Graz University of Technology

Balancing Central Pattern Generator based Humanoid Robot Gait using Reinforcement Learning

A thesis submitted in partial fulfilment of the

requirements for the degree of *Diplom-Ingenieur* by

Florian Hackenberger Bakk. rer. soc. oec

4th October 2007

Supervisor

O.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass
Institute for Theoretical Computer Science, Austria

Abstract

This thesis concerns itself with the development of a nonlinear feedback policy for balancing a humanoid robot during a walking gait. Finding such a policy is important as active balance control is required for long distance walking and for rough terrain in general. The system developed during the research consists of two modules. One module is a Programmable Central Pattern Generator which reproduces a walking trajectory supplied by the vendor of the robot used for this thesis. The walking pattern is then modified by the second component, a nonlinear feedback policy derived by a Reinforcement Learning agent. This architecture is appealing because the use of Programmable Central Pattern Generators enables the incorporation of feedback into the equations of the system generating the walking trajectories. Further, the pattern produced by the Central Pattern Generator is robust to perturbations and most importantly, the speed of the walking gait can be varied smoothly. Reinforcement Learning is a natural choice, as this task requires explorative interaction between the robot and the environment and learning from past experience. The state space chosen for the Reinforcement Learning task encodes the state of the robot in a single variable (the phase of the first oscillator), which enables the task to be solved in reasonable time in the first place. First we present the theory of Programmable Central Pattern Generators and introduce a new coupling scheme which improves upon the previously published approach. The second part of this thesis introduces the reader to Reinforcement Learning and presents the two algorithms used for deriving the feedback policy. The thesis concludes by describing the learning task and presenting the results. The learnt policy is able to make the robot walk for up to 70 seconds on average, which corresponds to about 42 complete (left and right) steps.

Keywords: humanoid robotics, programmable central pattern generator, reinforcement learning, balancing, robot control, walking gait

Kurzfassung

Diese Master Arbeit befasst sich mit der Entwicklung einer nicht-linearen Feedback Policy um einen humanoiden Roboter während des Gangs zu stabilisieren. Solch eine Policy zu finden is insofern wichtig, als eine aktive Stabilitätskontrolle für das Zurücklegen weiterer Strecken und für unebenen Untergrund unabdingbar ist. Das entwickelte System besteht aus zwei Modulen. Ein Modul ist ein Programmable Central Pattern Generator der eine Gang-Trajektorie wiedergibt, welche vom Hersteller des verwendeten Roboters zur Verfügung gestellt wird. Diese Trajektorie wird von einer zweiten Komponente, einer nicht-linearen Feedback Policy welche mit Reinforcement Learning abgeleitet wurde, modifiziert. Diese Architektur ist insofern attraktiv, als durch die Verwendung von Programmable Central Pattern Generatoren das Feedback in die Gleichungen des Systems zur Generierung der Trajektorien einfließen kann. Weiters ist das von dem Central Pattern Generator erzeugte Bewegungsmuster robust gegen Störeinflüsse und vor allem kann die Geschwindigkeit der Bewegung kontinuierlich angepasst werden. Reinforcement Learning is eine naheliegende Wahl, da die Aufgabe eine erforschende Interaktion zwischen dem Agenten und der Umgebung, sowie das Lernen aus Erfahrung erfordert. Die verwendete Zustandsrepräsentation für die Reinforcement Learning Aufgabe kodiert den Zustand des Roboters in einer einzigen Variable (der Phase des ersten Oszillators). Die Arbeit beginnt mit der Theorie von Programmable Central Pattern Generatoren und der Einführung eines neuen Kopplungschemas, welches den bisher publizierten Ansatz verbessert. Der zweite Teil beinhaltet eine Einführung in Reinforcement Learning Methoden sowie die Beschreibung der Algorithmen welche zur Entwicklung der Feedback Policy verwendet wurden. Die Arbeit schließt mit einer Darstellung der Lernaufgabe, sowie der Präsentation der Ergebnisse. Die gelernte Policy stabilisiert den Roboter während des Gangs für bis zu durchschnittlich 70 Sekunden, was in etwa 42 kompletten Schritten (links und rechts) entspricht.

Stichwörter: Humanoide Robotik, Programmable Central Pattern Generator, Reinforcement Learning, Stabilitätskontrolle, Roboterkontrolle, Gangart

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Florian Hackenberger
Graz, Oktober 2007

Acknowledgements

First, I would like to thank my adviser Prof. Wolfgang Maass for his guidance and for providing me with ideas and suggestions that have made this work possible. Furthermore, I wish to thank DI Gerhard Neuman for his dedicated support throughout this thesis. In addition, my thanks go out to all other members of the robotics group at the Institute for Theoretical Computer Science (IGI) for including me in their scientific and non-scientific discussions. Last but not least, I would like to thank Ludovic Righetti from the EPFL for his inspiring preceding work and for providing me with the source code written for his publications.

Contents

List of Figures

1 Introduction

Robotics has been an active research area for decades. Since humans are the ultimate benchmark for researchers in this area, it seems obvious to mimic their behaviour. But why bother with the difficulties of a legged robot, if wheeled robots are available and do not pose as many problems as a machine with legs does? Not only because researchers enjoy challenging tasks, but also because a legged robot has significant advantages over a machine with wheels. It is able to move on uneven ground, step over obstacles and generally has the advantage that feet provide a lot more friction with the surface compared to a wheel. Furthermore a wheeled robot is in general limited to horizontal movements, while a machine with legs can climb a ladder or use steps to travel up- or downwards. Another advantage is that existing infrastructure can be harnessed by a humanoid robot.

While the preceding arguments justify the choice of a legged robot, choosing a humanoid instead of a machine modelled on an animal with more than two legs entails many complications. The choice of a humanoid robot for this thesis was influenced by the availability of a Hoap-2 robot at a partner university, the availability of and experience with a model of this robot on the Webots platform and of course by the challenge and excitement of trying to teach a humanoid robot how to maintain balance while walking.

Creating a controller producing a robust walking gait for a humanoid robot is a very tough problem and balancing the robot during the walk is essential for longer distances, according to [Mcgeer 2007]. This is obvious when looking at a challenge with a 20,000$ reward announced in 2006[1]. According to [Mcgeer 2007], there are over 170 participants and no winner yet (as of September 2007). The challenge requires a robot to walk over a distance of 10km, while meeting an energy consumption constraint and avoiding several obstacles. The course is easily completed by a human. The obstacles are a size limiting arch at the beginning, twenty unevenly spaced stepping stones with a minimum height of 39cm, three panels with a minimum height of 5cm in a row (a judge may remove the middle panel at will) and a staircase which has to be climbed and descended.

This thesis isolates a sub-problem from the broad area of robust walking. It tries to solve the task of actively balancing a humanoid during a walking gait. The main hypothesis of this work is that a nonlinear feedback policy on the lateral hip and ankle joints, with the lateral tilt and velocity of the upper body and the phase

[1] `http://www.wprize.org/`

of a periodic walking trajectory as parameters, can improve the robustness of the walking gait significantly compared to a linear policy based on the same variables. The main findings of this thesis are empiric results supporting the hypothesis in the context of a simulation of a particular humanoid robot, the improvement of an existing learning framework for a Programmable Central Pattern Generator (PCPG) system and the successful application of two Reinforcement Learning (RL) algorithms to balancing a humanoid during a walking gait.

The most important improvement of the PCPG system is a new feedback for maintaining a certain phase offset between the individual oscillators. This approach simplifies the dynamics of the system and makes it more robust towards external perturbations. The linear feedback path for balancing the walking gait of the Hoap-2 robot proposed in [Righetti and Ijspeert 2006] is replaced by a RL system in this work. The advantages over the linear feedback, apart from being nonlinear, is that while the linear system is a reactive policy, the RL system can act anticipatory. This is made possible by the inclusion of variables such as the phase of the walking pattern as well as the linear lateral velocity of the upper body into the state space of the system. The system could, for example, deduce in advance that the lateral tilt is going to exceed a critical limit later if the angular velocity exceeds certain thresholds.

1.1 Related Research

There are numerous publications on designing controllers for humanoid robot gait. Among the earliest works in this area are [Kun and Miller 1996] and [Taga 1997], but the roots of research on controllers for legged robot date back even further. [Raibert 1986] is an early book introducing the reader to legged robots, mostly in the form of single leg hopping machines and quadrupeds. Several researchers explore the possibilities of Central Pattern Generator (CPG)s in this area. Some contributions use RL[2], but very few combine the two approaches. To the best of my knowledge this thesis is the first work to explore the approach of using a RL method in order to obtain a feedback policy for actively balancing a humanoid robot gait generated by a CPG system.

The foundation of this thesis is the idea of using a CPG based controller, as published in [Righetti and Ijspeert 2006], to generate a walking gait for a humanoid robot. They place their controller into a feedback loop in order to

[2]While it certainly is not an accurate measurement, searching for 'humanoid AND robotics' on IEEEXplore, Springerlink and ACM Digital Library and comparing the results to the response from 'humanoid AND robotics AND reinforcement AND learning' leads to the conclusion that 6.98% of the humanoid robotics publications are related to reinforcement learning. Results as of Oct. 1, 2007.

keep the robot upright during the walk using a linear feedback policy relative to the torso tilt acting in lateral[3] and sagittal[4] direction. In addition they reset the phase of the oscillators whenever the right leg touches the ground, which creates entrainment of the controller with the body dynamics of the robot according to them.

[Ogino et al. 2004] propose a two layer controller, consisting of a CPG based system which controls the actual movements as the lower layer and a RL learning system which chooses the parameters of the lower layer appropriately to create a walking trajectory towards a goal. The generated trajectory seems to be statically stable. Their work is in the context of the Robocup humanoid league challenge. One of the tasks in this challenge is to approach a football preparing for a kick towards a goal. Their Hoap-1 robot is equipped with a fish-eye-lens camera for tracking the position of the ball and the goal. The lower layer of their controller consists of two modules: a *trajectory controller* and a *phase controller*. The trajectory controller calculates the output sent to the robot using equations based on the sinus function and elliptic foot trajectories (using inverse kinematics for calculating the actual joint positions). The phase controller is comprised of two oscillators, one for each leg, which provide the phase to the trajectory controller. A feedback term ensures that the phase of the oscillator has a predefined value just as the foot touches the ground. The upper layer system consists of a Dynamic Programming based RL system mapping a state consisting of visual information and robot posture to actions setting the step direction and -height variables of the trajectory controller. The visual information provided to the learning system consists of the direction of and distance to a football as well as the direction of the goal. Their result is a controller which is able to position the Hoap-1 robot in front of the football, facing the goal reliably.

The work published in [Matsubara et al. 2006] is a controller capable of optimising a walking trajectory created by CPGs based on neural networks with a policy gradient RL method. The walking trajectory steers the movements of a 5-link biped robot, which is restricted to the sagittal plane by a support. Their controller generates a stable, and even robust walking trajectory for the under-actuated (no ankles) robot. However, due to the restricted number of degrees of freedom of the feet, active balance control is not included in their controller.

[Stilman et al. 2005] explore the applicability of Dynamic Programming RL methods to biped locomotion control. Their controller steers a planar 5-link walking machine using a policy generated by a grid-based Dynamic Programming algorithm. In order to make DP techniques applicable to such a high dimensional

[3] Latin *lateralis;* 'to the side'

[4] According to the *Anatomical terms of location* article on [Collaborative authorship 2007], the sagittal plane divides the body into sinister and dexter (left and right) portions.

control problem, they use a reduced dimensional state-action space by manually selecting the important dimensions. Reducing the number of dimensions is enabled by dividing the walking gait into a single- and double support phase. They assume a no-slip model for the double-, and a compass 2-link model for the single-support phase. Additionally using action discretisation, they managed to reduce the state-action space to 24,810,660 cells. The most impressive result of their work is the fact that their policy for walking is robust against an increase of 200% in thigh mass, among other disturbances like an up to 6% incline.

Choosing a feedback approach to balance control for this thesis was interesting and challenging as there have been great results with optimising trajectories but few publications focusing on feedback. Combining PCPGs with RL for solving the task was appealing because it was quite novel at the time of writing.

2 Programmable Central Pattern Generator Theory

This chapter will introduce the reader to the concept of a PCPG. The term PCPG first appeared in [Righetti and Ijspeert 2006], while the concept was previously referred to under the name Adaptive Central Pattern Generator in [Righetti, Buchli, and Ijspeert 2005]. According to [Ijspeert and Kodjabachian 1999], a CPG is a distributed biological neural network which can produce coordinated rhythmic signals without oscillating input from the brain or from sensory feedback. As an example, they mention that a cat exhibits a walking gait as soon as a simple signal is sent to its brain stem. By changing the amplitude of the signal, the cat's movements can be changed from a trotting- to a walking- and even a running gait. In this example the nervous cell in the cats brain stem and spinal chord are the neural network forming the CPG. The nervous signals required to make the cats leg joints produce the trajectories of the gait are the output of the neural network. According to [Righetti and Ijspeert 2006] a PCPG is a system of coupled nonlinear dynamic oscillators, which is able to encode a given rhythmic input signal into the limit cycles of the oscillators. While theoretically many different types of oscillators could be used in a PCPG, a modified version of the original Hopf oscillator has been chosen for simplicity. The modified oscillator is able to adapt its frequency and amplitude in response to a target signal. Using oscillators is interesting because they have the ability to synchronise with external driving signals, a property which can be useful in many ways. In this work, a coupling scheme is used to coordinate the movement of several joints of a robot. See section 2.3 for more details.

The system introduced in [Righetti and Ijspeert 2006] is defined in Cartesian coordinates which may be harder to grasp compared to an oscillatory system in polar coordinates and has several drawbacks regarding learning speed and coupling between individual oscillators, as well as between individual PCPGs. The system described in the following sections solves some of the problems. It was developed as a result of the research conducted for this thesis and solves a few difficulties of the original approach concerning the coupling of individual oscillators. The new system improves the results for the application to humanoid robotics.

2.1 Programming a Hopf Oscillator

The following sections will introduce the reader to the concept of *dynamic Hebbian learning* in adaptive frequency oscillators as described in [Righetti et al. 2005]. The authors developed an extended version of the Hopf oscillator which is able to adapt its frequency to the frequency of a driving signal and embeds the learning process into the dynamics of the oscillator.

2.1.1 The dynamics of a simple Hopf oscillator

The simple Hopf oscillator in polar coordinates is defined by the following differential equations[1]:

$$\begin{aligned} \dot{r} &= \gamma(\mu - r^2)r \\ \dot{\phi} &= \omega \end{aligned} \tag{2.1}$$

Where $r(t)$ is the radius, $\phi(t)$ is the phase of the two dimensional output of the system at time t (in seconds) and γ defines the *strength* of the attracting limit cycle i. e. how fast the oscillator returns to the limit cycle after a perturbation. The oscillator has a stable limit cycle defined by the constant value μ. It can be easily seen that $\sqrt{\mu}$ corresponds to the radius of the limit cycle relative to the coordinate frame of the system. ω defines the frequency of the oscillation in units of $\text{Hz} \cdot 2\pi$. The output of r and ϕ during several sample runs with varying initial conditions can be seen in Figure 2.1.

A Hopf oscillator has the nice property to forget perturbations acting on r after a while. This is due to the limit cycle being an attractor acting against the perturbation. Perturbations on the phase ϕ, however, are remembered indefinitely. According to [Righetti, Buchli, and Ijspeert 2006], the phase is marginally stable, whereas the system is damped perpendicularly to the limit cycle. This property can be easily recognised by looking at the system equations. If r is suddenly changed while the system is running (assuming that the perturbation vanishes immediately and r differs from $\sqrt{\mu}$ after the perturbation), the absolute value of the term $\mu - r^2$ increases and drives r towards $\sqrt{\mu}$. If, however, ϕ is perturbed by an external force the system does not correct the change. This is easy to see because $\dot{\phi}$ does not depend on ϕ, hence it cannot even detect the change.

2.1.2 An extended, frequency adaptive Hopf oscillator

The following sections will explain a modification to the simple Hopf oscillator introduced in section 2.1.1 which will enable the oscillator to learn a simple (one

[1] The widely known dot notation $\dot{f}$ refers to $\frac{df}{dt}$

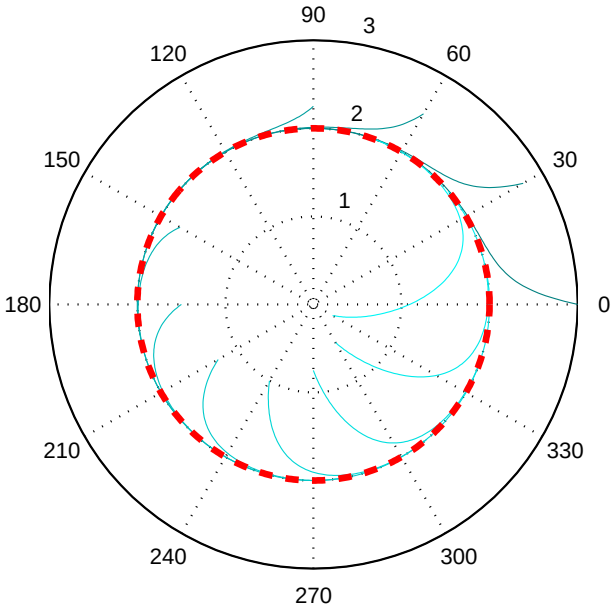


Figure 2.1: The limit cycle (red, dashed) of a simple Hopf oscillator with $\mu = 4; \omega = 0.2 \cdot 2\pi$ and a few evolutions of the system with varying initial conditions in a phase plot (ϕ plotted against r).

frequency component) periodic input signal. In order to be able to understand how the adaptation mechanism works, some synchronisation theory is required. The basics of this theory are introduced as well.

Synchronisation basics

The following section will explain some basic principles of synchronisation required to understand the frequency adaptive Hopf oscillator, based on [Pikovsky, Rosenblum, and Kurths 2001]. According to [Pikovsky et al. 2001], synchronous means *sharing the common time* or *occurring in the same time* and the related term *synchronisation* refers to a variety of phenomena found in almost all branches of natural sciences, engineering and social life, obeying universal laws. They define the adjustment of rhythms due to an interaction as the essence of synchronisation. As they show later on, even if the interaction is very weak synchronisation can nevertheless occur.

This section assumes that there are two oscillators present in the system. One oscillator (the forced or perturbed oscillator) receives an input from the other oscillator. This input is proportional to the output of the other oscillator and is applied to the phase as well as the amplitude, but just in either the x or y

direction, when thinking in Cartesian coordinates. In the examples below we consider the case where the force acts in the x direction. Therefore the force is multiplied by $\sin(\phi)$ when being applied to the phase ϕ and is multiplied by $\cos(\phi)$ when being applied to the radius r. The forced oscillator (the oscillator the force acts upon) as well as the oscillator producing the force are assumed to be quasilinear[2].

In order to understand how an external force applied to the phase and the amplitude acts on an autonomous oscillator, it is best to think in terms of a rotating coordinate system. The coordinate system is polar and its origin is equal to the origin of the coordinate system of the external force. The new coordinate frame rotates about its centre with frequency ω_e, the frequency of the force. For reasons of simplicity we assume that both $\omega_e > 0$ and $\omega > 0$ holds. Now imagine how the output of the autonomous oscillator would be represented within this new coordinate system if the force had zero amplitude and the autonomous oscillator would run with frequency ω, where $\omega < \omega_e$. It would be a point rotating clockwise with frequency $\omega_e - \omega$. Similarly if $\omega > \omega_e$ the point would rotate counter-clockwise with frequency $\omega - \omega_e$. In case of $\omega = \omega_e$, the point would not move at all.

The average external force acting on the autonomous oscillator can be represented as a constant force vector acting on the moving point (see Figure 2.2). Assuming that the way in which the coupling is implemented does not change over time, the direction and length of this vector is always the same, it just depends on the initial phase of the force and on the nature of the force (how exactly the coupling acts on the oscillator). In this case the force is assumed to be $\cos(t \cdot \omega_e + \bar{\phi}_e)$, where $\bar{\phi}_e$ is the initial phase of the force. Please note that the vector in Figure 2.2 represents the direction of the force on *average*. In fact the force at each point consists of a constant component and a rotating vector. This is depicted by the ellipsis around the tip of one of the force vectors in Figure 2.2.

In order to make it easier to understand why the average force can be represented by a vector with a fixed direction let us consider the following example: The force is defined by $F(t) = \cos(t \cdot \omega_e + \bar{\phi}_e)$ and is applied to the phase by subtracting $\epsilon F \sin(\phi)$ from $\dot{\phi}$, where ϵ is a small value < 1. It is also applied to the amplitude velocity by adding $\epsilon F \cos(\phi)$. We focus on the component of the force acting on the phase, since it is not forgotten over time as opposed to the component acting on the radius (see section 2.1.1). The component acting on the radius of the system therefore does not have any influence on synchronisation.

[2] An oscillator is called quasilinear (or quasi-harmonic), if the oscillation it exhibits is similar to a sine wave. When looking at the oscillation in a phase plot the limit cycle would be close to a circle.

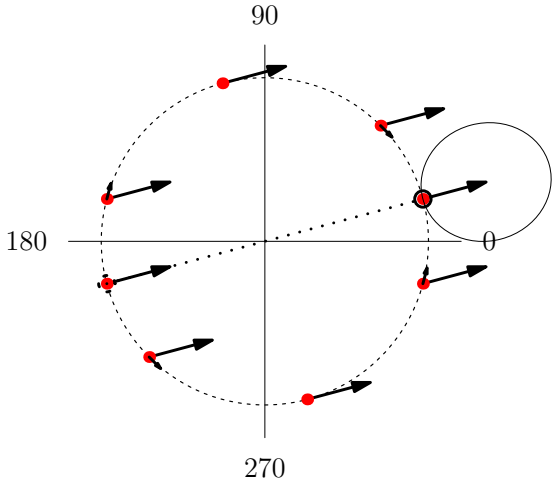


Figure 2.2: The phase point of the autonomous oscillator in a polar reference frame rotating with ω_e. The phase point has a stable (marked with a circle) as well as an unstable equilibrium point (marked with a dashed circle). The big arrows denote the average direction of the external force, whereas the smaller arrows indicate the direction on the limit cycle in which the phase point is dragged by the force.

The system equations become:

$$\begin{aligned} \dot{r} &= \gamma(\mu - r^2)r + \epsilon F \cos(\phi) \\ \dot{\phi} &= \omega - \epsilon F \sin(\phi). \end{aligned}$$

The two systems run with the same frequency, hence $\omega = \omega_e$ and ϕ as well as ϕ_e are running counter clockwise. The initial phase of the autonomous oscillator is $\phi_0 = \frac{\pi}{2}$ and the initial phase of the force is $\bar{\phi}_e = 0$. In order to simplify the notation, let us define $\phi_e = t \cdot \omega_e + \bar{\phi}_e$. Let us now check which conditions have to be satisfied in order to increase or decrease the phase velocity. In order to decrease the velocity the following condition has to be satisfied (note the '$-$' in $\dot{\phi} = \omega - \epsilon F \sin(\phi)$):

$$\cos(\phi_e) \cdot \sin(\phi) \quad > \quad 0$$

which expands to:

$$(\cos(\phi_e) > 0 \text{ AND } \sin(\phi) > 0) \text{ OR } (\cos(\phi_e) < 0 \text{ AND } \sin(\phi) < 0)$$

and further to:

$$(\phi_e = \left(\frac{3\pi}{2}, \frac{\pi}{2}\right) \text{ AND } \phi = (0, \pi)) \text{ OR } (\phi_e = \left(\frac{\pi}{2}, \frac{3\pi}{2}\right) \text{ AND } \phi = (\pi, 2\pi)) \quad (2.2)$$

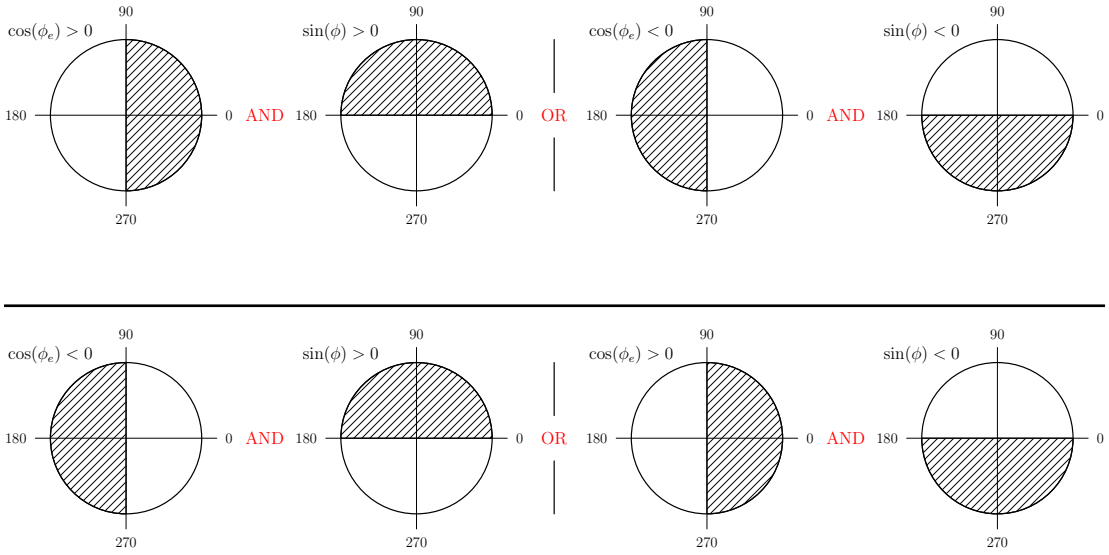


Figure 2.3: The upper part of the figure shows the conditions to be satisfied by the systems in order to decrease the phase velocity. The lower part shows the conditions for increasing the velocity.

We can proceed analogous for increasing the velocity which leads to:

$$(\cos(\phi_e) < 0 \text{ AND } \sin(\phi) > 0) \text{ OR } (\cos(\phi_e) > 0 \text{ AND } \sin(\phi) < 0)$$

and further to:

$$(\phi_e = \left(\frac{\pi}{2}, \frac{3\pi}{2}\right) \text{ AND } \phi = (0, \pi)) \text{ OR } (\phi_e = \left(\frac{3\pi}{2}, \frac{\pi}{2}\right) \text{ AND } \phi = (\pi, 2\pi)). \quad (2.3)$$

Please refer to Figure 2.3 for a visual representation of these conditions. Let us go back to the example. The initial value of ϕ is $\phi_0 = \frac{\pi}{2}$ and the initial phase of the external force is set to $\bar{\phi}_e = 0$. In order for the two oscillations to be synchronised the phase of the driven oscillator would have to decrease by $\frac{\pi}{2}$. After the first integration step (assuming Δt and $\omega = \omega_e$ are small) the first part of equation 2.2 is satisfied by the current phase values ($\phi = \frac{\pi}{2} + \omega \cdot \Delta t \Rightarrow \phi = (0, \pi)$ and $\phi_e = \Delta t \cdot \omega_e \Rightarrow \phi_e = (\frac{3\pi}{2}, \frac{\pi}{2})$). Therefore the velocity of the oscillators phase is decreased by the force. The condition is satisfied until $\phi_e = \frac{\pi}{2}$. At this point ϕ satisfies $\phi < \pi$, because $\omega = \omega_e$ holds and $\dot{\phi}$ was decreased by the force up to this point (we assumed $\epsilon > 0$). One integration step later the situation changes. The phase states satisfy the first condition of equation 2.3. The result is that the phase velocity increases. However, assuming that ϵ is small it increases only for

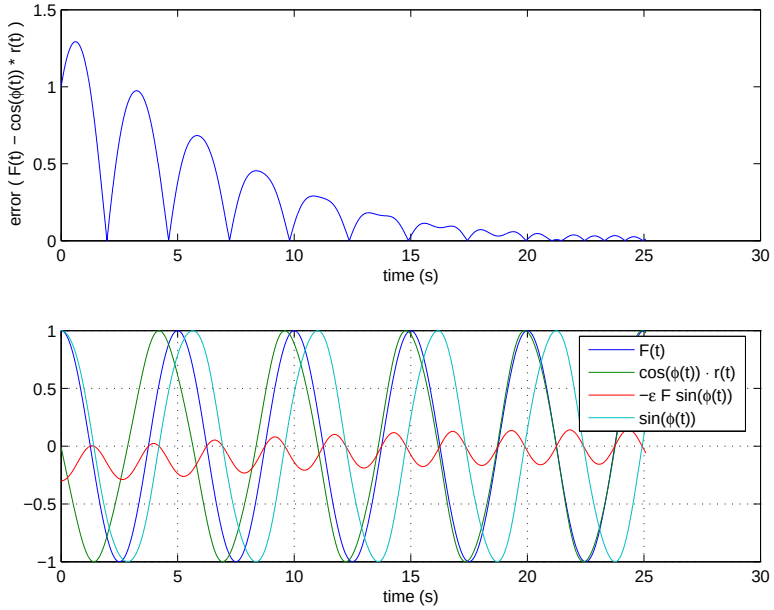


Figure 2.4: The evolution of a simple, perturbed Hopf oscillator. $\epsilon = 0.3$; $\mu = 1$; $\phi_o = \frac{\pi}{2}$; $\omega = \omega_e = 0.2 \cdot 2\pi$. The upper plot depicts the error between the force and the output of the oscillator. The lower plot shows the force $F(t)$, the output of the oscillator $\cos(\phi(t)) - r(t)$, the coupling $-\epsilon \cdot F \cdot \sin(\phi(t))$ acting on the phase velocity and $\sin(\phi(t))$.

a short amount of time, until $\phi > \pi$ is satisfied and therefore the second part of equation 2.2 holds. The next condition would be the second part in 2.3 for a short time and after that the initial condition would hold again. Therefore it decreases on average and in turn decreases the phase offset $\phi - \phi_e$. As the phase offset decreases, the ratio of the average amount subtracted and added to the phase during one revolution approaches 1. As soon as the phase offset is close to zero (in the general case of $\omega \neq \omega_e$ the offset would remain constant), the oscillators are phase locked. At this time the net feedback added to ϕ becomes zero. Please note that the phase offset oscillates indefinitely, while the amplitude of the oscillation depends on the strength of the force, which is proportional to ϵ in this example. See Figure 2.4 for an illustration of the evolution of the coupled oscillator.

After studying the case $\omega = \omega_e$, how do the systems behave if $\omega > \omega_e$? The case of $\omega \neq \omega_e$ is called detuning. If the coupling strength is zero the phase point will, instead of staying at a certain point, rotate either clockwise or anti-clockwise. As we have seen in the previous paragraph, the coupling is a force which rotates the phase point in the reference frame towards a specific position. Therefore the

frequency difference is nothing more than a force acting against the coupling force. Therefore, for small detuning the coupling strength out-weights the detuning force and the oscillators synchronise. Since the strength of the coupling force changes with the phase offset, it is easy to see that there is an equilibrium point where the counteracting forces vanish. This point defines the phase offset from the theoretic equilibrium point if $\omega = \omega_e$ would hold. If synchronisation occurs, the frequency of the forced oscillator is equivalent to the frequency of the oscillator exhibiting the force. It important to note that for large frequency difference, synchronisation does not occur at all. The region of ω_e where synchronisation is possible is called the Arnold region of the driven oscillator.

Adapting the frequency of a Hopf oscillator

This section is based on [Righetti et al. 2006]. The frequency adaptive Hopf oscillator has the ability to dynamically adapt its frequency to any periodic or even pseudo periodic[3] signals. The adaptive mechanism is called *dynamic Hebbian learning* because it is similar to correlation based learning, called *Hebbian learning*, observed in biological neural networks. The learning process is completely embedded in the system equations. The principle of the method (a perturbed oscillator, with its frequency state variable being coupled to the driving signal) can be applied to different kinds of oscillators, like relaxation oscillators and even strange attractors[4]. For the application of learning a humanoid walking gait, however, the Hopf oscillator is perfectly suitable. Therefore this work does not expand on applying *dynamic Hebbian learning* to other classes of oscillators.

In order to learn a given input signal, it is necessary to feed this signal into the system. The input signal (driving signal) is denoted F. The equation system 2.1 is then changed to:

$$\begin{aligned} \dot{r} &= \gamma(\mu - r^2)r + \epsilon F \cdot \cos(\phi) \\ \dot{\phi} &= \omega - \frac{\epsilon}{r} F \cdot \sin(\phi) \\ \dot{\omega} &= -\epsilon F \cdot \sin(\phi) \end{aligned} \tag{2.4}$$

The output of the system is defined as $G = r \cdot \cos(\phi)$. The term $\epsilon F \cdot \cos(\phi)$, added to $\dot{r}$ is just for completeness and does not play a role in the frequency adaptation process. The coupling to the phase ϕ is compensated for the perturbation on r (by

[3] In [Righetti et al. 2006] they choose the z variable of the Lorenz attractor as a chaotic, pseudo-periodic input signal. Please refer to the paper for details on learning such a signal.

[4] According to the *Attractor* article on [Collaborative authorship 2007], 'an attractor is informally described as strange if it has non-integer dimension or if the dynamics on it are chaotic'. Examples of strange attractors are the Hénon-, Rössler- and Lorenz attractors.

dividing the coupling by r) because it is undesirable for the radius to influence the coupling on ϕ. Hence the only difference between this system and the perturbed Hopf oscillator from section 2.1.2 is the additional equation $\dot{\omega} = -\epsilon F \cdot \sin(\phi)$. Assuming an input signal F with just a single frequency component, this equation changes ω towards ω_e. In order to understand why this scheme actually works (without going through a lengthy mathematical proof[5]), consider three cases while assuming an input signal of $F(t) = \cos(\phi_e)$:

1. The system is running with the same frequency as the input signal $\omega_e = \omega_0$ and the two systems are in-phase.
   This is the desired result of the learning process. In this case the term $\sin(\phi)$ is at its maximum, when F is at zero (at $\phi = \phi_e = \{\frac{\pi}{2}, \frac{3\pi}{2}\}$). In the interval $\phi = \phi_e = [\frac{3\pi}{2}, \frac{\pi}{2}]$ the term $\sin(\phi)$ is as much positive as it is negative and therefore $F \cdot \sin(\phi)$ is zero on average. Alternatively just consider $\int_0^{2\pi} \cos(\phi_e) \cdot \sin(\phi) = 0$ assuming $\omega_e = \omega$ and $\phi_e(t) = \phi(t)$. The same holds for the interval $\phi = [\frac{\pi}{2}, \frac{3\pi}{2}]$. The result is that the frequency as well as the phase offset between the input signal and the system output oscillate a little bit about their target values. Because of the oscillations about the target value of ω it is clear that ϵ should not be too large, as the amount of oscillation directly depends on it.

2. The system is running with a different frequency compared to the input signal $\omega_e \neq \omega_0$.
   As explained in section 2.1.2 and assuming $\epsilon = 0$, the phase point in the rotating reference frame would uniformly rotate either clockwise or counter-clockwise with frequency $\|\omega_e - \omega_0\|$. While the frequency adaptive oscillator does not depend on synchronisation itself, it does depend on a coupling between the oscillator and the external force. Assuming that $\epsilon > 0$ and the frequency ω_e is too different from ω_0 for synchronisation to kick in (hence ω_0 is not in the Arnold region of ω_e and ϵ), what would the effect of the external force be? It would change the evolution of ϕ. Splitting the limit circle in the rotating reference frame in Figure 2.2 in two halves along the dotted line, instead of rotating uniformly the phase point would spend more time on one side of the circle compared to the other. It would rotate slower at the side where the force acts against the rotation and faster at the other side. Considering the equation $\dot{\omega} = -\epsilon F \cdot \sin(\phi)$ in 2.4, the effect would be that the amount added to ω while the phase point is at one side of the circle is different from the amount added while being at the other side. Therefore, on average ω is decreased if $\omega_e < \omega_0$ and increased if $\omega_e > \omega_0$.

[5]The proof is available in [Righetti et al. 2006].

3. The system is running with a sightly different frequency compared to the input signal, but is synchronised to it.
   In this case, the phase point in the rotating reference frame will stay at a certain point within the reference frame, depending on the initial conditions and the strength of the coupling to the driving signal. At each integration step, the force acting on the frequency ω will change it a little bit towards ω_e until $\omega_e = \omega$ holds and the two systems are in phase, leading to the first case.

2.1.3 Making the amplitude adaptive

The system described in section 2.1.2 is able to adapt its frequency to one frequency component of a periodic input signal F. Our goal, however, requires the system to learn the frequency as well as the amplitude of F, assuming F consists of just one frequency component and is zero-centred. Righetti et al. [2005] described such a system in the Cartesian coordinate system. For this thesis we change their equations to match the previously introduced oscillator in the polar coordinate system.

The state variable α is added to the equation system, which becomes:

$$\begin{aligned} \dot{r} &= \gamma(\mu - r^2)r + \epsilon F \cdot \cos(\phi) \\ \dot{\phi} &= \omega - \frac{\epsilon}{r} F \cdot \sin(\phi) \\ \dot{\omega} &= -\epsilon F \cdot \sin(\phi) \\ \dot{\alpha} &= \eta F \cdot \cos(\phi) \cdot r \end{aligned} \tag{2.5}$$

The input signal is renamed to P_{teach} and F is redefined as $F(t) = P_{teach} - \alpha \cos(\phi)$. Therefore, F now acts as a kind of error signal, which decreases as the system starts to replicate the input P_{teach}. The output of the system is redefined as $G = \alpha r \cdot \cos(\phi)$, therefore α acts as an amplification term of the systems output. The learning rule for α is based on *Hebbian learning*. The value of α increases if F correlates with the output $r \cdot \cos(\phi)$ of the system, while η acts as a learning variable which prevents oscillation of α. Of course, η should be chosen to satisfy $0 < \eta < 1$, during the learning process.

2.2 Combining Programmable Hopf Oscillators

The adaptive Hopf oscillator, introduced in the previous section, is able to adapt its frequency and amplitude to a single frequency component of a driving signal. Reproducing an arbitrary periodic signal consisting of multiple frequency com-

ponents, however, requires a more sophisticated system. The system consists of coupled PCPGs and was introduced in [Righetti et al. 2005].

Before describing the system in detail it is necessary to establish the mathematical foundation in order to explain why the system actually works. The theory behind combining oscillators in order to represent an arbitrary periodic input signal is the widely known *Fourier decomposition*. According to [Bartsch 2001], any unique, piecewise monotone and continuous over the interval $[0, 2\pi]$, periodic Function $f(x)$ with a period of 2π can be expressed as a trigonometric series $s(x)$, the *Fourier series* defined as

$$f(x) = s(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} \left[a_k \cos(kx) + b_k \sin(kx)\right]. \tag{2.6}$$

The *Fourier coefficients* a_k and b_k are then defined as

$$\begin{aligned} a_k &= \frac{1}{\pi} \int_0^{2\pi} f(x) \cos(kx)\, dx; \quad k = [0, \infty] \\ b_k &= \frac{1}{\pi} \int_0^{2\pi} f(x) \sin(kx)\, dx; \quad k = [1, \infty]. \end{aligned}$$

The function $f(x)$ is decomposed into a sum of harmonic vibrations with discrete frequencies. Each element of the sum in equation 2.6 therefore represents one frequency component. Further, if the *Fourier series* is aborted after a finite number of members $k = n$, the result is equal to an approximation of f by a trigonometric polynomial of rank n. The rest of this section explains why *Fourier series* aborted after n members can be represented by a system of n amplitude adaptive oscillators.

The output of the modified Hopf oscillator defined in section 2.1.3 is $G = \alpha r \cdot \cos(\phi)$, which is very similar to $a_k \cos(kx)$. In equation 2.6, the term kx defines the frequency of each component. The frequency of one oscillator on the other hand is defined by ω relative to the time t which in turn is represented by x in the above equations. When looking at equation 2.5 it becomes clear that once the input signal has been learnt[6], the output of the oscillator becomes $\alpha r \cdot \cos(\omega t)$, since $F(t) = 0$ and therefore $\int \dot{\phi}\, dt = \phi = \omega t$ holds. The term $\frac{a_0}{2}$ is unnecessary if the requirement that the input function has to be zero-centred is fulfilled. That leaves $a_k \sin(kx)$ to be represented by an oscillator within a PCPG.

If it is possible to maintain a certain phase offset between two oscillators within a system, representing a sinus function with an oscillator based on a cosine function is trivial, because $\sin(\phi) = \cos(\frac{\pi}{2} - \phi)$ holds. As long as the input signal is present, the coupling term $\eta F \cdot \cos(\phi) \cdot r$ applied to $\dot{\omega}$ in combination with

[6]For simplicity it is assumed that the fit is perfect.

$\epsilon F \cdot \cos(\phi)$ added to $\dot{\phi}$ takes care that the oscillators retain the correct phase difference. If the input signal is switched off, however, the oscillators drift apart with respect to their phase offset. The solution is the introduction of a coupling term between the individual oscillators of a PCPG. Please note that the coupling scheme presented in the rest of this section is different compared to the work presented in [Righetti et al. 2005] and improves the published method as explained in section 2.3.3.

Apart from the phase offset $\phi_{i,\Delta}$, each component of the sum $f(t) = \alpha_0 r_0 \cdot \cos(\omega_0 t) + \sum_{i=1}^{N} [\alpha_i r_i \cdot \cos(\phi_{i,\Delta} + \omega_i t)]$ can be represented by an amplitude-adaptive Hopf oscillator as introduced in section 2.5. The phase offset can be kept by introducing a coupling scheme. In order to use the phase of the first oscillator as a coupling term for the subsequent systems, it is necessary to scale it to match the frequency of the driven oscillator. Therefore we introduce the term $R_i = \frac{\omega_i}{\omega_0}\phi_0$ for each oscillator but the first. The coupling term is then defined as $\tau \sin(R_i - \phi_{i,\Delta} - \phi_i)$ and added to $\dot{\phi}_i$. It is important to note that τ should be set to zero until the learning process is finished. Enabling the coupling during adaptation has a negative impact on the process and could lead to oscillations in the worst case. Assuming that all oscillators already converged to their target values of ω_i and α_i, the term $R_i - \phi_{i,\Delta}$ defines the target value for ϕ_i. The term $\tau \sin(R_i - \phi_{i,\Delta} - \phi_i)$ therefore speeds up the revolution of ϕ_i if the error is in the interval $(0, \pi)$ and decreases its speed for $(\pi, 2\pi)$. After a perturbation, the oscillator i therefore returns to a certain phase offset from the first oscillator within a time-frame depending on the value of τ. The remaining issue is how to obtain the required phase offset in the first place. This can easily be achieved by adding $\phi_{i,\Delta}$ as a state variable to the equation system of each oscillator and defining it as $\dot{\phi}_{i,\Delta} = \lambda \sin(R_i - \phi_{i,\Delta} - \phi_i)$. Assuming converged oscillators and $\tau = 0$, the term $R_i - \phi_i$ defines the target value for the phase offset. The state $\phi_{i,\Delta}$ will then converge to the required phase offset for the same reason as the coupling term works. The constant λ controls the speed of adaptation. Although a proof of the convergence properties of this scheme is beyond the scope of this thesis and has not yet been published else-were, the system works as expected in practise.

The following system is an extension of the equation system 2.5 with the changes

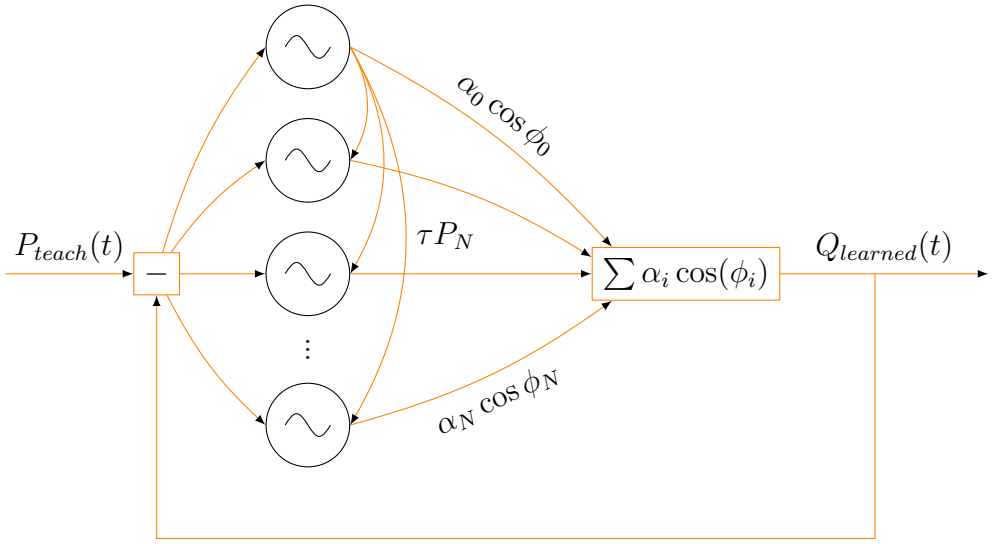


Figure 2.5: The structure of a network of coupled oscillators representing a PCPG. The term P_i is defined as $P_i = \sin(R_i - \phi_{i,\Delta} - \phi_i)$ for layout reasons. Adjusted from [Righetti et al. 2005].

discussed above added to it.

$$
\begin{aligned}
\dot{r}_i &= \gamma(\mu - r_i^2)r_i + \epsilon F \cdot \cos(\phi_i) \\
\dot{\phi}_0 &= \omega_i - \frac{\epsilon}{r_i} F \cdot \sin(\phi_i) \\
\dot{\phi}_i &= \omega_i - \frac{\epsilon}{r_i} F \cdot \sin(\phi_i) + [\tau \sin(R_i - \phi_{i,\Delta} - \phi_i)] \\
\dot{\omega}_i &= -\epsilon F \cdot \sin(\phi_i) \\
\dot{\alpha}_i &= \eta F \cdot \cos(\phi_i) \cdot r_i \\
\dot{\phi}_{i,\Delta} &= \lambda \sin(R_i - \phi_{i,\Delta} - \phi_i);\ \forall i > 0 \\
R_i &= \frac{\omega_i}{\omega_0} \cdot \phi_0
\end{aligned}
\tag{2.7}
$$

The equation system represents one oscillator in a network of N coupled systems as illustrated in Figure 2.5. The output of the network is defined as $Q_{learned} = \sum_{i=0}^{N} G_i = \sum_{i=0}^{N} \alpha_i r_i \cdot \cos(\phi_i)$. The error signal $F(t)$ is redefined as $F(t) = P_{teach} - \sum_{i=0}^{N} \alpha_i r_i \cdot \cos(\phi_i)$ in order to take the output of all oscillators into account. The variable γ controls how strong the attraction of the limit circle is. The higher γ is, the faster the oscillators recover from perturbations on the radius.

2.3 Further aspects of the PCPG system

The preceding sections have introduced the basic concepts and inner workings of PCPGs. This section discusses details related to applying the general approach to humanoid robotics, in particular to learning a walking gait for a humanoid robot. As the previously outlined approach is a modified version of the original approach, published in [Righetti et al. 2005, Righetti and Ijspeert 2006, Righetti et al. 2006], the changes compared to the published method are discussed as well.

2.3.1 Incorporating Feedback into the System

In order to create a robot controller which is able to react to changes in the environment or compensate for dynamic effects, it is necessary to modulate the movements of the robot. When using PCPGs for control, the oscillators itself can be modulated in order to apply feedback to the system. This approach has been outlined in [Righetti and Ijspeert 2006]. They propose to apply a feedback term (in their case a linear feedback relative to the lateral/saggital tilt of the robot) directly to the radius of all oscillators within a PCPG. Translating their equations to the polar coordinate system used to describe the oscillators in the previous sections leads to

$$\dot{r} = \gamma(\mu - r^2)r + \epsilon F \cdot \cos(\phi) + g \cdot f$$

where g is the feedback gain and f is the actual feedback value. This approach, however, has one significant disadvantage. At each integration step, the feedback is added to the current radius. That means the feedback policy has to compensate for the integration of the feedback over time if it wants to add a specific amount to the radius. In order to simplify the feedback task for the RL controller[7], the original feedback pathway was changed in order to represent a percentage added to the radius of the system. The following equation defines the modified feedback pathway:

$$\dot{r} = \gamma(\mu - r^2)r + \epsilon F \cdot \cos(\phi) + g \cdot [(1 + f) \cdot \sqrt{\mu} - r].$$

If the RL controller decides to add 30% to the radius of the system, it just sets $f = 0.3$ and the oscillator will smoothly adapt its oscillations to have a radius of $r = 1.3 \cdot \sqrt{\mu}$. The parameter g controls how fast the oscillator adapts to the given feedback. With the RL controller this schemes works better compared to the original approach, but one problem remains. Since the output of the oscillator is defined as $G = \alpha r \cdot \cos(\phi)$, both feedback paths do not have any effect if either $\phi = \frac{\phi}{2}$ or $\phi = \frac{3\pi}{2}$ holds. In the context of balancing the Hoap-2 is problematic, because it means that the feedback is sometimes ineffective and the

[7] The controller is described in section 4.2

intensity in general depends on the current phase of the oscillators. Therefore, another feedback pathway was devised which avoids this problem altogether. The feedback in this case simply represents an absolute value which is added to the output of one PCPG before sending the result to the robot. In this case f simply represents an angular value in radiant and is added to the output, which becomes $Q_{learned} = f + \sum_{i=0}^{N} \alpha_i r_i \cdot \cos(\phi_i)$. Unfortunately this scheme does not have the advantage of tight integration with the oscillators. Therefore the controller has to take care to produce smooth movements.

2.3.2 Combining several PCPGs

The system defined in section 2.2 is able to learn a single periodic input signal. The system required for the robot, however, has to reproduce ten input signals. Just using an array of isolated PCPGs would not work for the same reasons as combining several Hopf oscillators without feedback does not work. The signal learnt by the oscillators is always an approximation of the original trajectory and without feedback the phase offset between the individual oscillators or individual PCPGs would drift apart over time. Fortunately, the same scheme used for the oscillators can also be used for maintaining a phase difference between PCPGs. Although many different system layouts can be used, the PCPGs driving the Hoap are arranged in two chains. There is one chain for each leg and the first oscillator of each chain is anti-phase coupled to the first oscillator of the other chain. Each oscillator in these chains is coupled to its predecessor. Without a coupling between PCPGs the first oscillator of each network neither has a state variable $\phi_{i,\Delta}$ nor the coupling term $\tau \sin(R_i - \phi_i - \phi_{i,\Delta})$ added to $\dot{\phi}_i$, according to equation system 2.7. The inter-PCPG coupling system adds both terms to the first oscillator. But instead of defining R_i according to equation 2.7, it is defined as $R_i = \frac{\omega_i}{\omega_0^k} \cdot \phi_0^k$, where k denotes the index of the PCPG the system is coupled to. The coupling strength τ is change to τ_{ext} in order to be able to adjust it independently from the coupling within the PCPGs. Figure 4.2 illustrates the structure of the coupling system used for the Hoap-2 controller.

2.3.3 Differences compared to the original approach

The following section will sum up the changes of the proposed approach compared to the approach as published in [Righetti et al. 2005] and [Righetti and Ijspeert 2006].

The most obvious change is that while the original system is defined in the XY coordinate space, the modified equations are in the polar coordinate system for easier analysis. This brings them closer to the equation system of the frequency-adaptive oscillator proposed in [Righetti et al. 2006]. Additionally, it makes the

design of feedback easier because it can be directly applied to either the phase ϕ (where it does not vanish) or the radius r (where it decays over time), without splitting it in order to apply it to x and y which represent the state of the oscillator written in Cartesian coordinates. The simple amplitude- and frequency adaptive systems are mathematically equivalent as shown in the Appendix.

One of the problems with the system as published in [Righetti et al. 2005, Righetti and Ijspeert 2006] is the coupling scheme. The following discussion refers to the system defined by the following equations:

$$\begin{aligned}
\dot{x}_i &= (\mu - r_i^2)x_i - \omega_i y_i + \epsilon F + \tau \sin(R_i - \phi_{i,\Delta}) \\
\dot{y}_i &= (\mu - r^2)y_i + \omega x_i \\
\dot{\omega}_i &= -\epsilon F \cdot \frac{y_i}{r_i} \\
\dot{\alpha}_i &= \eta F \cdot x_i \\
\dot{\phi}_{i,\Delta} &= \sin(R_i - \phi_i - \phi_{i,\Delta}) \\
R_i &= \frac{\omega_i}{\omega_0} \cdot \phi_0 \\
\phi_i &= \operatorname{sgn}(x_i) \arccos(-\frac{y_i}{r_i}) \\
r_i &= \sqrt{x_i^2 + y_i^2}.
\end{aligned} \tag{2.8}$$

The system is equivalent to the one published in [Righetti et al. 2005] as the equations stated in [Righetti and Ijspeert 2006] are incorrect as confirmed by one of the authors. Some variables were renamed in order to match the terminology previously used in this thesis. While the system works fine in general, the coupling scheme has room for improvement. The state variable $\phi_{i,\Delta}$ fits the purpose and works as expected. The term $\tau \sin(R_i - \phi_{i,\Delta})$, however, while being similar to the Kuramoto coupling scheme[8] is not a good choice for this task. The problem is that it is modelled on a synchronisation scheme, while its purpose is just to keep a certain phase relationship between two oscillators. The term $\tau \sin(R_i - \phi_{i,\Delta})$ produces a sinusoidal signal which entrains the driven oscillator. The scheme works in principle, but it recovers from perturbations quite slowly. To a certain extent this slowness could be compensated for by increasing the coupling strength. The side effect would be a distorted output, because the coupling is not strictly Kuramoto-like. A Kuramoto coupling would apply to the phase only, while the coupling proposed in [Righetti et al. 2005] acts on x, which changes the radius of the oscillator in addition to the phase. While it would be easy to change the

[8] According to [Pikovsky et al. 2001, p. 280], the Kuramoto scheme is defined as a coupling between N mutually coupled oscillators with different frequencies. The coupling term $\frac{\epsilon}{N} \sum_{j=1}^{N} \sin(\phi_j - \phi_k)$ is added to $\dot{\phi}_k$ of each oscillator k. The coupling scheme is discussed in further detail in [Pikovsky et al. 2001] and [Daniels 2005].

coupling to a strictly Kuramoto-like scheme, the approach defined in equation 2.7 works even better.

The term R_i in the coupling $\tau \sin(R_i - \phi_i - \phi_{i,\Delta})$ is the phase signal of the first oscillator scaled to match the frequency of the ith oscillator. $R_i - \phi_i$ therefore denotes the current phase difference between the first and the ith oscillator. Subtracting $\phi_{i,\Delta}$, the desired phase offset, from the current phase difference gives the phase offset error. The sin function makes sure that the phase of the ith oscillator is accelerated if the error is in the interval $(0, \pi)$ and slowed down if the error is within $(\pi, 2\pi)$. Without applying the sin function it could happen that the oscillator e. g. tries to gain a complete revolution while its phase offset is correct already.

Apart from an improved coupling and therefore faster recovery from perturbations, the new coupling scheme has another advantage. Learning the phase differences can be done while the coupling term is disabled, which is impossible with the original system. The original phase coupling is basically a random perturbation as long as $\phi_{i,\Delta}$ has not been learnt correctly and may slow down the learning process. This is even more evident when using several coupled PCPGs, because in addition to the perturbations caused by the coupling between individual oscillators, there are perturbances caused by the coupling between the PCPGs. With the new scheme the coupling strength between individual oscillators and between PCPGs is adjustable after learning the input signal, which is not the case with the original coupling. Adjusting the coupling may be desirable if a controller changes the trajectories on the fly and want to adjust how fast the network converges back to the encoded pattern. This is impossible with the original scheme, because adjusting the coupling strength would change the pattern being produced by the system. Even if an on-line adaption of the coupling strength is not needed, it is impossible to set the coupling strength above a certain threshold, because the coupling would perturb the oscillators so much that they would not be able to produce the patterns required for learning the input signal. Figure 2.6 shows the improved reaction to random external perturbations affecting both the phase and the radius of all, but the first oscillator.

As it can be seen in Figure 2.6, the error of the original system increases toward the end of the simulation. This is due to parameters of the system being suddenly frozen just before the first perturbation begins. In order to compensate for this effect, the learning variables ϵ, η and λ of the new system are gradually decreased over a certain amount of time by simply multiplying them with c^t, where c is a *cooling* constant close to, but smaller than one.

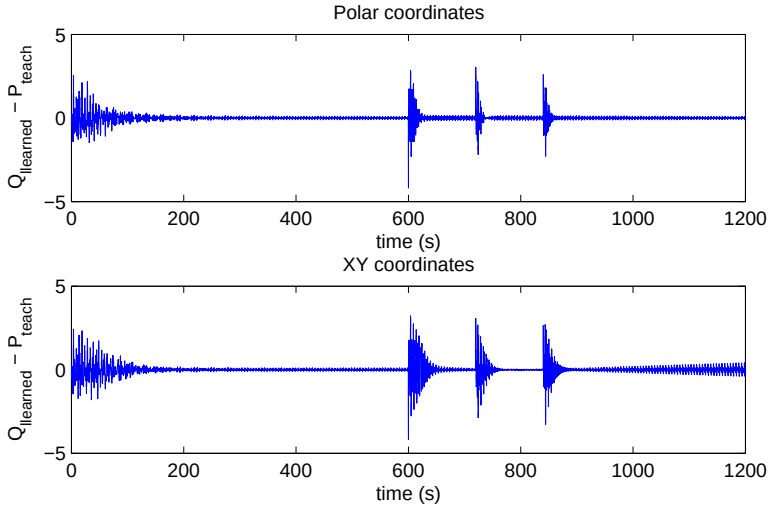


Figure 2.6: Error plot of a PCPG learning a periodic input signal and later during several perturbances. The upper plot is the version in polar coordinates with the improvements discussed in the text added to it, while the lower graph is the system as published in [Righetti et al. 2005]. The perturbations were applied to all but the first oscillator, in order to make plotting the error easier.

3 Improving Feedback Pathways using Reinforcement Learning

While having a PCPG representing a periodic function may be quite useful for robotics in general, it is not sufficient for the task of biped walking. The dynamics of a humanoid are complex and modelling them for one particular robot is already a huge task. Even if an accurate model is provided, manually designing a feedback policy for a PCPG which keeps the robot upright during walking seems quite tedious. Therefore, applying *Machine Learning (ML)* techniques in order to learn a nonlinear control policy is appealing. Among the numerous *ML* algorithms which are available, *RL* is a well suited approach for learning a feedback policy. RL has been applied in the field of robotics e. g. in [Morimoto et al. 2005], [Peters, Vijayakumar, and Schaal 2003] and [Ogino et al. 2004].

The following sections introduce the reader to the basics of RL in 3.1, outline the specific algorithms used for learning the feedback for the humanoid robot in sections 3.2.2 and 3.2.3 and finally describe the RL task for balancing the Hoap-2 during a walking gait in section 3.3.

3.1 A Brief Introduction to Reinforcement Learning

According to [Sutton and Barto 1998], RL means learning how to map situations to actions in order to maximise a numerical reward signal. It aims at enabling autonomous agents to learn how to behave in some appropriate fashion in some environment, from their interaction with this environment or from observations gathered from the environment, according to [Ernst, Geurts, and Wehenkel 2005]. It is an unsupervised learning approach and therefore it does not require *input–desired output* tuples. Reinforcement Learning does not define a learning method, but rather a learning problem. An RL setup consists of an agent and the environment the agent interacts with.

The agent interacts with the environment according to its current policy. In a particular state it chooses the next action from a set of available actions according to the policy. The policy can be expressed in terms of a mapping from state to action and completely defines the behaviour of the agent. The reward function is used to indirectly define the goal of the learning problem, since the agents sole objective is to maximise the reward it gets over time. The reward is the

only measure of success for the agent and its choice is therefore crucial for the quality of the learning result. This stems from the fact that all RL methods aim at maximising the return, which is the cumulated reward over time and defined as $\sum_{k=0}^{\infty} r_{t+k+1}$, starting at time t. In contrast to the reward function, the value function indicates what is good in the long term. According to [Sutton and Barto 1998], the value function is the total (discounted) reward an agent can expect to accumulate over the future, starting at a specific state. In other words, values are predictions of future reward. While the reward function is specified by the user, the value function is implicit and has to be learnt by the agent, although this is not compulsory. The environment model is an optional component which predicts the behaviour of the environment. It maps from a given state and an action to the next state and the given reward. According to [Sutton and Barto 1998] an environment model is an optional component within a Reinforcement Learning setup which is used for planning. Using a model, the agent can predict possible future situations before choosing a course of action. [Kaelbling and Moore 1996] depict two main strategies for solving a reinforcement learning problem:

- Searching in the behaviour space for a policy which performs well in the given environment (policy search algorithms)
- Using statistical- and dynamic programming[1] techniques, which estimate the return, to derive a good policy (value based algorithms)

Both strategies are widely used and can be applied to a variety of learning problems.

3.1.1 Elements of Reinforcement Learning Problems

The following section is based on [Sutton and Barto 1998] and will introduce the reader to the elements and important properties of a Reinforcement Learning problem in order to better understand the context of the algorithms used for learning the feedback to the CPGs.

The agent and the environment

In the context of Reinforcement Learning, the agent is the learner and the decision maker. It is responsible for inspecting its environment in terms of the *state*, for receiving the reward for the current state and for choosing the next action (see

[1] According to the *Dynamic Programming* article on [Collaborative authorship 2007], dynamic programming in mathematics and computer science, is a method of solving problems exhibiting the properties of overlapping sub-problems and optimal substructure that takes much less time than naive methods.

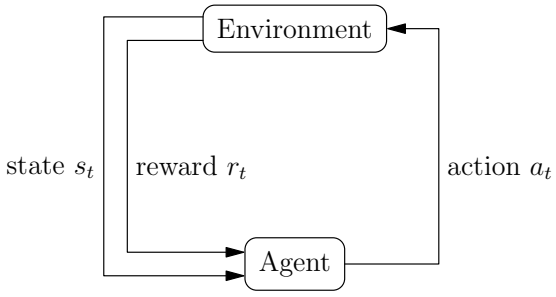


Figure 3.1: At each interaction step, the environment presents the state information $s_t$and the reward r_t for the current state to the agent. The agent chooses an action a_t, leading to state s_{t+1}.

Figure 3.1). The agent's choice of an action is the only interaction which influences the environment. An action can be abstract such as turning to the left, or concrete instructions such as the torque applied to a joint, depending on the problem at hand. However, all actions have to be part of an action set and some actions may not be available in certain states. Therefore the available actions are a function of the state and are defined as $a \in \mathcal{A}(s_t)$. The mapping from states to action probabilities is called the policy and is denoted as $\pi_t(s, a)$. It is defined as the probability of choosing action $a \in \mathcal{A}(s_t)$, when being in state $s = s_t$. The specific Reinforcement Learning algorithm chosen is responsible for evolving the policy towards the optimal policy over time.

The environment constitutes everything the agent cannot control, including the reward function. The agent may still have thorough knowledge of the dynamics of its environment. This is especially true for Reinforcement Learning tasks involving board games like chess. The agent may have a perfectly accurate model predicting the next state for a specific action, however, it does not have any control over the dynamics of the environment. The definition of the reward function is therefore part of the environment. It maps states $s \in \mathcal{S}$ (where $\mathcal{S}$ is the set of possible states) and optionally actions $a \in \mathcal{A}$ to numeric values $r \in \mathbb{R}$. In simple terms, the agent's goal is to maximise the cumulative reward it gets from the environment. Consequently the reward function is used to formalise the goal of the task. This provides a lot of flexibility for the definition of the task compared to supervised learning algorithms.

The Markov Decision Process

According to the *Markov property* article on [Collaborative authorship 2007], 'a stochastic process has the Markov property if the conditional probability distribution of future states of the process, given the present state and all past states,

depends only upon the present state and not on any past states, i.e. it is conditionally independent of the past states (the *path* of the process) given the present state'. The chess game, for example, satisfies the Markov property, because the future progression (and outcome) of the game only depends on the current board configuration and not on the history of the game. Processes exhibiting the Markov property are very convenient candidates for RL. Apart from a process, a state signal can have the Markov property as well. A Markov state completely retains all relevant information which influences the evolution of the process to the next state. It actually contains all information relevant to the whole future of the process, assuming all future state signals have the Markov property as well. If the agent perceives a state which has the Markov property, an optimal policy for the state is also an optimal policy for the complete history. Mathematically the Markov property for a state signal holds if the probability distribution

$$Pr\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\}$$

equals

$$Pr\left\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\right\}.$$

That is if the probability distribution of ending up in state s' at time $t+1$, after choosing action a_t at time t and knowing all state, action, reward tuples before that is equal to the probability distribution if only the previous state s_t and action a_t are known. It is obvious that for many environments it is impossible to pass a state signal exhibiting the Markov property to the agent, because either this signal would consist of too many dimensions or the process itself does not have the Markov property. Nevertheless, it is appropriate to think of the state signal as an approximation to a Markov state. Otherwise the RL method would have no chance finding a good policy.

If the state signal of a process has the Markov property, the process itself is Markovian too. A Markov decision process with a finite state and action space is called a finite Markov decision process. The probability of each possible next state s', given any current state s and action a, is defined by the transition probability function

$$\mathcal{P}^a_{ss'} = Pr\left\{s_{t+1} = s' | s_t = s, a_t = a\right\}.$$

The expected reward for any current state s, action a and successor state s' is then given by

$$\mathcal{R}^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\}.$$

3.2 Value based algorithms

The central element of value-based Reinforcement Learning algorithms is the estimation of a value function. According to [Sutton and Barto 1998], value functions

are functions of states or state-action pairs with respect to a particular policy. For Reinforcement Learning tasks satisfying the Markov property, the *state-value function* for following policy π is defined as

$$V^{\pi}(s) = E_{\pi}\left\{R_t \mid s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$$

where $E_{\pi}\{.\}$ denotes the expected return (cumulated rewards) with respect to policy π and R_t denotes the function calculating the return (in this case the infinite-horizon discounted reward). The value function is intended to quantify how good it is to follow a particular policy π starting in state s.

The value for taking action a in state s and following policy π thereafter is called the *action-value function* and is defined as:

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t \mid s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = a, a_t = a\right\}$$

The Q^{π} function can be used to quantify how good it would be to change the policy π, choosing a in state s and following the original policy thereafter. If $Q^{\pi}(s,a) > V^{\pi}(s)$ holds, it would be better to change the policy. Otherwise the current policy should be kept. Both equations are expressed in terms of infinite sums of future reward, which is quite inconvenient. Fortunately these equations satisfy recursive relationships.

The *Bellman equation* for V^{π} defines the relationship between the value of a state and the value of its successor state. [Sutton and Barto 1998] derive the Bellman equation for V^{π} as

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \\
&= E_{\pi}\left\{r_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right\} \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s\right\}\right] \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')\right] \qquad (3.1)
\end{aligned}$$

where $\mathcal{R}^a_{ss'}$ is the expected value of the next reward given any current state s, action a and successor state s', and $\mathcal{P}^a_{ss'}$ is the probability of each possible successor

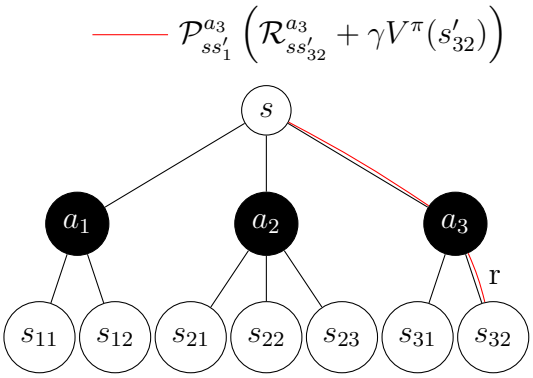


Figure 3.2: A state s and all its possible successor states s', resulting from the available actions $\mathcal{A}(s)$. Please note that the leafs may not be unique (e. g. two different actions may lead to the same successor state).

state s' given the current state s and action a, as previously stated. The equation 3.1 calculates the value of a state s by weighting each possible path to a successor state by its probability of occurring. Figure 3.2 shows one such path in red. The Q^π function can be written in terms of V^π and has a corresponding Bellman equation as well:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\left\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\right\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] = \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma \sum_a \pi(s',a)\cdot Q^\pi(s',a)\right]. \end{aligned}$$

Dynamic Programming

Dynamic Programming RL algorithms can be used to compute optimal policies, given a perfect model of the environment as a Markov Decision process. They combine *policy evaluation* and *policy improvement* algorithms to compute optimal policies. Policy evaluation computes the value V^π of a policy by iterating the Bellman equation which is guaranteed to converge to V^π. Policy improvement changes the current policy to a new greedy policy π' with respect to V^π by calculating

$$\pi'(s) = \arg\max_a Q^\pi(s,a) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right].$$

Policy iteration is a Dynamic Programming algorithm which alternately applies policy evaluation and policy improvement to yield an optimal policy. Value iteration is an extension of policy iteration. It just aborts policy evaluation after

one iteration, which is computationally cheaper compared to policy iteration, but can be proved to converge towards the optimal policy as well. Classical Dynamic Programming algorithms are of limited utility in practise, because of their restrictive requirement of an environment model and because of being computationally expensive. However, they are of theoretic importance, because many other RL methods are attempts to achieve the same effect, just more efficiently and without perfect environment models.

Temporal-Difference Learning

Temporal-Difference Learning estimates both the V^π and the Q^π function from experience. According to [Sutton and Barto 1998], 'Temporal Difference methods update estimates based in part on other learnt estimates, without waiting for a final outcome' and 'can learn directly from raw experience without a model of the environments dynamics'. The simplest TD(0) algorithm updates the value function as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right].$$

According to [Sutton and Barto 1998, p. 134], the TD(0) target is an estimate because it samples the expected return as well as using the estimated value $V(s_{t+1})$ instead of the true value $V^\pi(s_{t+1})$. TD learning uses experienced samples of the transition model instead of a predefined transition model as required by DP techniques. Usually one has to estimate the $Q(s, a)$ function. SARSA and Q-Learning are two well known algorithms based on such an estimate. As stated in [Sutton and Barto 1998], SARSA is an on-policy TD control method which estimates the action-value function according to

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right].$$

The name of the algorithm stems from the fact that the update rule uses the 4-tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. An on-policy method estimates Q^π for the current policy π and changes the policy π towards greediness with respect to the action-value function Q^π. On-policy methods therefore estimate the value of the policy while applying it. Q-Learning on the other hand, is a well known off-policy TD algorithm. The simplest variant is one-step Q-Learning, which updates the action-value function as defined in [Sutton and Barto 1998]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

The difference to SARSA learning, is that the Q function is updated independently from the current policy. One important aspect of TD algorithms is that they are guaranteed to converge to an optimal policy, if every state-action pair is visited infinitely often.

Eligibility traces

Temporal Difference methods, as already mentioned, update the value function based on value estimates of successor states. Monte Carlo methods on the other hand, update the function based on the future reward. Therefore a Monte Carlo method can update the value function only after a completed episode, while Temporal Difference methods can update the function after each step. Eligibility traces are a very important part of Reinforcement Learning, as they are used to bridge the gap between Temporal Difference- and Monte Carlo methods. They can be seen as a way of updating a value function towards an average of the returns of all succeeding states and not towards the return of only the next step as it is the case with ordinary Temporal Difference methods. As mentioned in the previous section, TD algorithms update their value function using r_{t+1}, the reward for the state following s_t after taking action a_t. Using eligibility traces, the update of the value function is performed using R_t^λ instead of r_{t+1}.

$$\begin{aligned} R_t^\lambda &= (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\cdot R_t^{(n)} \\ R_t^{(n)} &= \sum_{i=1}^{n}\left(\gamma^{i-1}r_{t+i}\right) + \gamma^n \cdot Q_t(s_{t+n}, a_{t+1}) \end{aligned}$$

[Sutton and Barto 1998] call $R_t^{(n)}$ the 'corrected n-step truncated return', because it is an n-step return corrected by the estimated value for all the truncated steps after the nth step. Since it is impossible to know the returns following the current state in advance, an alternative but equivalent method is used. In the case of action-value methods, for each state s and action a encountered during an episode, an eligibility trace $e_t(s,a)$ is introduced.

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \wedge a = a_t \\ \gamma\lambda e_{t-1}(s,a) & \text{else} \end{cases} \tag{3.2}$$

Hence the eligibility trace for a state s and action a is increased by 1 each time the state s is encountered and a is taken and is decayed by multiplication with γ, the discount rate, otherwise. Informally an eligibility trace keeps track of how recently a particular action in state s has been taken. They can also be thought of propagating TD information back in time. A proof of equivalence for the case of *state-value functions* can be found in [Sutton and Barto 1998].

The policy

A central element of the RL agent is its policy. The policy defines which action the agent chooses in a particular state, or for stochastic policies, how likely each

possible action is to be chosen. The most obvious policy is the *greedy* policy. In terms of the *action-value function* it is defined as $\pi(s) = \arg\max_a Q_t(s, a)$. The greedy policy means that the agent always chooses the action leading to the highest return with respect to its current action-value function. Obviously this policy is only useful if the Q function is accurate already. In terms of a trade-off between exploiting current knowledge and exploring in order to obtain new knowledge, the greedy policy is obviously the most exploitative policy imaginable. In order to successfully learn a good policy, balancing this trade-off is essential. Instead of always choosing the best action, one could choose a random action with a certain probability in order balance the trade-off a bit. The ϵ-greedy policy does just that. It chooses a random action with probability ϵ and is greedy with probability $1 - \epsilon$. While adjusting the exploitation/exploration balance is easy with the ϵ-greedy policy, it has one significant disadvantage. If not being greedy it chooses among the other available actions with equal probability, which can be harmful if the worst actions are very bad compared to the ones with higher Q function values. Policies following the *softmax* rule rank actions according to their value estimates. There are several ways to implement a softmax policy. As an example and according to [Sutton and Barto 1998] the softmax policy using a Gibbs distribution defines the probability of choosing action a as

$$\frac{\exp(\frac{Q_t(a)}{\tau})}{\sum_{b=1}^{n} \exp(\frac{Q_t(b)}{\tau})}$$

where τ is called the temperature. In the limit $\tau \to 0$ it approaches the greedy policy, while high values cause actions to approach equiprobability. Both ϵ-greedy, as well as softmax policies are suitable for adjusting the exploitation/exploration trade-off and the choice usually depends on the task at hand.

3.2.1 Function approximation

Instead of storing the value function in a table, function approximation — or more general, most supervised machine learning methods can be used to store the value function. The possible advantages are better generalisation and lower storage requirements. Most continuous tasks do not visit exactly the same state twice, although it would be convenient to treat different states as equivalent, because they are similar enough. Function approximation algorithms can provide this kind of generalisation. They enable a RL algorithm to generalise from previously experienced states to ones not experienced in the past. For continuous tasks using a table for storing the value function often becomes infeasible, because the algorithm would have to visit a huge amount of states and in some cases it would not work at all because one state may never be experienced again with exactly

the same values. A solution would be to discretise the state space, with the disadvantage that the partitioning, which has a high impact on the quality of the result, has to be done manually. A disadvantage of using function approximation with Temporal Difference methods is that the proof for convergence no longer holds.

There are numerous function approximation algorithms to choose from, both linear and nonlinear methods. Nonlinear methods are quite powerful, but often difficult to analyse. Feed-forward Neural Networks, for example, are according to [Neumann 2005, p.99] 'not used as often as linear approximators because ... they have a poor locality, learning can be trapped in local minima and after all we have very few convergence guarantees'. Linear methods on the other hand are simple to use and most of the time flexible enough. Linear methods like RBF-networks and tile codings are among the well known function approximation schemes used in RL and both are covered in [Sutton and Barto 1998].

SARSA(λ), one of the algorithms applied to learning the walking feedback is used with an RBF network as a function approximation scheme for the action-value function, therefore the rest of the section will concentrate on the linear case for the Q function. The $Q_t(s, a)$ function for each action a is represented as a parametrised function with the vector $\vec{\theta}_t$ as the only parameter. This is a generic description, allowing various function approximation schemes to be used. The vector $\vec{\theta}_t$ could be anything from the weights of several basis functions to the split points and leaf values of a decision tree. In section 3.2, it was explained how the various algorithms update the value function. The SARSA algorithm, for example, updates $Q(s_t, a_t)$ towards $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$. To be able to explain function approximation in the general case this value is denoted v. One of these updates would be presented to the function approximation algorithm as a sample mapping $s_t, a_t \rightarrow v_t$. A gradient descent algorithm would then update the feature action-value function according to

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \left[v_t - Q_t(s_t, a_t) \right] \nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \tag{3.3}$$

where $\nabla_{\vec{\theta}_t} f(\vec{\theta}_t)$ for any function f is the vector of partial derivatives,

$$\left(\frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(1)}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(2)}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_t(n)} \right)$$

and n is the number of entries in $\vec{\theta}_t$. Please note that the update rule above is specific for gradient descent methods and cannot be stated in general as it depends on the algorithm used.

Linear Function Approximation

A special case of gradient descent is linear function approximation with features. The Q_t function in this case is represented by a weighted sum over the response of n predefined features ϕ_i:

$$Q_t(s,a) = \vec{\theta}_t^T \cdot \vec{\phi}_t = \sum_{i=1}^{n} \theta_t(i)\phi_i(s,a).$$

where the feature function could be anything from a binary feature as used for coarse coding to a Radial Basis Function as used in section 3.2.2. In order to represent the changing Q_t function, the weights $\vec{\theta}_t$ are adjusted by the standard gradient descent update rule 3.3, where the gradient $\nabla_{\vec{\theta}_t} Q_t(s_t, a_t)$ in this case is simply $\vec{\phi}(s,a)$. .

3.2.2 SARSA(λ)-Learning with RBF centres

SARSA(λ) is an extension of the on-policy SARSA algorithm using eligibility traces, which is denoted by λ. The Q^π function of this algorithm is updated according to

$$\begin{aligned} Q_{t+1}(s,a) &= Q_t(s,a) + \alpha\delta_t e_t(s,a) \\ \delta_t &= r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \end{aligned}$$

where δ_t is the Temporal Difference error for action-value prediction. Please note that in the case of gradient function approximation algorithms, the eligibility traces 3.2 are changed towards the gradient of the Q function $\nabla_{\vec{\theta}_t} Q_t(s_t, a_t)$, instead of adding 1.

This thesis uses a linear function approximation with radial basis functions to represent the action-value function of the SARSA(λ) algorithm. The RBF definition, adapted from [Sutton and Barto 1998], is wrapped in order to make it depend on a specific action. This is useful for cases where there is just a small set of predefined actions available and it is therefore beneficial to have one separate function approximator per action.

$$\begin{aligned} \phi_i(s,a) &= \begin{cases} \Phi_i(s) & \text{if } a = A(i) \\ 0 & \text{else} \end{cases} \\ \Phi_i(s) &= \exp\left(-\frac{\| s - \mu_i \|^2}{2\sigma_i^2}\right) \end{aligned}$$

In this case σ and μ are single values in the one dimensional case and a covariance matrix and a vector in the multi-dimensional case respectively. In order to

represent the changing Q_t function, the weights $\vec{\theta}_t$ are adjusted by the standard gradient descent algorithm using eligibility traces. The update rule is therefore

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t + \alpha\delta_t\vec{e}_t \\ \delta_t &= r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \\ \vec{e}_t &= \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t). \end{aligned}$$

In the case of RBFs $\nabla_{\vec{\theta}_t} Q_t(s_t, a_t)$ for each entry i of $\vec{\theta}$ is simply the basis function $\phi_i(s, a)$. This algorithm has two major drawbacks. The first disadvantage is the curse of dimensionality, as the complexity of the algorithm depends exponentially on the number of state variables. Apart from computational complexity in higher dimensional tasks it has the inconvenience that the number of basis functions as well as σ and μ for each of them have to be manually chosen and affect the performance of the algorithm significantly.

Algorithm 1 Linear, gradient-descent SARSA(λ), an on-policy TD control algorithm adapted from [Sutton and Barto 1998, p. 212]. The algorithm implements an ϵ-greedy policy.

Initialise $\vec{\theta} = 0$
for each episode **do**
  Set $\vec{e} = \vec{0}$
  $s \leftarrow$ initial state of episode
  $a \leftarrow \pi(s, a)$
  for each step of episode **do**
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \Delta Q(s, a)$
    $Q_t \leftarrow \sum_i \theta(i) \cdot \phi_i(s, a)$
    Take action a, observe reward r and next state s'
    $a' \leftarrow \pi(s', a')$
    $Q_{t+1} \leftarrow \sum_i \theta(i) \cdot \phi_i(s', a')$
    $\delta \leftarrow r + \gamma Q_{t+1} - Q_t$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    $a \leftarrow a'$
  end for
end for

3.2.3 Extra-Tree-Based Batch mode Reinforcement Learning

Batch mode Reinforcement Learning [Ernst et al. 2005] learn a control policy from a set of four-tuples $(s_t, a_t, r_{t+1}, s_{t+1})$. This set of tuples is usually obtained from

recordings of an interaction between the agent and the environment. It simply represents a recording of all states visited along with the action taken and the reward received after reaching the next state. Compared to on-line RL methods it uses all available training data at once, as opposed to continuously updating an approximation of the value function with one tuple at a time. Being an offline learning algorithm, a batch mode reinforcement algorithm can harness supervised learning methods which cannot be used on-line. Some algorithms like artificial neural networks usually require several passes over a given training set to converge properly. These algorithms become available for RL methods in batch mode.

The rest of this section describes the Extra-tree based batch mode Reinforcement Learning algorithm as published in [Ernst et al. 2005]. The algorithm is used as an alternative approach to the SARSA(λ) with RBF centres method, described in section 3.2.2, for addressing the feedback task.

We use fitted Q-iteration as a form of batch mode Reinforcement Learning which enables the application of any regression algorithm, to the Reinforcement Learning task as stated in [Ernst et al. 2005]. This framework makes it possible to fit (using a set of four-tuples $(s_t, a_t, r_{t+1}, s_{t+1})$) any parametric or non-parametric approximation method to the Q-function. With each iteration the algorithm extends the horizon of the optimisation. After the Nth iteration, the approximation represents a Q_N function which corresponds to an N-step optimisation horizon. Consistent with [Ernst et al. 2005], just rewritten to match the previously used terminology, the Q_N function is defined as follows:

$$\begin{aligned} Q_0(s,a) &\equiv 0 \\ Q_N(s,a) &\leftarrow r_{t+1} + \gamma Q_{N-1}(s_{t+1}, \pi(s_{t+1})), \quad \forall N > 0. \end{aligned}$$

Q_N is proven to converge, but not necessarily to the optimal Q function, as stated in the paper. At each iteration, Q_N is calculated using any regression algorithm. For the experienced transitions the equations above match the DP equations. Therefore fitted Q-iteration is an approximate DP method.A pseudo code version for obtaining Q_N is printed in Algorithm 2. After stopping the algorithm (e. g. after a predefined number of iterations) an approximation of the optimal control policy can be derived by

$$\hat{\mu}_N^*(s) = \underset{a \in \mathcal{A}}{arg\,max}\, \hat{Q}_N(s,a).$$

A performance comparison of five different tree-based regression algorithms in the context of the batch-mode Reinforcement Learning framework is available in [Ernst et al. 2005]. The Extra-tree algorithm was chosen for this thesis simply because its implementation within the RL toolbox at the institute has already been used during research and the results from [Ernst et al. 2005] and [Geurts,

Algorithm 2 The fitted Q iteration algorithm, adapted from [Ernst et al. 2005]

Let $\mathcal{F}$ be a set of four-tuples $(s_t, a_t, r_{t+1}, s_{t+1})$
Set $N = 0$
Let $\hat{Q}_N$ be a function equal to zero everywhere on $\mathcal{S} \times \mathcal{A}$
repeat
$N \leftarrow N + 1$
Build the training set $\mathcal{T} = \{(i^l, o^l), l = 1, \ldots, \#\mathcal{F}\}$ based on $\hat{Q}_{N-1}$ and on the full set of four-tuples $\mathcal{F}$:

$$\begin{aligned} i^l &= (s_t^l, a_t^l) \\ o^l &= r_t^l + \gamma \underset{a \in \mathcal{A}}{max} \hat{Q}_{N-1}(s_{t+1}^l, a) \end{aligned}$$

Use the regression algorithm to induce the function $\hat{Q}_N(s, a)$ from $\mathcal{T}$
until stopping condition is met

Ernst, and Wehenkel 2006] looked promising.The algorithm published in [Geurts et al. 2006] can be used for classification, as well as regression tasks. In the case of batch mode Reinforcement Learning, only the regression version is useful.

According to [Geurts et al. 2006] the algorithm 'builds an ensemble of un-pruned decision or regression trees according to the classical top-down procedure'. They further note that it differentiates itself from other tree based ensemble methods, because it splits nodes by choosing cut-points independently of the target variable. An ensemble in this case consists of a certain number of trees which are chained together by averaging their output. According to the paper, the explicit randomisation of the cut-point and attribute combined with averaging the output over several trees should be able to reduce variance further than the randomisation schemes of other methods. The authors state that, assuming a balanced tree, the complexity of the tree growing procedure is $O(N \cdot \log N)$, where N is the size of the learning sample. Algorithm 3 shows the pseudo-code of the Extra-tree regression splitting algorithm. The algorithm has three tunable parameters. K denotes the number of attributes which are selected at random at each node, n_{min} specifies the minimum sample size to split a node and M sets the number of trees for averaging over samples.

The algorithm starts with M empty trees, each tree is then built recursively. The recursive tree-building algorithm adds a leaf to the tree if a stopping criterion is met or otherwise adds a node which splits the training set into a left and a right side. After splitting the algorithm recurses down both paths. The most important part of the algorithm is the splitting procedure. This procedure consist of two parts. The first part chooses K random splits, while the second part weights each

Algorithm 3 The Extra-tree regression splitting algorithm, adapted from [Geurts et al. 2006]

Build_extra_tree_ensemble(S)
Input: A training set S of tuples (i^l, o^l) as provided by fitted Q-iteration
Output: A tree ensemble $\mathcal{T} = \{t_1, \ldots, t_M\}$
for $i = 1 \ldots M$ **do**
  $t_i =$ **Build_extra_tree**(S)
end for
return $\mathcal{T}$

Build_extra_tree(S)
Input: A training set S of tuples (i^l, o^l)
Output: A tree t
if $|S| < n_{\min}$ **OR** all candidate variables are constant in S **OR** the output value is constant in S **then**
  return a leaf with the average output in S
else
  Randomly select K attributes $\{a_1, \ldots, a_k\}$ without replacement, among all (non constant in S) candidate attributes
  Draw K splits $\{s_1, \ldots, s_K\}$,
  where $s_i =$ **Pick_a_random_split**$(S, a_i), \forall i = 1, \ldots, K$
  Select a split s_*, such that $\text{Score}(s_*, S) = \max_{i=1,\ldots,K} \text{Score}(s_i, S)$
  Split S into subsets S_l and S_r according to s_*
  Build $t_l =$ **Build_extra_tree**(S_l) and t_r accordingly
  Create a node with the split s_*, attach t_l and t_r as left and right subtrees and return a tree t with this node as its root
end if

Pick_a_random_split(S, a)
Input: A subset S and an attribute a
Output: A split
Let $a^S_{\max}$ and $a^S_{\min}$ denote the maximal and minimal value of $a \in S$
Draw a random cut-point a_c uniformly in $[a^S_{\min}, a^S_{\max}]$
return the split $[a < a_c]$

Score(s, S)
Input: A split s and samples S
Output: The score of the split
return $\frac{\text{var}\{y|S\} - \frac{|S_l|}{|S|}\text{var}\{y|S_l\} - \frac{|S_r|}{|S|}\text{var}\{y|S_r\}}{\text{var}\{y|S\}}$

split and chooses the best among them. Choosing the splits involves selecting K attributes from all available attributes. An attribute in this case is a particular dimension of the training samples. For each of these attributes a random value between $[a_{\min}, a_{\max}]$ is chosen as the splitting threshold. The attribute and the random value together define a split. Each split is then assigned a score. The split with the highest value is chosen and applied. The score is defined as

$$\text{Score}(s, S) = \frac{\text{var}\{y|S\} - \frac{|S_l|}{|S|}\text{var}\{y|S_l\} - \frac{|S_r|}{|S|}\text{var}\{y|S_r\}}{\text{var}\{y|S\}}$$

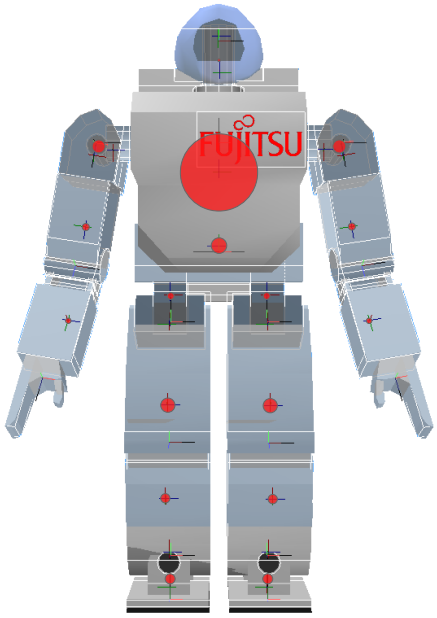
where s is a split point, S is training set, S_l and S_r are the left and right subsets of S after splitting, respectively and $\text{var}\{y|S\}$ is the variance of the output in the samples S. The score is highest if the sum of the variances of the output variable in each split is at its minimum. Intuitively it tries to minimise the output variances of the sub-trees.

A significant advantage of batch-mode reinforcement algorithms is their efficient use of the experienced state-action tuples, as opposed to Temporal Difference methods which throw away a tuple after using it to add a small increment to their value function. In the general case, batch mode reinforcement algorithms therefore usually require a lower number of episodes compared to TD methods. In order to improve a policy learnt using a batch mode reinforcement algorithm, additional samples can be generated by applying the the preliminary policy to the learning task. An advantage of the Extra-tree based approach compared to the RBF network is the fact that there is no need to specify anything manually. The partitions are chosen automatically by the Extra-tree algorithm, as opposed to the RBF linear approximation method where the centres and covariance matrices have to be selected by hand. Therefore, the Extra-tree based batch mode approach should yield a better approximation of the value function compared to the on-line SARSA(λ)-RBF approach.

3.3 Designing the Lateral Feedback Task

Unfortunately, the simulated robot using the Fujitsu trajectories learnt with PCPGs does not walk at all without lateral feedback to its ankles and hip joints. Although the trajectory produced by the CPGs is a walking pattern, it does not fit the dynamics of the robot within the Webots environment. Looking at Figure 3.3, it is obvious that the weight of the upper body of the Hoap-2 dominates. Therefore, its movements have to be very carefully generated in order to keep the robot upright.

The RL system has to learn a mapping from the input states to a lateral feedback on the ankle and hip joints. Mathematically it learns a non-linear function



Link	*Mass in kg*
Head	0.0669886
Neck	0.0208892
Body	2.37622
Lower body	0.49768
Shoulder	0.199031
Shoulder2	0.201439
Upper arm	0.218287
Lower arm	0.167592
Hand	0.0472
Hip	0.039388
Hip2	0.172696
Thigh	0.438537
Shin	0.285827
Ankle	0.171128
Foot	0.136753

Figure 3.3: The mass distribution of the Hoap-2 robot. The size of the red circles is linear with respect to the mass of the link it is drawn upon. The centre of the circles roughly represent the centre of mass of the link in question. Please note that the links *Head* and *Neck*, *Shou*lder and *Shoulder2*, *Hip* and *Hip2* as well as *Ankle* and *Foot* have been grouped in the drawing and that the values represent the left part of the robot where applicable. The values have been taken from [Cominoli 2005].

$g : s \in \mathcal{S} \mapsto \Delta f$, where Δf is the change of the feedback value applied to the hip and ankle joints. In essence it has to learn a subset of the dynamics of the robot and has to suppress the excitation of left/right movements during the walk. The reason for the left/right instability of the walking gait partly stems from the fact that the trajectory does not place the front foot evenly on the ground while taking a step forward and is further increased by a higher walking velocity (150% of the original walking trajectory speed).

The input state for the RL algorithm consists of the following values:

- Phase ϕ of the first oscillator (the hip joints run in anti-phase)
- Lateral (sidewards) tilt of the upper body of the robot
- Feedback signal of the previous learning step
- Optional: Linear lateral velocity of the upper body of the robot

The action consists of a single value which represents the change of the feedback applied to the PCPG system. Alternatively one could have used an absolute feedback action, but the relative feedback is more convenient because it is possible to use a discrete action set which improves performance, while still being able to reach every possible feedback value. Additionally, it has the advantage of producing a *smooth* feedback. The feedback applies to the lateral hip and ankle joints and makes the robot move its upper body to the left or right while keeping it mostly upright. In detail, the feedback represents a percentage of $\sqrt{\mu}$ added to the $\dot{r}$ in equation 2.7 of all oscillators of one DoF. Each given feedback value is added to the previous one (there are negative values as well) and is applied to the radius of the oscillator at each integration step. An offset feedback, simply representing a value added to the output of the systems (not integrated with the CPG equations) was briefly tried as well, but turned out to be inferior. To be fair, with a bit more tuning the offset feedback would probably have produced results comparable to the radius feedback.

The Webots system runs in a loop with the Webots controller. Each time the Webots controller is called, it can set the joints angles of the robot. After returning from the call, Webots integrates the physics equations of the simulation and therefore advances the time. The amount of time which passes between two successive controller steps is set to 8ms. Therefore, the equations of the PCPGs are integrated at the same rate. The RL system can only determine the difference between two states if a certain amount of time passes between them. Therefore it is invoked at a lower rate of 64ms. That means the RL system can inspect its state and apply an action every 64ms. After an action has been chosen, the Webots controller interpolates it linearly until the RL system is invoked again and chooses the next action.

4 Results

This chapter describes the applied research conducted, the setup used to teach a simulated humanoid robot how to walk and the results of simulation runs with Webots. During the research an attempt was made to apply the framework on the real Hoap-2 robot as well. Unfortunately writing and testing the basic software infrastructure necessary for interfacing the learning framework to the real robot took much longer than expected and there was not enough time left to conduct the planned experiments.

A variety of problems have been solved during the applied research. Solutions to these problems have been proposed by other researchers in the past and results have been published, but nevertheless the solutions had to be implemented for this thesis. The first problem to be solved was learning arbitrary periodic input signals using the PCPG system proposed by [Righetti and Ijspeert 2006]. While working on the implementation of the PCPG system, few improvements which have been mentioned above and are described in further detail in chapter 2 have been added. The next step was the integration of the PCPG system with the Hoap-2 library.[1] During this integration work, several bugs have been fixed and support for the gyroscope sensor needed for the active feedback has been written. From that point on, the interface between the PCPG system and the simulated Hoap-2 worked and steering the robot using the output of the Pattern Generators was possible.

Due to differences between the Webots versions used it was unfortunately not possible to reproduce the results from [Righetti and Ijspeert 2006]. The simulated robot was unable to walk because of lateral instabilities. After several unsuccessful attempts of optimising the input trajectory to the learning system and changing the variables of the walking system, implementing the RL feedback framework first seemed like the most promising path. After solving several technical difficulties involving the integration of the RL library with the PCPG Webots controller, the robot was finally able to walk one or two steps without tipping over. However, the results obtained so far left a lot to be desired. Before finally having a controller which produced stable walking trajectories for the Hoap-2, several different RL algorithms were tried, the feedback pathways were changed and the input trajectory to the PCPG system were further optimised. The final feedback policy is able to actively balance the simulated robot during the walk and even

[1] This library as well as the RL library was written by DI Gerhard Neuman.

recovers from excitations of the left–right movements. Still, sometimes the policy fails and the robot tips over. It is, however, quite certain that the excitation of the dynamics which makes the policy fail are due to the strictly stiff physics of the simulated joints and do not occur on the real robot because of motor backlash which has a damping effect.

4.1 Architecture

This section describes the architecture of the software used for the the lateral feedback task and additional components which were designed for interfacing with the real Hoap-2 robot.

On a high level the setup consists of a stand-alone program for learning the input trajectory with the PCPG system, and a Webots controller for steering the robot during the simulation. The stand-alone tool is invoked by a Matlab script for easy adjustment of the parameters and simple visualisation of the results. It takes a Fujitsu pulse trajectory[2] as an input and uses this trajectory as P_{teach} for the system described in chapter 2. The system is run for a configurable amount of time and the results are stored in a file which can be read by the Webots controller later on.

There are two versions of the Webots controller, one for each RL algorithm described in chapter 3, but apart from the necessary changes due to the different RL algorithm they are identical. The controllers use the Hoap-2 library and the RL Toolbox[3] developed at the Institute for Theoretical Computer Science. The Hoap-2 library provides an abstract interface to the Hoap-2 robot and can be switched from interfacing with a simulated robot to steering the real robot without having to develop a new controller. The integration work for the real robot interface in collaboration with Gerhard Neumann was part of this thesis. The architecture of the interface is described in section 4.1.1. The RL Toolbox provides a collection of RL related classes which can be combined in various ways. According to [Neumann 2005] nearly all common RL algorithms such as $TD(\lambda)$ learning for the V- and Q-Function, discrete Actor-Critic learning, dynamic programming approaches and prioritised sweeping as well as specialised algorithms for continuous state and action spaces are included.

Depending on the RL algorithm, learning takes place during the simulation run or after several completed runs (called episodes) as a batch process for the SARSA(λ) and the Extra-trees based batch mode approaches respectively. How-

[2]The trajectories supplied by Fujitsu are defined in pulses. One degree corresponds to 209 pulses according to [Fuj 2004].

[3]The toolbox was written by DI Gerhard Neumann and is available at `http://www.igi.tugraz.at/ril-toolbox`.

ever, the RL policy is always active during a simulation run. Every n simulation steps, the RL system receives a new state and calculates the action to execute for the next n steps. At each simulation step, the equation systems of the PCPGs are integrated and the feedback from the RL system is added. The result of the integration is then applied to the robot joints under control.

4.1.1 Extensions for the real Hoap-2

The Hoap-2 platform from Fujitsu consists of a Linux PC with the RTLinux kernel extensions version 3.2-pre1 from `http://www.rtlinux-gpl.org` interfacing to the robot itself via an USB 1.1 interface. Fujitsu's intention was to let the controller run in user- or kernel-mode on the supplied Linux PC. For today's standards, however, the computer is just not powerful enough for running sophisticated learning setups, which posed several challenges in order to create an interface to the platform. The first problem was that the robot expects a new position every 1ms in order to produce smooth motion. Secondly, there was no useful interface for conveniently writing positions and reading sensor information from user-space. Third, there were no means for debugging kernel modules on the PC (i.e. there was no serial interface cable attached to it). Fourth, I had no physical access to the machine while developing the interface, which is problematic when trying to develop kernel-mode software. Another challenge was that the *posture sensor* consists of a linear acceleration sensor and a gyroscope without any provided routines for converting the raw sensor readings to a posture.

In order to simplify the development of controllers for the Hoap-2 robot and to make the transition from a Webots controller to a Hoap-2 controller smooth, a system consisting of the following components was designed an implemented for this thesis.

- Extensions to the Hoap-2 library
- A server mediating between the controller and the kernel module
- An RTLinux kernel module

See Figure 4.1 for a diagram of the systems architecture. The Hoap-2 library is used as an abstraction layer by the controller and can either send commands to Webots or to the real robot. It provides facilities for reading sensor information and for setting the next positions. The Hoap-2 server forwards the commands received over TCP/IP from the controller (typically on another machine) to the RTLinux kernel module via a kernel pipe and pushes sensor information from the module to the controller. The real-time kernel module is responsible for driving the robot via the USB interface every 1ms, for calculating intermediate

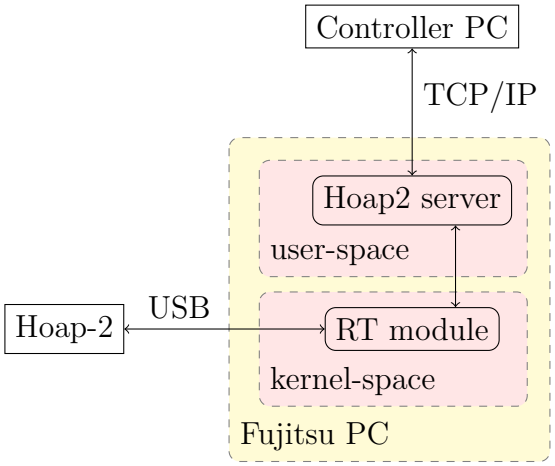


Figure 4.1: The architecture of the Hoap-2 interface software.

positions from the commands received from the controller and for reading sensor information from the robot and forwarding it to the controller via the server.

Apart from writing the interface for controlling the robot using the Hoap-2 library, the raw gyroscope values had to be integrated in order to provide the tilt of the trunk to the Webots controller. The implementation which was previously used did not provide satisfactory results. The authors calculated a moving average of the raw sensor values at start-up and subtracted the obtained value from the sensor readings thereafter. For some time it seemed that without implementing a Kalman filtering system using the accelerometer and the gyroscope, it would be impossible to obtain reliable measurements for more than a few seconds. However, after discovering that the existing implementation used integer values for the average (the authors probably assumed the precision would be adequate since the sensors readings are integers as well) and replacing it with a floating point variable, the measurements improved by magnitudes and were accurate enough. The integration method used is simple Euler integration whenever sensor values are sent to the controller. Using e. g. Runge-Kutta integration at the same rate as the routine sending values to the robot runs (1ms) would probably improve the results. Unfortunately the limited amount of time spent at the EPFL did not permit testing this approach. The control PC could, however, be too slow for this task anyway.

One problem of the architecture outlined above is the lag introduced by the kernel scheduler for the processes running the network server and the controller. This lag can be as high as 100ms in certain cases. In order to minimise the effect it is crucial to run these processes with the highest priority available[4], put the

[4]The highest priority on Linux systems is -20.

system into single-user mode and optionally set the scheduling policy to either SCHED_FIFO or SCHED_RR and the real-time priority to its maximum using the sched_setscheduler syscall and/or decrease the duration of the standard time slice.[5] See [Bovet and Cesati 2005] for more information about the Linux kernel and its scheduler.

Although it was not possible to run the experiments described in section 4.2, the system was implemented and tested with a controller from a project with the Hoap-2 robot by DI Helmut Hauser.

4.1.2 Additional Tools developed

An implementation of the PCPG system in Matlab for easier experimentation has been written. The scripts produce an analysis of the results of learning an arbitrary periodic input signal. Having an implementation of the PCPG system in Matlab enables a thorough study of its properties. A tool in Matlab for manipulating and refitting a trajectory with cubic splines in order to facilitate the manual optimisation of the walking trajectory was written, because the original trajectories from Fujitsu were quite difficult to learn with a function approximation scheme based on trigonometric functions. The reason is that these function approximations are not particularly good at modelling discontinuities with a low number of primitives. Keeping the original trajectories would have put a significant burden on the speed of the controller, as a higher number of oscillators would have had to be used. In order to make this tool useful for the trajectories from Fujitsu, a converter between Fujitsu pulse- and angular format was required. A converter for each direction of conversion has been developed in the course of completing this thesis. A yet unfinished library for plotting trajectories during a debugging session using the Kst tool[6] has been written as well.

The complete source code developed during the research is available under the GPL v2[7] upon request.

4.2 Learning Task Setup

The simulated robot used for the learning task is a model of the Hoap-2 research robot manufactured and sold by Fujitsu Automation. The real robot is about 50cm tall and weights approximately 7kg. The system has four joints in each arm and six joints in each leg and a total of 25 degrees of freedom. The simulation

[5] Changing the time slice usually requires a kernel patch and rebuild.

[6] From the project's website `http://kst.kde.org/`: 'Kst is the fastest real-time large-dataset viewing and plotting tool available and has basic data analysis functionality.'

[7] `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

software used is Webots 5.0.6, a proprietary package sold by Cyberbotics[8]. It includes a model of the Hoap-2 robot developed during the research for the Diploma thesis [Cominoli 2005]. The author tried very hard to make the model as accurate as possible, but stated 'Concerning the evaluation of the current version of the simulator, I do not think that the results are excellent'. One particular aspect of the model is that it cannot simulate motor backlash, which results from wear and manufacturing circumstances and is nonlinear in the servo angle. While motor backlash reduces the ability of the robot to follow trajectories exactly, it has the convenient property of damping movements. The Webots model on the other hand is strictly stiff and any disturbances are passed along the kinematic chain without any damping.

The Webots scene used for learning the feedback consists of the robot model and a huge level floor plate. The initial position of the robot closely matches a position along the walking trajectory supplied by Fujitsu.

Compared to [Righetti and Ijspeert 2006], the context of the problem this thesis solves is more difficult because the software used for simulation has been improved meanwhile and the approach from the original publication does not make the robot walk using the newer version. However, the realism of both simulation environments is discussible and it remains unclear which one fits the dynamics of the real robot better.

The system for controlling the joints of the robot consists of a network of PCPGs as shown in Figure 4.2. Each PCPG contains five oscillators and is trained with the Fujitsu walking trajectory for the corresponding joint as the target signal. After the learning phase the system is able to reproduce the walking pattern with a maximum error of 0.1430, 100 seconds after switching off the driving signal, while the error was at a maximum of 0.1356 between seconds 200–250. Please see Figure 4.3 for details.

The reward function of the RL system assigns a big negative reward to the 'failed' state (robot tipped over) and a small negative reward relative to the lateral tilt as soon as it is higher than 0.05 rad. The feedback is the improved version on the radius as defined in section 2.3.1 and the action a chosen by the agent is applied as follows (the gain is set to $g = 1$):

$$\begin{aligned} f_{\text{left},2} = f_{\text{right},2} &= a_{t-1} + a \cdot \text{sgn}(\phi_{\text{left},2}) \\ f_{\text{left},6} = f_{\text{right},6} &= a_{t-1} - a \cdot \text{sgn}(\phi_{\text{left},2}). \end{aligned}$$

[8] `http://www.cyberbotics.com`

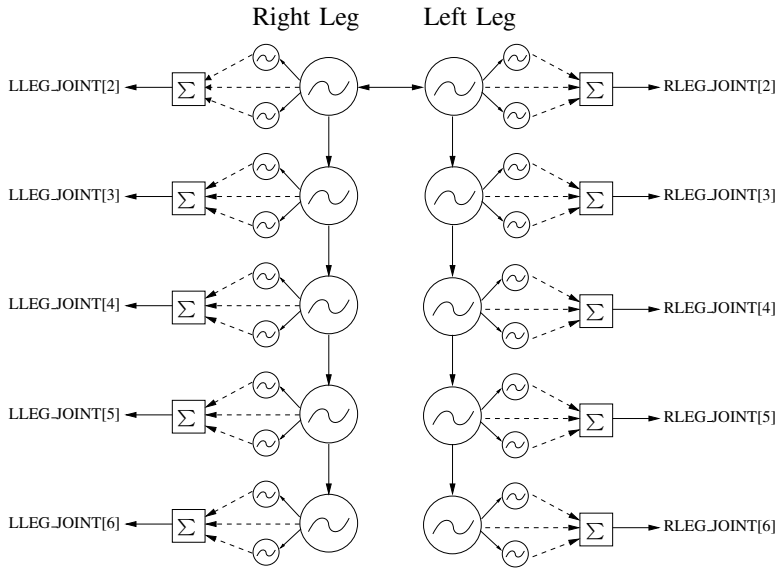


Figure 4.2: The structure of the PCPG system. Taken from [Righetti and Ijspeert 2006].

4.3 Results

This sections presents the results from four different RL runs on the task of balancing a simulated Hoap-2 robot during a walking gait. Three runs with the Sarsa(λ) algorithm using an Rbf network for Q function approximation and one run with the batch-mode Extra-tree based algorithm were conducted. The state signal consists of either three our four values, namely the lateral tilt of the upper body, the previously given feedback and the phase in the first case and in addition to the aforementioned states, the linear lateral velocity of the upper body in the four dimensional case. Table 4.1 shows various properties of the conducted simulation runs and the corresponding figure containing the results. The average number of steps shown in the table have a duration of 64ms each, therefore the best result obtained with a 1ms Webots time-step is an average walking duration of 70.4 seconds. Sometimes, however, the policy still fails and the robot tips over. As far as visual analysis of the simulation runs have shown it is due to an excitation of the left-right motion, which the policy sometimes fails to suppress properly. The remaining section will refer to the individual simulation runs by the index in table 4.1. The movements of the robot during an episode of run 2 can be seen in Figure 4.13.

Figure 4.8 confirms that the learnt Fujitsu trajectory does not fit the dynamics of the robot walking at 150% of the original speed. Generating a new input

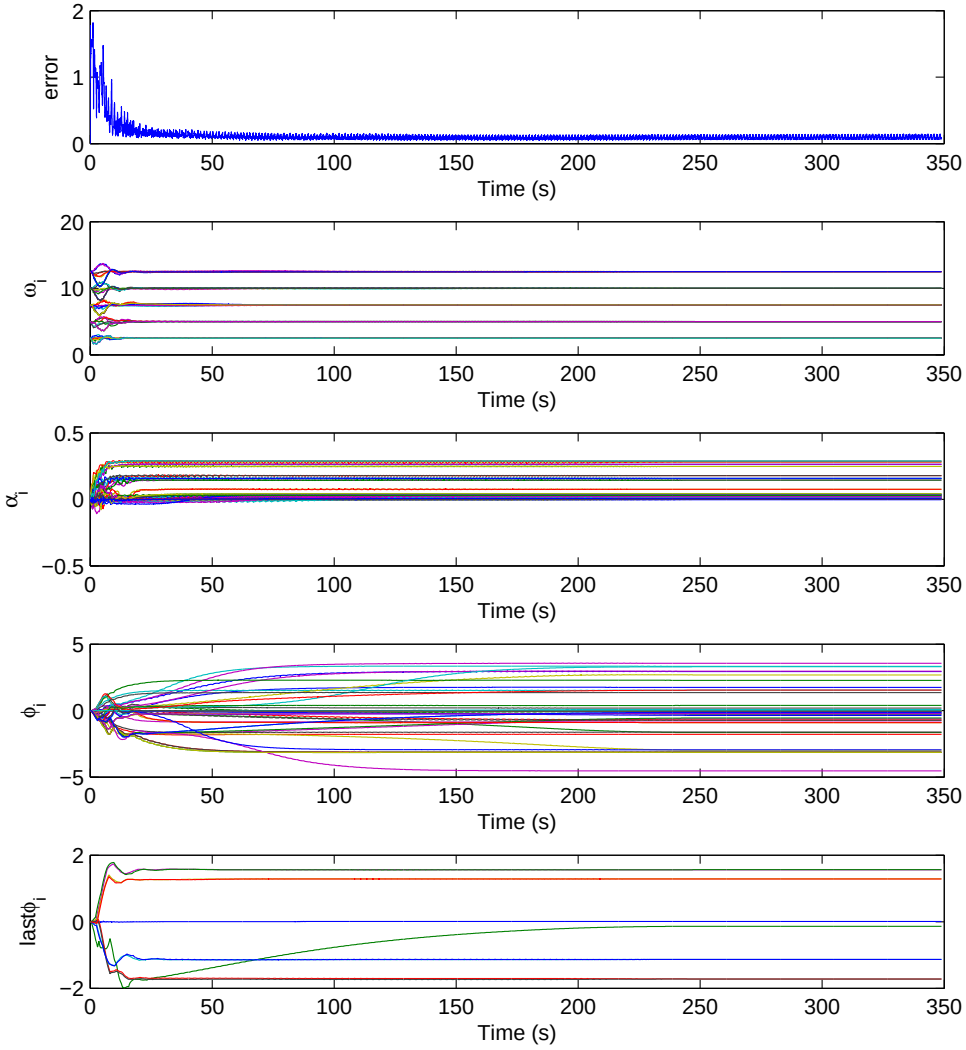


Figure 4.3: The results of learning the walking trajectory with the PCPG network. One cycle of the trajectory is 2.514 seconds long. Each PCPG contains 5 oscillators and learning stops after 250 seconds. After that the oscillators are free running for 100 seconds. The variables are set to: $\tau = 1, \epsilon = 4, \eta = 0.9, \mu = 1, \gamma = 1, \lambda = 0.4, \tau_{ext} = 10, \omega_{i,0} = 2.5 \cdot i$. The first row represents the error $P_{\text{teach}} - G$. The last row shows the evolution of the external phase offset state.

run index	algorithm	# states	# episodes (≈)	# steps (≈)	webots ts	figure
1	SARSA(λ)	3	6000	1100	1ms	4.4
2	SARSA(λ)	4	9000	820	1ms	4.5
3	Extra-trees	4	3540	360	1ms	4.6
4	SARSA(λ)	3	4500	1210	8ms	4.7

Table 4.1: The simulation runs, their results and parameters. The number of steps is the maximum average number of RL steps during the run averaged over 150 episodes. The Webots time-step denotes how often the physics equations are integrated in Webots and affects the complexity of the task significantly.

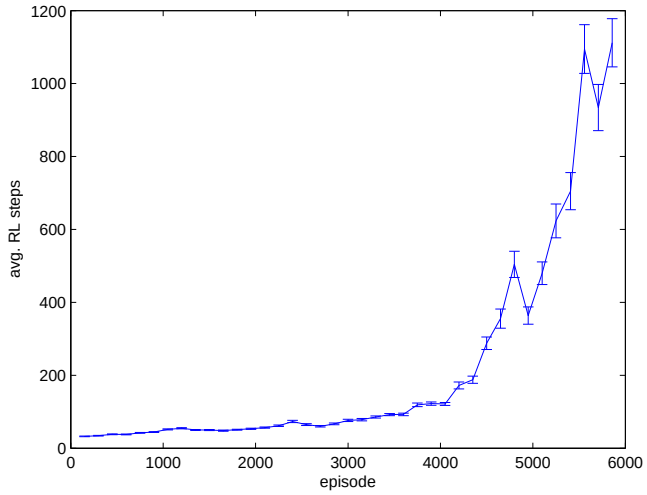


Figure 4.4: The SARSA(λ) algorithm with a three dimensional state vector and a Webots time-step of 1ms running for 5935 episodes. The error bars show the standard error of the number of steps per episode averaged over 150 episodes.

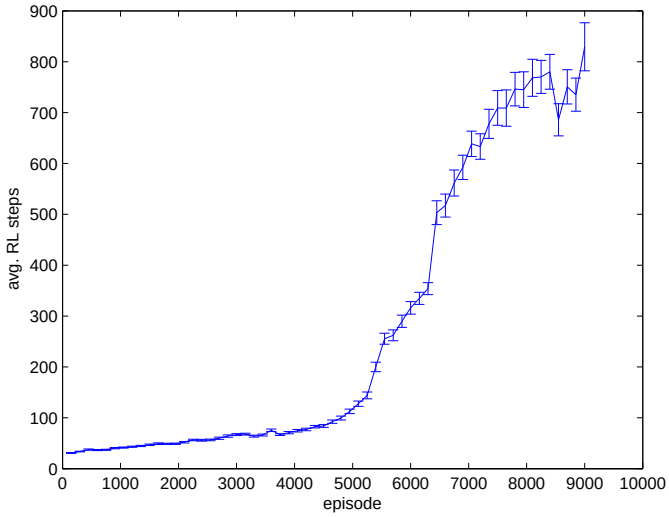


Figure 4.5: The SARSA(λ) algorithm with a four dimensional state vector and a Webots time-step of 1ms running for 9052 episodes. The error bars show the standard error of the number of steps per episode averaged over 150 episodes.

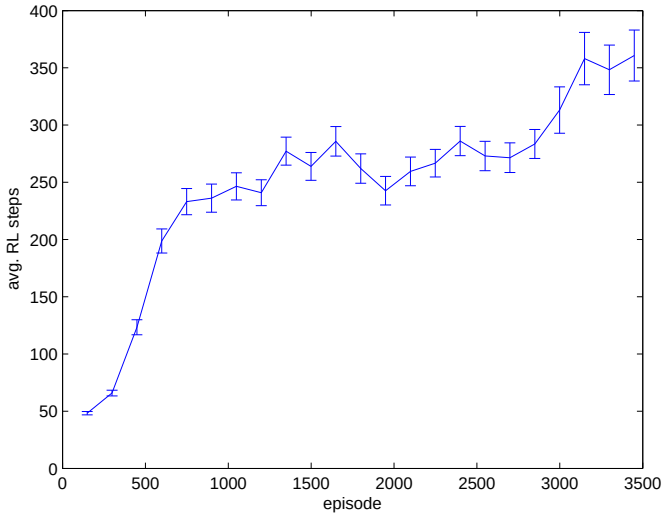


Figure 4.6: The Extra-tree based batch mode algorithm with a four dimensional state vector and a Webots time-step of 1ms running for 3540 episodes. The error bars show the standard error of the number of steps per episode averaged over 150 episodes.

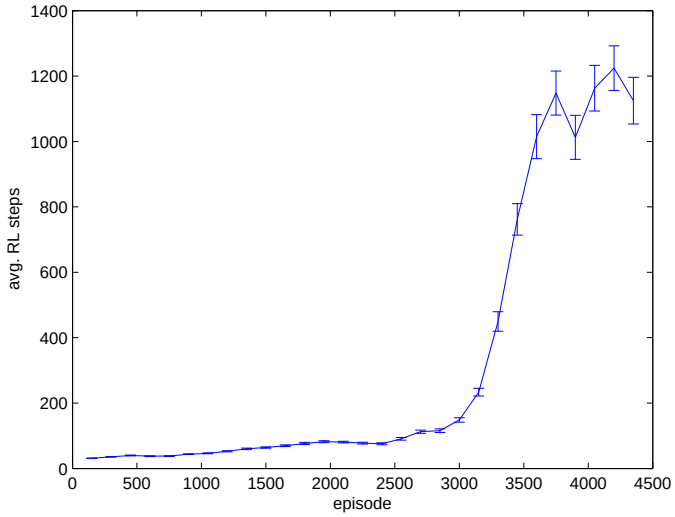


Figure 4.7: The SARSA(λ) algorithm with a three dimensional state vector and a Webots time step of 8ms running for 4542 episodes. The error bars show the standard error of the number of steps per episode averaged over 150 episodes.

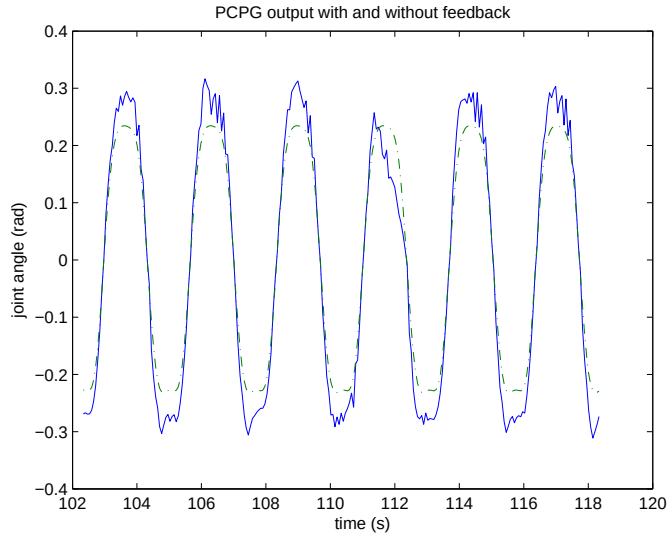


Figure 4.8: The output of the first PCPG without any feedback (green dash-dotted) and with the radius feedback applied (blue) during an episode of run 1.

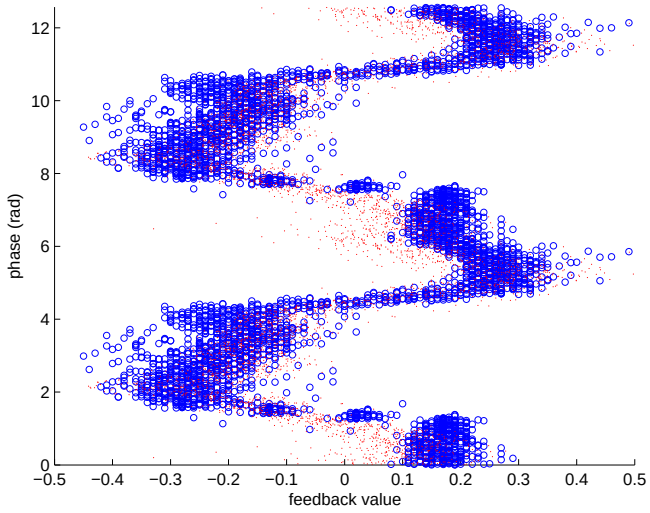


Figure 4.9: A scatter plot of the phase vs. the absolute feedback during an episode of run 2. The blue dots are the feedback as learnt by the RL agent, while the red dots represent a linear model of the feedback.

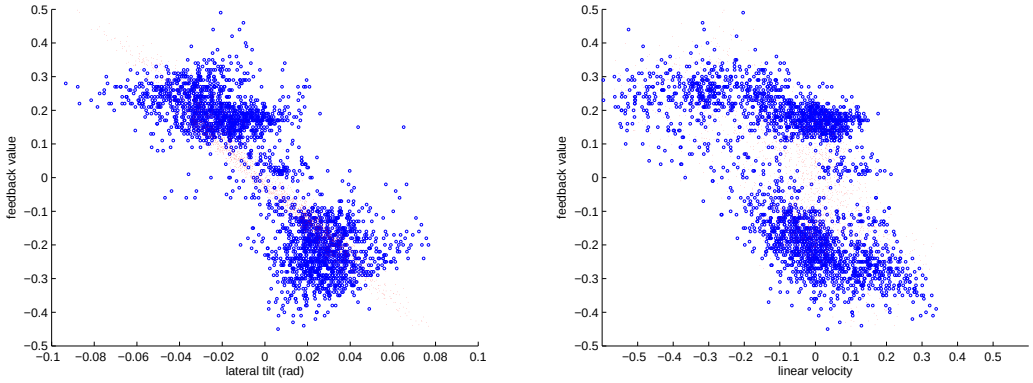


Figure 4.10: A scatter plot of the absolute feedback vs. the lateral tilt on the left and the linear velocity on the right during an episode of run 2. The blue dots are the feedback as learnt by the RL agent, while the red dots represent a linear model of the feedback.

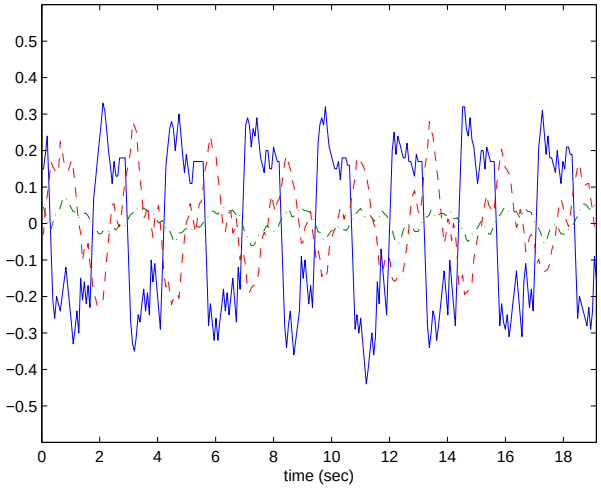


Figure 4.11: A plot of the feedback the policy acquired during simulation run 2 applied to the robot vs. the lateral tilt and linear velocity. The blue, solid line shows the absolute feedback applied. The red, dashed plot is the lateral linear velocity of the upper body and the green dash-dotted line represents the lateral tilt.

trajectory for the PCPG framework based on the old movements with an average feedback added to it would probably simplify the learning task and improve the overall stability of the walk. An interesting result is that the learnt policy from run 2 works for a slower walk at the original speed as well. It also works for a different Webots integration step (8ms instead of the learnt 1ms). This suggests that the feedback policy generalised quite well. When comparing Figures 4.4 and 4.7, please keep in mind that the task is much simpler with the 8ms integration time-step as the physics model is less accurate. When comparing Figure 4.6 to 4.5 and 4.4 it is obvious that the batch mode algorithm finds a good policy after recording a small number of additional episodes. It reaches an average number of steps per episode of 250 after just 1000 episodes compared to 5500 for the four- and 4500 episodes for the three-states SARSA(λ) algorithm. On the other hand, when taking the number of loaded episodes (5900) into account the batch mode algorithm seems to be slower, but the algorithm would probably work with fewer initial episodes as well, because many of the loaded episodes are duplicates or at least very similar to other episodes within the training set. Another difference between the on-line and batch mode results is the fact that the batch mode task was run with a maximum of 500 steps per episode up to episode 2800, while all on-line tasks were allowed to take up to 2500 steps per episode from the beginning.

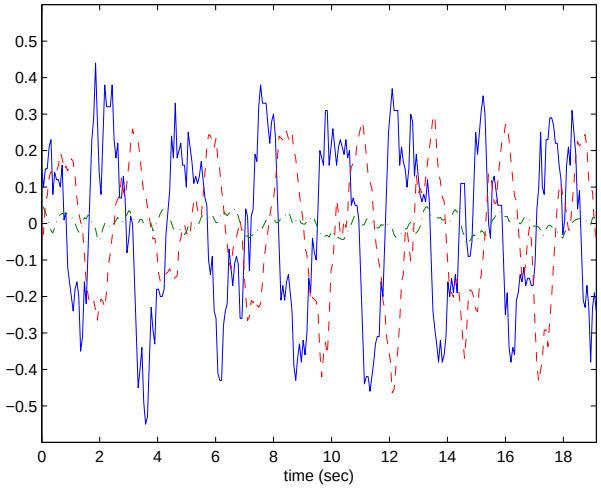


Figure 4.12: A plot of the feedback the policy acquired during simulation run 3 applied to the robot vs. the lateral tilt and linear velocity. The blue, solid line shows the absolute feedback applied. The red, dashed plot is the lateral linear velocity of the upper body and the green dash-dotted line represents the lateral tilt.

The batch mode task has a slightly different reward function as well. Instead of receiving a reward of -10 in case the robot tips over, the batch mode task assigns a value of -50 to the failed state.

Figures 4.9 and 4.10 show the evaluation of the learnt policy during one episode. It can easily be deduced from Figure 4.9 that the policy is in large parts responsible for compensating for the dynamic effects of the left/right swinging pattern of the upper body. The trajectory as learnt by the PCPGs does not place the front foot evenly on the ground at the moment it touches the floor. Without feedback, the upper body of the robot accelerates in the lateral direction and the robot tips sidewards. The feedback reduces this effect by firmly applying a force counteracting the side-ward acceleration. Figure 4.10 shows the relationship between the feedback and the lateral tilt as well as the linear lateral velocity. Especially the left plot nicely shows that the linear model cannot correctly reproduce the feedback policy. The right part of the plot shows why the policy learnt without the linear lateral velocity works as well. There are large clusters of points producing a different feedback for the same lateral velocity. This suggests that the feedback does not have a strong dependency on the lateral velocity, which is also supported by the parameters of the second linear model.

Finally, Figures 4.11 and 4.12 show the lateral tilt, the liner velocity and the

applied feedback during an episode of run 2 an 3 respectively.

4.3.1 Regression analysis of the policy

In order to confirm the hypothesis that a nonlinear policy is required to make the robot walk, a multivariate linear regression analysis of the policy learnt by SARSA(λ) with an RBF network was performed. The learnt policy is the result of a simulation run with 9000 RL episodes with a state vector including the linear lateral velocity of the upper body. The regression analysis tries to fit a linear model $y = X\beta + \epsilon$ to the learnt policy. Two versions of a fitted linear policy were tried. The first version with an observation vector of $y = [\text{feedback}]$ and regressors $X = [\text{lateral tilt}, \text{previous feedback}, \phi, \text{lateral linear velocity}]$. The parameters b turned out to be $b = [-0.6516, -0.2071, -0.0038, -0.2295]^T$ with a mean squared error of MSE $= \frac{\sum_{i=1}^{n}(y_i - [X \cdot b]_i)^2}{n} = 0.0041$. While this seems like the most obvious thing to do, it cannot work because the prediction is a value which is continuously added up during the simulation. Therefore, while the MSE on the observation is small, the error on the actual feedback applied to the joints becomes huge after a few steps because it is added up as well. For the second version the linear model was changed to produce absolute feedback values, instead of a relative feedback like the RL policy and the first linear model. The observation vector was set to be $y = \text{cumsum(feedback)}$ and the regressors were set to be $X = [\text{lateral tilt}, \phi, \text{lateral linear velocity}]$, where cumsum(a) calculates a vector containing the cumulative sum of the elements of a given column vector, that is $\text{cumsum}(a) = [a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, \sum_i a_i]^T$. The regression analysis led to the parameter vector $b = [-5.1971, -0.0063, -0.2904]$ with a MSE $= 0.0139$. Neither of those models managed to make the robot walk a single step, which supports the hypothesis that a nonlinear model is required. Nevertheless they provide interesting results. The parameter vector of the first analysis divided by the vector of standard deviations of the regressors $[0.0299, 0.2171, 1.8131, 0.1668]$ gives $[-21.79, -0.95, -0.0021, -1.38]$, while the same procedure applied to the results of the second analysis gives $[-173.82, -0.0035, -1.74]$. This result clearly shows that the most important state is the lateral tilt. This explains why the policy learnt without the linear velocity produces good results as well.

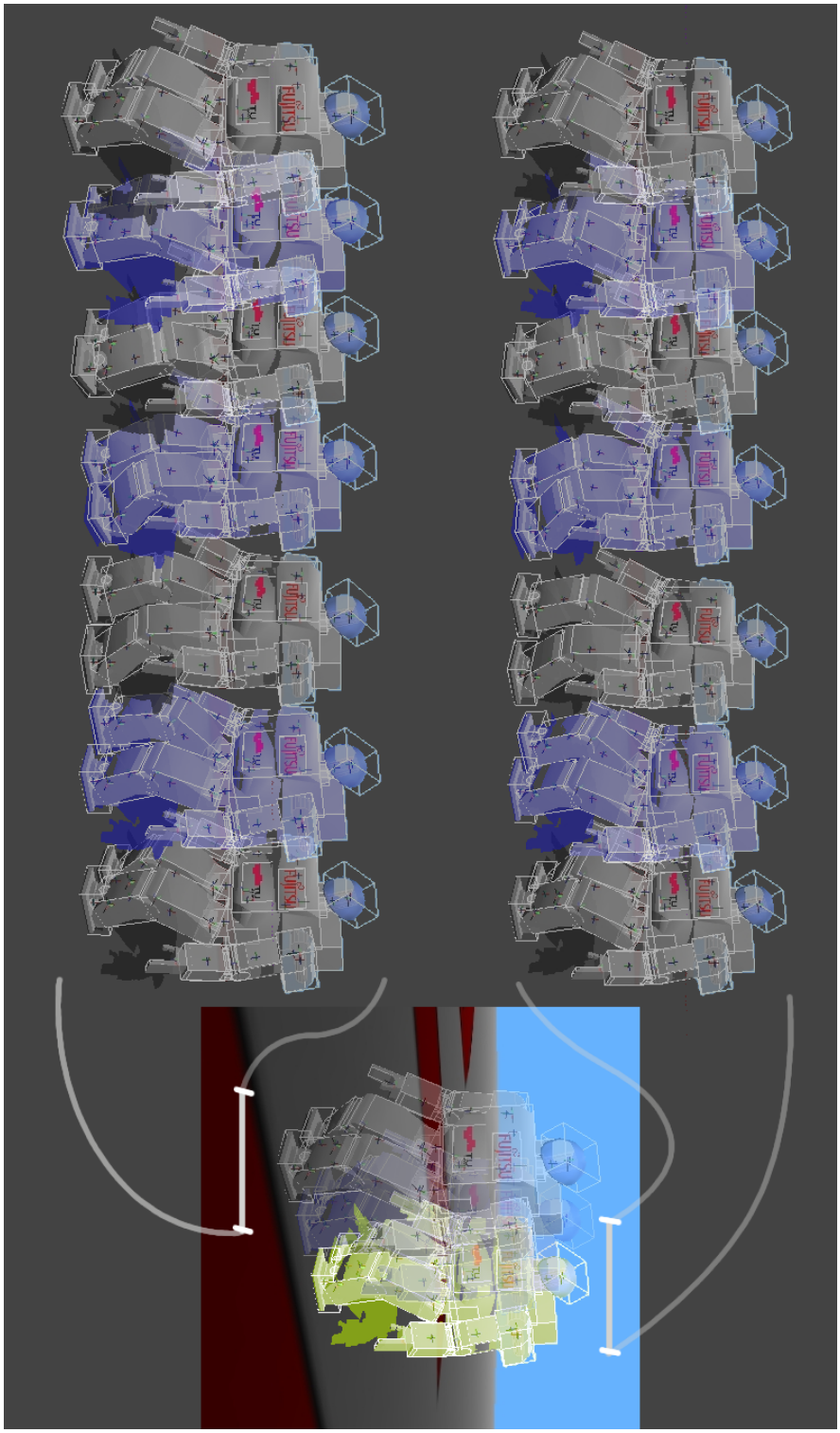


Figure 4.13: Walking gait of the Hap-2 robot during an episode of run 2. The frames are evenly spaced. The distance the robot moves from one frame to the next is therefore linearly related to the distance moved in the simulation.

5 Conclusion

Balancing a simulated humanoid robot while walking is a difficult task because simulations mostly use stiff models. This thesis successfully applied two Reinforcement Learning approaches to the task. Both algorithms were able to find a usable policy which managed to make the robot walk for up to 70 seconds on average without tipping over. Although the resulting policies could not be evaluated on the real robot due to time constraints, it is very likely that they would work because the dynamics of the robot compared to its simulation are easier to handle, because of the damping effect introduced by motor backlash.

The contribution of this thesis to the Programmable Central Pattern Generator approach is a new coupling term for keeping the phase difference which improves the robustness of the system to external perturbations, simplifies the dynamics and increases the flexibility of the system. The system is more robust because the coupling term acts only on the phase of the oscillators instead of perturbing the oscillator as a whole. It simplifies the dynamics because the system does not have to compensate for the perturbations of the coupling while learning the input signal. The system is more flexible due to the ability to adjust the speed of recovery from perturbations while the system is running.

Working on the PCPG system was challenging and fun, while finding a working RL approach, tuning the system and running the simulations was a bit tedious mostly because of endless technical difficulties involving the Webots software, the controllers and the integration with the RL library. Working with Webots was very time consuming because it is proprietary software and apart from being unable to debug it there is only a limited number of licences available at the institute, which prevented parallel execution of several simulation runs. An Open Source simulation software would probably have saved weeks of time. Due to limitations of the Webots software, the simulation runs took a very long time to complete and it was therefore not possible to try many different learning parameter settings in order to optimise the results. This is especially true for the batch mode run which took over a week to complete. The results from the batch mode and the on-line runs cannot be directly compared. Given enough time to optimise the learning parameters, it is very likely that the Extra-tree based batch mode approach would have led to better results compared to the on-line approach because it is superior with respect to approximating the value function.

Further Work One of the weaknesses of the approach outlined in this thesis is the learnt trajectory which does not fit the dynamics of the robot. There are several publications available aiming at evolving a controller to generate a walking trajectory which exploits the dynamics of the walking machine, rather than working against them. One possible approach may be to entrain the oscillators with the robot. The load sensors on the feet could, for example, generate a driving signal which is coupled to the oscillators producing the trajectories. This could potentially be used to create entrainment between the robot and the oscillators. An appropriate feedback signal would have to be defined, in order to synchronise the PCPGs with the feedback from the foot sensors. One could probably use ideas from research on synchronisation between a normal oscillator and a relaxation oscillator. Another, more promising, approach would be to optimise the parameters of the PCPGs using evolutionary algorithms starting from the current values. [Hein, Hild, and Berger 2007], for example, evolve a controller based on a neural oscillator. The feedback policy from this thesis could be used as one component of an error function, by increasing the error if the policy has to apply a high feedback to keep the robot upright.

The task solved in this thesis could be supplemented with further challenges by applying external forces to the robot, or changing the walking speed during the task. The addition of a slope for walking up or downwards would be especially interesting. This would probably require a modification of the feedback system in order to compensate for the saggital tilt. The current system could probably compensate for a small tilt of the ground in the lateral direction, while this has not been confirmed. Another interesting simulation run would be an evaluation of the policy with a modified weight distribution of the robot. One could, for example, increase the mass of one arm until a threshold is found which makes the policy starts to fail.

There are further opportunities to improve learning system. With the current implementation, the on-line algorithms are able to create a usable policy within two days of computation time, while the batch mode system is much slower. The problem is not the algorithm itself but the amount of data which has to be saved and reloaded after each simulation run. It is minimal in the on-line case because the algorithm does not need data from past experiences, as opposed to the batch mode approach additional episodes are recorded. The limitation comes from Webots which requires the controller to be reloaded after a reset of the simulation. One possible solution would be to split the controller into one small part which is reloaded by Webots upon reset and another process which holds the data and does the actual computation. These processes could then communicate via IPC[1] mechanisms. This approach would speed up the learning process for the

[1]Inter-Process Communication mechanisms are either provided by the operating system or

batch mode algorithm significantly and would probably lead to improvements in the on-line case as well.

After solving the performance problems it would be feasible to introduce additional variables to the state signal. These variables could, for example, include values derived from touch sensor information on the feet. On the other hand, the action space could be widened as well. It would be interesting to introduce a saggital feedback term as well as splitting the current lateral feedback into separate variables for the hip and ankle joints. Maybe the RL system could further improve the stability of the walk if it were able to change the walking speed on the fly.

by separate libraries and enable two or more running programs to share data and/or pass messages to each other.

Appendix

This chapter includes material which did not fit into the main text.

Proof of equivalence of the adaptive Hopf oscillator in Polar- and XY-coordinate systems

This section will prove that the frequency and amplitude adaptive Hopf oscillators in the XY- and polar coordinate system are equivalent. The following equations define the two systems. The first set of equations is the system in polar coordinates while the second set defines the oscillator in the XY coordinate space.

$$\begin{aligned}
\dot{r} &= (\mu - r^2)r + \epsilon F \cdot \cos(\phi) && (5.1)\\
\dot{\phi} &= \omega - \frac{\epsilon}{r} F \cdot \sin(\phi) && (5.2)\\
\dot{\omega} &= -\epsilon F \cdot \sin(\phi) && (5.3)\\
\dot{\alpha} &= \eta F \cdot \cos(\phi) \cdot r && (5.4)
\end{aligned}$$

$$\begin{aligned}
\dot{x} &= (\mu - r^2)x - \omega y + \epsilon F && (5.5)\\
\dot{y} &= (\mu - r^2)y + \omega x && (5.6)\\
\dot{\omega} &= -\epsilon F \cdot \frac{y}{r} && (5.7)\\
\dot{\alpha} &= \eta F \cdot x && (5.8)\\
r &= \sqrt{x^2 + y^2} && (5.9)
\end{aligned}$$

The functions

$$\begin{aligned}
x &= \cos(\phi) r && (5.10)\\
y &= \sin(\phi) r && (5.11)
\end{aligned}$$

which transform into

$$\begin{aligned}
\dot{x} &= \dot{r}\cos(\phi) - r\sin(\phi)\dot{\phi} && (5.12)\\
\dot{y} &= \dot{r}\sin(\phi) + r\cos(\phi)\dot{\phi} && (5.13)
\end{aligned}$$

when derived by $\frac{dx}{dt}$ and $\frac{dy}{dt}$ respectively, will be useful for the proof. The proof for the functions $\dot{\omega}$ and $\dot{\alpha}$ is short. Substituting 5.11 for y in 5.7 leads directly to 5.3 and replacing x in 5.8 by 5.10 results in 5.4. Transforming equations 5.5 and 5.6 requires substituting 5.12, 5.13, 5.10 and 5.11 for $\dot{x}$, $\dot{y}$, x and y respectively, which leads to

$$\begin{aligned} \dot{r}\cos(\phi) - r\sin(\phi)\cdot\dot{\phi} &= (\mu - r^2)\cdot r\cos(\phi) - \omega r\cdot\sin(\phi) + \epsilon F \\ \dot{r}\sin(\phi) + r\cos(\phi)\cdot\dot{\phi} &= (\mu - r^2)\cdot r\sin(\phi) + \omega r\cdot\cos(\phi). \end{aligned}$$

Solving the first equation for $\dot{r}$ and the second for $\dot{\phi}$ leads to

$$\begin{aligned} \dot{r} &= \frac{1}{\cos(\phi)}\left(\mu r\cdot\cos(\phi) - r^3\cos(\phi) - \omega r\cdot\sin(\phi) + \epsilon F + r\sin(\phi)\cdot\dot{\phi}\right) \\ \dot{\phi} &= \frac{1}{r}\left[\frac{1}{\cos(\phi)}\left(\mu r\cdot\sin(\phi) - r^3\sin(\phi) + \omega r\cdot\cos(\phi) - \sin(\phi)\cdot\dot{r}\right)\right]. \end{aligned}$$

Substituting these two equations into each other and simplifying the results finally leads to the result:

$$\begin{aligned} \dot{r} + r^3 = \mu r + \cos(\phi)\cdot\epsilon F \implies \dot{r} &= (\mu - r^2)r + \epsilon F\cdot\cos(\phi) \\ \omega = \frac{1}{r}\sin(\phi)\cdot\epsilon F + \dot{\phi} \implies \dot{\phi} &= \omega - \frac{\epsilon}{r}F\cdot\sin(\phi)\,\square \end{aligned}$$

Bibliography

Hans Jochen Bartsch. *Taschenbuch Mathematischer Formeln*. Fachbuchverlag Leipzig, 19th edition, 2001.

Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., third edition, 2005.

Collaborative authorship. Wikipedia, the free encyclopedia. `http://www.wikipedia.org`, 2007.

Pascal Cominoli. Development of a physical simulation of a real humanoid robot. Diploma thesis, Swiss Federal Institute of Technology, Lausanne, 2005.

Bryan C. Daniels. Synchronization of globally coupled nonlinear oscillators: The rich behavior of the kuramoto model. Technical report, Ohio Wesleyan University, 2005. URL `http://physics.owu.edu/StudentResearch/2005/BryanDaniels/kuramoto_paper.pdf`.

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005. URL `http://www.montefiore.ulg.ac.be/services/stochastic/pubs/2005/EGW05/ernst05a.pdf`.

Hoap-2 Instruction Manual. Fujitsu Automation Co., Ltd., 3rd edition, 2004. URL `http://jp.fujitsu.com/group/automation/downloads/en/services/humanoid-robot/hoap2/instructions.pdf`.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63:3–42, 2006.

Daniel Hein, Manfred Hild, and Ralf Berger. Evolution of biped walking using neural oscillators and physical simulation. In *Proceedings of the Robocup International Symposium*, 2007.

Auke Jan Ijspeert and Jérome Kodjabachian. Evolution and development of a central pattern generator for the swimming of a lamprey. *Artificial Life*, 5 (3):247–269, 1999. URL `http://birg2.epfl.ch/publications/fulltext/ijspeert99.ps.gz`.

Leslie P. Kaelbling and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL `http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/rl-survey.html`.

Andrija Kun and W. Thomas Miller, III. Adaptive dynamic balance of a biped robot using neural networks. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 1, pages 240–245, 1996. doi: 10.1109/ROBOT.1996.503784.

Takamitsu Matsubara, Jun Morimoto, Jun Nakanishi, Masa-aki Sato, and Kenji Doya. Learning cpg-based biped locomotion with a policy gradient method. *Robotics and Autonomous Systems*, 54:911–920, 2006.

T. Mcgeer. Dynamic walking robots and the w prize. *IEEE Robotics & Automation Magazine*, 14(2):13–15, 2007.

Jun Morimoto, Jun Nakanishi, Gen Endo, G. Cheng, C. G. Atkeson, and G. Zeglin. Poincaré-map-based reinforcement learning for biped walking. In *Proc. of the 2005 IEEE International Conference on Robotics and Automation (ICRA)*, 2005. URL `http://www.cs.cmu.edu/~cga/walking/xmorimo-icra05.pdf`.

Gerhard Neumann. The reinforcement learning toolbox, reinforcement learning for optimal control tasks. Master thesis, Graz University of Technology, Graz, Austria, 2005.

Masaki Ogino, Yutaka Katoh, Masahiro Aono, Minoru Asada, and Koh Hosod. Reinforcement learning of humanoid rhythmic walking parameters based on visual information. *Advanced Robotics*, 18:677–697, 2004. URL `http://www.er.ams.eng.osaka-u.ac.jp/Paper/`.

J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *Third IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, 9 2003. URL `http://citeseer.ist.psu.edu/peters03reinforcement.html`.

Arkady Pikovsky, Michael Rosenblum, and Jürgen Kurths. *Synchronization.* Cambridge Nonlinear Sciences Series 12. Cambridge University Press, Cambridge, UK, 2003 edition, 2001.

Marc H. Raibert. *Legged Robots That Balance.* MIT Press, Cambridge, MA, 1986.

Ludovic Righetti and Auke Jan Ijspeert. Programmable central pattern generators: an application to biped locomotion. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 1585–1590, Orlando, Florida, May 2006. URL `http://birg2.epfl.ch/publications/fulltext/righetti06.pdf`.

Ludovic Righetti, Jonas Buchli, and Auke J. Ijspeert. From dynamic hebbian learning for oscillators to adaptive central pattern generators. In *Proceedings of the 3rd International Symposium on Adaptive Motion in Animals and Machines*, Ilmenau, 2005. Verlag ISLE. URL `http://birg2.epfl.ch/users/righetti/amam05.pdf`.

Ludovic Righetti, Jonas Buchli, and Auke Jan Ijspeert. Dynamic hebbian learning in adaptive frequency oscillators. *Physica D*, 216:269–281, 2006.

Mike Stilman, Christopher G. Atkeson, James J. Kuffner, and Garth Zeglin. Dynamic programming in reduced dimensional spaces: Dynamic planning for robust biped locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2005. URL `http://www.cs.cmu.edu/~cga/walking/stilman-icra05.pdf`.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. URL `http://www.cs.ualberta.ca/~sutton/book/the-book.html`.

G. Taga. A model of integration of posture and locomotion. In *Proceedings of International Symposium on Computer Simulation in Biomechanics.*, 1997.

List of Acronyms

CPG ..... Central Pattern Generator A distributed biological neural network which can produce coordinated rhythmic signals without oscillating input from the brain or from sensory feedback.

ML ..... Machine Learning The process of developing mathematical and biologically inspired algorithmns and techniques which enable computers to learn.

PCPG ..... Programmable Central Pattern Generator Encodes a given rythmic trajectories as limit cycles of nonlinear dynamic systems.

RL ..... Reinforcement Learning Learning how to map situations to actions in order to maximise a numerical reward signal.