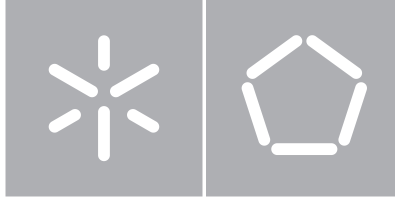
Universidade do Minho
Escola de Engenharia

João Francisco Freitas Santos Macedo

Towards the Improvement of Robot Motion Learning Techniques

Outubro de 2014



Universidade do Minho
Escola de Engenharia

João Francisco Freitas Santos Macedo

Towards the Improvement of Robot Motion Learning Techniques

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professora Cristina Manuela Peixoto dos Santos
Professor Lino António Costa

Outubro de 2014

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Acknowledgements

This thesis has been a journey, and a test to my ability to remain optimistic, focused and motivated. Thanks to many people, I've been able to overcome the many storms that could otherwise have ravaged me.

Any journey has a goal, and a path that must be followed in order to reach it. I thank my supervisor Cristina Santos and co-supervisor Lino Costa for being my map and compass and giving me the guidance I needed to define and reach that goal. Cristina has always believed in me and in my work, even when I was myself doubtful about it and she gave me the motivation to keep going. I'll never forget her patience and support.

I thank University of Minho and all my professors for giving me an education of excellence and providing me the tools I needed to follow this path. They helped me building the boat in which I sailed during this journey.

My friends and laboratory colleagues were the wind that kept me sailing in a good pace. In moments of more stress and tension, they were there with their jokes and stories. With them I had some of the most insightful, exciting and strange, but funny, conversations of my life. They were definitely the breadth of fresh air whenever I needed.

The support from my family was always constant. They were the anchor that held me and allowed me to rest, providing me financial and emotional support throughout my entire life. Its thanks to them that I even got the opportunity to start this journey in the first place, and for that I will be forever grateful.

Last but not least, I want to thank Lúcia, my safe haven. She makes me want to do better as a human being everyday. She is the person I always wanted to impress and that I strive to make happy.

To everyone, thank you. Without you, writing this thesis would not have been possible.

Towards the Improvement of Robot Motion Learning Techniques

This manuscript presents solutions and methods to address some of the many problems that arise when dealing with the complex task of motor skill learning in robots.

In the last years, several research lines have focused on learning motion primitives either through imitation learning or reinforcement learning. However, for many applications, learning a motion primitive of a single form is not enough and it is required that after being assimilated, the primitive is generalizable such that it can be executed in different contexts and for distinct instances of the same task. Therefore, the motion primitive must adapt a set of parameters according to the environment variables instead of always executing the exact same motor commands when it is put into action. Another aspect to have into consideration is how the learning process of motion primitives is guided. Some primitives are too complex to be learned all at once, i.e, learning all their intricacies without a properly structured approach may be intractable.

In this thesis, these aspects are mindfully taken into account, allowing to develop reinforcement learning techniques that are then used to teach a controller of a biped robot that is only able to generate stable locomotion on a flat surface, making it tolerant to a range of slope angles, perpendicular and/or parallel to the direction of walking. Legged locomotion is a relevant example of a complex and dynamic motor skill that has been the focus of intensive research for many years in robotics and it is expected for the techniques that are successful in the learning of such a hard task to be useful in other contexts.

In order to achieve this goal, three main steps, divided into chapters of this thesis, are taken. First, an existing algorithm - Cost-regularized Kernel Regression (CrKR) - originally introduced to allow learning to generalize parameterized policies is modified and extended into a new algorithm named CrKR^{++}. Some of the performed changes allow to use the algorithm for training sessions with a high number of samples, which is needed when it is intended to learn complex policies. This feat would be impracticable with the original version of the algorithm due to its high computational complexity. The remaining changes are issued with the purpose of improving the general effectiveness of the algorithm.

Second, a framework that enables storing, combining and mutual learning of parameterized policies is presented. This framework, where the CrKR^{++} algorithm plays a core role, provides the means, for instance, to create a movement primitives library or to perform gradual learning of a motor skill, being named Flexible Framework for Learning (F3L).

Finally, the developed framework is used to teach the controller of the biped robot to adapt its locomotion parameters according to the slope angles of the underlying surface. The achieved solution and intermediate steps are tested in simulation software with Dynamic Anthropomorphic Robot with Intelligence–Open Platform (DARwIn-OP) in carefully delineated experiments.

Rumo à Melhoria das Técnicas de Aprendizagem de Movimento em Robôs

Esta tese apresenta soluções e métodos que abordam alguns dos muitos problemas que surgem quando lidando com o complexo problema da aprendizagem de tarefas motoras em robôs.

Nos últimos anos, várias linhas de investigação focaram-se na aprendizagem de primitivas de movimento, quer pela aprendizagem via imitação quer pela aprendizagem via reforço. Contudo, em muitas aplicações, não basta assimilar uma primitiva numa única forma e pode ser necessário que depois de assimilada, uma primitiva seja generalizável de maneira a ser possível executá-la em diferentes contextos e para diferentes instâncias de uma mesma tarefa. Uma primitiva de movimento deve portanto nestes casos adaptar um conjunto de parâmetros de acordo com as condições do meio envolvente em vez de executar sempre os mesmos comandos motores quando colocada em ação. Outro aspeto a ter em consideração é ainda a forma como o processo de aprendizagem das primitivas de movimento é guiado. Algumas primitivas são demasiado complexas para serem apreendidas de uma vez só, isto é, aprender todas as suas nuances sem uma abordagem estruturada pode revelar-se extremamente difícil.

Nesta tese, estes dois aspetos são tidos em conta, o que permite desenvolver novas técnicas de aprendizagem via reforço que são depois usadas para ensinar um programa controlador de um robô bípede que é apenas capaz de lidar com superfícies planas, tornando-o tolerante a uma gama de inclinações em direções perpendiculares ou paralelas à direção do movimento. A locomoção com pernas é o exemplo definitivo de uma tarefa motora complexa e dinâmica que tem sido alvo de investigação intensiva durante anos na robótica. É de esperar que as técnicas que sejam bem sucedidas na aprendizagem de uma tarefa com este grau de dificuldade sejam também úteis em outros contextos.

Para atingir este objetivo, três passos principais, que se dividem em capítulos desta tese são dados. Em primeiro lugar, um algoritmo já existente - CrKR - ,originalmente criado para permitir a aprendizagem de políticas parametrizadas, é modificado e transformado num novo algoritmo denominado CrKR^{++}. Algumas das modificações feitas permitem usar o algoritmo em sessões de treino com um maior número de amostras, o que é necessário quando se pretende aprender políticas com um elevado grau de complexidade. Tal seria impossível com a versão original do algoritmo devido à sua elevada complexidade computacional. As restantes modificações são introduzidas com o propósito de melhorar a eficácia geral do algoritmo.

Em segundo lugar, uma *framework* que permite o armazenamento, a combinação e a aprendizagem mútua de políticas parametrizadas é apresentada. Esta *framework*, onde o algoritmo CrKR^{++} desempenha uma função nuclear, providencia os meios para, por exemplo, criar uma biblioteca de primitivas de movimento ou realizar aprendizagem gradual de uma tarefa motora sendo denominada de F3L.

Por fim, a *framework* desenvolvida é utilizada para ensinar o controlador do robô bípede a adaptar determinados parâmetros da locomoção em função da inclinação da superfície subjacente. A solução alcançada bem como os passos intermédios são testados em software de simulação com o robô DARwIn-OP em experiências cuidadosamente delineadas.

Contents

1 Introduction 1

1.1 Motivation 1

1.2 Goals 3

1.3 Contribution 3

1.4 Methodology and execution 4

1.5 Outline 5

1.6 Publications related with this thesis 5

2 Context and related work 7

2.1 Problems and challenges 7

2.1.1 The challenges in locomotion 7

2.1.2 The challenges in learning 9

2.2 Reinforcement learning 11

2.2.1 Notation and problem formulation 11

2.2.2 Progress and evolution of reinforcement learning algorithms 14

2.2.3 State of the art reinforcement learning algorithms 17

2.3 Movement generation and representation 19

3 Improving on Cost-regularized Kernel Regression 27

3.1 Cost-regularized Kernel Regression as a suitable algorithm for learning generalized policies 28

3.1.1 Outline 29

3.1.2 Computational Complexity 32

3.2 Performance and effectiveness 32

3.2.1 Circumventing computational complexity limitations 32

3.2.2 Costs standardization 37

3.2.3 Hints 40

3.2.4 Variance multiplier 40

3.2.5 Final algorithm 40

3.3 Implementation . . . 41
3.3.1 Architecture . . . 42
3.3.2 Parameters guide . . . 43
3.4 Tests . . . 46
3.4.1 Results . . . 49
3.4.2 Discussion . . . 54
3.5 Summary . . . 55

4 Flexible Framework for Learning 57
4.1 Multiple CrKR^{++} units . . . 57
4.2 Flexible Framework for Learning (F3L) operations . . . 58
4.2.1 Learning from zero . . . 59
4.2.2 Learning from baseline . . . 59
4.2.3 Branch . . . 59
4.2.4 Merge . . . 60
4.2.5 Mutual learning . . . 60
4.2.6 Partial learning . . . 61
4.2.7 Learn with hints . . . 61
4.3 Implementation . . . 61
4.3.1 Architecture . . . 62
4.3.2 Parameters and methods guide . . . 63
4.4 Summary . . . 64

5 Generalizing walking to slopes in any direction on DARwIn-OP 65
5.1 Baseline . . . 66
5.1.1 DARwIn-OP . . . 66
5.1.2 Controller . . . 66
5.2 Feedback mechanisms . . . 69
5.2.1 Ankle pitch correction . . . 69
5.2.2 Ankle roll correction . . . 70
5.2.3 Direction correction . . . 70
5.3 Achieving locomotion on sloped surfaces . . . 72
5.3.1 Task . . . 72
5.3.2 Simulation setup . . . 73
5.3.3 Cost function . . . 74
5.3.4 Controller parameters . . . 75
5.3.5 Learning structure . . . 75
5.3.6 Learning structure parameters . . . 78

5.4 Results 79
5.5 Discussion 86
5.6 Summary 87

6 Final remarks **89**
6.1 Conclusions 89
6.2 Future work 90

Appendices **97**

A Parameter values for the controller **97**

B Parameter values for the learning process **99**

List of Figures

2.1 Agent–Environment interaction . . . 12
2.2 Marionette analogy . . . 14
2.3 Typical policy learning algorithm steps . . . 15
2.4 Trend in policy improvement algorithms . . . 16
2.5 Different perturbation schemes . . . 17
2.6 Bionic CPG control process . . . 21
2.7 Applied CPG scheme . . . 22
2.8 Discrete movement Dynamic Motion Primitive (DMP) . . . 23
2.9 Rythmic movement DMPs . . . 23
2.10 Splines with different number of knots . . . 24

3.1 Evolution of a policy with CrKR . . . 31
3.2 Standardize Pool illustration . . . 39
3.3 Class diagram of CrKR^{++} implementation . . . 42
3.4 Beale's, Goldstein–Price and Booth's functions . . . 47
3.5 Triangular wave function . . . 48
3.6 Bivariate function . . . 48
3.7 Function through artificial landscape - results . . . 50
3.8 Triangular wave approximation - results . . . 51
3.9 Bivariate function approximation - results . . . 52
3.10 Time per iteration with and without execution splitting . . . 53
3.11 Function through artificial landscape: outcome policy . . . 53
3.12 Triangular wave approximation: outcome policy . . . 54
3.13 Bivariate function approximation: outcome policy . . . 54

4.1 Primitive combination . . . 58
4.2 Learn from zero operation . . . 59
4.3 Learn from baseline operation . . . 59
4.4 Branch operation . . . 59

4.5 Merge operation . . . 60
4.6 Mutual learning operation . . . 60
4.7 Partial learning operation . . . 61
4.8 Learn with hints operation . . . 61
4.9 Class diagram of F3L implementation . . . 62

5.1 DARwIn-OP - Degrees of freedom . . . 66
5.2 DARwIn-OP walking motions . . . 67
5.3 Oscillators coupling and influence . . . 67
5.4 Effect of the ankle pitch correction feedback . . . 70
5.5 Effect of the ankle roll correction feedback . . . 71
5.6 Effect of the direction correction feedback . . . 72
5.7 Direction correction mechanism when θ_{goal} is changed . . . 73
5.8 Final transformation applied to the cost function . . . 75
5.9 Initial primitive branching . . . 76
5.10 Learning $\gamma_{frontal}$ . . . 77
5.11 Learning $\gamma_{lateral}$ . . . 77
5.12 Learning γ_{omni} . . . 78
5.13 Costs before and after adding ankle pitch and ankle roll correction feedbacks . . . 80
5.14 Trajectory of the robot with and without direction correction feedback . . . 81
5.15 Trajectory of the robot on a flat surface when using direction feedback to turn . . . 81
5.17 Costs before and after performing the learning process . . . 83
5.19 DARwIn-OP walking up a slope . . . 85
5.20 DARwIn-OP walking down a slope . . . 86

List of Tables

A.1 Initial parameter values for the controller 97

B.1 Parameter values for the several stages of the learning process 99

List of acronyms

ASBG Adaptive Systems Behavior Group. 4

CPG Central Pattern Generator. 3, 20–22, 66

CrKR Cost-regularized Kernel Regression. v, vii, ix, xiii, 3–5, 18, 19, 27–29, 31, 32, 36, 37, 40, 46, 47, 55, 89

CrKR^{++} Cost-regularized Kernel Regression ++. v, vii, x, xiii, 4, 5, 41–44, 46, 47, 49, 55, 57–59, 61–64, 87, 89–91

DARwIn-OP Dynamic Anthropomorphic Robot with Intelligence–Open Platform. v, vii, x, xiv, 5, 9, 21, 65–67, 85–87, 91, 97, 99

DMP Dynamic Motion Primitive. xiii, 19, 22–25, 28, 71

eNAC Episodic Natural Actor Critic. 15, 16

F3L Flexible Framework for Learning. v, vii, x, xiv, 4, 5, 57, 58, 61–65, 86, 87, 89–91

FSR Force-Sensing Resistor. 66, 69, 91

GUI Graphical User Interface. 91

MDP Markov Decision Process. 12

MoMPs Mixture of Motor Primitives. 19

PI^2 Policy Improvement with Path Integrals. 16–18, 27

PI^2-CMA Path Integral Policy Improvement with Covariance Matrix Adaptation. 17, 18

PI^{BB} Policy Improvement through Black-box Optimization. 18

PoWER Policy learning by Weighting Exploration with the Returns. 16–19, 24, 27

REINFORCE reward increment = nonnegative factor $\times$ offset reinforcement $\times$ characteristic eligibility. 15, 16

RLPF Reinforcement Learning based on Particle Filters. 19

UML Unified Modeling Language. 42, 62

ZMP Zero Moment Point. 20

Chapter 1

Introduction

1.1 Motivation

The human desire for progress and the continuous search for better conditions for living have always existed. The 20th century has been marked by great technological advances which have turned into reality many dreams once thought to be unachievable. Electronics became generally adopted both at an industrial and consumer-grade levels, which had a great impact in areas like health, education and general well being. Computers brought the ability to store, process and access information in a way never seen before, and together with Internet, connected the whole world in a single global village. There is certainly an observable tendency for technology to keep evolving and improving people's lives.

In the early 21st century, research in areas like biotechnology, advanced materials and artificial intelligence started booming. Research in robotics is now starting to grow at a faster pace, being considered by many one of the promises for the near future technological developments. Last developments in robotics include, for instance, powered exoskeletons, with potential applications in heavy lifting related tasks and improvement of living conditions of people suffering muscle related diseases and unmanned vehicles, with the most known cases being Google® driverless car and Amazon's® Prime Air delivery drones projected to start operating in 2015.

Even though robotics are heavily applied in industrial environments, with large assembly lines constituted almost solely by automatic work performed by robots, intense research and development will be required in order to bring them to more domestic environments and to perform several mundane, but non trivial to implement tasks. The idea of robots as versatile household helpers and companions or as sentient beings completely capable of interacting with humans and the world around them has been portrayed several times by science fiction. However, this has not yet become a reality despite the continuous efforts by researchers to-

wards this goal. This reveals one of the greatest challenges in robotics: creating machines capable to learn and to adapt themselves to new situations.

One set of tasks where the aforementioned capacities are essential in order to achieve success is the control of movements, in particular, locomotion. Wheeled robots are typically efficient and easy to control; however, there are many tasks which require a great degree of agility for which wheeled robots are not fitted. Legged robots emerge as an alternative that has the potential to fill this gap. They can vary in a number of ways, including form and number of legs, ranging from centipede like robots, with an arbitrary number of pair of legs, to biped, humanoid, robots.The control of these kind of robots is not a trivial task, however. While designing functional walking gaits and performing the corresponding motor commands can generally be accomplished for specific environments and under general requirements, creating mechanisms that enable legged robots to adapt their movements according to the terrain and surrounding conditions is a much harder task. Not only it may be hard to achieve effective and efficient movement, it is sometimes difficult to ensure the robot's physical integrity, particularly in biped robots, where the risk of falling is greater due to the low number of footholds. Writing a general controller that without further modifications from the start is by itself able to perform the required adaptations in order to avoid these drawbacks, and of achieving human or animal level locomotion versatility seams to be very difficult. In order to reach this goal, a more dynamical approach is necessary. The robot must be able to collect data from its experience and use it to learn which actions improve its performance. This is usually measured by a reward function. In other words, the problem at hand is to maximize the cumulative value of this reward - a problem tackled by reinforcement learning, a branch of machine learning.

By using the array of techniques provided by reinforcement learning, it is possible to have a robot progressively learning a task such as walking, for a fixed scenario. However, learning to perform a motor skill for a single environmental setting is normally not very useful. Even though in humans and animals most of the features of a motor skill are the same for different scenarios, it is usually necessary to attain some adaptations when the environmental specifications change. For instance, throwing balls of different weights into a basket requires changing certain movement parameters. In order to accomplish this kind of dynamic behavior, specific reinforcement learning algorithms capable of evolving generalized parameterized policies must be used in combination with proper movement representations.

Nevertheless, learning a complex motor skill involving issuing commands to a great number of actuators which depend on many different environmental variables is a task that may be intractable unless the learning process is properly guided and structured. Because of this, the aforementioned algorithms should be very flexible. It should be possible, for instance, to learn simpler movement primitives first, separately, and then resume the learning process with these primitives combined. Doing so decreases the search space initially, and then enables each primitive to adapt to the others so their combination works better in accomplishing the task in

hands. In addition to this, other operations and features such as storing primitives, initialize the learning process with known initial information and control the learning algorithm parameters at different stages are desired. These capabilities are certainly a required step in order for the emergence of motor skill learning software that can become mainstream and usable to anyone who works with robots.

Humans and animals are capable of executing key motor skills even when facing new conditions, like injuries or new terrains with different characteristics, so it is understandable for one to expect robots to follow the same trail. But in order for this scenario to become a reality, novel approaches and improved algorithms designed for real world use, able to make the best use of previously collected data, deal with partial and noisy information, and to employ the appropriate exploration/exploitation strategy are required. Only then will robots be able to achieve a whole new level of impact on society and a more deep involvement with humans daily life.

We hope this work contributes to the advancements on this field, however modest this contribution might be.

1.2 Goals

This thesis follows the PTDC/EEA-CRO/100655/2008 project aims. The carried out research has the general goal of exploring and extending state of the art motion learning techniques and to employ them in order to achieve improved and adaptive locomotion in a biped robot.

More specifically, the work on this thesis has two objectives:

- Extending and improving existing solutions for motion learning in robotics. This is done by complementing a state of the art algorithm for optimizing generalized motor policies and devising a framework that allows one to perform several operations that facilitate the process of learning complex motor skills.
- Show how a Central Pattern Generator (CPG) based controller of a biped robot can be improved so that it learns to adapt its movement parameters according to the slope angles of the underlying surface, using the developed framework in the process.

. The results of this research are also expected to bring usable empirical knowledge for those who aim to implement adaptive locomotion controllers.

1.3 Contribution

The innovation aspects of this thesis are twofold. First, a state of the art algorithm - CrKR - that allows to generalize parameterized policies is modified and extended into a new algo-

rithm named CrKR^{++}. The main modification enables to circumvent some previous limitations regarding the computational complexity of the original training operation, while other modifications are issued with the purpose of improving the overall effectiveness of the algorithm. This means that with CrKR^{++}, training sessions with high number of samples can be performed, a necessary feature when a policy is hard to generalize, i.e, a large quantity of data is necessary to find a good mapping between the state and parameter values to be used. CrKR^{++} effectively expands the range of problems where CrKR could be employed usefully.

Second, a framework that enables storing, combining and mutual generalization of parameterized policies is presented. This framework relies on the CrKR^{++} algorithm as its learning engine and provides several options that facilitate the process of learning complex motor skills. The offered features allow different learning scenarios to occur and conveniently provide the means to make the learning process flexible. Some of these scenarios include:

- Generalizing two policies at the same time - which can translate, for instance, into different body parts of a robot learning to react to different environment variables during the execution of a task. The generalized policies continuously adapt to each other as the learning process happens.

- Improve upon a stored policy, with new environment conditions, a different reward function or different algorithmic parameters.

- Load two policies, but evolve only one of them, while keeping the other constant. This can be used if one of the policies has already reached a satisfactory level while the other has not.

Because of such possibilities, the framework has been named Flexible Framework for Learning (F3L).

1.4 Methodology and execution

The work here presented is part of a wider context, giving continuity to the numerous projects related with robot locomotion carried out by the people in the Adaptive Systems Behavior Group (ASBG) lab.

The locomotion learning algorithms are tested in simulation software, namely Webots™, which provides extra flexibility and convenience. Under a virtual environment, it is possible to reproduce the same initial conditions for every test, avoid the need of human intervention in the several experiments performed and substantially increase the learning speed since it is faster to execute each test. It is also possible to perform manipulations in parameters and variables that could not be changed in an authentic physical environment. Webots™ is a

physics based simulation software comprehending the aspects of rigid body dynamics. The simulations performed in it are therefore expected to provide realistic results.

Additionally, when introducing modifications to existing algorithms, theoretical or empirical evidence is provided in order to justify such changes.

1.5 Outline

The chapters in this thesis are structured as follows:

- **Chapter 2** discusses the main obstacles that exist for the implementation of motor skills in robotics and the challenges inherent to applying reinforcement learning techniques to them. It then proceeds to introduce the base theoretical aspects of reinforcement learning and presenting relevant state of art techniques for reinforcement learning and movement representation.
- **Chapter 3** presents a set of techniques for improving the efficiency and effectiveness of the CrKR algorithm, with emphasis on reducing its computational complexity. The resulting algorithm after the introduced modifications is named $CrKR^{++}$.
- **Chapter 4** presents the F3L framework which provides a set of operations that facilitate the structuring of the learning process of complex policies.
- **Chapter 5** attests the capabilities of F3L by using the techniques provided by the framework to train a controller for DARwIn-OP, making it capable of generating stable locomotion in moderately sloped surfaces.
- **Chapter 6** concludes the thesis putting its contributions into perspective and giving an overview on future research lines and possible improvements that may follow.

1.6 Publications related with this thesis

- J. Macedo and C. Santos and L. Costa. "Using Cost-regularizied Kernal Regression with a High Number of Samples", pp.261 - 266, IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC) May 14-15, 2014, Espinho, Portugal

Chapter 2

Context and related work

2.1 Problems and challenges

In this section, some of the most prominent difficulties that are faced when trying to implement motion learning in robotics, and in particular locomotion learning, are described.

2.1.1 The challenges in locomotion

Animals and humans perform locomotion in a very natural and spontaneous way. One could naively question how the implementation in robots of such an apparently easy task can be said to be so difficult, particularly for legged robots. However, after analyzing the problem, it is not hard to come to the conclusion that there is a high complexity inherent to the locomotion process. A brief overview of the numerous challenges that the implementation of locomotion in robots poses is provided next.

Environment Successfully achieving locomotion in diverse environments, specially outdoors where the terrain properties are more variable, requires accounting for several aspects. Firstly, the terrain's surface can have a non zero slope angle. This slope can also be parallel or perpendicular to the robot's trajectory. It may be necessary to modify the walking gait according to the slope angle, specially for robots with a low number of footholds where there is a higher chance of falling. This may also require treating potentially noisy information coming from the robot sensors in order to calculate the slope angle. Terrain and feet interaction in terms of friction can also vary: walking on a slippery terrain like ice is likely to require a different behavior compared to walking on top of concrete. Terrain can also be highly irregular or exhibit different deformation properties: sand and snow are two of the many examples of such diversity. Finally, a robot put in an uncontrolled environment has to deal with static and

dynamic obstacles which are likely to appear. The BigDog robot [Raibert and Blankespoor, 2008] is already capable of dealing with many of the aforementioned difficulties: it can walk on different types of terrain like mud and snow and on inclined surfaces. Also, [Kalakrishnan et al., 2009] shows LittleDog robot learning locomotion on rough terrain.

Robot's integrity It is important to minimize the risk of the robot being damaged or put into an irreversible state. Not only this will likely result in the robot's task not being accomplished, it may imply severe costs, since robots are frequently made of expensive materials and hardware. In order to ensure the robot's physical integrity, there are several considerations to have. Keeping balance is perhaps the number one issue related with locomotion in this matter. When a robot looses its balance due to a defective gait and falls, damage may be caused upon impact. Besides, if a stand up procedure is not available, human intervention is required in order for the robot to resume its task, which is inconvenient and may not always be possible. Trying to predict a fall and minimize its damage is also a relevant matter. Balance is not the only issue to consider, however. While performing locomotion, the robot needs to avoid collisions with obstacles. An unpredicted collision at a high speed can also cause severe damage. In [Rubrecht et al., 2012], a methodology to ensure safe behaviors of multibody robots in reactive control frameworks is shown.

Safety Ensuring the safety of the people who interact with a robot is an essential requirement. For small robots, this is not a real concern. However, as robots get bigger, stronger and faster, it is important that their movements do not harm people around. A big or fast enough robot can hurt someone if it falls or goes against him.

Hardware The mechanical properties of a robot play an important role in several aspects of locomotion. While most commercially available robots use stiff actuators, these have some drawbacks in the context of locomotion. Stiff actuators offer great precision. However, in walking and running activities, certain characteristics that stiff actuators may fail to deliver like good shock tolerance, less damage during inadvertent contact, more stable force control and potential for energy storage are of extreme importance. Recently, compliant actuators systems which resemble muscle-tendon systems in animals started to be used in legged locomotion research. These have shown to be capable of achieving promising results [Spröwitz et al., 2013].

Fulfilling the tasks Locomotion per se is not usually the goal of a task. Normally, it is a secondary activity needed to be performed in order to accomplish a more complex mission. The requirements of such a mission may impose difficulties in the locomotion exercise. Carrying a load, putting out a fire or helping a blind person navigate in a city are all possible

scenarios where additional considerations may be required in order to maintain balance and effectively achieve success. Having these considerations into account further increases the difficulty inherent to the locomotion process.

Nature has found ways to overcome these challenges. As a result, many of the methods used in robotics are bio-inspired.

2.1.2 The challenges in learning

Applying reinforcement learning in the context of motion learning is not a trivial task. There are a number of theoretical and practical obstacles which need to be addressed in order to implement learning in a physical world. Some of these obstacles are presented in the following paragraphs inspired on [Kober, 2012].

Dimensionality A typical robot has usually a high number of degrees of liberty. For instance, the DARwIn-OP robot has 20 actuators, which would translate in a 20-dimensional action space. Considering only the robot itself, the state space would have 20+20 (positions and velocities) dimensions, but depending on the problem, this number can be greater. For other robots like hexapods or octapods, for instance, the dimensionality of the problem can be even more explosive. Moreover, all these variables are continuous, which prevents the use of algorithms conceived for discrete contexts, at least directly. The problem of dealing with such a high number of dimensions is known as the *curse of dimensionality*. To use reinforcement learning to learn motion in robotics, smart representations of movement must be used alongside with proper algorithms conceived to deal with high dimensions. Naive techniques are unlikely to succeed, either because they would take an unacceptable amount of time to execute or simply because they would fail to converge in finding good solutions.

Testing a solution Unlike a software context where generally testing a solution is extremely fast and requires few effort, when learning motion primitives in real robots, it may take seconds, or even minutes to properly test a policy. The experiment needs to be prepared, the robot needs to execute the policy for a certain period of time in order for valid conclusions to be taken and in the end, if the process is not automated in some way, human intervention is required in order to put everything in place for a new test to start. It is difficult to replicate the same conditions in every test, auxiliary material can be required in the experiments, and the robot can be damaged if not supervised. All these drawbacks make it unpractical to run a reinforcement learning algorithm in a real, physical environment. For these reasons, often simulators are used to do this. However, the physics engine and the virtual world used in simulation frequently contains inaccuracies that cause the achieved solutions not to perform

so well in the real world. There is no definite way to deal with this, although there are ways to minimize the problem. For instance, a solution can be obtained in simulation and then improved in a real environment. Another technique is to introduce noise in actuators and sensory information in order to find solutions that can compensate for the inaccuracies inherent to the simulation software. Even though such possibilities exist, real world testing still remains a hard to avoid challenge in robot motion learning.

Specifying the reward function The reward function is what allows a reinforcement learning algorithm to work. It is based on it that the algorithm is able to effectively learn what is a good or bad solution. For this reason, designing the reward function is an extremely important part of solving a reinforcement learning problem. Such a task involves several considerations: while giving a reward only when the robot does what it is intended (e.g scoring a goal in robot football) may look like the obvious approach, such may happen so rarely that a reward is almost never given. This will result in the agent not being able to understand when a solution is better then other (even if both are bad) which is necessary for improvement to exist. A reward function that encompasses the notion of closeness to the desired goals is usually a better approach because it is able to gradually guide the learning process to a reasonable solution. However, constructing such a function is also a process that demands attention to certain details. It is necessary to define how each secondary objective has influence in the reward as well as how undesired actions penalize its value. Before a poorly constructed reward function, the learning process may achieve solutions that explore its faults instead of reaching the real intended goals. Another possible issue is related to the tendency that the reward function has to lead to a premature local maximum. This may trap the learning process in an early stage, and hinder the improvement of solutions.

Previous knowledge Being capable of using previous knowledge in order to acquire new movement skills or generalize existing ones is a greatly desired feature in robotics. Humans are capable of improving a previously acquired skill for a different set of conditions different than the ones verified during the period in which such skill was learned and are also capable of mixing learned movement primitives by sequencing and superposition. Having the capability of building on top of previous knowledge in robotics would allow for constant improvement of the performance in several tasks, importing knowledge between different robot models and make a better use of computational resources by reusing past acquired data.

2.2 Reinforcement learning

2.2.1 Notation and problem formulation

The problem of automatic learning of motion primitives in robotics is usually approached using reinforcement learning techniques. Before explaining the particularities of this field of machine learning and in order to contextualize the topics explained in this section, a quotation from [Sutton and Barto, 1998] is presented:

> The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

Reinforcement learning studies methods to improve solutions for problems based on a reward function which evaluates how good is a given solution in quantitative terms. The reward function can be based on a number of factors depending on the problem. It can be the number of goals scored for a football playing robot, the quantity of energy saved during the execution of a mission, or the success rate of a checkers strategy.

Unlike other machine learning techniques, the learner agent is not given examples of which decisions to take under a set of situations or training examples; instead, it must be able to find out by itself good solutions using the reward function as a "guide". For this to happen, the agent must also take the initiative to explore, this is, to make decisions different from what its previous knowledge indicates to be the best option. This is needed in order to find out if such actions may lead to an even greater reward. Deciding whether to explore or "play safe" according to known data is known as the exploration/exploitation dilemma.

An interesting fact that adds an even greater depth to the problems that reinforcement learning studies, is that the action that results in the greatest immediate reward may lead to worst future rewards when compared with other apparently not so good actions. In other words, the option with greatest instant benefits may not be the best in the long-term. The notion of value function helps to understand what the best decision really is. The optimal value of a state is the maximum reward possible to accumulate starting from that state. So the

best decision is reduced to performing the action that leads to the state with greatest optimal value. This concept will be further explained later on.

In reinforcement learning, for an agent–environment interaction context, the *agent* makes decisions by selecting which actions to perform at each time step, and the environment responds to these decisions, leading the agent to a new state. There is a continuous interaction between both sides. At each time step t, the agent receives a representation of the environment's state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states. Based on the state, the agent decides to take an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of actions available in state S_t. In $t+1$, as a consequence of its decision, the agent receives a reward, $R_{t+1} \in \mathbb{R}$, and the transition to a new state S_{t+1} happens (Fig. 2.1).

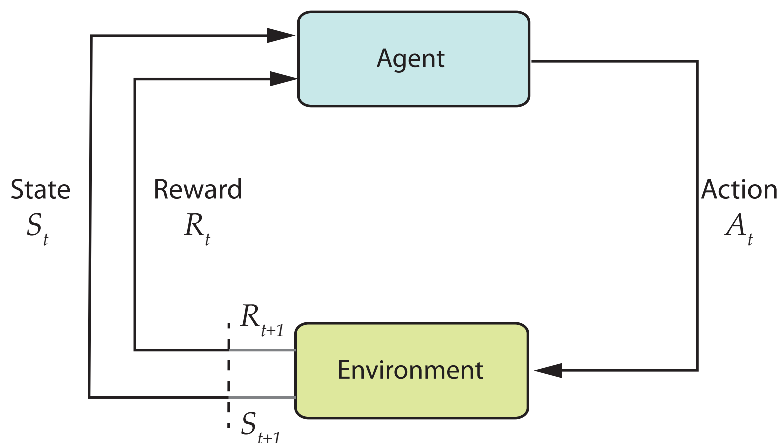


Figure 2.1: Agent–environment interaction from a reinforcement learning perspective. Image based on [Sutton and Barto, 1998].

In order to decide which action to take in a certain state S_t, the agent employs a policy π_t. This consists in a function that expresses the probability of taking an action a if the state is s and is represented by $\pi_t(a|s)$.

For many cases in reinforcement learning, one can define the state in such a way the problem can be modeled as a Markov Decision Process (MDP). For this to be possible, the following property must be verified:

$$P(R_{t+1} = r, S_{t+1} = s'|S_t, A_t) = P(R_{t+1} = r, S_{t+1} = s'|S_0, A_0, R_0, S_1, A_1, R_1, ..., S_t, A_t) \quad (2.1)$$

This means that the probability of arriving to a certain state and receiving a certain reward only depends on the last state and action. Even for cases where this property does not occur immediately, it is often possible to perform transformations on the problem formulation, namely in the definition of what is the state, so that it verifies.

In order to define the value of a state in a finite or infinite horizon, a discount rate λ can be introduced to determine the present value of future rewards such that $0 \leq \lambda \leq 1$. If $\lambda = 0$, only immediate rewards are valued, and all future rewards are accounted as 0. If λ is given the value 1, future rewards are considered in the same way as the immediate reward. The value function may then be defined formally as:

$$v_\pi(s) = E_\pi \left[\sum_{k=0}^{T} \lambda^k R_{t+k+1} | S_t = s \right] \tag{2.2}$$

where E_π denotes the expected value if the agent follows policy π, t denotes the current time step and T is the number of time steps to be considered in the evaluation horizon. In order for this summation to converge in an infinite horizon (when $T = \infty$), λ cannot be equal to 1.

Solving a reinforcement learning problem comes down to finding the optimal policy. The optimal value of a state $v_*(s)$ is the value of that state when using the optimal policy:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.3}$$

The Bellman equation states that $v_*(s)$ can also be expressed as:

$$v_*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s'|s,a)[r(s,a,s') + \lambda v_*(s')] \tag{2.4}$$

Under this result, determining the optimal policy when the value function is known is relatively easy: "For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy" [Sutton and Barto, 1998]. Equation (2.4) can be stated as follows: the optimal policy is the one where the chosen actions reflect the best compromise between the state-transition probabilities and the sum between the optimal values of those states (pondered with λ) with the rewards which would result from those transitions.

This whole mathematical framework and equation (2.4) serves as a basis for many reinforcement learning algorithms used in motion learning. Some of them will be presented later on.

2.2.2 Progress and evolution of reinforcement learning algorithms

As presented in section 2.2.1, a policy π is the function that dictates the action that the agent must take in response to the state of the environment. However, trying to work on the problem of motion learning from such definition may not be very practical in a context where the variables are continuous and the number of dimensions is so high. In alternative, one can express a policy as a function of a set of parameters θ, which results in a parameterized function $\pi(\theta, s)$ or $\pi_\theta(s)$ where s is the state. The episodic reward function becomes then $R(\tau(\pi(\theta, s)))$ where τ is a trial performed by the policy [Caldwell, 2012]. Such expression can also be simply shortened to $R(\theta)$ if the same initial conditions are considered for each trial. This conveys the idea that the reward is directly dependent on the parameters θ which modify the agent behavior, and that it is through the change of these values that the outcome of the agent's policy can be improved. By choosing an appropriate set of parameters, it is possible to reduce the dimensionality of the problem and to represent only the policies that follow a structure that is more suitable for learning a desired task (Fig. 2.2). On the other hand, there may be policies that are impossible to represent with only those parameters. Nevertheless, in most cases the benefits of using a parameter based representation for learning a motion task outweigh the drawbacks.

Figure 2.2: Marionette analogy. A few strings (parameters) are enough for the manipulator to produce the intended (high reward) movements (policy).

Even though an appropriate choice of parameters can greatly improve the learning time, maximizing $R(\theta)$ still implies searching in a high dimensional and continuous space. It is unfeasible to apply conventional numeric methods for this task, since evaluating R for a given θ demands a great deal of time in an optimization context as mentioned in section 2.1.2. Minimizing the number of required reward evaluations has become one of the main considerations when conceiving new learning algorithms to be used in robotics.

Many policy improvement methods use an iterative process of exploration where policies generated from perturbation of a basis policy (in this case θ_i) are executed K times. Each of these trials is named a "rollout". As an outcome the trajectories $\tau_{1,2,..,K}$ are generated. These contain all information about each trial that is used to update the parameter values to new

ones. So in each iteration i, we have:

$$\theta_{i+1} = \theta_i + \Delta\theta \tag{2.5}$$

The process, which can be observed in Fig. 2.3, is repeated until the desired reward is obtained from the most recent parameters set.

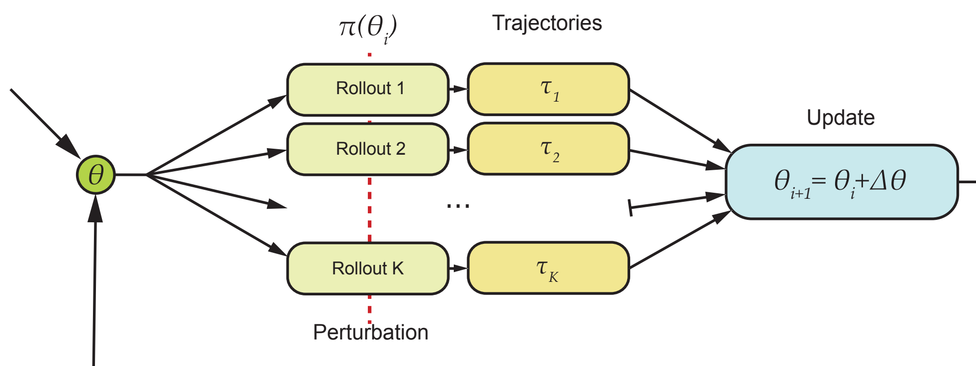


Figure 2.3: Typical policy learning algorithm steps.

Most policy improvement algorithms obey this mold, but of course, each one differs from others in several aspects. In [Stulp and Sigaud, 2012b], several algorithms are analyzed and a "fact sheet" is written for each of them, classifying it regarding the perturbation method, the data recorded for each trajectory, the update method and the policy search method (actor critic or direct policy). When looking at different fact sheets , it becomes apparent that there is a trend in the evolution of the algorithms (Fig. 2.4). Recent algorithms use a parameter perturbing methodology (as opposed to an action perturbing one) and perform the update following reward averaging methods (as opposed to gradient based ones).

A brief overview over some of the algorithms can help justify this trend [Stulp and Sigaud, 2012b]. An example of a not so recent one that will serve as a starting point is reward increment = nonnegative factor $\times$ offset reinforcement $\times$ characteristic eligibility (REINFORCE) [Williams, 1992]. REINFORCE fosters exploration through the use of a stochastic policy: the policy is executed K times with the same parameters; the state, action and reward at each time step are saved in the trajectories τ; due to the stochastic character of the policy, its nominal output - motor commands - is perturbed. The new parameters are then calculated based on an estimation of the gradient. A problem with this algorithm is that it requires a large number of rollouts to perform a parameter update, and doesn't make very efficient use of samples since the trajectories information cannot be used for later updates. Episodic Natural Actor Critic (eNAC) [Peters and Schaal, 2008] addresses this issue by estimating a value function V_{π_θ}, as more compact representation of long term-reward then $R(\tau)$, using it

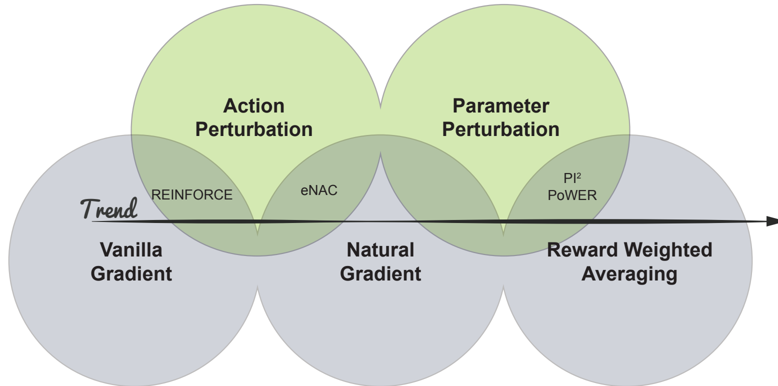


Figure 2.4: Trend in some policy improvement algorithms: reward increment = nonnegative factor × offset reinforcement × characteristic eligibility (REINFORCE) , Episodic Natural Actor Critic (eNAC) , Policy Improvement with Path Integrals (PI^2) and Policy learning by Weighting Exploration with the Returns (PoWER).

to perform the parameters update. The advantage of using a value function is that it allows some level of generalization: it is capable of providing estimates for the reward that a certain parameters set θ will produce based on information from previously obtained trajectories that used other parameter values. Because it estimates a value function to perform its parameter updates, eNAC is classified as an actor critic method.

REINFORCE and eNAC are both action perturbing methods, since the perturbation occurs every time step at the nominal commands level, i.e, $u_t = u_t^{\text{nominal}} + \epsilon_t$. There are several disadvantages when the perturbation is performed in this way: ***i)*** There is no dependency between the nominal commands generated in consecutive time steps, which leads to noisy trajectories in the action-space; ***ii)*** Consecutive perturbations may nullify each another; ***iii)*** The hight frequency variations that can happen in the issued commands may lead to dangerous behavior and cause damage to the robot.

Recent algorithms such as Policy learning by Weighting Exploration with the Returns (PoWER) [Kober and Peters, 2010] and Policy Improvement with Path Integrals (PI^2) [Theodorou et al., 2010b] use a different perturbation method: parameter perturbation (Fig. 2.5).

In this scheme, it is the parameters of the policy that are perturbed instead of the nominal commands. So we have $\pi_{\theta+\epsilon}(s)$ instead of $\pi_\theta(s) + \epsilon$. When using an appropriate movement representation, this method is able to avoid the issues present in action perturbing methods. Another distinguishable feature of these algorithms relative to eNAC and REINFORCE is the update step. ENAC and REINFORCE estimate gradients to calculate the update $\Delta\theta$, an approach which may not be robust when dealing with noisy reward functions. Besides, it is not trivial to tune the learning rate α, a parameter which has a great impact on the performance of these algorithms. PoWER and PI^2, on the other hand, calculate the update $\Delta\theta$ using a weighted average of the perturbation vectors $\epsilon_1, \epsilon_2, ..., \epsilon_K$ applied to θ and tested in the various rollouts.

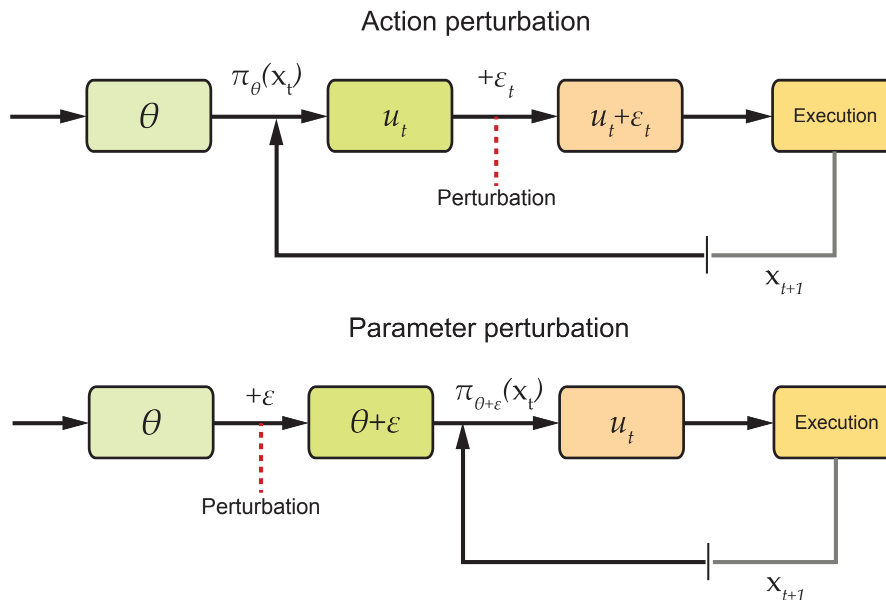


Figure 2.5: Different perturbation schemes

The weights given to each perturbation vector are based on the rewards achieved during the resulting trajectories $\tau_1, \tau_2, ..., \tau_K$. Such an approach results in a more stable update rule, less dependent on the turbulence of the reward function.

PI^2 and PoWER algorithms have shown to perform better than most older techniques, sometimes by orders of magnitude [Theodorou et al., 2010b]. Several other algorithms that were influenced by PI^2 and PoWER that brought even further improvements, such as Path Integral Policy Improvement with Covariance Matrix Adaptation (PI^2-CMA), have since been published.

Many complex tasks have already been achieved with the mentioned algorithms such as pancake flipping and archery aiming [Kormushev et al., 2013], balancing [Vlassis et al., 2009], ball in a cup [Kober and Peters, 2010], locomotion [Shen et al., 2012], jumping tasks [Theodorou et al., 2010b], pouring liquid to a cup [Tamosiunaite et al., 2011], pushing a door and picking objects [Kalakrishnan et al., 2012].

2.2.3 State of the art reinforcement learning algorithms

In this section a non-exhaustive list of reinforcement learning algorithms that can be used in different contexts, and in particular in motor skill learning, is presented. All of these have in common the attempt in reducing the number of reward evaluations necessary to be performed in order to improve the initial policy of the agent.

Policy learning by Weighting Exploration with the Returns (PoWER) Presented in [Kober and Peters, 2010], PoWER borrowed the principle of expectation-maximization from other machine learning branches in which the update rule for the parameters tries to maximize the lower bound on the expected return of the policy. According to the authors, this approach outperforms many previous well known methods, like Vanilla Policy Gradients, Finite Difference Gradients, Episodic Natural Actor Critic and Reward-Weighted Regression. This algorithm has been used to learn tasks like "Under-actuated Swing Up" - moving a hanging heavy pendulum to an upright position and stabilize it there in minimum time, and "Ball-in-a-Cup" - swinging and catching a ball in a cup with fast movements.

Policy Improvement with Path Integrals (PI^2) This is a method of probabilistic reinforcement learning, introduced in [Theodorou et al., 2010b] and derived from the framework of stochastic optimal control and path integrals. According to the authors, it makes an appealing theoretical connection between value function approximation using the stochastic HJB equations and direct policy learning by approximating a path integral. The final form of the algorithm is simple and has no open algorithmic tuning parameters besides the exploration noise while exhibiting numerically robust performance in high dimensional learning problems. PI^2 has been used, for instance, to optimize a jumping maneuver across a gap in a robot dog and to optimize the movement of a 10 degrees-of-freedom arm trying to pass through a via-point whilst minimizing joint accelerations. Later, PI^2 variations like PI^2-CMA [Stulp and Sigaud, 2012a] and Policy Improvement through Black-box Optimization (PI^{BB}) [Stulp and Sigaud, 2012b] emerged, further improving different aspects of the algorithm.

Cost-regularized Kernel Regression (CrKR) This algorithm was introduced with the objective of providing a tool to generalize motor primitives for difference instances of the same task [Kober et al., 2012]. It allows learning the mapping between the variables that characterize the instance of the task in hands and the values of the parameters of the policy to be executed in order to accomplish it.

In each iteration, the algorithm adjusts the agent's policy and variance according to the cost obtained from the trial performed using the last dictated parameter values. If the cost is high, the policy will only shift slightly towards those values around the state where the system was at the time of the trial. The variance will likely stay high around those state values, meaning that the achieved policy is still not satisfactory there, and more exploration is required to achieve low costs. If on the contrary, the cost suffered is low, the policy will heavily shift towards the used parameter values around the state where the system was at the time of the trial and the variance will be reduced, since the policy is becoming acceptable around the state tested. Each additional sample serves to refine the policy and adjust the exploration of the algorithm.

CrKR stood out at optimizing the metaparameters of DMPs used in striking movements in robot table tennis [Kober et al., 2012].

Mixture of Motor Primitives (MoMPs) This framework allows to combine several movement primitives learned by imitation in order to solve a complex motor task [Mülling and Kober, 2013]. Each primitive stored is associated with a set of parameters, the augmented state, that describe the situations occurred during demonstration. Using such information, and for a new situation, the primitives are combined using a gating network that weights the contribution of each primitive in the execution of the final movement. The weights of the gating network and the primitives themselves are updated during the learning process.

Remarkable results have been achieved with Mixture of Motor Primitives (MoMPs) when the technique was used to teach a robotic arm to play table tennis.

Reinforcement Learning based on Particle Filters (RLPF) This algorithm, presented in [Caldwell, 2012], is based in particle filters , a technique used in statistics for estimation of an unobservable underlying probability density function based on observed data. It constitutes an exception when compared to most reinforcement learning algorithms, in the sense that it performs global optimization instead of local optimization and does not obey the typical update rule. The motivations behind this approach, according to the authors, are to introduce an algorithm that is capable to go past local sub-optimal solutions and whose effectiveness is not as largely dependent on the provided initial policy. Reinforcement Learning based on Particle Filters (RLPF) has surpassed PoWER in the comparative tests conducted by its authors.

2.3 Movement generation and representation

Typically, one or more layers of abstraction are employed by a control program in order to issue commands to robot actuators. These layers may be important for several reasons: they can allow for a more intuitive way of representing movements for those who work with the robot; they can incorporate logic to control stability and safety of the robot, and finally, they can work as a mean of reducing the number of representable policies [Kober, 2012] which is specially important when performing reinforcement learning on top of these layers/movement representations. This is the case because the convergence speed of the algorithms can be greatly accelerated when smaller state and action spaces are in game. However, in order to find a good solution for a given problem, the policy parameterization cannot also be too limiting. An overly simple parameterization will certainly facilitate the convergence of the algorithms involved in the learning process, but at the risk of preventing solutions with acceptable quality to be achieved. It is therefore important, for each problem, to find the right compromise where the level of sophistication of the policy parameterization is enough to provide a rich

and flexible representation that can encode good solutions, but at the same time does not fall under the *curse of dimensionality*.

In the particular case of locomotion, these layers may also include the generation of individual movements needed to carry out a high level command such as walking in a certain direction.

Some of the techniques which work at the described level and that are used for movement generation are mentioned next. Note that these techniques are not mutually exclusive, being sometimes combined in order to achieve a desired goal.

Zero Moment Point (ZMP) A number of model-based locomotion controller implementations uses the zero moment point notion in gait planning and control. ZMP is the point where the reaction force between the foot and the ground does not produce any moment in the horizontal plane, keeping vertical inertia and gravity forces equal to zero. By ensuring the appropriate dynamics of the mechanism above the foot, it is possible to maintain dynamical balance [Vukobratović and Borovac, 2004] , assuming the ground provides enough friction. Examples of locomotion implementations based on ZMP related principles are [Hirose and Ogawa, 2007] and [Heo et al., 2012].

Central Pattern Generators (CPGs) While ZMP based approaches remain amongst the most popular for biped walking, they frequently require models or previous knowledge about the dynamics of the robot as well as the environment surrounding it in order to be employed. Such knowledge may only be partially available, something which may hinder the general application of these methods [Matos and Santos, 2012]; moreover, CPGs based approaches typically have a lower computational cost [Nor and Ma, 2013]; these are some of the arguments from those defending a bio-inspired approach based on CPGs.

Central pattern generators are neural networks present in both mammals and invertebrates that produce rhythmic output patterns of neural activity which intervene in actions like swallowing, respiration, walking, swimming and flying, between others. The first findings of locomotor CPGs occurred in 1911 [Brown, 1911] in experiences with cats. Since then, definite evidence of CPGs on other animals such as dogs and rabbits has been found; in the case of humans, only indirect evidence has been registered [Dimitrijevic et al., 1998]. Recent research [Büschges and Borgmann, 2013] has also extended the concept of a modular neural network organization for locomotion from invertebrates and lower vertebrates to mammals.

In the past, it was observed that a cat whose fore and mid-brain has been removed was capable to stand supported on a thread-mill, and transition between walking and running gaits while induced with electric stimulation of the mid-brain. The fact that the characteristics observed in these movements were the same when compared with a normal cat was quite significant and eventually the observed properties of CPGs captured the attentions of

roboticists.

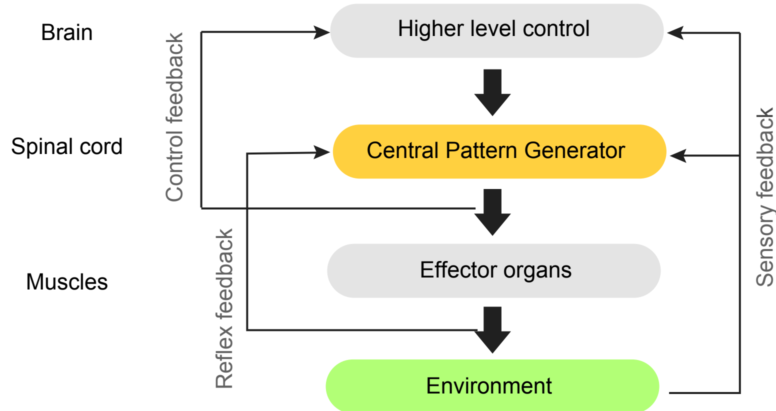


Figure 2.6: Simplified diagram of a bionic CPG control process. Image based on [Nor and Ma, 2013].

There are several examples of works regarding the implementation of locomotion using CPG inspired controllers. These works include, for instance, a salamander robot which can smoothly transition between walking and swimming [Ijspeert et al., 2007] and a quadruped robot (AIBO) performing omnidirectional locomotion [Matos and Santos, 2010]as well as generated walking gaits for hexapods [Cappelletto and Estévez, 2007] and bipeds [Morimoto et al., 2008].

CPG models have been designed at different levels of abstraction from detailed biophysical models to abstract systems of coupled oscillators [Ijspeert, 2008]. This last approach is more in line with typical implementations of CPGs in robotics: for instance, the common techniques when implementing CPGs for biped walking use neural oscillators or phase oscillators whose output is fed into pattern generation layers [Matos and Santos, 2012] . In the latter, the mathematical implementation translates itself into a dynamical system of equations, where the phase of the movements is governed by the oscillators. This system describes the relationships between different interveners, namely the movement parameters, the oscillator inputs, and the outputs - which can be positions or velocities in joint or task space, or torques. Normally, there is an oscillator for each body "division", which depending on the case, can range from a single joint to a whole member. These oscillators are coupled in order to establish synchronization between the different body parts during the gait's execution. Fig. 2.7 shows an example of such implementation performed on a DARwIn-OP robot.

CPGs modeled this way offer an attractive framework for learning and optimization algorithms: they exhibit limit cycle behavior, being capable to recover from small perturbations - which are common to occur in a non controlled environment; they usually have a small number of control parameters, which reduces the dimensionality of the solutions to be optimized

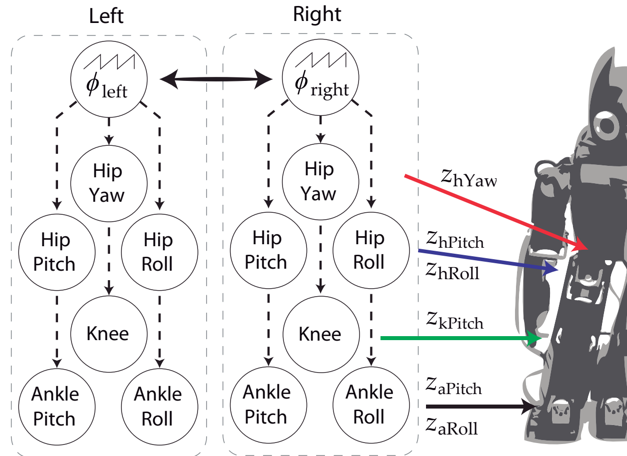


Figure 2.7: CPGs and corresponding phase oscillators, motion generators and corresponding joints. Bilateral coupling is represented by the bilateral arrow. In the labels, h stands for hip, k for knee, and a for ankle. Image adapted from [Matos and Santos, 2012].

and accelerates the convergence of the learning algorithms; when the value of the parameters is altered, even if abruptly, the produced movements are modulated smoothly, and finally, they are well suited to be integrated with sensory feedback mechanisms.

Dynamic Motion Primitives (DMPs) DMPs constitute a simple way of representing either rhythmic or discrete movement trajectories through the use of non linear dynamical systems. The characteristics of these systems are such that the produced trajectories reveal an attractor behavior, which is relevant when it is intended to produce movements which must evolve towards either a point or a limit cycle attractor. This is the case of a generic walking gait where a baseline rhythmic behavior is excepted. DMPs were proposed by Auke Ijspeert [Ijspeert, 2002] as a solution to obtain stable and flexible movement representations that could be subjected to improvement by reinforcement learning algorithms. They were first used in a context of learning a set of movements for a humanoid robot from demonstrations of a human teacher.

DMPs contemplate a linear dynamical system's component which shapes the attractor landscape (spring-damper system), and a non linear one, which influences the first with a perturbation [Haschke, 2012]. It is through variations in this perturbation that varied trajectories can be created: the non linear component consists on a summation of Gaussian functions, each with an associated weight. By varying the number of Gaussian functions, their centers, and the associated weights, one can generate the different trajectories.

DMPs have been used by a number of researchers. In [Theodorou et al., 2010b] , learning algorithms are used upon a DMPs movement representation in order to train a quadruped robot to jump over a gap. In [Theodorou et al., 2010a], the authors use DMPs to represent

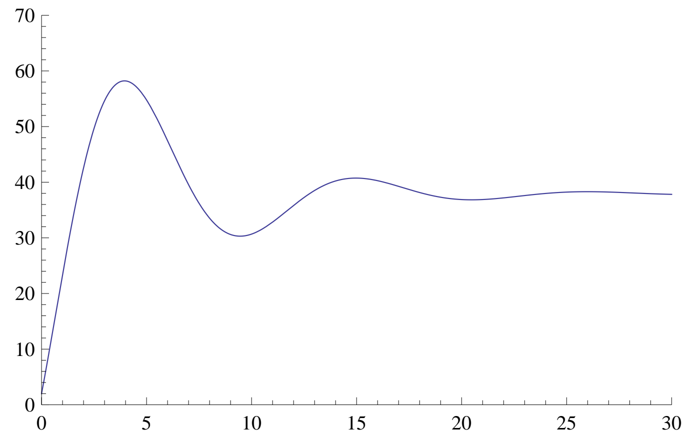


Figure 2.8: A discrete trajectory produced by a DMP.

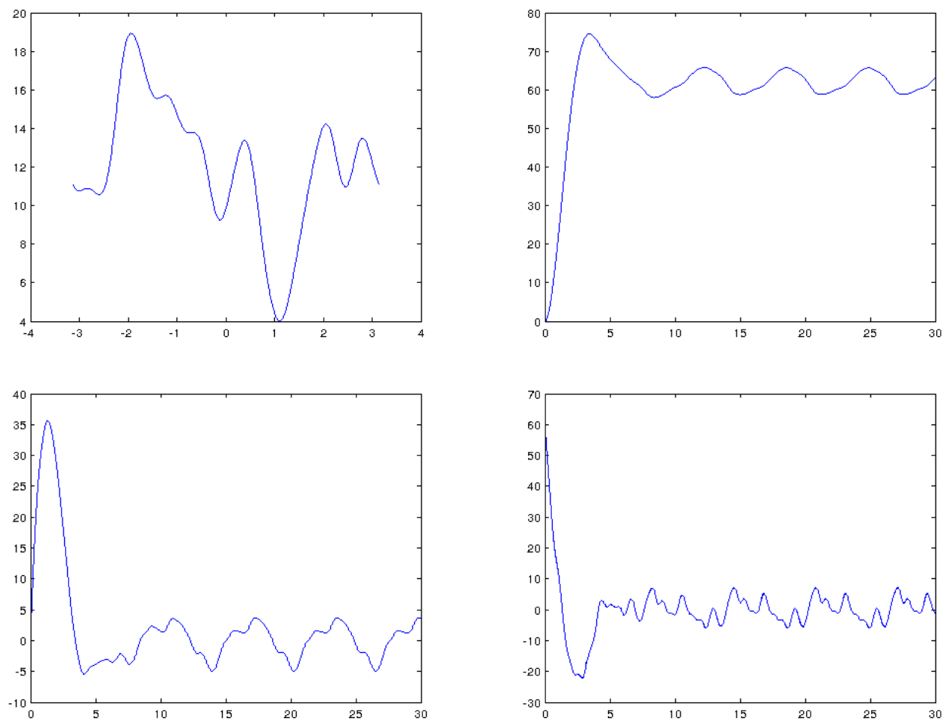


Figure 2.9: Different rythmic movement trajectories produced by DMPs.

throwing movements of a dart launcher, using learning algorithms to learn to set the DMPs parameters according to the position of the target. In [Mülling and Kober, 2013], a method based on mixing different DMPs learned from imitation is used to teach a robotic arm to play table tennis. In [Pongas et al., 2005], DMPs are used to represent rhythmic movements to

perform a drumming task.

DMPs offer many advantages: they can encode complex trajectories; they exhibit attractor dynamics (which results in tolerance to small perturbations in the executed trajectory); the encoded trajectories execution progress can be manipulated through the phase variable, and finally, their limit cycle anchor point or position can be changed online leading to smooth transitions in the produced trajectories to the new state. For these reasons, DMPs can be used in a wide range of situations and serve as a suitable movement representation to be used in learning tasks.

Splines Splines are a relatively popular concept in many subjects, like computer graphics and statistics. They can be seen as a method of constructing arbitrarily complex functions using piecewise-defined polynomial functions that connect smoothly. Different types of splines exist, although the most commonly used variations are the cubic spline and cubic Bézier splines.

In [Shen et al., 2012], splines are used to encode the trajectory of each servo on a quadruped robot while PoWER optimizes their parameters. First, splines with a reduced number of knots are used, and as more rollouts are executed and the policy is improved, more knots are added, gradually improving their representational power.

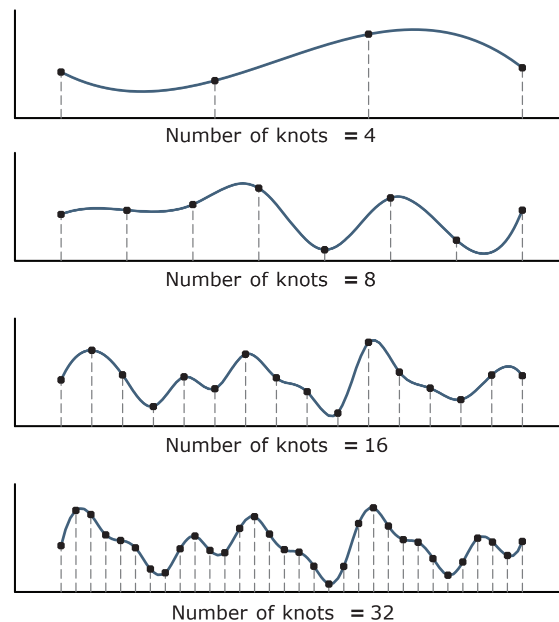


Figure 2.10: Splines with different number of knots. As the number of knots increases, so does the representational power of the spline. Image adapted from [Shen et al., 2012].

While the possibility of dynamically adding more complexity to the policy representation during learning is certainly advantageous, common spline-based approaches also exhibit some

drawbacks like explicit time indexation, thus becoming highly sensitive toward unforeseen perturbations in the environment that would disrupt the normal time flow [Ijspeert, 2002].

Reflexes There are also some authors who have implemented reflex based locomotion controllers. In this context a reflex is a motor response that is triggered by a certain event like a collision with an obstacle or loss of ground contact. The chain of movements resultant from reflexes originates the walking gait. Example of works that employ reflexes are those of [Klein and Lewis, 2012] and [Kimura et al., 2007]. Reflexes have the advantage of being fully feedback oriented, i.e, the behavior that is generated is fully dependent on the situation the agent is facing at each instant.

Mixture of motion primitives In [Mülling and Kober, 2013], a representation scheme based on the weighted combination of DMPs is used. Each DMP is learned via imitation learning for a specific situation. The combination of several DMPs allows to cover a wide set of states and generalize the knowledge acquired via imitation learning: depending on the conditions that the agent is facing at each moment, different weights are given to the several DMPs. This representation scheme is deeply bounded with the learning algorithms used with it, in which the relation between the state of the system and the weights to attribute to each DMP is learned.

Chapter 3

Improving on Cost-regularized Kernel Regression

Particularly in the last years, we have assisted to the development of improved reinforcement learning techniques capable to cope with the high dimensional and continuous character of the action and state space variables present in the context of motor skill learning. The PI^2 [Theodorou et al., 2010b] and PoWER [Kober and Peters, 2010] algorithms are good examples of this. When combined with a suitable movement representation, a number of tasks can be learned with these algorithms such as pancake flipping and archery aiming [Kormushev et al., 2013], ball in a cup [Kober and Peters, 2010], locomotion [Shen et al., 2012], jumping tasks [Theodorou et al., 2010b], and pushing a door and picking objects [Kalakrishnan et al., 2012].

Algorithms that made it possible for a robot to be capable of learning a movement that successfully accomplished the intended goals for a certain instance of a task were an important milestone in robotics and reinforcement learning. However, such capability alone is not enough in order to answer to most challenges when learning motor skills if the learned movements are not expandable to other instances of the tasks they are supposed to fulfill.

Humans appear to be able to learn movement templates, also called movement primitives [Schmidt and Wrisberg, 2000]. That is, they are capable of learning to execute movement tasks for an entire set of instances at once. For example, it is predictable that after a human learns how to throw a ball in order to hit a target in a certain position, it will take him less time to learn how to execute the same task for targets in other positions. This reveals a capability of generalizing the learned knowledge by adapting general parameters of the movement for related situations, while keeping its overall shape.

Mathematically, learning the optimal parameters to be used for a specific instance of a task,

is an optimization problem that can be formulated simply as:

$$\max_{\theta} \quad R(\pi_\theta) \tag{3.1}$$

where θ are the parameters to optimize, π is the agent's policy and R is the reward function. Because only a specific instance is being considered, the reward is ultimately only dependent on the chosen parameters. On the other hand, the problem of learning to adjust the parameters in response to a set of state variables is formulated as:

$$\max_{\gamma} \quad \int_{\mathbb{S}} p(s) R(\pi_{\gamma(s)}) \tag{3.2}$$

where $p(s)$ is the probability of the agent being in state s, $\pi_\gamma(s)$ is the agent's policy, R is the reward function, $\mathbb{S}$ is the state space and γ is the state to parameters mapping function $S \xrightarrow{\lambda} P$ where P is the parameters space. Here the problem is much more complex, since a whole function needs to be learned instead of a single set of constant parameters.

Manually defining the rules of this kind of adaptive behavior is tedious, and frequently near impossible when dealing with highly complex robotic systems and tasks, since these require several groups of actuators to work synergistically and in different ways according to the faced situation. It is therefore important to research algorithms capable of learning the mapping between the variables that characterize the instance of a given task and the values of the parameters to be set in order to accomplish it.

3.1 Cost-regularized Kernel Regression as a suitable algorithm for learning generalized policies

CrKR [Kober, 2012] is an algorithm that can learn a state to parameters mapping. It has been successfully used in the learning process of several tasks such as dart throwing, ball throwing and table tennis with different robots. It has also outperformed other techniques, such as finite difference gradient and reward weighted regression, achieving equal or lower costs with a smaller number of samples, sometimes by orders of magnitude [Kober, 2012]. CrKR has been used frequently to optimize the meta-parameters of DMPs, but it can be used with any movement representation that can be manipulated with a set of parameters. Nevertheless, the high computational complexity of the algorithm limits its application for tasks where a high number of samples is required to reach satisfactory results: each iteration takes increasingly longer and eventually the processing power and memory required to keep the algorithm running become prohibitive due to the required multiplications and inversions of matrices with sizes that grow with every iteration. In this chapter, a technique to deal with this issue is proposed: it is based on the idea that we can split the execution of the algorithm in several runs

with a smaller number of rollouts, while transferring the learned function between successive runs. Such modification allows circumventing the original algorithm limitations, introducing the possibility to use it with a high number of training samples without having to deal with an indefinitely increasing execution time per iteration. This is an important contribution, since in many cases it is the only way for the algorithm to develop solutions in a satisfactory level.

Besides the suggested approach to circumvent the high computational complexity of the CrKR, other modifications are presented. These improve on the efficacy of the algorithm, further increasing its usefulness.

3.1.1 Outline

An adapted transcription of CrKR follows:

Algorithm 1 CrKR algorithm

1: **Preparation steps:**
2: Determine initial state s_0, meta-parameters γ_0, and cost c_0
3: Initialize the corresponding matrices S, Γ, C
4: Choose a kernel k, K
5: Set a scaling parameter λ
6: **for all** iterations j **do**
7: Determine the state s_j specifying the situation
8: Calculate the meta-parameters γ_j by:
9: $\bar{\gamma}(s_j) = k(s_j)^T(K + \lambda C)^{-1}\Gamma$
10: Determine the variance $\sigma^2(s_j) = k(s_j, s_j) - k(s_j)^T(K + \lambda C)^{-1}k(s_j)$
11: Draw the meta-parameters from a Gaussian distribution $\gamma_j \sim \mathcal{N}(\gamma|\bar{\gamma}(s_j), \sigma^2(s_j)I)$
12: Execute the policy using the new meta-parameters
13: Calculate the cost c_j at the end of the episode
14: Update S, Γ, C accordingly
15: **end for**

CrKR performs many rollouts throughout its execution, where many sets of parameter values are tested for different states. Based on the output of the cost function resulting from these rollouts, it starts to understand which parameters result best for each state and shapes the learned state to parameters mapping to reflect this knowledge. More specifically, at each iteration j (lines **6-15** in algorithm 1), CrKR performs a trial with a new parameters vector γ_j (lines **11,12**) for the given state of the system which results in a certain cost c_j (line **13**). This cost is used to adjust the policy $\bar{\gamma}(s)$ and variance $\sigma^2(s)$ functions (line **14**) [1]: after a trial with a set of parameters θ_i when facing a state s_i, the policy will shift toward θ_i around s_i and the variance will decrease around s_i. The strength of these displacements is inversely proportional to the achieved cost and λ. The reasoning for the algorithm to use such an update rule is simple:

[1] $\bar{\gamma}(s)$ and $\sigma^2(s)$ can also be interpreted together as a stochastic policy, i.e, a policy with an inherent exploration component.

when some parameters achieve a low cost in a given state, it is likely that in the neighboring states a policy would perform similarly when using the same parameters. Besides, since a low cost result is being achieved for the state in cause, the exploration around it can be reduced, as this is a sign that the algorithm is finding better values for the policy in s_i. The λ parameter (lines **9,10**) regulates the $\bar{\gamma}(s)$ and $\sigma^2(s)$ "elasticity"; when using a higher λ, more samples are required to provoke significant changes in the policy and variance, but usually the algorithm converges to a better solution then with a lower λ, even though it takes longer. This is because if a larger quantity of data is required to affect the policy, more information is available to do so in a more precise and optimal way.

In short, the following observations can be made about the operation of the algorithm:

- The final policy, i.e, $\bar{\gamma}$ will be influenced by all trials performed.
- The influence in $\bar{\gamma}(s_j)$ of the parameter values used in a trial is proportional to the similarity between s_j and the state at which the trial was performed.
- The influence, in the final policy, of the parameter values used in a trial is inversely proportional to the cost that resulted from the trial.
- The lower the costs of the trials performed around a state s_i, the lower will be the variance around that state.
- The greater the number of the trials performed around a state s_i, the lower will be the variance around that state.
- $\sigma^2(s) \in]0, k(s,s)]$
- λ is a constant parameter that regulates how easily the $\bar{\gamma}(s)$ and $\sigma^2(s)$ functions are influenced by samples

The chosen kernel function quantifies the similarity between two states .Normally the output of a kernel between two states ranges from 0 to 1, where 1 corresponds to maximum similarity. When dealing with vectors of real numbers, a simple kernel function $k(s_i, s_j)$ can be, for instance $e^{-||s_i - s_j||^2}$. The kernel function is important because it is what defines how much a result obtained from a trial in a certain state is generalizable to other states. Also, the kernel function influences the learning convergence of the policy in hands and the complexity it can achieve. For instance, a Gaussian kernel with a high standard deviation will cause distant states to be considered more similar than a Gaussian kernel with a lower standard deviation. As a consequence, using a Gaussian kernel with a high standard deviation will cause the knowledge of every state to be considered more generalizable to other states. This would lead to a faster convergence because less data around each state would be required to predict the parameters to use for it, but at the cost of lower precision and policy quality.

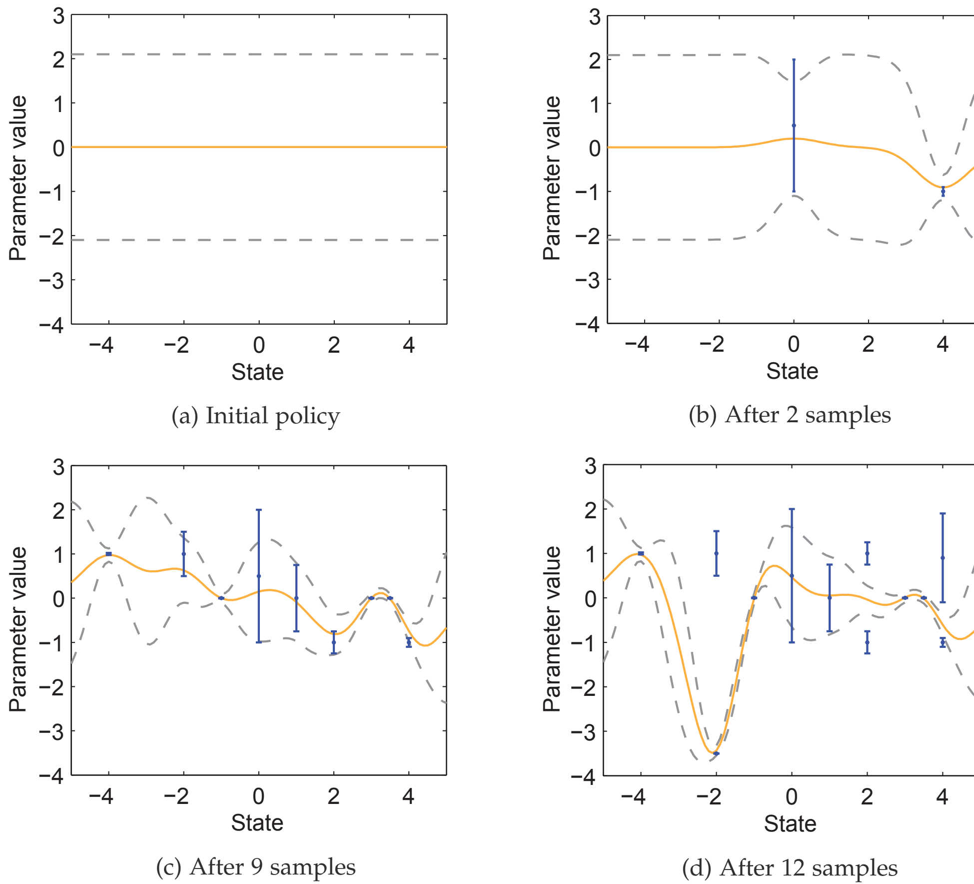


Figure 3.1: Evolution of a policy with CrKR after 0 (a), 2 (b), 9 (c) and 12 (d) samples. The dashed lines show the variance of the policy. The vertical bars represent trials: the location of each bar marks the attempted parameters and the state of the system at the time the corresponding trial was performed. The size of each bar is proportional to the cost resulting from the corresponding trial. In (b), it is clearly visible that the policy shift towards each set of parameters is inversely proportional to the cost that results from it. In the last graphic d, it can be observed that the variance is higher for those states that have not been faced or where the costs of the performed trials were higher. Image adapted from [Kober et al., 2012].

.

At each iteration, the constructed policy results in a summation of basis functions created from the samples. Each one contributes to incrementally refine the policy.

CrKR can be used as a passive learning method, where the state variables' values that the agent faces are the ones naturally occurring in the environment during its regular activity or as an active learning method, where specific tasks with artificially manipulated state variable values are used to train the agent.

3.1.2 Computational Complexity

The main bottleneck of algorithm 1 is in the matrix inversion performed in each iteration which is required to compute the parameters and variance:

$$(K + \lambda C)^{-1} \tag{3.3}$$

As $K + \lambda C$ is a square matrix of order n, where n is the number of training samples (or past iterations), the computational complexity of evaluating the expression (3.3) is:

$$O(t_{\text{inv}}(n))$$

where $t_{\text{inv}}(m)$ is the computational complexity of inverting a $m \times m$ matrix. The most part of algorithms used in practical applications to compute the inverse of a matrix operate in cubic time [Williams, 2011, Robinson, 2005]. Thus, it is reasonable to assume that evaluating the expression above takes $O(n^3)$ time. In addition, the memory used to store the matrices is $O(n^2)$.

3.2 Performance and effectiveness

3.2.1 Circumventing computational complexity limitations

In reinforcement learning, it is usually the testing of the solutions that takes more time. However, the performance of CrKR degrades considerably with each iteration, becoming a real concern if one intends to learn complex policies that demand a high number of trials to reach acceptable levels.

CrKR finds a policy of the form:

$$\pi(\gamma|s) = \mathcal{N}(\gamma|\bar{\gamma}(s), \sigma^2(s)I)$$

As mentioned in 3.1.2, the main problem is that each iteration of the algorithm uses incrementally more computational power, therefore creating a practical limitation on the number of samples it is possible to learn from before the algorithm starts to take unbearable amounts of time to run.

In order to go around this limitation, the execution *splits* are herein introduced; if we stop the execution of the algorithm at some iteration m, we are left with a $\bar{\gamma}(s)$ function that possibly needs more refinement. This refinement can be obtained by optimizing a new $\Delta\bar{\gamma}(s)$ function

that is to be added to the previous function, constituting the final state to parameters mapping:

$$\pi(\gamma|s) = \mathcal{N}(\gamma|\bar{\gamma}(s) + \Delta\bar{\gamma}(s), \sigma^2(s)I)$$

Splitting the execution of the algorithm in p rounds following the same pattern would yield a policy:

$$\pi(\gamma|s) = \mathcal{N}(\gamma|\bar{\gamma}(s) + \Delta\bar{\gamma}_1(s) + ... + \Delta\bar{\gamma}_p(s), \sigma^2(s)I)$$

Changing the denomination of the first policy developed before any split from $\gamma(s)$ to $\Delta\gamma_0(s)$ allows one to rewrite the same expression using a more uniform notation:

$$\pi(\gamma|s) = \mathcal{N}(\gamma|\Delta\bar{\gamma}_0(s) + \Delta\bar{\gamma}_1(s) + ... + \Delta\bar{\gamma}_p(s), \sigma^2(s)I)$$

In each round, the learned deterministic policy and all its corrections become the baseline policy in the next round, on which further improvement is attempted. This process that can be repeated as many times as required.

Calculating the deterministic component In algorithm 1, the deterministic component of the policy is calculated as:

$$\bar{\gamma}(s) = k(s)^T(K + \lambda C)^{-1}\Gamma$$

From now on, for expressiveness reasons, the vector resulting from the kernel between a state s and a matrix of states S (S being a matrix where each row is a different state) will be written as $k(s, S)$, i.e:

$$S = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \end{pmatrix}$$

$$k(s, S) = \begin{pmatrix} k(s, s_1) \\ k(s, s_2) \\ \vdots \\ k(s, s_m) \end{pmatrix}$$

where $k(s^i, s^j)$ is the kernel between states s^i and s^j. Also, the matrix containing the kernel between every pairwise state combination of states in a matrix S where each row is a different

state is written as:

$$K(S) = \begin{pmatrix} k(s_1, s_1) & k(s_1, s_2) & \cdots & k(s_1, s_m) \\ k(s_2, s_1) & k(s_2, s_2) & \cdots & k(s_2, s_m) \\ \vdots & \vdots & \ddots & \vdots \\ k(s_m, s_1) & k(s_m, s_2) & \cdots & k(s_m, s_m) \end{pmatrix}$$

Therefore, we now express the same line of algorithm 1 as:

$$\bar{\gamma}(s) = k(s, S)^T (K(S) + \lambda C)^{-1} \Gamma$$

So if we divide the algorithm in p rounds, where each round optimizes a policy that is to be added to the policies before, we have:

$$\begin{aligned} \bar{\gamma}(s) &= \Delta\gamma_0(s) + \Delta\gamma_1(s) + ... + \Delta\gamma_p(s) \\ \bar{\gamma}(s) &= k(s, S_0)^T (K(S_0) + \lambda C_0)^{-1} \Gamma_0 \\ &\quad + k(s, S_1)^T (K(S_1) + \lambda C_1)^{-1} \Gamma_1 \\ &\quad + \cdots \\ &\quad + k(s, S_p)^T (K(S_p) + \lambda C_p)^{-1} \Gamma_p \\ \bar{\gamma}(s) &= k(s, S_0)^T A_0 + k(s, S_1)^T A_1 + \cdots + k(s, S_p)^T A_p \end{aligned}$$

where:

$$A_i = (K(S_i) + \lambda C_i)^{-1} \Gamma_i$$

As it can be seen, instead of having to calculate the inverse of a large matrix, we now need only to calculate several inversions of smaller matrices. The benefit of this is even greater when we consider that after an pth split, only $k(s, S)$ and A_p need to be calculated as the matrices

from A_0 to A_{p-1} do not change. By using the following replacements:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{pmatrix}$$

$$T = \begin{pmatrix} S_0 \\ S_1 \\ \vdots \\ S_{p-1} \end{pmatrix}$$

We can simply write:

$$\bar{\gamma}(s) = k(s, T)^T A$$

Since $k(s, T)^T \in \mathbb{R}^{1 \times n}$ and $A \in \mathbb{R}^{n \times g}$, the computational complexity of calculating $\gamma(s)$ is reduced to $O(r^3 + ng)$ where r is the number of samples per round, n is the total number of past samples and g is the number of parameters. This is considerably better than $O(n^3)$, because r and g are fixed throughout the execution of the algorithm.

After a split p, A is updated as:

$$A := \begin{pmatrix} A \\ A_{p-1} \end{pmatrix}$$

Calculating the stochastic component The stochastic component of the policy is calculated as:

$$\sigma^2(s) = k(s, s) - k(s, S)^T (K + \lambda C)^{-1} k(s, S)$$

If we use the same technique we have:

$$
\begin{aligned}
\sigma^2(s) &= \Delta\sigma_0^2(s) + \Delta\sigma_1^2(s) + ... + \Delta\sigma_p^2(s) \\
\sigma^2(s) &= pk(s,s) \\
&\quad - k(s,S_0)^T(K(S_0) + \lambda C_0)^{-1}k(s,S_0) \\
&\quad - k(s,S_1)^T(K(S_1) + \lambda C_1)^{-1}k(s,S_1) \\
&\quad - \dots \\
&\quad - k(s,S_p)^T(K(S_p) + \lambda C_p)^{-1}k(s,S_p) \\
\sigma^2(s) &= pk(s,s) - k(s,S_0)^T B_0 k(s,S_0) - k(s,S_1)^T B_1 k(s,S_1) - \cdots - k(s,S_p)^T B_p k(s,S_p)
\end{aligned}
$$

where:

$$B_i = (K(S_i) + \lambda C_i)^{-1}$$

However each B_i matrix would have p lines and columns. B_i matrices of past rounds would not need to be recalculated, but since $k(s, S_i)$ needs to be calculated every iteration and for any i, the same would happen for the multiplication of B_i with $k(s, S_i)$ or $k(s, S_i)$ with B_i. The computational complexity of calculating $\sigma^2(s)$ this way would be $O(r^3 + nr)$ and the memory needed to keep all B_i matrices would be $O(nr)$ which would still be very prohibitive when attempting to run the algorithm with a high number of samples.

The variance function guides the algorithm, defining the exploration level for each state. It is constructed from the costs of the many executed trials and ideally its shape would match the one of a cost function $C(\gamma_i, s)$ that would evaluate a policy γ_i for each state s. As has been previously said:

- The lower the costs of the trials performed around a state s, the lower will be the variance around that state.
- The greater the number of the trials performed around a state s, the lower will be the variance around that state.

In the evolution of the variance function throughout the execution of CrKR it is generally possible to observe that:

- The costs and states of past trials define the shape of the variance function
- The higher the number of past trials, the more pronounced are the valleys in the shape of the variance function

The variance function does not need to be as precise as the parameters function since, semantically, $\bar{\gamma}(s)$ has an absolute identity, while the effects of the $\sigma^2(s)$, have a relative one: the

variance function only defines, in each state s, the amplitude of the interval centered in $\bar{\gamma}(s)$ where it is more likely to pick parameter values to be tested.

By keeping only the last completed B_i and S_i matrix, respective to the iterations between the last two performed splits[2], and with an high enough r, we can have a variance shape function whose locations of peaks and valleys would approximately match those of a cost function of the achieved policy:

$$\sigma_*^2(s) = k(s,s) - k(s,S_i)^T B_i k(s,S_i)$$

Furthermore, the general effect of the numbers of samples - making the valleys of the variance function more pronounced - can be artificially simulated by applying a transformation:

$$\sigma^2(s) = \sigma_*^2(s)\exp(-a \cdot n_{\text{samples}} \cdot (k(s,s) - \sigma_*^2(s)))$$

where n_{samples} is the number of past samples and a is the aging factor that controls the strength of the transformation.

Using this heuristic, the calculation of the variance has $O(r^3)$ computational complexity and the memory needed to keep all the matrices needed is $O(r^2)$.

Using these methods for the calculation of the $\gamma(s)$ and $\sigma^2(s)$, the computational complexity of each iteration of CrKR changes from $O(n^3)$ to $O(r^3 + ng)$ and the memory needed to store the required matrices for these operations changes from $O(n^2)$ to $O(r^2)$. Since r is constant, this is an important improvement that brings the possibility of learning from a much greater number of trials, therefore allowing a better policy to be achieved.

3.2.2 Costs standardization

One of the issues that might stop CrKR from improving a policy past a certain point is the scaling of the costs that are supplied to the algorithm or choosing an inadequate value for λ . The cost of a trial rates the vector of parameters that was used in it for the state the agent was facing. However, one might ask if this rating should be dependent on the costs of past trials performed in neighbor states.

Let us consider two different policy optimization problems A and B where we are using CrKR. Suppose that at some iteration in each of the instances of the algorithm, the agents in both problems are facing the states s_A and s_B and the values of the costs for the parameters used in the 5 situations where the respective instances states were more similar to s_A and s_B

[2]Before performing the first split, B is calculated normally. After performing the first split, B is calculated using the data respective to the iterations between the last two performed splits.

are $\mathscr{C}_A$ and $\mathscr{C}_B$ where:

$$\mathscr{C}_A = \begin{pmatrix} 4.3 & 3.9 & 4.4 & 4.22 & 4.12 \end{pmatrix}$$
$$\mathscr{C}_B = \begin{pmatrix} 6.1 & 6.01 & 5.3 & 5.74 & 4.72 \end{pmatrix}$$

Suppose now that the last trial in each of the problems' instances has the same cost $c_A = c_B = 4.6$. Looking at the values, this seams like a bad result considering the first set of costs $\mathscr{C}_A$, but a good one considering the second set $\mathscr{C}_B$; c_A is higher than every element of $\mathscr{C}_A$ but lower than every element of $\mathscr{C}_B$. This raises the question on whether using the cost of a trial directly is the best way to rate the parameters used under its circumstances, or if this rating should be relative to the average of previously registered costs around those circumstances. Consider now that $c_A = c_B = 9.99$ and that:

$$\mathscr{C}_A = \begin{pmatrix} 9.3 & 10.3 & 9.7 & 10.9 & 9.8 \end{pmatrix}$$
$$\mathscr{C}_B = \begin{pmatrix} 10.001 & 9.9997 & 10.0035 & 10.004 & 9.998 \end{pmatrix}$$

Even though $\overline{\mathscr{C}_A} \approx \overline{\mathscr{C}_B} \approx 10$, c_B is a much more impressive result in its context than c_A. The values of $\mathscr{C}_A$ deviate much more from the mean than those of $\mathscr{C}_B$, therefore the parameters corresponding to c_B should probably be given more weight (lower effective cost) as they appear to be an exceptionally good result.

Considering both types of situations presented, the following cost standardization scheme is proposed:

1. Keep a pool with state-cost pairs of the o latest trials
2. When a trial performed in a state s produces a cost c, pick the m most similar states from the pool [3], and obtain its corresponding costs $\mathscr{C}$. (Fig 3.2)
3. Standardize c to $\check{c}$ where:

$$\check{c} = \exp\left(\frac{c - \overline{\mathscr{C}}}{\sigma(\mathscr{C})}\right)$$

Apart from the application of the exponential, this is a commonly used standardization method. First, the average cost of $\mathscr{C}$ is subtracted to c, which yields a negative or positive value. A negative value represents a cost better than the average, and vice versa. Then, the obtained value is divided by the standard deviation of $\mathscr{C}$, scaling it appropriately according to how much it deviates from the average. Finally, an exponential function is applied to the result, so that

[3] By the default the algorithm uses euclidean distance for calculating the most similar states, but this behavior can be overridden if needed.

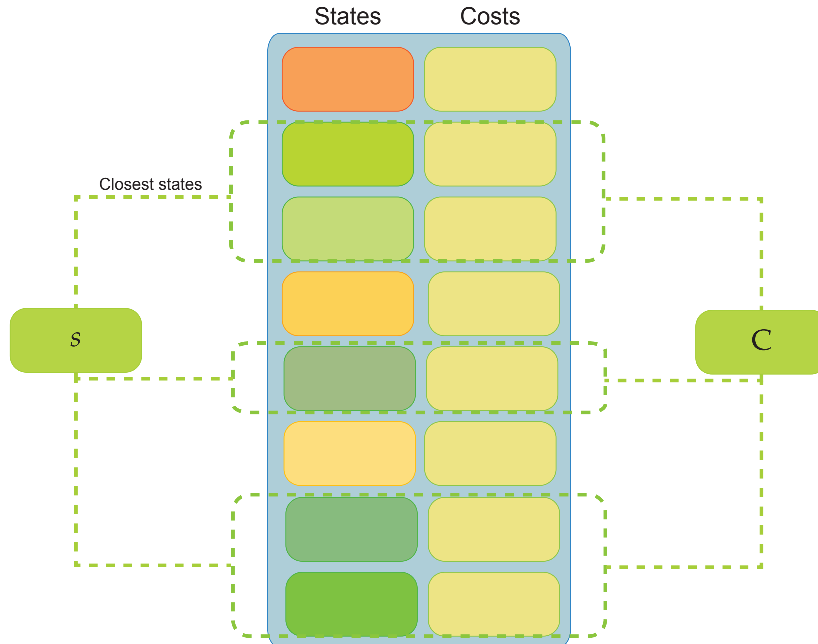


Figure 3.2: Illustration of the second step of the standardization process. The colors of the states represent how similar they are to the state of the trial whose cost is to be standardized. The costs belonging to the m trials whose circumstances are more similar to the actual trial are the ones used for calculating the standardized value of the cost.

the final standardized cost value is positive. Using this scheme, for the given examples and choosing $m = 5$, we have that $\check{c}_A = 0.98391$ and $\check{c}_B = 0.01181$. $\check{c}_B =$ is much lower than $\check{c}_A$ which goes according to the points presented previously.

There are some points that must be clarified when using this method:

- Before accumulating enough samples in the pool, the non standardized costs must be used.
- It is advised to define two different cost scaling parameters to use before and after filling the samples pool, for using with non-standardized (λ) and standardized costs ($\lambda_{\text{standardized}}$), respectively.
- It would not be appropriate to use standardized costs for the variance because, for any state, the average standardized cost will tend to 1. This would turn the variance function constant which would defeat its purpose of differentiating the exploration between states.

While this technique does not always bring better results, it makes the success of the algorithm less dependent on the ability to choose proper cost scaling parameters for each problem. After the samples pool is filled, the costs are standardized and $\lambda_{\text{standardized}}$ is used. A $\lambda_{\text{standardized}}$ that works well for a certain problem will likely perform well for other problems.

On the other hand, without costs standardization, the λ parameter needs to be completely tuned for every new problem.

3.2.3 Hints

CrKR can be fed with hints in order to give a head start to the policy improvement process. For instance, if some parameters are known to be optimal for a certain state s, they can be directly fed to CrKR with a very low cost. This will heavily shift the policy towards these parameter values around s. Lower degrees of certainty about the quality or optimality of some parameters can be compensated by using a higher cost, which allows the user to provide a clue to the algorithm without influencing its behavior in an exaggerated way.

3.2.4 Variance multiplier

Not always the variance function is going to be properly scaled to the problem in hands. To add the possibility of controlling the order of magnitude of the variance function, a multiplier parameter V can be introduced in the algorithm so that:

$$\sigma^2(s) = V\sigma_*^2(s)\exp(-a \cdot n_{\text{samples}} \cdot (k(s,s) - \sigma_*^2(s)))$$

3.2.5 Final algorithm

Herein follows the algorithm with all the proposed modifications:

Algorithm 2 CrKR^{++} - Final algorithm

Preparation steps:
Determine initial state s_0, parameters γ_0, and cost c_0
Initialize the corresponding matrices $S, \Gamma, C, T, A, B_{\text{last}}, C_{\text{absolute}}, S_{\text{last}}$
Choose a kernel k, K
Set a scaling parameter $\lambda, \lambda_{\text{standardize}}$
Set the number of samples per round r
Set a variance multiplier V
Set a variance aging factor a
Initialize sample pool P
Choose a pool size p and a number of m neighbors to consider when standardizing costs
for all iterations j **do**
Determine the state s_j specifying the situation
$\bar{\gamma}_{\text{Accumulated}}(s_j) = k(s_j)^T A$
$\Delta\bar{\gamma}(s_j) = k(s_j, S)^T (K(S) + \lambda C)^{-1} \Gamma$
if first round **then**
$\sigma_*^2(s_j) = k(s_j, s_j) - k(s_j, S)^T (K(S) + \lambda C_{\text{absolute}})^{-1} k(s_j, S)$
else
$\sigma_*^2(s_j) = k(s_j, s_j) - k(s_j, S_{\text{last}})^T \cdot B_{\text{last}} \cdot k(s_j, S_{\text{last}})$
end if
$\sigma^2(s_j) = V \sigma_*^2(s_j) \exp(-a \cdot n_{\text{samples}} \cdot (k(s_j, s_j) - \sigma_*^2(s)))$
$\gamma_j \sim \mathcal{N}(\gamma | \bar{\gamma}_{\text{Accumulated}}(s_j) + \Delta\bar{\gamma}(s_j), \sigma^2(s^j) I)$
Execute the policy using the new parameters
Calculate the cost c_j at the end of the episode
if P is filled **then**
Get the costs $\mathscr{C}$ from the m trials with most similar states to s_j from P
$\check{c}_j = \exp\left(\frac{c_j - \overline{\mathscr{C}}}{\sigma(\mathscr{C})}\right)$
$c_{j_{\text{final}}} = \frac{\lambda_{\text{standardized}} \check{c}_j}{\lambda}$
end if
Update $S, \Gamma, C, C_{\text{absolute}}$ accordingly // Update Γ with $\Delta\bar{\gamma}(s_j)$
Cache $K(S)$ to speed up next iteration
if round ends **then**
Update $T, A, B_{\text{last}}, S_{\text{last}}$ accordingly
Reset $S, \Gamma, C, C_{\text{absolute}}$, cached $K(S)$
end if
Update P adding the pair (s_j, c_j)
end for

3.3 Implementation

The CrKR^{++} algorithm was implemented in Matlab® following object-oriented programming principles in order to enhance extensibility, flexibility and modularity. The developed code is independent from the context of the problem in which it is used. It can be used not only

for movement learning, but any other reinforcement learning problem where there is a policy that must be optimized for a range of continuous states. Moreover, it was written with the possibility of being modified in the future in mind.

3.3.1 Architecture

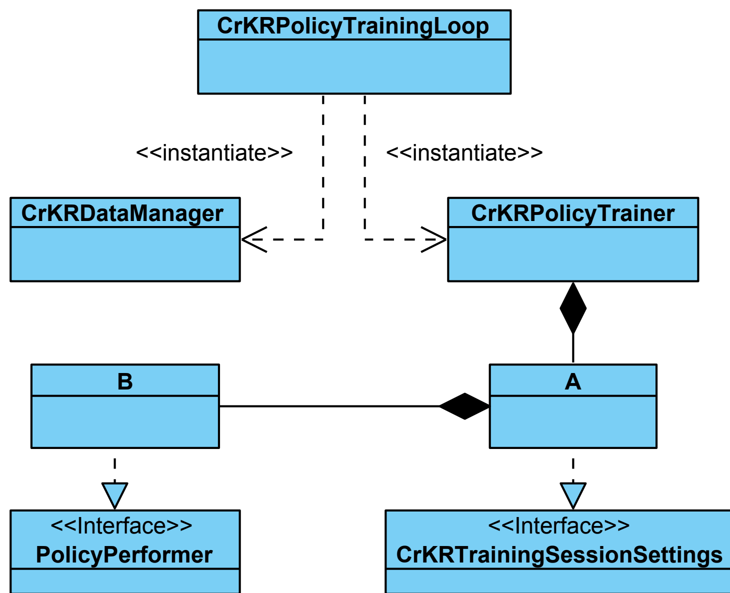


Figure 3.3: Unified Modeling Language (UML) class diagram of CrKR^{++} implementation.

The typical work-flow when using CrKR^{++} translates into the creation by the user of a *CrKR-PolicyTrainingLoop* class that instantiates a *CrKRPolicyTrainer*. Typically in this class, the user chooses how to implement the loop where the methods of *CrKRPolicyTrainer* are called. Normally, each iteration of this loop must call in succession the *getStochasticMetaParameters* and *get-DeterministicAccumulatedMetaParameters* methods from the instantiated *CrKRPolicyTrainer* and then the *calculateCost* method from the class that implements the *PolicyPerformer* interface. Finally, the method *train* from *CrKRPolicyTrainer* must be called with the state, parameters and cost as arguments. All in all, these steps should be similar to the following extract of code:

```
metaParametersCurrentPolicy = policyTrainer.getStochasticMetaParameters(actualState);
metaParametersAccumulatedPolicy =   ...
    policyTrainer.getDeterministicAccumulatedMetaParameters(actualState);
incurredCost = policyPerformer.calculateCost(metaParametersCurrentPolicy + ...
    metaParametersAccumulatedPolicy, actualState);
```

```
policyTrainer.train(actualState, metaParametersCurrentPolicy, incurredCost);
```

Frequently, one wants to periodically test the achieved policy or include specific changes in the training loop adapted to the problem in hands. Even though an example of a *CrKRPolicy-TrainingLoop* class is provided to the user, it is advised to change it according to the needs.

The following list provides a description on each element of the diagram presented in Fig. 3.3:

- **CrKRPolicyTrainingLoop** - The class that uses *CrKRPolicyTrainer* to train the policy. As the name says it is where the training loop is located.
- **CrKRPolicyTrainer** - This is the core class. It is where the algorithm matrices are, together with all the training and parameter calculation methods. In order to create an instance of this class, an instance of a class that implements *CrKRTrainingSessionSettings* must be supplied (**A** in Fig.3.3).
- **CrKRTrainingSessionSettings** - An interface that specifies all methods and properties necessary to define the settings class and its instance to supply to a *CrKRPolicyTrainer* (**A** in Fig.3.3).
- **PolicyPerformer** - An interface that specifies the methods that a policy performer class (**B** in Fig.3.3) must implement, namely: *init*, *calculateCost* and *finnish*. Of these, *calculateCost* is the most important, since it is the method that returns the cost in function of the state and parameters.
- **CrKRDataManager** - A class that contains utility code such as the methods to save the algorithm progress (matrices) and to express graphically the achieved policy after running it.

When one wants to use CrKR^{++} with a different problem, only the class that implements *CrKRTrainingSessionSettings* needs to be changed (**A** in Fig.3.3) with new parameters and an instance of a policy performer adjusted to the new context (**B** in Fig.3.3).

3.3.2 Parameters guide

Using CrKR^{++} for optimizing a policy requires choosing the values for a set of parameters specified in the class (**A** in Fig.3.3) that implements the *CrKRTrainingSessionSettings* interface. Some of these parameters are related with the learning process while others are related with testing and effectiveness measurement. It is important to know what each parameter does in order to make the appropriate changes if the results of a training session are not satisfactory. The following list provides information about the parameters that have a role in the learning process.

- **numberMetaParameters** - The number of parameters of the state to parameters function.
- **numberStateVariables** - The number of state variables of the state to parameters function.
- **maximumCost** - Maximum cost admitted for a trial to be considered in the learning process.
- **maximumCostStandardized** - Maximum cost admitted for a trial to be considered in the learning process when dealing with a standardized cost.
- **lambda** - The λ parameter explained in section 3.1.1. This is a very important parameter that ranges from 0 to ∞ and it can dictate the success of a training session. Setting this parameter too high will prevent the policy from changing. Setting it too low makes the policy very volatile. The right compromise must be found. It is important to have into account that the order of magnitude of this parameter must be chosen with attention to the order of magnitude of the costs returned by the *calculateCost* method of the provided policy performer.
- **lambdaForStandardizedCosts** - The $\lambda_{\text{standardized}}$ parameter explained in section 3.2.2. Its function is the same as the *lambda* parameter. The difference is that while *lambda* is used before the costs standardization pool is filled, this parameter is used after. It is easier to set as it is not dependent on the scale of the *calculateCost* method of the provided policy performer.
- **varianceMultiplier** - The V parameter explained in section 3.2.4.
- **minimumVariance** - A minimum variance value that prevents variance from ever reaching 0, which would stop the algorithm from continuing to improve the policy.
- **kernelStdDev** - A Gaussian kernel defined as $k(s_i, s_j) = M \cdot \exp\left(\frac{-||s_i - s_j||^2}{2\sigma^2}\right)$ is used by default in the implementation of CrKR^{++}. This sets the value of σ parameter.
- **kernelMultiplier** - This sets the value of the M parameter.
- **agingFactor** - The a parameter explained in section 3.2.1. Care must be taken when choosing this value. Thinking in terms of the intended half-life of the variance as the number of samples grows might help to set its value.
- **numberSamplesBefore** - This parameter must only be set with a value different from 0 if the progress of a previous training session is being loaded and resumed. It corresponds to the total number of trials the algorithm has performed in all past training sessions.
- **A** - Initial A matrix. This parameter must only be set a non-empty matrix if a training session is being resumed.

- **T** - Initial T matrix. This parameter must only be set a non-empty matrix if a training session is being resumed.
- **InvKpClast** - Initial B_{last} matrix. This parameter must only be set a non-empty matrix if a training session is being resumed.
- **Slast** - Initial S_{last} matrix. This parameter must only beset a non-empty matrix if a training session is being resumed.
- **givenExamples** - This is where the hints, explained in section 3.2.3 are specified. It is a cell array containing three matrices with same number of rows. Each row in all matrices corresponds to a hint. The first matrix contains the state, the second matrix the parameters and the third matrix the artificial cost of the hint which controls its strength.
- **standardizePool** - Initial standardization pool. This parameter must only be a non-empty matrix if a training session is being resumed.
- **standardizePoolNumberCurrentTryout** - The number of samples that have passed in the standardization pool plus 1. This parameter must only different from 1 if a training session is being resumed.
- **numberTryoutsConsideredWhenStandardizing** - The m parameter explained in section 3.2.2.
- **numberTryoutsConsideredWhenStandardizing** - The p parameter explained in section 3.2.2.
- **maxAdmissibleDistanceToUseStandardizePool** - In order for the standardization steps to be applicable, every state in the set of closest states must not be farther than this value in terms of euclidean distance.
- **policyPerformer** - An instance of a class that implements the *PolicyPerformer* interface.
- **rolloutsPerRun** - The number of samples per round .
- **minStateValues** - The lower vertex coordinate of the hyperrectangle that defines the states space from which states are generated (only used in active learning). It is an array with as many elements as the number of dimensions of the state space.
- **maxStateValues** - The upper vertex coordinate of the hyperrectangle that defines the states space from which states are generated (only used in active learning).

3.4 Tests

The most important modification that CrKR^{++} brings in relation to CrKR is the improvement of computational complexity through the execution splits technique. However this approach is not mathematically equivalent to having the algorithm being executed in the normal way. The tests here presented will focus on measuring the algorithm effectiveness and efficiency before and after the inclusion of this technique, in order to provide enough data for a discussion on whether the execution splits are a proper solution for the computational complexity problem.

The costs standardization technique depends on the choice of values for parameters that do not exist in CrKR. Therefore, no tests will be executed in order to compare both algorithms in this aspect since the obtained performances would be dependent on the chosen values.

Function through artificial landscape For this test, CrKR and CrKR^{++} are used to find a function $\gamma(s) = (a, b)$ whose output is evaluated by a function $C(s, (a, b))$ that dictates the cost of the parameters a, b for the state s. The goal of the algorithm for the given problem was to improve $\gamma(s)$ such that $C(s, \gamma(s))$ is as low as possible for any $s \in [-50, 50]$. C is a function created from the interpolation of three artificial landscapes and is defined in the following way:

$$w_1(s) = \begin{cases} 1 - \frac{s+50}{50} & s \geq -50 \land s \leq 0 \\ 0 & s > 0 \land s \leq 50 \end{cases}$$

$$w_2(s) = \begin{cases} \frac{s+50}{50} & s \geq -50 \land s \leq 0 \\ \frac{50-s}{50} & s > 0 \land s \leq 50 \end{cases}$$

$$w_3(s) = \begin{cases} 0 & s \geq -50 \land s \leq 0 \\ 1 - \frac{50-s}{50} & s > 0 \land s \leq 50 \end{cases}$$

$$\begin{aligned} C(s, (a, b)) = & \; w_1(s) \cdot 1000|l_1(a, b)| \\ & + w_2(s) \cdot 10|l_2(a, b)| \\ & + w_3(s) \cdot 1000|l_3(a, b)| \end{aligned}$$

where $w_i(s)$ are the weights given to the l_1, l_2, and l_3 functions, according to the state s. l_1, l_2 and l_3 are respectively Beale's, Goldstein–Price and Booth's functions (Fig. 3.4), commonly used to test optimization algorithms. The option to use artificial landscape functions was

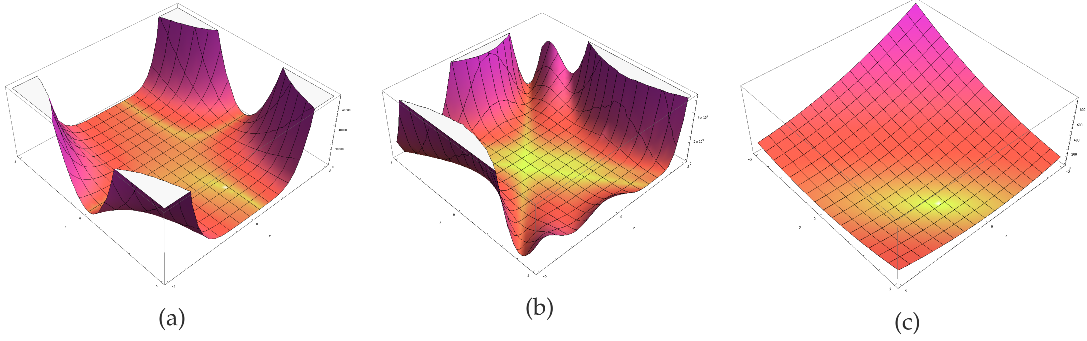


Figure 3.4: Beale's (a), Goldstein–Price (b) and Booth's (c) functions, integrated in the cost function used in the tests. These functions minima are located in (3,0.5) , (0,-1) and (1,3), respectively.

taken because these are fast to evaluate, and are therefore convenient to be used for a large number of tests. The chosen artificial landscapes for the tests exhibit some complexity, while not being too rugged, which could prevent the algorithm from improving the policy.

Triangular wave approximation In this test, CrKR and CrKR^{++} are used to approximate a triangular wave function (Fig. 3.5) f given by:

$$f(x) = \arcsin(\sin(x))$$

The cost C for each state s is given by the squared difference between f and the learned function γ:

$$C(s, \gamma(s)) = (f(s) - \gamma(s))^2$$

Bivariate function approximation In this test, CrKR and CrKR^{++} are used to approximate the bivariate function f given by:

$$f(x, y) = \exp\left(-\sqrt{x^2 + y^2}\right)$$

The cost C for each state $s = (s_x, s_y)$ is given by the squared difference between f and the learned function γ:

$$C(s, \gamma(s)) = \left(f(s_x, x_y) - \gamma(s)\right)^2$$

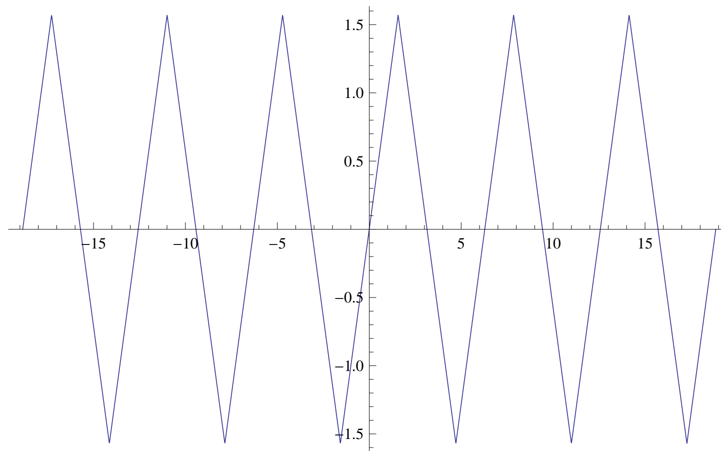


Figure 3.5: Triangular wave function

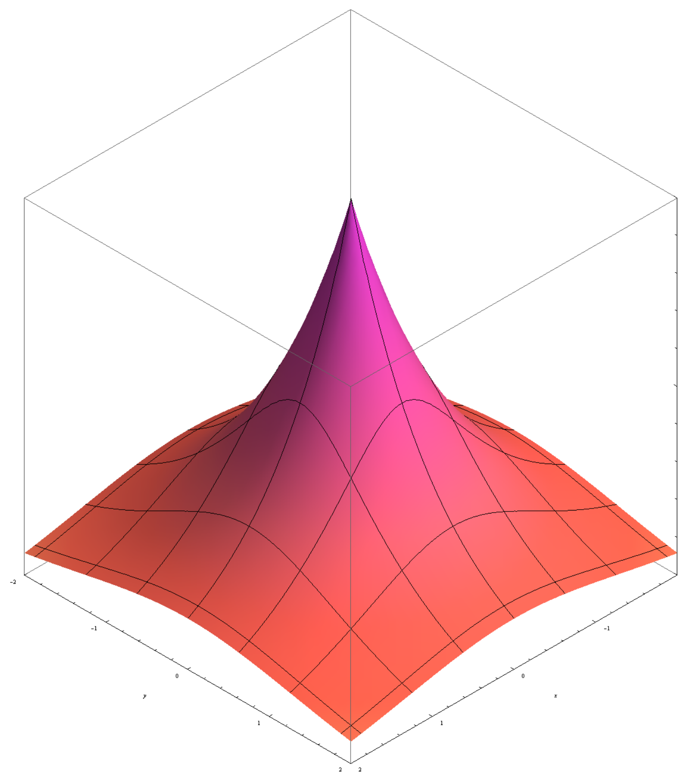


Figure 3.6: Bivariate function $f(x,y) = \exp\left(-\sqrt{x^2+y^2}\right)$

3.4.1 Results

Each test was repeated several times in order to verify if the effectiveness of the algorithm suffered any impact when introducing splits. In all test problems, the algorithm was executed for 900 rollouts for four different settings: with no execution splitting and with splits every 450, 300 and 150 rollouts (Fig. 3.7 ,Fig. 3.8 and Fig. 3.9). The $\gamma(s)$ function, set initially as $\gamma(s) = 0$ was evaluated every 60 rollouts by averaging the costs $C(s, \gamma(s))$ for various uniformly distributed state values that were kept the same for the whole experiment.

Also, to show the impact of the suggested technique in terms of performance, measurements of the execution times of the algorithm for the first test problem were taken for two settings: with no execution splitting and with splits every 300 rollouts (Fig.3.10), with each setting being tested 8 times.

Finally, in Fig. 3.11 , Fig. 3.12 and Fig. 3.13 the outcome of running CrKR^{++} algorithm for 12000 rollouts with splits every 300 rollouts is presented in order to better elucidate the reader on what the algorithm can effectively achieve.

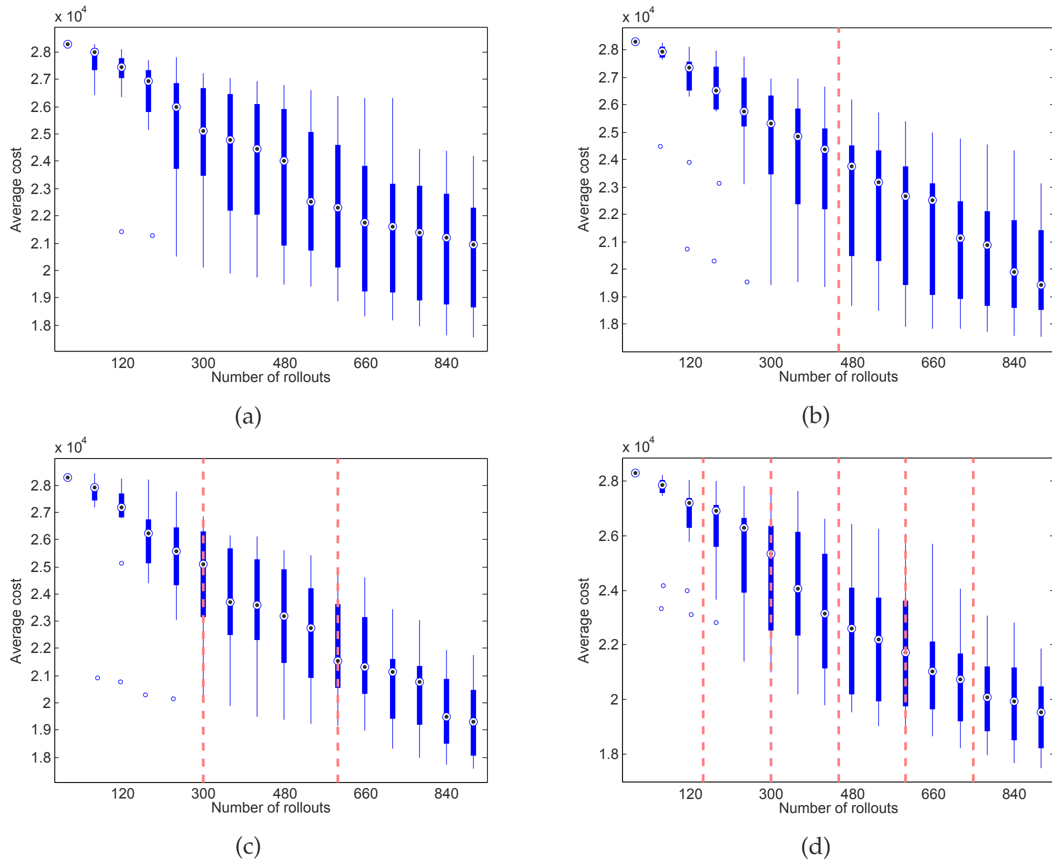


Figure 3.7: Function through artificial landscape: Results with no execution splitting (a) and with splits every 450 (b), 300 (c) and 150 (d) rollouts. The graphics show the average cost across states of the achieved $\gamma(s)$ function throughout the execution of the algorithm. Each setting was tested 12 times. The central dot in each box marks the median, and the edges the 25th (q_1) and 75th (q_3) percentiles. The whiskers extend to the data points in the interval $[q_1 - 1.5(q_3 - q_1), q_3 + 1.5(q_3 - q_1)]$, and the remaining points, considered outliers, are plotted individually.

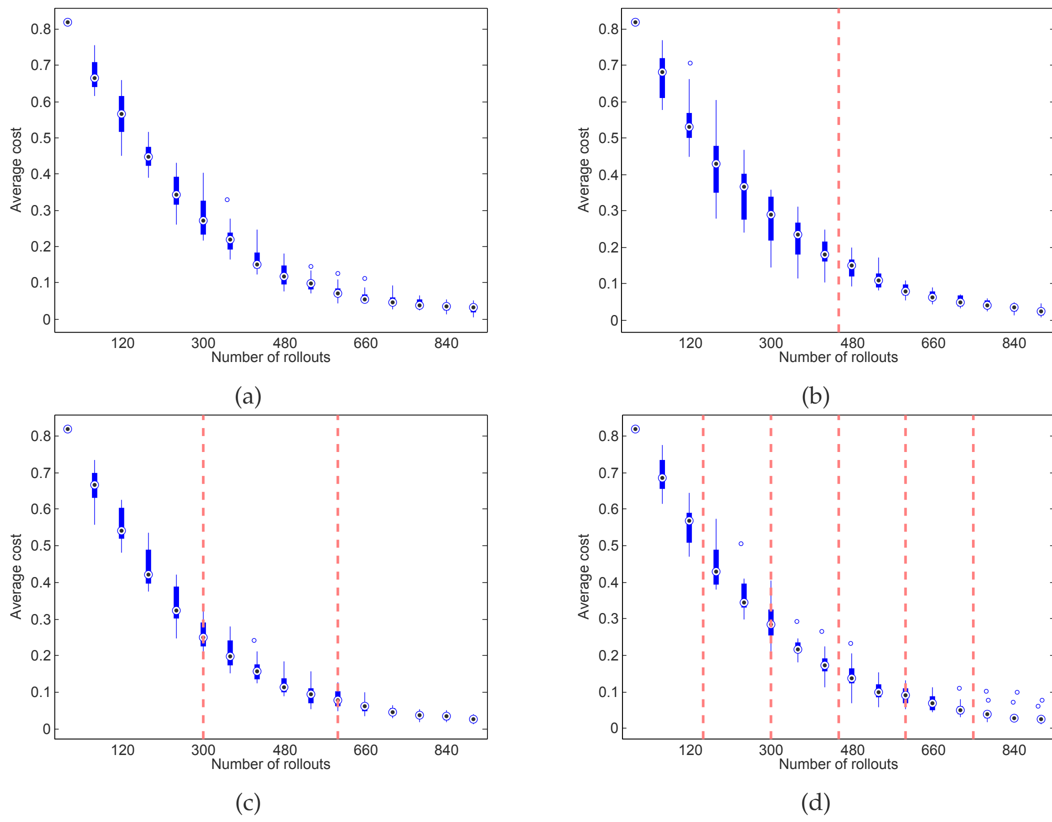


Figure 3.8: Triangular wave approximation: Results with no execution splitting (a) and with splits every 450 (b), 300 (c) and 150 (d) rollouts. The indications for reading the graphics are the same as in Fig. 3.7.

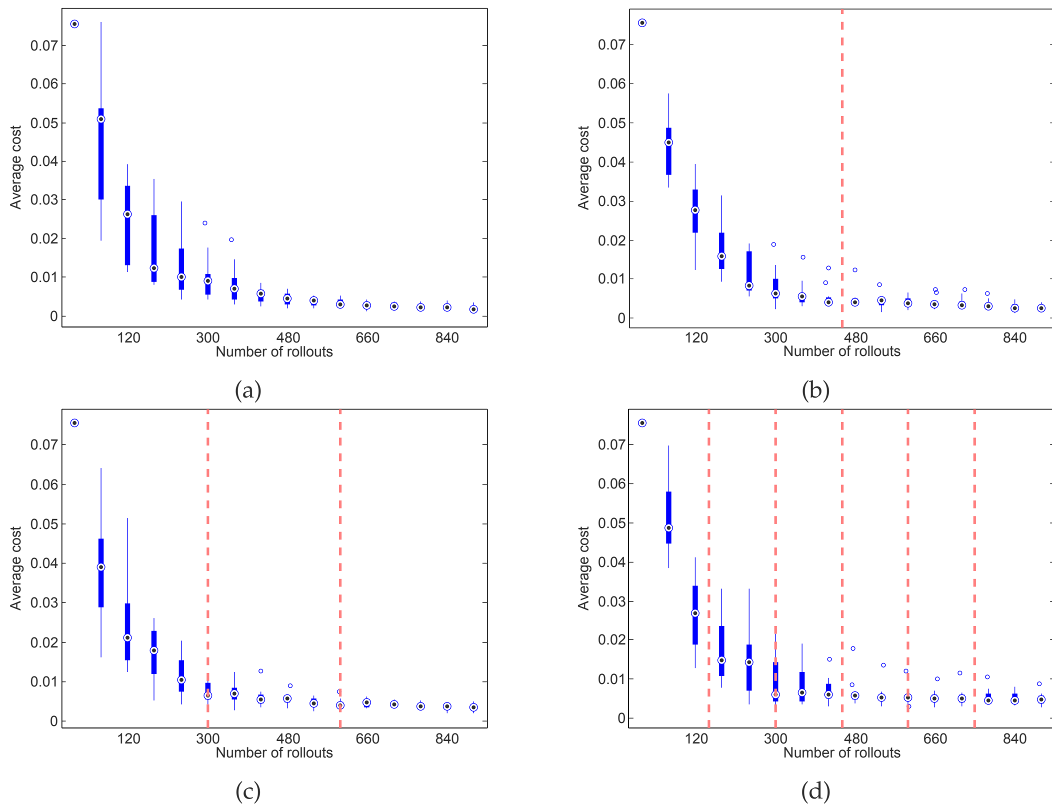


Figure 3.9: Bivariate function approximation: Results with no execution splitting (a) and with splits every 450 (b), 300 (c) and 150 (d) rollouts. The indications for reading the graphics are the same as in Fig. 3.7.

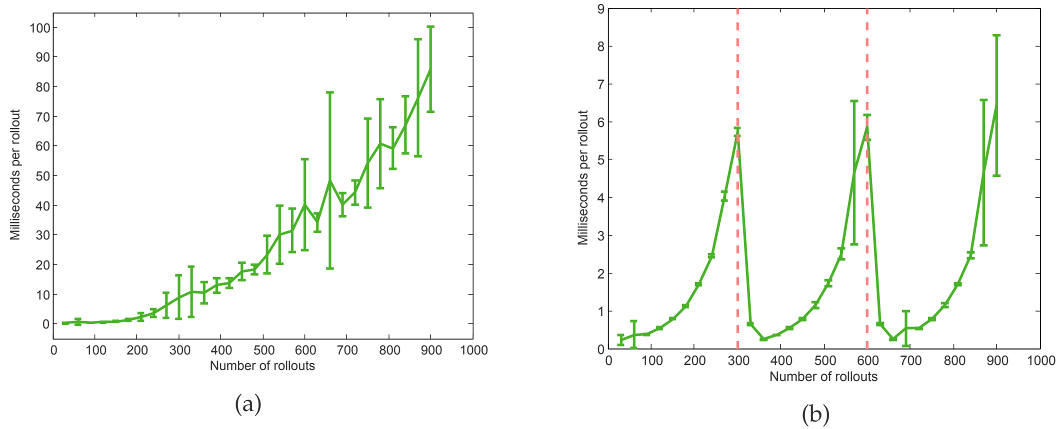


Figure 3.10: Function through artificial landscape - Results with no execution splitting (a) and with splits every 300 (b) rollouts. Each graphic shows the number of milliseconds per iteration throughout the execution of the algorithm in the one of the test problems. Each setting was tested 8 times. The values displayed are the resulting averages. The error bars illustrate the standard deviation for each measurement.

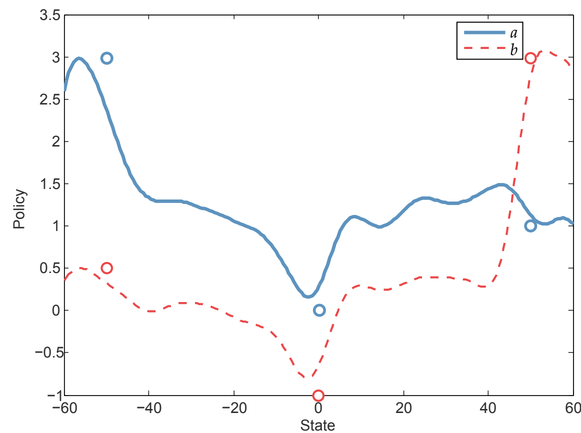


Figure 3.11: Function through artificial landscape: outcome policy, running the algorithm with splits every 300 rollouts throughout a total of 12000 rollouts. The circular markers indicate the optimal values for both parameters at states -50, 0 and 50.

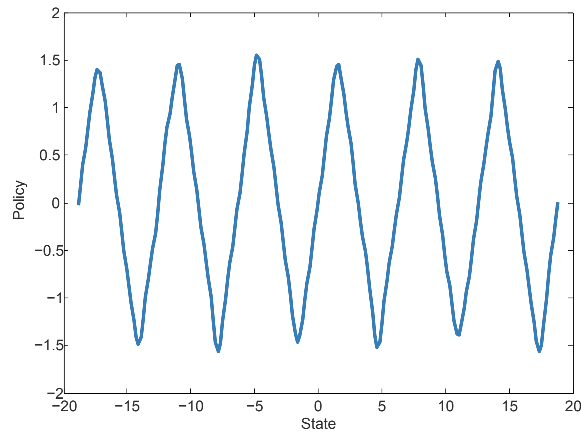


Figure 3.12: Triangular wave approximation: outcome policy, running the algorithm with splits every 300 rollouts throughout a total of 12000 rollouts.

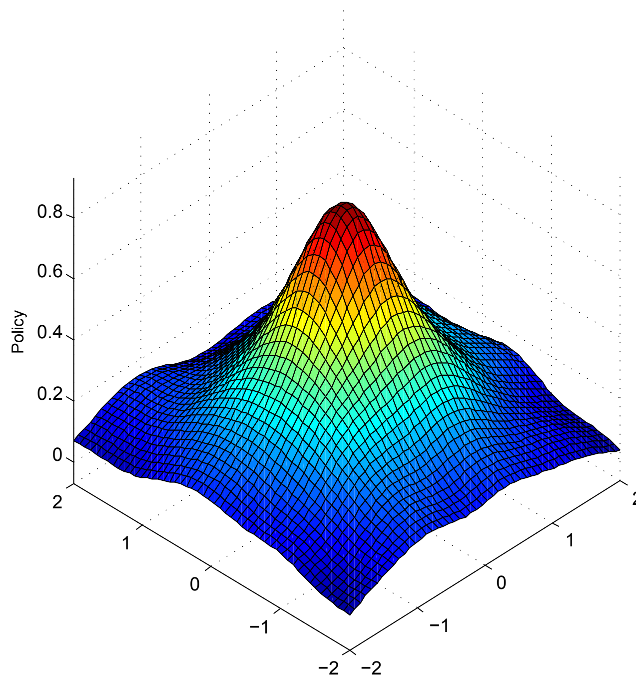


Figure 3.13: Bivariate function approximation: outcome policy, running the algorithm with splits every 300 rollouts throughout a total of 12000 rollouts.

3.4.2 Discussion

The results in Fig. 3.7, Fig. 3.8 and Fig. 3.9 reveal that the proposed execution splitting method has a very low or nonexistent apparent impact on the effectiveness of CrKR. In every setting, the final average cost of $\gamma(s)$ was within 1.92×10^4 and 2.12×10^4, for the first test problem,

2×10^{-2} and 4×10^{-2} for the second test problem or 2×10^{-3} and 5.5×10^{-3} for the third problem, which are all small intervals in the scale of the respective contexts.

The difference between the execution time before and after introducing the execution splits, expressed by the graphics in Fig. 3.10 is very noticeable: as expected, the original version of the algorithm has an execution time per iteration that grows indefinitely, ascending to more than 80 milliseconds, resulting from its $O(n^3)$ complexity (where n is the number of samples or past iterations). The modified version is much faster, never exceeding the 10 milliseconds per iteration, since the cubic growth $O(r^3)$ is limited by the number of rollouts r of each round.

Figures 3.11, 3.12 and 3.13 demonstrate the the capacity that CrKR^{++} has to achieve satisfactory policies. This is particularly observable in Figs. 3.12 and 3.13 when compared to Figs. 3.5 and 3.6, respectively, as the approximated functions greatly resemble the original ones: in Fig. 3.12, the average error between the outcome policy and the original function for the trained interval of states is ~ 0.039 and in Fig3.13 it is ~ 0.056. These are satisfactory values in the scale of the respective functions.

3.5 Summary

In this chapter, many changes were proposed to CrKR. From these, the most important was the introduction of the splits technique that allows to circumvent the limitations of CrKR related to its high computational complexity . This technique is based on the idea that the execution of the algorithm can be split in several rounds with a smaller number of rollouts, while transferring the learned state to parameters function between successive rounds. This approach was tested for different problems and settings in order to verify if the normal effectiveness of the algorithm was not affected. The obtained results suggested this is indeed the case.

The unaltered CrKR algorithm is limited to a few thousands samples before starting to take unbearable amounts of time and memory to be kept running for more iterations. The demonstrated technique allows to run CrKR for a much higher number of iterations while keeping processing and memory requirements low, therefore creating conditions to apply this algorithm to new contexts with higher levels of complexity.

Chapter 4

Flexible Framework for Learning

CrKR^{++} can be used in a learning agent or system to learn the mapping between some state variables and parameters of the system. Its performance in terms of computational complexity allows to learn complex mapping functions from long training sessions where a high number of samples is acquired. Besides the capabilities of the algorithm itself from a theoretical standpoint, a set of features were integrated in the implementation of CrKR^{++} as presented in chapter 3 such as the ability to resume and continue to work on previous progress from past training sessions and being able to integrate suggestions from the algorithm user as hints giving a head-start to the learning process. These have importance in practical contexts when applying the algorithm to real situations.

Despite all these characteristics, learning functions that map between high number of states and parameters is still hard to do timely with CrKR^{++}. In order to expand the range of situations where CrKR^{++} is applicable, F3L is herein introduced. F3L is a set of classes and methodologies that integrate CrKR^{++} and that through a set of operations allows the framework user to guide the learning process of high dimensional continuous policies. These operations rely mainly on gradual learning and task separation and can enhance the learning capabilities of the system at hands as well as increase its ease of use.

4.1 Multiple CrKR^{++} units

The common CrKR^{++} implementation allows one to optimize a mapping γ from states to parameters where:

$$\gamma : \mathbb{R}^n \to \mathbb{R}^m$$

In this context, each mapping γ learned from a CrKR^{++} instance or unit will be denominated as a primitive. The main idea in which F3L is based is that one can learn different mappings in distinct CrKR^{++} units separately that are to be combined into a full mapping.

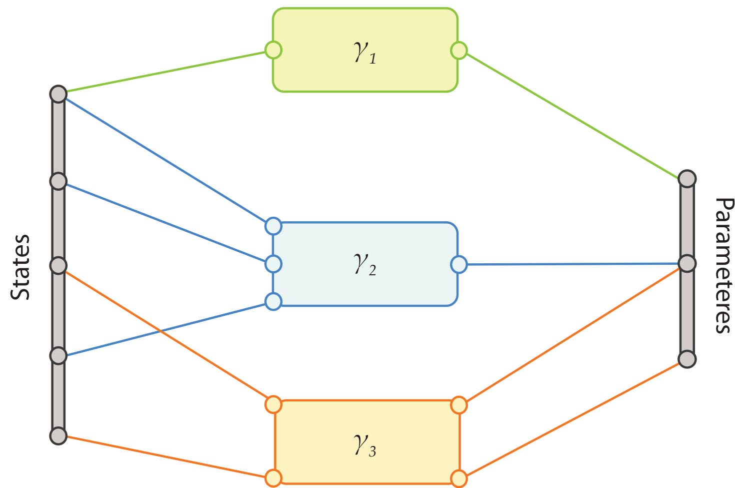


Figure 4.1: Primitive combination forming a full state to parameters mapping.

Learning each primitive in a different CrKR^{++} unit has several advantages:

- Each unit can be specified different algorithm settings, which brings extra flexibility.
- Each primitive can be learned in an exclusive task context with its own setup and reward function. This can be seen as creating special conditions to train a single aspect of a complex skill consisting in several aspects.

By specifying which state variables and parameters are considered for the mapping to be learned in each unit, a primitive combination is created as shown in Fig. 4.1. A combining function must also be defined by the framework user for primitives that output parameters in common. This will be later seen in detail in section 4.2.4.

4.2 F3L operations

F3L offers a range of possibilities that allow to structure and guide the learning process of a complex task. These possibilities translate themselves into a set of operations which are presented next.

4.2.1 Learning from zero

Simply learning a primitive with a single CrKR^{++} unit and without previous knowledge (Fig. 4.2).

Figure 4.2: Learn from zero operation. The arrow symbolizes the learning that occurs. f is the learned mapping function.

4.2.2 Learning from baseline

Further improve a previously learned primitive with a new training session (Fig. 4.3). This operation can be used in succession to allow adjusting the algorithm settings in the middle of the learning process.

Figure 4.3: Learn from baseline operation. A previously learned primitive is updated through a new training session.

4.2.3 Branch

To branch a primitive into two or more primitives (Fig. 4.4), so they can be learned in different ways by using different tasks, reward functions or algorithmic settings. Branching can be useful to divide a complex primitive into simpler primitives that can be learned one at a time.

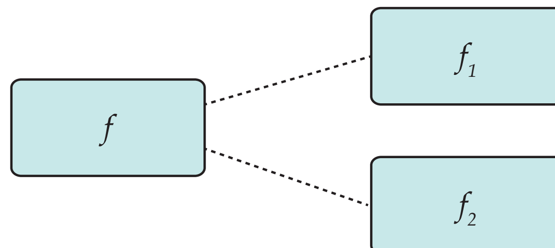


Figure 4.4: Branch operation. The dashed lines indicated that this operation is manipulative, and no learning happens in the process. f_1 and f_2 denote the primitives that result from the branching.

4.2.4 Merge

To combine two or more primitives into one (Fig. 4.5). This operation requires a combining function C that defines how the outputs of each primitive are integrated in the resultant primitive. When two primitives that share the same parameters in semantic terms are combined, it is the role of the combining function to dictate how the conflict is resolved.

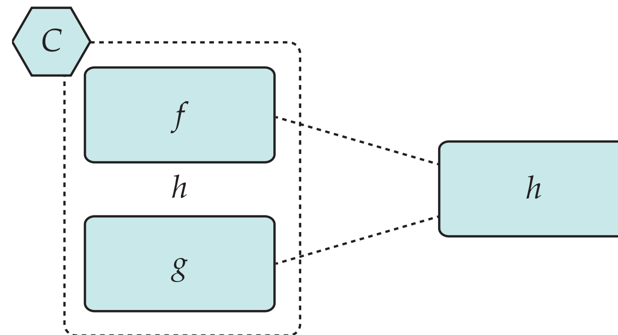


Figure 4.5: Merge operation. The dashed lines indicated that this operation is manipulative, i.e no learning takes part in it. f and g denote the primitives that are combined. h is the resulting primitive. C is the combining function: it takes as arguments the outputs of f and g and returns the output of h, i.e, $h = C(f(\square), g(\square))$.

4.2.5 Mutual learning

This consists in two or more primitives being learned simultaneously and under the same task context (Fig. 4.6). Each primitive's effect on the agent's performance might change the optimal mappings of the other primitives, so all primitives dynamically influence each other in the learning process. Ideally, as the process evolves, a synergistic relationship between the primitives is established.

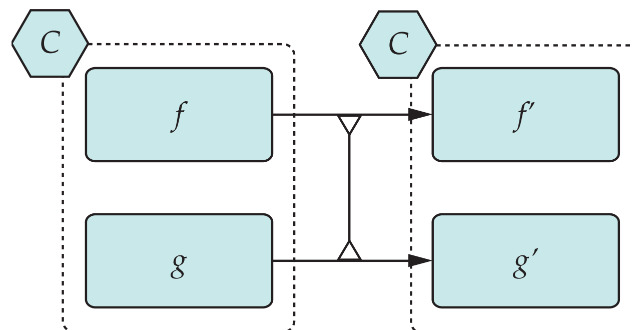


Figure 4.6: Mutual learning operation. The connection between the arrows illustrates the mutual influence that each primitive exercises upon the other. The triangles in the connection signal the primitives that suffer influence from others - in this case, both.

4.2.6 Partial learning

Happens in a training session when a group of primitives is merged with other group which is not fed with any samples and therefore does not change (Fig. 4.7). This obliges the first group of primitives to adapt to the presence of the second and improve the result of the combination of both in order to reduce the costs in the task that is being used in the learning process.

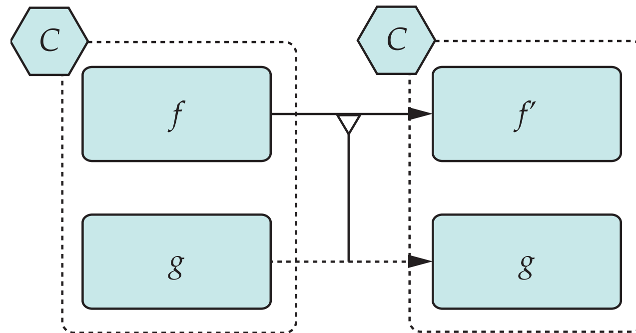


Figure 4.7: Partial learning operation. On opposition to Fig. 4.6, the connection between the arrows has only a triangle in one of the ends, indicating the only primitive that is being influenced by others. The dashed arrow illustrates that the associated primitive does not change during the training session.

4.2.7 Learn with hints

Learning a primitive when providing the respective CrKR^{++} unit hints, as explained in section 3.2.3 (Fig. 4.8).

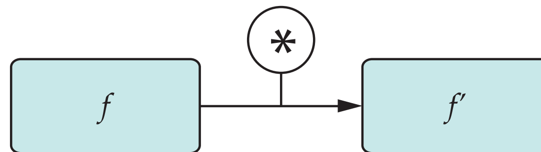


Figure 4.8: Learn with hints operation.

4.3 Implementation

F3L implementation consists in the set of Matlab® classes that implements CrKR^{++} plus a set of helper classes that allows to perform branch, merge, mutual learning and partial learning operations. They were implemented keeping the same ideals of extensibility, flexibility and modularity in mind regarding the implementation of CrKR^{++}.

4.3.1 Architecture

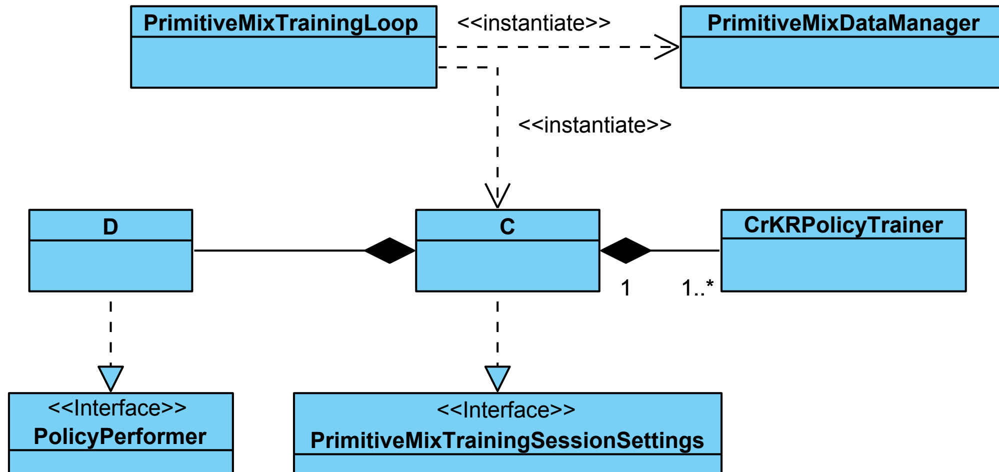


Figure 4.9: UML class diagram of F3L implementation.

The typical work-flow when using F3L translates into the creation by the user of a class (**C** in Fig.4.9) implementing *PrimitiveMixTrainingSessionSettings* to be used in *PrimitiveMixTrainingLoop*. In the constructor of this class the user instantiates the required *CrKRPolicyTrainers* or CrKR^{++} units that will train each of the primitives to be combined placing them into a cell array. The user also defines which CrKR^{++} units will be actively learning in the next training session[1] and initializes the class (**D** in Fig. 4.9) implementing the *PolicyPerformer* to be used. For example:

```
This.storedPrimitiveSettings{1} = LateralSlopedTerrainOffsetsSettings();
This.storedPrimitiveSettings{2} = LateralSlopedTerrainAmpsBalanceSettings();

This.storedLearningActive{1} = false;
This.storedLearningActive{2} = true;

for p=1:size(This.storedPrimitiveSettings,2)
    This.storedPrimitiveTrainers{p} = CrKRPolicyTrainer(This.primitiveSettings{p});
end

This.storedPolicyPerformer = SlopedTerrainLocomotionPerformer(); %Implements ...
    PolicyPerformer
```

[1]In order to perform partial learning, it is also necessary to set the value of the *varianceMultiplier* and *minimumVariance* properties to 0 in the settings class supplied to the inactive CrKR^{++} units.

Each CrKR^{++} unit maps from a subset of the total set of state variables to a subset of the total set of parameters (Fig. 4.1). In order to establish what states and parameters are respective to each CrKR^{++} unit, it is necessary to define the *parametersWrapperMapping* and *statesWrapperMapping* properties of the class. An example follows:

```
function value = get.stateWrapperMapping(This)
    value{1} = [1]; %Unit 1 maps from the 1st state variable ...
    value{2} = [1 2]; %Unit 2 maps from the 2nd state variable...
end

function value = get.parametersWrapperMapping(This)
    value{1} = [1 2 3];%...to the 1st 2nd and 3rd parameters
    value{2} = [2 4]; %...to the 2nd and 4th parameters
end
```

The framework user can modify the *PrimitiveMixTrainingLoop* class in order to add tests or other features, but must keep the code sections that implement the core of the combining process of the several CrKR^{++} units. The following list provides a description on each new element of the diagram presented in Fig. 4.9:

- **PrimitiveMixTrainingLoop** - The class where the parameters returned by the CrKR^{++} units are combined and supplied to the policy performer and where the train method is called for each CrKR^{++} unit. As the name says it is where the training loop is located.

- **PrimitiveMixTrainingSessionSettings** - An interface that specifies all methods and properties necessary to define the settings class and its instance needed to conduct the combining process happening in *PrimitiveMixTrainingLoop*.

- **PrimitiveMixDataManager** - A class that contains utility code such as the methods to save the training session progress (matrices) and to express graphically the achieved policy afterwards.

4.3.2 Parameters and methods guide

Using F3L for optimizing a policy requires choosing the values for a set of parameters and implementing a method, all of which is done in the class (**C** in Fig. 4.9) implementing the *PrimitiveMixTrainingSessionSettings* interface. Some of these are related with the learning process while others are related with testing and effectiveness measurement. The following list provides information about the parameters and methods that have a role in the learning process.

- **stateWrapperMapping** - Defines which state variables each CrKR^{++} unit maps from.

- **parametersWrapperMapping** - Defines which parameters each CrKR^{++} unit maps to.
- **policyPerformer** - An instance of a class that implements the *PolicyPerformer* interface. The *PolicyPerformer* instances initialized in each CrKR^{++} unit have no effect when using F3L.
- **primitiveTrainers** - A cell array where each cell contains a CrKR^{++} unit, i.e, an instance of a *CrKRPolicyTrainer*.
- **primitiveSettings** - A cell array containing instances that implement *CrKRTrainingSessionSettings* which are supplied in the creation of each *CrKRPolicyTrainer* that goes in *primitiveTrainers*.
- **learningActive** - A booleans cell array dictating which CrKR^{++} units will be actively learning in the next training session.
- **combinePrimitives** - The combining method that defines how the output parameters of all primitives are integrated together. It accepts as argument a matrix where each row contains the output of a different primitive, and each column is a parameter.

4.4 Summary

In this chapter a framework that augments and extends the capabilities of CrKR^{++} was presented. This framework gives the possibility of defining a learning scheme based on different operations that can be used to reduce the intractability inherent to the dimensionality of some problems. By giving the framework user the ability of structuring the learning process, a satisfactory policy can be achieved more easily.

The real life inspiration for F3L is that some tasks can more easily be accomplished when separated into smaller sub-tasks that can be learned individually. Developing the skills that allow to accomplish these tasks one at a time and then combining them is easier then attempting to learn the main task at once. In the next chapter, F3L is used to have a robot learning to extend its walking capabilities to sloped terrains.

Chapter 5

Generalizing walking to slopes in any direction on DARwIn-OP

In the last two chapters it was presented a succession of techniques that can be used when approaching reinforcement learning problems consisting in optimizing high dimensional continuous policies. In this chapter these techniques will be put into use in the task of generalizing the capabilities of a DARwIn-OP's simulation based controller capable of walking in flat terrain. This generalization will, to certain extents, turn the controller capable to walk on surfaces with a slope in any direction, i.e parallel and/or perpendicular to the walking direction. This will be achieved through the use of the learning mechanisms provided by F3L plus the integration of quantitative feedback mechanisms to the existing controller.

The goal of this chapter is twofold; in first place, to attest the capabilities of F3L and exemplify how it can provide a flexible way to approach a high dimensional continuous policy optimization reinforcement learning problem. Second, to act as a source of useful practical knowledge on future attempts to achieve complex, dynamic and environment-responsive locomotion through reinforcement learning.

5.1 Baseline

5.1.1 DARwIn-OP

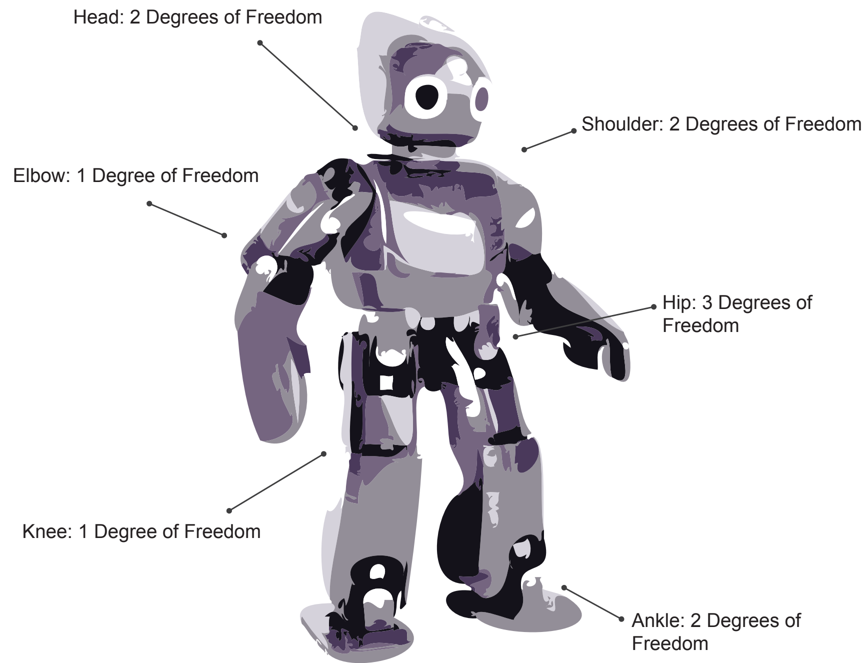


Figure 5.1: DARwIn-OP - Degrees of freedom.

DARwIn-OP is a humanoid biped robot platform. It has 20 degrees of freedom: 2 in the head, 3 in each arm and 6 in each leg (Fig. 5.1). Moreover, it has 4 Force-Sensing Resistor (FSR) sensors in the sole of each foot. These characteristics make it an adequate subject of study for testing locomotion learning algorithms.

5.1.2 Controller

The controller considered in the forthcoming sections is strongly based on the one described in [Matos and Santos, 2012]. It is a CPGs based controller implemented by means of a dynamical system that controls the robot's leg joints. Locomotion is achieved by the combination of three walking motions (Fig. 5.2) plus a turning motion:

- **Balancing** - Keep the robot's center of mass in a position that does not threaten balance.

- **Flexion** - Allows the moving foot to achieve vertical clearance during the swing phase of the step.
- **Compass** - Responsible for the propulsion of the body during locomotion.
- **Turning** - Allows the robot to turn.

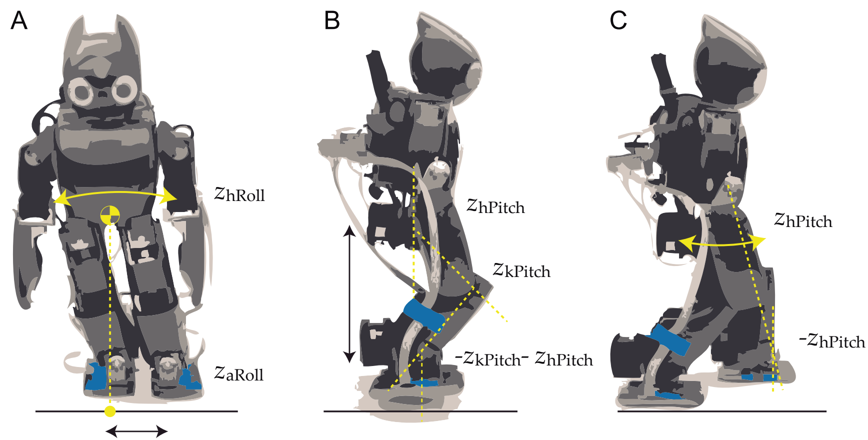


Figure 5.2: DARwIn-OP walking motions - balancing (**A**), flexion (**B**) and compass (**C**). Image adapted from [Matos and Santos, 2012].

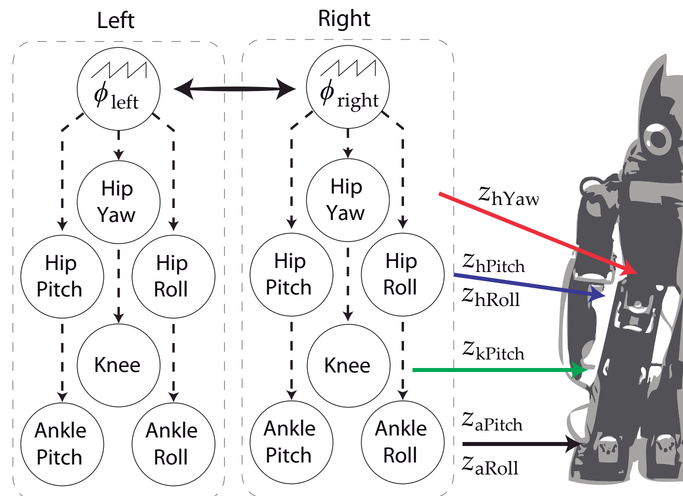


Figure 5.3: Oscillators coupling and influence in the controller's dynamical system. Image adapted from [Matos and Santos, 2012].

The set of rules governing this system is the following:

$$
\begin{aligned}
\dot{\phi}_i &= \omega + k\sin(\phi_i - \phi_o + \pi) \quad ,\forall (i,o) \in \{(\text{left},\text{right}),(\text{right},\text{left})\} \\
f^{\text{balancing}}_{i,\text{hRoll}} &= \begin{cases} -A_{\text{balancingLeft}}\,\omega \sin(\phi_i) & ,|\phi_{\text{left}}| > \frac{\pi}{2} \\ -A_{\text{balancingRight}}\,\omega \sin(\phi_i) & ,|\phi_{\text{left}}| \leq \frac{\pi}{2} \end{cases} \\
f^{\text{balancing}}_{i,\text{aRoll}} &= -f^{\text{balancing}}_{i,\text{hRoll}} \\
f^{\text{flex}}_{i,\text{hPitch}} &= \frac{A_{\text{flexHip}}\,\omega\phi_i}{\sigma^2} \exp\left(-\frac{\phi_i^2}{2\sigma^2}\right) \\
f^{\text{flex}}_{i,\text{kPitch}} &= \frac{A_{\text{flexKnee}}\,\omega\phi_i}{\sigma^2} \exp\left(-\frac{\phi_i^2}{2\sigma^2}\right) \\
f^{\text{flex}}_{i,\text{aPitch}} &= -(f^{\text{flex}}_{i,\text{hPitch}} + f^{\text{flex}}_{i,\text{kPitch}}) \\
f^{\text{turn}}_{i,\text{hYaw}} &= -A_{\text{turn}}\,\omega \sin\left(\phi_i + \frac{\pi}{2}\right) \\
f^{\text{compass}}_{i,\text{hPitch}} &= -A_{\text{compass}}\,\omega \sin\left(\phi_i + \frac{\pi}{2}\right) \\
f^{\text{compass}}_{i,\text{aPitch}} &= -f^{\text{compass}}_{i,\text{hPitch}} \\
\dot{z}_{i,\text{hRoll}} &= -\alpha(z_{i,\text{hRoll}} - O_{i,\text{hRoll}}) + f^{\text{balancing}}_{i,\text{hRoll}} \quad ,\forall i \in \{\text{left},\text{right}\} \\
\dot{z}_{i,\text{aRoll}} &= -\alpha(z_{i,\text{aRoll}} - O_{i,\text{aRoll}}) + f^{\text{balancing}}_{i,\text{aRoll}} \quad ,\forall i \in \{\text{left},\text{right}\} \\
\dot{z}_{i,\text{hYaw}} &= -\alpha(z_{i,\text{hYaw}} - O_{i,\text{hYaw}}) + f^{\text{turn}}_{i,\text{hYaw}} \quad ,\forall i \in \{\text{left},\text{right}\} \\
\dot{z}_{i,\text{hPitch}} &= -\alpha(z_{i,\text{hPitch}} - (O_{i,\text{hPitch}} + C_{i,\text{hPitch}})) + f^{\text{flex}}_{i,\text{hPitch}} + f^{\text{compass}}_{i,\text{hPitch}} \quad ,\forall i \in \{\text{left},\text{right}\} \\
\dot{z}_{i,\text{kPitch}} &= -\alpha(z_{i,\text{kPitch}} - (O_{i,\text{kPitch}} + C_{i,\text{kPitch}})) + f^{\text{flex}}_{i,\text{kPitch}} \quad ,\forall i \in \{\text{left},\text{right}\} \\
\dot{z}_{i,\text{aPitch}} &= -\alpha(z_{i,\text{aPitch}} - (O_{i,\text{aPitch}} + C_{i,\text{hPitch}} + C_{i,\text{kPitch}})) + f^{\text{flex}}_{i,\text{aPitch}} + f^{\text{compass}}_{i,\text{aPitch}} \quad ,\forall i \in \{\text{left},\text{right}\}
\end{aligned}
$$

with the additional restrictions:

$$
\begin{aligned}
O_{\text{left},j} &= O_{\text{right},j} \quad ,\forall j \in \{\text{hRoll,aRoll,hYaw,hPitch,kPitch,aPitch}\} \\
\omega_{\text{left}} &= \omega_{\text{right}}
\end{aligned}
$$

where:

- ϕ_{left} and ϕ_{right} are the two phase oscillators that keep the coordination between all the motions in the system (Fig. 5.3).
- ω is the rate at which the phase of the oscillators increases.
- k is the coupling strength between the oscillators.
- $f^{\square}_{\square}$ are the perturbations that generate the effect of each motion in the joins' positions.

- $A_{\square}^{\square}$ controls the emphasis with which the different motions are performed.
- $z_{\square,\square}$ are the robot's joint positions.
- $O_{\square}^{\square}$ are the offsets of the final generated rhythmic motion in each join.
- α controls the relaxation for the offsets.
- $C_{\square}^{\square}$ allow to control a baseline flexion in each leg that persists through the whole step.

ω, k, $A_{\square}^{\square}$, $O_{\square}^{\square}$, α and $C_{\square}^{\square}$ are all parameters of the system that can be changed in different environment conditions in order to provide more stable and/or faster locomotion. An important property of the aforementioned dynamical system is that the abrupt change of the value of any parameter does not cause an abrupt change in any of the joins' position. Instead it results in smooth modulations for the produced trajectories. This allows manipulating parameter values online without causing sudden movements which would very likely threaten the robot's balance.

5.2 Feedback mechanisms

The first step towards achieving locomotion on surfaces with a slope was adding feedback mechanisms that apply simple corrective behaviors during locomotion. Each of these mechanisms is explained next.

5.2.1 Ankle pitch correction

In general when robot's feet touch the ground, they must be parallel to it in order for the best support and balance to be achieved. It is also desirable for the weight of the robot to be distributed equally along the back and the front of the foot plant. Using the force sensors distributed in each corner of each foot plant, a mechanism to correct the ankle pitch angle can be established by the following transformation in the controller:

$$\begin{aligned}
F_{\text{pitch}} &= \eta_{\text{pitch}}((\text{FSR}_{\text{left,left,front}} + \text{FSR}_{\text{left,right,front}} + \text{FSR}_{\text{right,left,front}} + \text{FSR}_{\text{right,right,front}}) \\
&\quad - (\text{FSR}_{\text{left,left,back}} + \text{FSR}_{\text{left,right,back}} + \text{FSR}_{\text{right,left,back}} + \text{FSR}_{\text{right,right,back}})) \\
\dot{z}_{i,\text{aPitch}} &= -\alpha(z_{i,\text{aPitch}} - (O_{i,\text{aPitch}} + C_{i,\text{hPitch}} + C_{i,\text{kPitch}})) \\
&\quad + f_{i,\text{aPitch}}^{\text{flex}} + f_{i,\text{aPitch}}^{\text{compass}} + F_{\text{pitch}} \quad , \forall i \in \{\text{left}, \text{right}\}
\end{aligned}$$

where $\text{FSR}_{i,j,k}$ is the output of a FSR sensor in the i leg in the position (j,k) and η_{pitch} controls the strength of the applied correction.

This transformation increases or decreases $\dot{z}_{i,\text{aPitch}}$ according to the difference of measured forces in the sensors located in front of both feet plants and the sensors located in the back.

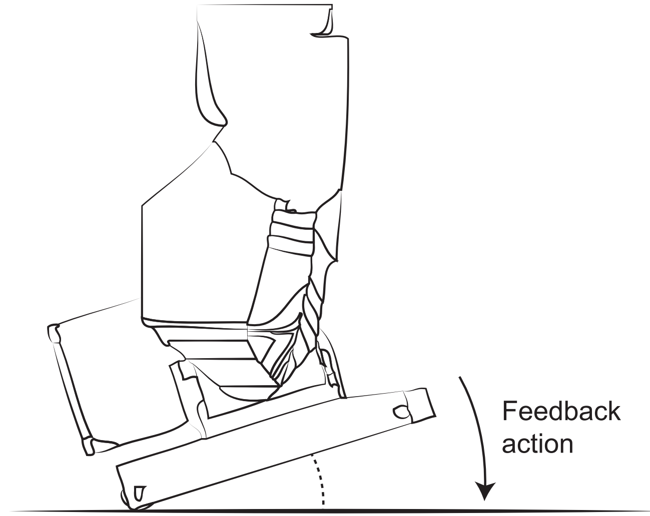


Figure 5.4: Effect of the ankle pitch correction feedback.

For instance, if the sum of the forces measured in the back sensors is greater than the sum of the forces measured in the front sensors, F_{pitch} will be negative, having the effect illustrated in Fig. 5.4.

5.2.2 Ankle roll correction

This correction mechanism is analogous to the one described in section 5.2.1 but it concerns the balance between the forces measured in the left and right sensors in each foot plant instead of the balance between the forces measured in the front and back sensors (Fig. 5.5). It can be applied with the following transformation in the controller:

$$\begin{aligned} F_{\text{roll}} = \eta_{\text{roll}}((&\text{FSR}_{\text{left,left,front}} + \text{FSR}_{\text{left,left,back}} + \text{FSR}_{\text{right,left,front}} + \text{FSR}_{\text{right,left,back}}) \\ &- (\text{FSR}_{\text{left,right,front}} + \text{FSR}_{\text{left,right,back}} + \text{FSR}_{\text{right,right,front}} + \text{FSR}_{\text{right,right,back}})) \\ \dot{z}_{i,\text{aRoll}} = &-\alpha(z_{i,\text{aRoll}} - O_{i,\text{aRoll}}) + f^{\text{balancing}}_{i,\text{aRoll}} + F_{\text{roll}} \quad , \forall i \in \{\text{left}, \text{right}\} \end{aligned}$$

where η_{roll} controls the strength of the applied correction.

5.2.3 Direction correction

Without some sort of direction control based on sensory information, it is very hard for a biped robot to keep walking in a straight line indefinitely. This phenomenon happens even in humans [Souman et al., 2009] . However small it may be, there is always a deviation rate that will slowly change the robot's facing direction. This is aggravated when the surface on which the robot is walking is inclined.

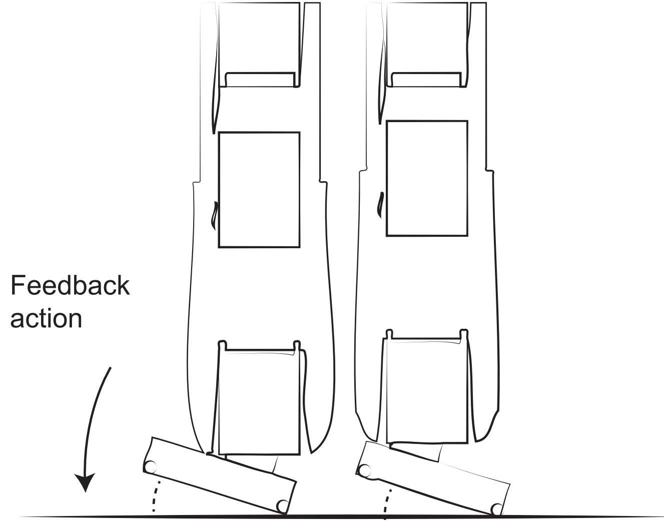


Figure 5.5: Effect of the ankle roll correction feedback.

In order to suppress this deviation, we can use a feedback mechanism that changes the A_{turn} value when needed, which perturbs the hip yaw join trajectory, causing the robot to turn. This can achieved with the following dynamical system inspired on the same dynamics that are used in DMPs [Ijspeert, 2002]:

$$\ddot{\theta} = \eta_{\alpha,\text{direction}}(\eta_{\beta,\text{direction}}(\theta_{\text{goal}} - \theta) - \dot{\theta})$$

,where θ is the facing direction of the robot, θ_{goal} is the intended direction and $\eta_{\alpha,\text{Direction}}$ and $\eta_{\beta,\text{Direction}}$ are parameters that regulate the damping properties of the dynamical system. While $\ddot{\theta}$ is not directly controllable, it is known that the A_{turn} controls the growing rate of $z_{\square,\text{hYaw}}$ which in its turn influences the robot's turning rate. So we can add the following rule to the controller:

$$\dot{A}_{\text{turn}} = \eta_{\alpha,\text{Direction}}(\eta_{\beta,\text{Direction}}(\theta_{\text{goal}} - \theta) - \dot{\theta})$$

While $\dot{A}_{\text{turn}}$ is not the same as $\ddot{\theta}$, it is related with it in a non linear fashion, which is enough to allow the dynamical system to attain the desired effect (Fig. 5.6). This feedback mechanism assumes that θ_{goal}, θ and $\ddot{\theta}$ are provided or calculated in someway beforehand. An advantage of this particular dynamical system is that changing the θ_{goal} value online does not cause any sudden change of the hip yaw angle. This allows using this mechanism not only to correct the direction of the robot but also to change it (Fig 5.7).

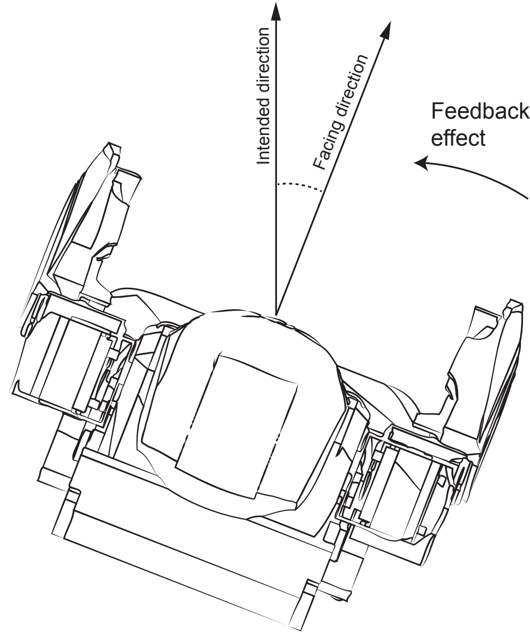


Figure 5.6: Effect of the direction correction feedback.

5.3 Achieving locomotion on sloped surfaces

5.3.1 Task

The task to accomplish is to have the robot adapting a subset of its controller parameters according to the inclination of the terrain. Without this, the default parameter values can only tolerate very small slope angles before the robot looses balance.

The inclination of the terrain is characterized by the slope in the walking direction and the slope in the direction perpendicular to the walking. From now on these will be referred to as frontal (s_{frontal}) and lateral (s_{lateral}) slopes, which are defined as:

$$
\begin{aligned}
s_{\text{frontal}} &= -\arctan(\nabla_{\mathbf{d}} f) \\
s_{\text{lateral}} &= -\arctan(\nabla_{\mathbf{r}} f) \\
\mathbf{r} &= R(-90^\circ) \cdot \mathbf{d}
\end{aligned}
$$

where $\mathbf{d}$ is the robot's facing direction unit vector, ∇v denotes the directional derivative along a vector v, and f is the function whose graph is the plane tangent to the terrain's surface in the robot's position. When the robot is walking up, s_{frontal} is negative. Likewise, when the terrain to the right is higher than the terrain to the left, s_{lateral} is negative.

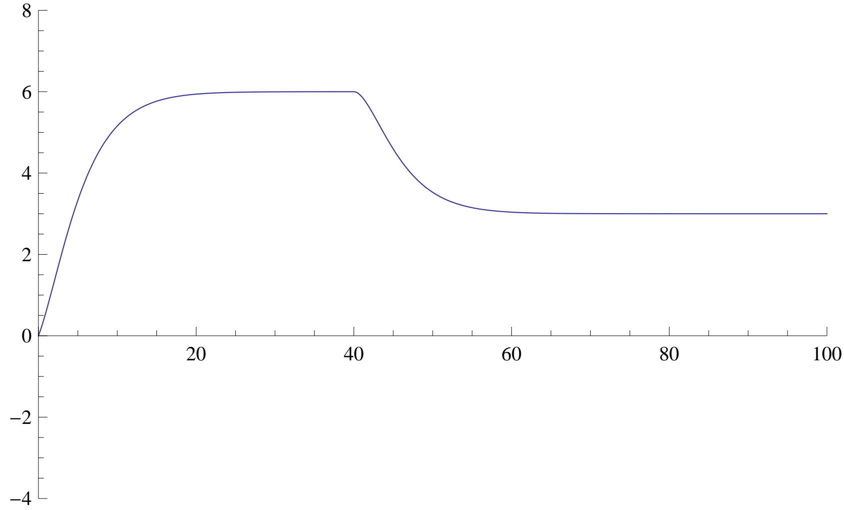


Figure 5.7: Plot of the development of θ when θ_{goal} is changed. In this example, θ_{goal} is set to 6 at $t = 0$ and set to 3 at $t = 40$.

With these definitions set, the task in hands is to learn and optimize a mapping:

$$\gamma_{walkSlope} : (s_{frontal}, s_{lateral}) \rightarrow P$$

where P is the set of parameter values that are adapted according to the slope of the terrain's surface. This set is formed by: $O_{\square,hPitch}$, $O_{\square,kPitch}$, $O_{\square,aPitch}$, $A_{flexHip}$, $A_{flexKnee}$, $C_{left,hPitch}$, $C_{left,kPitch}$, $C_{right,hPitch}$, $C_{right,kPitch}$, $O_{\square,aRoll}$, $A_{balancingLeft}$ and $A_{balancingRight}$ parameter values. Thus $\gamma_{walkSlope}$ can be written as:

$$\gamma_{walkSlope} : (s_{frontal}, s_{lateral}) \rightarrow (O_{\square,hPitch}, O_{\square,kPitch}, O_{\square,aPitch}, A_{flexHip}, A_{flexKnee}, C_{left,hPitch} \\ , C_{left,kPitch}, C_{right,hPitch}, C_{right,kPitch}, O_{\square,aRoll}, A_{balancingLeft}, A_{balancingRight})$$

which conveys the high dimensional nature of the problem with 12 parameters to be determined according to 2 state variables.

5.3.2 Simulation setup

In order to use reinforcement learning algorithms to optimize the mapping $\gamma_{walkSlope}$, one needs to be able to evaluate the fitness or cost of a given set of parameter values when facing a given $s_{frontal}$ and $s_{lateral}$. It is not trivial to setup a simulation that allows to do this; setting the slope angles and parameters immediately and at the same time the robot initiates its gait almost certainly causes the robot to fall. So instead, the simulation is setup so that it starts with the default parameters for the robot controller to walk on a flat terrain surface. Then, when the

robot initiates its gait, the parameters of the controller and slope angles of the terrain surface start to linearly fade into the final values. After 67 seconds, the final values are reached. The simulation keeps running for 33 more seconds and after that time it is terminated.

This setup avoids abrupt transitions of parameters and state variables, making it possible in each simulation to evaluate if the group of given parameter values allows to maintain stability for the supplied slope angles: inadequate parameters cause the robot to fall prematurely, while more adequate parameters can keep the robot in balance for longer, leading to lower costs.

5.3.3 Cost function

In order to evaluate the cost of a given set of parameters, the simulation is divided into a series of reward evaluation cycles, each with 960ms and 120 time steps. In each cycle i the reward R_i is a weighted geometric mean calculated as:

$$R_i = R_{\mathrm{e}}^{w_{\mathrm{e}}} \cdot R_{\mathrm{d}}^{w_{\mathrm{d}}}$$

subject to:

$$w_{\mathrm{e}} + w_{\mathrm{d}} = 1$$

where R_{e} and R_{d} are different reward metrics respective to efficiency and distance and w_{e}, w_{d} are their corresponding weights.

R_e quantifies how much of the traveled distance was in the intended direction and is calculated as:

$$R_{\mathrm{e}} = \frac{\Delta d}{D_{\mathrm{trav}}}$$

where d is the displacement of the robot in the intended direction and D_{travel} is the total traveled distance.

R_d accounts for the amount of traveled distance and is calculated as:

$$R_{\mathrm{d}} = \frac{D_{\mathrm{trav}}}{T_{\mathrm{cycle}} \cdot V_{\mathrm{max}}}$$

where T_{max} is the time of each reward evaluation cycle in seconds and V_{max} is an estimated upper bound for the robot's velocity. In the end of the simulation the episodic cost is calculated as:

$$C = \frac{1}{1 + \sum_i R_i}$$

where R_i is the reward of the evaluation cycle i.

In order to further reward low costs and penalize high costs, an additional transformation is applied to the cost (Fig. 5.8):

$$C_{\text{final}} = -0.061786 + 0.0618018 \cdot (49.381^{18.56996737C})$$

If the robot falls during the simulation, all the reward values for future evaluation cycles are considered as equal to zero.

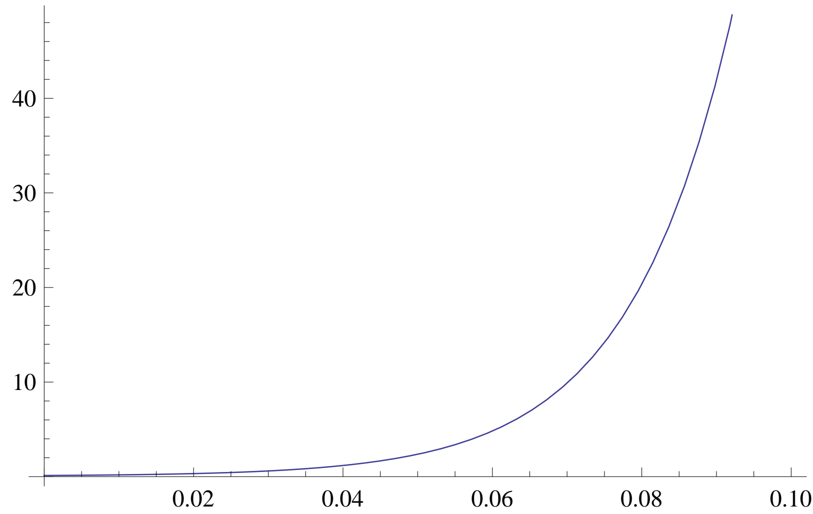


Figure 5.8: Final transformation applied to the cost function. This transformation increases the ratio between high and low costs.

5.3.4 Controller parameters

The controller parameters used in the simulations can be consulted in appendix A.

5.3.5 Learning structure

The applied learning structure divides $\gamma_{\text{walkSlope}}$ in different mappings that are learned progressively (Fig. 5.9)[1]:

- $\gamma_{\text{frontalA}} : s_{\text{frontal}} \to (O_{\square,\text{hPitch}}, O_{\square,\text{kPitch}}, O_{\square,\text{aPitch}})$
- $\gamma_{\text{frontalB}} : s_{\text{frontal}} \to (A_{\text{flexHip}}, A_{\text{flexKnee}})$
- $\gamma_{\text{lateralA}} : s_{\text{lateral}} \to (C_{\text{left,hPitch}}, C_{\text{left,kPitch}}, C_{\text{right,hPitch}}, C_{\text{right,kPitch}}, O_{\square,\text{aRoll}})$
- $\gamma_{\text{lateralB}} : s_{\text{lateral}} \to (A_{\text{balancingLeft}}, A_{\text{balancingRight}})$

[1] A,B,C and D are used to denote sub primitives in which the main primitives $\gamma_{\text{frontal}}, \gamma_{\text{lateral}}$ and γ_{omni} are branched

- $\gamma_{\text{omniA}} : (s_{\text{frontal}}, s_{\text{lateral}}) \rightarrow (O_{\square,\text{hPitch}}, O_{\square,\text{kPitch}}, O_{\square,\text{aPitch}})$
- $\gamma_{\text{omniB}} : (s_{\text{frontal}}, s_{\text{lateral}}) \rightarrow (A_{\text{flexHip}}, A_{\text{flexKnee}})$
- $\gamma_{\text{omniC}} : (s_{\text{frontal}}, s_{\text{lateral}}) \rightarrow (C_{\text{left,hPitch}}, C_{\text{left,kPitch}}, C_{\text{right,hPitch}}, C_{\text{right,kPitch}}, O_{\square,\text{aRoll}})$
- $\gamma_{\text{omniD}} : (s_{\text{frontal}}, s_{\text{lateral}}) \rightarrow (A_{\text{balancingLeft}}, A_{\text{balancingRight}})$

First γ_{frontalA} and γ_{frontalB} are trained. They are trained only with frontal slopes, as they are the primitives responsible for determining the parameter values that allow the robot to adapt to them (Fig. 5.10). Analogously, γ_{lateralA} and γ_{lateralB} are trained for the same kind of purpose for lateral slopes (Fig. 5.11). [2]

$\gamma_{\text{frontalA}}, \gamma_{\text{frontalB}}, \gamma_{\text{lateralA}}$ and γ_{lateralB} only admit a single state variable, but there may be situations where both slope angles are needed to determine the best parameters. For instance, knowing s_{frontal} may not be enough to determine the ideal $(A_{\text{flexHip}}, A_{\text{flexKnee}})$, as s_{lateral} may have some influence on the best choice for these parameters as well. For this reason $\gamma_{\text{omniA}}, \gamma_{\text{omniB}}, \gamma_{\text{omniC}}$ and γ_{omniD} are learned posteriorly, serving as a correction layer for those cases when knowing both s_{frontal} and s_{lateral} allows to output better parameter values (Fig. 5.12). The whole learning structure is schematized in Figs. 5.9, 5.10, 5.11 and 5.12.

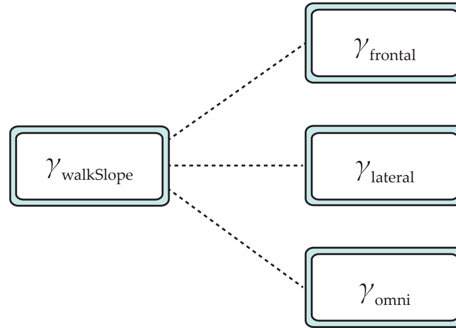


Figure 5.9: Initial primitive branching.

[2] It is not relevant whether γ_{frontal} or γ_{lateral} are trained first, since they do not depend on each other in any way before the training of γ_{omni}

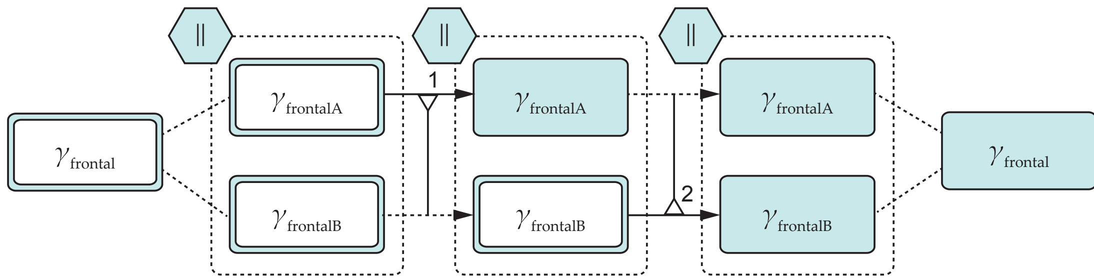


Figure 5.10: Learning γ_{frontal} gradually using partial learning, first for γ_{frontalA}, then for γ_{frontalB}.

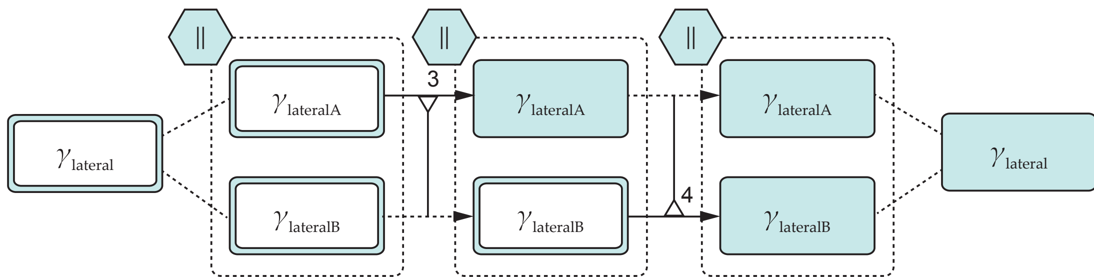


Figure 5.11: Learning γ_{lateral} gradually using partial learning, first for γ_{lateralA}, then for γ_{lateralB}.

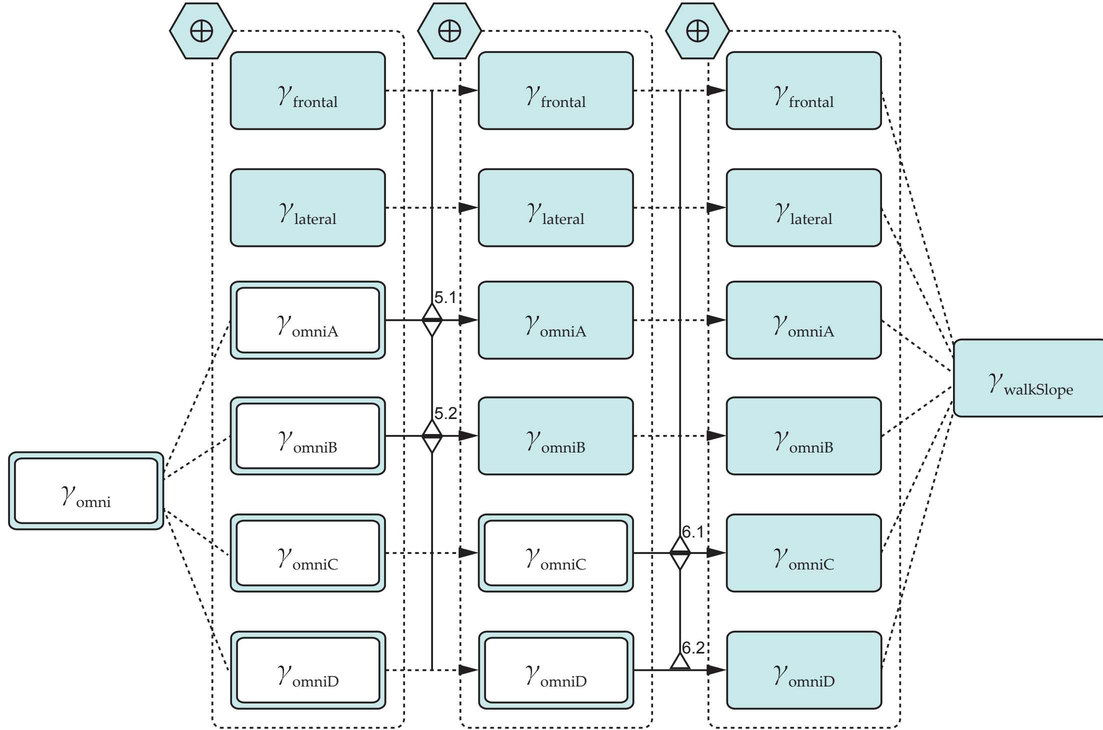


Figure 5.12: Learning γ_{omni}, a correction layer for when knowing both s_{frontal} and s_{lateral} allows to output better parameter values. Mutual learning of γ_{omniA} and γ_{omniB} is performed followed by mutual learning of γ_{omniC} and γ_{omniD}. In the end, all primitives are combined, constituting the $\gamma_{\text{walkSlope}}$ primitive.

Here $\|$ is the vector concatenation operator:

$$(a_1, a_{\dots}, a_n) \parallel (b_1, b_{\dots}, b_n) = (a_1, a_{\dots}, a_n, b_1, b_{\dots}, b_n)$$

and $\oplus$ an operator that concatenates vectors that correspond to different parameters and sums vectors that correspond to the same parameters, in this case defined as:

$$\oplus(\mathrm{x}, \mathrm{y}, \mathrm{z}, \mathrm{t}, \mathrm{w}, \mathrm{p}) = (\mathrm{x} + (\mathrm{z} \parallel \mathrm{t})) \parallel (\mathrm{y} + (\mathrm{w} \parallel \mathrm{p}))$$

5.3.6 Learning structure parameters

The used parameter values in the several steps of the learning structure can be consulted in appendix B.

5.4 Results

In order to test the effectiveness of the ankle pitch and ankle roll feedback mechanisms, the achieved cost for several states was measured before and after including them in the controller. The results can be consulted in Fig. 5.13. These feedback mechanisms caused the cost to decrease, on average ~ 0.058 in each tested state.

To illustrate the usefulness and test the effectiveness of the direction correction feedback, the robot's trajectory was recorded with this mechanism turned on and turned off. The graphic in Fig. 5.14 shows the difference between the recorded trajectories when it was intended for the robot to walk on a straight line. Also, to illustrate how well the direction correction feedback can be used for changing direction, the robot's trajectory was recorded during a simulation where the intended direction is different from the initial robot's facing direction. The recorded trajectory in this experiment is plotted in Fig. 5.15.

In Fig. 5.16, the costs evolution throughout the learning of the three subprimitives - γ_{frontal},γ_{lateral} and γ_{omni} - that together constitute the final $\gamma_{\text{walkSlope}}$ primitive is shown. The costs were evaluated every 90 rollouts during the algorithm execution. Each cost evaluation consisted in testing the achieved policy for a uniformly distributed set of states inside the range of states valid for the primitive being optimized. Then the obtained costs would be averaged - the final value is what is represented by the y-axis on Fig. 5.16.

The same tests and format used for Fig. 5.13 were used in Fig. 5.17 to show the difference between the performance of the controller before and after the learning is carried out. Here, the dark green cells show the states where the robot managed to complete a full simulation session without falling. On average, the robot managed to reduce the cost by ~ 0.566 in each tested state.

The ability to use the learned knowledge to adapt the robot's controller parameters while also changing direction was tested by recording the trajectory of the robot when walking around a pivot in a sloped surface. In each instant the robot would adapt its parameters according to the faced s_{frontal} and s_{lateral} values, and use the direction correction mechanism to set the intended direction to be perpendicular to the vector from the pivot to its position. Because the direction correction feedback is not immediate, the recorded trajectories are spiral-shaped instead of being completely circular. The robot fell when walking counter-clockwise but managed to complete several laps when walking clockwise. A video of this experiment is available at: `https://www.youtube.com/watch?v=QWtLn0PlHSE` .

Finally in Fig. 5.19 and Fig. 5.20, the robot is shown walking stably on surfaces with hard configurations. Videos of the robot walking in these conditions are available in: `https://www.youtube.com/watch?v=djlNcx9RKOA` and `https://www.youtube.com/watch?v=b3R0PzV0Big`.

	-0,2	-0,16	-0,12	-0,08	-0,04	0	0,04	0,08	0,12	0,16	0,2
-0,2	1,5438	1,2636	1,0932	0,8351	1,111	0,9291	0,9435	1,1336	1,1841	0,6217	1,7533
-0,16	0,658	1,2717	0,6306	0,5597	0,5664	0,5191	0,5687	0,5676	0,7054	1,0975	2,083
-0,12	0,9492	0,7012	0,6949	0,3052	0,3078	0,3144	0,3325	0,3915	0,4012	0,8821	1,2662
-0,08	0,8158	0,6988	0,3588	0,2707	0,1511	0,1393	0,1513	0,2168	0,5191	0,6655	2,6024
-0,04	1,0332	0,603	0,3121	0,1591	0,0719	0,0494	0,0897	0,2732	0,5797	0,6435	1,7069
0	0,7419	0,5768	0,3805	0,1658	0,0483	0,0488	0,0497	0,2159	0,4799	0,6174	1,2284
0,04	0,6769	0,4482	0,4074	0,1995	0,048	0,0475	0,071	0,1164	0,8057	0,6938	1,1957
0,08	0,4674	0,4195	0,5073	0,1957	0,1542	0,1445	0,1751	0,1729	0,3205	1,2284	1,0493
0,12	1,1702	0,6422	0,359	0,3303	0,3127	0,2733	0,2998	0,386	0,3382	1,9639	0,5028
0,16	0,6017	0,8663	0,7175	0,6037	0,5993	0,4313	0,5213	0,6089	0,8647	0,6682	6,0138
0,2	1,3197	1,3558	1,1782	0,9986	1,0165	0,8616	0,8715	1,4422	1,4914	2,9979	0,9512

(a)

	-0,2	-0,16	-0,12	-0,08	-0,04	0	0,04	0,08	0,12	0,16	0,2
-0,2	1,6862	1,2332	1,1546	0,8429	1,1262	0,8562	1,1353	1,1857	1,6499	1,8038	1,7643
-0,16	1,096	0,9708	0,8381	0,6379	0,6422	0,5299	0,5204	0,7311	0,838	0,9767	0,9427
-0,12	0,6868	0,6296	0,5053	0,3234	0,289	0,2745	0,3167	0,5322	0,4054	0,7757	1,2486
-0,08	0,6603	0,6152	1,098	0,1819	0,1445	0,1529	0,1707	0,1891	0,3498	0,7519	1,2329
-0,04	1,0935	0,8088	0,2311	0,3762	0,0817	0,0496	0,0827	0,277	0,6423	0,6457	1,8276
0	1,128	0,6584	0,2902	0,1714	0,0491	0,0486	0,0628	0,1978	0,4397	0,5592	1,7728
0,04	0,8023	0,5191	0,2497	0,1128	0,0484	0,0479	0,0486	0,1822	0,4332	0,8112	1,0614
0,08	0,907	0,5936	0,2053	0,204	0,1548	0,1447	0,1639	0,2146	0,2289	0,9463	1,1817
0,12	0,6452	0,4954	0,4401	0,3371	0,2879	0,2748	0,29	0,3932	0,5119	0,524	0,5968
0,16	1,2297	0,7469	0,6631	0,4848	0,4648	0,5263	0,4753	0,7758	0,7901	0,926	0,8018
0,2	1,3839	1,2287	1,1781	0,8691	0,8544	0,8822	0,781	1,0098	1,5271	1,6009	1,479

(b)

Figure 5.13: Costs before (Fig. 5.13a) and after (Fig. 5.13b) adding ankle pitch and ankle roll correction feedbacks. Each cell contains the cost obtained when running the simulation with the s_{frontal} and s_{lateral} values (radians) indicated in the left column and in the top row, respectively. Green cells represent lower costs and red cells represent high costs. These feedback mechanisms cause the cost to decrease, on average, by ~ 0.058 in each tested state.

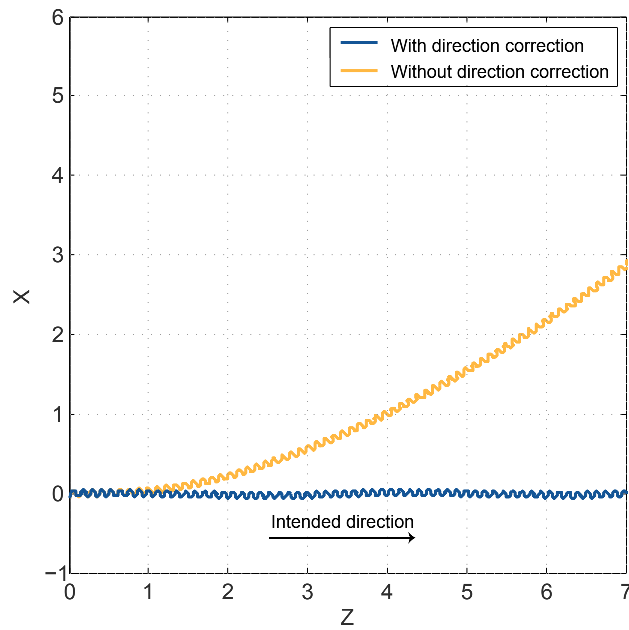


Figure 5.14: Trajectory of the robot with (blue) and without (orange) direction correction feedback on a flat surface.

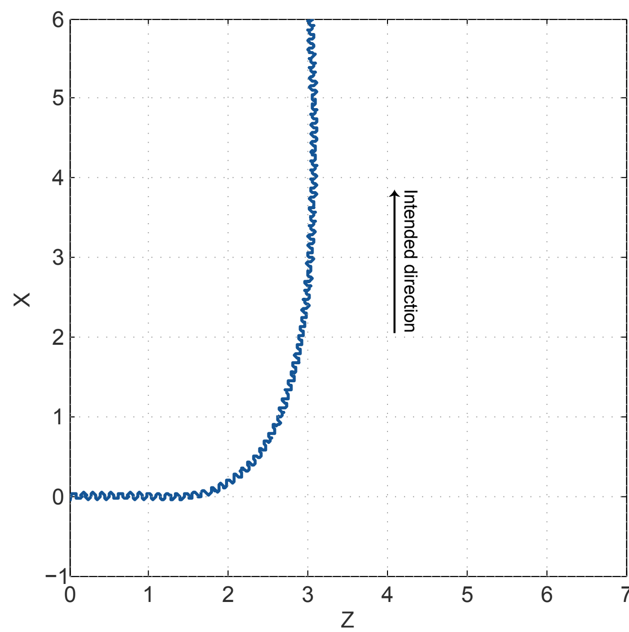
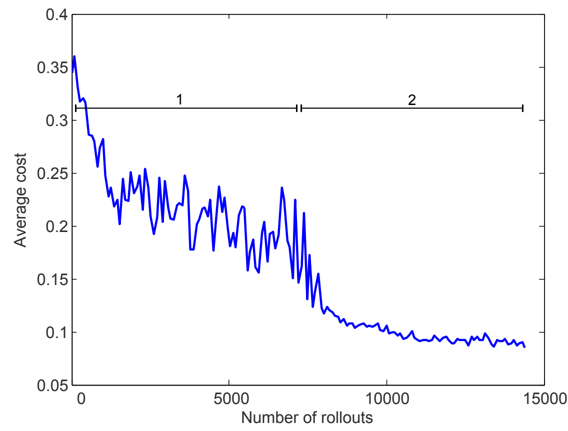
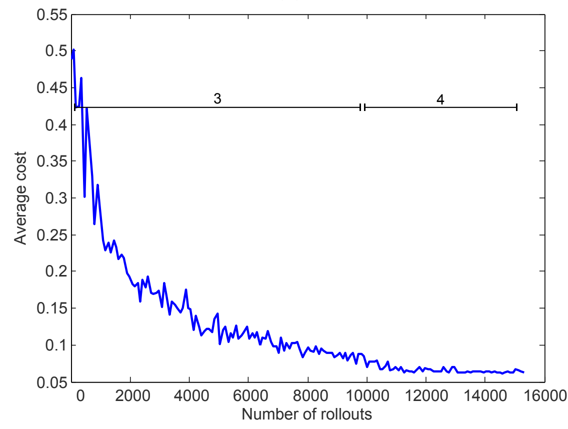


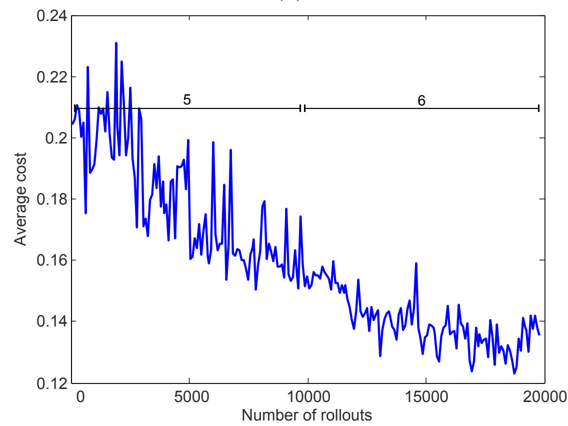
Figure 5.15: Trajectory of the robot on a flat surface when using direction feedback as a turning mechanism.



(a)

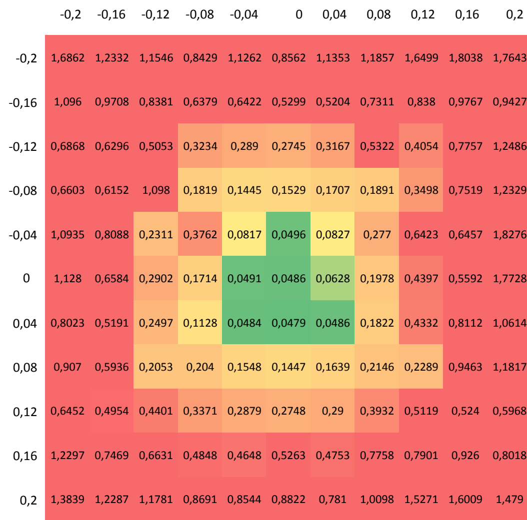


(b)



(c)

Figure 5.16: The costs evolution throughout the learning of the three subprimitives γ_{frontal} (Fig. 5.16a), γ_{lateral} (Fig. 5.16b) and γ_{omni} (Fig. 5.16c). The numbered line segments indicate the rollouts dedicated to each step delineated in the learning structure shown in Figs. 5.10, 5.11 and 5.12.



	-0,2	-0,16	-0,12	-0,08	-0,04	0	0,04	0,08	0,12	0,16	0,2
-0,2	1,6862	1,2332	1,1546	0,8429	1,1262	0,8562	1,1353	1,1857	1,6499	1,8038	1,7643
-0,16	1,096	0,9708	0,8381	0,6379	0,6422	0,5299	0,5204	0,7311	0,838	0,9767	0,9427
-0,12	0,6868	0,6296	0,5053	0,3234	0,289	0,2745	0,3167	0,5322	0,4054	0,7757	1,2486
-0,08	0,6603	0,6152	1,098	0,1819	0,1445	0,1529	0,1707	0,1891	0,3498	0,7519	1,2329
-0,04	1,0935	0,8088	0,2311	0,3762	0,0817	0,0496	0,0827	0,277	0,6423	0,6457	1,8276
0	1,128	0,6584	0,2902	0,1714	0,0491	0,0486	0,0628	0,1978	0,4397	0,5592	1,7728
0,04	0,8023	0,5191	0,2497	0,1128	0,0484	0,0479	0,0486	0,1822	0,4332	0,8112	1,0614
0,08	0,907	0,5936	0,2053	0,204	0,1548	0,1447	0,1639	0,2146	0,2289	0,9463	1,1817
0,12	0,6452	0,4954	0,4401	0,3371	0,2879	0,2748	0,29	0,3932	0,5119	0,524	0,5968
0,16	1,2297	0,7469	0,6631	0,4848	0,4648	0,5263	0,4753	0,7758	0,7901	0,926	0,8018
0,2	1,3839	1,2287	1,1781	0,8691	0,8544	0,8822	0,781	1,0098	1,5271	1,6009	1,479

(a)

	-0,2	-0,16	-0,12	-0,08	-0,04	0	0,04	0,08	0,12	0,16	0,2
-0,2	0,3588	0,112	0,1022	0,1012	0,1007	0,1018	0,1096	0,12	0,1032	0,0952	0,3617
-0,16	0,0792	0,0847	0,0526	0,0528	0,053	0,0528	0,0535	0,0536	0,0536	0,0544	0,0897
-0,12	0,0503	0,0517	0,0524	0,0522	0,0521	0,0516	0,0519	0,0516	0,0516	0,0524	0,051
-0,08	0,0926	0,0507	0,0507	0,0504	0,0507	0,0508	0,0511	0,0509	0,0506	0,0516	0,075
-0,04	0,1136	0,0502	0,0503	0,0502	0,0499	0,0499	0,0497	0,0495	0,0506	0,0501	0,0867
0	0,0961	0,049	0,0492	0,049	0,0488	0,0484	0,0487	0,0487	0,049	0,0492	0,0953
0,04	0,1083	0,0478	0,0481	0,0482	0,0477	0,0475	0,0481	0,048	0,0481	0,0481	0,0987
0,08	0,1993	0,1369	0,1315	0,111	0,0475	0,0478	0,0475	0,0473	0,0475	0,0474	0,1069
0,12	0,1446	0,1183	0,1244	0,1173	0,1063	0,1184	0,098	0,0487	0,1208	0,1203	0,126
0,16	0,2165	0,1674	0,1619	0,1374	0,1365	0,1379	0,1422	0,1586	0,1317	0,1806	0,1964
0,2	0,3739	0,2154	0,2088	0,2419	0,2377	0,2445	0,2557	0,2571	0,2778	0,2774	0,3119

(b)

Figure 5.17: Costs before (Fig. 5.17a) and after (Fig. 5.17b) performing the learning process. Green cells indicate s_{frontal} and s_{lateral} values where the robot did not fall during the simulation, and red cells indicate s_{frontal} and s_{lateral} values that led to high costs. The achieved improvement is reflected in the cost decreasing ~ 0.566, on average, in each tested state. The range of slope angles which the robot can sustain is greatly increased by the learning.

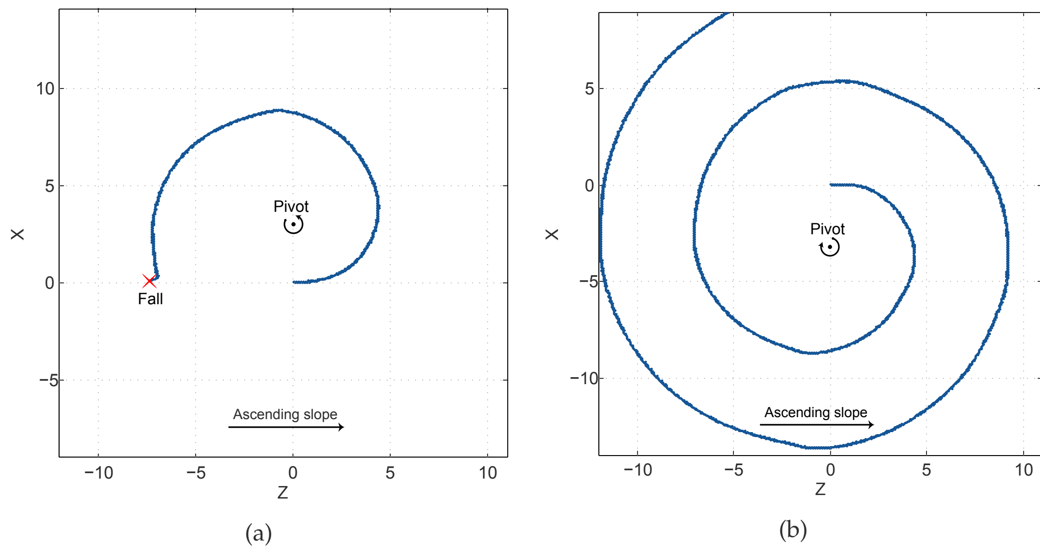


Figure 5.18: Trajectory of the robot when walking around a pivot, in a sloped surface with a slope of 0.08rad or 4.58°, using the knowledge acquired in the learning process. In each instant the robot adapts its parameters according to the faced s_{frontal} and s_{lateral} values, and uses the direction correction mechanism to set the intended direction to be perpendicular to the vector from the pivot to its position. The robot falls when walking counter-clockwise (Fig. 5.18a), but manages to complete a full lap when walking clockwise (Fig. 5.18b).

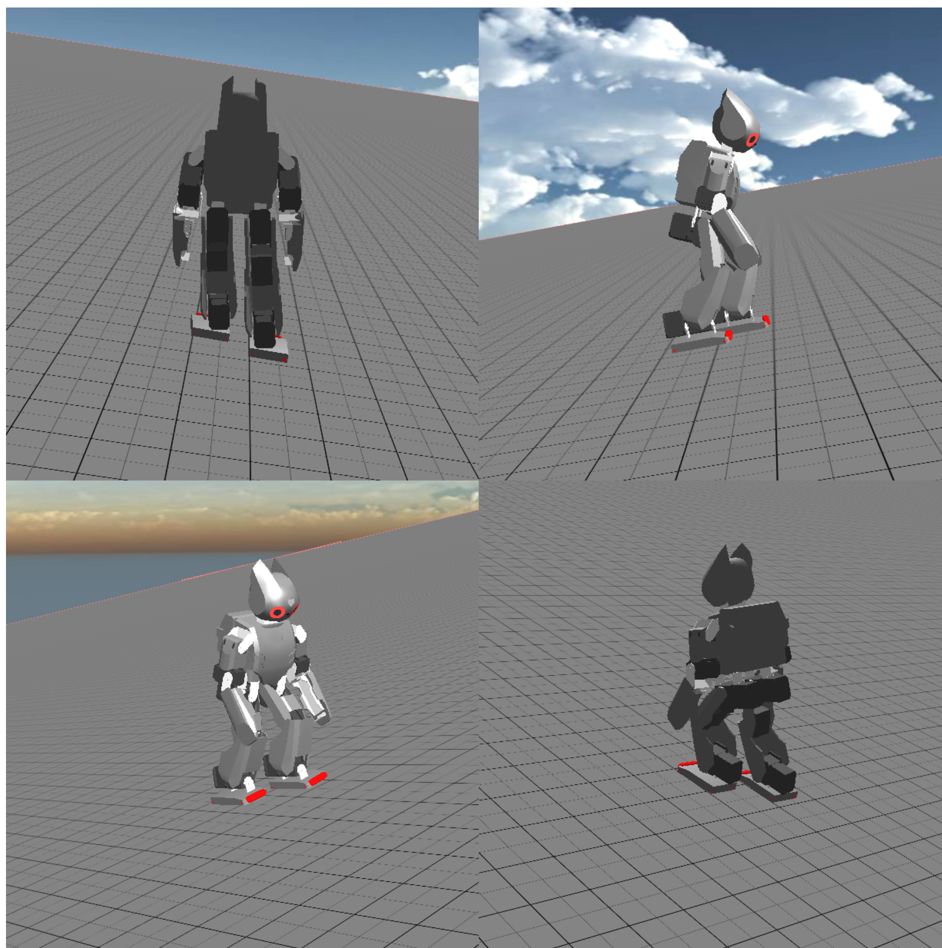


Figure 5.19: DARwIn-OP with a stable walking when s_{frontal} is equal to -0.16rad or -9.16° and s_{lateral} is equal to 0.16rad or 9.16°.

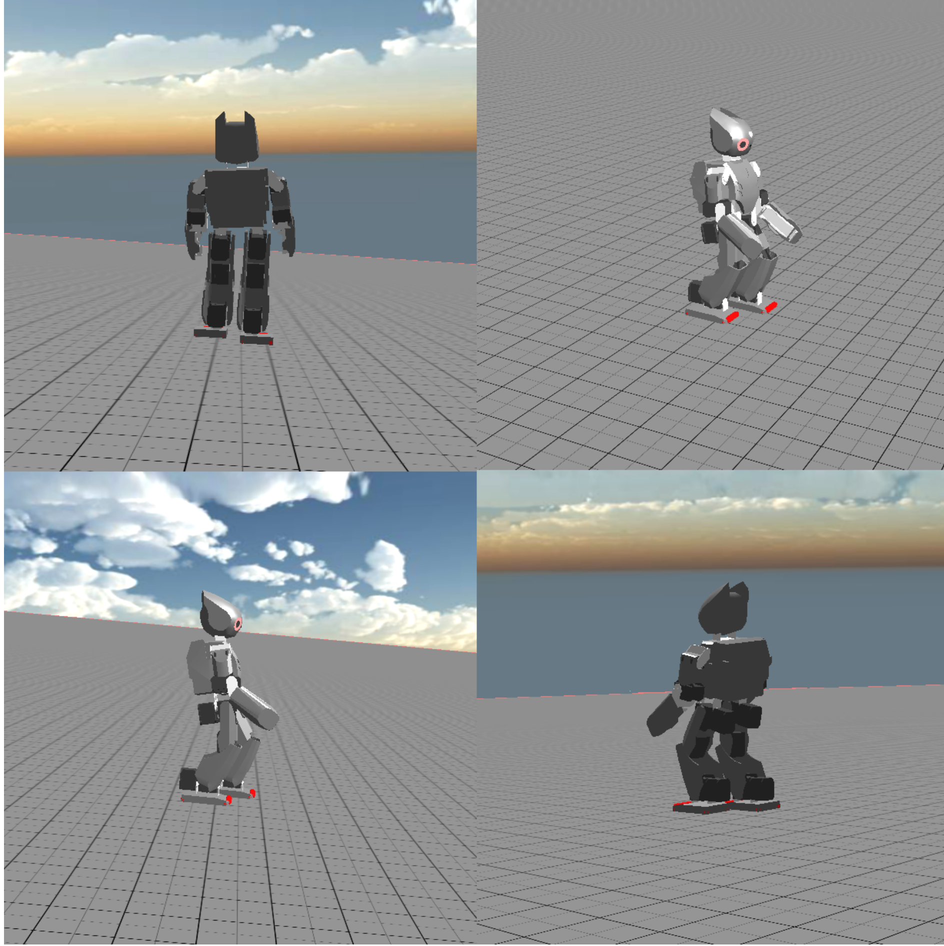


Figure 5.20: DARwIn-OP with a stable walking when s_{frontal} is equal to 0.1rad or 5.73° and s_{lateral} is equal to 0.08rad or 4.58°.

5.5 Discussion

From Fig. 5.13, it is possible to conclude that the ankle pitch and ankle roll correction feedback mechanisms can bring an overall slight improvement, on average, when considering the range of all tested s_{frontal} and s_{lateral} . However, some slope value combinations led to higher costs when these feedback mechanisms were used. This suggests that η_{pitch} and η_{roll} might have not been perfectly tuned, or that better dynamics for implementing this behavior might exist. In alternative, it could also be possible to learn a mapping to update these parameters according to s_{frontal} and s_{lateral} using F3L techniques.

The effectiveness of the direction correction feedback is noticeable in Figs. 5.14 and 5.15. In Fig. 5.14, it is possible to observe that this feedback mechanism can correct the robot's direction

in order to have it walking in a straight line. In Fig. 5.15, it can be observed that this feedback mechanism may also be used to change the direction while walking. While great changes of direction may lead to unstable behavior, this can be solved by imposing limits on A_{turn} and/or $\dot{z}_{\square,\text{hipYaw}}$, or by adjusting $\eta_{\alpha,\text{Direction}}$ and $\eta_{\alpha,\text{Direction}}$, trading a faster turning for a safer one. The direction correction mechanism was also important during the learning, because it forced the robot to go on a straight line and face the terrain slope in the intended direction. Without it, the learning algorithm could possibly find primitives which would map to parameters that wouldn't cause to robot to fall but would make it avoid the slope by changing direction.

In Fig. 5.16c it is possible to observe that the costs kept decreasing throughout the rollouts during the learning of each subprimitive, but started to stabilize in the end of the process for γ_{frontal} and γ_{lateral}. For γ_{omni}, it appears that there was still more potential for improvement as the recorded costs don't show significant signs of stabilization yet.

In Fig. 5.17, it can be observed how much the learned knowledge improved the ability for DARwIn-OP to walk in sloped terrains. The range of s_{frontal} and s_{lateral} value combinations on which DARwIn-OP can walk was dramatically extended. F3L and CrKR^{++} have successfully been capable of learning the complex mapping from 2 state variables to 12 parameters. Certainly, this mapping is not optimal, but it allows for an unquestionable and relevant improvement when comparing the final and the initial results. Moreover, further improvements could be achieved by continuing to develop the learned primitives. Some extreme examples of the application of the learned knowledge can be observed in Figs. 5.19 and 5.20.

Fig. 5.18, shows the trajectories of the DARwIn-OP when walking around a pivot in a sloped surface. Even though this is not a task for which it was specifically trained during the learning process, it manages to complete several laps without falling when walking clockwise, and half a lap counter-clockwise. It is likely that with an additional learning step, with a specific task designed to test this aspect, the robot would be capable of walking in both directions in a sloped terrain with moderate inclinations without falling.

5.6 Summary

In this chapter, the F3L framework presented in chapter 4 was used in order to generalize a DARwIn-OP's controller walking capabilities to sloped terrains. The achieved results are gratifying and promising and attest the fact that F3L can provide an effective and useful tool to structure the learning process of complex tasks. The feedback mechanisms also proved to be useful, especially the direction correction mechanism, as it can be used both to correct the direction of the robot and to change it while walking.

Chapter 6

Final remarks

6.1 Conclusions

Robotics is a field that is still in its early stages and it is expected to become more and more relevant in medical, social and educational contexts in the future. This thesis introduced some techniques that may have the potential to contribute to the development of robotics in those and other areas. Robots need to interact with the world in order to perform the tasks that they are designed for. These interactions will become increasingly more complex as they are used in a greater range of situations. However, it is extremely difficult for the hardware and software developers to prepare a robot for every situation that it might face. Therefore, achieving a stage where robots can learn and adapt most of their behavior is a crucial step towards unveiling the full potential that they have to offer.

While the techniques introduced in this thesis are specially conceived for tasks related with movement learning in robots, they can be used in other problems where there is the need to perform learning of continuous action and state space parameterized policies.

CrKR^{++} extends the capabilities of CrKR, bringing the possibility to learn complex primitives, even if in order to do that many training samples might need to be acquired. Continuously learning how to define controller parameters in the best way according to certain task conditions is a relevant capability that could improve robots performance in assignments like moving objects, performing locomotion, or opening and closing doors. It is in fact an extremely versatile and valuable adaptation mechanism that demonstrates how important CrKR^{++} can be.

F3L brings tools that allow to divide the learning process of a task in several steps and with that perform each step in separate learning modes involving multiple or single primitives. Each step can be configured individually with different learning parameters, cost functions or even training tasks. This makes it easier to apply CrKR^{++} for high dimensional tasks. Besides,

can F3L improve the tractability of some reinforcement problems if the the primitive to be learned is divided into subprimitives that can be learned faster and the learning process is conducted adequately.

In the last chapter, it was exemplified how F3L could be employed in a useful and concrete context. Together with feedback mechanisms, a controller that was only able to generate stable locomotion for almost flat surfaces was turned into one which could adapt to a range of frontal and lateral slope angles while walking. Some capability of changing direction during such task was also achieved. This ultimately attests the usefulness that the techniques presented in this thesis can provide.

We hope these contributions to help pushing the robotics field forward.

6.2 Future work

In order for the tools presented in this thesis to become more mature and usable by other researchers and roboticists, some development and research has still to be done. In the following we present some improvements and research lines that can add value to these tools.

Transparent parameter selection

Using CrKR^{++} requires one to set the values of a considerable number of parameters. Sometimes this demands having knowledge on how the algorithm works internally. Without this knowledge, the end result may be frustrating. It would be convenient to make this process more transparent for anyone using the algorithm, by automatically setting parameter values whenever possible and by creating precise and short documentation that anyone interested could follow in order to use the algorithm.

Dynamic representational power

CrKR^{++} could benefit from a mechanism that would allow one to have the representational power of the policy to be optimized growing with the number of samples. It is possible for the algorithm not being able to find a better policy because the used kernel functions are too wide to allow the encoding of small details . This speeds up the convergence of the algorithm but limits the maximum quality that can be achieved for the policy to be optimized. Adapting these kernels as the number of samples grows could improve the potential of the algorithm when used in long training sessions, without rendering it useless for short training sessions.

GUI to design and generate learning structures

Creating the classes that define the settings for each CrKR^{++} unit and that combine the several units into a primitive mix can be a slow and error-prone process. A Graphical User Interface (GUI) could automatize this process by allowing to generate all needed files after choosing the values for the settings of the CrKR^{++} units. Additionally this would be possible for a whole learning structure scheme.

F3L tests

The F3L framework should be tested for a larger number of problems. This would unveil its true potentialities and limitations.

Research and cataloging of learning patterns

There may be patterns of learning structures, i.e, sequences of F3L operations that work specially well for certain classes of problems. It would be relevant to explore and catalog the existence of such patterns.

Different dynamics for the ankle pitch and ankle roll feedback mechanisms

The ankle pitch and ankle roll correction mechanisms used in DARwIn-OP during the performed simulations only brought slight improvements to the preexisting controller. There may be other dynamics that would allow to use the FSR sensors in the feet of the robot in a more adequate way. It would be relevant to explore the existence of such dynamics.

Perform tests in real environments

The achieved results for DARwIn-OP walking on sloped surfaces were obtained via simulation. Despite the realistic physics engine used by the simulation software, it would be important to test the capabilities of F3L in a non virtual environment, in order to verify if it could still be able to achieve good results with the presence of noise in actuators and sensors.

References

[Brown, 1911] Brown, T. G. (1911). The Intrinsic Factors in the Act of Progression in the Mammal. *Proceedings of the Royal Society of London. Series B, Containing Papers of a Biological Character*, 84(572):308–319.

[Büschges and Borgmann, 2013] Büschges, A. and Borgmann, A. (2013). Network modularity: back to the future in motor control. *Current biology : CB*, 23:R936–8.

[Caldwell, 2012] Caldwell, D. G. (2012). Direct Policy Search Reinforcement Learning based on Particle Filtering. (2010):1–13.

[Cappelletto and Estévez, 2007] Cappelletto, J. and Estévez, P. (2007). Gait Synthesis in Legged Robot Locomotion using a CPG-Based Model. *...and Robotics: ...*, (September).

[Dimitrijevic et al., 1998] Dimitrijevic, M. R., Gerasimenko, Y., and Pinter, M. M. (1998). Evidence for a spinal central pattern generator in humans. *Annals of the New York Academy of Sciences*, 860:360–76.

[Haschke, 2012] Haschke, R. (2012). Learning Manipulation Patterns.

[Heo et al., 2012] Heo, J., Lee, I., and Oh, J. (2012). Development of Humanoid Robots in HUBO Labora-tory, KAIST. 30(4):367–371.

[Hirose and Ogawa, 2007] Hirose, M. and Ogawa, K. (2007). Honda humanoid robots development. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 365(1850):11–9.

[Ijspeert, 2002] Ijspeert, A. (2002). Learning attractor landscapes for learning motor primitives. *Advances in Neural Information Processing Systems*.

[Ijspeert, 2008] Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review. *Neural networks : the official journal of the International Neural Network Society*, 21(4):642–53.

[Ijspeert et al., 2007] 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 (New York, N.Y.)*, 315(5817):1416–20.

[Kalakrishnan et al., 2009] Kalakrishnan, M., Buchli, J., Pastor, P., and Schaal, S. (2009). Learning locomotion over rough terrain using terrain templates. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 167–172.

[Kalakrishnan et al., 2012] Kalakrishnan, M., Righetti, L., Pastor, P., and Schaal, S. (2012). Learning Force Control Policies for Compliant Robotic Manipulation. *ICML*.

[Kimura et al., 2007] Kimura, H., Fukuoka, Y., and Cohen, A. H. (2007). Adaptive Dynamic Walking of a Quadruped Robot on Natural Ground Based on Biological Concepts.

[Klein and Lewis, 2012] Klein, T. J. and Lewis, M. A. (2012). A physical model of sensorimotor interactions during locomotion. *Journal of neural engineering*, 9(4):046011.

[Kober, 2012] Kober, J. (2012). Learning Motor Skills: From Algorithms to Robot Experiments. (April).

[Kober and Peters, 2010] Kober, J. and Peters, J. (2010). Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203.

[Kober et al., 2012] 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.

[Kormushev et al., 2013] Kormushev, P., Calinon, S., and Caldwell, D. (2013). Reinforcement Learning in Robotics: Applications and Real-World Challenges. *Robotics*, 2(3):122–148.

[Matos and Santos, 2012] Matos, V. and Santos, C. (2012). Central Pattern Generators with phase regulation for the control of humanoid locomotion. *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*, pages 134–139.

[Matos and Santos, 2010] Matos, V. and Santos, C. P. (2010). Omnidirectional locomotion in a quadruped robot: A CPG-based approach. *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3392–3397.

[Morimoto et al., 2008] Morimoto, J., Endo, G., Nakanishi, J., and Cheng, G. (2008). A Biologically Inspired Biped Locomotion Strategy for Humanoid Robots: Modulation of Sinusoidal Patterns by a Coupled Oscillator Model. *IEEE Transactions on Robotics*, 24.

[Mülling and Kober, 2013] Mülling, K. and Kober, J. (2013). Learning to select and generalize striking movements in robot table tennis. *... Journal of Robotics ...*, pages 1–24.

[Nor and Ma, 2013] Nor, N. and Ma, S. (2013). A simplified cpgs network with phase oscillator model for locomotion control of a snake-like robot. *Journal of Intelligent & Robotic Systems*, pages 1–16.

[Peters and Schaal, 2008] Peters, J. and Schaal, S. (2008). Natural actor-critic. *Neurocomputing*, pages 280–291.

[Pongas et al., 2005] Pongas, D., Billard, a., and Schaal, S. (2005). Rapid synchronization and accurate phase-locking of rhythmic motor primitives. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2911–2916.

[Raibert and Blankespoor, 2008] Raibert, M. and Blankespoor, K. (2008). Bigdog, the rough-terrain quadruped robot. *Proceedings of the 17th ...*.

[Robinson, 2005] Robinson, B. S. (2005). Toward an Optimal Algorithm for Matrix Multiplication. *News Journal of the Society for Industrial and Applied*, 38:1–3.

[Rubrecht et al., 2012] Rubrecht, S., Padois, V., Bidaud, P., Broissia, M., and Da Silva Simoes, M. (2012). Motion safety and constraints compatibility for multibody robots. *Autonomous Robots*, 32(3):333–349.

[Schmidt and Wrisberg, 2000] Schmidt, R. A. and Wrisberg, C. A. (2000). *Motor Learning and Performance*, volume 2nd editio.

[Shen et al., 2012] Shen, H., Yosinski, J., Kormushev, P., Caldwell, D. G., and Lipson, H. (2012). Learning Fast Quadruped Robot Gaits with the RL PoWER Spline Parameterization. *Cybernetics and Information Technologies*, 12(3):66–75.

[Souman et al., 2009] Souman, J. L., Frissen, I., Sreenivasa, M. N., and Ernst, M. O. (2009). Walking straight into circles. *Current biology : CB*, 19(18):1538–42.

[Spröwitz et al., 2013] Spröwitz, A., Tuleu, A., Vespignani, M., Ajallooeian, M., Badri, E., and Ijspeert, A. J. (2013). Towards dynamic trot gait locomotion: Design, control, and experiments with cheetah-cub, a compliant quadruped robot. *Int. J. Rob. Res.*, 32(8):932–950.

[Stulp and Sigaud, 2012a] Stulp, F. and Sigaud, O. (2012a). Path integral policy improvement with covariance matrix adaptation. *arXiv preprint arXiv:1206.4621*.

[Stulp and Sigaud, 2012b] Stulp, F. and Sigaud, O. (2012b). Policy Improvement Methods: Between Black-Box Optimization and Episodic Reinforcement Learning. pages 1–34.

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (1998). *Reinforcement learning: An introduction*.

[Tamosiunaite et al., 2011] Tamosiunaite, M., Nemec, B., Ude, A., and Wörgötter, F. (2011). Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives.

[Theodorou et al., 2010a] Theodorou, E., Buchli, J., and Schaal, S. (2010a). A generalized path integral control approach to reinforcement learning. *The Journal of Machine Learning ...*.

[Theodorou et al., 2010b] Theodorou, E., Buchli, J., and Schaal, S. (2010b). Reinforcement learning of motor skills in high dimensions: A path integral approach. *2010 IEEE International Conference on Robotics and Automation*, 1(3):2397–2403.

[Vlassis et al., 2009] Vlassis, N., Toussaint, M., Kontes, G., and Piperidis, S. (2009). Learning model-free robot control by a Monte Carlo EM algorithm.

[Vukobratović and Borovac, 2004] Vukobratović, M. and Borovac, B. (2004). Zero moment point thirty five years of its life. *International Journal of Humanoid ...*, 01(01):157–173.

[Williams, 1992] Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning 1 Introduction. pages 1–27.

[Williams, 2011] Williams, V. V. (2011). Breaking the Coppersmith-Winograd barrier. *Tensor*, pages 1–71.

Appendix A

Parameter values for the controller

Table A.1: Initial parameter values for the DARwIn-OP controller used during the simulations.

Parameter	Value
ω	$\frac{\pi}{0.64}$
k	7
α	1
σ	$\frac{\pi}{6}$
$A_{\text{balancingLeft}}$	14
$A_{\text{balancingRight}}$	14
A_{flexHip}	15
A_{flexKnee}	30
A_{turn}	0
A_{compass}	11
$O_{\square,\text{hRoll}}$	0
$O_{\square,\text{aRoll}}$	0
$O_{\square,\text{hYaw}}$	0
$O_{\square,\text{hPitch}}$	-25
$O_{\square,\text{kPitch}}$	40
$O_{\square,\text{aPitch}}$	20
$C_{\text{left,hPitch}}$	0
$C_{\text{left,kPitch}}$	0
$C_{\text{right,hPitch}}$	0
$C_{\text{right,kPitch}}$	0
η_{pitch}	0.0025
η_{roll}	0.005
$\eta_{\alpha,\text{Direction}}$	4
$\eta_{\beta,\text{Direction}}$	0.1

Appendix B

Parameter values for the learning process

Table B.1: Parameter values for the several stages of the learning process of generalizing DARwIn-OP 's walking capabilities. The step numbers indicated in the table's headers are marked in Figs. 5.9, 5.10, 5.11 and 5.12.

Parameter	1	2	3	4	5.1	5.2	6.1	6.2
maximumCost	1.804							
maximumCostStandardized	10							
lambda	150							
varianceMultiplier	25	15	25	12	15	9	15	7.5
minimumVariance	0.25							
kernelStdDev	0.05				0.06			
kernelMultiplier	1							
agingFactor	0.000025							
maxAdmissibleDis...[1]	0.4							
lambdaForStandardizedCosts	25							
standardizePoolSize	300				450			
numberTryoutsCons...[2]	9							
rolloutsPerRun	900							
samplesToStop	7200	7200	9900	5400	9900		9900	

[1] maxAdmissibleDistanceToUseStandardizePool
[2] numberTryoutsConsideredWhenStandardizing