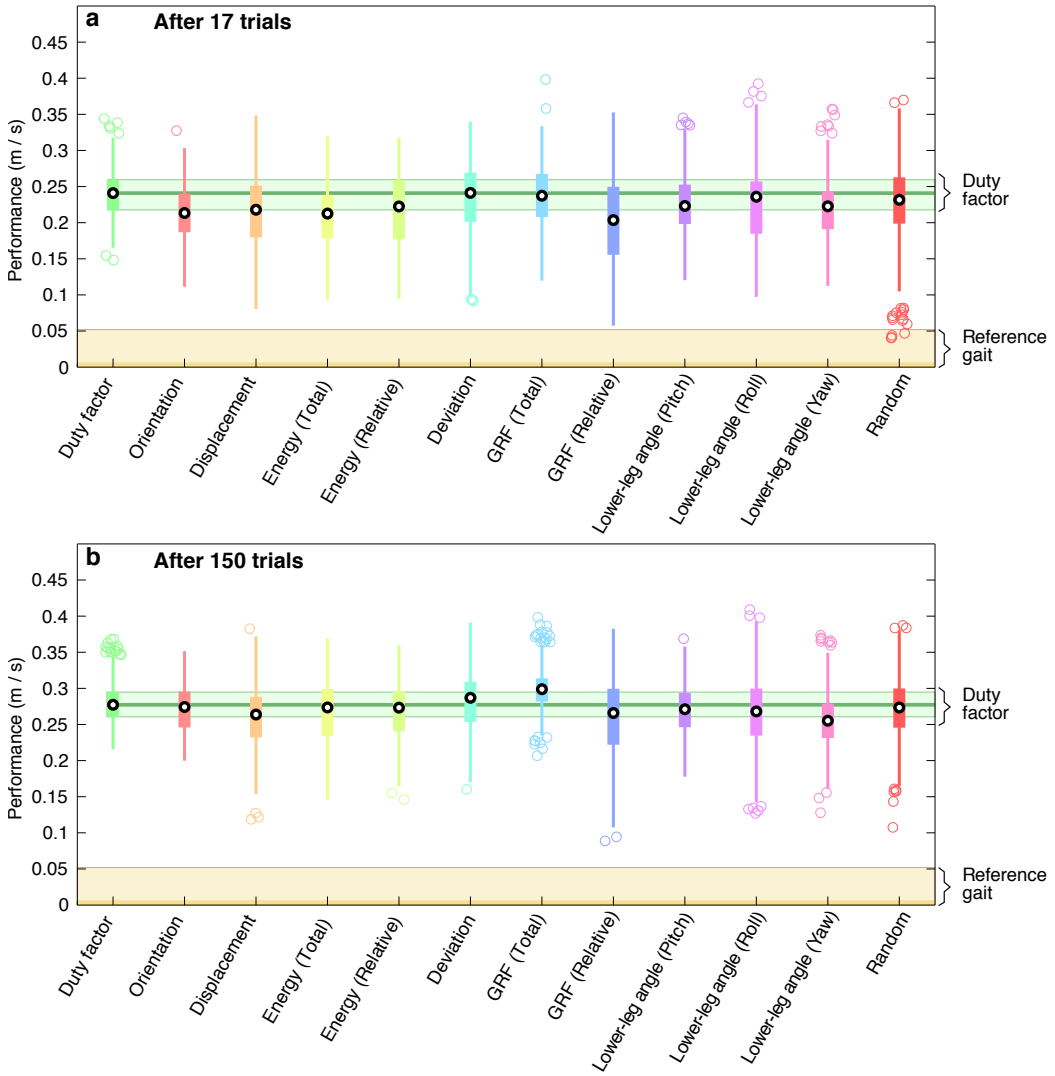


Figure 4.15

All 11 alternate, intentionally chosen descriptors (numbers 2-11) led to a median performance within 17% of the duty factor descriptor (median performance: 0.241 [0.19; 0.29] m/s). The difference in performance was effectively nonexistent with the deviation descriptor (0.241 [0.14; 0.31] m/s), the total GRF descriptor (0.237 [0.15; 0.30] m/s), and the lower-leg roll angle descriptor (0.235 [0.14; 0.31] m/s). The lowest performance was discovered with the relative GRF descriptor (16.7% lower than the duty factor descriptor, 0.204 [0.08; 0.31] m/s). In terms of statistical significance, the performance achieved with the duty factor descriptor was no different from the deviation ($p = 0.53$) and total GRF ($p = 0.29$) descriptors. With all the remaining descriptors, the difference in performance was statistically significant ($p < 10^{-3}$), but it did not exceed 0.04m/s. Additionally, the compensatory behaviors discovered with all our 11 alternate, intentionally chosen descriptors were always faster than the reference gait for all damage conditions.

Figure 4.15: **The Intelligent Trial and Error Algorithm is largely robust to alternate choices of behavior descriptors. (A, B)** The speed of the compensatory behavior discovered by Intelligent Trial and Error for various choices of behavior descriptors. Performance is plotted after 17 and 150 evaluations in panels A and B, respectively. Experiments were performed on a simulated, damaged hexapod. The damaged robot has each of its six legs removed in six different damage scenarios. Data are pooled across all six damage conditions. The evaluated behavior descriptors characterize the following: (i) Time each leg is in contact with the ground (**Duty factor**); (ii) Orientation of the robot frame (**Orientation**); (iii) Instantaneous velocity of the robot (**Displacement**); (iv) Energy expended by the robot in walking (**Energy (Total)**, **Energy (Relative)**); (v) Deviation from a straight line (**Deviation**); (vi) Ground reaction force on each leg (**GRF (Total)**, **GRF (Relative)**); (vii) The angle of each leg when it touches the ground (**Lower-leg angle (Pitch)**, **Lower-leg angle (Roll)**, **Lower-leg angle (Yaw)**); and (viii) A random selection without replacement from subcomponents of all the available behavior descriptors (i-vii) (**Random**). For the hand-designed reference gait (yellow) and the compensatory gaits found by the default duty factor behavior descriptor (green), the bold lines represent the medians and the colored areas extend to the 25^{th} and 75^{th} percentiles of the data. For the other treatments, including the duty factor treatment, black circles represent the median, the colored area extends to the 25^{th} and 75^{th} percentiles of the data, and the colored circles are outliers.

To check whether our alternate, intentionally chosen behavioral descriptors lead to better performance if allowed a higher number of evaluations, we extended the experiments to 150 trials on the robot (Fig. 4.15b). After 150 trials, the difference in performance between the duty factor behavioral descriptor (0.277 [0.24; 0.34] m/s) and our alternate behavioral descriptors was further reduced. For all but three alternate, intentionally chosen descriptors (displacement, total GRF and lower-leg yaw angle), the median performance was within 4% of the duty factor descriptor. The difference in performance was at $\pm 3.6\%$ with the orientation (0.274 [0.22; 0.32] m/s), total energy (0.274 [0.19; 0.33] m/s), relative energy (0.273 [0.20; 0.32] m/s), deviation (0.287 [0.21; 0.34] m/s), relative GRF (0.266 [0.15; 0.35] m/s), lower-leg pitch angle (0.271 [0.21; 0.34] m/s) and lower-leg roll angle (0.268 [0.17; 0.34] m/s) descriptors. In the three remaining behavioral descriptors, displacement, total GRF, and lower-leg yaw angle, the performance was 0.264 [0.18; 0.32] m/s, 0.299 [0.25; 0.35] m/s and 0.255 [0.18; 0.32] m/s, respectively (difference at $\pm 7.8\%$ of duty factor descriptor in all three cases). In terms of statistical significance, the performance achieved with the duty factor descriptor was barely statistically significantly different from the deviation descriptor ($p = 0.041$). In all the remaining descriptors, the performance difference was statistically significant ($p < 10^{-2}$), but no larger than 0.02m/s.

Our random behavioral descriptor also performed similarly to the duty factor descriptor. After 17 trials, the performance of M-BOA with the maps generated by the random descriptor was 0.232 [0.14; 0.30] m/s (4.2% lower than the duty factor descriptor performance). While the difference is statistically significant ($p < 10^{-3}$), the difference in performance itself was negligible at 0.01m/s. This difference in performance was further reduced to 3.6% after 150 trials (random descriptor performance: 0.274 [0.21; 0.34] m/s, duty factor description performance: 0.277 [0.24; 0.34] m/s, $p = 0.002$). Moreover, as with the intentionally chosen behavioral descriptors, the compensatory behavior discovered with the random descriptor was also faster than the reference gait.
These experiments show that the selection of the behavioral dimensions is not critical to get good results. Indeed, all tested behavioral descriptors, even those randomly generated, perform well (median > 0.20 m/s after 17 trials). On the other hand, if the robot's designers have some prior knowledge about which dimensions of variation are likely to reveal different types of behaviors, the algorithm can benefit from this knowledge to further improve results (as with the duty factor descriptor).

4.3.4 Partial conclusion

These five experiments demonstrate that Intelligent Trial and Error allows the robot to both initially learn fast gaits and to reliably recover after physical damage in less than 2 minutes. The results prove that these capabilities are substantially faster than state-of-the-art algorithms (Bayesian optimization and Policy Gradient). Moreover, we saw that Intelligent Trial and Error can help robots to adapt to new environments (e.g. differently sloped terrain) and that our results are qualitatively unchanged when using different behavioral characterizations, including randomly chosen behavioral descriptors. Finally we highlighted through these experiments that reducing the high-dimensional parameter space to a low-dimensional behavior space via the behavior-performance map is the key component for Intelligent Trial and Error (standard Bayesian optimization in the original parameter space does not find working controllers) and that other six-dimentional descriptors perform similarly in simulation than those that have been applied on the physical robot.
While natural animals do not use the specific algorithm we present, there are parallels between Intelligent Trial and Error and animal learning. Damage recovery in animals may occur without learning—for instance, due to the built-in robustness of evolved control loops (Grillner, 2003)—but if such pre-programmed robustness fails, many animals turn to learning (Wolpert et al., 2001). Like our robot, such learning likely exploits an animal's intuitions about how its intact body works to experiment with different behaviors to find what works best. Also like animals (Benson-Amram and Holekamp, 2012), Intelligent Trial and Error allows the quick identification of working behaviors with a few, diverse tests instead of trying behaviors at random or trying small modifications to the best behavior found so far. Additionally, the Bayesian optimization procedure followed by our robot appears similar to the technique employed by humans when they optimize an unknown func-

tion (Borji and Itti, 2013), and there is strong evidence that animal brains learn probability distributions, combine them with prior knowledge, and act as Bayesian optimizers (Pouget et al., 2013; Körding and Wolpert, 2004).
An additional parallel is that Intelligent Trial and Error primes the robot for creativity during a motionless period, after which the generated ideas are tested. This process is reminiscent of the finding that some animals start the day with new ideas that they may quickly disregard after experimenting with them (Derégnaucourt et al., 2005), and more generally, that sleep improves creativity on cognitive tasks (Wagner et al., 2004). A final parallel is that the simulator and Gaussian process components of Intelligent Trial and Error are two forms of predictive models, which are known to exist in animals (Ito, 2008; Bongard et al., 2006). All told, we have shown that combining pieces of nature's algorithm, even if differently assembled, moves robots more towards animals by endowing them with the ability to rapidly adapt to unforeseen circumstances.

4.4 Conclusion

In this chapter, we proposed to consider the question of damage recovery as a trial-and-error learning problem. Indeed, learning appears to be a promising way for robots to autonomously cope with damage conditions or environmental changes, without requiring expensive sensors or engineers attempting to forecast all the situations robots may encounter and the corresponding contingency plans. However, the review of the literature about learning algorithms, made in chapter 2 and in the introduction of this chapter, illustrates that learning algorithms require most of the time several hundreds of trial to learn a behavior, which make their application challenging on physical robots. Through the algorithms presented in this chapter, we show that combining simulation with physical tests is a successful approach to reduce the number of trials required to learn new behaviors and to cope with the reality gap problem.
We propose two different algorithms that are both based on the hypothesis that: *simulations of the intact robot provide useful information that allows damaged, physical robots to quickly adapt.* The experimental results of these two algorithms show that this hypothesis holds for a large set of damage situations and allows the algorithms to require only a few dozen of physical trials. Situations in which this hypothesis does not hold are likely to exist, but our experiments reveal that they are rare. In chapter 6, we discuss in more details the situations in which no working solutions can be found in the behavioral-performance maps.
With our first algorithm, T-Resilience, we proposed to consider damage situations as a source of reality gap, which can be handled by the transferability approach. This approach, which guides an evolutionary process toward solutions that work similarly in simulation and in reality, allowed our robot to cope with a large variety of damage situations in only 20 minutes and 25 physical trials. However, with this algorithm, the majority of the learning time is spent to run the evolutionary

algorithm in simulation for a dozen of generations between each trial and it requires a significant computational power accessible to the robot.
Based on these first results, we designed our second algorithm, the Intelligent Trial and Error algorithm, which uses pre-computed behavior-performance maps generated via an evolutionary algorithm to guide a policy search algorithm. Our method allows robots to adapt to a large variety of situations in less than two minutes and a dozen of trials. The videos of the experimental results show that the behaviors generated with this algorithm are more dynamic and insect like than those generated with the T-Resilience. This observation stems from the ability of the Intelligent Trial and Error algorithm to produce behaviors that are finely tuned in simulation without slowing down the adaption process.
Our experiments revealed that the abilities of our robot to cope with a large variety of situations come form large diversity of high-performing behaviors included in the behavior-performance maps generated with MAP-Elites. This type of behavioral repertoires is an effective way to encapsulate the creativity of evolutionary algorithms by creating a smaller search space only filled by high-performing behaviors. By seeding a Gaussian Process with the information contained in the maps, M-BOA maximizes the usefulness of this source of knowledge and allows our robot to adapt in less than 2 minutes. It also worth noting that M-BOA requires little computational power and can run on an embedded computer or a smartphone, as its main step consists in inverting a matrix.
In the next chapter, we will show how the Intelligent Trial and Error algorithm can be extended to deal with three question or problems that commonly affect robotic experiments: (1) transferring knowledge from one task to the next ones; (2) dealing with solutions that cannot be evaluated on the physical robot and may hurt the learning process; and (3) adapting prior knowledge (like behavior-performance maps), that may be initially misleading in some circumstances, to maximize their usefulness. With these extensions, the Intelligent Trial and Error algorithm aims to become a generic framework that can be applied to many types of robot to allow them to quickly learn a large variety of tasks.

Chapter 5

Knowledge Transfer, Missing Data, and Misleading Priors

The results and text of this chapter have not yet been published.

Other contributors:

- Jean-Baptiste Mouret, Pierre and Marie Curie University (Thesis supervisor)

Author contributions:

- **A.C.** and J.-B.M. designed the study. **A.C.** performed the experiments. **A.C.** and J.-B.M. analyzed the results.

Contents

5.1 Introduction

In the previous chapter, we saw how the creativity of evolutionary algorithms can be used to automatically discover vast varieties of behaviors. These range from walking in every direction, to the numerous possible ways to walk in a straight line. We also saw how this creativity can be combined with machine learning algorithms (Support Vector Machines, (Cortes and Vapnik, 1995; Schölkopf and Smola, 2002), and Bayesian Optimization (Mockus, 2013)) to allow our robot to deal with unforeseen situations; like mechanical damages. However, these advances are far from solving all the issues that usually affect learning experiments applied on physical robots. During long term missions, like planet or deep ocean exploration (Sanderson, 2010; Yoerger, 2008), autonomous robots have to use a large variety of behaviors, to adapt each of these behaviors according to the encountered situations, to deal with sensing problems, or to adapt to the inaccuracies of the information that have been provided to the robot.

In this chapter, we will consider three aspects that often prevent robots from being fully autonomous and robust in practical problems. In the following section, we will present a method that allows robots to transfer knowledge between tasks. Such ability is beneficial because the methods proposed in the previous chapter allow the robot to adapt its behaviors to the current situation, but only one behavior at a time. Consequently, the robot will have to repeat this process for all the behaviors contained in its repertoire. This can make the adaptation process very time consuming, even if it takes less than two minutes per behavior. Transferring knowledge across tasks should reduce the required time to learn how to achieve the subsequent tasks.

In the second part of this chapter, we will propose an extension of our Intelligent Trial and Error algorithm to deal with solutions or behaviors that cannot be evaluated. This may happen for various reasons, such as safety issues or sensor limitations, and may result in missing data during the learning phase. Depending on the way this missing data is handled, the learning performances can be dramatically impacted. Our method allows the robot to be robust to missing data and minimize its negative impact on learning performance.

In the third section of this chapter, we consider the use of prior information that is not well-suited to the robot and may hurt the learning performance. For example, using a forward model with a metric scale may mislead a robot that perceives its position in pixels. However, when the prior information is initially not well-suited, it does not mean that it does not contain any useful information. We propose an extension of our algorithm to automatically adapt the prior information to maximize their potential utility.

These three extensions of our framework will be evaluated individually with simulated experiments in order to gather statistics and to demonstrate their benefits. In the last part of this chapter, the entire framework will be evaluated on a physical setup where all these improvements are required to be able to complete the robot's mission.

5.2 Knowledge Transfer

5.2.1 Motivations

In order to accomplish their missions, which will become steadily more complex, robots have to be able to perform not only a single task, but also a large variety of them. For example, a legged robot has to be able to walk in every direction (see chapter 3), or a robotic arm needs to be able to reach multiple targets. Moreover, some of these tasks may depend on the current situation, or on unpredictable factors. The robot might also be confronted with additional tasks that have not been anticipated. For example, a new bin can be added to the scene and the robotic arm can be asked to reach it after having learned to reach the already present bins. For these reasons, it is likely that the number of tasks or the tasks themselves are unknown at the beginning of the robot's mission.
As a consequence of this variety of tasks, in an unexpected situation (e.g., like after mechanical damage), the robot will have to adapt not only one, but all its behaviors to continue its mission. Even if fast adaptation is possible (see chapter 4), it seems ineffective to re-learn (adapt) each behavior independently, as some knowledge can be common between them. Transferring this knowledge from one learning phase to another may improve the adaptation abilities of our robots and allow them to learn new, unanticipated tasks faster. For example, the robotic arm could re-use the information gathered when learning how to reach the first bin in order to reach the second bin faster. Similarly, the hexapod robot could use the knowledge that avoiding the use of certain legs allows it to walk forward, to learn how to walk backward.
Knowledge transfer is a long-standing question in artificial intelligence that initially emerged from classification problems, where information can be transferred from one data set to others (Thrun and Pratt, 1998). In classification, some features can be shared among different classes in different data sets (or classification scenarios). For example, learning how to classify buses can be aided by features of cars, or motorbikes by features of bicycles. After learning one of these instances, we can expect to learn to recognize the second one quicker. This concept of transferring knowledge has several names in the literature, for example, Transfer Learning (Taylor and Stone, 2009; Taylor et al., 2007; Singh, 1992), Multi-Task Learning (Micchelli and Pontil, 2004; Bonilla et al., 2007; Williams et al., 2009; Yu et al., 2005), or Contextual Policy Search (Kupcsik et al., 2014). It also has several definitions, for example, Taylor and Stone (2009) state that "the insight behind transfer learning is that generalization may occur not only within tasks, but also across tasks." However, one may argue that Knowledge Transfer is more than across-tasks-transfer; it can be across environments (which may be considered as different tasks as well), or across several robots. For example, in social learning (Galef and Laland, 2005) the source of knowledge is not only the past experience of the agent, but also the experience of the other individuals. Knowledge Transfer has been applied in several research domains, for example, psychology, or ethology (Thorndike and Woodworth, 1901;

Skinner, 1953; Billing, 2007; Galef and Laland, 2005), machine learning (classification or regression) (Thrun and Pratt, 1998; Thrun, 1996), reinforcement learning (Taylor and Stone, 2009; Konidaris and Barto, 2006), neural networks (Caruana, 1997; Shultz and Rivest, 2000), and robotics (Doncieux, 2013; Kupcsik et al., 2014; Benureau and Oudeyer, 2013).

In spite of all these studies concerning the concept of knowledge transfer, several questions remain open. In Pan and Yang (2010), the authors highlight three fundamental questions: "In transfer learning, we have the following three main research issues: (1) what to transfer, (2) how to transfer, (3) when to transfer". However, from our point of view, one of the main questions may be "what is in common to all the robot's tasks or environments?" and the answer of this question would likely solves the three previously exposed questions.

It is very likely that what is common to all the tasks is the robot itself. In most robotics applications, the tasks can be expressed according to the robot's state. For example, reaching a target with a robotics arm means to place the robot's end effector at a particular location and walking forward can be expressed as moving the center of mass of the robot. For robotics manipulation, the state of the robot can include the state of the manipulated object. In the same way, all the observations can be expressed as a part of the robot's state.

Consequently, knowledge can be easily transferred if all the acquisitions are related to robot's state, and if all the tasks are expressed in terms of the robot's state (as in state-space control, Friedland (2012)). In other words, by building a *generic knowledge* centered on the robot itself (the robot state function), all this knowledge can be used for each tasks, contrary to the performance functions associated to the tasks that are *task specific* (for example; the distance to a target).

In the following sections, we will present a simple way to integrate this knowledge transfer ability in *Bayesian Optimization* (BO, see section 2.4) and in our Intelligent Trial and Error algorithm (see chapter 4). While the presented method is fully compatible with the Intelligent Trial and Error algorithm, in these sections, we will only consider the traditional expressions of BO, in order to keep the equations as simple as possible.

5.2.2 Principle

In the traditional Bayesian Optimization framework, like in most learning algorithms (see chapter 2), the task (or problem) is formalized as a cost or reward function that needs to be optimized. We can take the example of the robotic arm reaching a target bin; its cost function is usually defined as the distance between its gripper and the target (here expressed as a reward function that has to be maximized, with $\mathbf{x}$ as a potential solution):

$$\text{Reward}(\mathbf{x}) = -\text{dist}(\text{Target} - \text{State}(\mathbf{x})) \tag{5.1}$$

The learning algorithm then performs several evaluations and uses the corresponding observations of the reward values to train the GP that approximates the reward

function (see fig. 5.1A). The acquisition function exploits this model of the reward function to select the next potential solution that will be tested:

$$\begin{array}{rcccl} \text{Observations} & = & \text{Reward}(\chi_{1:t}) & & \\ \text{Model}(\mathbf{x}) & = & GP(\text{Observations}) & = & \widetilde{\text{Reward}}(\mathbf{x}) \\ \text{Acquisition}(\mathbf{x}) & = & UCB(\text{Model}(\mathbf{x})) & = & \mu(\mathbf{x}) + \alpha.\sigma(\mathbf{x}) \end{array} \tag{5.2}$$

where GP stands for Gaussian Process and UCB refers to an acquisition function (see section 2.4.3). $\chi_{1:t}$ corresponds to the trials performed up to the time-step t. We can observe that in this context the observations, the model and the acquisition function depend on the "Target" variable. They are thus task-specific and need to be changed each time the robot considers a different target.
When our robotic arm has to reach several targets, each of these targets corresponds to a sub-reward function:

$$\text{Reward}_i(\mathbf{x}) = -\text{dist}(\text{Target}_i - \text{State}(\mathbf{x})) \tag{5.3}$$

Two alternatives can be considered to deal with these multiple targets. Probably, the most common one is to consider all these targets as a single task (i.e., reaching targets), for example by summing or averaging the sub-reward functions:

$$\text{Reward}(\mathbf{x}) = -\frac{1}{N}\sum_{i=1}^{N} \text{dist}(\text{Target}_i - \text{State}(\mathbf{x})) \tag{5.4}$$

The learning algorithm then searches for the solution (or the set of solutions) that maximizes this global reward function. Nonetheless, in this case, the number of targets and the targets themselves have to be known a priori to define this reward function, which prevents the algorithm to precisely deal with unanticipated new tasks. In some cases depending on the type of controller, the solutions may have generalization abilities. For example, close-loop controllers that take in input a target location, may be able to generalize over new targets after being trained on a few predefined ones. However, in this case, the performance of the algorithm directly depends on the generalization ability of the solutions, which is a long standing question in machine learning (Cesa-Bianchi et al., 2004; Cohn et al., 1994).
The second alternative consists in considering the targets individually by their corresponding reward functions ($\text{Reward}_i(\mathbf{x})$). In this case, the robot has several tasks to solve (because it has several reward functions) and these tasks can be initially unknown and added during the robot's mission. The method can be seen like several learning processes running in parallel to solve independently all the tasks. In this situation, the knowledge transfer across these learning processes can be very beneficial, as it can allow the algorithm to achieve the new tasks faster. Unfortunately, this transfer is impossible if we use the common BO framework presented above (or most of other learning algorithms' framework), because everything, from the observations to the acquisition functions, is task-specific (i.e., relative to one target).

The method that we will introduce in the next section will rely on this second alternative solution (that considers each target as an individual task) but we will slightly change the traditional BO framework to make it robot-specific instead of task-specific. By doing so, the algorithms will be able to solve all the tasks and to transfer knowledge across them.

5.2.3 Method Description

As introduced before, the robot's state function is the common part of all the robot's tasks. The main idea of our method is thus to transfer knowledge about this state function instead of knowledge about the reward functions. This idea of state-based knowledge transfer can be integrated into Bayesian Optimization by estimating the robot's state function via Gaussian Processes (see section 2.4.2). This model of the state function, which is robot-specific, can be specialized according to a current task in order to generate a model of the reward function of this task, which can then be used to learn the task. This specialization of state function model can be done for all the tasks that the robot has to address. The Bayesian Optimization framework can easily handle these specializations by computing the reward function in the acquisition function based on the mean of the Gaussian Process.
In state-based Bayesian optimization, the observations and thus the model (GP) represent the robot state function (see fig. 5.1B). Based on this generic knowledge about the robot, we can specialize the acquisition function according to the current task (or target):

$$\begin{aligned} \text{Observations} &= \text{State}(\chi_{1:t}) \\ \text{Model}(\mathbf{x}) &= GP(\text{Observations}) \quad = \widetilde{\text{State}}(\mathbf{x}) \\ \text{Acquisition}(\mathbf{x}) &= UCB\Big(\text{Reward}\big(\text{Model}(\mathbf{x})\big)\Big) \\ &= -\text{dist}(\text{Target} - \mu(\mathbf{x})) + \alpha.\sigma(\mathbf{x}) \end{aligned} \tag{5.5}$$

Here, only the acquisition function is specific to the tasks and needs to be changed when the robot switches from one task to another. New tasks can be added by changing the target variable too, and all the observations obtained so far will be used to solve this new task. This method progressively builds up and refines the model of the robot's state function based on all the observations acquired during each task, and uses this model to perform the tasks. This method thus builds and exploits at the same time its model while learning to achieve the different tasks that the robot accomplishes. Such an approach has already been used and proved to be effective, for example in Kupcsik et al. (2014); Deisenroth and Rasmussen (2011). However, in these studies, the built models are used to predict the step-based reward (or state) values several time-steps ahead, which is computationally costly, and they use these prediction to update the policy thanks to, for example, the REPS algorithms (see section 2.3 and (Peters et al., 2010)). In this chapter, we propose to use the model to directly update and compute the acquisition function of the BO algorithm according to the current target. This methods is both computationally

effective and easy to implement.
Figure 5.1 shows the differences between the standard reward-based BO and the State-based BO. It also shows that some situations can be deceptive and how these two different approaches handle this situation. According to its two observations, the reward-based BO (fig. 5.1A) expects a growth of the reward function and thus proposes to test a sample far from the actual position of the target. The state-based BO deals easily with this situation because it obtains more information from the observations. In this example, one observation says that the target is on the left, the second says that the target is on the right, thus the algorithm knows that it has to select a sample in between. This property may allow the algorithm to perform better in some difficult situations, as it will be suggested by the experimental results presented in this chapter.
This approach has several advantages. Firstly, it is able to consider on the fly new tasks that have not been anticipated. For example, after having learned to reach 20 targets, the robot can learn to reach an additional target randomly generated. When using algorithms that consider all the different targets as a single task, this situation is typically handled by the generalization ability of the obtained controllers, which may be ineffective when the task space is large and the number of training targets low. An important aspect of our approach is that all observations acquired when learning the previous tasks are used to learn the new ones, maximizing the knowledge transfer abilities of the algorithms. Moreover, nothing prevents our approach to change its current task arbitrarily during the learning processes, instead of considering them only consecutively. This ability may be useful if the robot needs to improve its performance simultaneously on several tasks or if one task becomes suddenly a priority. For example, if a walking robot learns to walk in every direction, it can change its current task (i.e., the considered direction of the movement) according to the situation, even if it has not reached a stopping criterion. Typically, if the robot learns to walk forward as fast as possible, it will be useful for the robot to change its target direction and start learning to turn before reaching an obstacle, even if the maximum forward speed has not been reached yet. Secondly, the computational cost of changing tasks or adding new ones is very low because the same state function model is used for all the tasks and the task specification is only needed when looking at the maximum of the acquisition function. Consequently, no costly matrix inversion or hyper-parameter tuning is required when considering different tasks. Lastly, the ability of the algorithm to progressively build its model according to the tasks it has to solve makes the algorithm data efficient. This active learning [1] property of our approach allows the model to be very accurate on regions actually useful for the robot's mission, instead of trying to model accurately the

[1] This property corroborates the observation from (Baranes and Oudeyer, 2013) saying that sampling the goal space (goal babbling) is more efficient than sampling the motor space (motor babbling), because of the potential high redundancy in the motor space. Indeed, several solutions in the motor space may lead to the same state (or behavior), thus learning a mapping between the entire motor space to the state space can be inefficient if the objective is to find only motor solutions that complete the task.

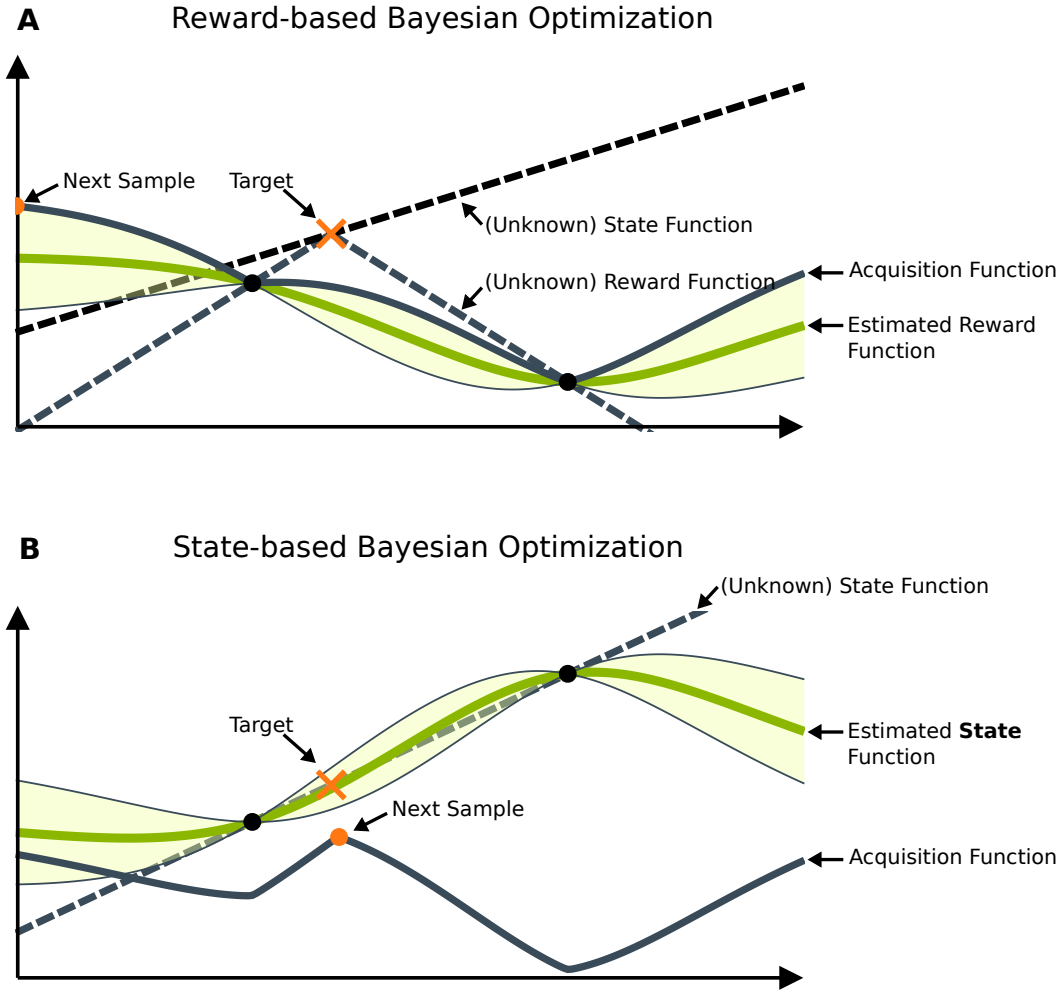


Figure 5.1: Differences between reward-based and State-based Bayesian Optimization. (A) In reward-based Bayesian Optimization, the acquisition function (for example UCB, see chapter 2.4.3) is computed based on the observations of the reward function. (B) In State-based Bayesian Optimization, the acquisition function is computed based on the observations of the state function, and it is specialized according to the target location. Changing the task only requires changing the target location without changing the state function model (i.e., the GP).

entire state function of the robot that may contain useless regions.
Robots' state is often expressed with multi-dimensional vectors. For example, the state of the robotic arm can be the final position of the end-effector, which is a 3 dimensional vector. However, in the vast majority of Bayesian Optimization studies, the functions approximated with the Gaussian Process are scalar functions. In the following sections, we will show how Gaussian Processes can be extended to multi-dimensional regression in order to allow our method to be applied on multi-dimensional state functions. In the experimental results section, we will show how this method can be applied on our robotic arm to successively learn 20 randomly generated targets and how it performs compared to traditional methods.

5.2.4 Multi-Channels Regression with Bayesian Optimization

5.2.4.1 Related works

In most of experiments that use Gaussian Processes (GP), GP are employed to approximate mono-dimensional functions, like reward or cost function (Lizotte et al., 2007; Calandra et al., 2014; Kober et al., 2012; Martinez-Cantin et al., 2007, 2009):

$$f : \mathbb{R}^n \to \mathbb{R} \tag{5.6}$$

Nevertheless, the robots state functions are most of the time multi-dimensional functions (for example the 3D position of the end effector of a robotic arm):

$$f : \mathbb{R}^n \to \mathbb{R}^m \tag{5.7}$$

For the remaining of this manuscript, we will distinguish the input space ($\mathbb{R}^n$) and the output space ($\mathbb{R}^m$) by calling the dimensions of the output space "channel". In this section, we will demonstrate that Gaussian Processes can be extended to deal with functions that have multi-dimensional input space and multiple output channels.
There are two alternatives to achieve multi-channel regression with Gaussian Processes: (1) Considering each channel separately and using on GP per channel, or (2) considering all the channels together and using one single model for all the channels. The first method, also known as multi-kriging (Williams and Rasmussen, 1996), considers independently each channel of the function. It has, for example, been used in Kupcsik et al. (2014) in which the authors used individual GP models for each output channel in order to approximate states of the robot and the environment. They applied their method on two robotic setups where robots had to throw projectiles to distinct targets. Based on the acquired state function model, their algorithm predicts the expected reward used afterward in a policy search algorithm that efficiently learns robot skills. While, using independent GPs is relatively straightforward, it suffers from a scalability problem. The computational cost will be strictly proportional to the number of channels, as this approach requires computing the inverse kernel matrix and to optimize the likelihood for each GP. For some situations where a large amount of data (several hundreds) is required, the

computational cost of one GP starts to be all but negligible and may prevent the regression of multi-dimensional functions. In Kupcsik et al. (2014), the computation time seems to represents one of the major obstacles that the authors addressed. For example, they investigated the impact of using GPUs on the computational time when sampling and computing predictions. The authors also investigated the use of sparse Gaussian Processes (Snelson and Ghahramani, 2005; Quiñonero-Candela and Rasmussen, 2005; Titsias, 2009) that are devoted to approximate the GPs by reducing the size of the kernel matrix by selecting a few samples that represent all the data set.

The second alternative consists in using a single model for all the channels, which may allow data acquired about one channel to influence the other channels. This link between the different channels can be useful when a strong correlation exists between the channels. For example, when the channels are sampled independently, the algorithm can infer the shape of the function in channels that are not well sampled based on the samples coming from other channels. In Seeger et al. (2005), the authors give the example of estimating the concentration map of Uranium. The problem is that detecting uranium concentration is difficult, while detecting carbon is easier. The idea was to sample more densely the concentration of carbon in order to infer the concentration of uranium because these two "outputs" are strongly correlated.

One method (Rasmussen and Williams, 2006) that exploits this correlation consists in defining different covariance functions to deal with samples coming from the different channels ($k^{i,j}(\mathbf{x}^i, \mathbf{x}^j)$). One covariance function is used to compare samples that come from the same channel (in this case $i = j$), while samples that come from different channels (for example samples of uranium density and samples of carbon density) are compared with a different covariance function (in this case $i \neq j$). We can note that typically there is one covariance function of each channel ($k^{i,i} \neq k^{j,j}$) and one covariance function for each couple of channels ($k^{i,j} \neq k^{j,k}$). Now, let consider M different data sets, one for each channel $i \in M$, $D^i : \{\mathbf{x}^i, y^i\}$. The observations from all channels are concatenated and processed together using the standard Gaussian Process framework:

$$P(f(\mathbf{x})^i|D^N,\mathbf{x}) = \mathcal{N}(\mu_t^i(\mathbf{x}), {\sigma_t^i}^2(\mathbf{x}))$$

where :

$$\mu_t^i(\mathbf{x}) = \mu_0^i + \mathbf{k}^{i\intercal}\mathbf{K}^{-1}(\mathbf{P}_{1:t} - \boldsymbol{\mu_0})$$

$${\sigma_t^i}^2(\mathbf{x}) = k^{ii}(\mathbf{x},\mathbf{x}) - \mathbf{k}^{i\intercal}\mathbf{K}^{-1}\mathbf{k}^i$$

$$\mathbf{K} = \begin{bmatrix} K^{1,1} & \cdots & K^{1,i} & \cdots & K^{1,M} \\ \vdots & & \ddots & & \vdots \\ K^{i,1} & \cdots & K^{i,i} & \cdots & K^{i,M} \\ \vdots & & \ddots & & \vdots \\ K^{M,1} & \cdots & K^{M,i} & \cdots & K^{M,M} \end{bmatrix}$$

$$\mathbf{K}^{i,j} = \begin{bmatrix} k^{i,j}(\boldsymbol{\chi}_1^i, \boldsymbol{\chi}_1^j) & \cdots & k^{i,j}(\boldsymbol{\chi}_1^i, \boldsymbol{\chi}_{t_j}^j) \\ \vdots & \ddots & \vdots \\ k^{i,j}(\boldsymbol{\chi}_{t_i}^i, \boldsymbol{\chi}_1^j) & \cdots & k^{i,j}(\boldsymbol{\chi}_{t_i}^i, \boldsymbol{\chi}_{t_j}^j) \end{bmatrix}$$

$$\mathbf{k}^i = \begin{bmatrix} k^{i,1}(\mathbf{x}, \boldsymbol{\chi}_1^1) & \cdots & k^{i,1}(\mathbf{x}, \boldsymbol{\chi}_{t_1}^1) & \cdots & k^{i,M}(\mathbf{x}, \boldsymbol{\chi}_1^M) & \cdots & k^{i,M}(\mathbf{x}, \boldsymbol{\chi}_{t_n}^M) \end{bmatrix}^\intercal$$

$$\mathbf{P}_{1:t} = \begin{bmatrix} P_1^1 & \cdots & P_{t_1}^1 & \cdots & P_1^M & \cdots & P_{t_n}^M \end{bmatrix}^\intercal$$

$$\boldsymbol{\mu_0} = \begin{bmatrix} \mu_0^1 & \cdots & \mu_0^1 & \cdots & \mu_0^M & \cdots & \mu_0^M \end{bmatrix}^\intercal \tag{5.8}$$

P_t^m represents the t^{th} observation on the m^{th} channel and μ_0^m corresponds to the initial mean value for the m^{th} channel. The correlations between the different channels are automatically inferred thanks to the different hyper-parameters of each $k^{i,j}$ function that are autonomously defined by the log-likelihood optimization (see chapter 2.4.2.2). This method is promising when the samplings differ from one channel to the others (i.e., when $\mathbf{x}^i \neq \mathbf{x}^j$, Seeger et al. (2005)) because it allows the algorithm to use data from the other channels to make more accurate predictions on areas that have not been sampled in the considered channel. Unfortunately, this approach also increases dramatically the computational cost, as the size of the kernel matrix is equal to the total number of samples on all channels:

$$\text{size}(K) = \sum_{i \in M} t_i \tag{5.9}$$

This growth of the kernel size directly impacts the computational cost of the algorithm, as the complexity of the matrix inversion is $O(n^3)$. In the cases where all the channels are sampled simultaneously ($\mathbf{x}^i = \mathbf{x}^j$ and $t_i = t_j \ \forall i,j \in M$), this means that the computation cost is multiplied by M^3, M being the number of channels. Moreover, here we only consider the inversion of the kernel matrix; the optimization of the hyper-parameters is even more impacted in this method. The number of hyper-parameters is multiplied by $M*(M+1)/2$ (or M^2 if $k^{i,j} = k^{j,i}$) increasing the complexity, and the required time, of the likelihood optimization. These points show clearly that this method cannot be extended to a large number of channels (scalability problem). Several works try to mitigate this limitation thanks to computational approximations of the kernel matrix (Micchelli and Pontil (2004); Seeger et al. (2005); Boyle and Frean (2005)).

This approach of multiple covariance functions for multi-tasks and knowledge transfer has inspired several works using various techniques to combine the covariance functions. For example, Boyle and Frean (2005) define two covariance functions, one for the auto-covariance (i.e., from the same channel) and another for the cross-covariance (i.e., between two different channels). Bonilla et al. (2007) use a Kronecker product to combine covariance functions and then tune an inter-task covariance matrix thanks to likelihood optimization. This technique has been employed to control a robotic arm in different payload context to reach different targets (Williams et al., 2009; Yeung and Zhang, 2009)

5.2.4.2 Fast multi-Channels regression

In this chapter, our objective is to extend the abilities of our algorithms without losing in computational speed. We decided to use a single model for all the channels, but, in order to mitigate the scalability issue and reduce the computational cost, the regression is achieved by using the same kernel matrix and the same hyper-parameters for each channel and only changing the initial mean function and the observations according to the considered channel. This approach is equivalent to having one GP per channel but, in this case, the GPs share the same kernel matrix and hyper-parameters. Because the vast majority of the computational cost comes from the hyper-parameter optimization and the inversion of the kernel matrix, sharing these components over the GPs makes the computational cost independent of the number of channels. This method can consequently be applied on functions with a large number of channels, but it makes the assumption that every channel can be approximated using the same hyper-parameters. For the experiments of this chapter, we validated this assumption by independently modeling each channel with different GPs (first alternative) and comparing the hyper-parameters obtained after likelihood optimization. The results (not presented in this manuscript) showed that the hyper-parameters converged to the same values. One of our hypotheses that may explain why this assumption holds in our case is that the channels are sampled simultaneously (each sampled point provides information for all the channels), which is not always possible (see the uranium mapping example exposed before). Another hypothesis is that, in our experiment, the two channels are likely to be similarly influenced by each observation, as the variations of the X or Y coordinates are likely similar. For example, with a one degree of freedom robot, the X/Y coordinates correspond to $L * \cos(\theta), L * \sin(\theta)$, which show the same variations (with an offset of $\pi/2$).

From a mathematical point of view, this approach can be formulated by:

$$
\begin{aligned}
& P(f(\mathbf{x})|\mathbf{Pc}_{1:t}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu_t}(\mathbf{x}), \sigma_t^2(\mathbf{x})\mathbf{I}) \\
& \text{where :} \\
& \boldsymbol{\mu_t}(\mathbf{x})^\intercal = \boldsymbol{\mu_0} + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{Pc}_{1:t} - \boldsymbol{\mu c_0}) \\
& \sigma_t^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1}\mathbf{k} \\
& \mathbf{K} = \begin{bmatrix} k(\boldsymbol{\chi}_1, \boldsymbol{\chi}_1) + \sigma_{noise}^2 & \cdots & k(\boldsymbol{\chi}_1, \boldsymbol{\chi}_t) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{\chi}_t, \boldsymbol{\chi}_1) & \cdots & k(\boldsymbol{\chi}_t, \boldsymbol{\chi}_t) + \sigma_{noise}^2 \end{bmatrix} \\
& \mathbf{k} = \begin{bmatrix} k(\mathbf{x}, \boldsymbol{\chi}_1) & k(\mathbf{x}, \boldsymbol{\chi}_2) & \cdots & k(\mathbf{x}, \boldsymbol{\chi}_t) \end{bmatrix}^\intercal \\
& \mathbf{Pc}_{1:t} = \begin{bmatrix} P_1^x & \dots & P_1^z \\ \vdots & \ddots & \vdots \\ P_t^x & \dots & P_t^z \end{bmatrix} \\
& \boldsymbol{\mu c_0} = \begin{bmatrix} \boldsymbol{\mu_0} \\ \vdots \\ \boldsymbol{\mu_0} \end{bmatrix} \\
& \boldsymbol{\mu_0} = \begin{bmatrix} \mu_0^x & \dots & \mu_0^z \end{bmatrix}
\end{aligned}
\tag{5.10}
$$

The main differences with the classic Gaussian Process expression (see section 2.4.2) are in blue. The idea consists in changing the $\mathbf{P}_{1:t}$ vector, which usually contains the scalar observation when the approximated function has a single channel, into a matrix where each line represents a multi-dimensional observation. We call this matrix $\mathbf{Pc}_{1:t}$. The mean constant is also extended into a matrix in order to set the initial mean at different values according to the channel. Each line of $\boldsymbol{\mu c_0}$ is identical and contains the initial mean value of the each channel ($\boldsymbol{\mu_0}$). Thanks to these two modifications, the mean function of the GP outputs a line vector gathering all channels of the approximated functions in one vector.
Usually, vectors are represented in column. The previous expression can be adapted in order to output column vectors simply by transposing the expression:

$$
\begin{aligned}
& P(f(\mathbf{x})|\mathbf{Pc}_{1:t}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu_t}(\mathbf{x}), \sigma_t^2(\mathbf{x})\mathbf{I}) \\
& \text{where :} \\
& \boldsymbol{\mu_t}(\mathbf{x}) = \boldsymbol{\mu_0}^\intercal + (\mathbf{Pc}_{1:t}^\intercal - \boldsymbol{\mu c_0}^\intercal)\mathbf{K}^{-1}\mathbf{k} \\
& \sigma_t^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1}\mathbf{k}
\end{aligned}
\tag{5.11}
$$

This method is completely scalable to the number of channels and has a negligible impact on the computation cost. This aspect will allow us to model multi-dimensional robot state function and transfer knowledge from one task to another.

5.2.4.3 Likelihood Optimization of Multi-Channels models

The hyper-parameters of a multi-channels Gaussian Process can be automatically determined in the same way as with standard Gaussian Processes by using the log-likelihood optimization (see chapter 2.4.2.2 and Rasmussen and Williams (2006)).

As a reminder, here is the original log-likelihood expression:

$$\log p(\mathbf{P}_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) = -\frac{1}{2}(\mathbf{P}_{1:t} - \mu_0)^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t} - \mu_0) - \frac{1}{2}\log \mid \mathbf{K} \mid -\frac{n}{2}\log 2\pi \tag{5.12}$$

Our multi-channels GP can be decomposed on several independent mono-channel GPs, one for each channel (see the previous section). Based on this observation, we can compute the likelihood of the multi-channels model as the product of the likelihood of each channel. The log operator and the matrix representation of the different observation over all the channel ($\mathbf{Pc}_{1:t}$) allows the expression to be similar to the original one (equation 5.12):

$$\begin{aligned} \log p(\mathbf{P}^1_{1:t} \dots \mathbf{P}^M_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) &= \log \textstyle\prod_{k=1}^{M} p(\mathbf{P}^k_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) \\ &= \textstyle\sum_{k=1}^{M} \log p(\mathbf{P}^k_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) \\ &= -\frac{1}{2}\mathrm{tr}((\mathbf{Pc}_{1:t} - \boldsymbol{\mu c_0})^\intercal \mathbf{K}^{-1}(\mathbf{Pc}_{1:t} - \boldsymbol{\mu c_0})) \\ &\quad -\frac{M}{2}\log \mid \mathbf{K} \mid -\frac{n * M}{2}\log 2\pi \end{aligned} \tag{5.13}$$

The differences are depicted in blue in the final expression. According to our knowledge, the trade-off between data-fit term and the complexity penalty defined in Rasmussen (1996) is not perfect and may lead to over-learning phenomenon usually counterbalanced with cross validation (see chapter 2.4.2.2). One of the differences between the equations 5.12 and 5.13 consists in multiplying by M the term that regulates the model complexity (Rasmussen and Williams, 2006) in order to conserve the same trade-off. Interestingly this modification did not provide the expected results during our preliminary experiments, most likely because of early convergence to local maximum during the log-likelihood optimization. A modification that reduces the complexity penalty will systematically be more interesting (for the optimization process) than a modification that improves the data-fit term because it is very unlikely that this modification improves in the same way all the channels (to get a improvement multiplied by a factor of M). Consequently, the optimization process will be attracted by the local maximum that tends to have the simplest models. For this reason we decided for our experiments to remove this M factor in the likelihood expression, which leads to this simplified expression:

$$\begin{aligned} \log p(\mathbf{Pc}_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) = -\frac{1}{2}\mathrm{tr}((\mathbf{Pc}_{1:t} - \boldsymbol{\mu c_0})^\intercal \mathbf{K}^{-1}(\mathbf{Pc}_{1:t} - \boldsymbol{\mu c_0})) \\ -\frac{1}{2}\log \mid \mathbf{K} \mid -\frac{n}{2}\log 2\pi \end{aligned} \tag{5.14}$$

The gradient of this expression can be easily computed as the sum of the gradient

of the different data-fit terms (one observation vector per channel):

$$\begin{aligned}\frac{\partial}{\partial\theta_j}\log p(\mathbf{Pc}_{1:t}\mid\boldsymbol{\chi}_{1:t},\boldsymbol{\theta}) &= \frac{1}{2}\textstyle\sum_{k=1}^{M}(\mathbf{P}_{1:t}^k-\mu_0^k)^\intercal\mathbf{K}^{-1}\dfrac{\partial\mathbf{K}}{\partial\theta_j}\mathbf{K}^{-1}(\mathbf{P}_{1:t}^k-\mu_0^k)\\ &\qquad -\frac{1}{2}\mathrm{tr}\left(\mathbf{K}^{-1}\frac{\partial\mathbf{K}}{\partial\theta_j}\right)\\ &= \frac{1}{2}\textstyle\sum_{k=1}^{M}\mathrm{tr}\left((\boldsymbol{\alpha}^k\boldsymbol{\alpha}^{k\intercal})\dfrac{\partial\mathbf{K}}{\partial\theta_j}\right)-\frac{1}{2}\mathrm{tr}\left(\mathbf{K}^{-1}\frac{\partial\mathbf{K}}{\partial\theta_j}\right)\\ &\qquad \text{where}\qquad \boldsymbol{\alpha}^k=\mathbf{K}^{-1}(\mathbf{P}_{1:t}^k-\mu_0^k)\\ &= \frac{1}{2}\mathrm{tr}\left((\mathbf{K}^{-1}\textstyle\sum_{k=1}^{M}\left((\mathbf{P}_{1:t}^k-\mu_0^k)(\mathbf{P}_{1:t}^k-\mu_0^k)^\intercal\right)\mathbf{K}^{-1})\dfrac{\partial\mathbf{K}}{\partial\theta_j}\right)\\ &\qquad -\frac{1}{2}\mathrm{tr}\left(\mathbf{K}^{-1}\frac{\partial\mathbf{K}}{\partial\theta_j}\right)\end{aligned} \tag{5.15}$$

The structure of $\mathbf{Pc}_{1:t}$, which is composed of one column of observations for each channel, allows the expression to be simplified thanks to this equality:

$$\sum_{k=1}^{M}\mathbf{P}_{1:t}^k\mathbf{P}_{1:t}^{k\intercal}=\mathbf{Pc}_{1:t}\mathbf{Pc}_{1:t}^\intercal \tag{5.16}$$

With this simplification, the gradient of the log-likelihood is exactly the same as the original expression of standard GP but with the matrix representation of the observations ($\mathbf{Pc}_{1:t}$, see the previous section):

$$\frac{\partial}{\partial\theta_j}\log p(\mathbf{Pc}_{1:t}\mid\boldsymbol{\chi}_{1:t},\boldsymbol{\theta})=\frac{1}{2}\mathrm{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^\intercal-\mathbf{K}^{-1})\frac{\partial\mathbf{K}}{\partial\theta_j}\right)\ \text{where}\ \boldsymbol{\alpha}=\mathbf{K}^{-1}(\mathbf{Pc}_{1:t}-\boldsymbol{\mu}\boldsymbol{c_0}) \tag{5.17}$$

5.2.5 Experimental Validation

We evaluate the performance improvement provided by the knowledge transfer ability of our algorithm on the second setup of the robotic arm (see appendix B.2). In this experiment, the robot has to sequentially reach *20 targets* defined in the camera's image, by using the same controller as in the previous chapter (see chapter 4 and appendix B.3), which only defines the final angular position of the eight joints. The robot state is defined as the X and Y coordinates of the laser point in the camera's image. The aim of this experiment is to show that transferring the observations acquired when learning to reach one target decreases the number of trials required to reach the subsequent targets.

The relative position of the targets has a strong impact on the learning performances: learning to reach a target close to the previous targets can be particularly easy, while learning to reach a target in an area never explored can be a challenging task. Moreover, the order of the target has an impact too: if the first target is really

difficult to reach and requires intensively exploring the search space, the subsequent targets will greatly benefit from this initial step. In order to have experimental results that are not impacted by these factors, we decided to completely randomize the targets' position and to perform a large number of replicates to gather statistics. Unfortunately, performing a very high number of replicates is materially impossible on the physical robot, thus we run these experiments only in simulation. In the last section of this chapter, we will evaluate our algorithm and its additional features details on the physical robot.
In order to ensure that all the targets are actually reachable by the robot, the targets are randomly generated following this procedure: The targets are first randomly generated in the reachable space of the robot and are defined by two polar coordinates chosen between $\pm 0.9 \times \pi/2$ for the angular position and between 10% and 90% of the robot's total length for the distance. We selected these values in order to remove the borders of the reachable space that may be too difficult to be reached. If these two coordinates were selected through a uniform random distribution, the target locations will not be uniform in the Cartesian space (they will be more concentrated close to the center of the reachable space). For this reason, we used the square root of the random variable used for the radius coordinate. This procedure allows the target to be evenly spread over the reachable space of the robot. These random targets are then multiplied by a transformation matrix to define them in pixels and in the camera's frame (see appendix B.2). The values of the parameters used for these experiments can be found in appendix C.4.

5.2.5.1 Reference Experiments

In addition to the experimental validation of our algorithm, we implemented three reference algorithms:

- Random Search
- Traditional Bayesian Optimization (BO)
- State-based Bayesian Optimization (without knowledge transfer)

These algorithms allow us to highlight the impacts of sub-part of the proposed method on the robot's learning abilities.
The first reference algorithm is a naive random search, where random joint positions are successively generated until reaching the desired target. This experiment aims at showing that this task cannot be solved without a minimum of learning skills. The second reference algorithm is the usual Bayesian Optimization algorithm where the performance function is the distance between the position of the gripper (at the end of the motion) and the target. Because all the techniques presented in this chapter are based on Bayesian Optimization, this experiment will be used as a reference when comparing the performance of the proposed algorithms. The third reference algorithm consists in using the State-Based Bayesian Optimization algorithm, presented in this section, but without knowledge transfer between each

target (the observations are discarded). This experiment will show how modeling the robot's state, instead of the performance function, affects the results. The benefits provided by the knowledge transfer will be highlighted by comparing our approach to all the considered reference experiments that do not transfer knowledge between each target.
We assumed at the beginning of this chapter that the active learning ability of our approach, which makes it focus on regions that are useful for the robot's mission, is likely to improve its data efficiency. We evaluate this aspect with two other reference algorithms:

- A pre-trained Gaussian Process (GP)
- The PILCO algorithm (Deisenroth and Rasmussen, 2011)

The *pre-trained GP* is constructed with 151 samples covering all the motor space and is then used to make the robot reach the targets. The 151 samples, which correspond to the maximum number of samples allowed for the other methods, are generated thanks to a BO algorithm that only considers the standard deviation of the GP in the acquisition function (i.e., pure exploration). This automatically evenly distributes the samples to minimize the uncertainty over all the motor space. Once these 151 samples have been generated and a GP trained, this model is used to select one behavior that is the most likely to reach the current target. In this experiment, the robot can perform only one trial to reach the target, but it relies on a model built with 151 samples. This experiment evaluates the generalization ability of the trained GP.
The PILCO algorithm is a model-based, data-efficient policy search approach, which uses GPs to actively build a forward model of the robot (typically, the dynamic model of the robot). The algorithm updates its policy via a gradient-based optimization based on the predictions provided by the model. The policy is then tested on the robot and the resulting observations are used to improve the model. This process is then repeated until the robot reaches the desired state.
PILCO is designed to be a "step-based algorithm" (see section 2.3), whereas in this thesis we consider tasks for which we can only evaluate the reward (and possibly the state) at the end of each episode. Assessing the robot's reward or state at the end of each trial allows us to use inexpensive tracking systems (for instance, the simple tracking used in the present experiments, see appendix B.2) and/or on-board sensing. In addition, in experiments with the hexapod robot, the complete state would need to be described by many variables (at least 42 if we only consider the position and velocity of the six legs and the main body), which makes it hard to learn. In order to compare directly PILCO with our approach, we configured it to be an "episode-based algorithm" by setting the prediction horizon to 1 step[2].

[2]It is very likely that PILCO would requires fewer trials (episodes) to reach the goal if it could rely on the data of the intermediate states of each episode, typically at each time step. However, this would not respect the constraints of our experimental setup and it is likely that the performance of Intelligent Trial and Error algorithm would also be improved if each trial provides more data.

With this modification, PILCO and our approach are relatively similar: 1) they both build a model that predicts the robot's final state according to the commands (or the corresponding parameters) that are executed; 2) they both select the next trial based on a reward function that combines the predictions of the model and the goal; 3) each trial on the robot generates data that is used to improve the model. However, they differ in the way the policy is updated. While our approach updates the policy parameters by maximizing the acquisition function, with a global optimization approach (CMAES, with different starting points and several restarts), PILCO maximizes the expected reward (which is equivalent to our performance function) based on the gradients provided by the reward function, the model and the controller. This gradient-descent starts from the current policy and makes the policy update thus more "local" than ours. The implementation of this reference algorithm is based on the MATLAB-code provided by Deisenroth et al. (2013a). The algorithm is applied on the same (virtual) experimental setup as the other algorithms, and the parameters are set to their default value. The employed policy type is based on the linear controller, but with no input (i.e., by only using the controller's bias), which is equivalent to the open-loop controllers used in the other experiments.

5.2.5.2 Results

The algorithms consider that the target has been reached when the laser point (from the laser pointer attached on the gripper, see appendix B.3) is within a radius of 50 pixels, which roughly corresponds to 3.3 centimeters. The maximum number of attempts to reach one target is limited to 151. Each experiment is replicated 200 times.

We analyze how many iterations are required to reach one (random) target (see Fig. 5.2 A). The random search algorithm is able to reach the target in only 43% of the replicates and 75% of these replicates require more than 40 trials to reach the target (unsuccessful replicates are stopped after 151 trials). The traditional BO and the state-based BO experiments perform much better and are not statistically different (p-value=0.75), the classic Bayesian algorithm is successful in 96.5% of the replicates and the State-Based Bayesian Optimization is successful in 99% of them (see Fig: 5.3). The median number of required trials for these two reference experiments is respectively 20 and 20.5 trials.

This first comparison shows that the modifications we made to model the robot state function instead of the performance function has no effects on the learning performance of the algorithm. We can also hypothesize that the small success rate improvement is due to the additional information provided by modeling the robot state as presented in the previous section (section 5.2.3).

After this first target, the algorithms have to reach subsequently 19 additional targets (for a total of 20 targets). We consider the cumulative number of trials required to reach every target because the number of trials to reach one specific target is clearly influenced by the number of trials, the location and the difficulty

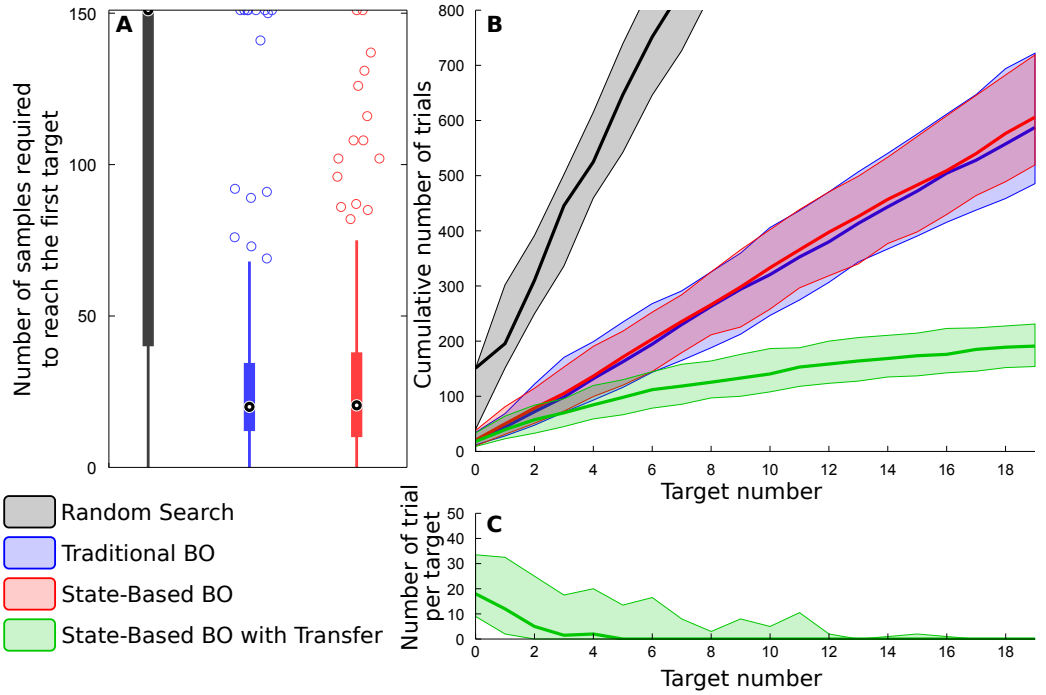


Figure 5.2: The benefits of transferring knowledge. (A) Number of trials required to reach the first target. The results of State-based BO with transfer (that are not shown in this pannel) are similar to the State-based BO because we consider here only the first target. (B) Cumulative number of trials required to reach subsequently each of the 20 targets. (C) Number of trials required to reach each target when using knowledge transfer. The decrease of the number of trials shown here is due to the knowledge transfer between each task. The middle, solid lines represent medians, while the colored areas extend to the 25th and 75th percentiles. The experiment has been replicated 200 times in simulation for each of the 4 tested algorithms.

of the previous targets. For example, reaching a new target can require a large number of trials if the algorithm performed few trials before and it can require very few trials if the search space has been extensively explored before trying to reach this new target. Recording the cumulative number of trials and performing multiple replicates of the experiments reduce these biases in the presented results. For each new target, the number of required trials contains both the number of trials used to reach this new target but also the number of trials required to reach all the previous targets (see Fig 5.2 B).

Before actually starting the learning process, 10 random trials are performed in order to initialize the algorithm's model. This kind of procedure is customary when algorithms start from scratch (Lizotte et al., 2007; Calandra et al., 2014; Cully et al., 2015). However, in the number of trials reported in this chapter, we only consider trials that have been intentionally chosen and the 10 initial random trials are not taken into account. Indeed, there is no certainty that these samples have participated in the success of the algorithm and counting them may artificially penalize the learning speed of the reference algorithms. This is why, we chose to ignore these samples in our comparison to prove that the performance differences between our algorithm and the reference ones are not induced by this artificial initialization procedure.

The reference experiments do not transfer knowledge between each target, for this reason the cumulative number of trials required to reach the targets is a linear progression with the slope corresponding to the number of trials to reach the first target (see Fig. 5.2 A). Like for the first target, the classic Bayesian Optimization algorithm and the State-based Bayesian Optimization algorithm perform very similarly. When knowledge transfer is enabled, the State-based Bayesian Optimization clearly shows a large improvement of its learning abilities. While about 600 trials (median: 606 trials for State-based BO and 587.5 trials for BO) are required to reach the 20 targets without transfer, this number falls to a median of 191 trials with the State-based Bayesian Optimization with Knowledge Transfer. For the last targets, the additional number of trials to reach new targets becomes very low: more than 75% of the 200 replicates reach the targets in one shot between the 12^{th} and 20^{th} targets (Fig. 5.2 C). The remaining 25% are likely to require more trials because of the reasons exposed before (e.g., dependency to the prior experiences, relative position of the targets).

The conclusion that can be drawn from these experimental results is that the modification of standard Gaussian Processes that allows the algorithm to model the multi-dimensional (or channel) robot state function has little impact on the optimization performance. Moreover, this modification allows the algorithm to transfer knowledge from one task to another and this significantly (p-value$< 10^{-66}$, Wilcoxon rank sum test) reduces the time needed to achieve next tasks.

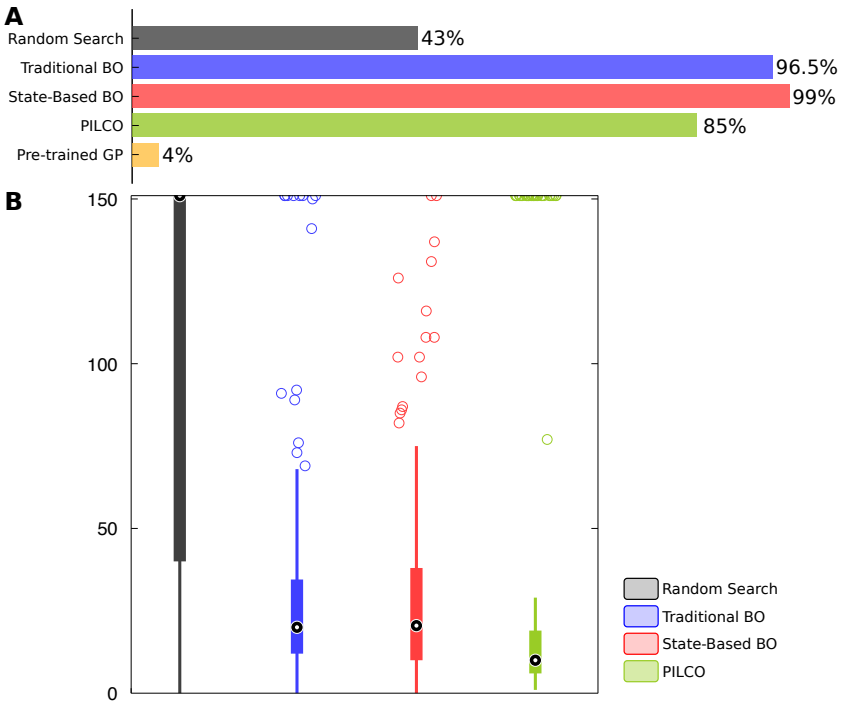


Figure 5.3: Reaching the first target. A) Success rate for reaching the first target with the five compared algorithms. B) Comparison of the number of trials required to reach the first target. The experiment has been replicated 200 times in simulation for each of the tested algorithms.

5.2.5.3 Comparison with PILCO and with a pre-trained GP

In the control experiment using a pre-trained GP, the robot is allowed to perform only one trial to reach the target, but it relies on a model built with 151 samples. Like in the previous experiments, this experiment has been replicated 200 times with the same random targets as before. The results (see Fig: 5.3A) show a success rate of only 4%, which illustrates how the active learning ability of our approach significantly improves its performances. It may appear surprising that the random search performs better than this "pre-trained GP" alternative. The reason is that only the single trial of this alternative approach is taken into account. This means that even if some of the 151 trials performed to build the model reached the targets, the algorithm has to select again a point that will reach the target, which appears to be difficult. One explanation of this difficulty can be the inaccuracies of the model that appear when the model tries to cover the entire search space. This aspect is highlighted by the success rate of the other tested algorithms, which select their next trial in an active fashion (i.e., based on the previous trials and the observed performance).

With active learning approaches, the success rate is significantly higher: 85% for PILCO, 96.5% for the traditional BO, and 99% for the State-based BO. All these differences are statically significant with the p-values$< 10^{-4}$ (based on the Fisher's exact test) (except between traditional BO and State-based BO, p-value=0.1747), and show that using active learning approaches, which focus on regions that are useful for the robot's mission, is more effective than building global models.

While our approach has a better success rate than PILCO, we can see that PILCO requires about two times less trials than the other approaches to reach the first target (see Fig: 5.3B, median: 10 trials for PILCO versus 20.5 trials for State-based BO, p-value= 10^{-5}). We can hypothesize that this performance difference comes from the way that the policies are updated, since the generation of the model is very similar in these two approaches. The policy update process of PILCO, which is more "local" than our approach, is likely an advantage in the experimental setup that we consider in this chapter because it is relatively smooth and convex and thus suitable for gradient-based approaches (e.g., classic visual servoing techniques can be used to solve the task, Espiau et al. (1992)). Moreover, the high redundancy of the robot leads to not only one solution, but rather an hyper-plan of solutions (i.e., several different configurations lead the robot to the desired location). This variety of solutions does not affect PILCO, as it will reach the closest configuration according to its starting policy, but does affect our approach because the global exploration of the acquisition function (here, UCB) tends to simultaneously investigate all the possible alternatives, which is likely to slow the process down. However, this advantage in this particular setup may be a disadvantage in other ones, typically those with local extrema. Finally, we can also hypothesize that the relatively low success-rate of PILCO compared to State-based BO is due to unlucky initializations of the GP that may make PILCO start too far from a good solution. It would be interesting to perform the same kind of comparison on more challenging

experimental setups to see how the differences between these two approaches are affected.

5.3 Missing Data

5.3.1 Motivations

When performing experiments, some behaviors cannot be properly evaluated on the physical robot. This can happen for different reasons: (1) The robot may be outside the sensor's range, for example when the robot is not visible from the camera's point of view, making it impossible to assess its performance. (2) The sensor may return intractable values (infinity, NaN,...). A typical situation where this kind of illogical values may occur is when the SLAM algorithm diverges and is unable to infer the robot's position. (3) The behavior may fail a sanity check, which is executed before running the behavior on the robot and prevent the behavior from damaging the robot. A classic sanity check can be to check if a behavior is likely to produce self-collisions when it's applied on a robotic arm. For example, this check can be performed in simulation before testing the behavior in reality. All these situations generate missing data about the behaviors' performance and this represents a significant problem that may impact the algorithms' results.
While it is very likely that most of experiments are affected by these situations, the way the authors deal with them are unfortunately rarely specified. Moreover, making the hypothesis that it is always possible to evaluate a behavior is a strong limitation for the robot's autonomy and in this case should be clearly mentioned in research articles.

5.3.2 Principle

Different solutions exist to deal with missing data. The simplest way consists in redoing the evaluation. This may work, but only if the problem is not deterministic, otherwise the algorithm will be continuously redoing the same, not working, evaluation. A second solution consists in assigning a very low value to the behavior's performance, like a punishment. This approach will work with evolutionary algorithms because the corresponding individual will very likely be removed from the population in the next generation. By contrast, this approach will have a dramatic effect on algorithms using models of the reward function, like Bayesian Optimization, as the models will be completely distorted. Figure 5.4 B shows how setting a missing data to zero affects the estimated reward function in a Bayesian optimization process. With these model based algorithms, another solution may be to set the missing data to the value previously predicted by the model, but the drawback of this approach is that it will attach the model to a potentially wrong value (see Fig. 5.4 C).
These different methods to deal with missing data do not fit well with the Bayesian Optimization framework. For this reason we propose in this section a new approach,

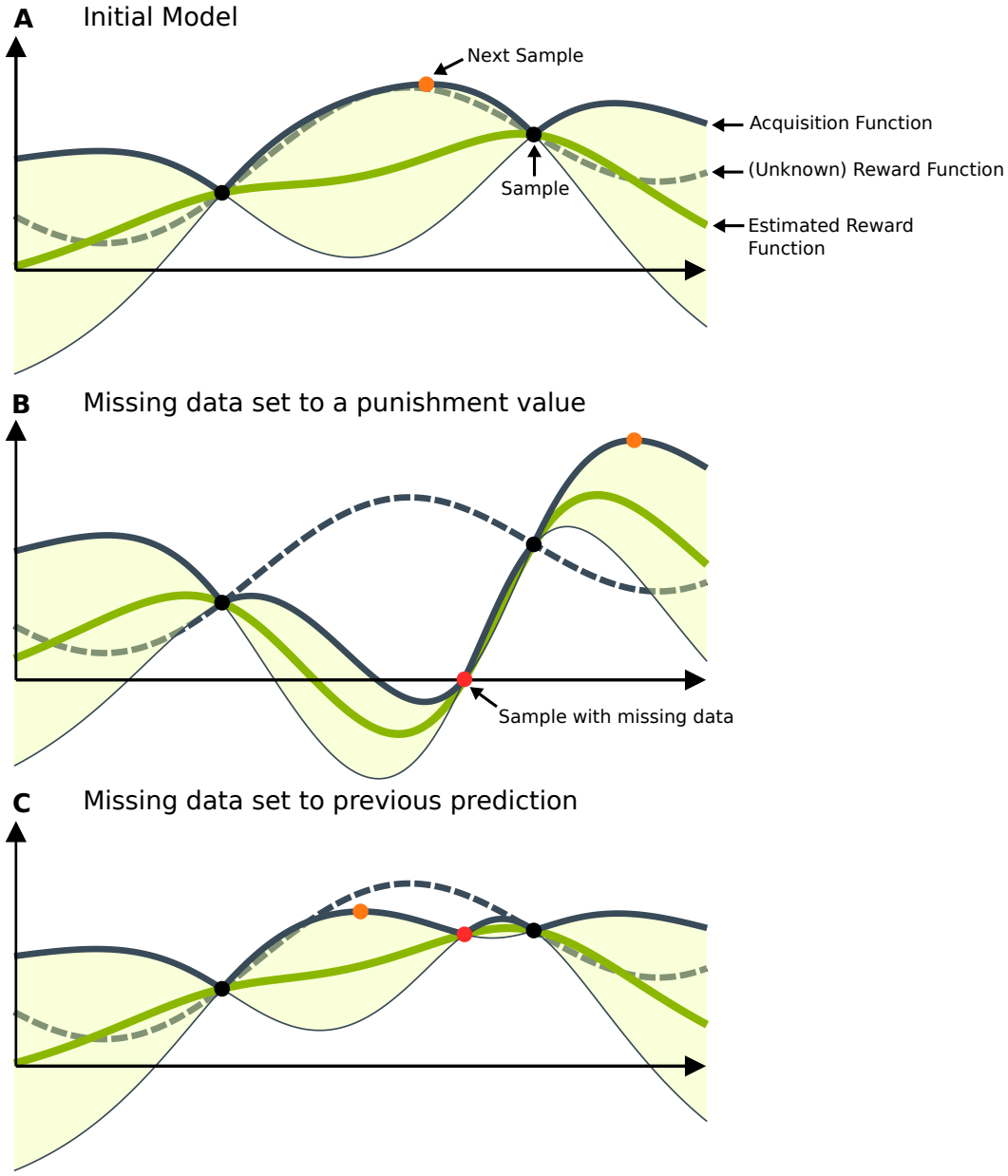


Figure 5.4: Common ways to deals with solutions that cannot be evaluated. (A) Initial model, the performance of the "next sample" cannot be assessed. The two other sub-panels show the impact of commonly used techniques to deal with missing data. (B) Samples with missing data receive a punishment value (here 0), distorting the model as a consequence. (C) Samples with missing data receive the value previously predicted by the model, fixing it to a potentially wrong value.

compatible with Bayesian Optimization, which preserves the model's stability. The overall idea is to encourage the algorithm to avoid regions around behaviors that could not be evaluated, which may contain other behaviors that are not evaluable too, but without providing any performance value, which is likely to increase the model's instability.
The behaviors' attractiveness is defined based on the acquisition function (see section 2.4.3), which is mainly governed by the expected performance and variance around this prediction, which represents how much the search space has been explored in the behavior's vicinity. As shown in the previous section, most of model instabilities come from attempts to change the expected performance. Consequently, our idea consists in providing to the algorithm the information that a behavior has already been tried, in order to reduce its variance, but without fixing the expected performance to a hazardous value.

5.3.3 Method Description

In order to provide the information that some behaviors have already been tried, we propose to define a blacklist of samples. Each time a behavior cannot be properly evaluated, this behavior is added into the blacklist (and not in the pool of tested behaviors). Because the performance value is not available, only the behavior's location in the search space is added to the blacklist. In other words, the blacklists are a list of samples with missing performance data.
Thanks to this distinction between valid samples and blacklisted ones, the algorithm can consider only the valid samples when computing the mean of the Gaussian Process and both valid and blacklisted samples when computing the variance. By ignoring blacklisted samples, the mean will remain unchanged (Fig. 5.5 B) and free to move according to future observations (Fig. 5.5 C). By contrast, the variance will consider both valid and blacklisted samples and will "mark" them as already explored (Fig. 5.5).
The mathematical formulation of this idea is relatively straightforward. The mean equation, $\mu_t(\mathbf{x})$, remains unchanged, while the kernel of the variance equation, $\sigma_t^2(\mathbf{x})$, is extended to the blacklisted samples ($\mathbf{bl}$). Note that, while the mathematical description is given based on the classic Bayesian Optimization algorithm for reading convenience, this idea is fully compatible with the previously described multi-channel Bayesian Optimization algorithm and the Intelligent Trial and Error algorithm.

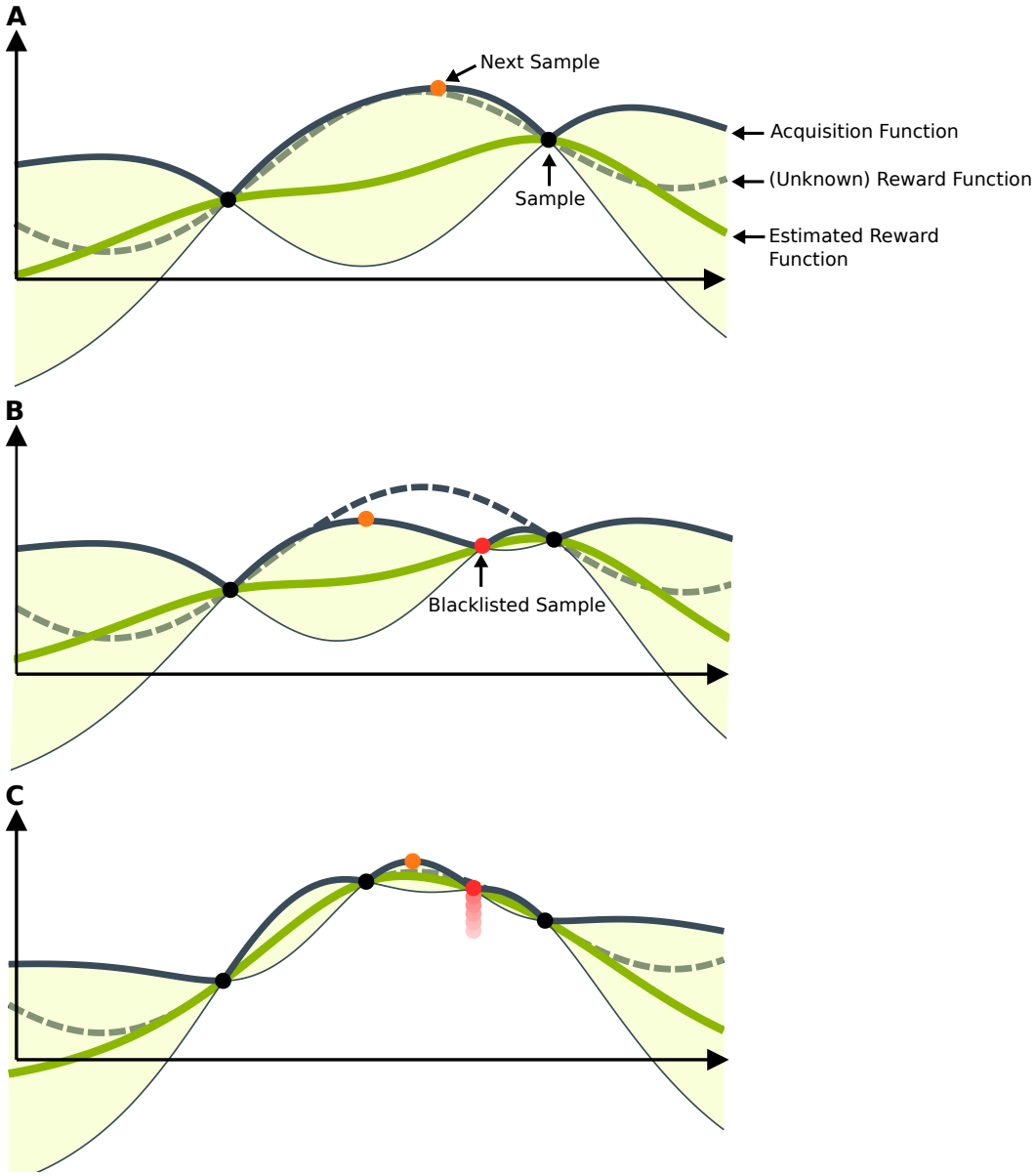


Figure 5.5: Blacklist principles. (A) Initial model (the same as Fig. 5.4). (B) The variance around the blacklisted sample is reduced but the mean of the model remains unchanged. (C) After testing another solution, the model can adapt to the new observation because the performance value predicted for the blacklisted sample is not fixed.

$$P(f(\mathbf{x})|\mathbf{P}_{1:t}, \mathbf{x}) = \mathcal{N}(\mu_t(\mathbf{x}), \sigma_t^2(\mathbf{x}))$$

where :

$\mu_t(\mathbf{x}) = \mu_\mathbf{0} + \mathbf{k}^\intercal \mathbf{K}^{-1}(\mathbf{P}_{1:t} - \mu_\mathbf{0})$

$\sigma_t^2(\mathbf{x}) = k(\mathbf{x},\mathbf{x}) - \mathbf{k_{bl}}^\intercal \mathbf{K_{bl}}^{-1} \mathbf{k_{bl}}$

$$\mathbf{K_{bl}} = \begin{bmatrix} & & & k(\boldsymbol{\chi}_1, \mathbf{bl}_1) & \cdots & k(\boldsymbol{\chi}_1, \mathbf{bl}_n) \\ & \mathbf{K} & & \vdots & \ddots & \vdots \\ & & & k(\boldsymbol{\chi}_t, \mathbf{bl}_1) & \cdots & k(\boldsymbol{\chi}_t, \mathbf{bl}_n) \\ k(\mathbf{bl}_1, \boldsymbol{\chi}_1) & \cdots & k(\mathbf{bl}_1, \boldsymbol{\chi}_t) & k(\mathbf{bl}_1, \mathbf{bl}_1) + \sigma^2_{noise} & \cdots & k(\mathbf{bl}_1, \mathbf{bl}_n) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{bl}_n, \boldsymbol{\chi}_1) & \cdots & k(\mathbf{bl}_n, \boldsymbol{\chi}_t) & k(\mathbf{bl}_n, \mathbf{bl}_1) & \cdots & k(\mathbf{bl}_n, \mathbf{bl}_n) + \sigma^2_{noise} \end{bmatrix}$$

$$\mathbf{k_{bl}} = \begin{bmatrix} k(\mathbf{x}, \boldsymbol{\chi}_1) & \cdots & k(\mathbf{x}, \boldsymbol{\chi}_t) & k(\mathbf{x}, \mathbf{bl}_1) & \cdots & k(\mathbf{x}, \mathbf{bl}_n) \end{bmatrix}$$

(5.18)

This representation uses the same definitions as in equation 5.19 in section 2.4 and changes are depicted in blue to highlight how to add the blacklist of samples.
The extension of the kernel matrix increases the computational cost of this method, as this second matrix needs to be inverted too and is larger than matrix used for the mean. However, this addition cost can be significantly mitigated by using the block-wise inversion theorem (Press et al., 1996):

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \\ -(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{C}\mathbf{A}^{-1} & (\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \end{bmatrix}$$

(5.19)

This theorem says if we known $\mathbf{A}^{-1}$ then computing the inverse of the global matrix only requires computing the inverse of this expression $(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})$ (depicted in blue in the previous equation). In our case this is very useful, as the $\mathbf{A}^{-1}$ is the inverse of the mean's kernel matrix (thus already inverted) and the size of $(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})$ is equal to the number of black listed samples, which is likely to be notably lower than the number of correct samples. Consequently, thanks to this theorem, computing the variance of the GP with blacklist only additionally requires to inverse a small matrix and to perform a few matrix products.

5.3.4 Experimental Validation

We replicated the experiment from the previous section (section 5.2.5) but in this section each sample evaluation has a 5% chance to fail, generating missing data. This experiment is also replicated 200 times for each compared algorithms.
We first evaluated the impact of these missing data on the learning algorithms when they are handled with "punishment values". In this experiment, the punishment values are set to -1500 pixels (corresponding roughly to -1 meter) for the reward based algorithm and $[-1500; -1500]$ for the stated based algorithms.
We take as a reference point the total number of trials required to reach the 20 targets when there are no missing data (i.e., the 5% chance that it occurs is dis-

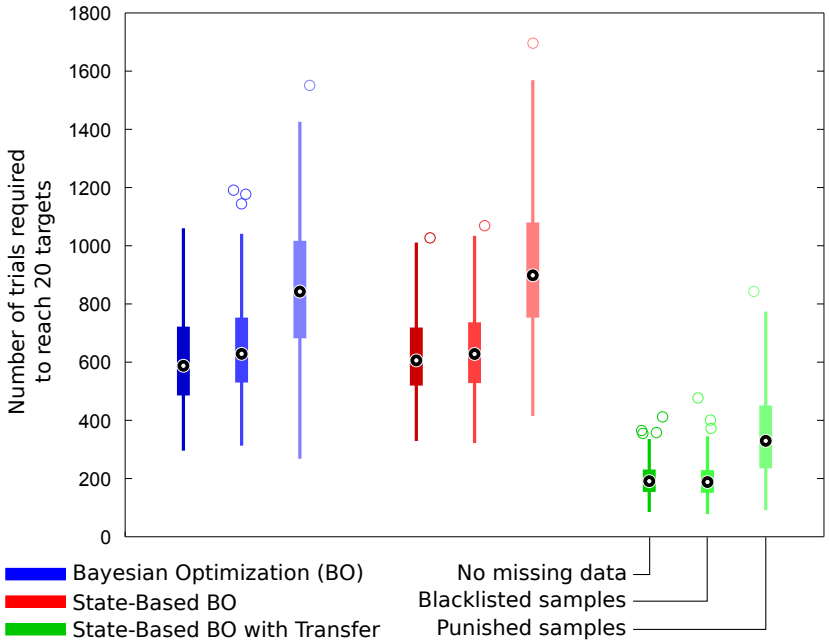


Figure 5.6: Impact of missing data on the learning performance. For the three tested algorithms we compared the total number of trials required to reach 20 targets. Each trial has a 5% chance to fail and to generate missing data. For each algorithm, the number of trials for the experiments without missing data is plotted for reference and the approach using blacklists of samples or punishment values are compared. The experiment has been replicated 200 times in simulation for each of the 3 tested algorithms and their 3 variants.

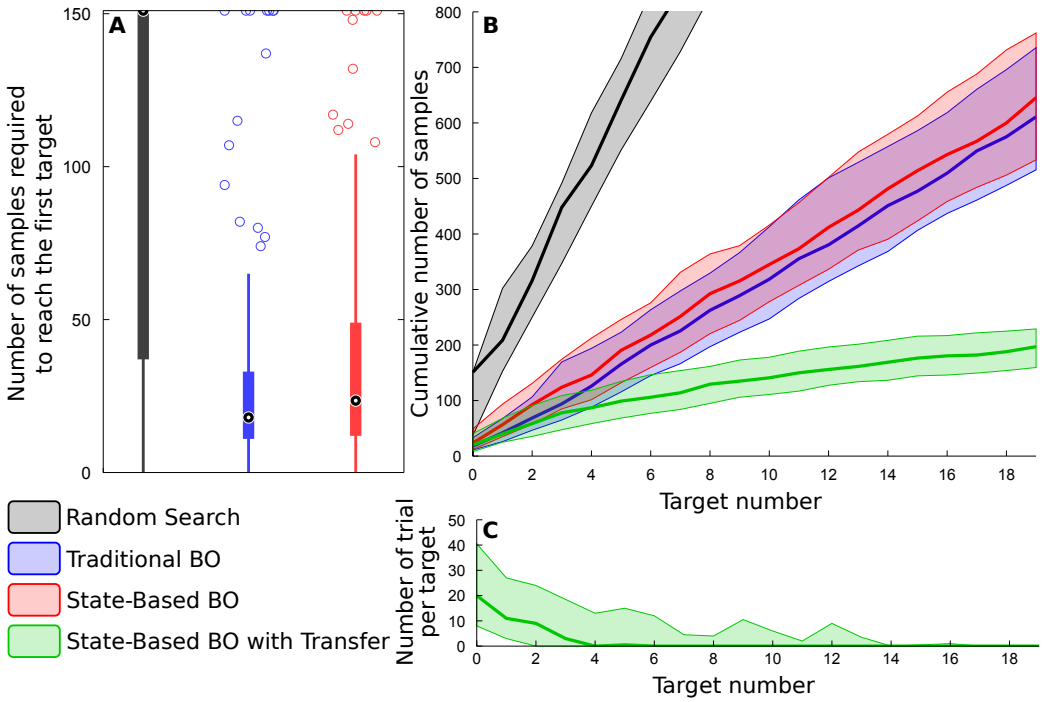


Figure 5.7: Blacklists of samples do not affect the learning performance. (A) Number of trials required to reach the first target. (B) Cumulative number of trials required to reach subsequently 20 targets. (C) Number of trials required to reach each target. The decrease shown here is due to the knowledge transfer between each task. The middle, solid lines represent medians, while the colored areas extend to the 25th and 75th percentiles. Both experiments have been replicated 200 times in simulation for each of the 5 tested algorithms.

abled). Theoretically, this number should increase by about 5% if the missing data are well managed. The results show, as expected, that the punishment values have a important impact on the number of trials (Fig. 5.6). For all the tested algorithms (standard Bayesian Optimization (BO), state-based BO, and state-based BO with knowledge transfer), the number of trials increases by about 50% (BO: 43.3%, state-based BO 48.3%, state-based BO with knowledge transfer 72.3%, all these differences are statistically significant: p-values$< 10^{-28}$) while when the missing data are handled with blacklists this number of trials only increases by about 3% (BO: 6.9%, state-based BO 3.6%, state-based BO with knowledge transfer -1.6%, the two first increases are statistically significant, p-values <0.02 but not the last one, p-values=0.35).

This experiment demonstrates that using blacklists of samples to deal with missing data minimizes their impact on the learning performances. When comparing the results of algorithms using blacklists (Fig. 5.7) and those without missing data (Fig. 5.2), the differences are very subtle. For example, thanks to the knowledge transfer, our algorithm is able to reach the 20 targets in less than 200 trials (median:

197 trials, 5^{th} and 95^{th} percentiles: [123; 316] trials when there are some missing data, versus 191 [107.5; 324.5] trials when there is no missing data). Moreover, the ability to reach in one shot the targets after some previous experiments is preserved. In this case, after the 14^{th} target, more than 75% of the replicates reach the targets in one shot. The only difference may be the performance difference between the standard BO and the State-based BO, which seems to be larger than in the previous experiment. Nevertheless, this difference is not statically significant (when computing the p-value point by point for all the 20 targets, the p-value varies between 0.015 and 0.95).
Solutions that cannot be evaluated and which generate missing data is a problem that impacts most of experiments using real robots and represents one of the numerous problems that prevent learning algorithms from being widely applied on physical setups. Throughout these experiments, we showed that using blacklists of samples is a computationally efficient and easy way to setup way to cope with this issue. While the experiments presented in this section are only performed in simulation in order to perform a large number of replicates, the last section of this chapter presents an experiment in which the whole framework of our algorithm, including the use of blacklists but also additional features presented in the next section, are evaluated on the physical robot in order to show its benefits in a real context.

5.4 Misleading Priors

5.4.1 Motivations

As we showed in chapter 4, using priors is one of the key components to achieve fast adaptation. On the one hand, if well chosen, these priors can guide the search and provide various information about the search space and/or reduce the search space. On the other hand, if not well chosen, these priors may be misleading and hurt the learning performances. For example, the forward kinematic model of the robot arm is typically a knowledge that can easily be given a priori to the robot, but which may be misleading for several reasons: (1) they may be not well aligned with the actual search space, like when forward model is defined based on the robot's frame while observations are computed based on the camera's frame. (2) Another reason is when units differ: if the forward model is given in meters while the observations are in pixels, the unknown conversion between these two sources of knowledge prevent the algorithm from relying on the forward model. (3) The worst situation is when the priors are strictly uncorrelated with the task, for example if we provide to the robot arm priors about the hexapod robot. (4) When the robot is damaged, the priors may also be misleading, as they do not take into account the current situation of the robot. All these examples show how potentially useful prior knowledge can become more misleading than helpful.

5.4.2 Principle

In this section, we present a method that allows the algorithm to autonomously adapt these priors to exploit all their potential usefulness or in the worst situation to disregard them. In most of the examples exposed in the previous paragraph, priors contain meaningful information, which cannot be used to solve the task because of conversion or alignment problems.
To address these problems, our idea consists in transforming the priors' information to let them match with the acquired observations. One of the most common ways to transform data is to use linear transformations or, when the data are multi-dimensional, transformation matrices. Moreover, the information provided by the priors can be extended to homogeneous coordinates (or projective coordinates) in order to implement translations thanks to usual transformation matrices. The algorithm will autonomously determine the adequate transformation according to the observations by maximizing the likelihood of the Gaussian Process (see section 2.4.2.2). This method will allow the robot to maximize the benefits provided by the prior information or in the worst cases to mitigate their negative impacts.

5.4.3 Method Description

State-based Bayesian Optimization (see section 5.2.2) allows us to model functions that have multiple output channels. A consequence of this multi-dimensionality is that the prior knowledge should also have outputs with multiple channels. These multi-channels priors can easily be extended to homogeneous coordinate by increasing the number of output dimensions by one and by fixing the value of this additional dimension to 1. With this extension, usual transformation matrices will be able to scale, rotate, translate and even disregard (with a null matrix) the information provided by the priors. These matrices (here named $\mathbf{T_r}$) can be added directly in the traditional GP formulation in which the prior ($\boldsymbol{\mu}(\mathbf{x})$) is a vector:

$$P(f(\mathbf{x})|\mathbf{P}_{1:t}, \mathbf{x}) = \mathcal{N}(\mu_t(\mathbf{x}), \sigma_t^2(\mathbf{x}))$$

where :

$$\begin{aligned}
&\mu_t(\mathbf{x}) = \mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\mathbf{x}) \\ 1 \end{bmatrix} + \mathbf{k}^\intercal \mathbf{K}^{-1} \left(\mathbf{P}_{1:t} - \left(\mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) \\ 1 \end{bmatrix} \right)^\intercal \right) \\
&\sigma_t^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1} \mathbf{k} \\
&\boldsymbol{\mu}_0(\mathbf{x}) = \begin{bmatrix} \mu_0^1(\mathbf{x}) \\ \vdots \\ \mu_0^n(\mathbf{x}) \end{bmatrix} \\
&\boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) = \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_1) & \dots & \boldsymbol{\mu}_0(\boldsymbol{\chi}_t) \end{bmatrix} \\
&\mathbf{T_r} = \begin{bmatrix} \theta_{Tr}^1 & \theta_{Tr}^2 & \dots & \theta_{Tr}^{n+1} \end{bmatrix}
\end{aligned} \tag{5.20}$$

While the presented equation only considers the modifications that should be added to the original GP framework, this technique is fully compatible with all those presented in this chapter and can be combined into a generic framework that allows

robots to autonomously transfer knowledge between tasks, deal with missing data and to adapt the provided priors. For example, this expression can be extended to state-based BO:

$$P(f(\mathbf{x})|\mathbf{Pc}_{1:t}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu_t}(\mathbf{x}), \sigma_t^2(\mathbf{x})\mathbf{I})$$

where :

$$\begin{aligned}
\boldsymbol{\mu_t}(\mathbf{x}) &= \mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\mathbf{x}) \\ 1 \end{bmatrix} + \left(\mathbf{Pc}_{1:t}^\intercal - \mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) \\ 1 \end{bmatrix} \right) \mathbf{K}^{-1}\mathbf{k} \\
\sigma_t^2(\mathbf{x}) &= k(\mathbf{x}, \mathbf{x}) - \mathbf{k}^\intercal \mathbf{K}^{-1} \mathbf{k} \\
\boldsymbol{\mu}_0(\mathbf{x}) &= \begin{bmatrix} \mu_0^1(\mathbf{x}) \\ \vdots \\ \mu_0^n(\mathbf{x}) \end{bmatrix} \\
\boldsymbol{\mu_0}(\boldsymbol{\chi}_{1:t}) &= \begin{bmatrix} \boldsymbol{\mu_0}(\boldsymbol{\chi}_1) & \dots & \boldsymbol{\mu_0}(\boldsymbol{\chi}_t) \end{bmatrix} \\
\mathbf{T_r} &= \begin{bmatrix} \theta_{Tr}^{1,1} & \theta_{Tr}^{1,2} & \dots & \theta_{Tr}^{1,n+1} \\ \vdots & & \ddots & \vdots \\ \theta_{Tr}^{p,1} & \theta_{Tr}^{p,2} & \dots & \theta_{Tr}^{p,n+1} \end{bmatrix}
\end{aligned} \tag{5.21}$$

The transformation matrix's coefficients are considered as additional hyper-parameters of the model and automatically tuned, online, by maximizing the likelihood of the model according to the observations. The likelihood expression remains unchanged (see equation 5.14) and its derivative according the weights of the transformation matrix (θ_{Tr}) is defined as:

$$\frac{\partial}{\partial \theta_{Tr}} \log p(\mathbf{P}_{1:t} \mid \boldsymbol{\chi}_{1:t}, \boldsymbol{\theta}) = \mathrm{tr}\left((\mathbf{Pc}_{1:t} - \boldsymbol{\mu_0})^\intercal \mathbf{K}^{-1} \frac{\partial}{\partial \theta_{Tr}} \left(\mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) \\ 1 \end{bmatrix} \right) \right) \tag{5.22}$$

The derivative in this equation can be easily computed, as it involves only a linear combination of the matrix coefficient and the mean vector. This leads to a simple expression of the gradient if we consider that vector θ_{Tr} contains all the coefficients of the matrix sorted line by line.

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}_{Tr}} \left(\mathbf{T_r} \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}) \\ 1 \end{bmatrix} \right) = \begin{bmatrix} \mathbf{M}(\boldsymbol{\chi})^\intercal & 0 & \dots & 0 \\ 0 & \mathbf{M}(\boldsymbol{\chi})^\intercal & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \mathbf{M}(\boldsymbol{\chi})^\intercal \end{bmatrix} \tag{5.23}$$

where :

$$\mathbf{M}(\boldsymbol{\chi}) = \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}) \\ 1 \end{bmatrix}$$

The ability to adapt the priors and thus the model to the observations has several interesting properties. First, the transformation matrix will adapt globally the prior over all the search space (e.g. it will find the right alignment and the right scale factor) and the Gaussian Process will then correct itself locally to take into account

local phenomenon or small inaccuracies (e.g. damage situations or inaccuracies between the priors and the reality). Second, the algorithm has the possibility to ignore the provided priors by setting all the weights of the transformation matrix to zero or to very low values. This will be done autonomously when the likelihood optimization will suggest that a model with a null matrix is the best way to match the acquired observations. This is an interesting feature as it allows us to provide several sources of prior knowledge and let the algorithm decide which one, or which combination of these priors is actually the most helpful.
This last property share some similarities with what Rasmussen and Williams (2006) (Chapter 2.7) describe in their book. Based on the works from O'Hagan and Kingman (1978) and from Blight and Ott (1975), the authors show how to use the likelihood optimization to determine a vector that is involved in a scalar product with another vector, which is composed by the outcomes of several mean functions. Compared to this approach, the technique presented in this section takes advantage of the multi-channel property introduced previously in this chapter in order to use transformation matrices and homogeneous coordinates instead of a scalar product. Moreover, this section aims to show examples where these techniques can be practically useful.

5.4.4 Experimental Validation

We continue to use the same robotic arm experimental setup (in simulation) to evaluate this additional extension of our algorithm. For this experiment, the robot has access to its forward model as a prior knowledge. Concretely, the forward model is a function that takes as argument the joints' position of the robot and outputs the corresponding position of the gripper projected on the ground. Because the robot has 8 degrees of freedom, the forward model is defined as $f : \mathbb{R}^8 \rightarrow \mathbb{R}^2$. In this case, the prior does not reduce the problem dimensionality. The forward model outputs the position in meter and according to the robot's base frame, while the observations and the targets are computed in pixel defined in the camera's base frame. These differences of reference frame and scale factor make it difficult to exploit the information provided by the priors and they may be, in the worst situations, more misleading than helpful (see the following result section).
The algorithm adapts its prior thanks to a 2 x 3 transformation matrix included in the model as presented before. This matrix has two lines because the robot state function has two channel (the X and the Y coordinates of the gripper) and three columns in order to perform rotations and translations of the coordinates that are outputted by the forward model. These outputs are extended by one additional dimension to turn these coordinates into homogeneous coordinates. There are consequently 6 hyper-parameters that have to be optimized by the likelihood optimization (in addition to the other hyper-parameters for the kernel function, see section 2.4.2.2). All the hyper-parameters of the model are determined after the initialization procedure and re-computed after each trial in order to take into account the new data.

In simulation, the virtual robot and its forward model match perfectly and this makes the task particularly easy. In this case, the algorithm only needs to infer the right transformation matrix thanks to a few samples and then it can rely on the forward model to reach every target in "one shot". The results show that this is successfully achieved only based on the 10 random trials perform at the beginning of the experiment to initialize the model (see the following results section). To increase the task difficulty, we consider situations in which the robot is damaged to show the ability of the algorithm to globally adapt the priors but also to locally adapt its model according to the damage. In addition to the situation in which the robot is intact, four damage conditions are tested in simulation (see Fig. 5.9 C). In the considered situations, the robot's joint can be affected in two ways: (1) the joint can be stuck at a particular position (here, 45°) and does not respond to any received command; (2) the joint has a permanent 45° offset, meaning that the position reached by the motor will always be displaced by 45° according to the received command (in this case, the motor is still able to move). We apply these two types of damage on two different joint locations: (1) close to the base and (2) close to the gripper. The location of the altered joint may influence the difficulty of the task.
In this experiment, all the features detailed previously are combined to allow the robot to transfer knowledge, face missing data and adapt its prior knowledge. Like in the previous experiment of this chapter, the experiments are replicated 200 times (for each tested algorithms), each trial has a 5% chance of generating missing data (and being blacklisted). The target is considered as being reached when the distance between gripper and the target is lower than 50 pixels. In total, the robot will have to reach subsequently 20 targets in order to show that the knowledge transfer abilities remains preserved after these additional changes of the algorithm. The values of the parameters used for these experiments can be found in appendix C.4.

5.4.4.1 Results

In a first experiment, we investigated the benefits of adapting the priors. We considered 3 variants of the State-based BO algorithm: (1) without priors, (2) with raw, not adapted priors (using the "raw" information provided by the forward model) and (3) with adapted priors (the information from the forward model is autonomously adapted thanks to transformation matrix in the model). The influence of these different priors is characterized by the number of samples (or trials) required to reach the first target. In this experiment the notion of knowledge transfer is not taken into account because we are considering only one target. Consequently, the compared algorithms will show the same results as their variant using knowledge transfer.
Contrary to our expectations, the raw prior does not hurt the learning performances of our algorithm (see Fig. 5.8). It is even the opposite, as the median number of trials required to reach the target is lower with the raw priors than with no priors at all (14 trials with raw priors versus 21 trials without priors), but this difference is not

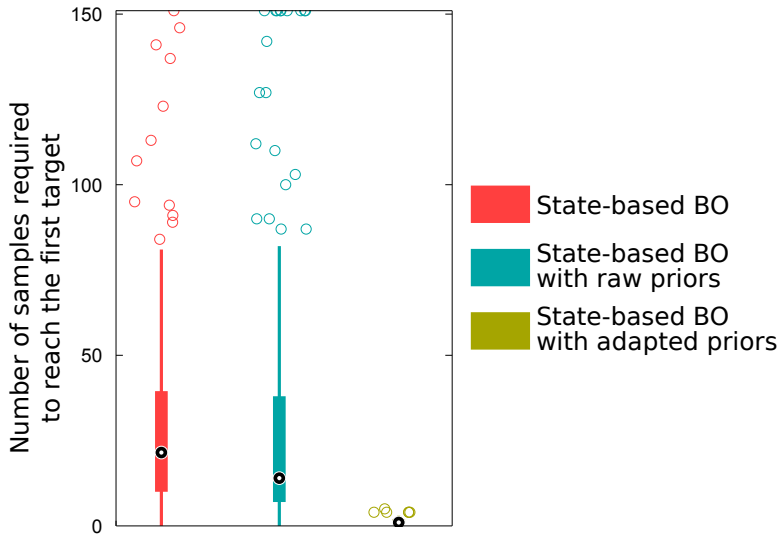


Figure 5.8: The benefits of autonomously adapting the prior knowledge. The number of trials required to reach one target is depicted according to the use of priors that may be misleading. When the algorithm has the ability to autonomously adapt it priors, the target is reached almost every time in one shot thanks to the initial 10 random samples of the initialization procedure (not counted in the number of trial) that were enough to infer the right transformation of the priors.

statistically significant (p-value = 0.08). Moreover, the success rate is better when the algorithm does not use any prior (99.5% without priors version 95.5 with raw priors). This phenomenon can likely be explained by the fact that the forward model outputs values in meters that are relatively small (lower than 1 meter) compared to the observations in pixels (several hundreds of pixels) and that these different sources of information are added regardless their units. Consequently, after the 10 random trials that initialize the model, the prior is completely neglected and is just considered as noise by the model. It is likely that in the opposite situation, i.e., if the priors are several orders of magnitude larger than the observations, the learning performance may be dramatically affected.

When the algorithm is able to autonomously adapt its prior according to the observations, the actual benefits of the prior are revealed: the priors allow the algorithm to reach each target in almost one trial (median in 1 trial, 5^{th} and 95^{th} percentiles in [1; 3] trials). The fact that the robot is able to reach the first target in one trial shows that the algorithm is able to autonomously infer the correct transformation to adapt the priors in less than the 10 first random trials (performed during the initialization of the model, and not counted in the total of required samples plotted in the figures, see section 5.2.5.2).

When the robot has to reach several targets sequentially, the results are similar (see Fig. 5.9). Like in the previous experiment, as long as the robot remains intact, the priors allow the algorithm to reach each target in almost one trial (see Fig.

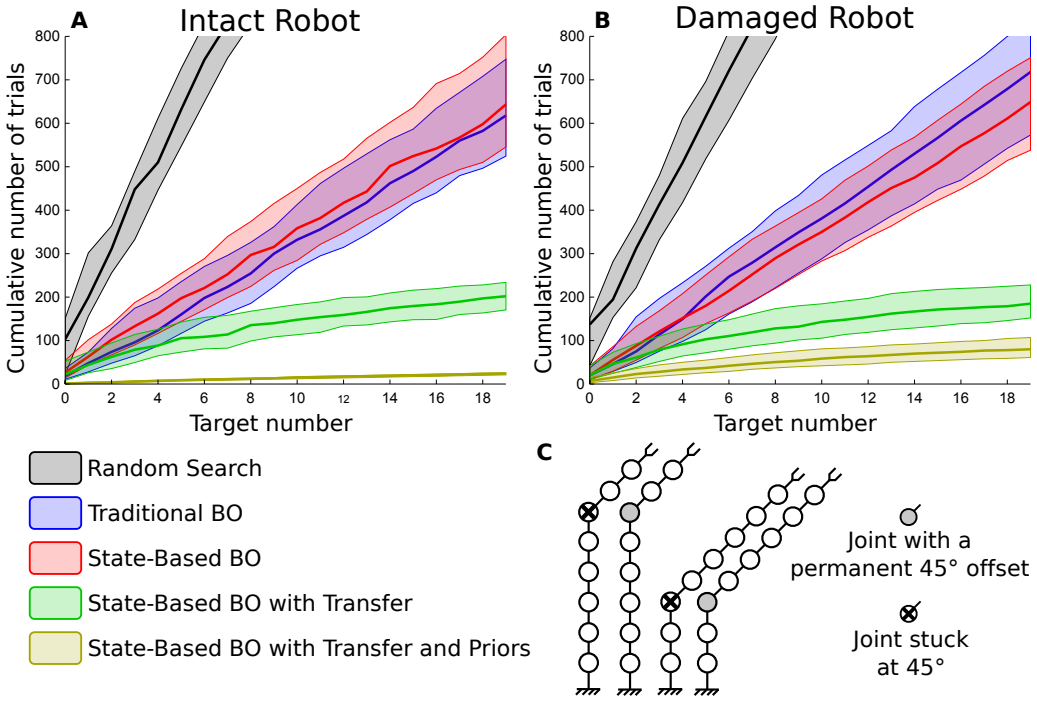


Figure 5.9: The benefits of using prior information. (A) Cumulative number of trials required by the intact robot to reach 20 targets. (B) In this case, the robot is damaged in four different ways defined in (C). The middle, solid lines represent medians, while the colored areas extend to the 25th and 75th percentiles. Both experiments have been replicated 100 times for each of the 5 tested algorithms and for each of the damage situations.

5.9 A). The first target is reached in one trial (median in 1 [1; 3] trials) while reaching subsequently the 20 targets requires 24 trials (24 [17.5; 30] trials). It is possible to reach 20 targets in less than 20 trials when a movement previously executed reached the currently considered target. In other words, when the robot has reached a position, which is afterward specified as a target (or within the 50 pixels radius), the algorithm considers that it already achieved its task, because it already has a solution for this task and move on the next target. This is why the 5^{th} percentile of the results presented in this experiment needs only 17.5 trials to reach 20 targets.

The benefits provided by the priors, when they are well adapted, are clearly visible in this experiment. All the compared approaches, which do not use priors, require significantly more time to reach the 20 targets, for example the State-based BO with Transfer needs 202 trials in median (202 [136.5; 307] trials, $p < 2.4 \times 10^{-34}$), which is one order of magnitude slower than the same algorithm but with the adapted priors.

When the robot is damaged, the task is more challenging for the variant using prior knowledge. In this situation, the prior (here the forward model) is even more inconsistent according to the reality, because, in addition to the orientation and scale problem, the model does not take into account the damage. Nevertheless, the results show that the priors remain useful, as the algorithm requires about 2.5 times less trials[3] to reach the 20 targets than the same variant but without using the priors (80 [44.4; 148.2] trials for State-based BO with Transfer and Priors versus 185 [105.4; 309.4] trials for State-based BO with Transfer, see Fig. 5.9 B).

With this additional feature that adapts the prior information to the acquired observations, our algorithm is able to maximize the utility of the priors and accomplish its missions significantly faster. In our experiments, we showed that our algorithm is able to turn a forward model initially useless into a valuable tool to reach the targets instantaneously. We also showed that this new feature does not alter the robot's ability to deal with damage situations. In the following section, we will show that these results hold with the physical robot its inherent inaccuracies.

Algorithm 5 State-Based BO with Transfer, Priors and blacklists

for $i < N_{init}$ **do**
  $\boldsymbol{\chi} \leftarrow$ RANDOM
  EVAL_SOLUTION($\boldsymbol{\chi}$)
OPTIMIZATION_LOG-LIKELIHOOD
while REMAINS_TARGET **do**
  $\mathbf{T} \leftarrow$ SELECT_TARGET(State_space)
  $\forall \mathbf{x} \in$ Search_space:
    $P(f(\mathbf{x})|\mathbf{Pc}_{1:t+1}, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_{t+1}(\mathbf{x}), \sigma^2_{t+1}(\mathbf{x})\mathbf{I})$
  $\boldsymbol{\chi} \leftarrow \arg\max_{\mathbf{x}} \Big(-\text{dist}\Big(\mathbf{T} - \boldsymbol{\mu}_{t+1}(\mathbf{x})\Big) + \kappa\sigma_t(\mathbf{x})\Big)$
  EVAL_SOLUTION($\boldsymbol{\chi}$)
  OPTIMIZATION_LOG-LIKELIHOOD
procedure EVAL_SOLUTION
  $\mathbf{Pc} \leftarrow$ STATE(physical_robot($\mathscr{C}(\boldsymbol{\chi})$))
  if $\mathbf{Pc} \neq \emptyset$ **then**
    $\boldsymbol{\chi}_{t+1} \leftarrow \boldsymbol{\chi}$
    $\mathbf{Pc}_{t+1} \leftarrow \mathbf{Pc}$
  else
    $\mathbf{bl}_{n+1} \leftarrow \boldsymbol{\chi}$

5.5 Evaluation on the physical robot

5.5.1 The whole framework

In this last section, we evaluate the whole framework of our state-base BO with Transfer, Priors and blacklists algorithm. This framework is summarized in the pseudo-code 5. For reading convenience, the mathematical notation are reported in the appendix D.3

Behind the term $P(f(\mathbf{x})|\mathbf{Pc}_{1:t+1}, \mathbf{x})$, the algorithm gathers all the features presented previously: (1) the state-base BO defined in the equation 5.11 and which allows the algorithms to transfer knowledge from the previous tasks to the following ones, (2) the use of blacklists of samples introduced in equation 5.18 and (3) the automatic adaptation of the priors that can be found in equation 5.21. The optimization of the log-likelihood follows the equations 5.17 for the kernel's hyper-parameters

[3]It is important to mention that in this experiment, contrary to those presented in the previous chapter, the priors do not reduce the search space. This explains why the reduction of the number of trials decreases "modestly" (2.5 times) in this experiment while it decreased by an order of magnitude in the previous chapter. We decided to use the forward model as a prior knowledge instead of a behavior-performance map because the very fast adaptation abilities provided by the maps make the benefits provided by the new features introduced in this chapter less visible. For example, the number of trials decreases from 6 to 4 after transferring knowledge. Is it thanks to the knowledge transfer that it decreased or just because of the variability of the results (especially important in these experiments)? The higher difficulty of these experiments, which do not use behavior-performance maps, allows the results to be unambiguous.

and 5.22 for the coefficient of the transformation matrix (the matrix is visible in appendix D.3). The values of the parameters used for the experiments presented in this chapter can be found in appendix C.4.
The *Remains_target* function returns true while a target remains to be reached and the *Select_Target* selects one target in the remaining ones. This last function is called at the beginning of every iterations, which allows the algorithm to switch his current target as soon as need, depending on external constraints. In our experiment the robot changes his target only after reaching its current one. However, we can imagine situations in which the robot may need to change its target more frequently. For example, a robot learning to walk in every direction, have to change its current target (i.e. walking direction) according on appearance of obstacles, even if it did not managed to completely reach its target yet. Such strategy is fully possible with the algorithm presented in this chapter, and may, for example, allow robots to learn to walk and in the same time avoid obstacles, and thus removes the obligation to replace the robot at its initial position between each trial, like it is customary in the literature.

5.5.2 Experimental setup

For this last experimental validation, we apply our algorithm on the physical robot arm (see Appendix B.2) and evaluate its performance on the same task than the previous experiments performed in simulation (reaching subsequently 20 targets). The reality through the looseness of the robot's joints, the inaccuracies of the video camera and the different light conditions adds another layer of difficulty in the task. In this situation, the blacklists of samples are particularly useful, as several situations make it impossible to assess the location of the laser point. For example, the laser point can be outside the camera's range (see Fig. 5.10 A-B) or hidden by an obstacle (here a structure pillar, see Fig. 5.10 C-D). Moreover, light conditions may affect the tracking system, like when the laser point is in the shadow of the arm (the celling lamp is placed on the top of the setup).
Another reason that makes it impossible to evaluate samples is the sanity check of auto-collisions that may fail. Before executing any behavior on the physical robot, a sanity check is launched to verify, in simulation, that no auto-collisions occur (For this sanity check, the simulation takes into account the undergoing damage[4]). If the sanity check fails, the behavior is not tested on the physical robot and the sample is blacklisted. This procedure is required to be able to perform several replicates of the experiment without repeatedly worn the robot. The artificial 5% chance that a sample becomes blacklisted, used in the simulated experiments, is disabled in this experiment because there are already too many reasons for blacklisting samples.
In this real context, the ability to adapt the prior is also mandatory to be able to deal with the different sources of information: the targets are defined in the

[4]While not realistic, this sanity check is useful to perform a large number of replicates and to prevent to worn out or break the robot during the experiments. Moreover, the algorithm could work identically without this sanity check.

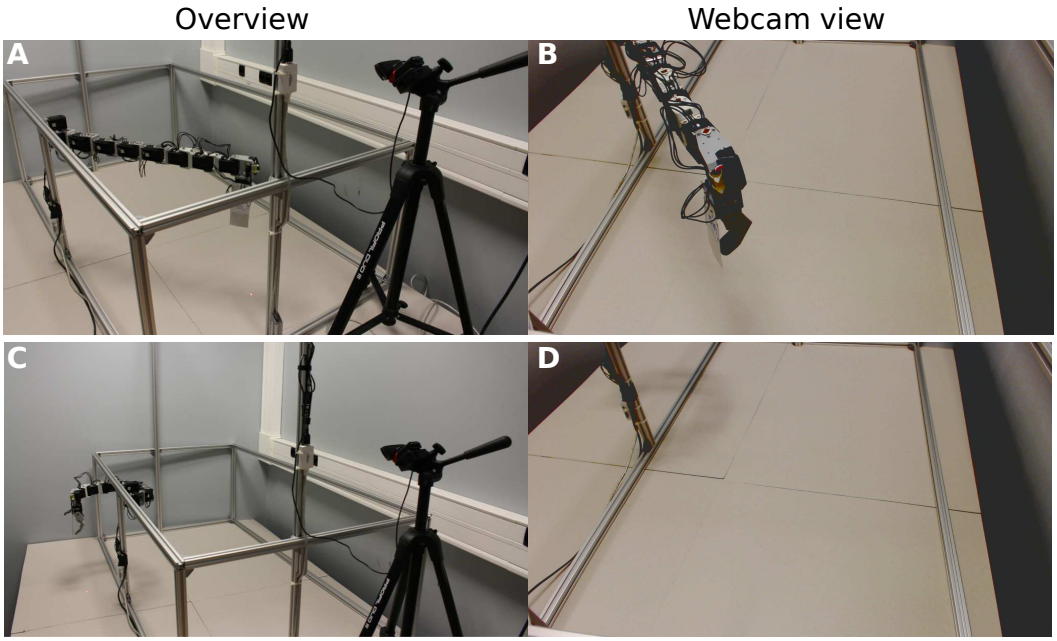


Figure 5.10: Two examples of situation where the laser point is not visible. (A-B) The laser point is too close from the camera and consequently outside the camera's range. (C-D) The laser point is hidden by the pillar of the structure.

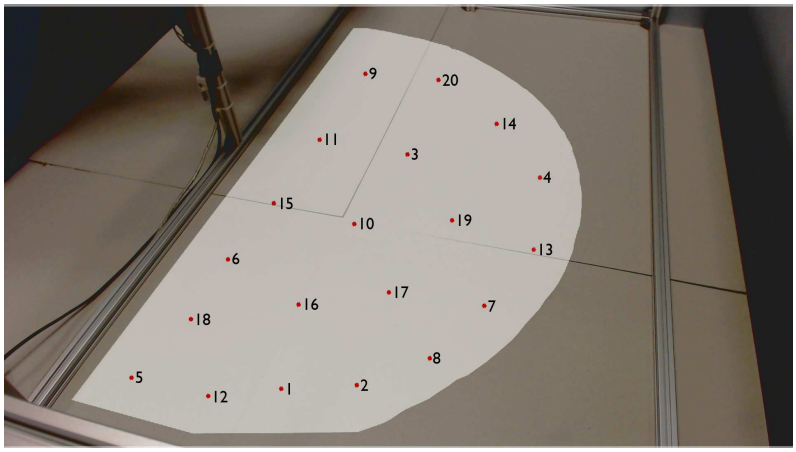


Figure 5.11: The reachable space of the robot, in white, has been determined by recording the position of the laser point when moving the arm while it is completely unfold. The white region has then been contracted by 10 pixels to remove points that may be too difficult to be reached (the border of the reachable space). Based on the resulting reachable area, depicted in white, a k-meansSeber (1984) algorithm has been used to evenly spread 20 targets. The order of the target has been generated randomly and is displayed in this picture.

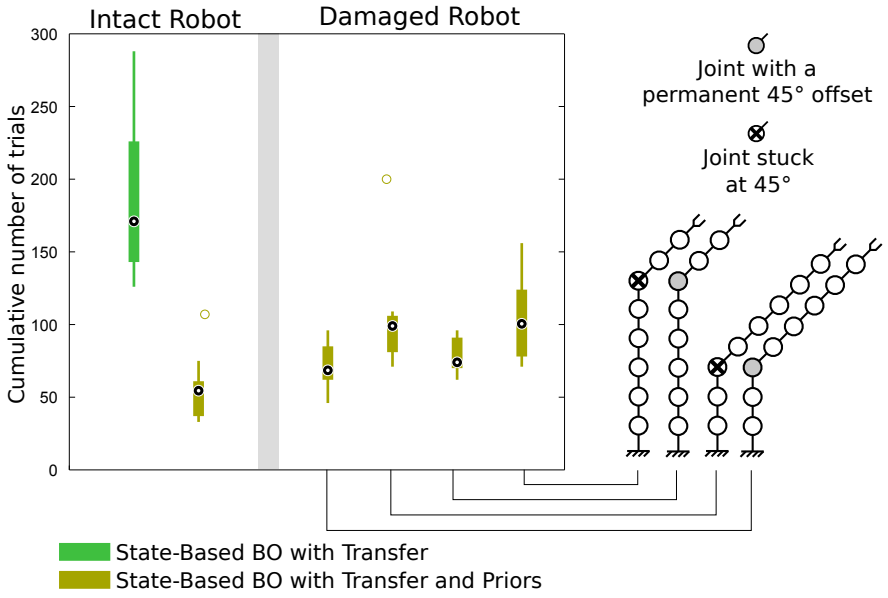


Figure 5.12: Learning to sequentially reach several targets with a real robot. The total number of trials required to reach 20 targets is depicted. For the intact robot, the same algorithm is launched but without prior knowledge in order to show the benefits it provides. The algorithm is also tested on four damage conditions.

camera's frame (in pixel), the position of the laser is also assessed in the camera's frame (in pixel) but the forward model, used as a prior, is based on the robot's base frame and computed in meters (like in the previous experiments). Note that in this experiment, the camera is placed in an arbitrary position and orientation that allow it to perceive the entire scene.

We apply the State-based BO with Transfer and Priors on the physical robot and we consider both the intact robot and the four damage situations (see Fig. 5.9 C). The experiment is replicated 10 times per situation. For comparison, the State-base BO with Transfer (but without Priors) is also tested on the physical robot. This reference experiment is only tested on the intact robot because it requires a significantly larger number of trials and consequently a larger amount of time to reach the 20 targets. We conducted 10 replicates of this reference experiment. Because the number of replicates is lower than in the previous experiments (it is materially challenging to perform more replicates), the targets are predefined and always the same for all the runs (see Fig. 5.11). This removes the variability coming from the random selection of the targets, but imposes relation between the different targets. For example, reaching the second target after the first one will be particularly easy (see Fig. 5.11), while the third one will probably not benefit from any knowledge acquired previously. In order to perform several replicates, the radius of the targets has been extended to 75 pixels (corresponding approximately to 5 centimeters) to slightly reduce the learning time.

5.5.3 Experimental Results

The experimental results (see Fig. 5.12) show that the proposed algorithm allows the robot to quickly reach the 20 targets, even when damaged. When the robot is intact and uses its forward model as a prior, it reaches the 20 targets in about 55 trials (54.5 [33; 107] trials corresponding to 4.9 [2.9; 12.7] minutes) while without the prior it requires about 170 trials (171 [126; 288] trials and 29.9 [14.0, 170.8] minutes). Like in the previous experiment in simulation, the use of prior allows the robot to divide by about three the number of required trials. By contrast, it requires twice more trials to reach the 20 targets in reality than in simulation (medians: 24 versus 54.5 trials). This comes from the differences between the theoretical forward model and real robot (i.e., reality gap problem, Koos et al. (2013b)). While in simulation, the virtual robot and the forward model match perfectly, the looseness of the robot and the inaccuracies of the cameras make the physical system notably different from the forward model.
When the robot is damaged, the number of trials increases modestly (median over the 4 damage conditions: 84 trials versus 54.5 trials when intact). Globally, when the robot is damaged, it requires in median 8.1 [5.0; 24.2] minutes to adapt to the situation and reach the 20 targets. The situations with blocked motor seems to be more challenging than the other damage conditions, regardless the location of the damage. It is likely that the algorithm uses its ability to transform its priors in order to integrate the permanent offset. It also worth noting that even when the robot is damaged, the algorithm with priors reaches the 20 targets faster than the algorithm that does not use the priors on the intact robot. Globally, the robot needs in average between 55 and 85 trials to reach the 20 targets while being damaged or just to cope with the reality gap problem.
The results present two outliers that may come from a problem during the learning process. For example, if a motor is blocked or overheats, it switches to a security mode and stops functioning. Such problem that occurs during the learning process is dramatic because it makes the previous observation outdated, as they do not consider the current condition of the robot. A possible way to mitigate this problem could be add a time stamp on the samples, which can then be taken into account when computing the covariance matrix. With this information about the acquisition time, recent samples will more strongly influence the model than older ones.

5.6 Conclusion

In this chapter we presented improvements that allow the intelligent trial and error algorithm (mainly its adaptation step, using the Model-based Bayesian Optimization) to be applied to more complex scenarios or to cope with additional problems. In particular we described how Gaussian Processes can be extended to model multi-channel functions and we propose to define the tasks (or the targets) in the state-space of the robot instead of using score or performance functions. This allows the algorithm to transfer knowledge between different tasks. While this technique

has no impact on the learning performance when considering a single task, the knowledge transfer ability that it provides reduced the time required to achieve the subsequent tasks. We also proposed a way to deal with solutions that cannot be tested on the system, for different reasons like when it is impossible to measure the performance or when a sanity check fails and prevents from executing the solution on the robot. These situations generate missing data that are often a conundrum when porting a learning algorithm on a physical robot and which can have dramatic effects on the algorithms' performance. We proposed to use blacklists of samples to make Gaussian Processes robust to these missing data. This technique does not affect the learning performances compared to situations without missing data and is computationally efficient. Finally, we presented how to add more flexibility when using prior information. The previous chapter suggested that using such prior knowledge is one of the keys to accelerate learning processes. Unfortunately, these priors are not always provided in a form that maximizes the amount of information that can be exploited. In this chapter we saw several examples of situation that turn priors into misleading information, like scale factor problems. The proposed technique allows the algorithm to automatically adapt the priors according to its observations in order to maximize the potential information that can be exploited. In the worst situations, the algorithm can also autonomously disregard its priors and start learning from scratch.

We individually evaluated the benefits of all these features in simulation and applied the algorithm with all these new features on a physical robotic arm that has to subsequently reach several targets. All combined, these methods allow the system to be very flexible and adaptive: without any calibration or configuration, the robot is able to reach targets defined in the camera image, which is placed in an arbitrary position. The robot autonomously adapts its prior information about itself (forward model) and after a few trials reaches the first target. It then uses its previous experiences to reach the subsequent targets faster until reaching them in only one shot.

The presented algorithm is not specific to one kind of robot and it makes minimal assumptions about its morphology or its capabilities. Consequently, the algorithm can be employed with many different types of robot and may make them easier to be deployed without requiring extensive calibrations or expensive repairs after being damaged. For example, this algorithm might allow our hexapod robot to adapt to damage situations while walking in every direction.

Chapter 6

Discussion

Contents

Throughout this manuscript, we introduced three main concepts that allow robots to adapt quickly in various situations thanks to learning algorithms that are both *fast* and *creative*:

- Using simulations to learn faster: combining simulations and tests on physical robots is an efficient way to reduce the time required to learn a behavior.

- Gathering collection of solutions into Behavioral Repertoires: the creativity of evolutionary algorithms can be encapsulated in *Behavioral Repertoires*, which gather large collections of both high performing and diverse solutions that are sorted according to their behavior (via the behavioral descriptor).

- Exploring the information provided by Behavioral Repertoires: Bayesian Optimization (with the extensions introduced in the previous chapter) can employ the information contained in the behavioral repertoire to rapidly explore it and find an adequate behavior according to the situation.

In the following sections, we will discuss the limitations of each of these concepts and give examples of situations in which our methods may be difficult to apply. We will propose improvements that may mitigate these limitations and potentially increase the abilities of robots. In particular, we will show that several of the concepts presented in this manuscript and some of the improvements that we will propose share similarities with observations made in neuroscience about the learning processes that happen in mammals' brain. We will also highlight the links between our approaches and the field of developmental robotics, which takes inspiration from the cognitive development of infants and aims to allow robots to autonomously develop their own representations, to discover their environment and to learn new skills to be able to face new situations (Lungarella et al., 2003).

6.1 Using simulations to learn faster

Our experiments show that using the simulation is a promising approach to speed up learning processes. Indeed, behaviors can be evaluated faster in simulation than in reality and several behaviors can be evaluated in parallel, which is impossible when a single robot is available. However, behaviors found in simulation often do not work as expected in reality because learning (evolutionary) algorithms tend to exploit inaccuracies of the simulation to artificially boost their performance (Jakobi et al., 1995; Boeing and Braunl, 2012; Koos et al., 2013b). Across all this manuscript, we hypothesize that there exist behaviors that work similarly in simulation and in reality, even when the simulation and the reality differ significantly, like in damage conditions. Our experimental results illustrate that this hypothesis holds in at least all the scenarios we tested (see experiments in section 4.3.3.1). Nonetheless, it may exist situations in which this hypothesis is wrong. For example, manipulation tasks can be difficult to model in simulation because they often involve complex interactions. Another example is when the simulated environment is very different from the real one. Typically, it is unlikely to find a behavior that would allow a robot to climb stairs in reality if the simulation only relies on a flat ground.

To address this problem, we can imagine to allow the robot to progressively create, refine or enhance the simulator based on its experience in order to make it as close as possible to the reality. The algorithm presented by Bongard et al. (2006) (see section 4.1.2), which uses simple tests on the physical robot to infer its morphology, is a method that can typically be used to improve the simulator. Based on the behaviors executed during the robot's mission and corresponding observations, the algorithm can modify the simulation parameters (e.g., the robot's virtual morphology, the parameters of the virtual environment) to reproduce the observations in simulation or define the actions that have to be performed the following day to remove the ambiguities. With this updated self-model, the robot may more efficiently continue its mission and gather additional data that will be used afterward to further improve the self-model.

The algorithm presented by Schmidt and Lipson (2009) is another example of potentially useful algorithm that can be used to create a simulator from scratch based on the robot's experience. This approach uses a multi-objective genetic programming algorithm to infer the equations that explain the observed data. The algorithm outputs a set of equations that correspond to different trade-offs between simplicity and accuracy of the equations. For example, it has been used to infer the equations of physics that drive the movements of a double pendulum. A similar approach can be used to let robots build their own simulators by discovering the laws of physics and the consequences of their actions. Nonetheless, while it is possible to automatically designing a simulator based on noisy observations for simple systems like a double pendulum, doing the same with complex systems like legged or humanoid robots remains a serious challenge (Zagal et al., 2009).

The concept of replaying the robot's actions in simulation is similar to what seems to occur in the brain of mammals during sleep periods. Indeed, it has been observed

that some neural activities that occurred during awake periods are replayed during the night (Wilson and McNaughton, 1994; Peyrache et al., 2009). For example, it has been observed with some birds (the Zebra finch) that the pattern of neural activity produced while the bird is awake and singing, is reproduced while the bird is sleeping (Dave and Margoliash, 2000). As pointed out by Derégnaucourt et al. (2005), this observation "suggests the possibility of song rehearsal during sleep". The same observation has been made with place cells in rats' brain. These cells, which fire depending on the rat's location (O'keefe and Nadel, 1978; Moser et al., 2008), produce the same pattern of activity during spatial experiments and during the following sleep periods (Wilson and McNaughton, 1994; Skaggs and McNaughton, 1996a; Louie and Wilson, 2001; Peyrache et al., 2009; de Lavilléon et al., 2015). These experiments suggests that rats are replaying during their dream the path they followed when they were awake.
Moreover, a large number of studies revealed evidences that these replays during sleep periods play an important role in the developmental learning of both animals and humans (Derégnaucourt et al., 2005; Peigneux et al., 2004; Maquet et al., 2000; Gais and Born, 2004; Hobson and Pace-Schott, 2002). For example, Derégnaucourt et al. (2005) show that a deterioration in structure of the zebra Finch's song happens after night-sleep, and is followed by a rapid improvement after intense morning singing with a significant variation in the song structure. In de Lavilléon et al. (2015), the authors artificially produced rewarding stimulation in mice's brain during activity replays of place cells in order to create artificial memories. The produced place-reward association incited the mice to move to this location the follow day even if they never received rewards in reality. All these studies suggest that replaying actions or events during off-line periods (like dreams) appear to be important parts of the learning process of animals and humans. We can thus imagine that similar strategies may be instrumental to improve the cognitive abilities of robots.
It is interesting to note that replays of neural activities occur not only during sleep periods, but also during awake periods: before the task (Diba and Buzsáki, 2007; Dragoi and Tonegawa, 2011) and after the task (Foster and Wilson, 2006). While re-activations after the experience (for example, spatial experience) are supposed to play a role in the evaluation the behaviors, like in a reinforcement learning process, the pre-activations may suggest that the brain is "simulating" the states the animal may encounter in order to improve its anticipation or planning abilities (Diba and Buzsáki, 2007; Ferbinteanu and Shapiro, 2003; Jeannerod, 2001). There are also several observations that suggest that the brain is using *neural simulation* in order to "provide the self with information on the feasibility and the meaning of potential actions" (Jeannerod, 2001). It has been observed that these replays or pre-plays are compressed in time by a factor up to 6 or 7 (Euston et al., 2007; Diba and Buzsáki, 2007; Skaggs and McNaughton, 1996b). These properties shares similarities with our algorithms (Both the T-Resilience and the Intelligent Trial and Error) that first evaluate solutions in simulation before executing them in reality and take advantage of the time compression provided by the simulation.

6.2 Gathering collections of solution into Behavioral Repertoires

In the chapter 4, we showed that the dimensionality reduction and the information provided by the behavior-performance maps are the two key factors that allow our robots to adapt quickly. We presented two algorithms that generate such behavioral repertoires: TBR-Evolution and MAP-Elites (see chapter 3). These two algorithms require the users to define the dimensions of the behavioral space in which he is interested in seeing variation in. This choice is critical because it influences the types of solution that can be discovered by the algorithms. For example, in a first experiment, we defined the behavioral descriptors as the X/Y position of the robot at the end of its movement. The behavioral repertoires produced with this configuration then contained turning behaviors, as turning allows the robot to reach new locations (see section 3.3.2.1). In a second experiment, we defined the behavioral descriptor as the proportion of time that each leg touches the ground, and in this case, the resulting repertoires contained behaviors that walk in a straight line (see section 3.3.2.2). The performance function also plays an important role in the types of solution produced by the algorithms. For instance, the walking behaviors that walk on a straight line, mainly emerged because the performance function promoted behaviors with a high forward walking speed. Even if the algorithms that employ the behavioral repertoires (like the Intelligent trial and Error) can be robust to the different behavioral descriptors (see experiments in section 4.3.3.5), selecting the most appropriate behavioral descriptor or performance function remains an open question or requires knowledge from an expert.

While our concept of behavioral repertoires comes from robotics (i.e., how to learn several actions, like walking in every direction, see chapter 3), we can note that some evidences suggest that animals may have a form of behavioral repertoire encoded brain into their brain. For example, it has been shown (Graziano, 2006; Graziano and Aflalo, 2007) that primates use different regions of their cortex for different *primitive* actions: the excitation of different areas of a monkey's brain makes the animal execute different primitive actions, like grasping or hand-mouth interactions. Moreover, in Graziano and Aflalo (2007), the authors underline that "*One way to describe the topography of the cerebral cortex is that 'like attracts like.'The cortex is organized to maximize nearest neighbor similarity or local continuity.*" These two properties are similar to those of behavioral repertoires, which contain several low-level behaviors (that can be regarded as "primitive") that are arranged according to their similarities (behaviors with similar behavioral descriptors are closely located in the repertoire).

From a higher level perspective, we can observe that the way our algorithms build behavioral repertoires shares similarities with what happen in the development of infants. In order to discover the relation between their movements and their perceptions, infants perform repeatedly analogous behaviors. This natural developmental process is called *"body babbling"*. In Meltzoff and Moore (1997), the authors explain

"what is acquired through body babbling is a mapping between movements and the organ-relation end states that are attained." They define organ-relation end states as the relative position of the organs that can be can be monitored via proprioception. The authors illustrate this notion with the example of infants that perform vocal babbling to learn the mapping between the movement of their tongue, mouth and lips to the auditory consequences.

A direct parallel can be drawn between Behavioral Repertoires and body babbling. The mapping between movements and perceptions acquired through body babbling is analogous to a Behavioral Repertoires, which maps the parameter values (that define the performed movements) to the corresponding behavioral descriptors (that define the corresponding perceptions). The way both of these mapping are generated is also similar. The behavioral repertoires and infants' internal mapping are both construct by performing random movements.

The concept of body babbling has been particularly studied in the field of developmental robotics (Lungarella et al., 2003; Demiris and Dearden, 2005; Saegusa et al., 2009; Baranes and Oudeyer, 2013; Rolf et al., 2010). The presumed central role of body babbling in the development of infants makes it a promising tool for developmental robotics. In particular, it has been recently opposed two models of babbling: (1) motor babbling and (2) goal babbling. The main difference between these two concepts is that instead of randomly exploring the motor space, like in motor babbling, goal babbling focuses the exploration around particular goals (defined in the goal space). Several studies showed evidences that the body babbling of infants is likely goal-directed (Rolf et al., 2010; Von Hofsten, 2004).

Goal-directed babbling has been successfully implemented in robotics (Baranes and Oudeyer, 2013; Rolf et al., 2010), typically to allow robotic arms to autonomously discover their inverse kinematic model. For example, the "SAGG-Random" algorithm[1] (for Self-Adaptive Goal Generation) (Baranes and Oudeyer, 2013) randomly selects a goal in the goal space and the robot uses a progressively built model to reach the goal. Each failed attempt is used to refine the model and allows to the robot to become more accurate. Once the goal is reached, another goal is selected and the process repeats. The objective of the algorithm is to build a model that allows the robotic arm to reach any possible point of its reachable space from any initial condition.

In order to explore the goal space, the robot has to know what is a "goal" among its sensory space. Indeed, the sensory space of often larger than the goal space, which makes the link between goals and observations nontrivial. For example, in reaching tasks with a robotic arm, the robot needs to know that a goal is defined by the position of the target, while it can perceive the angular positions of his joints, the applied torques and potentially much other information. The constraint of

[1]This algorithm is a variant of the SAGG-Robust Intelligent Adaptive Curiosity. In this algorithm, the goal, instead of being randomly selected, is selected via a heuristic (an artificial curiosity) that select the goal that is expected to maximize the learning progress of the robot. In this discussion we consider only the "random" variant because this algorithm is relatively similar to MAP-Elites.

pre-defining the goal space contrasts with the objectives of developmental robotics, as this field aims to allow robots to discover on their own their abilities and in particular the main structures present in their perception.
Moreover, a goal space is likely to contain goals that are unrealizable, like the points outside the reachable space of the robot (Baranes and Oudeyer, 2013). Such goals may mislead goal-babbling approaches, which may spend some time trying to reach these impossible targets. The probability of selecting an unrealistic goal is likely to increase with the size of the goal or sensory space. For example, if the sensory space is the image of the robot's camera, then the dimensionality of the sensory space is equal to the number of pixels times the number of color that each pixel can display. Randomly selecting a goal in such sensory space corresponds to randomly selecting the color of each pixel. It is then impossible for a robot to find a movement that generates this type of perception. This is a typical example of a goal space that cannot be handled with traditional goal-babbling approaches without providing a noticeable amount of prior-knowledge (i.e., how to extract goals from the images).
While both TBR-Evolution and MAP-Elites can be considered as body-babbling approaches, they cannot be categorized neither as a motor babbling nor as goal babbling approach but rather as a combination of these two types of approach. The exploration of the search space, while performed in the parameter/motor space (only the parameter values are altered by the algorithms), is focused on particular regions of the behavioral or goal space. TBR-Evolution focuses on novel behaviors and MAP-Elites focuses on elites ones (those that are both diverse and high-performing). While the notion of behavioral descriptor is related to the notion of goal space (because both of them are subspaces of the sensory space), our algorithms does not rely on randomly generated goals but rather on descriptors they actually observed. Thanks to this property, both of the TBR-Evolution algorithm and MAP-Elites can deal with misleading or large goal spaces. In our previous example of the camera image, our algorithms will focus on behaviors that generate images considered as enough interesting to be added into the repertoires (because they are novel or an elite). These behaviors will then be modified in order to produce potentially interesting variations of these images and thus progressively explore the space of the possible images. These properties show that Behavioral Repertoires and the algorithms used to generate them could be promising tools to study the body babbling in developmental robotics
The ability of TBR-Evolution to focus on novel behaviors is also related to the concept of intrinsic motivation. In Oudeyer et al. (2007), the authors defined an intrinsic motivation as "the maintenance of an abstract dynamic cognitive variable". The novelty value of each behavior, computed thanks to the internal archive, is an abstract variable that determines on which regions of the search space the algorithm will focus. In this sense, the novelty search is similar to an artificial curiosity because it fosters the algorithm to discover novel behaviors. The definition of artificial curiosity proposed by Oudeyer et al. (2007) states that it aims to maintain at a maximal level the learning progress of the argent. In order to continuously improve his knowledge, the robot is fostered to explore new situations. While the

implementations of the Intelligent Adaptive Curiosity (introduced by Oudeyer et al. (2007)) and the Novelty Search (Lehman and Stanley, 2011a) differ, we can observe that both of them follow the definition of artificial curiosity. In addition to this link between the Novelty Search and the artificial curiosity, the different similarities between our methods and developmental robotics that we highlighted in this section, like the generation of behavioral repertoire and body babbling, suggest that evolutionary computation can provide new concepts of tools to developmental robotics. Such cross-fertilisation may be beneficial for both of these two research fields that have some goals in common (e.g., automatically creating autonomous agents or exploring unknown sensory-motor spaces, Lungarella et al. (2003); Lehman and Stanley (2011b); Delarboulas et al. (2010); Baranes and Oudeyer (2013)).

6.3 Exploring the information provided by Behavioral Repertoires

We saw in this manuscript that considering the behavioral repertoires as a new search space is an effective method to reduce the number of required trials to learn a behavior (see chapter 4 and 5). However, this approach imposes a strong hypothesis: *the solution of the problem should be in the repertoire, otherwise it cannot be found by the algorithm.* While we faced only one situation during our experiments in which the solution was not in the repertoire (see section 4.3.3.2), this problem may become more frequent if the behavioral space is not well chosen and if it does not contain diverse enough solutions. This problem is another illustration of the fact that the choice of the behavioral descriptors is critical.

In order to define the right behavioral descriptors, a typical method consists in foreseeing the different situations the robot may have to face or the different behaviors it may need and to select a behavioral descriptor that embraces a large variety of behaviors that may be useful to the robot. This procedure is similar to the traditional method employed in engineering for damage recovery (see chapter 4), which attempts to foresee the different ways a robot can become damaged in order to pre-design the different contingency plans of the robot. Compared to this traditional approach, the advantage of behavioral repertoire-based adaptation is that it works on an higher level, which makes easier to find a good solution that works on a large variety of situation. Indeed, the Intelligent Trial and Error does not require to precisely anticipate every situation the robot may face, but rather to define, via the behavioral descriptors, a family of behaviors that is enough diverse to contain promising solutions.

While we demonstrated that our approach works for a large variety of situations and robots, we can imagine to mitigate this limitation with the same alternative as the one we proposed to address the challenge of anticipating every situations: we can let robots learn on their own the most adequate behavioral descriptors based on their own experience. For example, and in addition to updating and improving the simulation as suggested in section 6.1, the robot can employ the

information gathered during the beginning of its mission in order to infer which are the behavioral dimensions that provide most of potentially useful behaviors and then re-generating or updating the repertoires according to this new behavioral space. For example, the algorithm can select the behavioral descriptors that produce the most diverse and high-performing collection of behaviors, or those that generate the largest diversity of behaviors.

The generation of the new behavioral repertoires can be executed when the robot is not active, for example during the night, or deported on a remote computing facilities (on the cloud). Once behavioral repertoire generated, for instance the day after, the robots may benefit from their improved source of knowledge to more efficiently carry on their mission and continue to gather data that will be used the following night. We can imagine that, via such alternation of active and passive periods that improve both the simulators and the behavioral repertoires, the robots can progressively become more robust and discover new behaviors or abilities.

This approach of day and night cycles shares several similarities with the restructurings that occur during mammals' dreams. As mentioned in beginning of this discussion, actions or events experienced during the day are replayed in the brain during the night, but dreams appear to be an important component of how the memory in general, and motor skill in particular, are consolidated, structured and *re-structured* (Gais and Born, 2004; Stickgold et al., 2001; Stickgold, 2005). In particular, it has been shown that dreams are instrumental to find solutions to problems, as commonly expressed by the concept of "sleeping on a problem" (Stickgold, 2005). For example, subjects are more likely to find a better solution to cognitive, visual or motor tasks after sleep than after wakefulness (Wagner et al., 2004; Stickgold et al., 2000; Walker et al., 2002; Huber et al., 2004). In the same vein as bio-inspired robotics (Pfeifer et al., 2007), implementing dream-like processes in robots that restructure their experience may allow them to become more robust, effective and autonomous.

This concept of dreaming robots will be investigate in a European project that started recently and which is named "DREAM: Deferred Restructuring of Experience in Autonomous Machines". The goal of this project is to combine tools from evolutionary computation or machine learning (like deep learning, LeCun et al. (2015)) with observations in neuroscience or in Psychology in order to design developmental learning approach that takes inspiration from the cognitive development of infants.

Chapter 7

Conclusion

Throughout this manuscript, we defined and evaluated a set of algorithms to make robots able to deal with unforeseen situations (like a mechanical damage) by learning on their own an adequate solution. To achieve this objective, the proposed learning algorithms attempt to combine both *speed* and *creativity*. These two properties are important because *quickly* finding a solution is critical in many situations, for instance in search and rescue missions, and *creatively* discovering new solutions allow the system to deal with truly unexpected situations.

In our review of the literature (chapter 2), we considered the Evolutionary Algorithms and the Policy Search algorithms, which are the two main families of learning algorithms used to learn low-level behaviors (motor skills). Evolutionary Algorithms are more creative than most of optimization algorithms because they are able to deal with large search spaces without being severely affected by local optimums. The main drawback of these algorithms is that they usually require several hundreds or thousands of evaluations to find a solution, which makes them difficult to apply on physical robots. Conversely, Policy Search algorithms are fast learning algorithms that are able to find solutions in about a hundred evaluations. Nonetheless, this learning speed is mainly explained by the fact that Policy Search algorithms are local search approaches, which need a good initialization procedure (for example, via an human demonstration) and usually converge toward the closest local optimum. This constraint limits the application of Policy Search algorithms on autonomous robots that operate on remote sites, as in such condition no teacher is available. At the end of our review, we presented the Bayesian Optimization algorithm, which is one of the fastest Policy Search algorithms. This algorithm is less affected by local optimums and does not need particular initialization procedure. However, it is limited to small search spaces, which prevents the algorithm to explore large search spaces to find creative solutions.

In this thesis, we proposed new algorithms that allow to combine the advantages of these three families of algorithm and to mitigate their limitations. With our algorithms, robots are able to adapt to many damage conditions in less than two minutes (which corresponds to a handful of evaluations) and to learn how to achieve 20 tasks in less than 10 minutes.

We first proposed to use Evolutionary Algorithms to generate collections of potentially useful behaviors in simulation. We called these collections "Behavioral Repertoires". The main advantage of these behavioral repertoires is that they project large parameter search space (with several dozens of dimension) into small behav-

ioral spaces (with less than a dozen of dimension). In addition of being smaller and thus easier to explore, the behavioral space generated by the behavioral repertoires contains exclusively behaviors that are predicted to be high performing. As a summary, behavioral repertoires encapsulate the creativity of evolutionary algorithm into small behavioral spaces that gather large collections of both high performing and diverse solutions that are sorted according to their behavior (via the behavioral descriptor).

We presented two algorithms, TBR-Evolution and MAP-Elites, that can be employed to generate behavioral repertoires (chapter 3). These algorithms generate several hundreds or thousands of behaviors without requiring learning each of them separately (with independent learning processes). We illustrated the abilities of our algorithms in two experiments. In the first experiment, our hexapod robot learned to walk in every direction, while in the second experiment, it discovered a large variety of ways to walk by differently using its legs. In both of these experiments, the proposed algorithms have been able to find large varieties of behaviors. Typically, MAP-Elites found 1270 controllers that allow the robot to walk in every direction and at different speeds and more about 13 000 different ways to walk on a straight line.

While the generation of these behavioral repertoires is only feasible in simulation, as it requires evaluating several million solutions, we showed that the information contained in these collections of behaviors are instrumental to allow robot to adapt to unforeseen situations like mechanical damage. Indeed, combining information coming from a simulator with tests on physical robots is an efficient way to reduce the time required to learn a behavior.

In particular, we presented the Intelligent Trial and Error algorithm (chapter 4), which explores the behavioral repertoire via a Bayesian Optimization algorithm. The Intelligent Trial and Error algorithm takes advantage of the creative solutions contained in the repertoires and of the speed of Bayesian Optimization to exploit the information contained in the repertoire to find an adequate behavior according to the situation. Our experiments revealed that the proposed algorithm allows robots to adapt to a large variety of damage situations in less than 2 minutes (i.e., less than 15 trials) while several hours are usually required with traditional learning methods (see chapter 2 and section 4.1.1). We first tested the algorithm on a physical hexapod robot injured in five different ways. For all these damages situations, the robot has been able to adapt in less than 2 minutes, even in the most difficult scenario in which it lost two legs. We then illustrated the fact that this algorithm is not specific to one type of robot by testing its performance on a robotic arm. The results showed that the robot is able to adapt in less than 30 seconds to the 14 damage scenarios tested. Moreover, we demonstrated that this algorithm can also be used to adapt to environmental changes, like sloping ground. Finally, we proved that the algorithm is robust to the choice of behavioral descriptor used to create the behavioral repertoires, even if this aspect is particularly critical, as it defines the types of solutions that will be found by the algorithm.

In the last part of this manuscript (chapter 5), we proposed three extensions of the

Intelligent Trial and Error algorithm to make it able to deal with three problems that often affect robotic experiments: (1) transferring knowledge from one task to faster learn the following ones, (2) dealing with solutions that cannot be evaluated on the robot, which may hurt learning algorithms and (3) adapting prior information that may be misleading, in order to maximize their potential utility.

We first proposed to extend the Gaussian Processes, used in the Bayesian Optimization, to allow the algorithm to transfer knowledge across different tasks. This extension is based on acquiring the observation directly in the state space of the robot instead of in its rewards space, because a reward is less informative than a whole state. These enhanced acquisitions allow the algorithm to generalize its knowledge over the different tasks the robot has to achieve. We evaluated this property with a robotic arm that has to reach subsequently 20 targets and the results revealed that the information gathered while reaching the first targets allows the robot to reach almost instantaneously the last ones.

With our second extension, we considered the problem of solutions that cannot be evaluated on the robot, which may happen for different reasons (see section 5.3) and may hurt the learning performance of the algorithms. We proposed to use a blacklist of solutions, which allows the algorithm to be robust to the missing data by avoiding problematic solutions. Our experimental validation showed that thanks to this simple and computationally effective extension, these missing data do not affect the learning performance of the algorithm.

The last extension of the Intelligent Trial and Error algorithm that we presented in this manuscript allows the algorithm to maximize the usefulness of prior information, for example provided by behavioral repertoires. Exploiting this source of knowledge is one of the keys that explain the results of the Intelligent Trial and Error algorithm (see section 4.3.3.3). However, it may happen that the information contained in the behavioral repertoire is not expressed in a form that is not meaningful for the algorithm. We proposed to make the algorithm able to autonomously adapt, via an homogeneous transformation, the information provided by the prior to make it match with the acquired observations and thus to maximize its potential utility. Our experiments showed that with less than 10 random evaluations, the algorithm has been able to infer the right transformation between a forward model (expressed in meter in the robot's base frame) and the observations of the location of the robot's gripper (expressed in pixel in the camera's based frame, which is placed in an arbitrary position close to the robot). Thanks to the discovered relation between the forward model and the observations, the robot successfully reached his targets always in the first trial.

With all these extensions, the Intelligent Trial and Error is designed to be a generic algorithm that can be applied to many types of robot, to adapt to unforeseen situation, to learn a large variety of actions, and to simultaneously improve the robot's performance on several tasks.

While all the algorithms presented in this manuscript stem from an engineering perspective, several parallels can be drawn between them and observations made on the neural activities in mammal's brain (chapter 6). In particular, several ob-

servations, like the pre-plays or replays of neural activities, suggest that the brain uses a "neural simulation" of actions to gather information that is used to improve its abilities. This neural process is similar to the main concept of our algorithms, which execute actions in simulation to gather knowledge about their consequences and to rapidly select the most promising behaviors that has to be tested in reality. Moreover, the links between our algorithms and the research field of developmental robotics suggest that Evolutionary algorithms may provide useful tools to this domain. For example, the Intelligent Trial and Error can be a valuable tool to generate a goal-oriented babbling, via the MAP-Elites algorithm. The information gathered thanks to this babbling can then be exploited to produce complex behaviors or to generate high-level knowledge, like object affordances from the robot's interactions its environment. The Intelligent Trial and Error algorithm is also a promising substrate to design active/passive cycles, which can take inspiration from day and night cycles that appear to play an important role in the developmental learning of both animals and humans.

Bibliography

Abramowitz, M., Stegun, I. A., et al. (1966). Handbook of mathematical functions. *Applied Mathematics Series*, 55:62. (Cité en page 39.)

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276. (Cité en page 31.)

Angeli, A., Doncieux, S., Meyer, J.-A., and Filliat, D. (2009). Visual topological slam and global localization. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 4300–4305. IEEE. (Cité en pages 52 et 96.)

Argall, B. D., Chernova, S., Veloso, M., and Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483. (Cité en page 111.)

Avriel, M. (2003). *Nonlinear programming: analysis and methods*. Courier Corporation. (Cité en page 9.)

Bachoc, F. (2013). Cross validation and maximum likelihood estimations of hyper-parameters of gaussian processes with model misspecification. *Computational Statistics & Data Analysis*, 66:55–69. (Cité en page 42.)

Bagnell, J. A. and Hneider, J. G. S. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1615–1620. IEEE. (Cité en page 33.)

Baldassarre, G. and Mirolli, M. (2013). *Intrinsically motivated learning systems: an overview*. Springer. (Cité en page 8.)

Baranes, A. and Oudeyer, P.-Y. (2013). Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73. (Cité en pages 149, 191, 192 et 193.)

Barfoot, T., Earon, E., and D'Eleuterio, G. (2006). Experiments in learning distributed control for a hexapod robot. *Robotics and Autonomous Systems*, 54(10):864–872. (Cité en pages 21, 25 et 129.)

Beer, R. D. and Gallagher, J. C. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive behavior*, 1(1):91–122. (Cité en page 21.)

Bellingham, J. G. and Rajan, K. (2007). Robotics in remote and hostile environments. *Science*, 318(5853):1098–102. (Cité en page 90.)

Benson-Amram, S. and Holekamp, K. E. (2012). Innovative problem solving by wild spotted hyenas. *Proceedings of the Royal Society B: Biological Sciences*, 279(1744):4087–4095. (Cité en page 139.)

Benureau, F. and Oudeyer, P.-Y. (2013). Autonomous reuse of motor exploration trajectories. In *Development and Learning and Epigenetic Robotics (ICDL), 2013 IEEE Third Joint International Conference on*, pages 1–8. IEEE. (Cité en page 146.)

Berenson, D., Estevez, N., and Lipson, H. (2005). Hardware evolution of analog circuits for in-situ robotic fault-recovery. In *Proc. of NASA/DoD Conference on Evolvable Hardware*, pages 12–19. (Cité en pages 51 et 93.)

Bernard, A., André, J.-B., and Bredeche, N. (2015). Evolution of cooperation in evolutionary robotics: The tradeoff between evolvability and efficiency. In *Advances in Artificial Life, ECAL*. (Cité en pages 11 et 22.)

Bertsekas, D. P., Bertsekas, D. P., Bertsekas, D. P., and Bertsekas, D. P. (1995). *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA. (Cité en page 9.)

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2008). Robot programming by demonstration. In *Springer handbook of robotics*, pages 1371–1394. Springer. (Cité en page 8.)

Billing, D. (2007). Teaching for transfer of core/key skills in higher education: Cognitive skills. *Higher education*, 53(4):483–516. (Cité en page 146.)

Blanke, M. and Schröder, J. (2006). *Diagnosis and fault-tolerant control*. Springer. (Cité en page 90.)

Blight, B. and Ott, L. (1975). A bayesian approach to model inadequacy for polynomial regression. *Biometrika*, 62(1):79–88. (Cité en page 175.)

Blum, M. and Riedmiller, M. (2013). Optimization of gaussian process hyperparameters using rprop. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. (Cité en page 41.)

Boeing, A. (2009). *Design of a physics abstraction layer for improving the validity of evolved robot control simulations*. Citeseer. (Cité en page 22.)

Boeing, A. and Braunl, T. (2012). Leveraging multiple simulators for crossing the reality gap. In *Control Automation Robotics & Vision (ICARCV), 2012 12th International Conference on*, pages 1113–1119. IEEE. (Cité en pages 22 et 188.)

Bongard, J. (2007). Action-selection and crossover strategies for self-modeling machines. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, pages 198–205. ACM. (Cité en page 241.)

Bongard, J. and Lipson, H. (2005). Nonlinear system identification using coevolution of models and tests. *IEEE Transactions on Evolutionary Computation*, 9(4):361–384. (Cité en page 94.)

Bongard, J., Zykov, V., and Lipson, H. (2006). Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121. (Cité en pages 22, 25, 50, 51, 90, 92, 93, 94, 96, 102, 111, 129, 140, 188 et 240.)

Bongard, J. C. (2013). Evolutionary robotics. *Communications of the ACM*, 56(8):74–83. (Cité en page 50.)

Bonilla, E. V., Chai, K. M., and Williams, C. (2007). Multi-task gaussian process prediction. In *Advances in neural information processing systems*, pages 153–160. (Cité en pages 145 et 154.)

Booker, A. J., Dennis Jr, J. E., Frank, P. D., Serafini, D. B., Torczon, V., and Trosset, M. W. (1999). A rigorous framework for optimization of expensive functions by surrogates. *Structural optimization*, 17(1):1–13. (Cité en page 35.)

Borji, A. and Itti, L. (2013). Bayesian optimization explains human active search. In *Advances in Neural Information Processing Systems 26 (NIPS)*, pages 55–63. (Cité en pages 35, 112 et 140.)

Boyle, P. and Frean, M. (2005). Dependent gaussian processes. *Advances in neural information processing systems*, 17:217–224. (Cité en pages 153 et 154.)

Braitenberg, V. (1986). *Vehicles: Experiments in synthetic psychology*. MIT press. (Cité en page 20.)

Brochu, E., Brochu, T., and de Freitas, N. (2010a). A bayesian interactive optimization approach to procedural animation design. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 103–112. Eurographics Association. (Cité en page 39.)

Brochu, E., Cora, V. M., and De Freitas, N. (2010b). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*. (Cité en pages 35, 36, 37, 42, 43, 114 et 115.)

Brooks, R. et al. (1986). A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23. (Cité en page 2.)

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and autonomous systems*, 6(1):3–15. (Cité en pages 1 et 2.)

Caccavale, F. and Villani, L., editors (2002). *Fault Diagnosis and Fault Tolerance for Mechatronic Systems: Recent Advances*. springer. (Cité en page 91.)

Calandra, R., Seyfarth, A., Peters, J., and Deisenroth, M. P. (2014). An experimental comparison of bayesian optimization for bipedal locomotion. In *Proceedings of 2014 IEEE International Conference on Robotics and Automation (ICRA)*. (Cité en pages 25, 35, 45, 51, 114, 115, 123, 126, 127, 128, 129, 151 et 162.)

Carlson, J. and Murphy, R. R. (2005). How UGVs physically fail in the field. *IEEE Transactions on Robotics*, 21(3):423–437. (Cité en page 2.)

Caruana, R. (1997). Multitask learning. *Machine learning*, 28(1):41–75. (Cité en page 146.)

Cesa-Bianchi, N., Conconi, A., and Gentile, C. (2004). On the generalization ability of on-line learning algorithms. *Information Theory, IEEE Transactions on*, 50(9):2050–2057. (Cité en page 147.)

Chang, C. and Lin, C. (2011). Libsvm: a library for support vector machines. *ACM Trans. on Intelligent Systems and Technology*, 2(3):27. (Cité en pages 57, 59, 60 et 101.)

Cheney, N., MacCurdy, R., Clune, J., and Lipson, H. (2013). Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 167–174. ACM. (Cité en page 11.)

Chernova, S. and Veloso, M. (2004). An evolutionary approach to gait learning for four-legged robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2562–2567. IEEE. (Cité en pages 21, 25 et 51.)

Christensen, D. J., Larsen, J. C., and Stoy, K. (2013). Fault-tolerant gait learning and morphology optimization of a polymorphic walking robot. *Evolving Systems*, pages 1–12. (Cité en pages 93 et 129.)

Cliff, D., Husbands, P., and Harvey, I. (1993). Explorations in evolutionary robotics. *Adaptive behavior*, 2(1):73–110. (Cité en page 21.)

Clune, J., Beckmann, B., Ofria, C., and Pennock, R. (2009). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 2764–2771. (Cité en page 230.)

Clune, J., Mouret, J.-B., and Lipson, H. (2013). The evolutionary origins of modularity. *Proceedings of the Royal Society of London B: Biological Sciences*, 280(1755):20122863. (Cité en pages 11, 14 et 54.)

Clune, J., Stanley, K., Pennock, R., and Ofria, C. (2011). On the performance of indirect encoding across the continuum of regularity. *IEEE Trans. on Evolutionary Computation*, 15(3):346–367. (Cité en pages 50, 51, 59, 228 et 230.)

Cohn, D., Atlas, L., and Ladner, R. (1994). Improving generalization with active learning. *Machine learning*, 15(2):201–221. (Cité en page 147.)

Corbato, F. (2007). On Building Systems That Will Fail. *ACM Turing award lectures*, 34(9):72–81. (Cité en page 90.)

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297. (Cité en page 144.)

Cox, D. D. and John, S. (1997). Sdo: A statistical method for global optimization. *Multidisciplinary design optimization: state of the art*, pages 315–329. (Cité en page 44.)

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187. (Cité en page 15.)

Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. (2015). Robots that can adapt like animals. *Nature*, 521(7553):503–507. (Cité en pages 93 et 162.)

Currie, K. and Tate, A. (1991). O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86. (Cité en page 51.)

Daniel, C., Neumann, G., and Peters, J. R. (2012). Hierarchical relative entropy policy search. In *International Conference on Artificial Intelligence and Statistics*, pages 273–281. (Cité en page 26.)

Dantzig, G. B. (1998). *Linear programming and extensions*. Princeton university press. (Cité en page 9.)

Darwin, C. R. (1859). *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. (Cité en page 10.)

Dave, A. S. and Margoliash, D. (2000). Song replay during sleep and computational rules for sensorimotor vocal learning. *Science*, 290(5492):812–816. (Cité en page 189.)

de Garis, H. (1990). Genetic programming: Building nanobrains with genetically programmed neural network modules. In *International Joint Conference on Neural Networks (IJCNN)*, pages 511–516. IEEE. (Cité en page 50.)

De Jong, K. A. (2006). *Evolutionary computation: a unified approach*. MIT press. (Cité en page 14.)

de Lavilléon, G., Lacroix, M. M., Rondi-Reig, L., and Benchenane, K. (2015). Explicit memory creation during sleep demonstrates a causal role of place cells in navigation. *Nature neuroscience*, 18(4):493–495. (Cité en page 189.)

Dean, T. and Wellman, M. (1991). *Planning and control*. Morgan Kaufmann Publishers Inc. (Cité en page 51.)

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons. (Cité en pages 15 et 98.)

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, 6(2):182–197. (Cité en pages 16, 17, 59, 62 et 98.)

DeDonato, M., Dimitrov, V., Du, R., Giovacchini, R., Knoedler, K., Long, X., Polido, F., Gennert, M. A., Padır, T., Feng, S., et al. (2015). Human-in-the-loop control of a humanoid robot for disaster response: A report from the darpa robotics challenge trials. *Journal of Field Robotics*, 32(2):275–292. (Cité en page 2.)

Defretin, J., Marzat, J., and Piet-Lahanier, H. (2010). Learning viewpoint planning in active recognition on a small sampling budget: a kriging approach. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 169–174. IEEE. (Cité en page 44.)

Deisenroth, M., Mchutchon, A., Hall, J., and Rasmussen, C. E. (2013a). Pilco policy search framework. `http://mloss.org/software/view/508/`. (Cité en page 160.)

Deisenroth, M. and Rasmussen, C. E. (2011). Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472. (Cité en pages 33, 45, 46, 148 et 159.)

Deisenroth, M. P., Fox, D., and Rasmussen, C. E. (2015). Gaussian processes for data-efficient learning in robotics and control. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 37(2):408–423. (Cité en page 46.)

Deisenroth, M. P., Neumann, G., Peters, J., et al. (2013b). A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142. (Cité en pages 9, 10, 24, 26, 27, 28, 29, 30, 31 et 33.)

Deisenroth, M. P., Rasmussen, C. E., and Fox, D. (2011). Learning to control a low-cost manipulator using data-efficient reinforcement learning. In *Robotics: Science and Systems Conference*. (Cité en pages 33 et 46.)

Delarboulas, P., Schoenauer, M., and Sebag, M. (2010). Open-ended evolutionary robotics: an information theoretic approach. In *Parallel Problem Solving from Nature, PPSN XI*, pages 334–343. Springer. (Cité en pages 8 et 193.)

Delcomyn, F. (1971). The Locomotion of the Cockroach Pariplaneta americana. *Journal of Experimental Biology*, 54(2):443–452. (Cité en pages 102 et 228.)

Demiris, Y. and Dearden, A. (2005). From motor babbling to hierarchical learning by imitation: a robot developmental pathway. *International Workshop on Epigenetic Robotics*. (Cité en page 191.)

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38. (Cité en page 32.)

Derégnaucourt, S., Mitra, P. P., Fehér, O., Pytte, C., and Tchernichovski, O. (2005). How sleep affects the developmental learning of bird song. *Nature*, 433(7027):710–716. (Cité en pages 140 et 189.)

Devert, A., Bredeche, N., and Schoenauer, M. (2008). Unsupervised learning of echo state networks: A case study in artificial embryogeny. In *Artificial Evolution*, pages 278–290. Springer. (Cité en page 11.)

Diba, K. and Buzsáki, G. (2007). Forward and reverse hippocampal place-cell sequences during ripples. *Nature neuroscience*, 10(10):1241–1242. (Cité en page 189.)

Ding, X., Wang, Z., Rovetta, A., and Zhu, J. (2010). Locomotion analysis of hexapod robot. *Proceedings of Conference on Climbing and Walking Robots (CLAWAR)*, pages 291–310. (Cité en pages 100 et 228.)

Dinh, H., Aubert, N., Noman, N., Fujii, T., Rondelez, Y., and Iba, H. (2013). An effective method for evolving reaction networks in synthetic biochemical systems. *Evolutionary Computation, IEEE Transactions on*. (Cité en page 11.)

Dissanayake, M. G., Newman, P., Clark, S., Durrant-Whyte, H. F., and Csorba, M. (2001). A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241. (Cité en page 252.)

Doncieux, S. (2013). Transfer learning for direct policy search: A reward shaping approach. In *Development and Learning and Epigenetic Robotics (ICDL), 2013 IEEE Third Joint International Conference on*, pages 1–6. IEEE. (Cité en page 146.)

Doncieux, S. and Mouret, J.-B. (2014). Beyond black-box optimization: a review of selective pressures for evolutionary robotics. *Evolutionary Intelligence*, 7(2):71–93. (Cité en pages 16 et 87.)

Dorigo, M. and Birattari, M. (2010). Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer. (Cité en page 15.)

Dragoi, G. and Tonegawa, S. (2011). Preplay of future place cell sequences by hippocampal cellular assemblies. *Nature*, 469(7330):397–401. (Cité en page 189.)

Dryanovski, I., Valenti, R. G., and Xiao, J. (2013). Fast visual odometry and mapping from rgb-d data. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2305–2310. IEEE. (Cité en pages 96, 227, 246 et 252.)

Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping: part i. *Robotics & Automation Magazine, IEEE*, 13(2):99–110. (Cité en pages 52 et 96.)

Eiben, A. E. and Smith, J. (2015). From evolutionary computation to the evolution of things. *Nature*, 521(7553):476–482. (Cité en page 11.)

Eiben, A. E. and Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer. (Cité en pages 4, 10, 11, 13 et 14.)

Endres, F., Hess, J., Engelhard, N., Sturm, J., Cremers, D., and Burgard, W. (2012). An evaluation of the RGB-D SLAM system. In *Proc. IEEE ICRA*. (Cité en pages 52 et 96.)

Erden, M. S. and Leblebicioğlu, K. (2008). Free gait generation with reinforcement learning for a six-legged robot. *Robotics and Autonomous Systems*, 56(3):199–212. (Cité en pages 93 et 129.)

Espiau, B., Chaumette, F., and Rives, P. (1992). A new approach to visual servoing in robotics. *Robotics and Automation, IEEE Transactions on*, 8(3):313–326. (Cité en pages 164 et 232.)

Euston, D. R., Tatsuno, M., and McNaughton, B. L. (2007). Fast-forward playback of recent memory sequences in prefrontal cortex during sleep. *science*, 318(5853):1147–1150. (Cité en page 189.)

Ferbinteanu, J. and Shapiro, M. L. (2003). Prospective and retrospective memory coding in the hippocampus. *Neuron*, 40(6):1227–1239. (Cité en page 189.)

Fiacco, A. V. and McCormick, G. P. (1990). *Nonlinear programming: sequential unconstrained minimization techniques*, volume 4. Siam. (Cité en page 42.)

Filliat, D., Kodjabachian, J., and Meyer, J.-A. (1999). Incremental evolution of neural controllers for navigation in a 6-legged robot. In *Proc. of the Fourth International Symposium on Artificial Life and Robots*. (Cité en page 51.)

Floreano, D., Mitri, S., Magnenat, S., and Keller, L. (2007). Evolutionary conditions for the emergence of communication in robots. *Current biology*, 17(6):514–519. (Cité en page 11.)

Floreano, D. and Mondada, F. (1994). Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In *Proceedings of the third international conference on Simulation of adaptive behavior: From Animals to Animats 3*, number LIS-CONF-1994-003, pages 421–430. MIT Press. (Cité en pages 11 et 20.)

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial intelligence through simulated evolution*. John Wiley. (Cité en page 15.)

Forrester, A. I. J. and Keane, A. J. (2009). Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45(1):50–79. (Cité en page 35.)

Foster, D. J. and Wilson, M. A. (2006). Reverse replay of behavioural sequences in hippocampal place cells during the awake state. *Nature*, 440(7084):680–683. (Cité en page 189.)

Frean, M. and Boyle, P. (2008). Using gaussian processes to optimize expensive functions. In *AI 2008: Advances in Artificial Intelligence*, pages 258–267. Springer. (Cité en page 44.)

Friedland, B. (2012). *Control system design: an introduction to state-space methods*. Courier Corporation. (Cité en page 146.)

Fuchs, a., Goldner, B., Nolte, I., and Schilling, N. (2014). Ground reaction force adaptations to tripedal locomotion in dogs. *Veterinary journal*, 201(3):307–15. (Cité en pages 3 et 91.)

Gais, S. and Born, J. (2004). Declarative memory consolidation: mechanisms acting during human sleep. *Learning & Memory*, 11(6):679–685. (Cité en pages 189 et 194.)

Galef, B. G. and Laland, K. N. (2005). Social learning in animals: empirical studies and theoretical models. *Bioscience*, 55(6):489–499. (Cité en pages 145 et 146.)

Geng, T., Porr, B., and Wörgötter, F. (2006). Fast biped walking with a sensor-driven neuronal controller and real-time online learning. *The International Journal of Robotics Research*, 25(3):243–259. (Cité en pages 25 et 34.)

Glynn, P. W. (1987). Likelilood ratio gradient estimation: an overview. In *Proceedings of the 19th conference on Winter simulation*, pages 366–375. ACM. (Cité en pages 30 et 31.)

Godzik, N., Schoenauer, M., and Sebag, M. (2003). Evolving symbolic controllers. In *Applications of Evolutionary Computing*, pages 638–650. Springer. (Cité en pages 11 et 53.)

Goldberg, K. and Chen, B. (2001). Collaborative control of robot motion: robustness to error. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 655–660. (Cité en page 91.)

Gomez, F. J. (2009). Sustaining diversity using behavioral information distance. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 113–120. ACM. (Cité en page 18.)

Görner, M. and Hirzinger, G. (2010). Analysis and evaluation of the stability of a biologically inspired, leg loss tolerant gait for six-and ei ght-legged walking robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4728–4735. (Cité en page 90.)

Graziano, M. (2006). The organization of behavioral repertoire in motor cortex. *Annual review of neuroscience*, 29(May):105–34. (Cité en page 190.)

Graziano, M. S. and Aflalo, T. N. (2007). Mapping behavioral repertoire onto the cortex. *Neuron*, 56(2):239–251. (Cité en page 190.)

Griffiths, T. L., Lucas, C., Williams, J., and Kalish, M. L. (2009). Modeling human function learning with gaussian processes. In *Advances in Neural Information Processing Systems 21 (NIPS)*, pages 553–560. (Cité en pages 35 et 114.)

Grillner, S. (2003). The motor infrastructure: from ion channels to neuronal networks. *Nature reviews. Neuroscience*, 4(July):573–586. (Cité en page 139.)

Gruau, F. (1994). Automatic definition of modular neural networks. *Adaptive behavior*, 3(2):151–183. (Cité en page 51.)

Gutierrez, J. M. P., Hinkley, T., Taylor, J. W., Yanev, K., and Cronin, L. (2014). Evolution of oil droplets in a chemorobotic platform. *Nature communications*, 5. (Cité en page 11.)

Haldane, J. (1932). *The causes of evolution.* Macmillan. (Cité en page 10.)

Hansen, N. (2006). The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer. (Cité en pages 15, 26 et 237.)

Hartland, C. and Bredeche, N. (2006). Evolutionary robotics, anticipation and the reality gap. In *Robotics and Biomimetics, 2006. ROBIO'06. IEEE International Conference on*, pages 1640–1645. IEEE. (Cité en page 96.)

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *Unsupervised learning.* Springer. (Cité en page 8.)

Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation.* Prentice Hall. (Cité en page 7.)

Hobson, J. A. and Pace-Schott, E. F. (2002). The cognitive neuroscience of sleep: neuronal systems, consciousness and learning. *Nature Reviews Neuroscience*, 3(9):679–693. (Cité en page 189.)

Hoffmann, M., Marques, H., Arieta, A., Sumioka, H., Lungarella, M., and Pfeifer, R. (2010). Body Schema in Robotics: A Review. *IEEE Transactions on Autonomous Mental Development*, 2(4):304–324. (Cité en page 92.)

Holland, O. and Goodman, R. (2003). Robots with internal models a route to machine consciousness? *Journal of Consciousness Studies*, 10(4-5):4–5. (Cité en page 92.)

Hoos, H. H. and Stützle, T. (2005). *Stochastic local search: Foundations and applications.* Morgan Kaufmann. (Cité en page 102.)

Hornby, G., Lohn, J. D., Linden, D. S., et al. (2011). Computer-automated evolution of an x-band antenna for nasa's space technology 5 mission. *Evolutionary computation*, 19(1):1–23. (Cité en page 11.)

Hornby, G., Takamura, S., Yamamoto, T., and Fujita, M. (2005). Autonomous evolution of dynamic gaits with two quadruped robots. *IEEE Transactions on Robotics*, 21(3):402–410. (Cité en pages 21, 25, 50, 51, 53, 54 et 129.)

Huber, R., Ghilardi, M. F., Massimini, M., and Tononi, G. (2004). Local sleep and learning. *Nature*, 430(6995):78–81. (Cité en page 194.)

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 4, pages 2588–2595. IEEE. (Cité en page 26.)

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review. *Neural Networks*, 21(4):642–653. (Cité en page 59.)

Ijspeert, A. J., Crespi, A., Ryczko, D., and Cabelguen, J.-M. (2007). From swimming to walking with a salamander robot driven by a spinal cord model. *science*, 315(5817):1416–1420. (Cité en page 51.)

Ito, M. (2008). Control of mental activities by internal models in the cerebellum. *Nature Reviews Neuroscience*, 9(4):304–313. (Cité en page 140.)

Jacobs, R., Jordan, M., Nowlan, S., and Hinton, G. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87. (Cité en page 53.)

Jakimovski, B. and Maehle, E. (2010). In situ self-reconfiguration of hexapod robot oscar using biologically inspired approaches. *Climbing and Walking Robots. In-Tech*. (Cité en page 90.)

Jakobi, N. (1998). Running across the reality gap: Octopod locomotion evolved in a minimal simulation. In *Evolutionary Robotics*, pages 39–58. Springer. (Cité en page 22.)

Jakobi, N., Husbands, P., and Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. *Proceedings of the European Conference on Artificial Life (ECAL)*, pages 704–720. (Cité en pages 22, 23, 95, 96 et 188.)

Jarvis, S. L., Worley, D. R., Hogy, S. M., Hill, A. E., Haussler, K. K., and Reiser II, R. F. (2013). Kinematic and kinetic analysis of dogs during trotting after amputation of a thoracic limb. *American journal of veterinary research*, 74(9):1155–1163. (Cité en pages 3 et 91.)

Jeannerod, M. (2001). Neural simulation of action: a unifying mechanism for motor cognition. *Neuroimage*, 14(1):S103–S109. (Cité en page 189.)

Jensen, M. T. (2005). Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation. *Journal of Mathematical Modelling and Algorithms*, 3(4):323–347. (Cité en page 16.)

Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70. (Cité en page 35.)

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492. (Cité en pages 26, 35 et 44.)

Kajita, S. and Espiau, B. (2008). *Handbook of Robotics*, chapter Legged Robots, pages 361–389. Springer. (Cité en page 90.)

Kim, H., Jordan, M. I., Sastry, S., and Ng, A. Y. (2003). Autonomous helicopter flight via reinforcement learning. In *Advances in neural information processing systems*, page None. (Cité en page 33.)

Kimura, H., Yamashita, T., and Kobayashi, S. (2001). Reinforcement learning of walking behavior for a four-legged robot. In *Proceedings of IEEE Conference on Decision and Control (CDC)*, volume 1, pages 411–416. IEEE. (Cité en pages 25, 34 et 54.)

Klaus, G., Glette, K., and Tørresen, J. (2012). A comparison of sampling strategies for parameter estimation of a robot simulator. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 173–184. (Cité en page 96.)

Kluger, J. and Lovell, J. (2006). *Apollo 13*. Mariner Books. (Cité en pages 3, 90 et 91.)

Knowles, J. D., Watson, R. A., and Corne, D. W. (2001). Reducing local optima in single-objective problems by multi-objectivization. In *Evolutionary multi-criterion optimization*, pages 269–283. Springer. (Cité en pages 16 et 18.)

Ko, J., Klein, D. J., Fox, D., and Haehnel, D. (2007). Gaussian processes and reinforcement learning for identification and control of an autonomous blimp. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 742–747. IEEE. (Cité en page 33.)

Kober, J., Bagnell, J. A., and Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, page 0278364913495721. (Cité en pages 4, 8, 9, 10, 26, 29, 30, 32, 91, 111, 123, 126 et 129.)

Kober, J., Mohler, B., and Peters, J. (2008). Learning perceptual coupling for motor primitives. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 834–839. IEEE. (Cité en page 33.)

Kober, J. and Peters, J. (2011). Policy search for motor primitives in robotics. *Machine Learning*, 84(1):171–203. (Cité en pages 26, 28 et 33.)

Kober, J., Wilhelm, A., Oztop, E., and Peters, J. (2012). Reinforcement learning to adjust parametrized motor primitives to new situations. *Autonomous Robots*, 33(4):361–379. (Cité en pages 45 et 151.)

Kodjabachian, J. and Meyer, J.-A. (1998). Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects. *Neural Networks, IEEE Transactions on*. (Cité en page 50.)

Kohavi, R. et al. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. (Cité en page 41.)

Kohl, N. and Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2619–2624. IEEE. (Cité en pages 25, 26, 29, 34, 51, 102, 126, 127, 128, 129, 239 et 240.)

Konidaris, G. and Barto, A. (2006). Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 489–496. ACM. (Cité en page 146.)

Koos, S., Cully, A., and Mouret, J.-B. (2013a). Fast damage recovery in robotics with the t-resilience algorithm. *The International Journal of Robotics Research*, 32(14):1700–1723. (Cité en pages 23, 25, 50, 51, 53, 54, 55, 93 et 129.)

Koos, S. and Mouret, J.-B. (2011). Online discovery of locomotion modes for wheel-legged hybrid robot s: a transferability-based approach. In *Proc. of CLAWAR*, pages 70–77. World Scientific Publishing Co. (Cité en page 96.)

Koos, S., Mouret, J.-B., and Doncieux, S. (2013b). The transferability approach: Crossing the reality gap in evolutionary robotics. *IEEE Trans. on Evolutionary Computation*, pages 122–145. (Cité en pages 22, 23, 48, 54, 55, 57, 92, 95, 96, 97, 184 et 188.)

Koos, S., Mouret, J.-B., and Doncieux, S. (2013c). The transferability approach: Crossing the reality gap in evolutionary robotics. *Evolutionary Computation, IEEE Transactions on*, 17(1):122–145. (Cité en page 129.)

Körding, K. P. and Wolpert, D. M. (2004). Bayesian integration in sensorimotor learning. *Nature*, 427(6971):244–247. (Cité en page 140.)

Koren, I. and Krishna, C. M. (2007). *Fault-tolerant systems*. Morgan Kaufmann. (Cité en pages 3 et 90.)

Kormushev, P., Calinon, S., and Caldwell, D. G. (2010). Robot motor skill coordination with em-based reinforcement learning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3232–3237. IEEE. (Cité en pages 26 et 33.)

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press. (Cité en page 15.)

Kuffner, J. J. and LaValle, S. M. (2000). Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE. (Cité en page 51.)

Kuindersma, S., Grupen, R., and Barto, A. (2011). Learning dynamic arm motions for postural recovery. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 7–12. IEEE. (Cité en page 45.)

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, pages 79–86. (Cité en page 31.)

Kupcsik, A., Deisenroth, M. P., Peters, J., Loh, A. P., Vadakkepat, P., and Neumann, G. (2014). Model-based contextual policy search for data-efficient generalization of robot skills. *Artificial Intelligence*. (Cité en pages 34, 145, 146, 148, 151 et 152.)

Kushner, H. J. (1964). A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Fluids Engineering*, 86(1):97–106. (Cité en pages 42 et 43.)

Lachenbruch, P. A. and Mickey, M. R. (1968). Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11. (Cité en page 41.)

Larranaga, P. and Lozano, J. A. (2002). *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media. (Cité en page 15.)

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press. (Cité en page 61.)

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444. (Cité en page 194.)

Lee, S., Yosinski, J., Glette, K., Lipson, H., and Clune, J. (2013). Evolving gaits for physical robots with the hyperneat generative encoding: the benefits of simulation. In *Applications of Evolutionary Computing*. Springer. (Cité en page 230.)

Lehman, J., Risi, S., D'Ambrosio, D., and Stanley, K. O. (2013). Encouraging reactivity to create robust machines. *Adaptive Behavior*, page 1059712313487390. (Cité en page 23.)

Lehman, J. and Stanley, K. (2011a). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2). (Cité en pages 4, 18, 19, 53, 62, 87 et 193.)

Lehman, J. and Stanley, K. (2011b). Evolving a diversity of virtual creatures through novelty search and local competition. In *Proc. of GECCO*, pages 211–218. ACM. (Cité en pages 11, 19, 55, 57, 59, 62 et 193.)

Lenski, R. E., Ofria, C., Collier, T. C., and Adami, C. (1999). Genome complexity, robustness and genetic interactions in digital organisms. *Nature*, 400(6745):661–664. (Cité en pages 11 et 22.)

Lewis, M. A., Fagg, A. H., and Bekey, G. (1994). Genetic algorithms for gait synthesis in a hexapod robot. *Recent trends in mobile robots*, pages 317–331. (Cité en page 20.)

Lewis, M. A., Fagg, A. H., and Solidum, A. (1992). Genetic programming approach to the construction of a neural network for control of a walking robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2618–2623. IEEE. (Cité en page 50.)

Lin, C.-M. and Chen, C.-H. (2007). Robust fault-tolerant control for a biped robot using a recurrent cerebellar model articulation controller. *Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(1):110–123. (Cité en page 91.)

Lipson, H. and Pollack, J. B. (2000). Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978. (Cité en pages 1, 11 et 21.)

Lizotte, D. J. (2008). *Practical bayesian optimization*. University of Alberta. (Cité en pages 43 et 128.)

Lizotte, D. J., Wang, T., Bowling, M. H., and Schuurmans, D. (2007). Automatic gait optimization with gaussian process regression. In *Proceedings of the the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 944–949. (Cité en pages 25, 35, 36, 45, 51, 114, 123, 126, 127, 129, 151 et 162.)

Louie, K. and Wilson, M. A. (2001). Temporally structured replay of awake hippocampal ensemble activity during rapid eye movement sleep. *Neuron*, 29(1):145–156. (Cité en page 189.)

Lungarella, M., Metta, G., Pfeifer, R., and Sandini, G. (2003). Developmental robotics: a survey. *Connection Science*, 15(4):151–190. (Cité en pages 187, 191 et 193.)

MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472. (Cité en page 39.)

Mahdavi, S. and Bentley, P. (2006). Innately adaptive robotics through embodied evolution. *Autonomous Robots*, 20(2):149–163. (Cité en pages 51, 93 et 129.)

Maquet, P., Laureys, S., Peigneux, P., Fuchs, S., Petiau, C., Phillips, C., Aerts, J., Del Fiore, G., Degueldre, C., Meulemans, T., et al. (2000). Experience-dependent changes in cerebral activation during human rem sleep. *Nature neuroscience*, 3(8):831–836. (Cité en page 189.)

Marescaux, J., Leroy, J., Gagner, M., Rubino, F., Mutter, D., Vix, M., Butner, S. E., and Smith, M. K. (2001). Transatlantic robot-assisted telesurgery. *Nature*, 413(6854):379–380. (Cité en page 1.)

Martinez-Cantin, R., de Freitas, N., Brochu, E., Castellanos, J., and Doucet, A. (2009). A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, 27(2):93–103. (Cité en pages 44 et 151.)

Martinez-Cantin, R., de Freitas, N., Doucet, A., and Castellanos, J. A. (2007). Active policy learning for robot planning and exploration under uncertainty. In *Robotics: Science and Systems*, pages 321–328. (Cité en pages 44 et 151.)

Matérn, B. et al. (1960). Spatial variation. stochastic models and their application to some problems in forest surveys and other sampling investigations. *Meddelanden fran statens Skogsforskningsinstitut*, 49(5). (Cité en page 39.)

Meltzoff, A. N. and Moore, M. K. (1997). Explaining facial imitation: A theoretical model. *Early Development & Parenting*, 6(3-4):179. (Cité en page 190.)

Mendel, G. (1865). *Experiments in plant hybridisation*. (Cité en page 10.)

Metropolis, N. and Ulam, S. (1949). The monte carlo method. *Journal of the American statistical association*, 44(247):335–341. (Cité en page 28.)

Metzinger, T. (2004). *Being no one: The self-model theory of subjectivity*. MIT Press. (Cité en page 92.)

Metzinger, T. (2007). Self models. *Scholarpedia*, 2(10):4174. (Cité en page 92.)

Meyer, J.-A. (1996). Artificial life and the animat approach to artificial intelligence. *Artificial intelligence*, pages 325–354. (Cité en page 1.)

Micchelli, C. A. and Pontil, M. (2004). Kernels for multi–task learning. In *Advances in Neural Information Processing Systems*, pages 921–928. (Cité en pages 145 et 153.)

Mitchell, T. M. (1997). *Machine learning. WCB*. McGraw-Hill Boston, MA:. (Cité en page 7.)

Mitri, S., Floreano, D., and Keller, L. (2009). The evolution of information suppression in communicating robots with conflicting interests. *Proceedings of the National Academy of Sciences*, 106(37):15786–15790. (Cité en page 11.)

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533. (Cité en pages 10 et 24.)

Mockus, J. (2013). *Bayesian approach to global optimization: theory and applications*. Kluwer Academic. (Cité en pages 35, 112 et 144.)

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2(117-129):2. (Cité en page 43.)

Mondada, F., Franzi, E., and Ienne, P. (1994). *Mobile robot miniaturisation: A tool for investigation in control algorithms*. Springer. (Cité en pages 20 et 22.)

Montanier, J.-M. and Bredeche, N. (2013). Evolution of altruism and spatial dispersion: an artificial evolutionary ecology approach. In *Advances in Artificial Life, ECAL*, volume 12, pages 260–267. (Cité en pages 11 et 22.)

Moore, G. E. (1975). Progress in digital integrated electronics. In *International Electron Devices Meeting*, volume 21, pages 11–13. IEEE. (Cité en page 92.)

Moriarty, D. E. and Miikkulainen, R. (1996). Evolving obstacle avoidance behavior in a robot arm. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 468–475. MIT Press Cambridge, MA. (Cité en page 21.)

Moser, E. I., Kropff, E., and Moser, M.-B. (2008). Place cells, grid cells, and the brain's spatial representation system. *Annu. Rev. Neurosci.*, 31:69–89. (Cité en page 189.)

Mostafa, K., Tsai, C., and Her, I. (2010). Alternative gaits for multiped robots with leg failures to retain maneuverability. *International Journal of Advanced Robotic Systems*, 7(4):31. (Cité en page 90.)

Mouret, J.-B. and Clune, J. (2015). Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*. (Cité en pages 54 et 76.)

Mouret, J.-B. and Doncieux, S. (2009). Using behavioral exploration objectives to solve deceptive problems in neuro-evolution. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 627–634. ACM. (Cité en page 18.)

Mouret, J.-B. and Doncieux, S. (2010). Sferes$_{v2}$: Evolvin' in the Multi-Core World. In *Proc. of IEEE CEC*, pages 4079–4086. (Cité en pages 59 et 104.)

Mouret, J.-B. and Doncieux, S. (2012). Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 20(1):91–133. (Cité en pages 18, 98, 101 et 102.)

Mouret, J.-B., Doncieux, S., and Meyer, J.-A. (2006). Incremental evolution of target-following neuro-controllers for flapping-wing animats. *From Animals to Animats 9*. (Cité en page 52.)

Mouret, J.-B., Koos, S., and Doncieux, S. (2012). Crossing the reality gap: a short introduction to the transferability approach. In *Proceedings of the workshop "Evolution in Physical Systems"*, ALIFE. (Cité en pages 23 et 97.)

Mumford, M. D. (2003). Where have we been, where are we going? taking stock in creativity research. *Creativity Research Journal*, 15(2-3):107–120. (Cité en page 3.)

Murphy, R. R. (2004). Trial by fire [rescue robots]. *Robotics & Automation Magazine, IEEE*, 11(3):50–61. (Cité en page 1.)

Murphy, R. R., Tadokoro, S., Nardi, D., Jacoff, A., Fiorini, P., Choset, H., and Erkmen, A. M. (2008). Search and rescue robotics. In *Springer Handbook of Robotics*, pages 1151–1173. Springer. (Cité en pages 1 et 2.)

Nagatani, K., Kiribayashi, S., Okada, Y., Otake, K., Yoshida, K., Tadokoro, S., Nishimura, T., Yoshida, T., Koyanagi, E., Fukushima, M., and Kawatsuma, S. (2013). Emergency response to the nuclear accident at the Fukushima Daiichi nuclear power plants using mobile rescue robots. *Journal of Field Robotics*, 30(1):44–63. (Cité en pages 1 et 2.)

Nelson, A., Barlow, G., and Doitsidis, L. (2009). Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370. (Cité en page 100.)

Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer. (Cité en page 33.)

Nguyen, A., Yosinski, J., and Clune, J. (2015). Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. (Cité en page 23.)

O'Hagan, A. and Kingman, J. (1978). Curve fitting and optimal design for prediction. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–42. (Cité en page 175.)

O'keefe, J. and Nadel, L. (1978). *The hippocampus as a cognitive map*, volume 3. Clarendon Press Oxford. (Cité en page 189.)

Oliveira, M. A. C., Doncieux, S., Mouret, J.-B., and Santos, C. P. (2013). Optimization of humanoid walking controller: Crossing the reality gap. In *Proceedings of Humanoids*. (Cité en pages 23 et 54.)

Oudeyer, P.-Y., Kaplan, F., and Hafner, V. V. (2007). Intrinsic motivation systems for autonomous mental development. *Evolutionary Computation, IEEE Transactions on*, 11(2):265–286. (Cité en pages 8, 192 et 193.)

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359. (Cité en pages 8 et 146.)

Papadimitriou, C. H. and Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. Courier Corporation. (Cité en pages 9 et 14.)

Pareto, V. (1896). *Cours d'économie politique*, volume 1. F. Rouge. (Cité en page 15.)

Parker, G. (2009). Punctuated anytime learning to evolve robot control for area coverage. *Design and Control of Intelligent Robotic Systems*, pages 255–277. (Cité en page 94.)

Peigneux, P., Laureys, S., Fuchs, S., Collette, F., Perrin, F., Reggers, J., Phillips, C., Degueldre, C., Del Fiore, G., Aerts, J., et al. (2004). Are spatial memories strengthened in the human hippocampus during slow wave sleep? *Neuron*, 44(3):535–545. (Cité en page 189.)

Peters, J., Mülling, K., and Altun, Y. (2010). Relative entropy policy search. In *AAAI*. (Cité en pages 32, 34 et 148.)

Peters, J. and Schaal, S. (2006). Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219–2225. IEEE. (Cité en pages 29 et 30.)

Peters, J. and Schaal, S. (2008a). Natural actor-critic. *Neurocomputing*, 71(7):1180–1190. (Cité en pages 29, 30, 31 et 33.)

Peters, J. and Schaal, S. (2008b). Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697. (Cité en pages 29 et 33.)

Peyrache, A., Khamassi, M., Benchenane, K., Wiener, S. I., and Battaglia, F. P. (2009). Replay of rule-learning related neural patterns in the prefrontal cortex during sleep. *Nature neuroscience*, 12(7):919–926. (Cité en page 189.)

Pfeifer, R. and Bongard, J. (2007). *How the body shapes the way we think: a new view of intelligence*. MIT press. (Cité en pages 1, 21 et 50.)

Pfeifer, R., Lungarella, M., and Iida, F. (2007). Self-organization, embodiment, and biologically inspired robotics. *science*, 318(5853):1088–1093. (Cité en pages 50 et 194.)

Pouget, A., Beck, J. M., Ma, W. J., and Latham, P. E. (2013). Probabilistic brains: knowns and unknowns. *Nature neuroscience*, 16(9):1170–1178. (Cité en page 140.)

Prassler, E. and Kosuge, K. (2008). *Handbook of Robotics*, chapter Domestic Robotics, pages 1253–1281. Springer. (Cité en page 90.)

Pratt, G. and Manzo, J. (2013). The darpa robotics challenge [competitions]. *Robotics & Automation Magazine, IEEE*, 20(2):10–12. (Cité en page 2.)

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1996). *Numerical recipes in C*, volume 2. Cambridge university press Cambridge. (Cité en pages 41 et 169.)

Pretorius, C., du Plessis, M., and Cilliers, C. (2012). Simulating robots without conventional physics: A neural network approach. *Journal of Intelligent & Robotic Systems*, pages 1–30. (Cité en page 96.)

Pugh, J. K., Soros, L., Szerlip, P. A., and Stanley, K. O. (2015). Confronting the challenge of quality diversity. In *Proceedings of the Annual conference on Genetic and evolutionary computation (GECCO)*. ACM. (Cité en page 79.)

Qu, Z., Ihlefeld, C. M., Jin, Y., and Saengdeejing, A. (2003). Robust fault-tolerant self-recovering control of nonlinear uncertain systems. *Automatica*, 39(10):1763–1771. (Cité en page 91.)

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source robot operating system. In *Proceedings of ICRA's workshop on Open Source Software*. (Cité en pages 52 et 227.)

Quiñonero-Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate gaussian process regression. *The Journal of Machine Learning Research*, 6:1939–1959. (Cité en page 152.)

Raibert, M., Blankespoor, K., Nelson, G., Playter, R., and the BigDog Team (2008). Bigdog, the rough-terrain quadruped robot. In *Proceedings of the 17th World Congress The International Federation of Automatic Control*. (Cité en page 1.)

Raibert, M. H. (1986). Legged robots. *Communications of the ACM*, 29(6):499–514. (Cité en pages 51 et 60.)

Rasmussen, C. E. (1996). *Evaluation of Gaussian processes and other methods for non-linear regression*. PhD thesis, Citeseer. (Cité en page 156.)

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press. (Cité en pages 33, 35, 36, 37, 39, 40, 41, 112, 114, 152, 155, 156 et 175.)

Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE. (Cité en pages 41 et 237.)

Rolf, M., Steil, J. J., and Gienger, M. (2010). Goal babbling permits direct learning of inverse kinematics. *Autonomous Mental Development, IEEE Transactions on*, 2(3):216–229. (Cité en page 191.)

Rückstiess, T., Sehnke, F., Schaul, T., Wierstra, D., Sun, Y., and Schmidhuber, J. (2010). Exploring parameter space in reinforcement learning. *Paladyn, Journal of Behavioral Robotics*, 1(1):14–24. (Cité en page 27.)

Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems. (Cité en page 9.)

Russell, S., Norvig, P., and Davis, E. (2010). *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, NJ. (Cité en pages 1, 7, 9 et 51.)

Sacks, J., Welch, W. J., Mitchell, T. J., Wynn, H. P., et al. (1989). Design and analysis of computer experiments. *Statistical science*, 4(4):409–423. (Cité en page 35.)

Saegusa, R., Metta, G., Sandini, G., and Sakka, S. (2009). Active motor babbling for sensorimotor learning. In *Robotics and Biomimetics, 2008. ROBIO 2008. IEEE International Conference on*, pages 794–799. IEEE. (Cité en page 191.)

Samuelsen, E. and Glette, K. (2014). Some distance measures for morphological diversification in generative evolutionary robotics. In *Proceedings of the 16th Annual conference on Genetic and evolutionary computation (GECCO)*. ACM. To appear. (Cité en page 50.)

Sanderson, K. (2010). Mars rover spirit (2003-10). *Nature*, 463(7281):600. (Cité en pages 2 et 144.)

Santello, M. (1998). Postural hand synergies for tool use. *The Journal of Neuroscience*, 18(23):10105–10115. (Cité en page 112.)

Saranli, U., Buehler, M., and Koditschek, D. (2001). Rhex: A simple and highly mobile hexapod robot. *The International Journal of Robotics Research*, 20(7):616–631. (Cité en pages 100 et 228.)

Schaal, S. (1999). Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242. (Cité en page 8.)

Schaal, S. (2003). Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *2nd International Symposium on Adaptive Motion of Animals and Machines*. (Cité en page 59.)

Schapire, R. E. (1990). The strength of weak learnability. *Machine learning*, 5(2):197–227. (Cité en page 53.)

Schleyer, G. and Russell, A. (2010). Adaptable gait generation for autotomised legged robots. In *Proceedings of Australasian Conference on Robotics and Automation (ACRA)*. (Cité en page 90.)

Schmidt, M. and Lipson, H. (2009). Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85. (Cité en pages 15 et 188.)

Schmitz, J., Dean, J., Kindermann, T., Schumm, M., and Cruse, H. (2001). A biologically inspired controller for hexapod walking: simple solutions by exploiting physical properties. *The biological bulletin*, 200(2):195–200. (Cité en pages 100 et 228.)

Schölkopf, B. and Smola, A. J. (2002). *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT press. (Cité en page 144.)

Seber, G. (1984). *Multivariate observations*, volume 41. Wiley New York. (Cité en pages 62 et 182.)

Secretan, J., Beato, N., D Ambrosio, D. B., Rodriguez, A., Campbell, A., and Stanley, K. O. (2008). Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1759–1768. ACM. (Cité en page 11.)

Seeger, M., Teh, Y.-W., and Jordan, M. (2005). Semiparametric latent factor models. Technical report. (Cité en pages 152 et 153.)

Shultz, T. R. and Rivest, F. (2000). Using knowledge to speed learning: A comparison of knowledge-based cascade-correlation and multi-task learning. In *ICML*, pages 871–878. (Cité en page 146.)

Siciliano, B. and Khatib, O. (2008). *Springer handbook of robotics*. Springer. (Cité en pages 51, 61, 117, 228 et 233.)

Simpson, T. W., Mauery, T. M., Korte, J. J., and Mistree, F. (1998). Comparison of response surface and kriging models for multidisciplinary design optimization. *American Institute of Aeronautics and Astronautics*, 98(7):1–16. (Cité en pages 35 et 44.)

Sims, K. (1991). *Artificial evolution for computer graphics*, volume 25. ACM. (Cité en page 11.)

Sims, K. (1994). Evolving 3d morphology and behavior by competition. *Artificial life*, 1(4):353–372. (Cité en page 11.)

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4):323–339. (Cité en page 145.)

Singh, S. P., Barto, A. G., Grupen, R., and Connolly, C. (1994). Robust reinforcement learning in motion planning. *Advances in neural information processing systems*, pages 655–655. (Cité en page 9.)

Skaggs, W. E. and McNaughton, B. L. (1996a). Replay of neuronal firing sequences in rat hippocampus during sleep following spatial experience. *Science*, 271(5257):1870–1873. (Cité en page 189.)

Skaggs, W. E. and McNaughton, B. L. (1996b). Theta phase precession in hippocampal. *Hippocampus*, 6:149–172. (Cité en page 189.)

Skinner, B. F. (1953). *Science and human behavior*. Simon and Schuster. (Cité en page 146.)

Smith, J. M. (1992). Evolutionary biology. byte-sized evolution. *Nature*, 355(6363):772–773. (Cité en pages 11 et 22.)

Smola, A. and Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems*, 9:155–161. (Cité en page 102.)

Smola, A. J. and Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222. (Cité en page 102.)

Snelson, E. and Ghahramani, Z. (2005). Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264. (Cité en page 152.)

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25 (NIPS)*, pages 2951–2959. (Cité en pages 35 et 37.)

Souza, J. R., Marchant, R., Ott, L., Wolf, D. F., and Ramos, F. (2014). Bayesian optimisation for active perception and smooth navigation. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 4081–4087. IEEE. (Cité en page 44.)

Sproewitz, A., Moeckel, R., Maye, J., and Ijspeert, A. (2008). Learning to move in modular robots using central pattern generators and online optimization. *The International Journal of Robotics Research*, 27(3-4):423–443. (Cité en pages 59, 129, 228 et 230.)

Srinivas, N., Krause, A., Kakade, S. M., and Seeger, M. (2009). Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*. (Cité en page 44.)

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212. (Cité en page 21.)

Stanley, K. O. and Lehman, J. (2015). *Why Greatness Cannot Be Planned*. Springer. (Cité en page 19.)

Stanley, K. O. and Miikkulainen, R. (1996). Efficient reinforcement learning through evolving neural network topologies. *Network (Phenotype)*, 1(2):3. (Cité en page 45.)

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127. (Cité en pages 11 et 14.)

Stein, M. L. (1999). *Interpolation of spatial data: some theory for kriging*. Springer. (Cité en page 39.)

Steingrube, S., Timme, M., Wörgötter, F., and Manoonpong, P. (2010). Self-organized adaptation of a simple neural circuit enables complex robot behaviour. *Nature Physics*, 6(3):224–230. (Cité en pages 100 et 228.)

Stickgold, R. (2005). Sleep-dependent memory consolidation. *Nature*, 437(7063):1272–1278. (Cité en page 194.)

Stickgold, R., Hobson, J. A., Fosse, R., and Fosse, M. (2001). Sleep, learning, and dreams: off-line memory reprocessing. *Science*, 294(5544):1052–1057. (Cité en page 194.)

Stickgold, R., Whidbee, D., Schirmer, B., Patel, V., and Hobson, J. A. (2000). Visual discrimination task improvement: A multi-step process occurring during sleep. *Journal of cognitive neuroscience*, 12(2):246–254. (Cité en page 194.)

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 111–147. (Cité en page 41.)

Stulp, F., Herlant, L., Hoarau, A., and Raiola, G. (2014). Simultaneous on-line discovery and improvement of robotic skill options. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1408–1413. IEEE. (Cité en page 33.)

Stulp, F. and Sigaud, O. (2012). Path integral policy improvement with covariance matrix adaptation. *arXiv preprint arXiv:1206.4621*. (Cité en pages 26 et 27.)

Stulp, F. and Sigaud, O. (2013). Robot skill learning: From reinforcement learning to evolution strategies. *Paladyn, Journal of Behavioral Robotics*, 4(1):49–61. (Cité en page 26.)

Sturm, J., Plagemann, C., and Burgard, W. (2008). Adaptive body scheme models for robust robotic manipulation. In *Robotics: Science and Systems*. (Cité en page 94.)

Sutton, R. S. and Barto, A. G. (1998a). *Introduction to Reinforcement Learning*. MIT Press. (Cité en pages 8 et 9.)

Sutton, R. S. and Barto, A. G. (1998b). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge. (Cité en page 9.)

Sutton, R. S., Barto, A. G., and Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *Control Systems, IEEE*, 12(2):19–22. (Cité en page 9.)

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y., et al. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063. Citeseer. (Cité en pages 10 et 24.)

Syswerda, G. (1991). A study of reproduction in generational and steady state genetic algorithms. *Foundations of genetic algorithms*, 2:94–101. (Cité en page 26.)

Tarapore, D. and Mouret, J.-B. (2014a). Comparing the evolvability of generative encoding schemes. In *Proceedings of ALife 14*, pages 55–62. MIT Press. (Cité en page 51.)

Tarapore, D. and Mouret, J.-B. (2014b). Evolvability signatures of generative encodings: beyond standard performance benchmarks. *arXiv preprint arXiv:1410.4985*. (Cité en page 51.)

Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *The Journal of Machine Learning Research*, 10:1633–1685. (Cité en pages 145 et 146.)

Taylor, M. E., Stone, P., and Liu, Y. (2007). Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167. (Cité en page 145.)

Tedrake, R., Zhang, T., and Seung, H. (2005). Learning to walk in 20 minutes. In *Proc. of Yale workshop on Adaptive and Learning Systems*. (Cité en pages 25, 34 et 51.)

Tesch, M., Schneider, J., and Choset, H. (2011). Using response surfaces and expected improvement to optimize snake robot gait parameters. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1069–1074. IEEE. (Cité en pages 25, 45, 123, 128 et 129.)

Tesch, M., Schneider, J., and Choset, H. (2013). Expensive multiobjective optimization for robotics. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 973–980. IEEE. (Cité en page 45.)

Theodorou, E., Buchli, J., and Schaal, S. (2010). A generalized path integral control approach to reinforcement learning. *The Journal of Machine Learning Research*, 11:3137–3181. (Cité en pages 28 et 32.)

Thorndike, E. L. and Woodworth, R. S. (1901). The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review*, 8(4):384. (Cité en page 145.)

Thrun, S. (1996). Is learning the n-th thing any easier than learning the first? *Advances in neural information processing systems*, pages 640–646. (Cité en pages 8 et 146.)

Thrun, S., Burgard, W., Fox, D., et al. (2005). *Probabilistic robotics*. MIT press Cambridge. (Cité en page 252.)

Thrun, S. and Pratt, L. (1998). *Learning to learn*. Kluwer Academic Publishers. (Cité en pages 145 et 146.)

Thrun, S. B. (1992). Efficient exploration in reinforcement learning. Technical report. (Cité en page 9.)

Titsias, M. K. (2009). Variational learning of inducing variables in sparse gaussian processes. In *International Conference on Artificial Intelligence and Statistics*, pages 567–574. (Cité en page 152.)

Toffolo, A. and Benini, E. (2003). Genetic diversity as an objective in multi-objective evolutionary algorithms. *Evolutionary Computation*, 11(2):151–167. (Cité en pages 98 et 101.)

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460. (Cité en page 1.)

Ueno, T., Nakamura, Y., Takuma, T., Shibata, T., Hosoda, K., and Ishii, S. (2006). Fast and stable learning of quasi-passive dynamic walking by an unstable biped robot based on off-policy natural actor-critic. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 5226–5231. IEEE. (Cité en page 33.)

Valsalam, V. K. and Miikkulainen, R. (2008). Modular neuroevolution for multi-legged locomotion. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 265–272. ACM. (Cité en page 51.)

Veiga, F. and Bernardino, A. (2013). Active tactile exploration for grasping. In *ICRA Workshop on Autonomous Learning*. (Cité en page 44.)

Verma, V., Gordon, G., Simmons, R., and Thrun, S. (2004). Real-time fault diagnosis. *Robotics & Automation Magazine*, 11(2):56–66. (Cité en page 90.)

Visinsky, M., Cavallaro, J., and Walker, I. (1994). Robotic fault detection and fault tolerance: A survey. *Reliability Engineering & System Safety*, 46(2):139–158. (Cité en pages 3 et 90.)

Vogeley, K., Kurthen, M., Falkai, P., and Maier, W. (1999). Essential functions of the human self model are implemented in the prefrontal cortex. *Consciousness and cognition*, 8(3):343–63. (Cité en page 92.)

Von Hofsten, C. (2004). An action perspective on motor development. *Trends in cognitive sciences*, 8(6):266–272. (Cité en page 191.)

Wagner, U., Gais, S., Haider, H., Verleger, R., and Born, J. (2004). Sleep inspires insight. *Nature*, 427(6972):352–355. (Cité en pages 140 et 194.)

Waibel, M., Floreano, D., and Keller, L. (2011). A quantitative test of hamilton's rule for the evolution of altruism. *PLoS-Biology*, 9(5):970. (Cité en pages 11 et 22.)

Walker, M. P., Brakefield, T., Morgan, A., Hobson, J. A., and Stickgold, R. (2002). Practice with sleep makes perfect: sleep-dependent motor skill learning. *Neuron*, 35(1):205–211. (Cité en page 194.)

Warwick, K. and Shah, H. (2015). Can machines think? a report on turing test experiments at the royal society. *Journal of Experimental & Theoretical Artificial Intelligence*, (ahead-of-print):1–19. (Cité en page 1.)

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292. (Cité en page 9.)

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, University of Cambridge England. (Cité en page 9.)

Watson, R. A., Ficici, S. G., and Pollack, J. B. (2002). Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Robotics and Autonomous Systems*, 39(1):1–18. (Cité en page 22.)

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1994). *Genetic reinforcement learning for neurocontrol problems*. Springer. (Cité en page 21.)

Whitley, D., Starkweather, T., and Bogart, C. (1990). Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14(3):347–361. (Cité en page 11.)

Williams, C., Klanke, S., Vijayakumar, S., and Chai, K. M. (2009). Multi-task gaussian process learning of robot inverse dynamics. In *Advances in Neural Information Processing Systems*, pages 265–272. (Cité en pages 145 et 154.)

Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In *Advances in Neural Information Processing Systems 8*, pages 514–520. MIT press. (Cité en page 151.)

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256. (Cité en pages 27, 28, 30 et 31.)

Wilson, D. (1966). Insect walking. *Annual Review of Entomology*, 11(1):103–122. (Cité en pages 100 et 228.)

Wilson, M. (2002). Six views of embodied cognition. *Psychonomic bulletin & review*, 9(4):625–636. (Cité en page 50.)

Wilson, M. A. and McNaughton, B. L. (1994). Reactivation of hippocampal ensemble memories during sleep. *Science*, 265(5172):676–679. (Cité en page 189.)

Wischmann, S., Floreano, D., and Keller, L. (2012). Historical contingency affects signaling strategies and competitive abilities in evolving populations of simulated robots. *Proceedings of the National Academy of Sciences*, 109(3):864–868. (Cité en page 11.)

Wolpert, D. M., Ghahramani, Z., and Flanagan, J. R. (2001). Perspective and Problems in Motor Learning. *Trends in Cognitive Sciences*, 5(11):487–494. (Cité en pages 111 et 139.)

Xu, R., Wunsch, D., et al. (2005). Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678. (Cité en page 8.)

Yeung, D.-Y. and Zhang, Y. (2009). Learning inverse dynamics by gaussian process regression under the multi-task learning framework. In *The Path to Autonomous Robots*, pages 1–12. Springer. (Cité en page 154.)

Yoerger, D. R. (2008). Underwater robotics. In *Springer handbook of robotics*, pages 987–1008. Springer. (Cité en page 144.)

Yosinski, J., Clune, J., Hidalgo, D., Nguyen, S., Zagal, J., and Lipson, H. (2011). Evolving Robot Gaits in Hardware: the HyperNEAT Generative Encoding Vs. Parameter Optimization. *Proc. of ECAL*. (Cité en pages 21, 25, 50, 51, 59, 106, 129, 228 et 230.)

Yu, K., Tresp, V., and Schwaighofer, A. (2005). Learning gaussian processes from multiple tasks. In *Proceedings of the 22nd international conference on Machine learning*, pages 1012–1019. ACM. (Cité en page 145.)

Zagal, J., Delpiano, J., and Ruiz-del Solar, J. (2009). Self-modeling in humanoid soccer robots. *Robotics and Autonomous Systems*, 57(8):819–827. (Cité en pages 94 et 188.)

Zagal, J., Ruiz-del Solar, J., and Vallejos, P. (2004). Back to reality: Crossing the reality gap in evolutionary robotics. In *Proceedings of IFAC Symposium on Intelligent Autonomous Vehicles (IAV)*. (Cité en page 95.)

Zykov, V. (2008). *Morphological and behavioral resilience against physical damage for robotic systems*. PhD thesis, Cornell University. (Cité en page 92.)

Zykov, V., Bongard, J., and Lipson, H. (2004). Evolving dynamic gaits on a physical robot. In *Proceedings of Genetic and Evolutionary Computation Conference, Late Breaking Paper (GECCO)*, volume 4. (Cité en pages 11, 21, 25 et 51.)

Appendix A

The Hexapod Experiments

A.1 The Hexapod Robot

Physical robot The robot is a 6-legged robot with 3 degrees of freedom (DOFs) per leg (see Fig. A.1). Each DOF is actuated by position-controlled servos (MX-28 Dynamixel actuators manufactured by Robotis). The first servo controls the horizontal (front-back) orientation of the leg and the two others control its elevation. An RGB-D camera (Xtion, from ASUS) is fixed on top of the robot. Its data are used to estimate the forward displacement of the robot via an RGB-D SLAM algorithm[1] (Dryanovski et al., 2013) from the robot operating system (ROS) framework[2] (Quigley et al., 2009).

Simulator The simulator is a dynamic physics simulation of the undamaged 6-legged robot on flat ground (Fig. 4.10). We weighted each segment of the leg and the body of the real robot, and we used the same masses for the simulations. The simulator is based on the Open Dynamics Engine (ODE, `http://www.ode.org`).

A.2 The Hexapod Genotypes and Controllers

A.2.1 The first version (24 parameters)

Genotype and parametrized controller The genotype is a set of 24 parameter values defining the angular position of each leg joint with a periodic function γ of time t, parametrized by an amplitude α and a phase shift ϕ (Fig. 3.2, right):

$$\gamma(t,\ \alpha,\ \phi) = \alpha \cdot \tanh\left(4 \cdot \sin\left(2 \cdot \pi \cdot (t + \phi)\right)\right) \tag{A.1}$$

Angular positions are updated and sent to the servos every 30ms. The main feature of this particular function is that the control signal is constant during a large portion of each cycle, thus allowing the robot to stabilize itself. In order to keep the "tibia" of each leg vertical, the control signal of the third servo is the opposite of the second one. Consequently, positions sent to the i^{th} leg are:

- $\gamma(t,\ \alpha_1^i,\ \phi_1^i)$ for servo 1;
- $\gamma(t,\ \alpha_2^i,\ \phi_2^i)$ for servos 2;

[1] `http://wiki.ros.org/ccny_openni_launch`
[2] `http://www.ros.org`

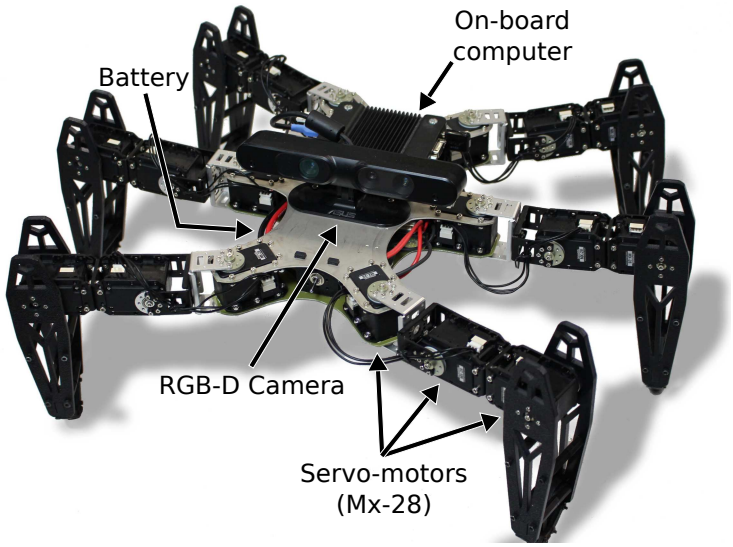


Figure A.1: The hexapod robot, which is used to evaluate the performance of the algorithms proposed in this manuscript.

- $-\gamma(t,\ \alpha_2^i,\ \phi_2^i)$ for servos 3.

The 24 parameters can each have five different values $(0, 0.25, 0.5, 0.75, 1)$ and with their variations, numerous gaits are possible, from purely quadruped gaits to classic tripod gaits.
This controller is designed to be as simple as possible so that we can show the performance of proposed algorithm in a straightforward setup. Nevertheless, our algorithms do not put any constraint on the type of controllers and many other controllers are conceivable (e.g. bio-inspired central pattern generators like Sproewitz et al. (2008) or evolved neural networks like in (Yosinski et al., 2011; Clune et al., 2011)).

Reference controller Our reference controller is a classic tripod gait (Siciliano and Khatib, 2008; Wilson, 1966; Saranli et al., 2001; Schmitz et al., 2001; Ding et al., 2010; Steingrube et al., 2010). It involves two tripods: legs 1-4-5 and legs 2-3-5 (Fig. 4.10). This controller is designed to always keep the robot balanced on at least one of these tripods. The walking gait is achieved by lifting one tripod, while the other tripod pushes the robot forward (by shifting itself backward). The lifted tripod is then placed forward in order to repeat the cycle with the other tripods. This gait is static, fast, and similar to insect gaits (Wilson, 1966; Delcomyn, 1971). Table SA.1 shows the 24 parameters of the reference controller. The amplitude orientation parameters (α_{i_1}) are set to 1 to produce the fastest possible gait, while the amplitude elevation parameters (α_{i_2}) are set to a small value (0.25) to keep the gait stable. The phase elevation parameters (ϕ_{i_2}) define two tripods: 0.25 for legs 2-3-5; 0.75 for legs 1-4-5. To achieve a cyclic motion of the leg, the phase orientation values (ϕ_{i_1}) are chosen by subtracting 0.25 to the phase elevation values

Table A.1: Parameters of the reference controller.

Leg number		1	2	3	4	5	6
	α_{i_1}	1.00	1.00	1.00	1.00	1.00	1.00
First joint	ϕ_{i_1}	0.00	0.00	0.50	0.50	0.00	0.00
	α_{i_2}	0.25	0.25	0.25	0.25	0.25	0.25
Two last joints	ϕ_{i_2}	0.75	0.25	0.25	0.75	0.75	0.25

(ϕ_{i_2}), plus a 0.5 shift for legs 1-3-5, which are on the left side of the robot. The actual speed of the reference controller is not important for the comparisons made in this manuscript: it is simply intended as a reference and to show that the performance of classic, hand-programmed gaits tend to fail when damage occurs.

Random variation of controller's parameters For the genotype mutation, each parameter value has a 10% chance of being changed to any value in the set of possible values, with the new value chosen randomly from a uniform distribution over the possible values. For all the experiments, the crossover is disabled.

A.2.2 The second version (36 parameters)

Genotype and parametrized controller The angular position of each degree of freedom is governed by a periodic function γ parametrized by its amplitude α, its phase ϕ, and its duty cycle τ (the duty cycle is the proportion of one period in which the joint is in its higher position). The function is defined with a square signal of frequency 1Hz, with amplitude α, and duty cycle τ. This signal is then smoothed via a Gaussian filter in order to remove sharp transitions, and is then shifted according to the phase ϕ.
Angular positions are sent to the servos every 30 ms. In order to keep the "tibia" of each leg vertical, the control signal of the third servo is the opposite of the second one. Consequently, angles sent to the i^{th} leg are:

- $\gamma(t,\ \alpha_{i_1},\ \phi_{i_1},\ \tau_{i_1})$ for DOF 1
- $\gamma(t,\ \alpha_{i_2},\ \phi_{i_2},\ \tau_{i_2})$ for DOF 2
- $-\gamma(t,\ \alpha_{i_2},\ \phi_{i_2},\ \tau_{i_2})$ for DOF 3

This controller makes the robot equivalent to a 12 DOF system, even though 18 motors are controlled.
There are 6 parameters for each leg (α_{i_1}, α_{i_2}, ϕ_{i_1}, ϕ_{i_2}, τ_{i_1}, τ_{i_2}), therefore each controller is fully described by 36 parameters. Each parameter can have one of these possible values: 0, 0.05, 0.1, ... 0.95, 1. Different values for these 36 parameters can produce numerous different gaits, from purely quadruped gaits to classic tripod gaits.
This controller is designed to be simple enough to show the performance of the algorithms in an intuitive setup. Nevertheless, our algorithm will work with any

Table A.2: Parameters of the reference controller.

Leg number		1	2	3	4	5	6
First joint	α_{i_1}	1.00	1.00	1.00	1.00	1.00	1.00
	ϕ_{i_1}	0.00	0.00	0.50	0.50	0.00	0.00
	τ_{i_1}	0.5	0.5	0.5	0.5	0.5	0.5
Two last joints	α_{i_2}	0.25	0.25	0.25	0.25	0.25	0.25
	ϕ_{i_2}	0.75	0.25	0.25	0.75	0.75	0.25
	τ_{i_2}	0.5	0.5	0.5	0.5	0.5	0.5

type of controller, including bio-inspired central pattern generators (Sproewitz et al., 2008) and evolved neural networks (Yosinski et al., 2011; Clune et al., 2011, 2009; Lee et al., 2013).

Reference controller With this version of the controller, our reference controller is the same as before but with all the duty cycle parameters (τ_i) set to 0.5 so that the motors spend the same proportion of time in their two limit angles. The corresponding parameters are given in Table A.1.

Random variation of controller's parameters Each parameter of the controller has a 5% chance of being changed to any value in the set of possible values, with the new value chosen randomly from a uniform distribution over the possible values.

Appendix B

The Robotic Arm experiments

B.1 The Robotic Arm: First setup

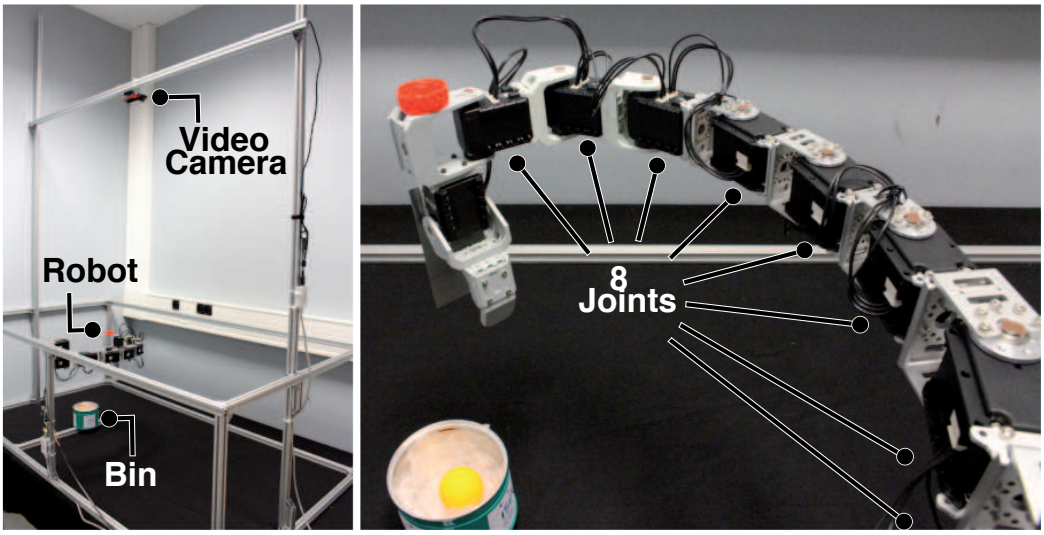


Figure B.1: The second robotic arm experimental setup.

Physical Robot The physical robot is a planar robotic arm with 8 degrees of freedom (Fig. B.1) and a 1-degree-of-freedom gripper. The robot has to release a ball into a bin (a variant of the classic "pick and place" task in industrial robotics). To assess the position of the gripper, a red cap, placed on top of the gripper, is tracked with a video camera. The visual tracking is achieved with the "cmvision" ROS package, which tracks colored blobs (`http://wiki.ros.org/cmvision`). The transformation between the robot's frame and the camera frame is known and provided to the algorithm to compute the position of the cap in the robot's frame. The eight joints of the robot are actuated by position-controlled servos manufactured by Dynamixel. To maximize the reliability of the the arm, the type of servo is not the same for all the joints: heavy-duty servos are used near the base of the robot and lighter ones are used for the end of the arm. The first joint, fixed to the base, is moved by two MX-28 servos mounted in parallel. The second joint is moved by an MX-64 servo. The 3 subsequent servos are single MX-28s, and the 3 remaining servos are AX-18s. All the robot's joints are limited to a motion range

of $\pm\pi/2$.

Simulator The generation of the behavior-performance map is made with a simulated robot in the same way as for the hexapod experiment. For consistency with the simulated hexapod experiments, we used the dynamic (as opposed to kinematic) version of the simulator, based on the ODE library. Any joint configuration that resulted in the arm colliding with itself was not added to the map.

B.2 The Robotic Arm: Second setup

Physical Robot In the second setup of the robotic arm experiments, the robot is identical to the one used in the fist setup (see appendix B.1), only the position of the camera differs. In this setup, the camera is placed in an arbitrary position which is unknown by the robot and the algorithm. Moreover, instead of tracking a red cap, like in the initial setup, the camera tracks the red dot of a laser pointer placed on the tip of the gripper (Fig. B.2).
This setup has been imagined to release several assumption usually made in visual-servoing works (Espiau et al., 1992) and to show that learning strategies can be used for this kind of task. First, the transformation between the robot's frame and the camera frame is not provided. Second, the camera does not track the robot itself, but only the red dot projected on the ground by the laser pointer. This projection adds another unknown transformation that the algorithm as to deal with and which can be sometimes not strictly linear (uneven ground, dot not visible). Moreover, no assumption about the camera's quality is made, meaning that only the position of the dot at the end of the movement is assess and not its trajectory. This constraint prevent the algorithms from using most of gradient based approaches.

Simulator The simulator used for this second setup is identical to the one used in the fist setup (see appendix B.1). In order to simulate the projection of the camera placed in an arbitrary position, 3 random angles $(\theta_r, \theta_p, \theta_y)$ are used to compute a roll/pitch/yaw rotation matrix:

$$rot = \begin{bmatrix} c\theta_y * c\theta_p & -s\theta_y * c\theta_r + c\theta_y * s\theta_p * s\theta_r & s\theta_y * s\theta_r + c\theta_y * s\theta_p * c\theta_r \\ s\theta_y * c\theta_p & c\theta_y * c\theta_r + s\theta_y * s\theta_p * s\theta_r & -c\theta_y * s\theta_r + c\theta_y * s\theta_p * c\theta_r \\ -s\theta_p & c\theta_p * s\theta_r & c\theta_p * c\theta_r \end{bmatrix} \tag{B.1}$$

Where c and s stands for cos() and sin() respectively. Because the laser dot is projected on the ground, the z coordinate of its position is always equal to zero and because a camera is also a projection of the depth component, the previous matrix can be reduced to its first 2x2 block:

$$rot = \begin{bmatrix} \cos(\theta_y) * \cos(\theta_p) & -\sin(\theta_y) * \cos(\theta_r) + \cos(\theta_y) * \sin(\theta_p) * \sin(\theta_r) \\ \sin(\theta_y) * \cos(\theta_p) & \cos(\theta_y) * \cos(\theta_r) + \sin(\theta_y) * \sin(\theta_p) * \sin(\theta_r) \end{bmatrix} \tag{B.2}$$

The meter/pixel conversion is achieved by adding a scale factor before the matrix that convert 1 meter into 1500 pixels.

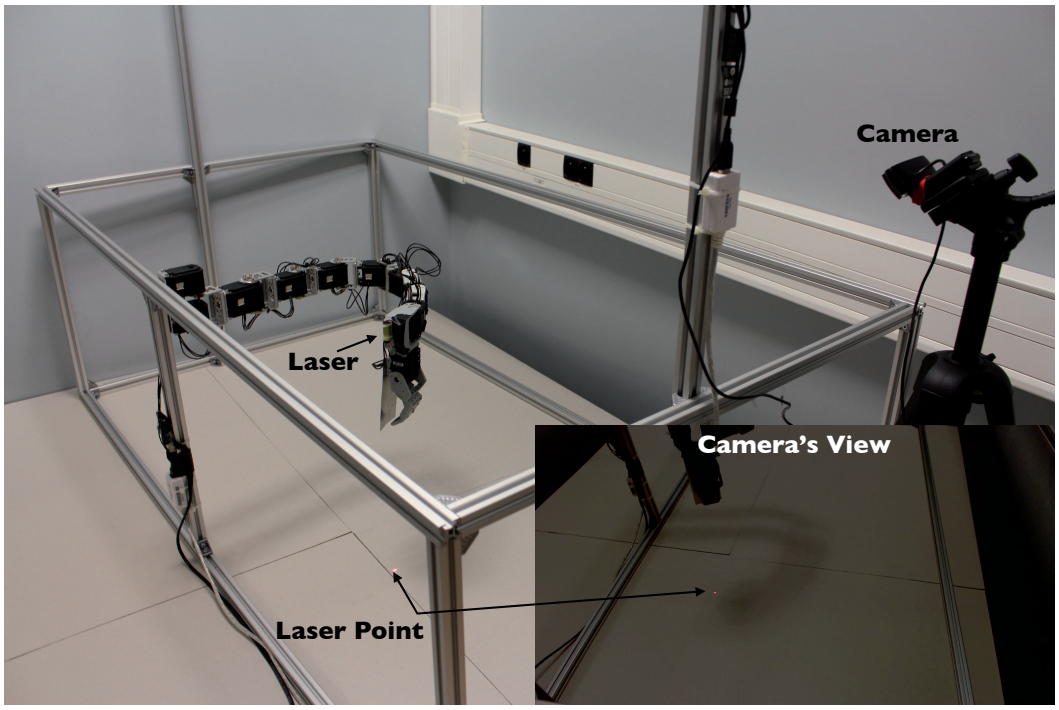


Figure B.2: The second robotic arm experimental setup.

B.3 The Arm Controller

Parametrized controller The controller defines the target position for each joint. The controller is thus parametrized by eight continuous values from 0 to 1 describing the angle of each joint, which is mapped to the the total motion range of each joint of $\pm\pi/2$. The 8 joints are activated simultaneously and are driven to their target position by internal PID controllers.
We chose this simple control strategy to make the experiments easy to reproduce and highlight the contribution of Intelligent Trial and Error for damage recovery. More advanced control strategies, for instance visual servoing (Siciliano and Khatib, 2008), would be more realistic in a industrial environment, but they would have made it hard to analyze the experimental results because both Intelligent Trial and Error and the controller would compensate for damage at the same time.

Randomly varying the controller's parameters Each parameter of the controller (section "Parametrized controller") has a 12.5% chance of being changed to any value from 0 to 1, with the new value chosen from a polynomial distribution as described on p. 124 of (Deb, 2000), with $\eta_m = 10.0$.

Appendix C

Parameters values used in the experiments

C.1 TBR-Evolution experiments

- Parameters used for the experiments on the virtual robot:
  - TBR-Evolution, Novelty Search and NS with Local Competition experiments:
    * Population size: 100 individuals
    * Number of generations: 10 000
    * Mutation rate: 10% on each parameters
    * Crossover: disabled
    * ρ: 0.10 m
    * ρ variation: none
    * k:15
  - "Nearest" and "Orientation" control experiments:
    * Population size : 100 individuals
    * Number of generations : 50 0000 (100 * 500)
    * Mutation rate : 10% on each parameters
    * Crossover : disabled
- Parameters used for the experiments on the physical robot:
  - TBR-Evolution and the control experiment:
    * Population size: 100 individuals
    * Number of generations: 3 000 generations
    * Mutation: 10% on each parameters
    * Crossover: disabled
    * ρ: 0.10 m
    * ρ variation: none
    * Transfer period: 50 iterations
    * τ: -0.05 m

C.2 T-Resilience experiments

- Population size: 100 individuals
- Number of generations: 1000
- Transfer frequency: every 40 generations
- Evaluation duration: 3 seconds
- Transferability threshold: 0.1 meter

C.3 Intelligent Trial and Error experiments

C.3.1 Experiments with the hexapod robot

Main parameters of MAP-Elites

- parameters in controller: 36
- parameter values (controller): 0 to 1, with 0.05 increments
- size of behavioral space: 6
- possible behavioral descriptors: $\{0, 0.25, 0.5, 0.75, 1\}$
- iterations: 40 million

Main parameters of M-BOA

- σ^2_{noise}: 0.001
- α: 0.9
- ρ: 0.4
- κ: 0.05

C.3.2 Experiments with the robotic arm

Main parameters of MAP-Elites

- parameters in controller: 8
- controller parameter values: 0 to 1 (continuous)
- dimensions in the behavioral space: 2
- simulated evaluations to create the behavior-performance map: 20 million

Main parameters of M-BOA

- σ^2_{noise}: 0.03
- ρ: 0.1
- κ: 0.3

C.4 State-Based BO with Transfer, Priors and blacklists

For all the experiments presented in chapter 5, we used the RPROP algorithm (Riedmiller and Braun, 1993) to optimize the log-likelihood of the model and the CMAES algorithm (Hansen, 2006) to find the maximum in the acquisition function. The values of the algorithm's parameters are the same for all the experiments and are defined as:

- Kernel noise parameter: $\sigma^2_{noise} = 0.01$
- Max number of iteration per target: 150
- Performance threshold to reach a target: 50px
  (changed to 75 for experiments in reality)
- Number of initial random samples: 10
- UCB exploration parameter: $\kappa = 20$
- Number of RPROP iterations : 300
- Number of RPROP restarts: 10

Appendix D

Other

D.1 Appendix for the T-resilience experiments

D.1.1 Implementation details for reference experiments

D.1.1.1 Local search

Our implementation of the local search (algorithm 6) starts from a randomly generated initial controller. A random perturbation c' is derived from the current best controller c. The controller c' is next tested on the robot for 3 seconds and the corresponding performance value $\mathscr{F}_{real}(c')$ is estimated with a SLAM algorithm using the RGB-D camera. If c' performs better than c, c is replaced by c', else c is kept. For both the stochastic local search and the policy gradient method (section D.1.1.2), a random perturbation c' from a controller c is obtained as follows:

- each parameter c'_j is obtained by adding to c_j a random deviation δ_j, uniformely picked up in $\{-0.25,\ 0,\ 0.25\}$;
- if c'_j is greater (resp. lower) than 1 (resp. 0), it takes the value 1 (resp. 0).

The process is iterated during 20 minutes to match the median duration of the T-Resilience (Table D.1; variant *time*). For comparison, the best controller found after 25 real tests is also kept (variant *tests*).

Algorithm 6 Stochastic local search (T real tests)

$c \leftarrow$ random controller

for $i = 1 \rightarrow T$ **do**
  $c' \leftarrow$ random perturbation of c
  if $\mathscr{F}_{real}(c') > \mathscr{F}_{real}(c)$ **then**
    $c \leftarrow c'$

new controller: c

D.1.1.2 Policy gradient method

Our implementation of the policy gradient method is based on Kohl and Stone (2004) (algorithm 7). It starts from a randomly generated controller c. At each iteration, 15 random perturbations c'^i from this controller are tested for 3 seconds on the robot and their performance values are estimated with the SLAM algorithm,

using the RGB-D camera. The number of random perturbations (15) is the same as in (Kohl and Stone, 2004), in which only 12 parameters have to be found. For each control parameter j, the average performance $A_{+,j}$ (resp. A or $A_{-,j}$) of the controllers whose parameter value c'^i_j is greater than (resp. equal to or less than) the value of c_j is computed. If A is not greater than both $A_{+,j}$ and $A_{-,j}$, the control parameter c_j is modified as follows:

- c_j is increased by 0.25, if $A_{+,j} > A_{-,j}$ and $c_j < 1$;
- c_j is decreased by 0.25, if $A_{-,j} > A_{+,j}$ and $c_j > 0$.

Once all the control parameters have been updated, the newly generated controller c is used to start a new iteration of the algorithm.
The whole process is iterated 4 times (i.e. 60 real tests; variant *tests*) with a median duration of 24 minutes to match the median duration of the T-Resilience (Table D.1). For comparison, the best controller found after 2 iterations (i.e. 30 real tests; variant *time*) is also kept.

Algorithm 7 Policy gradient method ($T \times S$ real tests)

$c \leftarrow$ random controller

for $i = 1 \rightarrow T$ **do**
  $\{c'^1, c'^2, \ldots, c'^S\} \leftarrow S$ random perturbations of c
  for $j = 1 \rightarrow S$ **do**
    $A_{0,j}$ = average of $\mathscr{F}_{real}(c'^i)$ for c'^i such as $c'^i_j = c_j$
    $A_{+,j}$ = average of $\mathscr{F}_{real}(c'^i)$ for c'^i such as $c'^i_j > c_j$
    $A_{-,j}$ = average of $\mathscr{F}_{real}(c'^i)$ for c'^i such as $c'^i_j < c_j$
    if $A_{0,j} > \max(A_{+,j}, A_{-,j})$ **then**
      c_j remains unchanged
    else
      if $A_{+,j} > A_{-,j}$ **then**
        $c_j = \min(c_j + 0.25,\ 1)$
      else
        $c_j = \max(c_j - 0.25,\ 0)$

new controller: c

D.1.1.3 Self-modeling process (Bongard's algorithm)

Our implementation of the self-modeling process is based on Bongard et al. (2006) (algorithm 8). Unlike the implementation of Bongard et al. (2006), we use internal measurements to assess the consequences of actions. This measure is performed with a 3-axis accelerometer (ADXL345) placed at the center of the robot, thus allowing the robot to measure its orientation.

Algorithm 8 Self-modeling approach (T real tests)

$pop_{model} \leftarrow \{m^1, m^2, \ldots, m^{S_{model}}\}$ (randomly generated or not)
empty training set of actions Ω

for $i = 1 \rightarrow T$ **do**
  selection of the action which maximises variance of predictions in pop_{model}
  execution of the action on the robot
  recording of robot's orientation based on internal measurements
  addition of the action to the training set Ω
  N_{model} iterations of MOEA on pop_{model} evaluated on Ω

selection of the new self-model
$pop_{ctrl} \leftarrow \{c^1, c^2, \ldots, c^{S_{ctrl}}\}$ (randomly generated)

N_{ctrl} iterations of MOEA on pop_{ctrl} in the self-model

selection of the new controller in the Pareto front

Robot's model. The self-model of the robot is a dynamic simulation of the hexapod built with the Open Dynamics Engine (ODE); it is the same model as the one used for T-Resilience experiments. However this self-model is parametrized in order to discover some damages or morphological modifications. For each leg of the robot, the algorithm has to find optimal values for 5 parameters:

- length of middle part of the leg (float)
- length of the terminal part of the leg (float)
- activation of the first actuator (boolean)
- activation of the second actuator (boolean)
- activation of the last actuator (boolean)

The length parameters have 6 different values: {0, 0.5, 0.75, 1, 1.25, 1.5}, which represents a scale factor with respect to the original size. If the length parameter of one part is zero, the part is deleted in the simulation and all other parts only attached to it are deleted too. We therefore have a model with 30 parameters.

Action set. As advised by Bongard (2007) (variant II), we use a set of actions where each action uses only one leg. The first servo has 2 possible positions (1,2): $-\pi/6$ and $\pi/6$. For each of these two positions, we have 3 possible actions (a,b,c) as shown on Figure D.1. There are consequently 6 possible actions for each leg, that is, 36 actions in total.

Parameters. A population of 36 models is evolved during 2000 generations. The initial population is randomly generated for the initial learning scenario. For other

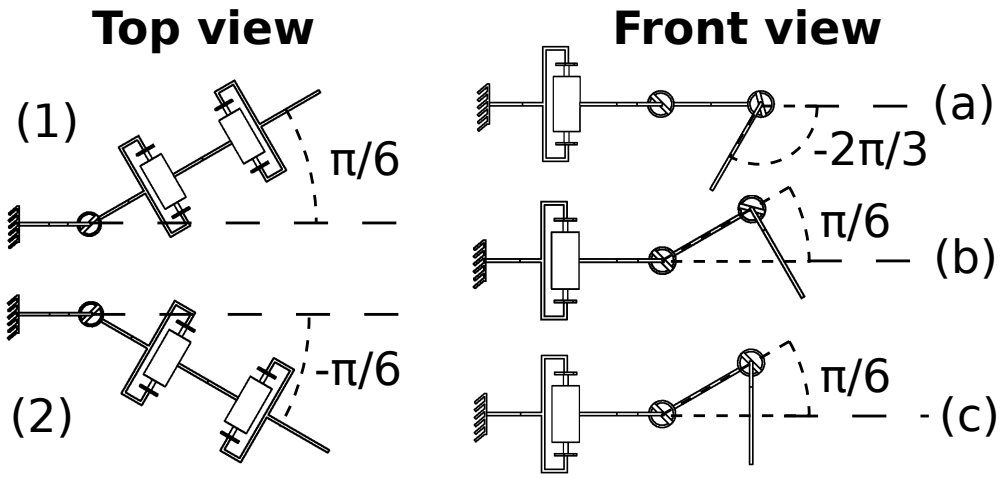
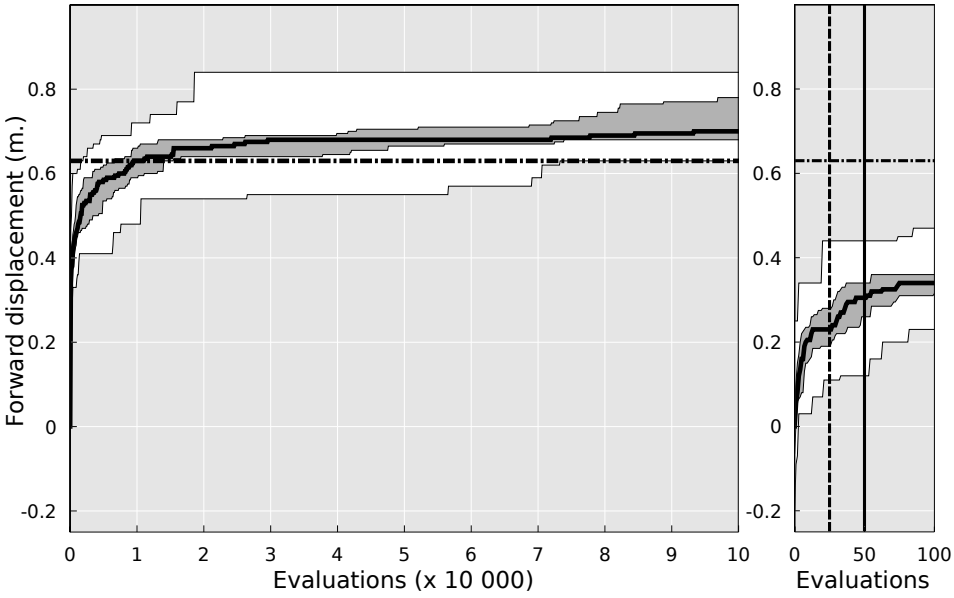


Figure D.1: The six possible actions of a leg that can be tested on the robot: (1,a), (2, a), (1,b), (2, b), (1,c), (2, c).

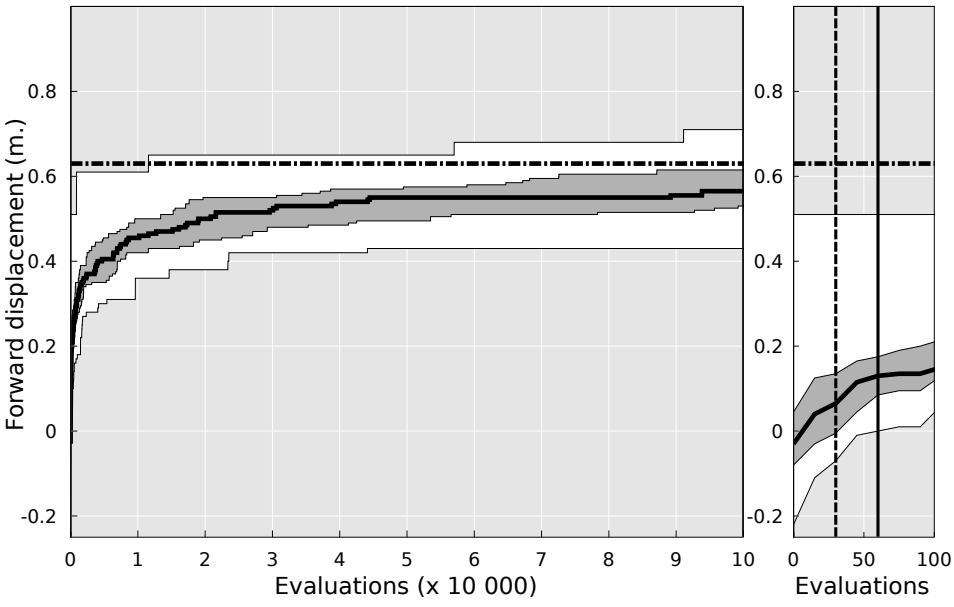
scenarios, the population is initialized with the self-model of the undamaged robot. A new action is tested every 80 generations, which leads to a total of 25 actions tested on the real robot. Applying a new action on the robot implies making an additional simulation for each model at each generation, leading to arithmetic progression of the number of simulation needed per generation. Moreover, 36×36 additional simulations are needed each time a new action has to be selected and transferred (the whole action set applied to the whole population). In total, about one million simulations have been done per run ($(25 \times 26/2) \times 36 \times 80 + 36 \times 36 \times 25 = 968400$).
The self-modeling process is iterated 25 times (i.e. 25 real tests; variant *tests*) before the optimization of controllers occurs, which leads to a median duration of 250 minutes on overall. (Table D.1). For comparison, the best controller optimized with the self-model obtained after 25 minutes of self-modeling is also kept (i.e. after 11 real tests; variant *time*).

D.1.2 Validation of the implementations

To ensure that the observed poor performances are not caused by an implementation error, the local search and the policy search have been tested in simulation with higher numbers of evaluations. Each algorithm has been executed 40 times with 10^5 evaluations on the simulated hexapod robot, which is used as a self-model with the T-Resilience algorithm. Results are depicted on Figure D.2.
These experiments in simulation demonstrate that the small number of evaluations is the cause of the poor performances of these two algorithms in our experiments (cases A to E). Walking controllers are achieved after about $1,000$ evaluations (me-



(a) Local search.



(b) Policy search.

Figure D.2: Performances obtained with the local search (a) and the policy search (b) in simulation (40 runs; 10^5 evaluations). Thick black curves depict median performance values, dark areas are delimited by first and third quartiles and light areas by lower and upper bounds. Horizontal dashed lines depict the performance of the reference controller in simulation. Figures on the right show the progression of performance values during the first 100 evaluations. Median number of evaluations used in our experiments on the robot for the *tests* and the *time* variants are respectively depicted by vertical dashed lines and vertical solid lines. Local search and policy gradient search are both able to find good controllers, provided that they are executed during enough iterations.

dian performance greater than 0.4 m). After 2×10^4 evaluations, both algorithms converge to behaviors with good performances (figure D.2; median performances 0.66 m for local search and 0.50 m for policy search). These performances have to be balanced with the high number of required evaluations that is most of the time not feasible with a real robot and not compatible with our damage recovery problem. In our experiments with the real robot (section 4.2.3), we only performed between 25 and 60 evaluations, which is not enough for the algorithms to find efficient controllers, even in simulation (figure D.2; median performances 0.23 m after 25 evaluations and 0.30 m after 50 evaluations for local search; 0.06 m after 30 evaluations and 0.13 m after 60 evaluations for policy search).
These results indicate that the poor performances observed with both algorithms in our experiments (cases A to E) are mainly caused by low numbers of evaluations performed on the robot.

D.1.3 Median durations and number of tests

Algorithms	Median duration (min.)	Median number of real tests
Local search	20 (10)	50 (25)
Policy search	25 (13)	60 (30)
T-Resilience	19 (19)	25 (25)
Self-modeling	25 (250)	11 (25)

Table D.1: Median duration and median number of real tests on the robot during a full run for each algorithm, for the "time" variant. Number in parenthesis correspond to the "tests" variant.

D.1.4 Statistical tests

	Local search		Policy search		Self-modeling		Ref.
	tests	time	tests	time	time	tests	
A	0.008	0.008	0.008	0.008	0.008	0.008	1.000
B	0.008	0.016	0.016	0.016	0.008	0.008	0.063
C	0.016	0.151	0.008	0.008	0.008	0.008	0.063
D	0.151	0.548	0.016	0.087	0.063	0.008	0.063
E	0.008	0.008					0.063
F	0.008	0.063					0.063

Table D.2: Statistical significance when comparing performances between the T-Resilience and the other algorithms (Ref. corresponds to the reference gait). P-values are computed with Wilcoxon rank-sum tests.

D.2 Methods for the Intelligent Trial and Error algorithm and experiments

D.2.1 Notations

- $\mathbf{c}$: Parameters of a controller (vector)
- $\mathbf{x}$: A location in a discrete behavioral space (i.e. a type of behavior) (vector)
- χ: A location in a discrete behavioral space that has been tested on the physical robot (vector)
- $\mathscr{P}$: Behavior-performance map (stores performance) (associative table)
- $\mathscr{C}$: Behavior-performance map (stores controllers) (associative table)
- $\mathscr{P}(\mathbf{x})$: Max performance yet encountered at $\mathbf{x}$ (scalar)
- $\mathscr{C}(\mathbf{x})$: Controller currently stored in $\mathbf{x}$ (vector)
- $\chi_{\mathbf{1:t}}$: All previously tested behavioral descriptors at time t (vector of vectors)
- $\mathbf{P_{1:t}}$: Performance in reality of all the candidate solutions tested on the robot up to time t (vector)
- $\mathscr{P}(\chi_{\mathbf{1:t}})$: Performance in the behavior-performance map for all the candidate solutions tested on the robot up to time t (vector)
- $f()$: Performance function (unknown by the algorithm) (function)
- σ^2_{noise}: Observation noise (a user-specified parameter) (scalar)
- $k(\mathbf{x}, \mathbf{x})$: Kernel function (see section "kernel function") (function)
- $\mathbf{K}$: Kernel matrix (matrix)
- $\mathbf{k}$: Kernel vector $[k(\mathbf{x}, \chi_1), k(\mathbf{x}, \chi_2), ..., k(\mathbf{x}, \chi_t)]$ (vector)
- $\mu_t(\mathbf{x})$: Predicted performance for $\mathbf{x}$ (i.e. the mean of the Gaussian process) (function)
- $\sigma^2_t(\mathbf{x})$: Standard deviation for $\mathbf{x}$ in the Gaussian process (function)

D.2.2 Hexapod Experiment

Main Behavioral descriptor (duty factor) The default behavioral descriptor is a 6-dimensional vector that corresponds to the proportion of time that each leg is in contact with the ground(also called duty factor). When a controller is simulated, the algorithm records at each time step (every 30 ms) whether each leg is in contact with the ground (1: contact, 0: no contact). The result is 6 Boolean time series

(C_i for the i^{th} leg). The behavioral descriptor is then computed with the average of each time series:

$$\mathbf{x} = \begin{bmatrix} \frac{\sum_t C_1(t)}{\text{numTimesteps}(C_1)} \\ \vdots \\ \frac{\sum_t C_6(t)}{\text{numTimesteps}(C_6)} \end{bmatrix} \tag{D.1}$$

During the generation of the behavior-performance map, the behaviors are stored in the maps's cells by discretizing each dimension of the behavioral descriptor space with these five values: {0, 0.25, 0.5, 0.75, 1}. During the adaptation phase, the behavioral descriptors are used with their actual values and are thus not discretized.

Alternate Behavioral descriptor (orientation) The alternate behavioral descriptor tested on the physical robot (we investigated many other descriptors in simulation: Supplementary Experiment S5) characterizes changes in the angular position of the robot during walking, measured as the proportion of 15ms intervals that each of the pitch, roll and yaw angles of the robot frame are positive (three dimensions) and negative (three additional dimensions):

$$\mathbf{x} = \begin{bmatrix} \frac{1}{K}\sum_k U(\Theta^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Theta^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(\Psi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Psi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(\Phi^T(k) - 0.005\pi) \\ \frac{1}{K}\sum_k U(-\Phi^T(k) - 0.005\pi) \end{bmatrix} \tag{D.2}$$

where $\Theta^T(k)$, $\Psi^T(k)$ and $\Phi^T(k)$ denote the pitch, roll and yaw angles, respectively, of the robot torso (hence T) at the end of interval k, and K denotes the number of 15ms intervals during the 5 seconds of simulated movement (here, $K = 5s/0.015s \approx 334$). The unit step function $U(\cdot)$ returns 1 if its argument exceeds 0, and returns 0 otherwise. To discount for insignificant motion around 0 rad, orientation angles are only defined as positive if they exceed 0.5% of π rad. Similarly, orientation angles are only defined as negative if they are less than -0.5% of π rad.

Performance function In these experiments, the "mission" of the robot is to go forward as fast as possible. The performance of a controller, which is a set of parameters (see appendix A.2.2: Parametrized controller), is defined as how far the robot moves in a pre-specified direction in 5 seconds.
During the behavior-performance map creation step, the performance is obtained thanks to the simulation of the robot. All odometry results reported on the physical robot, during the adaptation step, are measured with the embedded simultaneous location and mapping (SLAM) algorithmDryanovski et al. (2013). The accuracy of this algorithm was evaluated by comparing its measurements to ones made by hand on 40 different walking gaits. These experiments revealed that the median

measurement produced by the odometry algorithm is reasonably accurate, being just 2.2% lower than the handmade measurement (Supplementary Fig. D.3d).
Some damage to the robot may make it flip over. In such cases, the visual odometry algorithm returns pathological distance-traveled measurements either several meters backward or forward. To remove these errors, we set all distance-traveled measurements less than zero or greater than two meters to zero. The result of this adjustment is that the algorithm appropriately considers such behaviors low-performing. Additionally, the SLAM algorithm sometimes reports substantially inaccurate low values (outliers on Supplementary Fig. D.3d). In these cases the adaptation step algorithm will assume that the behavior is low-performing and will select another working behavior. Thus, the overall algorithm is not substantially impacted by such infrequent under-measurements of performance.

Stopping criterion In addition to guiding the learning process to the most promising area of the search space, the estimated performance of each solution in the map also informs the algorithm of the maximum performance that can be expected on the physical robot. For example, if there is no controller in the map that is expected to perform faster on the real robot than 0.3m/s, it is unlikely that a faster solution exists. This information is used in our algorithm to decide if it is worth continuing to search for a better controller; if the algorithm has already discovered a controller that performs nearly as well as the highest value predicted by the model, we can stop the search.
Formally, our stopping criterion is

$$\max(\mathbf{P}_{1:t}) \geq \alpha \max_{\mathbf{x} \in \mathscr{P}}(\mu_t(\mathbf{x})), \quad \text{with } \alpha = 0.9 \tag{D.3}$$

where $\mathbf{x}$ is a location in the discrete behavioral space (i.e. a type of behavior) and μ_t is the predicted performance of this type of behavior. Thus, when one of the tested solutions has a performance of 90% or higher of the maximum expected performance of any behavior in the map, the algorithm terminates. At that point, the highest-performing solution found so far will be the compensatory behavior that the algorithm selects. An alternate way the algorithm can halt is if 20 tests on the physical robot occur without triggering the stopping criterion described in equation D.3: this event only occurred in 2 of 240 experiments performed on the physical robot described in the main text. In this case, we selected the highest-performing solution encountered during the search. This user-defined stopping criterion is not strictly necessary, as the algorithm is guaranteed to stop in the worst case after every behavior in the map is tested, but it allows a practical limit on the number of trials performed on the physical robot.

Initiating the Adaptation Step The adaptation step is triggered when the performance drops by a certain amount. The simplest way to choose that threshold is to let the user specify it. Automating the selection of this value, and the impact of triggering the algorithm prematurely, is an interesting question for future research in this area.

D.2.3 Robotic Arm Experiment

Behavioral descriptor Because the most important aspect of the robot's behavior in this task is the final position of the gripper, we use it as the behavioral descriptor:

$$\text{behavioral\_descriptor}(\text{simu}(\mathbf{c})) = \begin{bmatrix} x_g \\ y_g \end{bmatrix} \tag{D.4}$$

where (x_g, y_g) denotes the position of the gripper once all the joint have reached their target position.
The size of the working area of the robot is a rectangle measuring $1.4m \times 0.7m$. For the behavior-performance map, this rectangle is discretized into a grid composed of 20000 square cells (200×100). The robot is 62cm long.

Performance function Contrary to the hexapod experiment, for the robotic arm experiment the performance function for the behavior-map creation step and for the adaptation step are different. We did so to demonstrate that the two can be different, and to create a behavior-performance map that would work with arbitrary locations of the target bin.
For the *behavior-performance map generation step* (accomplished via the MAP-Elites algorithm), the performance function captures the idea that all joints should contribute equally to the movement. Specifically, high-performance is defined as minimizing the variance of the joint angles, that is:

$$\text{performance}(\text{simu}(\mathbf{c}))) = -\frac{1}{8}\sum_{i=0}^{i=7}(p_i - m)^2 \tag{D.5}$$

where p_i is the angular position of joint i (in radians) and $m = \frac{1}{8}\sum_{i=0}^{i=7} p_i$ is the mean of the joint angles. This performance function does not depend on the target. The map is therefore generic: it contains a high-performing controller for each point of the robot's working space.
For the *adaptation step* (accomplished via the M-BOA algorithm), the behavior-performance map, which is generic to many tasks, is used for a particular task. To do so, the adaption step has a different performance measure than the step that creates the behavior-performance map. For this problem, the predicted performance measure is the Euclidean distance to the target (closer is better). Specifically, for each behavior descriptor $\mathbf{x}$ in the map, performance is

$$\mathscr{P}(\mathbf{x}) = -||\mathbf{x} - \mathbf{b}|| \tag{D.6}$$

where $\mathbf{b}$ is the (x, y) position of the target bin. Note that the variance of the joint angles, which is used to create the behavior-performance map, is ignored during the adaptation step.
The performance of a controller on the physical robot is minimizing the Euclidean distance between the gripper (as measured with the external camera) and the target bin:

$$\text{performance}(\text{physical\_robot}(\mathscr{C}(\chi))) = -||\mathbf{x_g} - \mathbf{b}|| \tag{D.7}$$

where $\mathbf{x_g}$ is the position of the physical gripper after all joints have reached their final position, $\mathbf{b}$ is the position of the bin, and $\mathscr{C}(\chi)$ is the controller being evaluated (χ is the position in simulation that controller reached).
If the gripper reaches a position outside of the working area, then the camera cannot see the marker. In these rare cases, we set the performance of the corresponding controller to a low value (-1 m).
For the control experiments with traditional Bayesian optimization on the physical robot, self-collisions are frequent during adaptation, especially given that we initialize the process with purely random controllers (i.e. random joint angles). While a single self-collision is unlikely to break the robot, hundreds of them can wear out the gearboxes because each servo continues to apply a force for a period of time until it determines that it cannot move. To minimize costs, and because we ran 210 independent runs of the algorithm (14 scenarios $\times$ 15 replicates), we first tested each behavior in simulation (taking the damage into account) to check that there were no self-collisions. If we detected a self-collision, the performance for that behavior was set to a low value (-1m).
Auto-collisions are much less likely with Intelligent Trial & Error because the behavior-performance map contains only controllers that do not self-collide on the undamaged, simulated robot. As a consequence, in the Intelligent Trial & Error experiments we did not simulate controllers before testing them on the physical robot.

Stopping criterion Because the robot's task is to release a ball into a bin, the adaptation step can be stopped when the gripper is above the bin. The bin is circular with a diameter of 10 cm, so we stopped the adaptation step when the red cap is within 5 cm of the center of the bin.

D.2.4 Selection of parameters

All of the data reported in this section comes from experiments with the simulated hexapod robot, unless otherwise stated.

Selecting the ρ value For ρ between 0.1 and 0.8, we counted the number of behaviors from the map that would be influenced by a single test on the real hexapod robot (we considered a behavior to be influenced when its predicted performance was affected by more than 25% of the magnitude of the update for the tested behavior): with $\rho = 0.2$, the update process does not affect any neighbor in the map, with $\rho = 0.4$, it affects 10% of the behaviors, and with $\rho = 0.8$, it affects 80% of them. Additional values are shown in Supplementary Fig. D.3c.
The previous paragraph describes tests we conducted to determine the number of behaviors in the map affected by different ρ values, but those experiments do not tell us how different ρ values affect the performance of the algorithm overall. To

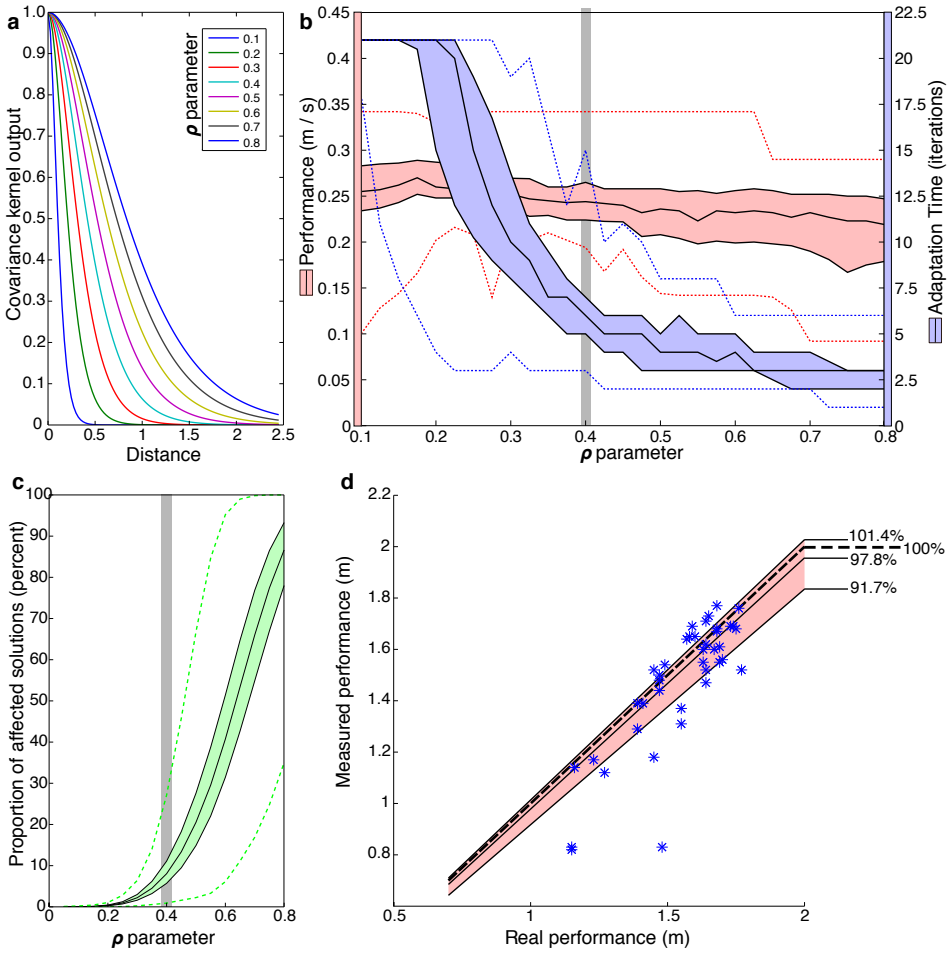


Figure D.3: **The effect of changing the algorithm's parameters. (a) The shape of the Matérn kernel function for different values of the ρ parameter. (b) Performance and required adaptation time obtained for different values of ρ.** For each ρ value, the R-BOA algorithm was executed in simulation with 8 independently generated behavior-performance maps and for 6 different damage conditions (each case where one leg is missing). **(c) The number of controllers in the map affected by a new observation according to different values of the ρ parameter. (d) The precision of the odometry value**. The distances traveled by the physical robot, as measured manually ("real performance") is compared to the measurements automatically provided by the simultaneous location and mapping (SLAM) algorithm("measured performance"). The dashed black line indicates the hypothetical case where SLAM measurements are error-free and thus are the same as manual measurements. In (b), (c) and (d), the middle, black lines represent medians and the borders of the shaded areas show the 25^{th} and 75^{th} percentiles. The dotted lines are the minimum and maximum values. The gray bars show the ρ value chosen for the hexapod experiments in the main text. See Supplementary Methods for additional details and analysis.

assess that, we then repeated the experiments from the main paper with a set of possible values ($\rho \in [0.1 : 0.025 : 0.8]$) in simulation (i.e., with a simulated, damaged robot), including testing on 6 separate damage scenarios (each where the robot loses a different leg) with all 8 independently generated replicates of the default 6-dimensional behavior-performance map. The algorithm stopped if 20 adaptation iterations passed without success according to the stopping criteria described in the main text and section D.2.2: Stopping criterion. The results reveal that median performance decreases only modestly, but significantly, when the value of ρ increases: changing ρ from 0.1 to 0.8 only decreases the median value 12%, from 0.25 m/s to 0.22 m/s (p-value $= 9.3 \times 10^{-5}$ via Matlab's Wilcoxon ranksum test, Supplementary Fig. D.3b). The variance in performance, especially at the extreme low end of the distribution of performance values, is not constant over the range of explored values. Around $\rho = 0.3$ the minimum performance (Supplementary Fig. D.3b, dotted red line) is higher than the minimum performance for more extreme values of ρ.

A larger effect of changing ρ is the amount of time required to find a compensatory behavior, which decreases when the value of ρ increases (Supplementary Fig. D.3b). With a ρ value lower than 0.25, the algorithm rarely converges in less than the allotted 20 iterations, which occurs because many more tests are required to cover all the promising areas of the search space to know if a higher-performing behavior exists than the best-already-tested. On the other hand, with a high ρ value, the algorithm updates its predictions for the entire search space in a few observations: while fast, this strategy risks missing promising areas of the search space.

In light of these data, we chose $\rho = 0.4$ as the default value for our hexapod experiments because it represents a good trade-off between a high minimum performance and a low number of physical tests on the robot. The value of ρ for the robotic arm experiment has been chosen with the same method.

Selection of the κ value For the hexapod robot experiments, we chose $\kappa = 0.05$. This relatively low value emphasizes exploitation over exploration. We chose this value because the exploration of the search space has already been largely performed during the behavior-performance map creation step: the map suggests which areas of the space will be high-performing, and should thus be tested, and which areas of the space are likely unprofitable, and thus should be avoided.

For the robotic arm experiments, we chose $\kappa = 0.3$, which emphasizes exploration more, because it experimentally leads to better results.

D.2.5 Running time

Computing hardware All computation (on the physical robots and in simulation) was conducted on a hyperthreaded 16-core computer (Intel Xeon E5-2650 2.00GHz with 64Gb of RAM). This computational power is mainly required for the behavior-performance map creation step. Creating one map for the hexapod experiment took 2 weeks, taking advantage of the fact that map creation can easily

be parallelized across multiple cores. Map creation only needs to be performed once per robot (or robot design), and can happen before the robot is deployed. As such, the robot's onboard computer does not need to be powerful enough to create the map.

For the hexapod robot experiment, the most expensive part of adaptation is the Simultaneous Localization And Mapping (SLAM) algorithmDryanovski et al. (2013); Thrun et al. (2005); Dissanayake et al. (2001), which measures the distance traveled on the physical hexapod robot. It is slow because it processes millions of 3D points per second. It can be run on less powerful computers, but doing so lowers its accuracy because fewer frames per second can be processed. As computers become faster, it should be possible to run high-accuracy SLAM algorithms in low-cost, onboard computers for robots.

The rest of the adaptation step needs much less computational power and can easily be run on an onboard computer, such as a smartphone. That is because it takes approximately 15,000 arithmetic operations between two evaluations on the physical robot, which requires less than a second or two on current smartphones.

Measuring how long adaptation takes (hexapod robot) The reported time to adapt includes the time required for the computer to select each test and the time to conduct each test on the physical robot. Overall, evaluating a controller on the physical hexapod robot takes about 8 seconds (median 8.03 seconds, 5^{th} and 95^{th} percentiles [7.95; 8.21] seconds): 0.5-1 second to initialize the robot, 5 seconds during which the robot can walk, 0.5-1 second to allow the robot to stabilize before taking the final measurement, and 1-2 seconds to run the SLAM algorithm. Identifying the first controller to test takes 0.03 [0.0216; 0.1277] seconds. The time to select the next controller to test increases depending on the number of previous experiments because the size of the Kernel Matrix (K matrix, see Methods and Supplementary Fig. 4), which is involved in many of the arithmetic operations, grows by one row and one column per test that has been conducted. For example, selecting the second test takes 0.15 [0.13; 0.22] seconds, while the 10^{th} selection takes 0.31 [0.17; 0.34] seconds.

D.3 Notation for the State-Based BO with Transfer, Priors and blacklists algorithms

$$
\begin{aligned}
&\boldsymbol{\mu}_{t+1}(\mathbf{x}) = \mathbf{T_r}\begin{bmatrix} \boldsymbol{\mu}_0(\mathbf{x}) \\ 1 \end{bmatrix} + \left(\mathbf{Pc}_{1:t}^{\intercal} - \mathbf{T_r}\begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) \\ 1 \end{bmatrix}\right)\mathbf{K}^{-1}\mathbf{k} \\
&\sigma_{t+1}^2(\mathbf{x}) = k(\mathbf{x},\mathbf{x}) - \mathbf{k_{bl}}^{\intercal}{\mathbf{K_{bl}}}^{-1}\mathbf{k_{bl}} \\
&\mathbf{K} = \begin{bmatrix} k(\boldsymbol{\chi}_1,\boldsymbol{\chi}_1) & \cdots & k(\boldsymbol{\chi}_1,\boldsymbol{\chi}_{t+1}) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{\chi}_{t+1},\boldsymbol{\chi}_1) & \cdots & k(\boldsymbol{\chi}_{t+1},\boldsymbol{\chi}_{t+1}) \end{bmatrix} \\
&\mathbf{K_{bl}} = \begin{bmatrix} & & & k(\boldsymbol{\chi}_1,\mathbf{bl}_1) & \cdots & k(\boldsymbol{\chi}_1,\mathbf{bl}_n) \\ & \mathbf{K} & & \vdots & \ddots & \vdots \\ & & & k(\boldsymbol{\chi}_t,\mathbf{bl}_1) & \cdots & k(\boldsymbol{\chi}_t,\mathbf{bl}_n) \\ k(\mathbf{bl}_1,\boldsymbol{\chi}_1) & \cdots & k(\mathbf{bl}_1,\boldsymbol{\chi}_t) & k(\mathbf{bl}_1,\mathbf{bl}_1)+\sigma_{noise}^2 & \cdots & k(\mathbf{bl}_1,\mathbf{bl}_n) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{bl}_n,\boldsymbol{\chi}_1) & \cdots & k(\mathbf{bl}_n,\boldsymbol{\chi}_t) & k(\mathbf{bl}_n,\mathbf{bl}_1) & \cdots & k(\mathbf{bl}_n,\mathbf{bl}_n)+\sigma_{noise}^2 \end{bmatrix} \\
&\mathbf{k_{bl}} = \begin{bmatrix} k(\mathbf{x},\boldsymbol{\chi}_1) & \cdots & k(\mathbf{x},\boldsymbol{\chi}_t) & k(\mathbf{x},\mathbf{bl}_1) & \cdots & k(\mathbf{x},\mathbf{bl}_n) \end{bmatrix} \\
&\boldsymbol{\mu}_0(\mathbf{x}) = \begin{bmatrix} \mu_0^1(\mathbf{x}) \\ \vdots \\ \mu_0^n(\mathbf{x}) \end{bmatrix} \\
&\boldsymbol{\mu}_0(\boldsymbol{\chi}_{1:t}) = \begin{bmatrix} \boldsymbol{\mu}_0(\boldsymbol{\chi}_1) & \cdots & \boldsymbol{\mu}_0(\boldsymbol{\chi}_t) \end{bmatrix} \\
&\mathbf{T_r} = \begin{bmatrix} \theta_{Tr}^{1,1} & \theta_{Tr}^{1,2} & \dots & \theta_{Tr}^{1,n+1} \\ \vdots & & \ddots & \vdots \\ \theta_{Tr}^{p,1} & \theta_{Tr}^{p,2} & \dots & \theta_{Tr}^{p,n+1} \end{bmatrix}
\end{aligned}
\tag{D.8}
$$