Eindhoven University of Technology
Department of Mathematics and Computer Science

Control of a Pneumatic Robot Arm by means of Reinforcement Learning

Sander Maas

Master's thesis

Exam committee:	prof. dr. Mark de Berg (supervisor TU/e)
	dr. Alexandra Cristea (TU/e)
	ir. Boudewijn Verhaar (supervisor Philips)
	dr. Marco Wiering (UU)

Eindhoven, October 2005

Contents

Abstract

We applied Reinforcement Learning on a real robot arm, in order to control its movements. The arm is actuated by two pneumatic artificial muscles, that expose a highly non-linear behavior. To enable a significant speed-up of the learning process, an empirical simulation is constructed, based on real robot observations. Furthermore, we introduce a learning strategy to facilitate learning. Using the simulation model and the learning strategy, Reinforcement Learning was able to quickly learn how to control the robot arm in simulation. Our experiments show that the obtained controller also performs well on the robot arm. Furthermore, Reinforcement Learning is able to adapt itself to changes in the behavior of the robot arm.

Acknowledgements

I would like to thank the following people. First of all, my Philips supervisor Boudewijn Verhaar for always being dedicated and for being a great help on any given problem. My Philips coaches Bart Dirkx, Thom Warmerdam and Erik Gaal for all discussions, ideas, insights and support they gave me. The people at Philips really improved my engineering skills. With respect to Artificial Intelligence I owe thanks to Marco Wiering, who always had time for all my questions (and there were many). Furthermore, I thank my professor Mark de Berg for supervising me.

With respect to the chip and sensors I want to thank Henk van Gijn, Erik Bastiaans, Cor van de Klooster and Ad van de Vleuten. Regarding the robot arm I thank Peter Menten and Henk Vaes.

I thank the other students Erik Manders, Marco Pedersoli, Stef van den Brink, Thijs Verhees, Peter Schipperen and Antoine Milot for numerous discussions and pleasure. And last but not least, I want to thank my parents and my girlfriend for all their support throughout my study period.

Chapter 1

Introduction

This chapter gives an introduction to the report. Section one gives a motivation for this research. Next, in section two the problem statement is discussed. After that, section three explains the control tasks. Finally, section four gives an overview of the rest of the report.

1.1 Motivation

Industrial robots. For several decades already, industrial robots have proved their value by obtaining high performance in terms of speed, accuracy and reliability. Robots are built that even have nanometer precision. This state-of-the-art technology is reached by making the construction as ideal as possible. The constructions are very stiff, the behavior is almost perfectly linear and every degree of freedom can independently be controlled. Moreover, the operating environment of the robots is known a-priori and the number of objects is limited. This way, relatively simple control techniques can be used to make exact positioning possible.

Domestic environments. However, these designs are not suitable for application in domestic environments. Domestic environments are very divers and change continuously. For example, the temperature and humidity, clothes on the ground, the remote control on the sofa, etc. Also, the robots should be able to interact with many objects that cannot be defined a-priori. Moreover, their heavy construction and strength makes interaction with humans dangerous. Therefore, the robots should be lightweight and their force should be limited.

The requirements for the robots are different as well. A vacuum cleaner does not need to have nanometer precision. However, the costs and energy consumption should be significantly lower.

From the above it is clear that the construction of domestic robots is different from their industrial equivalents. A domestic robot has a less ideal construction and it has to cope with more challenging environments. As a result, the control problem of these robots is much more complex.

Artificial Intelligence. Problems as object recognition, self-localization and muscle control are very difficult to solve with the current state of technology.

However, the proof that all these problems can be solved robustly and efficiently is all around us. Humans and animals cope with these problems without any noticeable effort. Muscles can be made stiff or compliant. They can be controlled at different temperatures and humidity. The movements can be fast, smooth and accurate, even if they are fatigued. Inspired by biology, methods are designed that imitate the biologic solutions. This is a part of the field of Artificial Intelligence (AI). The goal of this research project is to investigate whether some AI techniques can be applied to control robots that operate in domestic environments.

AI techniques potentially provide some interesting advantages over approaches based on classical control theory. Their ability to adapt to changes makes that they may be able to cope with, for example, dynamic environments and wear. Another advantage is that AI methods are generally applicable: the same algorithm can be used on different systems. Finally, usually there is no need for detailed system knowledge due to their inherent black-box nature.

1.2 Problem statement

The problem is stated as follows:

> Investigate the merits of some Artificial Intelligence (AI) methods in their ability to perform a difficult control task. To benchmark the methods, the goal is to control a real robot arm that is actuated by pneumatic artificial muscles. The algorithm should be given as little system knowledge as possible. The performance is evaluated by means of point-to-point movements.

To be able to assess the performance, the idea was that another student would tackle the problem in the classical way. However, the vacancy had not been filled in time to include a comparison in this report.

1.3 Subtasks in the control problem

Before we elaborate on the problem statement, we first define what we mean by *controlling* a system. Complex movements, like writing with a pen, can be divided in subtasks. In [13], the following subtasks are defined:

i planning

ii trajectory formation

iii inverse kinematics

iv inverse dynamics

v actuator control

Planning deals with the desired movement on a higher level. *Trajectory formation* translates this planning into a path. Both items are not part of our problem, since these are related to path generation. The inverse kinematics deals with the angular rotations of the different joints. This is a difficult problem when the system exhibits a high degree of *kinematic redundancy.* This means that the same movement can be realized by many joint patterns. The construction of the robot arm we use, is such that there is no kinematic redundancy. That leaves the two last items to solve. The task of *inverse dynamics* is to derive the joint movements: rotational velocity, rotational acceleration and jerk. The last item refers to the actual motor commands to control the actuators (muscles). As will be explained in the next chapter about the robot arm, we use two muscles per joint. We say the system is overactuated, since there are more muscles than degrees of freedom. In this context we speak about *muscle redundancy.*

In the remainder of this report, we refer to the last two items when we speak about the control problem of the robot arm.

1.4 Outline

In chapter two we explain the robot arm and formalize the control problem. In chapter three is discussed how the empirical simulation is constructed and analyzed. In chapter four we explain the learning algorithm we apply. In chapter five the experiments are described. In chapter six scalability issues are explained. Finally we draw conclusions and give recommendations in chapter seven.

Chapter 2

Robot arm

In this chapter we explain the robot arm we use. With the help of many Philips engineers, we have created this arm from scratch.

In section one we explain the set-up of the robot arm. Section two introduces the muscles that are used. Next, section three discusses the construction of the robot arm. Finally, section four explains the control problem.

2.1 Basic set-up of the robot arm

Figure 2.1 shows a photo of the robot arm. Figure 2.2 shows a schematic picture. The basic parts consist of the four muscles (1a, 1b, 2a and 2b), the two joints, the spring and the frame.

As can be seen, the muscles come in pairs. The two muscles $1a$ and $1b$ determine the position of the first joint. With the term *joint* we not only refer to the hinge but rather we mean the entire arm that is controlled by $1a$ and $1b$. Analogously, the muscles $2a$ and $2b$ steer the second joint. In this report we only discuss the first joint, with the exception of Section 6 where we also take the second joint into account. When we do experiments with a single-joint arm, we replace the spring by a steel bar to mechanically fix the second joint. Figure 2.3 gives a close-up of only the first joint.

The muscles only generate a pull force. To be able to control a joint, we could choose to use one actuator and a spring. However, we want to be able to control the arm faster and to introduce the challenge of redundancy. Therefore we use two muscles for one degree of freedom: the system is *overactuated.* The setting we use, with two muscles working on either side of the hinge, is called an agonist and antagonist setting, much like the human's biceps and triceps. The resulting torque exerting on the joint determines the system dynamics. The resulting torque is constituted by the forces of both muscles and the torque of gravity working on the arm. We control the pull force of a muscle by controlling

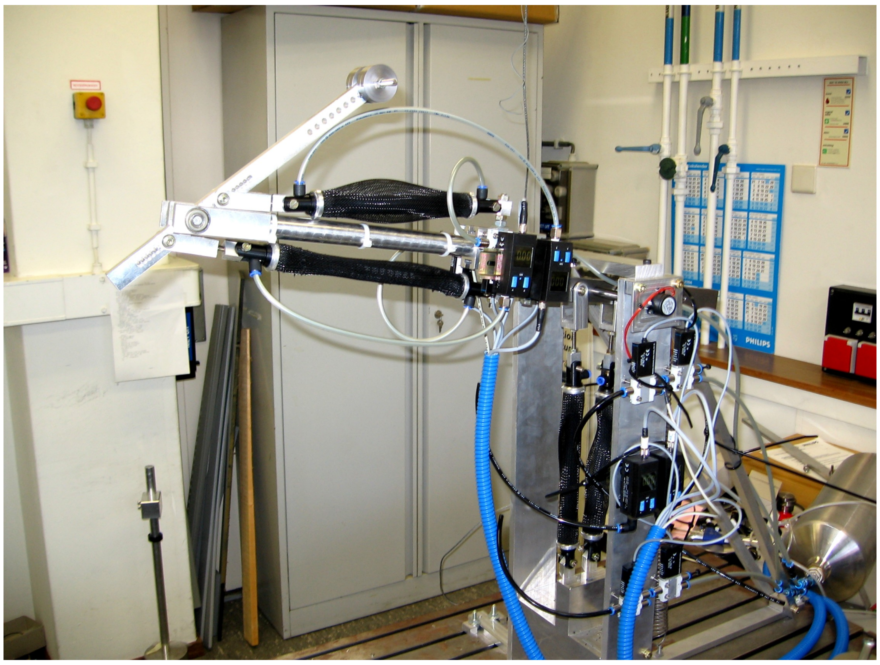
Figure 2.1: Photo of the robot arm.

the pressure in the muscle. The device that enables the control of pressure is called a *valve*.

The rest of this chapter is organized as follows. In section two we discuss the properties of the muscles. After that, section three explains construction of the robot arm and its state variables. Finally, in section four the control problem is formalized.

2.2 Artificial muscles

In this section we discuss in detail all relevant aspects of the muscles. We also point out why this actuator is difficult to control.

2.2.1 Introduction to McKibben actuators

The so-called McKibben muscles are the actuators that are used on the robot arm. These actuators are chosen, because these muscles are suitable for application in domestic environments, as has been shown in [3].

In the literature one can find these muscles under several names: McKibben muscle, Braided Muscles, Rubbertuator, Shadow Air Muscle [6] and Fluidic muscles [9]. These muscles are not exactly the same, but rather are variants of the same idea.

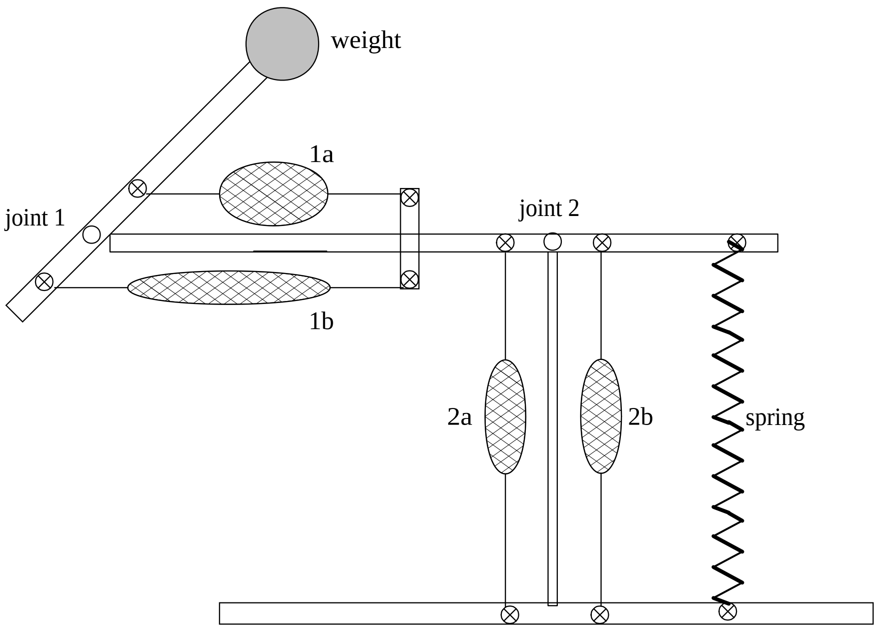

Figure 2.2: Schematic picture of the robot arm.

2.2.2 Operation

Refer to Figure 2.4 for a photo of a decoupled muscle and to Figure 2.5 for a picture of an inflated muscle and a stretched muscle. A muscle is composed of a gas-tight elastic tube surrounded by a braided sleeve (or weave). The sleeve consists of interwoven strands (also called threads). When the tube is pressurized it inflates and presses radially against the sleeve. The stretching of the sleeve causes the fiber curvature to generate a braid fiber tension. This tension is integrated at both end points. This results in a pulling force. The higher the pressure in the tube, the more the sleeve is stretched and hence, the more pulling force is generated.

No pulling force will be generated if there is no pressing contact between the tube and the sleeve. Therefore, the muscle can only be controlled effectively if the tube and sleeve touch. When the pressure in the muscles is low and is increased, a strange effect appears: the tube is actually getting longer (and therefore, generates a small pushing force). This effect can be compared with a balloon that grows in length while it expands.

2.2.3 Properties

The force function of the muscle is not linear. On the contrary, due to its material properties this relation is highly non-linear. The incorporation of these properties into a model is not easy. That is one of the reasons that a system

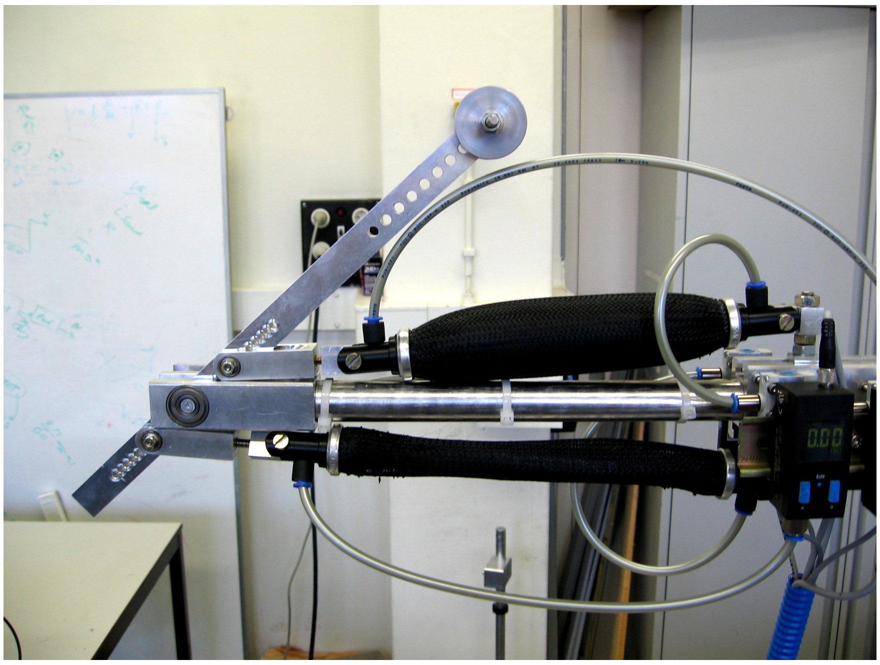

Figure 2.3: Close-up photo of one joint.

actuated by these muscles is hard to control by conventional control theory. Here lays the challenge for AI algorithms. In order to grasp the difficulties, we make an overview of the muscle's properties [33] [3] [7] [6] [16].

Advantages. First we list some advantages. The muscle demonstrates a high power-to-weight ratio. This high ratio is a result of the conversion from pressure to tensional force. A high ratio is useful for a robot arm with multiple joints, where the actuators are part of the moving arm. Another interesting feature is that their behavior exposes an analogy with biological muscles. Namely, they show similar length-load curves. The muscles are compliant, which can be an important advantage for domestic appliances. Furthermore they have a simple design, are flexible (no precise aligning needed), are easy to mount and have low costs.

Drawbacks. However, the muscle also has quite a number of drawbacks. We first name the general drawbacks and afterwards we take a closer look at the more serious problems. To start with, it needs a pneumatic storage for compressed air or CO2, which is not very practical for mobile applications. The life expectancy is not high: the braid fiber wears off at the clamping points. The muscle has a limited displacement, depending on its original length. A typical contraction rate is between 20% and 30%. Energy is needed to deform the rubber membrane, lowering the output force.

Problems. The muscles also exhibit a more serious problem. The muscle is troubled by inherent dry friction. The friction arises from three causes: friction

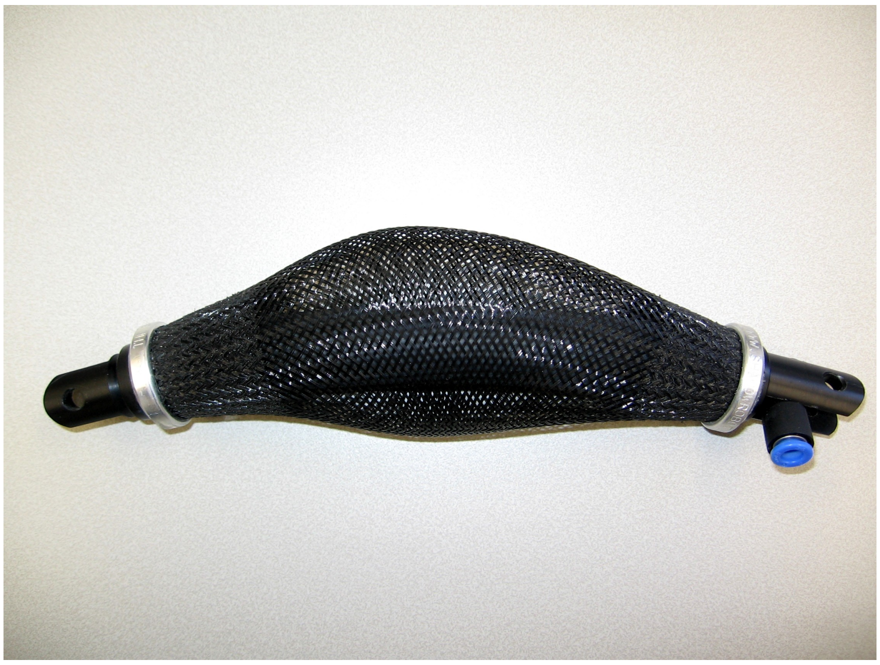
Figure 2.4: Photo of a decoupled muscle.

between the sleeve strands and the tube, friction between the strands themselves and hysteresis inherent in the rubber. The friction results in *hysteresis* and a *threshold pressure*.

Threshold pressure means that a pressure difference needs to be exceeded before the sleeve deforms and hence, before the pull force changes. *Hysteresis* means that the muscle's behavior is dependent on its history. Thus the threshold pressure is one aspect of hysteresis. In theory all previous states of the muscle influence its current state. For the muscle the hysteresis means that a part of its energy gets lost as heat. This also results in a temperature change of the muscle and this affects the muscle's operation.

Force function. To get an impression of the non-linear behavior of the muscle, we give a function commonly used in the literature. This function approximates the force of the muscles. Of course, it neglects effects such as hysteresis. The force function is as follows [7]:

$$F(\rho, \epsilon) = \frac{\pi D_o^2 \rho}{4} \Big(\frac{3}{\tan^2 \theta_0} (1-\epsilon)^2 - \frac{1}{\sin^2 \theta_0} \Big) \tag{2.1}$$

where D_0 and θ_0 represent the diameter and the weave angle at rest, i.e. when the muscle is not inflated and is decoupled. To see how θ_0 is calculated we refer to [7]. Parameter ρ denotes the pressure and ϵ represents the contraction ratio: $\epsilon = 1 - \frac{l}{l_0}$ with l the length of the tube and l_0 the length at rest.

Specifications. Finally, we give the specifications of the muscles that we use on the robot arm. The diameter D_0 is 30 mm, the rest length L_0 is 180 mm.

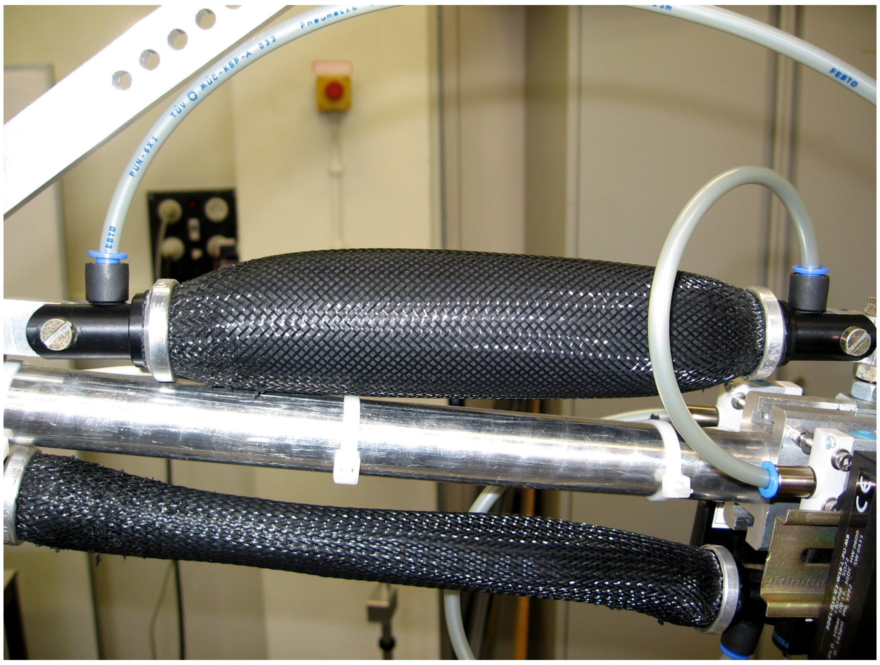

Figure 2.5: Photo of an inflated muscle and a stretched muscle.

Fully stretched it measures 290 mm. θ_0 is 45°. The weight is about 80 gram. It can pull 35 kg at 3.5 bar, the maximum pull is 70 kg. Detailed specifications can be found in [6].

2.3 Construction of the robot arm

The muscles and the valves together determine the operational behavior of the arm. Having discussed the muscles, we now direct our attention to the valves that we have chosen. After that, we determine the state variables. Finally we explain the chip and the sensors that we use.

2.3.1 Valves

In this section we explain the valves that are used to adjust the pressure for the muscles.

There are different types of valves available. A basic choice is between a continuous and a binary valve. We have chosen for a binary valve, a so-called ‘bang-bang’ valve. That means that the valve is either open or closed: air is flowing to (or out of) the muscle or not. Binary valves are cheaper, faster and have a smaller size than continuous valves. With these binary types one can simulate a continuous air flow (like a continuous valve would give), up to certain orders at least. This can be achieved by alternately opening and closing the valves. This technique is called Pulse Width Modulation (PWM) [20]. Drawbacks of PWM are noise and wear.

The amount of air flow is dependent on the pressure difference between supply pressure and the pressure in the muscle. For the requirements of the valve we deal with two rather contradictory requirements. On the one hand, we wish to make the flow steps as small as possible in order to control the pressure very accurately. On the other hand, the time needed to inflate the muscle should not be too large in order to control the muscles rapidly.

The first requirement implies a high maximum switching frequency. The second one demands a high standard nominal flow rate. However, for the flow rate one should keep in mind the volume of the muscles. We estimate that the muscles may contain about 0.4 liter of uncompressed air. A too high flow rate would result in an immediately inflated muscle, despite a high switching frequency. The valve type that we have chosen is a Festo MHE2-MS1H-3/2G-M7-K. The switching frequency is 330 Hz and the flow rate is 100 l/min.

The type of valves we use are members of the high switching family. The price you pay for this fast reaction time is that the valves are unidirectional. Therefore we have to use two valves per muscle. One valve is used to let air flow into the muscle, and the other valve is used to let it out.

Instantaneous control. We can only control the robot arm by adjusting the pressure in the muscles. This adjustment is achieved by switching the valves on or off. The 'bang-bang' valves are limited in two ways. First, they are limited in time: there exists a minimal amount of time that a valve is on or off, because it takes time to switch. Second, they have a limitation in flow: the amount of air flowing to or out of the muscles cannot be regulated. This flow is dependent on the pressure difference between supply and muscle. We conclude that we do not have an instantaneous control of the pressure, rather we rely on the speed of the air flow.

2.3.2 State variables

In this section we investigate the state variables of the robot arm. We aim to find a subset (preferably the smallest one) that is able to represent the state of the system at any given time. Based on this analysis, the sensors are chosen in the next section. We find the state variables by analyzing the equations of rotational motion for the robot arm.

Rotational kinematics. For one joint, the dynamics of the moving bar can be described using the equations of rotational kinematics:

$$\theta_{k+1} = \theta_k + \omega_k \Delta t + \frac{1}{2}\dot{\omega_k}\Delta t^2 \tag{2.2}$$

$$\omega_{k+1} = \omega_k + \dot{\omega_k}\Delta t \tag{2.3}$$

where θ_k denotes the angle of the moving bar with the static bar at timestep k, ω_k is the average rotational velocity between timesteps k and $k+1$ and $\dot{\omega_k}$ is the average rotational acceleration between timesteps k and $k+1$. The timestamps are indicated with k and $k+1$ and Δt is the time between those two timestamps.

Rotational acceleration. To calculate the rotational acceleration $\dot{\omega_k}$, we use the second law of Newton: $F = ma$. Since we work with rotational equations we need the rotational equivalent:

$$\tau_r = I\dot{\omega}_k$$

where τ_r is the resultant torque and I is the moment of inertia. If τ_r and I are known, $\dot{\omega_k}$ can be calculated: $\dot{\omega_k} = \frac{\tau_r}{I}$. The moment of inertia I is dominated by the weight at the end of the arm (refer to Figure 2.3):

$$I = mr^2$$

where m denotes the mass of the weight on the arm and r the radius (the distance from weight to hinge). Both the mass m and the radius r are constant, so the moment of inertia I is constant. That implies that the rotational acceleration only varies with τ_r. This resultant torque τ_r is dependent on the torque that both muscles $1a$ and $1b$ exert and on the torque of gravity.

$$\tau_r = \tau_{1a} + \tau_{1b} + \tau_g$$

Torque is defined as $\tau = Fl$, where F is the force and l the length of the lever. The lever l depends in all three cases on the angle θ the joint makes. For the force of gravity, we know $F_g = mg$. Both g and m are constant and therefore F_g is constant. The forces F_{1a} and F_{1b} depend on the forces that both muscles exert. Because of the hysteresis property, these forces depend on all previous angles θ and pressures ρ_{1a} and ρ_{1b}. Note that the friction force in the hinge is negligible because of the use of bearings. In the remainder of this report, we denote the pressure in muscle $1a$ with ρ_1 and the pressure of muscle $1b$ with ρ_2 (since we only speak about two muscles).

State variables. In summary, θ, ρ_1 and ρ_2 together define the torque τ. Since the moment of inertia I is constant, these three inputs define the rotational acceleration $\dot{\omega_k}$. Looking at the rotational kinematics equations 2.2 and 2.3, this is not enough. If we want to calculate the values for the next timestep, we also need ω. We denote the observations at timestep k as o_k:

$$o_k = (\theta_k, \omega_k, \rho_{1,k}, \rho_{2,k})$$

Because of the hysteresis property, we need all previous angles and pressures. In practice however, we expect that only the last few timesteps will be enough. At this moment we cannot be sure how many we should include. The equations of rotational kinematics show that we deal with a second-order system. We cannot change the rotational acceleration $\dot{\omega_k}$ instantly, since we cannot change the pressures instantly. Therefore, we also need to take the behavior of the air flow in the muscle and pipes into account, increasing the system's complexity with an order of magnitude. We investigate this matter in detail in chapter 3. In that chapter we show how many timesteps should be used and how the state σ_k is defined. The state σ_k should at least include the state variables at timestep k, i.e. o_k.

Conclusion. To conclude, we showed that the state variables can be represented by θ, ω, ρ_1 and ρ_2. This derivation is based on the equations of rotational kinematics. If we would know the force function of the muscles, we could calculate o_{k+1} using numerical integration methods like Runge-Kutta or the Midpoint method [22].

These equations are idealized. The mechanical system is far from perfect. Other variables might influence the system as well. The muscles' temperature might have an impact on their behavior. The temperature rises because of the friction and compression of air. For now we neglect these influences. A detailed analysis on the state variables, based on empirical data, is performed in the next chapter.

Note that we are not after a perfect mechanical system. We want our AI algorithm to cope with noise and influences that are not measured. However, in order to deal with the system the algorithm surely needs to measure the major influences.

2.3.3 Chip

The valves and sensors need to be accessible to the algorithm. To this purpose, we use a chip that acts as an interface between sensors/valves and the PC. In this section, we explain more about the chip.

Microcontroller. The chip we use is a microcontroller of manufacturer `Microchip`, type PIC16F877. The chip is extended with a wrapper called DLP-245PB, developed by `DLP Design`. This composition of chip and wrapper is set as a standard for the Home Robotics project. A few of its advantageous properties are its easy use of the USB connection and its bidirectional pins (flexibility). Since we use Linux Suse 9.3 as operating system, we use the open-source driver from FTDI, called FT245BM332. This driver handles the USB protocol. The PIC microcontrollers have flash memory. This has the advantage that we can reprogram the chip software whenever we like. It also allows to add functionality to the software that is tailored to our needs. We rewrote most of the software, see Appendix A for more information. The circuit board with the chip is shown in Figure 2.6.

Reading sensors and setting valves. In this paragraph we explain in more detail how we physically control the valves and read the sensors, via the chip. But first we give some important concepts of the USB protocol, since this determines the communication between PC and chip.

USB. The Universal Serial Bus (USB) is host controlled. The PC is the USB host and the peripheral device, in our case the chip, is the client. The USB host starts all initiatives, so the client (the chip in our case) cannot send bytes over the USB connection to the PC. The chip has a buffer in which it stores these bytes, and the PC explicitly has to access this buffer. All USB transactions are grouped into timing intervals called frames [17]. For USB 1.1, a frame is a time interval of 1 ms. The client determines the number of bytes that are put in a frame. In USB terms, our chip is a low speed device and

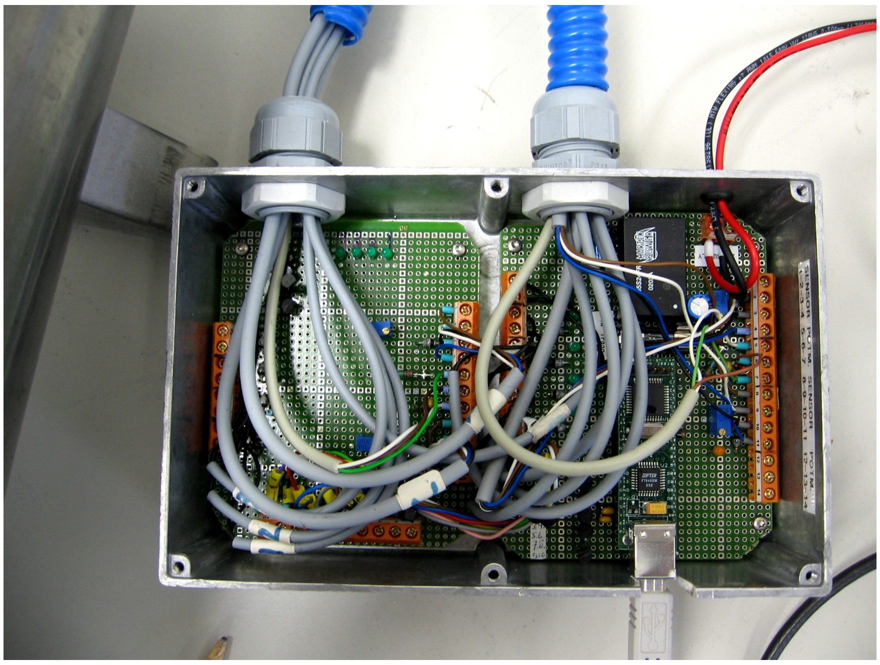
Figure 2.6: Photo of the circuit board with chip.

therefore supports a data rate of maximum 1.5Mb/s, i.e. 1500 bits per frame (187 bytes per frame).

Valves. Every valve is connected to one of the chip's binary output pins, see Figure 2.7. The output is either $0V$ or $5V$. A low output voltage deactivates the valve, which in our case means the valve is closed. A high output voltage opens the valve. All eight valves are connected to the same port. Our modified software has functions to set an output pin separately, but also to set an entire port at once. We use this latter option because this allows to set all valves at once with only one request.

Sensors. The chip has eight analog input pins. Two analog inputs are already reserved, and the other six inputs are connected to the sensors (this is explained in the next section). To read one sensor value, the following steps are taken. First, a request is sent to the chip. Then, the chip reads the sensor value, applies an A/D conversion, and stores the result in its buffer. The A/D conversion takes approximately 3.3 microseconds. Meanwhile, the PC waits 1 ms. After that, the PC again sends a request to read the buffer and the content of the buffer is sent to the PC. The first request takes one frame and the second request requires two frames. We have six sensors, so iterating this process six times would require 18 frames, and therefore approximately 18 ms. With respect to the frequency of the valves and the envisioned calculation time for the controller (in the order of a few milliseconds, we expect), this would be

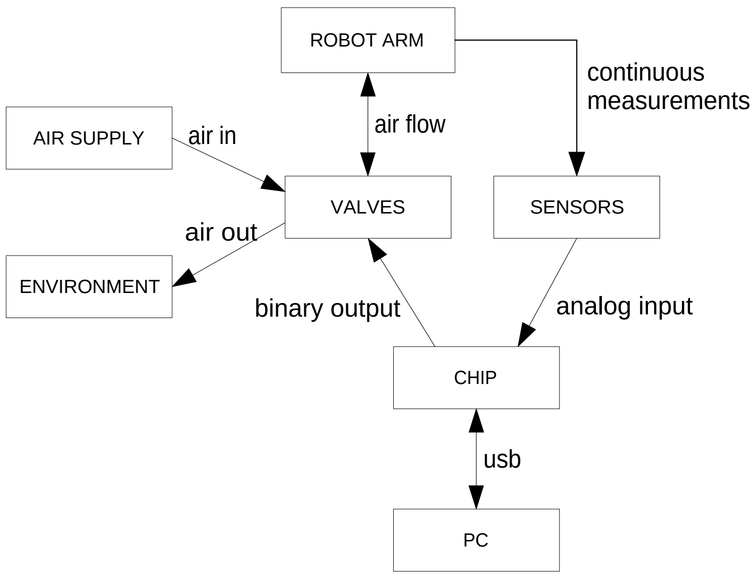

Figure 2.7: Valves and sensors.

a potential bottleneck. Therefore we adapted the software to read all sensor values at once, reducing the number of frames to only three.

The maximum allowed inlet voltage at the analog ports is $5.12V$. The inlet voltage is converted to 10 bits precision, so the inlet voltage is converted to an integer value between 0 and 1023. For each volt there are $1024/5.12 = 200$ distinctive values.

2.3.4 Sensors

We saw that the state variables are represented by θ, ω, ρ_1 and ρ_2. In this section we explain the sensors we use to measure this.

There are three different quantities that we can measure: angle, air pressure and rotational velocity. For the sake of simplicity of the robot arm, we have chosen to discard a velocity sensor. The rotational velocity is simply derived numerically, by approximating:

$$\omega = \dot{\theta} = \frac{\partial \theta}{\partial t} \approx \frac{\theta_k - \theta_{k-1}}{\Delta t}$$

For the pressure sensor we use type SDE1-D10-G2-W18-L-PU-M8 of manufacturer Festo [9]. This sensor has a measuring range between 0 and 10 bar. It gives an electrical output between 0 and 10V. A resistance is added at the output [27]. The voltage as it comes in at the chip is $0.83\frac{V}{\text{bar}}$. As we saw in the previous section about the chip, every volt is divided in 200 values. We effectively have $0.83\frac{V}{\text{bar}} \cdot 200\frac{\text{values}}{V} = 166\frac{\text{values}}{\text{bar}}$.

For the measuring of θ we use a rotational potentiometer (or shorter, a potmeter). This is a simple and cheap device. The potential is between 0 and $5V$. The potmeter is able to measure 270 degrees. So effectively it has $1024/270 \approx 3.8$ values per degree.

When the range of the sensors is higher than the range we effectively use, we can scale the output voltage of the sensors by a resistance between the sensors and the chip. This is a way to scale up the resolution of the measurements. In our case, the stroke of the arm is 90 degrees, so we scaled the resolution to $1024/90 = 11.38$ values per degree.

2.4 Control problem

2.4.1 Specification of the control problem

In this section we give the specification of the control problem. We consider the case with one joint, controlled by two muscles $1a$ and $1b$.

Point-to-point movements. The observed state variables o_k of the robot arm at timestep k is described by:

$$o_k = (\theta_k, \omega_k, \rho_{1,k}, \rho_{2,k})$$

where θ is the angle of the joint, $\omega = \dot{\theta}$ the rotational velociy (the time derivative of θ), ρ_1 the pressure in muscle $1a$ and ρ_2 the pressure in muscle $1b$. In the next chapter we investigate how many timesteps of the state variables and actions we should include in the state representation σ_k.

The state variables θ, ρ_1 and ρ_2 are read by analog sensors and converted to 10-bit digital values:

$$\theta \in [0, 1023], \rho_1 \in [0, 166], \rho_2 \in [0, 166]$$

The pressure does not exceed 1 bar in order to spare the muscles. State variable ω is derived numerically and is continuous. Without system knowledge we cannot calculate its domain. We know that its domain is bounded by the physical system (it is dependent on the maximum force of the muscles and the moment of inertia). Empirically its domain lies in:

$$\omega \in [-4, 4]$$

The unit of ω is the number of θ units per ms.

The goal is to be able to perform point-to-point movements. For every point-to-point movement, we define a goal angle (goal point). We define a goal state as any state that has this goal angle and has zero velocity. The goal of the controller is to bring the robot arm in one of the goal states σ_g. The set of *goal states* Σ_g is defined as all states that satisfy the specified goal angle θ_g with zero rotational velocity:

$$\Sigma_g(\theta_g) = \{\sigma_k | \theta_g = \theta_k, \omega_g = \omega_k = 0\} \tag{2.4}$$

We impose no constraints on the pressures in the goal state. Constraints might be advantageous though. For example, if we prefer medium pressure we might be more flexible for movements in the next tasks of the controller. Stiff muscles are suited for accurate control, but compliant muscles are suited for fast movements.

In practise, we allow a certain deviation from θ_g and ω_g. The muscles are not intended for very accurate control and therefore it would be virtually impossible to reach an exact angle θ_g. Furthermore, we deal with a real system that inherently has sensor noise.

Control. We control the robot arm by changing the pressures in the muscles. At every timestep k we choose exactly one of three possible actions for every muscle: let air in, let air out or do nothing. For a joint with two muscles we thus have nine combined actions. We want to reach one of the goal states σ_g as fast as possible. Expressed in our control problem this boils down to reaching a goal state in as few timesteps (and thus, actions) as possible. For the sake of air consumption, it might be advantageous to specify some criteria on the type of actions that is executed. However, we impose no constraints on it.

On a lower level, these three possible actions per muscle are set by controlling the valves. A muscle has two valves: one that lets air flowing in from the air supply source and one that lets air flowing out to the environment. A valve can be either open or closed. At every timestep k we can set the status of a valve exactly once. The size of the timestep is determined by the delay we introduce (on purpose, to lower the control frequency), the communication with the chip and the calculation time of the control algorithm.

We require that at every timestep k at most one valve of a muscle is open. See Table 2.1 for the coding scheme [1]. Note that if one would allow to open both valves of a muscle, the net air change is not simply zero. The behavior is then dependent on the current pressure. We have three straightforward actions per muscle if we set at most one valve per muscle open.

action	valve air in	valve air out
air in	open	closed
air out	closed	open
nothing	closed	closed

Table 2.1: Valve coding.

In the remainder of the report, we define an *action* as a combined action for both muscles.

2.4.2 Why is the control difficult?

The control of the robot arm is a non-trivial problem, for several reasons:

- **Non-linear**: The muscles generate different forces at different lengths and pressures. This function is highly non-linear.
- **Hysteresis**: Hysteresis of the muscles, also causes temperature influences that affects muscle operation.
- **Muscle redundancy**: In theory, there is an infinite combination of forces that both muscles might exert for the same angle.
- **Rotational velocity**: If the robot arm has a rotational velocity, the lengths of both muscles change. That causes a change in the muscle's tube volume which results in a different pressure that in turn causes another pull force.

[1]This coding schema holds for the software. The actual placement of the valves prevents that both valves can be opened at the same time.

- **Moment of inertia**: In general, if the joint is not appropriately slowed down in time the moment of inertia will cause the robot arm to overshoot its desired angle. To make the problem more challenging, we attached a mass to the joint to let the controller cope with the moment of inertia.
- **Valves**: First, the behavior of the valves is non-linear. Second, the fact that our valves are binary makes it more difficult. The nine combined actions cannot be expressed on a continuous interval, since there is no total ordering. That makes it difficult to use a direct function mapping from a state to one of the actions. Third, the air flow is dependent on the source pressure and the pressure in the muscles. The flow speed cannot be controlled.
- **Accuracy**: The resulting change in θ of one action depends not only on the opening time of the valves, but also on the stiffness of both muscles.
- **Leak**: The muscles and the connections suffer from a leakage.
- **Muscles**: Every muscle behaves differently. Every muscle should be considered unique.

A desirable property of the system is the *stability*: if no action is executed, the state of the robot arm does not change much. Only the rotational velocity and the leak change the state. The inverted pendulum is an example of a non-stable system. The stableness is a nice property: once we reach a goal state we can be sure the system stays there (except for the leakage).

2.4.3 Operation of the assembled robot arm

In this section we explain the control cycle of the robot arm.

1. The sensor values $\theta_k, \rho_{k,1}, \rho_{k,2}$ are read. In Section 2.3.3 we already explained this process.
2. The state variable ω_k is derived numerically.
3. The state representation σ_k is adapted. In any case, the sensor values o_k are included. Depending on the definition of state, the previous action α_{k-1} is also included. The least recent sensor values and the least recent action are removed from the state.
4. Based on state σ_k, the controller makes its calculations and selects one of the nine actions α_k.
5. The valves are set correspondingly to α_k.
6. In order to let the effects of the just chosen action be noticeable, a delay is imposed. Depending on the control frequency we want to obtain, we can change this delay.

Close the valves or not? Note that the valves are only set once in this cycle. We do not close the valves during the time interval where the sensors are read and the action is selected. By the time the new selected action is executed, the sensor values are likely to be changed. This might impose a problem. In this paragraph we explain this issue.

Consider a valve that lets air into the muscle. Suppose we would close this valve during the time interval of reading the sensors and selecting an action. The air flow will not stop instantly at the moment we close the valve. A pressure build-up will occur at the point just before the valve since this air flow 'bumps' against the valve that has just been closed. On the side of the valve a pressure drop will settle in. It takes some time before the pressure inside the muscle stabilizes. Therefore, the measured pressure will vary in the time interval. Moreover, the muscle's force changes as well, changing the angle. If the valve is opened again, these effects may still play a role. In this context we speak about the 'memory' of air flow.

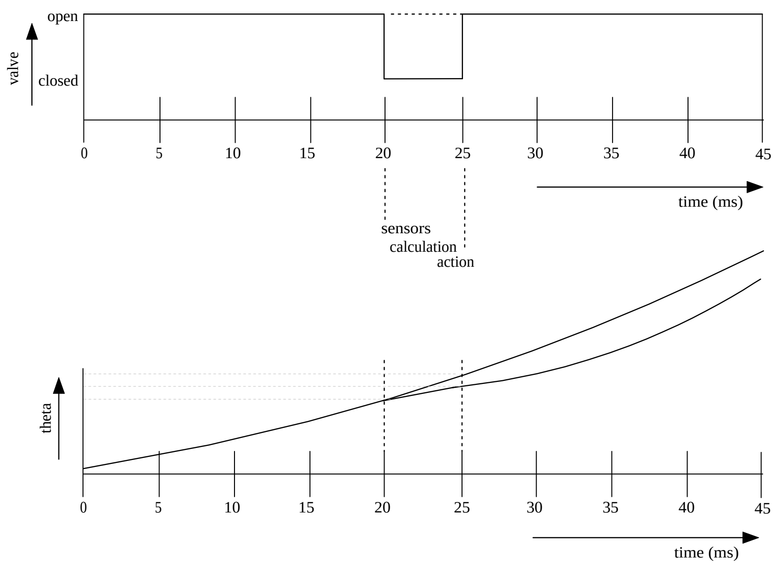

Figure 2.8: Extrapolation effects. Top: Valve control in time. Bottom: An imaginary effect on θ in time.

Refer to Figure 2.8. In the top picture, we open a valve the first 20 ms (this is the delay). Then we reserved 5 ms to read the sensors, select a new action and set the valve open again. In this time interval of 5 ms we have the option to either close the valves or let them unchanged. An imaginary effect on the angle θ is shown in the bottom picture. Even if we close the valve, θ is likely to change, as we just explained. We expect this change to be smaller than if we would leave the valves unclosed. The point we want to make, is that there will *always* be some effects on the sensor values during the time we read the sensors and calculate, even if we close the valves.

If the changes in the sensors are not negligible this might impose a serious problem. The controller bases its new decision on outdated sensor information. Only if the controller has the capacity to *extrapolate* the sensor values it would be aware (in some sense) of the ‘true’ state at the moment the new action is executed, or at least it could take it into account. We will go into more detail about this matter in the next chapter about the empirical simulation.

As we already stated, for now we have chosen not to close the valves. The main reason is two fold: it allows for faster and more smooth control. The robot arm can achieve a higher acceleration. Moreover, it is beneficial if α_k matches α_{k-1}, because an additional valve switch would give rise to pressure drops and pressure build-ups as we already explained.

Chapter 3

Empirical simulation

In this chapter we discuss the empirical simulation we developed. The first section gives motivations to build a simulation. Next, section two explains the construction of the simulation we use. After that, in section three the results are analyzed. In section four the transfer problem to the robot arm is investigated. We end with our conclusions in section five.

3.1 Introduction

Motivations. Before we go into detail about the simulation, we first elucidate our motivations to focus our attention on constructing a simulation. After this has been clarified, the simulation process is discussed in the next section.

There are several considerations to develop a simulation. First we list some of the general advantages:

- **Time**: In general, learning algorithms need a lot of training. This can be very time consuming on a physical system. Since a simulation is not bounded by physical laws this might result in a significant time gain. This is also beneficial for finding the appropriate parameters for the learning algorithm more quickly.
- **Harmful situations**: Harmful situations (if any) on the real system are avoided in simulation.
- **Wear**: Experimenting in simulation does not cause physical wear on the real system. This allows much more testing.
- **Instantaneous control**: In simulation one is able to select the state instantly. This might be advantageous to apply a learning strategy, see Chapter 4.
- **Sharing**: Possibility of sharing the simulation with other people or working in parallel.

Empirical simulation vs. Model. The usual way of constructing a simulation is by making a mathematical model of the system at hand. The use of a model is contrary to the research objective, since we aim to avoid the use of system knowledge. The difficulty to model the muscle's behavior was the reason in the first place to investigate Artificial Intelligence techniques.

Another approach towards a simulation is by the use of observed data. For the robot arm, this would be the pressure sensors and the potmeter. We say that the simulation is constructed by the use of *empirical data.* To make the distinction clear, we coin the term *empirical simulation* in the remainder of this report to distinguish from a model approach. We omit the term *empirical* if this is clear from the context. The use of empirical data, or *observations*, is potentially more powerful than a model, since the latter inherently represents a simplification of the reality.

We list some motivations to prefer an empirical simulation over a model:

- **Modelling is hard**: Modelling the behavior of both the muscles and the valves is difficult [33].
- **No simplification**: An approximating model is a simplification of the reality, where some influences are neglected. Using observed data circumvents this drawback.
- **Generally applicable**: A model is specifically designed for one system. If the robot arm's construction is changed, the model has to be revisited. This does not hold for an empirical simulation, where only the new empirical data is needed.

Of course there are drawbacks as well. An important one is that one loses the sense of 'what is going on' (within some boundaries). A model cannot be perfect, but it gives more intuition and may provide explanations for some effects. Another drawback concerns the difficulty of gathering the empirical data. We elaborate on this in more detail in the next section, where we explain how we construct our empirical simulation.

3.2 Constructing the empirical simulation

In this section we explain how we construct our empirical simulation. We first give the general idea and after that we elaborate on the separate steps.

3.2.1 Building the empirical simulation

First the goal of the simulation is defined. After that, the general idea is explained.

Goal of the simulation. The idea behind a simulation is that it can seamlessly replace the system it tries to predict. The execution of an action α_k in state σ_k brings the robot arm in a new state σ_{k+1}. This latter state can be constructed from o_{k+1}, σ_k and α_k. Therefore, a simulation that predicts the mapping $(\sigma_k, \alpha_k) \to o_{k+1}$ suffices.

We define a *transition* as the change-over from a state σ_k to the next sensor values o_{k+1}, under the influence of an action α_k.

General idea. Assume that we already have a controller that operates on the robot arm. That is, there is already an algorithm that chooses the actions to carry out on the robot arm. For the moment we are not interested in the working of this controller, all we care about is that it provides us with transitions from one state to new sensor values, based on its action decisions. To build our empirical simulation, we store all these transitions in a *database*.

When we decide we have enough transitions in our database, we filter our database. That is, we make sure the filtered database represents a good distribution of transitions. After the filtering has been performed, we have a database with explicit input-output examples at our disposal. The *input* is the tuple (σ_k, α_k) and the *output* is the next observation o_{k+1}. To come to our empirical simulation we fit a function to these input-output relations. Usually we speak about a *function approximator* instead of a function fitter. In this context, the output is referred to as the *target* and the fitting procedure is often called *training* or *learning*. The use of a function approximator implies that this simulation is only suitable for non-stochastic systems (since a function is by definition a mapping from an input to exactly one output).

To summarize, the empirical simulation is constructed as follows:

1. Collect transitions $(\sigma_k, \alpha_k, o_{k+1})$ in a database, using an existing controller.
2. Filter the database to get a good transition distribution.
3. Apply a function approximator with input (σ_k, α_k) and target o_{k+1}.

Now that the general picture is clear, we study each of the subparts in more detail.

3.2.2 Collect transitions

To generate transitions, we need an existing controller. This might be a controller upon we are trying to improve, a joystick controlled by hand, some predefined trajectories or the learning method we wish to use.

Sound states. We immediately have seized the main problem of our approach. Namely, that the controller ought to visit a 'representative' part of the state space. The term 'representative' causes the trouble. A part of the state space is valid, but will never be encountered in practice. We coin the terms *sound* and *unsound states* for this matter. For example, in case one muscle exerts full force whereas the other barely exerts a force, a valid state would be that the moving bar is more tilted towards the forceless muscle. This is a valid state, but it will never be encountered unless an external force is applied. This we would call an *unsound state*. Since we do not assume external forces, these unsound states are of little interest to us. The set of sound states is constituted by the effective working area of the robot arm. In general, we do not know what part of the state space is sound. Knowing this would require a considerable analysis of the system's behavior, violating our research aim. Therefore, a lack of parts of the state space in the database may be either due to the controller (when the state is sound but has not been encountered) or it might concern unsound states.

Collect sound states. Collecting sound states is a very important phase in the process of making an empirical simulation. This collection constitutes the basis of the empirical simulation. Fortunately, in case of the robot arm we are able to find the sound states in a more or less structured way. In this paragraph we explain the strategy.

The simulation's input consists of the state σ_k and the action α_k. Ideally, we want to measure the influence of all nine actions on all states. The most important part of the state is constituted by the most recent observations θ_k, ω_k, $\rho_{1,k}$ and $\rho_{2,k}$. Therefore, we only focus on these four values and the nine actions.

First we explain how to start in random begin states with zero velocity. The variables that we can influence most easily are the pressures (since we control the valves). The angle is mainly a result of the pressures and the previous angles (refer to Equation 2.1). In order to visit many states with a zero velocity, we choose to first deflate both muscles totally. After that, we generate two random values in $[0, 1200]$ that denote how many milliseconds each muscle is inflated again. Finally, we wait some time until the oscillations are damped. The deflation part tries to minimize the correlation with the previous state. The two random values that represent the inflation are meant to assure that many sound combinations of pressures and angles are visited.

We still miss two dimensions: the velocity and the effects of the actions. We solve this by executing two kinds of action sequences: either a sequence with an approximately zero velocity or a sequence with a non-zero velocity. We randomly decide what sequence is executed.

A sequence that focuses on zero velocity consists of uncorrelated random actions. The other sequence incorporates velocity by executing actions multiple timesteps. The actions are chosen randomly. Another random value determines how many timesteps in a row these actions are executed in the sequence.

For both sequences we need to make sure that many sound parts of the state space are visited. We solve this by starting from a randomly generated begin state. By iterating these random begin positions and sequences often enough, a large number of sound states with all nine actions is visited. In practice, we let the robot arm iterate this procedure for two hours, effectively storing $200,000$ transitions in the database. We use 200 actions per sequence.

3.2.3 The database and the filtering

We store all transitions in a database. To avoid data redundancy in the database, we choose to identify the combination of `(sequence(k), step(k))` as *primary key*. Then, it suffices to store the following fields in the database table:

$$\texttt{database field = (sequence(k), step(k)}, o_k, \alpha_k)$$

The use of the sequence number and action number makes the database independent from the number of timesteps that we happen to choose for the state. The transition set (that uses σ_k instead of o_k) can always be reconstructed from the database. Before we apply the function approximator to this set of transitions we first filter this set.

Reasons to filter the transition set. The goal of the filtering process is to get an (approximately) uniform distribution of transitions (to get a good example set of the system behavior). We want a good distribution for two reasons. First, to discard redundancy in the transition set (unnecessary time-consuming double states). Second, it is a way to abstract from the process of collecting the transitions. This process is not perfect and most likely is biased by incorporating some states more often than others. We want the function approximator to focus on all states equally well, not only on the states that happen to occur in the transition set as a result of the bias.

We can begin the filtering procedure when we have 'enough' transitions. For the moment we leave it as an open question how many transitions will be 'enough'. We now discuss the filtering procedure.

Filtering procedure. A good distribution of transitions boils down to a good distribution of inputs (σ_k, α_k). Some states σ_k are not encountered. As we saw, this can be due to the controller or it can be an unsound state. Since we cannot know the cause of their absence, we treat the input space in an unbiased manner. The input space is discretized by overlaying a uniform grid. If the state consists of observations and actions from multiple timesteps, this would result in a very high dimensional grid. Because of the (expected) high degree of redundancy of the multiple time steps, in combination with the non-encountered unsound states, the input space is very sparsely filled. Therefore we only discretize the transitions to the observation o_k and action α_k at timestep k. This results in a $5D$ uniform discretization. The idea of the filtering is to select at most n (an integer that is specified a-priori) transitions that correspond with every grid element. Selected transitions are assigned to the filtered set. Some grid elements may contain n transitions, others may contain a few or zero transitions.

Ideally we wish that this selection process represents a 'typical' distribution. However, such a process would be computationally very demanding. It is more efficient to randomly select these n transitions. By choosing a sufficiently rich discretization this results in a good distribution.

The whole filtering process can be done efficiently by traversing the transition set twice. In the first iteration, we create a uniform grid that we call *count*. Every transition k belongs to one grid element i of *count*. After the iteration, $count(i)$ represents how many transitions are associated with grid element i. In the second traversal, *count* is used as follows. Let transition k belong to grid element i and n be an a-priori defined number that denotes how many transitions may belong to a grid element. If $count(i) \leq n$, transition k is included in the filtered set. If $count(i) > n$, the probability of including the transition is $\frac{n}{count(i)}$.

To put things in perspective, we give the values that we used. We divide the (o_k, α_k) space as follows: $(|\theta|, |\omega|, |\rho_1|, |\rho_2|, |\alpha|) = 8 \times 8 \times 8 \times 8 \times 9 = 36,864$ grid elements. We use $n = 5$ and thus have an upperbound of $184,320$ values in the filtered set. In practice, it takes half a minute in MATLAB to filter a set of $200,000$ transitions. A typical filtered set consists of about $40,000$ transitions.

Previously we stated that we may start with the filtering after we have gathered 'enough' transitions. But how should we interpret 'enough'? This question is difficult to answer, since we do not know a-priori how many sound states we might expect and when the current controller has encountered 'enough' of them. A possible strategy is as follows. We may filter if we have about, say, $200,000$ transitions in the database. To check whether the original $200,000$ transitions had been enough, we may collect $200,000$ more transitions. We then filter the whole set of $400,000$ transitions and check if there are any significant differences in this new transition set. If many more bins are filled, we know the distribution has improved and that we need to gather more transitions. If not, we may assume we have most of the sound states.

3.2.4 The function approximator

The last step in the procedure is to train a function approximator on the explicit input-target relations. The goal is to construct a function fit that predicts the next sensor values o_{k+1}, based on inputs (σ_k, α_k).

Various methods can be applied for the function fit. For example, neural networks or a least squares fit. In order to make a decision on what method to take, we first discuss some aspects related to function approximation. After this has been made clear, we discuss our design decisions regarding the function approximator.

General theory. In the remainder, we abbreviate the term function approximator with *FA*. With *training* an FA we express a way to learn a mapping f from input variables to output variables. In our case this would be $f(\sigma_k, \alpha_k) \rightarrow o_{k+1}$. The input-output or input-target pairs are known as *training pairs.*

In general, we have two criteria we would like the FA to fulfil: a good performance and a broad generalization.

The performance of an FA is usually measured by the *mean squared error*, calculated over all input-output relations that are used for the training procedure. That is one of the reasons why we filter the transition set before training an FA on it.

With *generalization* we express the demand that the function should also give 'reasonable' outputs for inputs it has not explicitly been trained for. For example, suppose the function has trained on input-output pair (a, b) and that we now consider the function mapping for a pair (a', b') the function is not explicitly trained for. Intuitively this generalization means that output b' should not deviate much from output b if a' is close to a. Of course, this generalization depends on the degree of correlation of the inputs.

An important notion in this discussion is *overfitting.* This refers to the case where the function performs very well on the dataset it has been trained on, but has a poor generalization. It behaves like it has memorized the entire dataset, but not learned the underlying relation. In our case, where the filtered dataset contains about 40,000 transitions, the chances of overfitting are quite small. First of all, this is due to the high number of transitions. It would be quite an accomplishment for an FA to memorize all transitions quite well (for an FA that only uses a small amount of memory at least). Second, the FA trains on a well distributed dataset (because of the filtering), so we know that all transitions are rather different. If a large subset of the transitions would be about the same, then the probability of overfitting in these areas is higher.

There are various ways in which an FA can be trained. One difference comes from the way the FA uses the training set. If the FA would use the filtered database as a whole, it would be referred to as *batch learning*. Another approach is called *incremental learning*, where one applies one training pair at a time.

Another distinction in the training method can be made between online and offline learning. With *online learning* we refer to the case where the FA is trained immediately when a training pair becomes available. In *offline learning*, the FA is trained after all training pairs have become available. Using offline learning one can apply both incremental and batch learning, whereas with online learning one can only employ incremental learning.

For neural networks, an important aspect to take into account with incremental learning is a phenomenon called *unlearning*. All weights together determine the output of a neural network. The learning algorithm tries to adapt the weights in order to decrease the MSE over the input-target pairs (training pairs). However, a weight change for one training pair can be disadvantageous for another one. That is one of the reasons the learning rate is set to a low value. Unlearning occurs when certain inputs are not encountered frequently while other training pairs make weight changes that are disadvantageous for these inputs. For more information about neural networks, we refer to [25] for an introduction and [11] for a more thorough treatment.

Design decisions. Now we direct our attention to our design decisions regarding the FA.

We have chosen to learn an empirical simulation *offline* (as opposed to online), due to several reasons:

- **Easier**: If we would have chosen for online learning because we wanted to use the simulation immediately, we would have to keep track of what (σ, α) pairs already have been learned and what pairs have not been encountered yet. This is difficult, because one has to come up with a criterium where the border between known/unknown is.
- **Batch learning**: There are more advanced training algorithms available for batch learning, since these algorithms can make use of second-order information to decrease the error over the entire batch. Methods that use incremental learning use one transition at a time, and can therefore only use the gradient (to see in what direction the error of that transition decreases the most). Moreover, the order in which the transitions are used matters for incremental learning.
- **Tuning**: Storing the transitions enables to learn a simulation multiple times to find the right parameters for the FA. We can train over and over again and easily replace the current empirical simulation.
- **Filtering**: As we already explained, filtering of the dataset has important advantages. When using offline learning, we can filter the batch a-priori in order to get a good distribution and obtain a better generalization.

As FA we choose to use a feedforward neural network. As a training algorithm we use batch learning with the Levenberg-Marquardt algorithm. This algorithm has the best performance/speed ratio among all neural networks that are available in the Neural Network Toolbox from MATLAB, according to the tests of MATHWORKS [8]. We also tried a Least Square Fitting. Since we obtained slightly worse results and because it took much more time, we stick to the neural network approach.

An advantage of a Least Squares Fit, is that the weights can be analyzed. This gives a clue about what values in the state are important, and which values can be omitted. With NNs the investigation of the relative contributions of the values is not straightforward.

We use three function approximators to predict the sensor values of timestep $k+1$. One to estimate θ_{k+1}, one for $\rho_{1,k+1}$ and one for $\rho_{2,k+1}$. For ω_{k+1} we can choose to either use a FA or derive it from the predicted θ_{k+1}. We choose the latter option, and fix Δt to 33 ms. When we speak about our empirical simulation, we actually deal with these three function fits and the derived ω_{k+1}.

3.3 Analysis of the empirical simulation

In this section we introduce tools to analyze the performance of our empirical simulation. The goal is to assess to what extent the empirical simulation matches the real observed values, i.e. the real robot arm.

Another goal of the analysis is to investigate whether the state as defined so far provides enough information to find a prediction. If very powerful function fitters perform poorly, it is very strong evidence that the state does not contain enough information to make a good fit possible. An RL agent cannot reliably base its actions on such a wrong state prediction.

The empirical simulation is a fit on the filtered set of transitions, i.e. $(\sigma_k, \alpha_k) \rightarrow o_{k+1}$ for all transitions k. For every transition we have the real observed values o_{k+1}, but also the simulated (predicted, fitted) equivalent. Since the simulation is a function of σ_k and α_k we write $sim(\sigma_k, \alpha_k)$. For convenience we abbreviate this notation to sim_{k+1}, to denote the simulated values for $k+1$:

$$sim_{k+1} = sim(\sigma_k, \alpha_k) \tag{3.1}$$

As already mentioned, the simulation is distributed over three function fits and the derived ω_{k+1}. Unless explicitly stated otherwise, we discuss the general case and denote it with sim_{k+1}.

The analysis tools we introduce compare the simulated values with the observed values. The quality of the *fit* can be benchmarked on transitions from the *filtered set*. The quality of the fit as a *simulation model* should be benchmarked on transitions from the *generalization set*. This latter set contains all transitions from the database that were not included in the filtered set.

Residue. As a first straightforward approach, we calculate for every transition k the error in the simulation:

$$res_k = sim_{k+1} - o_{k+1} \tag{3.2}$$

In Figure 3.1 an example of res is given for θ. The units on the x-axis denote the transitions k. The y-axis denotes θ-units. The black dots represent

θ_{k+1}, the blue dots represent sim_{k+1} (for θ) and the residue is plotted in red dots.

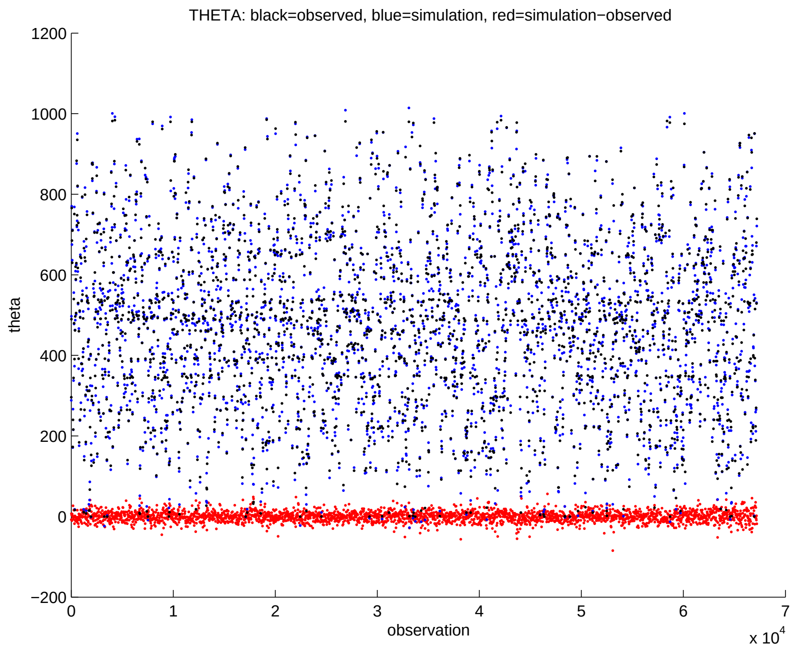

Figure 3.1: Residue of θ.

In the ideal case the residue is 0 for all transitions. However, this plot does not reveal much information. Some transitions have a relatively high residue, whereas others have a residue of almost 0. We want to discover what kind of transitions have a relatively high residue.

We are not limited to only plot the bare residue against the transitions. We obtain more information if we *classify* the transitions on the x-axis. A classification can be based on our expectations, our intuition or just on our common sense. A classification means that the x-axis does not represent single transitions anymore, but rather sets of transitions. If we discover a so-called *trend* in the plot, we have revealed a certain relation between the residue and the classification. When the residue plot reveals a trend, we can use this information to take measures to decrease the residue error.

For example, we expect a relation between ω and θ. Namely, if the rotational velocity is high, successive states are further apart. Furthermore, the exact moment of reading the sensor value θ becomes an important factor since Δt is not exactly constant. Refer to Figure 3.2, where the residue of θ is plotted against ω. This figure indeed shows a trend. However, in this case there is not much we can do to improve the residue at higher velocities.

Another possibility would be to classify the states to *time*. If there exists a trend in the residue that is related to time, we know other influences play a role, for example the temperature.

Isoplot. If we take a step back, we realize that the purpose of a simulation is to predict a *change* in the state variables. That leads to another method to

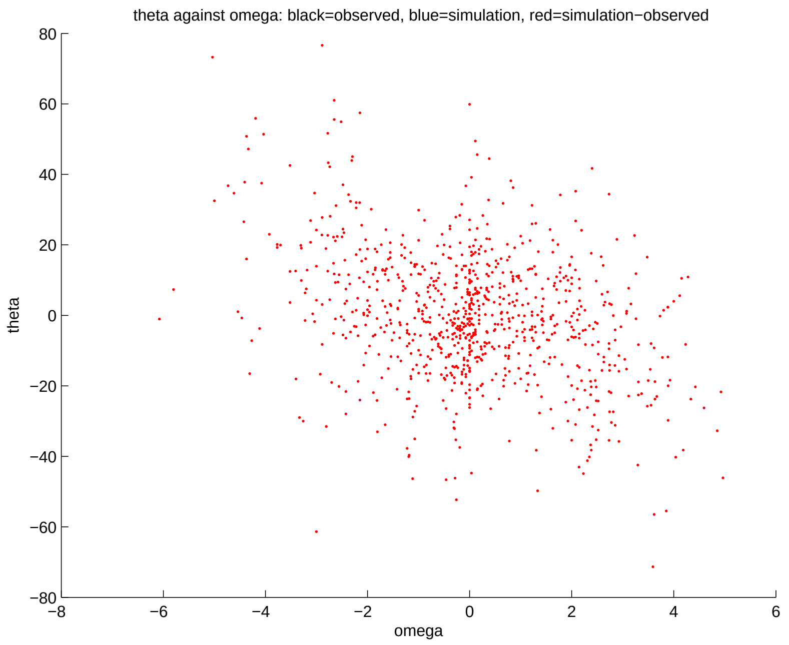

Figure 3.2: Residue of θ plotted against a classification in ω.

analyze the performance. We can plot the observed change in a state variable with respect to the simulated change. We call this an *isoplot*:

$$\Delta sim_{k+1} = sim_{k+1} - o_k \tag{3.3}$$
$$\Delta o_{k+1} = o_{k+1} - o_k \tag{3.4}$$

The term 'iso' means 'the same', like in the terms 'isobar' and 'isotherm'. See Figures 3.3 and 3.4 for examples of a good and a bad isoplot. The 45°-line is the ideal line, where $\Delta sim_{k+1} = \Delta o_{k+1}$. In the figures, the x-axis contains the values for the real observed change Δo_{k+1} and the y-axis has the values for the predicted change Δsim_{k+1}. The transitions are classified on both axes. For example, on the line $x = 0$ all transitions reside where the real observed change had been 0. All red dots on this line represent the predicted change of the simulation.

The broader the spread in an isoplot, the worse the simulation. We call this the *bandwidth*. If the orientation of the scatter of points is not 45°, there is a structural error in the simulation. That means the observed change is biased towards a structural deviation on one side of the 45° line. Figure 3.4 gives an example of ρ_1.
To visualize the distribution of the bandwidth, colors may be used to indicate how many points are used. Refer to Figure 3.5 for an example.

Use of the analysis tools to discover errors. The use of residues and isoplots reveals much information about the quality of the simulation and helps to

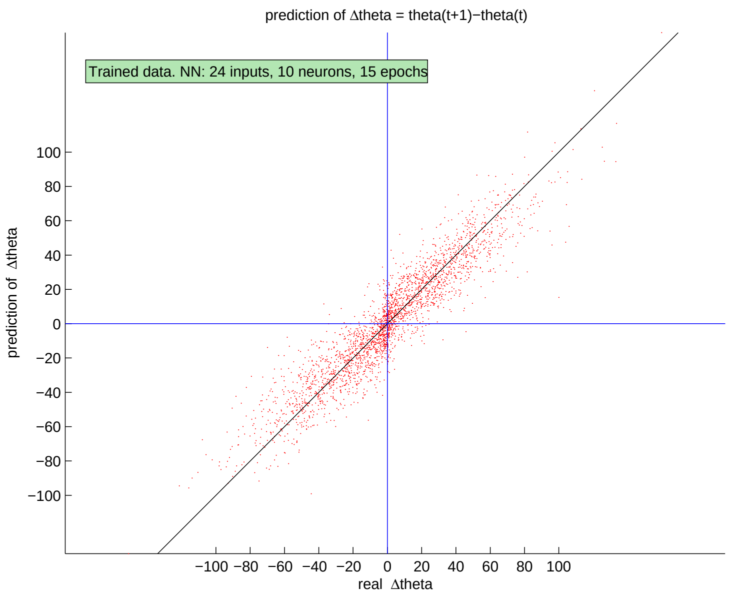

Figure 3.3: Example of a good isoplot (θ).

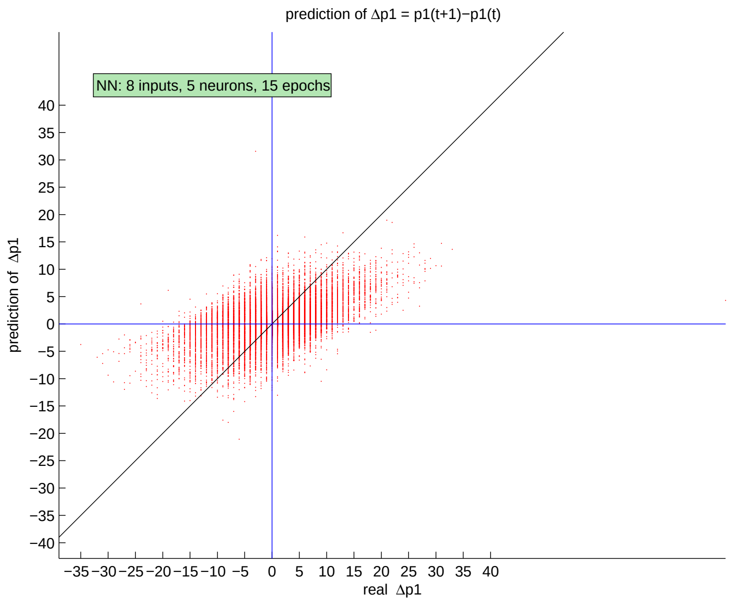

Figure 3.4: Example of a bad isoplot (ρ_1).

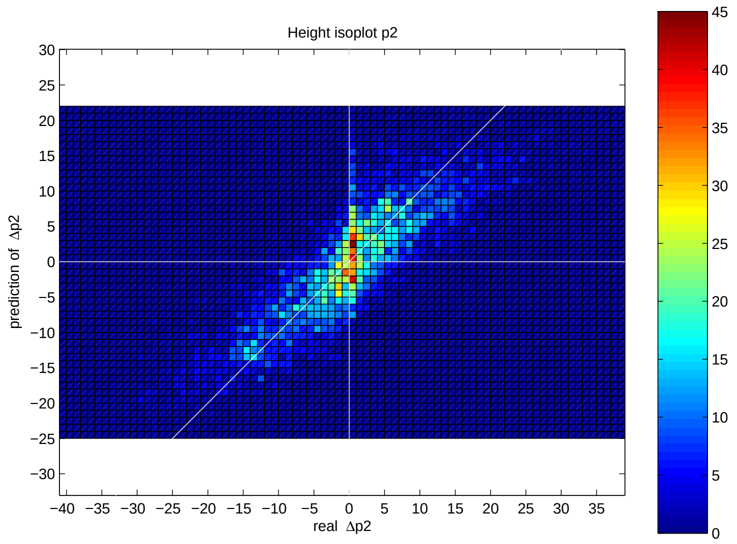

Figure 3.5: Example of a colored isoplot (ρ_2).

explain why and where the simulation predicts the wrong values (error sources).

A simulation performs poorly if the orientation of the scatter is not 45° and if the bandwidth is broad. There might be two factors that cause poor simulation results:

1. Bad fitting of the FA.
2. The data does not contain enough information to obtain a reasonable prediction.

The first cause can be solved by trying several neural networks with different amounts of hidden neurons and different training algorithms. Another function fitter like a least square fitter is also an option. Our experience is that most function approximators give analogous results. If all yield the same bad fitting, we might conclude that the reason of the bad fitting is not likely to be found in the function approximator.

In that case, we should focus on the second point: the data does not contain enough information. This might be due to several causes:

- An error in the sensors: too noisy or a systematic error.
- A state variable is lacking.
- The sensors and state variables are right, but information from previous timesteps is necessary (history).
- The issue about opening or closing the valves, as explained in Section 2.4.3.

In our case the second point (bad data) was indeed the cause. By use of the residues and isoplots we revealed many errors. The errors rose because of various causes: sensor errors, chip errors, driver errors, Linux problems, software implementation errors, database errors and errors in the construction.

In Appendix A we explain how we discovered and solved these problems. Note that the residues and isoplots do not reveal all error sources (see Appendix A for more information).
The conclusion of this analysis is as follows:

- Indeed we may leave the valve position unchanged during the reading of the sensors and the action selection process. The simulation is able to extrapolate the effects in o_{k+1}.
- The sensors indeed have quite a big noise factor, but this is not too problematic. In the appendix we also give a solution to reduce the noise.
- The rotational velocity cannot be predicted accurately. Apparently, the transitions do not contain enough information in order to enable a satisfying ω prediction. Possible causes:
  - The system construction is not stiff enough and the eigenfrequency is too low.
  - Indeed a velocity sensor and/or acceleration sensor is necessary.
  - The numerical derivation of ω is too poor.
  - The severe wear that the potmeter shows. This is due to the installation of the sensor on the system.
  - The inclusion of Δt in the state representation improves the isoplots. The drawback is, that this input cannot be used in simulation. A possible solution is to calculate the mean and standard deviation from the database, and use this to fluctuate Δt.
- We do not expect that a significant state variable is lacking.
- Indeed history is necessary, see Equation 3.5.

As a result of the analysis in Appendix A, we define state σ_k as follows:

$$\sigma_k = (o_k, o_{k-1}, \alpha_{k-1}, o_{k-2}, \alpha_{k-2}) \tag{3.5}$$

Quality analysis. The previous paragraph used the residues and isoplots in order to reveal and solve problems. Now that most problems are solved, we focus on the quality of the simulation.

The residue plot and the isoplot give a graphical overview of the quality of the simulation. We can also apply statistics on the simulation errors. In particular we are interested in the standard deviation of the error.

$$std_{sim} = \sqrt{\frac{1}{N}\sum_{i=0}^{N}\left(res_i - \overline{res}\right)^2} \tag{3.6}$$

with $\overline{res}$ the mean residue. A good fit should have a mean residue of almost 0.

We want to know the influence of the simulation error over multiple timesteps. The error between two timesteps might be small and is not of any concern, but the influence of a small error on every step in a sequence might deviate the eventual outcome significantly. We want to know the *error propagation.*

For the moment, assume the state as defined in Equation 3.5 captures all state information. Further assume that the observed data contains Gaussian noise. This noise can be due to sensor noise, electrical noise, differences in Δt, etc. Then we can assume that the simulation error follows a *Gaussian distribution*, with the standard deviation of the error as defined in Equation 3.6.

Now suppose the errors are *uncorrelated*, i.e. the error at timestep k has no relation with the error at $k-1$. Then we may apply the known equation that adding two Gaussian distributions yields a Gaussian distribution, with the variances added. Extending this equation to x timesteps gives the following result:

$$std_{sim}(x) = \sqrt{x(std_{sim})^2} \tag{3.7}$$

Equation 3.7 shows that the standard deviation of the error grows with the square root of the number of timesteps. This is the expected *error propagation.* Of course, this result should be considered a rough estimation. The state representation is not likely to be perfect. Also, the function fit is not perfect and introduces an error.

The simulation from the isoplot of Figure 3.3 has a standard deviation of 10. The control frequency of this data had been 30 Hz. The standard deviation of the error after 30 steps will then be $\sqrt{30 \cdot 10^2} = 54$. In degrees this is about $5°$. Taken into account that most movements can be done in one second, this seems not too problematic.

In this paragraph we showed that the standard deviation of the simulation error increases by the square root of the number of timesteps. We showed that the results are accurately enough to be useable for a controller.

3.4 Transfer

In this section we discuss the transfer problem to the real system. With the *transfer problem* we mean all issues related to the gap between simulation and real system.

In the remainder of this report, we use the term *agent* to refer to a learning controller.

Noise. The addition of noise to the simulation's predictions might improve the transferability. One way to see this, is to realize that the simulation is a *function.* That is, it maps an input to only one output. On real systems there is always a small deviation due to noise. But there is a more striking reason to deliberately insert some noise. An agent adapts its strategy according to the predictions of the simulation. If it relies heavily on these predictions it will not learn to cope with (small) deviations. Since the function fits undoubtly introduce a small deviation, namely an error, the insertion of noise seems necessary. The question remains: how much noise should we insert?

In [10] a structured investigation to the influence of noise is performed. They trained agents in simulation, with various amounts of noise added to the simulation's results. The agents were 'transferred' to another simulation, also with various amounts of noise. The conclusion was that the addition of noise can greatly improve the transferability robustness of an agent, but that too much noise needs much more training with only a marginal effect on transferability.

[10] uses uniform noise. We propose to use Gaussian noise. We explain this choice as follows. With every neighborhood of (state, action) pairs, there will be a neighborhood of next sensor values. The state is not perfect, and the function fit cannot be perfect either. Therefore, the fit has to find a tradeoff. It tries to place its predictions such that the MSE is minimized. Suppose the training set is representable for the state space, and that the fit does its job right. Then there is a high probability that the fit's prediction resembles the real outcome. The bigger the difference in real outcome and prediction, the lower the probability the fit would have chosen this prediction (since this would increase the MSE).

Furthermore, [10] expresses the amount of noise in percentages (absolute percentage of the domain, not a relative percentage of the predicted change). For example, a noise of 1% of θ would mean $0.01 \cdot 1024 \approx 10$ θ units. However, we propose to relate the noise to the simulation's standard deviation of the error. This way, we let the agent experience the simulation's uncertainty (i.e. its error).

To summarize, we insert Gaussian noise with a standard deviation equal to the standard deviation of the error.

We let the amount of noise also depend on ω. There are two reasons for this choice. First, if the simulation predicts very little movement (small ω), the insertion of noise disturbs this prediction. Second, the simulation predicts big changes in θ, the probability of errors is bigger. The residue plot of θ against ω supports these theoretical reasons, refer to Figure 3.2. We implement this observation as follows. If $\omega > 1$, the noise is as we discussed. If $\omega < 0.2$, the noise is reduced to 20% of the standard deviation of the error. For $0.2 \leq \omega \leq 1$ we linearly interpolate.

Correlated noise. Earlier we assumed that the errors between two timesteps are not related. However, this is not realistic. Neighboring states will have neighboring predictions (due to generalization) and therefore they will have similar errors. Moreover, similar states that occur abundantly in the training set are likely to have smaller errors. This also implies a correlation. These states have smaller errors since the function fit tries to minimize the Mean Squared Error over all transitions in the filtered set. Note that the filtering procedure is a way to diminish this effect.

An easy way to express this correlation is by applying a moving average of the noise:

$$std_{ma}(k) = \frac{n-1}{n} \cdot std_{ma}(k-1) + \frac{1}{n} \cdot noise(k) \tag{3.8}$$

where n is the number of timesteps we want to correlate and $noise(k)$ represents the noise that is generated at timestep k.

The value of $std_{ma}(k)$ serves as the noise we inject at the simulation's prediction. Therefore, we want the standard deviation of this distribution to be of the same magnitude as std_{sim}. In Appendix B.1 we derive the following equation:

$$std_{ma} = \frac{std_{noise}}{\sqrt{2n-1}} \tag{3.9}$$

Equation 3.9 shows how to set the standard deviation for *noise*. In our case, we expect a correlation between 5 consecutive timesteps. Therefore, if we want to set $std_{ma} = 10$, we need to set $std_{noise} = 10 \cdot \sqrt{2 \cdot 5 - 1} = 30$.

Invalid predictions. The simulation is as good as its training set. The idea is that the simulation generalizes over parts that it did not explicitly train for. But what happens if the states become unsound? The predictions might be very erroneous in these states. This is potentially a big problem.

If the simulation predicts a whole sequence of states, the predicted states are deviating more and more from the state the robot arm would give. In case the states in this sequence remain sound states, this is not a problem, since the simulation has been trained for them. But it gets worse when the states start to become unsound states.

There are various reasons why the sound states can become unsound. Among them are:

- The insertion of noise.
- Prediction errors of the simulation.
- The fact that we use different function fitters for each state variable. Their predictions are not related.

The first item is necessary for transferability. The second item is inevitable. The third item might be solved by using one function approximator. For example, one NN with four outputs.

This problem can be monitored as follows. Recall we use a filtering before the function fit is applied. The $5D$ grid contains the number of transitions for each grid element. We display during simulation for every state the value of the associated grid element. We empirically concluded that the problem did not occur significantly.

Another solution might be, to synchronize the state with the database. That is, replace the state with the most similar state from the database. This can be done every 10 states, for example.

3.5 Conclusion

In this chapter we introduced an empirical simulation, based on real robot observations. The simulation is constructed in three steps. First, system data is acquired, by storing transitions in a database. Second, the database is filtered in order to yield a good distribution. Third, a function approximator finds a mapping from (state,action) pairs to the next state.

We proposed tools to analyze the simulation. This analysis revealed many system errors. Furthermore, the analysis also allows to identify the necessary state variables in the state.

We explained how an agent can be transferred to the robot arm. The transfer can be improved by adding noise to the simulation's predictions. This noise can be correlated with previous predictions.

The advantages of this approach are as follows. First, no modelling is needed. Second, there is no simplification because real observations are used. Finally, the method is generally applicable. If the construction changes, the method of constructing the simulation stays the same.

The drawbacks are as follows. First and foremost, the requirement of a good transition acquisition method. This is not possible on general systems. However, in case of the robot arm this is straightforward. Second, this simulation only works for non-stochastic systems. This is because a function is fitted, that can only predict one outcome.

Chapter 4

Learning algorithm

In this chapter the learning algorithm is explained. Section one discusses related work. Next, section two explains the learning algorithm we use, Reinforcement Learning. Section three discusses a strategy to guide the learning process. Then, section four gives suggestions to improve the learning method. After that, section five compares Evolutionary algorithms with Reinforcement Learning. Section six gives conclusions.

4.1 Related work

In this section we show that not all learning methods are suitable for the problem at hand. We introduce two methods that are encountered frequently in the literature.

Van der Smagt [28] uses neural networks on the SoftArm from Bridgestone Corp. This is an out-of-the-box robot arm that uses two muscles per joint. His method shows improvement over the existing PID controller that comes with the SoftArm. The method is trained for a specific trajectory of desired angles, velocities and accelerations at every timestep.

The SoftArm uses servo-valves. Servo-valves take care that a desired pressure is set. Therefore, the control signal is a desired pressure ρ_{k+1}. The time difference between timestamps k and $k+1$ is 20 ms. The neural network is a mapping from current state and desired state $(\theta_k, \omega_k, \dot{\omega}_k, \theta_d, \omega_d, \dot{\omega}_d)$ to the next pressure ρ_{k+1}. Only the pressure of one muscle is calculated, because the sum of both pressures is kept constant. When an action is done, the outcome θ_{k+1}, ω_{k+1} and $\dot{\omega}_{k+1}$ is observed. This yields a training pair for the neural network: $(\theta_k, \omega_k, \dot{\omega}_k, \theta_{k+1}, \omega_{k+1}, \dot{\omega}_{k+1})$ with target ρ_{k+1}.

There are several reasons why we cannot use this approach. We use binary valves and therefore the controller itself should accomplish the desired pressure by opening or closing the valves. Moreover, the sum of the pressures in both muscles (i.e. the stiffness) is kept constant, whereas we want to exploit the compliancy of the muscles. But the most important reason is that this approach only uses one-step lookahead training samples. This is sufficient for this approach, since a trajectory of angles, velocities and accelerations is *given*.

A one-step lookahead approach is not sufficient for a general point-to-point movement. The movement should be explicitly or implicitly planned a-priori in order to reach the desired end-point as fast as possible. In Chapter 1 we defined this as the problem of *inverse dynamics*.

As the above method shows, not every learning algorithm is suitable for the problem at hand. In the literature, there are two methods that are encountered frequently to solve difficult control problems. The two methods include *Reinforcement Learning* and *Evolutionary algorithms*.

We choose to use Reinforcement Learning. The motivation for this choice is given in Section 4.5, where we compare Reinforcement Learning with Evolutionary algorithms.

4.2 Reinforcement Learning

4.2.1 Introduction

In this section Reinforcement Learning is explained. This is a potentially very powerful algorithm, based on an analogy with the way humans and animals learn behavior. The most famous application of Reinforcement Learning is created by Tesauro [26], whose program beat the human world champion in backgammon.

The rest of this section is organized as follows. Subsection two gives the basic theory. Subsection three explains how the value-functions are estimated. Next, subsection four gives three commonly used methods to solve the temporal-difference problem. Subsection five discusses eligibility traces. Finally, subsection six explains the design decisions.

4.2.2 Basic theory

There are three kinds of learning methods: supervised, unsupervised and reinforcement learning. *Supervised methods* have an analogy with a student and teacher relation: the methods get feedback on their performance. *Unsupervised methods* do not get feedback. Instead, they learn by constructing their own structure. *Reinforcement Learning* falls in between supervised and unsupervised methods. The agents learn autonomously (i.e. partly unsupervised), but use reinforcement signals (i.e. partly supervised).

The feedback from supervised learning is instructive: the good target (an action, in our case) is given. The feedback from Reinforcement Learning is evaluative: only some feedback about the action is given, but not what the correct action would have been. The idea of Reinforcement Learning is based on an analogy with humans and animals: good behavior is rewarded and bad behavior is punished.

Reinforcement Learning methods assume that the state σ has the *Markov property*, i.e. the state captures all necessary information and no history information is needed. A learning problem where the Markov property holds is called a *Markov Decision Process* (MDP).

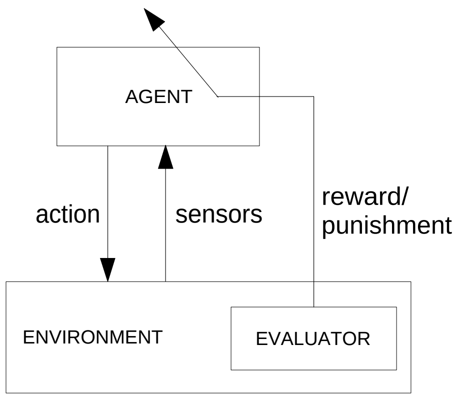

Figure 4.1: Reinforcement Learning.

RL diagram. The diagram of Figure 4.1 shows the basic components of a *Reinforcement Learning* problem. In the remainder we abbreviate 'Reinforcement Learning' to *RL*. The RL agent operates in an environment. The *environment* is defined to be everything that cannot be changed arbitrarily by the agent. Using an approach of trial-and-error, the agent interacts with the environment.

In RL problems, the human designer defines a *reward function*. This function assigns to every state σ or every (state, action) pair (σ, α) pair a *reward*. This reward function is represented in Figure 4.1 by the *evaluator*. From the agent's point of view, the evaluator is part of the environment. The agent's goal is to obtain as much reward as possible. To this purpose it holds a *policy* π, that maps a state σ to an action α.

Learning is an iterating process. An *episode* is a sequence of states and actions. An episode ends when a terminal state is reached, or when a certain number of actions is exceeded (cut-off). This cut-off number is specified by the designer.

Applied to the robot arm, the representation of the environment is by definition the state σ that we already derived in the previous chapter. The interaction consists of choosing one of the nine possible actions and observing the environment using the sensors. The designer's goal is to get the robot arm as fast as possible in a goal state $\sigma_g \in \Sigma_g$. The goal states are terminal states.

Definition of Return. The notion of a reward function introduces the *temporal credit assignment problem*. This is defined as the problem to assign credit or blame to the experienced state and actions. An action might have far-reaching effects, so how do we know whether the action just taken had been a wise decision? So how should we judge, or reward, an action? In the next paragraph we define the reward function we use.

First we introduce the definition of return. The *return* is defined as the discounted cumulative rewards of the future timesteps, starting at timestep k (i.e. in state σ_k).

$$R_k = \sum_{i=0}^{\infty} \gamma^i r_{k+1+i} \tag{4.1}$$

where:

r_{k+1}	reward observed at timestep $k+1$ (determined by σ_k or (σ_k, α_k))
γ	discount factor, $0 \leq \gamma \leq 1$

The return for terminal states is defined to be 0. The *discount factor* γ is specified by the designer and remains fixed. It makes rewards in the future less valuable than more immediate rewards (if $\gamma < 1$). We say that γ determines the *reward horizon*: the number of timesteps the agent 'looks ahead'. For problems where we have a guarantee that a terminal state will always be reached we may set $\gamma = 1$. This happens in the game of Tic-Tac-Toe for example. In our case we bound the length of the episodes but we cannot be sure to reach a terminal state, so we should choose $\gamma < 1$. We discuss this parameter in more detail when we discuss the properties of our parameters.

Reward function. How do we define the reward function for the robot arm? We can think of the robot arm as a deterministic system. Assume for the moment that we have a perfect model: *model*$(\sigma_k, \alpha_k) \rightarrow \sigma_{k+1}$. This would make it possible (at least, in theory) to go through all paths that are possible from a begin state σ_b and calculate the distance to the nearest goal state σ_g. We define the *distance* from state σ_b to a goal state σ_g as the minimal number of actions necessary to arrive in state σ_g, starting in state σ_b. Such a minimal number of steps does indeed exist. Our idea is to let this minimal distance be represented in the return.

Recall the agent's goal is to gain as much reward as possible. Note that an RL agent is *not explicitly required* to reach a goal state. If this happens, it is a side effect of minimizing costs (i.e. minimizing punishments or maximizing rewards). By constructing the reward function in the right way, we can match the agent's goal with our goal as designer. We want the robot arm to take a shortest path to a goal state. Inspired by the existence of a minimal distance, we construct our reward function:

$$\text{reward } r_{k+1} = \begin{cases} 50 & \text{, if action } \alpha_k \text{ leads to a goal state} \\ -1 & \text{, otherwise} \end{cases} \tag{4.2}$$

As a first rule, we stimulate the agent to head for goal states by giving a high positive reward to the (single) action that led to a goal state. The reward of 50 is chosen to keep the positive and negative rewards symmetric. The minimal return is: $\sum_{k=0}^{\infty} - \gamma^k = -\frac{1}{1-\gamma}$. With $\gamma = 0.98$ this equals -50. As a second rule, we punish the action with -1 if it did not lead to a goal state. With these two rules we guide the agent to the desired behavior. Note that we do not punish the agent if an episode exceeds a maximum number of actions. The reason is that the number of actions is not part of the state. The agent would not be able to derive why it had been punished on this particular action.

This reward function is chosen, because we want to minimize the number of actions. We could also use a *local reinforcement signal*, i.e. specify the reward function based on the direct implications of the action. For example, if the angular distance $\theta_g - \theta_k$ increases, we could give more punishment than if this

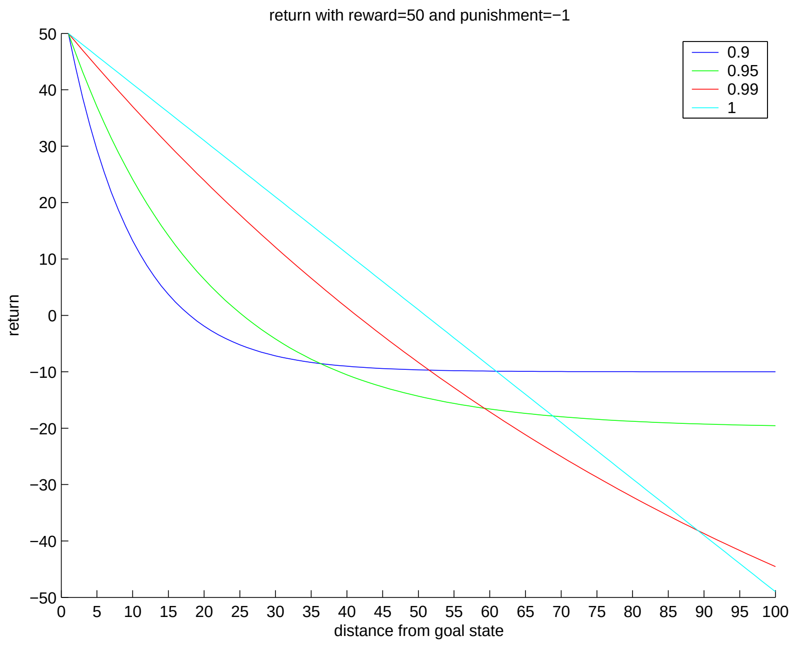

Figure 4.2: Return function for the optimal policy, for different γ.

distance decreases. The reason we refrain from using such a reward function, is that we do not have a *distance measure.* To obtain the fastest controller, it is not necessary that the angular distance should decrease monotonously. The true factor we want to minimize is the number of actions. Using another reward function might bias the solution too much.

Discount factor. On the basis of Figure 4.2 we explain the implications of the discount factor. The x-axis represents the states, ordered by their distance to a goal state. At the y-axis the associated return is shown if the maximum possible return is obtained, refer to Equation 4.3.

$$R(d) = 50\gamma^d + \sum_{k=0}^{d-1} -\gamma^k \tag{4.3}$$

The discount factor γ determines the relative weight of actions in the coming timesteps. The smaller γ, the less actions in the next timesteps are important. The positive reward has no influence anymore when the distance d is such that $\frac{\partial R(d)}{\partial d} \approx 0$. This means that the return is only determined by the punishments. In this case, the goal state does not influence the return anymore. It means that every possible action in a state has the same value, so there is no distinction between states. This way, no 'wise' decision can be made anymore.

It is important to choose proper values for γ and the reward function to get a good trade-off. A bigger γ means that more states are included in the return. So exponentially more experiences are necessary to get a good estimation of the return. But we should also not choose it too small, because otherwise the

return might not look enough ahead, so that high rewards from goal states only have influence in a very small environment. This depends on how the goal states are distributed in the state space.
Before we analyze the implications of this reward function, we first introduce the value function.

Value function. The function that gives the expected return value for every state, following policy π, is called the *state-value function* $V^{\pi}(\sigma)$:

$$\begin{aligned} V^{\pi}(\sigma) &= E_{\pi}\{R_k|\sigma_k = \sigma\} && (4.4) \\ &= E_{\pi}\{\sum_{i=0}^{\infty}\gamma^i r_{k+i+1}|\sigma_k = \sigma\} \\ &= E_{\pi}\{r_{k+1} + \gamma\sum_{i=0}^{\infty}\gamma^i r_{k+i+2}|\sigma_k = \sigma\} \\ &= E_{\pi}\{r_{k+1} + \gamma V^{\pi}(\sigma_{k+1})|\sigma_k = \sigma\} && (4.5) \end{aligned}$$

In words V^{π} is defined as "the expected value of the infinite sum of the discounted rewards when the system is started in state σ and the policy π is followed forever". We emphasize that V is defined as the *expected* return (i.e. the statistical mean). This is important in a stochastic environment.

To enable an introduction to the next concepts, we assume for the moment that the RL agent follows the optimal policy, denoted with π^*. This is the policy that gains the highest return. The associated state-value function is denoted by: $V^*(\sigma) = \max_{\pi} V^{\pi}(\sigma)$. Before we are able to analyze the implications of the reward function, we need to make clear that V^* is *defined*. The influences on the return are the discount factor γ, the rewards that are obtained, the states that are visited and the actions that are taken. If we assume we follow policy π^*, these influences *are fixed and define together* the return in any state (and hence, at any timestep).

Sometimes it is more convenient also to take the actions into consideration, instead of only the states. This is called the *action-value function*:

$$Q^{\pi}(\sigma, \alpha) = E_{\pi}\{R_k|\sigma_k = \sigma, \alpha_k = \alpha\} \quad (4.6)$$

The state-value function and the action-value function are related:

$$V^{\pi}(\sigma_k) = \max_{\alpha} Q^{\pi}(\sigma_k, \alpha) \quad (4.7)$$

The optimal action-value function is $Q^*(\sigma, \alpha) = \max_{\pi} Q^{\pi}(\sigma, \alpha)$.

V^{π} can only be used if the next state can be predicted, i.e. if a one-step lookahead is possible. Therefore, a model is necessary to use V^{π}. Q^{π} on the other hand, can be used without having a model. This is explained in more detail in Section 4.2.3.

To fully comprehend the use of V or Q we need some more intuition about their properties. These depend on the reward function. As will turn out, the use of the action-value function Q is two-fold. First, it is a way for decision making. Second, it is used to cope with the temporal credit assignment problem. Both are topics of the next paragraphs.

Properties of the parameters. Via the reward function we tell the agent what it needs to optimize. We show that Equation 4.2 satisfies our goals, by giving three important properties of V^* and Q^* (i.e. when the optimal policy π^* is followed):

1. A state with a higher return has a higher probability that a goal state is faster to reach.
2. From every state there is a shortest path towards a goal state that has a monotonously increasing return.
3. The path with the steepest increasing return gives the highest probability of arriving in the fastest way in a goal state.

The first property follows trivially from the definition of minimal distance and the punishment on actions. We deliberately say 'probability' since we cannot discard the exploration actions (we explain the concept of exploration actions in the next paragraph). Also, the environment can be stochastic. The second property can be seen as follows. There is always a path to a goal state, and therefore there exists a shortest path. We may rewrite the return as $R_k = r_{k+1} + \gamma R_{k+1}$. If σ_{k+1} is not a goal state, r_{k+1} is negative and thus $R_k < \gamma R_{k+1}$. The third property is a corollary of the first two properties.

In the properties above we only used one part of the reward function: the distance we aimed at is represented in the punishment on each action. Why should we give a positive reward at all? A high reward implies that its influence reaches further than with a small reward would be accomplished. The ratio between reward (50 in our case) and punishments (-1) is important. Another effect of high rewards is that the distinction between goal-reaching paths and non-goal-reaching paths is emphasized. That is especially useful if the estimation of Q^π is not yet very good. If a goal state is encountered (perhaps by an exploration action), this 'positive' experience is expressed more pronounced in the return when the reward is high.

Policy. Earlier we assumed that the optimal policy π^* was used. Now we drop this assumption. The problem of calculating Q^π or V^π is the topic of the next subsection. For the moment it is important that we realize that these functions indeed exist. RL holds an *estimation* of Q^π or V^π. The estimation of Q^π at timestep k is called Q^k. Analogously, the estimation of V^π is V^k.

The only information[1] an RL agent has about its environment is Q^k. Therefore, the action decisions of the agent are based on this estimation. If not, every decision would be a random one. The *policy* of an RL agent is a function that maps a state σ to an action α, using Q^k.

With the properties of the reward function, the policy choice might seem straightforward: in any state σ always pick the action α that gives the highest $Q^k(\sigma, \alpha)$. This is called the *greedy policy.* We say that this policy makes extensive use of *exploitation*: the use of the current state of knowledge. However, we cannot assume that Q^k is a perfect approximation of Q^π. We have no perfect knowledge of the return. The only way to get this experience is by interacting with the environment. That is why *exploration* is important. The conflict

[1]Note: This does not hold for Actor-Critic learning, see the next section.

between exploitation and exploration is fundamental and recurs in many problems. In [5] some practical experiments for RL are given. In the remainder of this paragraph we discuss some exploration policies.

ϵ-greedy exploration policy. An easy way to deal with it, is to take exploration actions once in a while. A common policy is called the ϵ-greedy policy. This policy takes with probability ϵ a random action and with probability $1-\epsilon$ the greedy action. When the estimation Q^k is still very poor, we set ϵ higher and let it gradually reduce as the experience grows. The ϵ-greedy approach is the most extensively used policy in the literature.

Boltzmann exploration policy. On the robot arm, truly random exploration actions are not desired, since one bad action can ruin the movement by taking out the speed. A bit more sophisticated is *Boltzmann exploration*, also known as *Gibbs softmax method.* This policy calculates a probability distribution for the actions, based on the action-value function. The probability of choosing action α is:

$$P(\alpha) = \frac{e^{Q^k(\sigma,\alpha)/T}}{\sum_{\alpha'} e^{Q^k(\sigma,\alpha')/T}} \tag{4.8}$$

The *temperature* parameter T can be decreased over time to decrease exploration. The temperature has a big influence on the probability distribution and should be tuned with great care [12].

Max-Boltzmann exploration policy. We can also combine the ϵ-greedy and the Boltzmann policy. Then we take with probability $1-\epsilon$ the greedy action, and with probability ϵ we do an exploration step with the Boltzmann policy [29]. This way, we still have control about the percentage of exploration steps, and can also choose the level of exploration using the temperature. This is the policy that we apply. To force the policy to choose a non-greedy action in case of an exploration step, we disable the greedy action (i.e. the action with the highest $P(\alpha)$).

In this paragraph we showed that a policy necessarily needs to use the current estimation of Q^π. In the next section we introduce three techniques to come to an estimation of V^π or Q^π.

4.2.3 Estimating the value function

In the previous section we explained that the policy π chooses its actions based on an estimation of Q^π (or V^π). Eventually we want π to become the optimal policy π^*. This yields a difficult problem: If the Q^k function is not a good estimation, then the policy chooses a non-optimal action. If the policy takes a non-optimal action, a non-optimal reward is obtained and Q^k is updated with this non-optimal reward. This is a bootstrapping problem, since one estimation is based on another one.

In [23, Section 4.6], *Generalized Policy Iteration* is explained. This basically means that this bootstrapping problem is 'tackled' by interweaving the estimation of V^π and changing π simultaneously in order to let them converge together. In our case, the policy strategy is fixed, except for its parameters. Therefore we focus on the estimation of V^π and Q^π.

The problem of estimating the state-value function V^π or action-value function Q^π is called the *policy evaluation* or the *policy estimation* problem. In this section we first discuss two approaches that calculate V^π (under a fixed policy): Dynamic Programming and Monte Carlo learning. These approaches have different assumptions and therefore solve this problem in different ways. After that, a new approach is introduced that combines properties of both methods. This new approach is called Temporal Differences learning and constitutes the basis of modern RL.

Note that, for the moment, we are not bothered by the storage problem of V^k or Q^k. We assume that the function can be stored in a look-up table. We return to the subject of storing this function afterwards.

Dynamic Programming. Dynamic Programming (DP) assumes a model of the system: finite state and action sets, the transition probabilities are known and the expected rewards are known. DP is based on the Bellman equations [23, Chapter 3.7].

DP approximates V^π using an iterative policy evaluation approach. The first approximation is called V^0. This function is initialized randomly. The terminal states are defined as 0. The update rule of DP is based on Equation 4.5:

$$V^{l+1}(\sigma) = E_\pi\{r_{k+1} + \gamma V^l(\sigma_{k+1}) | \sigma_k = \sigma\}$$

We say that this update rule applies a *full update* on every state: the update is based on the values of all possible successor states and their probability of occurring. This is possible since DP assumes a perfect model. Further, the update rule bases its estimation V^{l+1} on another estimation V^l. Therefore, DP uses *bootstrapping*.

Because of their assumption of a perfect model and the great computational costs, DP methods are of limited utility. However, the theoretical basis of RL is founded on DP theory.

Monte Carlo learning. Monte Carlo learning (MC) does not assume a model but rather uses *experience*: sample sequences of states, actions and rewards from an environment. Even if a simulation is used, MC does not require the probability distribution of the transitions to be known.

An important property of MC methods is that they use the return, instead of the rewards. That is, MC methods wait until the end of the episode to determine the increment of $V^k(\sigma_k)$. This implies that the RL task should be divided into episodes. MC does not assume a model and therefore it cannot perform full updates. Instead, observations (experience) are used to update its estimation. We say that MC uses *sample updates.* A possible update rule for MC methods is as follows:

$$V^{k+1}(\sigma_k) = V^k(\sigma_k) + \beta[R_k - V^k(\sigma_k)]$$

This equation is based on Equation 4.4. A natural way to estimate the expected state-value function is by averaging its observations. We say that MC uses a *sample average* method if $\beta = \frac{1}{k}$. However, we can also choose a fixed stepsize for β. Choosing a fixed stepsize has the property that more recent updates have more weight. That might be an advantage in non-stationary environments.

Temporal Difference learning. Temporal Difference (TD) learning has been one of most important concepts in RL. TD combines the ideas of both Monte Carlo and Dynamic Programming.
The update rule of TD learning is as follows:

$$V^{k+1}(\sigma_k) = V^k(\sigma_k) + \beta[r_{k+1} + \gamma V^k(\sigma_{k+1}) - V^k(\sigma_k)]$$

As we saw before, MC tries to estimate Equation 4.4, so it uses a sample return R_k instead of the expected return $E_\pi\{R_k\}$. TD uses the same equation, but the target $r_{k+1} + \gamma V^k(\sigma_{k+1})$ is used instead of R_k. DP tries to estimate Equation 4.5, so it uses the estimation $V^k(\sigma)$ instead of $V^\pi(\sigma)$. TD uses bootstrapping, just like DP does: it bases its estimation on another estimation. TD combines the sampling of MC with the bootstrapping from DP. The complete algorithm of the Temporal Differences algorithm is shown in Algorithm 1, adapted from [23].

Algorithm 1 TD(0) Algorithm.

```
 1: initialize
 2:         $V(\sigma) \leftarrow$ random;
 3:         $\pi \leftarrow$ policy to be evaluated;
 4: repeat(for each episode)
 5:     initialize $\sigma$;
 6:     repeat(for each step of episode)
 7:         $\alpha \leftarrow$ action given by $\pi$ for $\sigma$;
 8:         take action $\alpha$;
 9:         observe next state $\sigma'$;
10:         observe reward $r$;
11:         $V(\sigma) \leftarrow V(\sigma) + \beta[r + \gamma V(\sigma') - V(\sigma)]$;
12:         $\sigma \leftarrow \sigma'$;
13:     until $\sigma$ is terminal (or the episode is cut-off)
14: until controller performs well
```

Recall the return of a terminal state is 0.
Advantages of TD prediction methods:

- No model of the environment, rewards and next-state probability distributions is needed.
- Natural implementation in an online, fully incremental fashion. This is important, since some applications are continuous and have no bounded episodes.

What can be said about the convergence of V^k to V^π of this algorithm? We cite from [23, Section 6.2]: "For any fixed policy π, the TD algorithm described above has been proven to converge to V^π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability one if the step-size parameter decreases according to the usual stochastic approximation conditions." The 'usual stochastic conditions' include the following two requirements [23, Section 2.8]. First, the update steps should be large enough to eventually overcome any initial conditions or random fluctuations. Second, eventually the steps should become small enough to assure convergence.

4.2.4 Temporal Difference methods

In this section we discuss three variants of TD learning, called SARSA, Q-learning and Actor-Critic. SARSA and Q-learning replace the update rule in the TD algorithm by their own variant, but are almost similar to the TD algorithm as given above. These update rules differ from each other because the methods have different goals. Actor-Critic works quite differently, as we will see.

SARSA. First we treat SARSA. SARSA tries to estimate Q^π, by using the update rule:

$$Q(\sigma_k, \alpha_k) = Q(\sigma_k, \alpha_k) + \beta[r_{k+1} + \gamma Q(\sigma_{k+1}, \alpha_{k+1}) - Q(\sigma_k, \alpha_k)] \tag{4.9}$$

The update rule of SARSA uses $Q(\sigma_{k+1}, \alpha_{k+1})$, so before the update rule can be applied, a new action for timestep $k+1$ needs to be chosen. This choice is done by the policy π. After the update has been done, this action is executed in the next iteration. The update rule of SARSA gives rise to its name: State - Action - Reward - State - Action.

SARSA converges with probability one to an optimal policy and action-value function if the policy is greedy in the limit with infinite exploration [21].

Q-learning. Q-learning tries to estimate the optimal policy, i.e. Q^*, instead of Q^π. Its update rule is therefore different from SARSA's. The difference is in the selection of the action α_{k+1}. Whereas SARSA lets the policy π choose the next action α_{k+1}, Q-learning always chooses the action b that yields the highest return (according to the current estimation of Q^π). Its update rule is:

$$Q(\sigma_k, \alpha_k) = Q(\sigma_k, \alpha_k) + \beta[r_{k+1} + \gamma \max_b Q(\sigma_{k+1}, b) - Q(\sigma_k, \alpha_k)] \tag{4.10}$$

The selection of action b does not imply that this action is indeed executed at timestep $k+1$ after this update. The action b is only important for the update rule. Recall it is the policy π that chooses what actions are taken. For Q-learning this is no different, but Q-learning tries to abstract from the policy π in order to estimate Q^*.

On-policy vs off-policy. In this context, we say Q-learning is an *off-policy* method. Q-learning tries to abstract from the policy π by updating the function disregarding the policy's choice of actions. Of course, the policy still has an influence in what actions are taken for real and therefore, what states are visited. Since SARSA tries to estimate Q^π, we say that it is an *on-policy* method. The updating depends exclusively on the policy that is being followed.

In case the greedy action is chosen there is no difference between the update rules of SARSA and Q-learning. For the ϵ-greedy exploration policy, this means that the probability of different updates equals ϵ. Their different update rules result in different behavior. SARSA explicitly takes the policy into account, and therefore also possible exploration steps. In some RL problems, exploration steps may result in high punishments. Therefore, SARSA tries to get the highest average reward. Q-learning on the other hand, tries to learn the optimal path to a goal state. It always assumes that normally only the best action is chosen.

Tabular Q-learning converges to the optimal policy when all state-action pairs are experienced an infinite number of times. *Tabular* means that the function is stored in a look-up table.

Actor-Critic Methods. In the previous two methods, the policy is directly based on the current estimation of the action-value function Q^{π} or Q^{*}. Actor-Critic methods work in a different way. Here the policy and the value function are stored separately. That is, the policy is explicitly decoupled from V^{π}. In this context, the policy is referred to as the *actor.* The policy cannot access the action-value function and therefore cannot base its decisions on the action-value function. The part that estimates V^{π} is called the *critic.* The critic obtains the rewards and gives the TD error δ_k as feedback to the actor:

$$\delta_k = r_{k+1} + \gamma V(\sigma_{k+1}) - V(\sigma_k) \tag{4.11}$$

The TD error is used to evaluate how good or bad action a_k had been in state σ_k. If $\delta_k > 0$, the tendency to apply α_k more often in that state should be stimulated. If it is negative, it should be weakened. Note that the learning is always *on-policy*, since the learning of the critic and the TD error is always based on the policy that is followed by the actor.

An important advantage of Actor-Critic methods is their minimal computation demands in order to select actions. Any method just learning action values, like SARSA and Q-learning, must search through all actions in order to select an action. If there are many discrete actions this might be too slow. This advantage is even more pronounced if we consider continuous actions, where SARSA and Q-learning might have to discretize the action (this holds for conventional SARSA and Q-learning as we have discussed it).

A drawback of the AC method is that it uses two functions. These functions influence each other and should improve together. This added complexity makes the learning process more delicate.

4.2.5 Eligibility traces

Eligibility traces are an additional method to solve the temporal-credit assignment problem, on top of the TD algorithm. In this subsection we discuss the concept of eligibility traces briefly. The explanation is somewhat simplified, since it assumes that the (state, action) pairs can be stored in a look-up table. However, eligibility traces can also be used in combination with neural networks. For more information we refer to [23, Chapter 7].

In RL, only the current state action pair (σ_k, α_k) is updated with reward r_{k+1}. TD methods use the current estimation Q^k to solve the temporal-credit assignment problem (refer to Section 4.2.2).

Eligibility traces help to solve the problem in another way. They keep track of what (state, action) pairs have been visited most recently and update them using reward r_{k+1} as well. An eligibility trace is a scaler associated with every state σ or with every (state, action) pair (σ, α). Parameter λ expresses how much the previous (state, action) pairs participate in the reward. λ is called the *decay rate*. The eligibility trace associated with the current state or current (state, action) pair is increased by one.

All eligibility traces e are set to 0 at the start of an episode. When $\lambda = 0$, only σ_k or (σ_k, α_k) participate in the reward. When $\lambda = 1$, all previous (state, action) pairs participate (theoretically speaking). It can be said that eligibility traces form a bridge from TD to Monte Carlo methods.

Almost all TD methods can be combined with eligibility traces. We speak about SARSA(λ) and Q(λ) in this case. Step 11 from Algorithm 1 is replaced by the steps from Algorithm 2. If Q^k is used instead of V^k, the loop is done for all states and all actions.

Algorithm 2 Updating steps for TD(λ)

```
1: $\delta \leftarrow r + \gamma V(\sigma') - V(\sigma)$;
2: $e(\sigma) \leftarrow e(\sigma) + 1$;
3: for all states $\sigma''$ do
4:     $V(\sigma'') \leftarrow V(\sigma'') + \beta\delta e(\sigma'')$
5:     $e(\sigma'') \leftarrow \gamma\lambda e(\sigma'')$
6: end for
```

4.2.6 Design decisions

In this section we discuss our design decisions.

TD Method. First we decide what TD method to use: SARSA, Q-learning or Actor-Critic. In the previous section we mentioned that Actor-Critic has an important advantage when using continuous actions. This would be the case for the robot arm if we would have chosen continuous valves (so-called servo-valves). However, we deal with binary valves and therefore the advantage to use Actor-Critic becomes weak. Its additional complexity makes us decide to discard the Actor-Critic method. Q-learning has an advantage since it is off-policy, but there is a catch as we will explain next. For the moment we leave the choice between SARSA and Q-learning undecided.

Storage of action-value function. We have to decide how to store the action-value function. As we have a big state space, a look-up table is no practical solution in terms of time and space. A look-up table is especially no option anymore when some dimensions in the state space are continuous.

In practice the RL agent will never experience all possible states. Only a subset of the state space is visited. We need a way to *generalize* over this limited subset of the state space. This generalization is possible by the use of function approximators (FA).

Unfortunately, there is a catch when using function approximators. Off-policy bootstrapping combined with function approximation can lead to divergence and infinite MSE [23, Section 8.5]. That is, Q-learning has a probability of divergence, whereas SARSA does not. An example can be found in [30]. This is an important theoretical advantage of SARSA. However, we should not abandon Q-learning too easily, since [23, Section 8.5] also claims: "To the best of our knowledge, Q-learning has never been found to diverge in this case [ϵ-greedy policy], but there has been no theoretical analysis." Therefore, we will leave it to our experiments to see if either SARSA or Q-learning is to be preferred.

Gradient-descent methods. Gradient-descent methods minimize the error on the observed examples by adjusting their parameters in the direction where the error shrinks the most. Gradient-descent methods are among the most widely used function approximation methods and are particularly well-suited to reinforcement learning [23, Chapter 8.2]. TD methods are easily combined with a gradient-descent method.

Two forms of gradient-based FA are widely used in RL. One form is multi-layer artificial neural networks using the error backpropagation algorithm. Another form are linear methods, like coarse coding, tile coding, radial basis functions and Kanerva coding [23, Chapter 8.3].

We want to use a gradient-descent method, mainly for two reasons. The first one we already mentioned: TD methods are easily implemented with gradient-descent methods. Second, we would like to incorporate eligibility traces. These lend themselves naturally to be implemented using gradient-descent.

We stick to a gradient-descent method. We choose to use multi-layer artificial neural networks. In the remainder we will simply use the term *neural network* or use the abbreviation NN. Since the state contains sensor values of multiple timesteps, we actually deal with a *time-delay* NN.

We use the backpropagation algorithm to train the neural network, with one hidden layer. For the hidden layer we use the tansig transfer function, and for the output layer we use a linear transfer function [8]. To initialize the neural network, we randomly assign values in $[-0.3, 0.3]$ to the weights. In order not to bias the inputs of the NNs, we choose to normalize the inputs. This is very important. Otherwise some inputs would have a bigger influence than others, purely based on the magnitude of their domain. We also normalize the outputs. Otherwise big targets would imply big updates. Without normalization we would have to decrease the learning rate for the output weights.
We use a NN for the following reasons:

- **Non-linearity:** We do not want to make linear assumptions since we do not know the nature of the action-value function we have.
- **Continuous inputs:** There is no need to discretize the inputs.
- **Generalization:** NNs are known to generalize well since all weights determine the outcome. Other methods, like radial basis networks work very locally.
- **Generally applicable:** Using tile coding for example, one initially has to decide how to classify the tiles. This requires some knowledge about

the relative importance of the state variables. An NN approach is a more black-box approach.

- **Proven capability:** NNs have been used for decades already, and have proven to work well. In RL literature NNs are used abundantly.

Of course, NNs have drawbacks as well. One is that the design of the neural net is not yet well understood. The number of hidden neurons and the learning rate are merely chosen by the experience of the designer than it can be derived. Second, NNs converge to a local optimum. Finally, NNs suffer from *unlearning* (see Section 3.2.4 for more information). Unlearning is a very serious problem that we should handle with care. One worry is that the controller might always pick only one or two actions in some states. The NN will optimize its prediction for these two actions and will have a bad estimation for the other actions. One way to deal with this is to distribute the action-value function over the actions. In other words: introduce a NN for every action. We use nine NNs that together estimate the action-value function.

State. The standard benchmarking problem in Reinforcement Learning literature is the Inverted Pendulum problem [19]. For this problem, the goal is always the same: balance an inverted pendulum on a cart as long as possible. For every state there is only one optimal action. In our case this is not true. In Chapter 3 we have defined the state of the robot arm. But dependent on the goal angle we head for, the optimal action in every state will differ. We actually have an RL problem per goal angle: with every goal angle another Q^{π} is associated. However, we expect that goal angles close to each other (small angular distance) are correlated. Therefore, we choose to parameterize the state with the desired goal angle:

$$\sigma_k = (\theta_g, o_k, o_{k-1}, \alpha_{k-1}, o_{k-2}, \alpha_{k-2}) \tag{4.12}$$

Because of the correlation between goal angles, we expect the generalization property of NNs to be sufficient. The advantage of parameterizing the state is that we still only need nine NNs in this case.

4.3 Shape the learning process

4.3.1 Introduction

A generally good learning algorithm by itself does not guarantee fast learning on all control problems. In complex problems it might prove valuable to incorporate bias to give leverage to the learning process [12]. In [12], some forms of bias are given. These forms include:

- **shaping:** A teacher presents very simple problems to solve first, then gradually exposes the learner to more complex problems.

- **local reinforcement signals:** The use of immediate rewards eases the temporal-credit assignment problem.

- **imitation:** Imitate behavior of another agent (another controller or a human using a joystick).

- **problem decomposition:** Decompose a learning problem into smaller ones, and providing useful reinforcement signals for the subproblems.
- **reflexes:** Give a set of a-priori designed reflexes to give the agent a starting behavior that makes it wander through the interesting states.

Recall our research aim is to avoid the use of system knowledge. The agent should autonomously learn the problem, without guidance in its action choices. Taking this aim into account, what bias or combination of biases can we use for our problem at hand? *Shaping* is possible, by starting with 'easy-to-reach' goals. As will turn out in the next subsection, we use this kind of bias to guide the learning process. *Local reinforcement signals* are possible by making (for example) the reward function dependent on the absolute distance $|\theta_k - \theta_g|$. This can be extended by incorporating the velocity as well. However, this kind of reward function is more tricky, since this local reward does not necessarily equal the true distance to a goal. The use of *imitation* could work as well. We might connect a keyboard with three buttons per muscle (air in, air out, nothing). However, this violates our research aim that the AI should find the 'optimal' solution autonomously. The use of *problem decomposition* is possible, but requires more knowledge of the system. We could decompose a point-to-point movement by dividing the movement into different phases. For every phase we could use a dedicated agent, but this also could be the same agent. For example, one phase could be to make swift movements to the desired direction, but not caring too much about the end position. Another phase would then take care of the more accurate movement of positioning the arm in the right angle. The latter phase only cares about a small area around the end position, and could therefore be learned more easily and more accurately. Though this might improve the performance, we decided to refrain from a division in phases unless all other methods would fail. Finally, the use of *reflexes* would require to at least build a controller by some simple thumbrules. Again, that violates our research aim that no a-priori knowledge should be used.

We therefore concentrate on shaping. This is the topic of the remainder of this section. For the moment we assume that we only deal with one goal state. Extension to multiple goal states is straightforward and will be discussed at the end of this section.

4.3.2 Motivations for shaping

From the agent's point of view, episodes are not correlated to each other. The selection of begin states of these episodes is done by a process external to the agent. We show that this selection process needs to be chosen carefully and why shaping might improve the learning process a lot.

Note that many RL problems, like the Inverted Pendulum problem or maze problems, usually have the same begin state. For those problems, a shaping strategy might be of less importance. In our case, we want the agent to be able to start from every possible state to reach every possible goal angle.

We have three reasons to apply shaping. First we mention the reasons briefly, then we discuss these in more detail:

- RL learns the problem **backwards**.

- RL needs to **keep on visiting states**. This is because of the unlearning property of the NNs, to improve on previous solutions and it is one of the assumptions of the convergence proofs.
- Shaping might improve the **learning speed** of the agent.

We propose a solution for the first reason, and a solution for the second reason. The solutions are merged to form the shaping strategy. The third reason (speed) is a corollary of the first two.

4.3.2.1 Learn backwards

Motivation. The RL agent is required to visit goal states. This can be seen as follows. Recall the goal of an RL agent is to gather as much reward (or as little punishment) as possible. The reward function as we designed it assures that better (i.e. shorter) paths to a goal state result in higher cumulative rewards. However, as long as the agent does not visit goal states, the actions on these desired paths will not be favored above other actions. There is simply no benefit of these actions above other actions, from the agent's perspective at least. This results in an equally bad expectation $Q(\sigma, \alpha)$ for all actions α in state σ. We conclude that the encountering of goal states is *indispensable* with the reward function we use. This does not necessarily hold for other reward functions. For example, the use of local reinforcement signals circumvents this requirement.

We ought to make sure the agent *will* regularly find itself in goal states. Note that the learning strategy has no influence on the agent's policy (except for ϵ and T), so how can we achieve that goal states are reached? A possible answer to this question is: choose the begin states close to states the agent has already been trained for. Next, we formalize the notion and introduce our strategy.

Classification of begin states. In Section 4.2.2 we defined the *distance* of a state as the minimum number of actions that is needed to reach a goal state. Unfortunately, we do not know the true distance. We approximate this notion by assuming the distance is in some way related to the required angular displacement:

$$d(\sigma_k) \sim |\theta_g - \theta_k| \tag{4.13}$$

This relation does not yield a true measure of difficulty of a begin state, but rather should be considered a classification of begin states. In general we expect that bigger angular displacements require more time (in our case, the notion of time is expressed as the number of actions). The term distance only refers to a single (begin) state. We use the term *deviation* to express the maximum allowed distance to the goal state. Every begin state that has a distance equal or less than the current deviation may be chosen as a begin state for an episode. We consider the interval $[\theta_g - \texttt{deviation}, \theta_g + \texttt{deviation}]$ as the 'known' begin states. Note that we assume the same deviation for angles lower and higher than θ_g. If either of the two exceeds one of the bounds 0 or 1023, we adapt the intervals, leaving the value of the deviation parameter unchanged.

Selection process. In this paragraph we explain our shaping strategy. We start with the general idea, and gradually put more details into it.

General idea. We exploit the fact that we have a database and a simulation at our disposal. The database contains real observed sound states and therefore it can be used to select sound begin states. The simulation is beneficial since it gives an instantaneous control of the begin states. On the real system we cannot bring the arm to a desired begin state, since we have no controller yet. However, shaping can be applied on the robot arm as well, as we discuss in Section 4.3.3.

The general strategy is to start the learning process with begin states that are very close to a goal state. As we explained before, we cannot translate 'very close' in terms of the number of actions that are needed. Rather we express this notion in terms of angular displacement. Therefore we start with a deviation of a few degrees off the goal angle.

If the agent improves its performance, we gradually increase the deviation. If it turns out that the performance actually drops too much, we lower the deviation a bit.

Exploration. We have to specify when and how we increase or decrease the deviation. But first we decide how to tackle the exploration-exploitation problem.

The nature of RL learning is by trial-and-error. This makes sense since the agent has no a-priori knowledge. We therefore set the initial exploration rate high. Initially we want the exploration actions to be truly random. Therefore the temperature T is set high as well.

The eventual learning problem is solved when the deviation (i.e. the 'known' interval) equals the complete angular domain. By the time the problem is nearly solved, we want the agent to make more extensively use of exploitation. Then we give ϵ a low value.

Furthermore, the use of truly random exploration actions is not desirable since one wrong action might spoil a whole movement. An example is the case where the agent tries to accelerate the arm by putting air in one of the muscles. It does not make sense when halfway the motion an exploration action suddenly lets some air out. This would spoil the acceleration and smoothness of the motion. Therefore we want T to be low at the end. This assures that only relevant actions are tried.

The deviation tells how far the learning problem is solved thus far. Therefore, we make ϵ and T functions of the deviation. When the deviation is minimal, we set $\epsilon = 0.5$ and $T = 20$. The value of T has been decided empirically, the value of 20 generates random actions. When the deviation is maximal, we set $\epsilon = 0.05$ and $T = 1$. For the rest, we linearly interpolate ϵ and T, refer to Figure 4.3.

Performance criterium. Earlier we explained that we increase or decrease the deviation, based on the agent's performance. Now we discuss how to evaluate the performance.

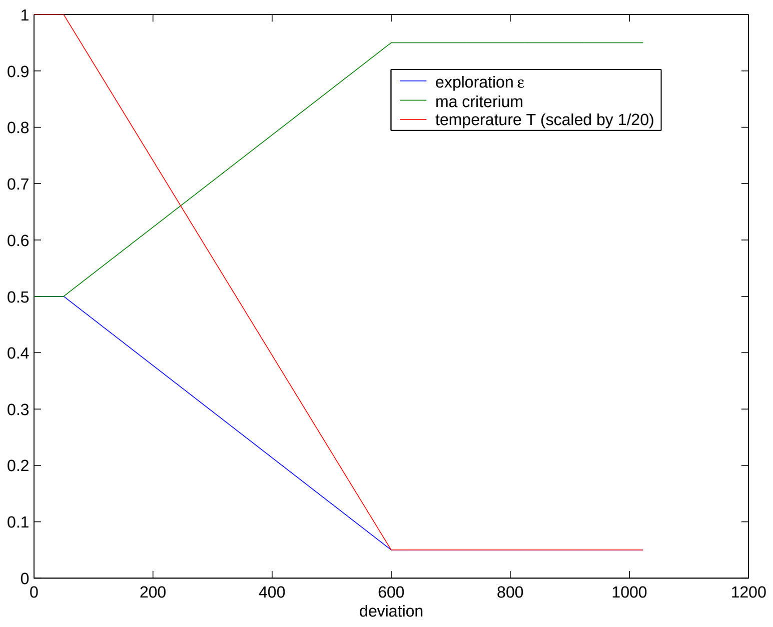

Figure 4.3: Exploration, temperature and moving average of criterium as function of deviation.

We keep track of the agent's performance $p(k)$ by calculating the success rate of the last 500 episodes (i.e. a moving average). We classify an episode as a success when a goal state has been reached. Otherwise it is a failure.

$$p(k) = \begin{cases} p(k-1) * 0.998 + 0 * 0.002, & \text{on failure} \\ p(k-1) * 0.998 + 1 * 0.002, & \text{on success} \end{cases} \tag{4.14}$$

A success rate of 100% does not seem realistic. Especially in the beginning, when the exploration rate is very high, it is not very likely to end many times in a goal state. Even after extensive training, the high exploration rate makes it rather impossible to compensate for. Therefore, we use a *criterium* that denotes when the agent's performance is acceptable. Because of the high exploration rate, we start with a low criterium. By the time the deviation is almost at its maximum value, we want a very good performance and therefore use a more strict criterium.

The criterium is set as follows. The minimum success rate we demand is 50%. At the end, we demand that at least 95% of the episodes is successful. The criterium is a function of the deviation. We linearly interpolate the criterium based on the deviation, refer to Figure 4.3. Now that criterium is discussed, we explain how to increase or decrease the deviation.

Deviation adjustment. We may increase or decrease the deviation with a constant value (absolute stepsize) or we may adjust the deviation with a percentage (relative stepsize). We have chosen the latter option. We expect that an increase of 10 θ-units has more impact if the deviation is only 50 than when it is 300.

Every 500 episodes, the performance is evaluated. If the percentage exceeds the criterium, we increase the deviation with 10%. If not, the deviation is decreased with 3%. Increasing the deviation makes the control problem a bit harder, and on top of that we also demand a higher criterium. However, we expect that the added begin states will not differ too much from the already known begin states. The generalization properties of the neural networks should help the agent on this matter. The difficulty is also a bit compensated by the lowering of the exploration rate and the shrinking of the randomness of the exploration steps. But it may be that the increment had been too high and the new deviation is too difficult for the agent. To relieve the agent from this burden, we incorporate the decrement of the deviation. This decrement is useful, as we explain in the next section.

4.3.2.2 Keep on visiting states

There are several reasons why the agent needs to keep on visiting states:

- **Convergence proofs:** Theoretically, the convergence proofs of both SARSA and Q-learning require that all states will be visited infinitely often.
- **Unlearning of NNs:** If the agent does not keep on visiting all states, the prediction of Q^π might suffer in these states. This is the reason why we decrease the deviation if the performance is not met. Also, it assures that goal states are visited, if the begin states are close to goal states. This prevents the unlearning of goal states.
- **Keep improving:** RL is supposed to improve. This is done by keep on trying to gain more reward (and hence, to find shorter paths).

We can guarantee that the agent keeps on visiting the whole state space, by selecting begin states randomly in the state space. Earlier we discussed the concept of adjusting the deviation. We explained that only begin states that lie within the deviation interval are chosen. So far, we did not make clear *how* we choose these begin states. The merging of those two concepts is straightforward: choose randomly *begin states* from the database that satisfy the *deviation.*

Extension to multiple goal states can be done by keeping a vector that contains the allowed deviation for all goal angles. We simply use the same deviation for all goal states. We also calculate the criterium on all goal angles together.

Discretize the interval. We could easily just query the database for random states. However, in the previous chapter about the empirical simulation, we already remarked that the collection procedure that gathers transitions is likely to be biased. States that occur often in the database have a higher probability to be selected. States that are sparsely available have a probability that can almost be neglected.

Therefore, we try to circumvent this bias somewhat by discretizing the interval $[\theta_g - \text{deviation}, \theta_g + \text{deviation}]$ in eight disjunct (sub)intervals. The higher the discretization, the less the database's bias plays a role, because the sparse states get a higher probability. The bias could be reduced even further by discretizing for the other state variables ω, ρ_1 and ρ_2 as well.

Every discretized interval has an equal probability to be chosen. The chosen interval determines the lower and upper bound of the database query. For example, suppose the deviation is 200. We discretize the interval:

$$[\theta_g - \text{deviation}, \theta_g + \text{deviation}]$$

in eight intervals:

$$[\theta_g - 200, \theta_g - 150), \ldots, (\theta_g + 150, \theta_g + 200]$$

Suppose the left interval is chosen. Then the database is queried for a random state σ_k, with the constraint $\theta_g - 200 \leq \theta_k < \theta_g - 150)$.

Probability distribution. The discretization has the additional advantage that we can assign a probability distribution over the discretized intervals. One might wonder why that is an advantage, since we showed that we ideally want a uniform probability distribution.

Choose probability distribution. Defining a probability distribution allows more control on the shaping process. We want the agent to focus more on states that are closer to the allowed deviation. The reasoning behind this decision is that the agent will travel anyway along states closer to a goal state (in successful episodes). To give more weight to states that are further away might thus be more fair, and allows the agent to focus on expanding its knowledge.

Two distributions. As we explained, we only hold one deviation, that expresses both the deviation in $[\theta_g - \text{deviation}, \theta_g]$ and also in $[\theta_g, \theta_g + \text{deviation}]$. We refer to the first interval as the *lower interval*, and the latter as the *higher interval*. Both intervals are adapted such that they remain in the valid domain of θ, i.e. $[0, \theta_g]$ and $[\theta_g, 1023]$.

The intervals need not be of equal size. For example, suppose the deviation is 300 and $\theta_g = 200$. The lower interval is adapted to $[0, \theta_g]$. The lower interval is now smaller than the higher interval. We still want to give higher probabilities to states further away from θ_g. Therefore it is easier just to take two discretizations, one for each interval. The sum of the probability distribution of the two discretizations should add to one (since only one state is chosen).

When the intervals are not of equal size, we adapt the probability distribution. The interval ratio for the example above is $200 : 300 = 0.4 : 0.6$. Refer to Table 4.1 for an example of a probability distribution for the two intervals. In this table, the weights 0.4, 0.3, 0.2 and 0.1 are used for the four subintervals from the lower and the higher interval.

Pobability distribution based on angle. In the example above we based the probability distribution on the interval ratio. This is fair since it keeps into account how many angles each interval represents. This ratio only plays a role when the lower and higher interval are not of equal size. We choose to use another ratio.

	lower domain			
equal ratio	0.5*0.4	0.5*0.3	0.5*0.2	0.5*0.1
skewed ratio	0.4*0.4	0.4*0.3	0.4*0.2	0.4*0.1
	higher domain			
equal ratio	0.5*0.1	0.5*0.2	0.5*0.3	0.5*0.4
skewed ratio	0.6*0.1	0.6*0.2	0.6*0.3	0.6*0.4

Table 4.1: Probability distribution.

We want to take the position of $\theta_g \in [0, 1023]$ into account. Once the controller is trained and actually used, the begin states are set by a user of the controller and not anymore by our shaping strategy. Suppose $\theta_g = 200$. We assume that begin states with higher angles are more relevant than lower angles. We guide the agent to focus more on the higher angles, by assigning a higher probability for the higher interval.

We express this thought by calculating the *ratio* between all angles lower and higher than θ_g:

$$ratio = \frac{\theta_g}{1023 - \theta_g} \tag{4.15}$$

Suppose again that $\theta_g = 200$. Then the ratio between lower angles and higher angles would be $\frac{200}{823} \approx 0.24$, so the ratio is $0.24 : 1 \approx 1 : 4$. The probability of a higher begin angle should then be four times as high. We normalize this ratio:

$$r = \begin{cases} \frac{ratio}{ratio+1}, & \text{lower domain} \\ \frac{1}{ratio+1},, & \text{higher domain} \end{cases} \tag{4.16}$$

We use two discretizations of 4 bins, with the weights 0.4, 0.3, 0.2 and 0.1, multiplied by the respective r. Furthermore, we only select begin states that have a velocity 0.

4.3.3 Alternatives

The shaping strategy we apply is just one of many. In this subsection we discuss some alternative shaping techniques. It is worthwhile to present some of them, since they might prove useful on other problems.

Depend on current position. Another strategy might be to let the exploration rate depend on the current θ_k. If this angle lies in the 'known' area, i.e. $\theta_k \in [\theta_g - \text{deviation}, \theta_g + \text{deviation}]$, then we set exploration rate ϵ low. If the angle is located outside this area, we may gradually increase ϵ. Intuitively, this makes sense. However, this strategy would only work for off-policy methods, like Q-learning. It does not work for on-policy methods like SARSA. The reason is that SARSA tries to estimate Q^π, i.e. the action-value function for policy π. If the exploration rate is changed continuously, the policy varies continuously as well. For SARSA we may only change the exploration rate gradually.

Store episodes. All episodes can be stored. This way, very efficient episodes can be revisited. That is, we abandon the policy for the moment, and only update the Q function with the actions on this path. This can help to speed up the convergence. The danger is, that states in the path will not be entirely the same. This implies that it is not guaranteed that a goal state is reached, even if we execute the exact same sequence of actions.

We can also focus on unsuccessful episodes, by starting again from these begin states. The idea is, that the agent will explore this environment, and eventually will find a goal state. However, as we discussed in our motivations for shaping, an RL agent learns backwards. Therefore, it is better to learn from positive reinforcements rather than negative reinforcements.

Cut-off episodes. We cut off episodes if the agent exceeds a certain number of steps. Aside from the fact that such an episode might never end, there is another reason to do so. The agent needs to keep on visiting goal states, as we already discussed. If the agent finds itself in unknown states, the probability of reaching a goal state shrinks, as a result of the trial-and-error approach. The Q values for these states are all negatively updated, since the agent has no experience (yet) that a goal state can easily be reached from these states. If the allowed deviation grows, the agent learns that an easy solution exists for states inside the deviation interval. The states that were negatively updated earlier, now need to be updated positively. We can avoid this a bit, by cutting off an episode if the agent deviates too much from the desired goal angle.

Shaping on the robot arm. The use of a simulation is beneficial for the shaping process, since then there is instantaneous control of the state. However, if there is no simulation, shaping can still be applied. For example, both muscles are inflated a random amount of milliseconds. Then, dependent on the allowed deviation, the goal angle is determined. This is just the other way around: in simulation we choose a goal and start close to this goal. On the real system we start in a random position, and set the goal close to this position. This does not work if the aim is to learn all goal states from one particular begin state (since we have no controller yet that brings the arm in this begin state). However, in general we are interested to learn to start from all possible states.

4.4 Suggestions

In this section we discuss some suggestions about how to improve the useability of the algorithm.

Performance guarantees. No performance guarantees can be given for the RL agent. Even if the agent performs very well, it does not always need to give satisfying results. We cannot even guarantee that the agent will not continuously let the arm bump against the construction.

Of course, this limits the practical application possibilities of such a controller. However, there are methods that allow to prove guarantees. In [18], Lyapunov Design is used to limit the action choices in order to guarantee safe behavior. A Lyapunov function must be designed. This is a function that descends one every step. In a bounded number of steps a limit should be reached.

It requires more system knowledge to construct such a function. This is a violation of our research aim. However, such a design might be necessary for applications that require a minimal performance.

Multiple updates. In [32], every single action leads to multiple updates of the Q function. We can apply a similar idea. In our case, the state is parameterized with the goal angle θ_g. That means that every update $Q(\sigma_k, \alpha_k)$ can be applied for all states where only the goal angle θ_g is different. The reward may differ for the different goal angles, but the transition to the state remains the same.

Speed-up the learning. An agent that estimates an internal model of the environment might speed-up the learning. Note that such a model is different from our empirical simulation. In our case, the agent is not aware whether it operates on the robot arm or in simulation. An example of an RL agent that estimates a model is Dyna [23, Chapter 9]. Another way is the re-use of important previous experiences [2].

We refrained from using an incorporated model, since the learning speed is no bottleneck. This is because we were able to derive a simulation. However, if we do not have a simulation, the incorporation of an internal model might greatly enhance the learning speed.

Inverse simulation. In chapter 3 the empirical simulation is explained. This simulation performs a mapping $(\sigma_k, \alpha_k) \rightarrow o_{k+1}$, or equally, $(\sigma_k, \alpha_k) \rightarrow \sigma_{k+1}$. In the experiments in chapter 5 we show that this simulation is able to predict the robot arm very well.

This idea can also be employed in reverse direction:

$$(\sigma_{k+1}, \alpha_k) \rightarrow \sigma_k \tag{4.17}$$

In words: if the current state is σ_{k+1} and action α_k has been executed, then σ_k is a possible previous state. This allows to calculate the states the agent might have come from.

This can help an RL agent in the following way. Suppose σ_{k+1} is a goal state. Then:

$$\begin{aligned} Q(\sigma_{k+1}, \alpha_{k+1}) &= 0 \\ Q(\sigma_k, \alpha_k) &= 50 \end{aligned}$$

The first equation holds by definition. The second equation is a result of the knowledge that action α_k directly leads to goal state σ_{k+1}. This idea can be applied iteratively for timestep $k-1, k-2$, etc.:

$$\begin{aligned} Q(\sigma_{k-1}, \alpha_{k-1}) &= \gamma 50 - 1 \\ Q(\sigma_{k-2}, \alpha_{k-2}) &= \gamma(\gamma 50 - 1) - 1 \end{aligned}$$

The inverse simulation can be learned, just like the normal simulation. This way, a cloud can be predicted of all states that lie on a distance of a certain number of actions from a goal state. Suppose we calculate this cloud for five actions. Then we predict $9 + 9^2 + 9^3 + 9^4 + 9^5 = 66,429$ possible paths for a single goal state. This might be done for many goal states. The big advantage is that we can predict a huge number of 'free' Q updates already. This way, good initial predictions for Q can be found.

A possible problem might be that the inverse simulation does not exist for all actions. For example, consider a state at timestep $k + 1$ where muscle $1a$ is fully inflated and muscle $1b$ is empty. Then the action that lets air out of muscle $1a$ and lets air flow to muscle $1b$ could not have been executed at timestep k.

The effect is basically the same as the shaping strategy, of starting close to goal angles. Since this way too, quickly a lot of useful updates are made. Also, the effect can be compared with revisiting efficient episodes, since this method also re-uses experiences [2].

Tracking control. In the current solution, we defined a point-to-point movement as a movement that ends when a goal state is reached. This allows for a straightforward separation into episodes. However, classical control theory implements point-to-point movements as a controller that tracks the system continuously. A bandwidth is specified, where θ_k should stay within. The controller's task is to assure that the arm stays within this area. Note that the criterium on ω becomes obsolete in this case.

To allow for a better comparison between an RL agent controller and a classical controller, it would be interesting to implement this definition as well. In this case, there are no terminal states anymore. In theory, the episodes do not end anymore, i.e. become infinite. To train, one can just cut-off an episode and start a new one to train on a different goal angle.

The reward function needs to be revisited. It might be changed to a punishment on angular distance to the goal angle, i.e. $\theta_k - \theta_g$. This way, the RL agent tries to avoid future punishments by staying as close to the goal angle as possible. The punishment on a switch might not be needed anymore, because to stay in the bandwidth corrections might be needed continuously.

One might think that the current approach is also able to do this tracking control. One could just use an infinite loop: every time a goal is reached, just start a new episode. This does not work well, since the controller will do nothing as long as the arm is in a goal state. The controller only starts intervening when the arm gets out of the goal state. A tracking controller, on the other hand, would make continuous corrections in order keep the arm positioned within the bandwidth.

The use of a tracking controller makes the robot arm more useable in practice. Furthermore, the problem of ω is not key anymore. Still, a good estimation of the velocity is useful, but it is not used as a criterium for a goal state anymore.

4.5 Evolutionary algorithms

4.5.1 Introduction

In general, Reinforcement Learning problems can be solved in two ways. One solution is to search in the value function space, i.e. to estimate the value function. This is the way as we discussed it in Section 4.2. Another solution is to search in the policy space. That means that a direct mapping from states to actions is searched. Evolutionary algorithms search in the policy space.

Evolutionary algorithms are based on an analogy with natural selection. Every iteration, a new population of solutions is created using genetic operations such as mutation and cross-over. The best performing solutions are combined and changed, in order to form a new population. The performance of a solution is evaluated by means of a *fitness function*, that is specified by the human designer.

4.5.2 ESP

In particular, we looked at a neuro-evolution method called Enforced Sub-Populations (ESP) [10]. The term *neuro-evolution* means that neural networks are evolved. The goal of ESP is to find a neural network that maps a state σ to the best action α.

ESP works by keeping different subpopulations that contain neurons with weighted connections to inputs and outputs. Neurons from the subpopulations are selected and used to generate feedforward neural networks. A neuron is assigned to different combinations with neurons from other subpopulations. This way, the average fitness is determined for each neuron. Using evolutionary operations like crossover and mutation new subpopulations with neurons are generated.

The prospects of ESP are promising. In [10], ESP outperforms SARSA and Q-learning on many difficult control problems. We list the following advantages of ESP with respect to RL. First, ESP may find a reasonable controller more quickly. It might be able to find good neural network weights more quickly, whereas RL has to start with the randomly assigned initial weights. Another advantage is that ESP seems to work well, even if the state is non-Markov. The ability to work well on non-Markov problems comes from the fact that ESP evolves recurrent neural networks [11]. These networks are able to take *hidden states* into account. Hidden states are states of the system that are not captured in the state representation. Recurrent networks use weights that go back to previous layers. That enables them to keep track of history. Finally, ESP has been shown to be suitable for a multi-agent approach, i.e. agents that need to co-operate.

The advantages of RL with respect to Evolutionary algorithms in general are as follows. First, RL leans on every step. An evolutionary algorithm throws away a lot of information when a new population is formed. Next, RL is better suited for stochastic problems. For evolutionary algorithms it would imply that every individual's fitness should be tested many times. Another reason is that

Evolutionary algorithms are not suited for fine tuning. This is because the genetic operations result in a high probability that the result is worse. Finally, Reinforcement Learning keeps on learning after transfer. Evolutionary algorithms cannot learn unless different populations are tried on the robot arm.

Apart from the above advantages of RL, we have chosen to apply RL above some Evolutionary algorithms because of the following reasons. First, ESP is relatively new with only few publications in the literature yet. Second, earlier work at Philips Applied Technologies (AppTech) showed good results using RL [19]. Third, earlier work at AppTech showed that Genetic Programming (another form of an Evolutionary approach) is still in its infancy with respect to non-linear controllers [4].

Note that a combination between an Evolutionary approach and Reinforcement Learning might be possible. First, an Evolutionary algorithm can find a reasonable controller. Next, RL can be used for fine tuning. Care must be taken to assure that the solution of the Evolutionary algorithm is in line with the reward function. That is, the solution of the Evolutionary algorithm should lead to the highest rewards for the RL agent.

4.6 Conclusion

In this chapter we discuss Reinforcement Learning (RL) and give motivations to use this method above Evolutionary algorithms. We explain the theory and show how we apply RL on the robot arm. We use SARSA or Q-learning and use one neural network per action. We also give suggestions to improve the current approach.

Furthermore, we discuss a shaping strategy to guide the learning process. We show that this shaping strategy is useful, since the agent needs to learn backwards (with the reward function we use). Also, the strategy is used to keep on visiting states. This is necessary, since it avoids unlearning, lets the agent improve and is part of the convergence proofs. The strategy might improve the learning speed of the agent.

Chapter 5

Experiments

In this chapter we discuss the experiments we performed for benchmarking our approach. The first section discusses the empirical simulation. In the second section, the control factors and the benchmarking are treated. Second three explains how the experimental set up is constructed. Then, section four describes the experiments in simulation. Section five discusses the transfer experiments. After that, section six discusses the general results and conclusions, with a reflection to the research questions.

5.1 Empirical simulation

In this section we discuss the issues related to the empirical simulation. We start by deciding at what frequency we control the robot arm. Based on this result, we derive the simulation. We end this section by defining the goal state area.

Control frequency. We want a high control frequency for fast control, but should be aware there are limitations.

One limit is partly due to our definition of state, since we cannot put too many timesteps in the state. Therefore, the imposed delay between these timesteps should be chosen wisely. If we *measure* the system with a too low Δt, only the high frequency effects are represented in the state. This is disastrous, since then the interesting dynamic behavior is not included anymore, making the representation useless. The resistance of the air pipes is another reason a too high frequency is prohibitive. The pipes impose a delay on the air flow to the muscles. If we measure faster than this delay, this effect is not represented anymore. The delay is about 10 ms, refer to Appendix A.

There is another reason why we should be careful at selecting the control frequency. If we *control* the muscles at a frequency around one of the system's eigenfrequencies, we may bounce the system into this eigenfrequency. This is unwanted behavior since it is difficult to compensate for.

Classical control theory solves this problem by performing a thorough analysis of the system. However, we cannot do so. Instead, we try two frequencies and construct empirical simulations out of them. Based on the simulation's quality analysis (see Chapter 3) we decide what frequency to use.

Approximately 7 ms is needed to read the sensors, select an action, write information to the database and to execute the action. By imposing a delay of 43, the control frequency is set to 20 Hz. Similarly, a delay of 18 ms results in a frequency of 40 Hz. For both delays, we fill the database using the collection procedure as we discussed in Chapter 3.

The simulations we use have 10 neurons and are trained with the Levenberg-Marquardt algorithm. We use pre- and postprocessing of the input and targets. We set the learning rate to 0.05 and the number of epochs (i.e. the number of times the training batch is visited) to 15.

frequency	θ	ω	ρ_1	ρ_2
20	15.04 (67.26)	0.26	2.86 (12.79)	2.85 (12.74)
40	9.67 (61.16)	0.32	1.87 (11.83)	1.86 (11.76)

Table 5.1: Standard deviations.

Table 5.1 shows the standard deviations of the errors (refer to Chapter 3). The numbers between brackets denote the expected standard deviations after one second, according to Equation 3.7. This equation gives an estimation of the error propagation in simulation.

The sensor values are better predicted at a frequency of 40 Hz. However, the derived state variable ω is better at 20 Hz. This is remarkable, since this value is derived from θ, and this latter value is better at 40 Hz. A reason could be, that the time interval Δt has a relatively higher deviation on small intervals.

We choose to perform our experiments at a frequency of 30 Hz. This seems to be a nice trade-off between the derivation of ω and predicting the sensor values. The standard deviations of this simulation are shown in Table 5.2. Table 5.2 shows different results than Table 5.1. This is because other muscles are used and a new potmeter has been installed.

frequency	θ	ω	ρ_1	ρ_2
30	6.70 (36.70)	0.19	1.94 (10.63)	1.52 (8.33)

Table 5.2: Standard deviation of chosen simulation.

Poor ω estimation. The estimation of ω remains a problem. There are various reasons for this.

The main cause is the potmeter. The potmeter's axis is partly placed inside the (concave) joint's axis. In order to measure the joint's angle, the potmeter's axis should rotate together with the joint's axis. Currently, a screw is put inside the joint's axis. This screw pushes against the potmeter's axis. This makes the movements of the potmeter axis overdetermined since the potmeter's housing is also fixed (to the frame). This results in wear. A simple junction to connect both axis would solve the problem. During the experiments, the potmeter showed wear after only 10 hours of experiments already. This makes an accurate prediction impossible.

Second, the numerical derivation of ω is poor. The approximation $\omega = \frac{\theta_k - \theta_{k-1}}{\Delta t}$ is not ideal. The approximation represents the *average* rotational velocity. Hence, a bigger Δt implies a worse estimation of the velocity at timestamp k. However, a smaller Δt implies a bigger influence of sensor noise on θ. In classical control theory one usually samples (i.e. measures) the system at a frequency ten times higher than the control frequency. This can be done with the current system as well, by using the idle time of the chip to continuously measure the sensors, and possibly apply a filter on the values. Or more simple, a moving average of the ω estimation. The use of a filter has the drawback that it introduces a delay, since the filter is based on multiple timesteps.

Third, another problem that is encountered is the use of a non real-time operating system (Linux, no Real-Time Linux). The standard deviation of Δt can be derived from the database: about 1.5 ms.

Definition of the goal state area. So far we did not define the goal state area, in terms of θ and ω units. Recall that 11.37 θ units equals 1° (refer to Section 2.3.3). We define the goal state area as follows:

$$\theta_g - 25 \leq \quad \theta \quad \leq \theta_g + 25 \tag{5.1}$$

$$-1 \leq \quad \omega \quad \leq 1 \tag{5.2}$$

This goal state area is partly determined by Table 5.2, and partly by empirical observations. Empirically we found that these values are suitable with respect to learning time in simulation and transfer possibilities.

The goal state area is rather big. For the angle it means that the deviation may not be more than 2°. However, one should keep in mind that the muscles also impose a limitation on this deviation. Very accurate movements are only possible with very stiff muscles (note the analogy with human muscles).

When an episode ends, the velocity may still be as much as 1 $\theta/$ ms = 1000 $\theta/$ second $\approx 90°$ per second. This is a high rotational velocity. One might think that the controller only needs to bring the arm to the right direction, without taking care of slowing the arm down. This is not true, as the database shows that typical episodes have velocities of $\omega > 3$. With the current system, we may be able to push the criterium to 0.5 $\theta/$ ms, but this would still be a rather big value. As we explained, other measures are necessary to improve the criterium significantly. The learning method will stay exactly the same.

Empirically we found that many episodes end with a lower velocity than $|\omega| = 1$. A reason could be that episodes with such velocities often miss the goal area. In simulation this is due to the inserted noise on θ. On the robot arm this can be due to the sensor noise. Another reason could be that the robot arm is not perfectly deterministic.

5.2 Control factors and benchmarking

In this section we explain the training and testing phase. Next, we discuss the parameters we may set. Finally, we discuss how we benchmark the performance.

Training and testing phase. The *training phase* starts with the agent's first episode, having the minimum deviation. The training phase ends when two criteria are satisfied:

- The whole domain of θ is known, i.e. the deviation equals 1024.
- The agent's performance satisfies at least its criterium (see Section 4.3.2.1).

The *testing phase* is performed after the agent has been trained. This phase might be done in simulation or on the real system. The learning rate β is set to 0 to refrain the agent from learning during the benchmarking. We set the number of episodes to 10,000 in simulation.

Parameters. In this paragraph we identify the parameters we may set in the experiments. We make a distinction between *control factors* and *noise factors*. As the name already suggests, control factors are parameters that can be set. Noise factors are those that are too difficult, too expensive or impossible to set.

In our case the *control factors* are: λ for eligibility traces, discount factor γ, number of neurons of the NNs, learning rate β, choice between SARSA and Q-learning, number of actions we allow before we cut off an unsuccessful episode, reward function, control frequency (i.e. the imposed delay between actions) and the policy.

In fact, we have more if we also take our shaping strategy and simulation into account. For the shaping strategy we have the adaption of the deviation, performance criterium, exploration rate ϵ, temperature T and the number of steps in an episode. For the simulation we have the number of neurons, the filtering procedure, and the amount of noise we add (and some others, like the learning rate, the number of epochs and the training algorithm).

Our *noise factors* include: initial (randomly assigned) weights of the NNs, exploration steps and the begin states of episodes. Only the first noise factor is really out of our control. The effect of exploration steps cannot be foreseen, but we can control their amount and randomness by setting ϵ and T. The begin states of episodes are picked randomly. However, as we already explained, we have influence on them via the concept of deviation and the probability distribution.

Benchmarking. In order to make a comparison between different controllers possible, we need a way to benchmark their performance. This benchmarking is performed in the testing phase.

The RL agent's goal is to obtain as much reward as possible. Therefore, the most interesting performance characteristic is the *average cumulative reward* per episode. By definition, this is determined by the average number of actions and switches per episode and the percentage of episodes that reaches a goal state.

To allow a comparison with other control methods, we also measure the percentage of successful episodes, the average number of actions (in an episode) and the average number of switches (in an episode). We define a *switch* as a change in valve configuration. This gives an idea about the smoothness of the movement. A final benchmarking criterium is the number of episodes needed for the training phase.

To allow for a statistical comparison between different controllers, we record 10,000 point-to-point movements. The movements start in all possible begin states and head for all possible goal angles.

5.3 Design of Experiments

In the previous section we discussed the control and noise factors we deal with. In this section, we follow a structured approach (the Taguchi method — see below) in order to set up our experiment framework.

Structured experiments. There are many parameters at stake. It would be virtually impossible to perform an exhaustive test of all possible combinations — even if we discretize continuous parameters like γ. Therefore, we want a structured way in order to set up a testing framework. The purpose of such a framework is to minimize the number of experiments while still being able to derive important conclusions. We can derive such frameworks by applying theory of *Design of Experiments* (DOE). [1] describes DOE as follows: "DOE is a powerful statistical technique for determining the optimal factor settings of a process and thereby achieving improved process performance, reduced process variability and improved manufacturability of products and processes."

There are multiple DOE methods occurring abundantly in the literature, among which the Taguchi method and the (fractional) factorial design method [14]. We use the Taguchi method, mainly because of the intuitive plots that are generated. A clear introductory tutorial can be found in [1]. For more detailed information we refer to [24].

Two levels. We only use two values for each control factor. In Taguchi terminology this is referred to as two *levels*. Using more levels would imply a more complicated analysis and more time needed for the experiments. The use of two levels should be considered an *awareness-raising session*, just to give an idea about the most important control factors and their desired order of magnitude.

Five control factors. Ideally, we would like to measure the influences of all control factors and all relevant interactions between them. An *interaction* between two factors exists if the effect of a factor is not the same at different levels of the other factor. In fact, we speak about 2-interactions in this case. Interaction between more factors is possible as well.

In the general case, all possible combinations of the control factors should be considered in the experiments. With two levels that would mean 2^c experiments, with c the number of control factors. An important property of the Taguchi method is that one can exploit a-priori knowledge. This knowledge consists of the interactions the designer expects. This way, only a carefully selected subset of all combinations may be considered, saving much time and effort.

However, one should be very careful with assigning expected interactions. Interactions that are not specified are assumed to be non-existent. In our case, we want to let the experiments point out the measured interactions. Of course, this decision comes at a cost: we cannot benefit from a small subset of experiments. Therefore, we choose to strip the number of control factors to five. In this section we derive an experimental set up. In the subsequent sections, this set up is used for different experiments.

Orthogonal array. After the investigation of the control factors and their interactions, the next step in the Taguchi method is to find the most suitable orthogonal array. After that, the control factors and interactions are assigned to the array's columns.

Since we use five control factors and want to investigate all 2-interactions, we choose the standard L_{16} as orthogonal array. Refer to Table 5.3. The elements -1 and 1 represent the two levels of the control factors, with -1 being the lower level and 1 the higher level. In the header of the array we label the five control factors as a, b, c, d and e and show all 2-interactions between them. These control factors and 2-interactions are assigned to the corresponding array columns. The rows represent the trials (experiments) that are to be done. At the right of the matrix, we write the result of the trial (the average number of actions etc.).

From Table 5.3, it is clear to see why L_{16} is suitable for our purpose. The array contains all 2-interactions between columns 1, 2, 4, 11 and 12 (see the corresponding labels). The 2-interactions obey a XOR-relation, see columns 1, 2 and 3 for example. The term *orthogonal array* comes from the fact that the inproduct of every two columns is 0, i.e. the columns are perpendicular to each other. The array L_{16} takes only the control factors and 2-interactions into account. If we would test all combinations exhaustively, it would require $2^5 = 32$ trials.

	a	b	ab	c	ac	bc	de	ce	bd	ad	d	e	ae	be	cd
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	1
3	-1	-1	-1	1	1	1	1	-1	-1	-1	-1	1	1	1	1
4	-1	-1	-1	1	1	1	1	1	1	1	1	-1	-1	-1	-1
5	-1	1	1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1
6	-1	1	1	-1	-1	1	1	1	1	-1	-1	1	1	-1	-1
7	-1	1	1	1	1	-1	-1	-1	-1	1	1	1	1	-1	-1
8	-1	1	1	1	1	-1	-1	1	1	-1	-1	-1	-1	1	1
9	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1
10	1	-1	1	-1	1	-1	1	1	-1	1	-1	1	-1	1	-1
11	1	-1	1	1	-1	1	-1	-1	1	-1	1	1	-1	1	-1
12	1	-1	1	1	-1	1	-1	1	-1	1	-1	-1	1	-1	1
13	1	1	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	-1
14	1	1	-1	-1	1	1	-1	1	-1	-1	1	1	-1	-1	1
15	1	1	-1	1	-1	-1	1	-1	1	1	-1	1	-1	-1	1
16	1	1	-1	1	-1	-1	1	1	-1	-1	1	-1	1	1	-1

Table 5.3: Orthogonal Array L16 with assigned columns.

Run. We replicate the trials, in order to take the variation in results into account. This variation occurs due to noise. We call every set of 16 trials a *run*. The initial weights of the NNs are the most important noise factor. Within a run, we use the same initial NNs at the start of every trial. Since there is a distinction between 5 and 10 neurons, we generate two sets of nine NNs per run. The weights of all NNs are initialized by assigning random numbers in $[-0.3, 0.3]$.

Analysis and interpretation. After all experiments are done, the statistical analysis and interpretation is performed. In this paragraph we briefly explain how this is done with the Taguchi method.

Loss function. Every trial number has n results, with n the number of runs that are performed. For every trial number, its *loss value* is calculated. The equation for the loss function differs, depending on the desired outcome of the results.

Taguchi proposes three standard loss functions, that all use a logarithmic scale. Taguchi designed these equations for cases in which the costs cannot easily be estimated. For example, suppose a product does not meet its specifications completely. If the error is rather small, it may not have a big influence. However, if the error is a bit bigger, it might induce major costs and consequences. In such cases, a small difference in result may have a significant influence. His equations are based on empirical studies. In our case, the quality loss can easily be expressed in the amount of cumulative reward per episode the agent gets. There is no reason to assume that the quality loss is quadratic or exponential. Therefore we assume a linear relation.

We use two quality characteristics. One is the *mean result* for each trial t. The other is the *standard deviation* for every trial t. An important advantage is that we compare our results with their original units (i.e. in cumulative reward). This makes the analysis more intuitive.

Note that Taguchi refers to his equations as *Signal-To-Noise* ratios. Since the mean and standard deviation are no ratios, we simply use the term *loss* in the remainder of this chapter.

Calculate effects. The loss functions give a characteristic value for each trial. The 16 values of each loss function together form the *loss table*. We derive two loss tables: one for the mean and one for the standard deviation.

The next step is to calculate the *average loss table*. This table contains two values for each $L16$ column, i.e. for each control factor and 2-interaction. The two values represent the average loss value of the lower and higher level of the factor or 2-interaction. As an example, refer to Table 5.3 again. For control factor a in column 1, the loss average of the first eight trials represents the loss average for the lower level (-1). Similarly, the second half of the trials is used to calculate the average loss for the higher level (1). If these two values differ significantly, the influence of that control factor or interaction is significant.

Visualization. The next step is the identification of significant control and interaction effects that influence the loss value. To this purpose, the average loss table is visualized. This plot is called the *effects plot*. This plot consists of 15 subplots, one for each column. In every subplot the two average values of that column are linearly connected. The more a line is tilted, the more the two values differ, and the more significant the influence of that column is. An example is shown in Figure 5.2. Note that the connection line might suggest that a linear relation exists between the two values. This is *not* the case. It might just as well be that an intermediate level results in a very different average loss value. Once again, this experiment with two levels is meant to get an estimation of the main influences. For fine tuning purposes the experiment should be performed with levels that lie closer to each other, or with the use of more levels.

The *estimated effect* is defined as the absolute difference between the two average loss values. A plot that intuitively visualizes these estimated effects is called a *half-normal probability plot*. In the remainder we sometimes abbreviate this to *hnpp*. This plot contains 15 elements, that are associated with the columns. The columns are ranked to their estimated effects, in ascending order. The x-axis represents the estimated effects. The bigger the estimated effect, the further it is plotted to the right. The y-axis represents the ranking of the columns. The higher the ranking, the higher on the y-axis. The y-values are expressed in the so-called per cent probability:

$$P_i = \frac{i - 0.5}{15} \times 100 \tag{5.3}$$

where i represents the rank of the column. In this plot, the statistical insignificant control factors and interactions fall on the straight line, whereas the important effects deviate from this line. For examples we refer to Figure 5.4 and Figure 5.5.

Experimental set up. In this section we derived the experimental set up. This set up is used in the following sections. The five control factors are determined and associated with the labels a, b, c, d and e. Using the analysis and visualization tools, important conclusions are derived about interactions. Also, the optimal settings are determined.

5.4 Experiments in simulation

In the previous section we set up the experiment framework. In this section we perform experiments using this test set up, and interpret the results.

The experiments are performed on an Intel Pentium III processor at 1200 MHz with 256 MB RAM. The database that we use is MySQL. The filtering and function fitting are performed in Matlab. The learning algorithm is written in C.

5.4.1 Punish a switch

One of the benchmark criteria is the average number of switches. A switch is not desirable because it conflicts with a smooth movement: more valve configuration changes lead to a more ‘nervous’ behavior. If we judge the agent on the average number of switches it uses, we should express this in the reward function. We experimented with a punishment of -8 versus no punishment.

Since we are only interested in the above question, we do not discuss this experiment in detail. The experiment and its results are shown in Appendix C. The conclusion is that the agent uses a few more actions if a switch is punished. However, if no punishment is given, the amount of switches is more than doubled. Punishing a switch decreases the number of episodes needed for training. This can be explained, because the number of efficient paths to a goal state is diminished. This comes from the fact that the punishment implies a moment of inertia on the action choice: a wrong action is easily spotted, since it rapidly leads away from a goal state. Furthermore, in preliminary experiments, we found that a switch punishment improved the transfer to the robot arm. This can be explained from the fact that the agent depends less on the precise simulation predictions. A switch on almost every timestep can never be a good solution, but it might give the best results in simulation.

Note that a punishment on a switch is a potentially dangerous bias of the reward function. To avoid the punishments, the agent might learn that doing nothing is the best solution. However, since we force the policy to take exploration steps, this drawback is circumvented.

To summarize, we let the experiments continue with a punishment of -8 on a switch. Note that we bias the reward function by putting a punishment on a switch. With this punishment, we have introduced a trade-off between speed and smoothness.

5.4.2 All goal angles

In the previous subsection we concluded that a punishment on a switch is desirable. We continue with determining the influence of five control factors and their interactions, and determine their optimal settings.

Control factors and their levels. In this paragraph we discuss the five control factors we use.

First, we investigate the influence of the learning rate β of the NNs. Second, the number of neurons of the NNs. This parameter can be considered an indication of the difficulty of estimating Q. Next, a very interesting question is whether SARSA or Q-learning is more suitable for the problem at hand. Does Q-learning indeed suffer from divergence (see Section 4.2.6)? Furthermore, we include λ to investigate whether eligibility traces help to relief the temporal-credit assignment problem. Finally, we investigate how far the agent should plan ahead, by including γ in the investigation. These five factors, with their abbreviation and two levels, are shown in Table 5.4.

Control factor	Label	Level -1	Level +1
learning rate	a	0.05	0.10
number of neurons	b	5	10
TD-method	c	SARSA	Q-learning
eligibility traces	d	0	0.3
discount factor	e	0.95	0.98

Table 5.4: Five control factors with their levels.

All other control factors remain fixed. The number of actions in an episode should be larger than $\frac{1}{1-\gamma}$. We set the number of steps in an episode to 100. The punishment on a switch is set to -8.

Analysis of cumulative reward. First, we focus on the average cumulative reward per episode. This criterium allows to compare the differences in learning abilities between the different control factor settings. The results and plots of all performance criteria are included in Appendix C.2.

Table 5.5 shows the average cumulative reward (of the testing phase) for eight runs. The differences in results are significant, since the differences can be as much as 50 reward units (to put this in perspective, this equals 50 actions, or 6 switches, or a combination). Taken into account that the results represent the *average* cumulative reward per episode, the selection of the right settings may be very rewarding.

trial	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
1	4.75	9.34	0.55	3.59	4.56	2.52	5.09	6.83
2	2.90	7.81	-1.35	7.03	4.80	6.69	4.76	8.15
3	10.24	10.10	-0.14	9.12	-5.89	10.43	11.30	-10.62
4	0.17	4.06	2.84	3.06	3.29	4.51	3.08	3.41
5	8.26	4.60	-0.01	3.07	5.50	8.62	-6.53	-1.46
6	12.30	9.18	8.86	11.42	0.67	10.60	8.86	11.92
7	8.00	10.66	-6.27	9.74	6.23	8.13	-25.85	9.12
8	1.10	-8.18	9.66	2.10	-8.11	-0.90	-17.39	3.90
9	-1.54	6.58	2.67	-11.17	8.23	7.24	5.40	8.81
10	11.00	9.81	12.10	11.54	9.27	8.21	10.53	8.98
11	2.10	9.22	1.23	9.15	-8.74	-10.12	-14.69	7.69
12	8.64	2.39	-2.12	4.20	4.26	5.78	-1.20	-7.74
13	8.57	10.60	6.80	-4.40	2.80	8.36	2.39	-6.64
14	5.66	8.37	5.26	8.57	6.88	1.00	7.27	9.70
15	-2.01	-36.70	6.84	3.82	8.48	-10.58	-2.53	2.89
16	-0.19	-8.72	-0.43	6.84	-3.99	-15.81	-22.67	0.31

Table 5.5: Average cumulative reward per episode in the testing phase.

Figure 5.1 shows the main effects plot for all runs *separately*. Every run is represented with a color. This figure reveals that the lines have roughly the same order between all subplots. For example, the red line (run 1) lies in most subplots on top, whereas the black line (run 7) is usually below. This is a striking result. It implies that the initial weights of the neural networks are the most important factor. The figure also shows that the steepness of the lines differs per run. This means that the sensitivity of the control factors and interactions is dependent on the initial weights as well.

The main conclusion is that the initial weights of the neural networks are dominating. This is not surprising. The fact that the initial weights have a big influence on the solution (a local minimum) is a known problem of neural networks. A common approach is to try several different (randomly assigned) initial weights and to choose the best performing neural network among them.

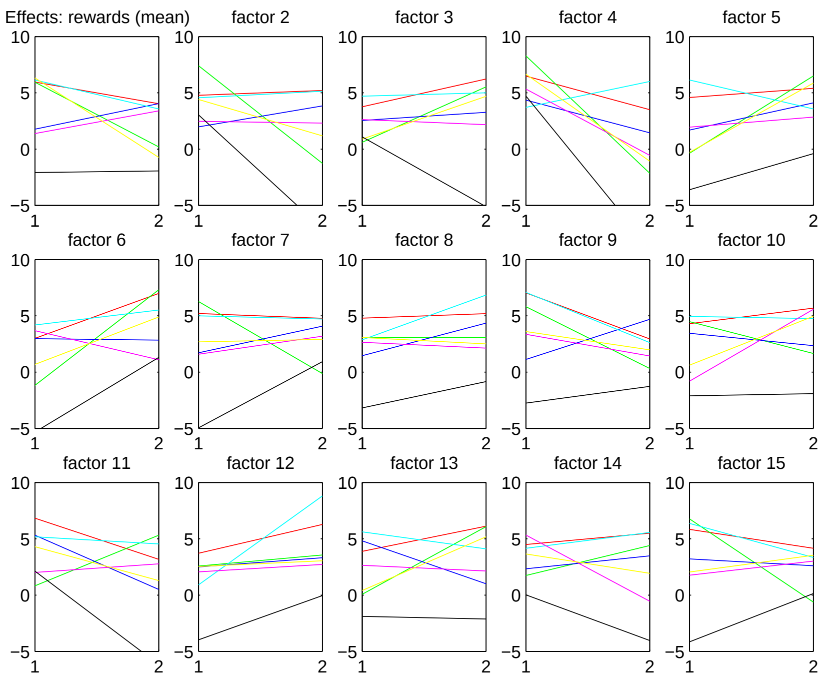

Figure 5.1: Effects plot for all runs separately.

Control and interaction effects. We make two effect plots of the cumulative average reward. Figure 5.2 shows the mean, whereas Figure 5.3 depicts the standard deviation. The associated half-normal probability plots are depicted in Figure 5.4 and Figure 5.5, respectively. In this paragraph we discuss some notable results. In the next paragraph, we determine the optimal settings.

Studying both figures, we conclude there are not many significant factors. This can be verified by looking at the half-normal probability plots. The biggest difference for both the mean and the standard deviation is at subplot 4. This column is associated with the TD method: the lower level is associated with SARSA and the higher level represents Q-learning. The difference in mean between these two methods is about 5 reward units, in the favor of SARSA. Moreover, the standard deviation is lower for SARSA as well. We expected these results, since SARSA takes the action selection method into account (it is on-policy), and therefore its average reward is likely to be higher. Analogous results have been observed in [23, Chapter 6.5] (at the cliff-walking problem).

For the standard deviation, a second notable difference occurs at column 8. This column represents an interaction between columns 4 and 12 (refer to the orthogonal array in Table 5.3), i.e. between the TD method and the discount factor. Column 8 shows a big standard deviation at its lower level. The lower level of this interaction represents the cases where columns 4 and 12 have the same level: SARSA with $\gamma = 0.95$ or Q-learning with $\gamma = 0.98$. Basically, column 8 tells that we should avoid these two combinations. A possible explanation could be, that SARSA needs to be able to plan further ahead, since it explicitly estimates the followed policy. Q-learning learns off

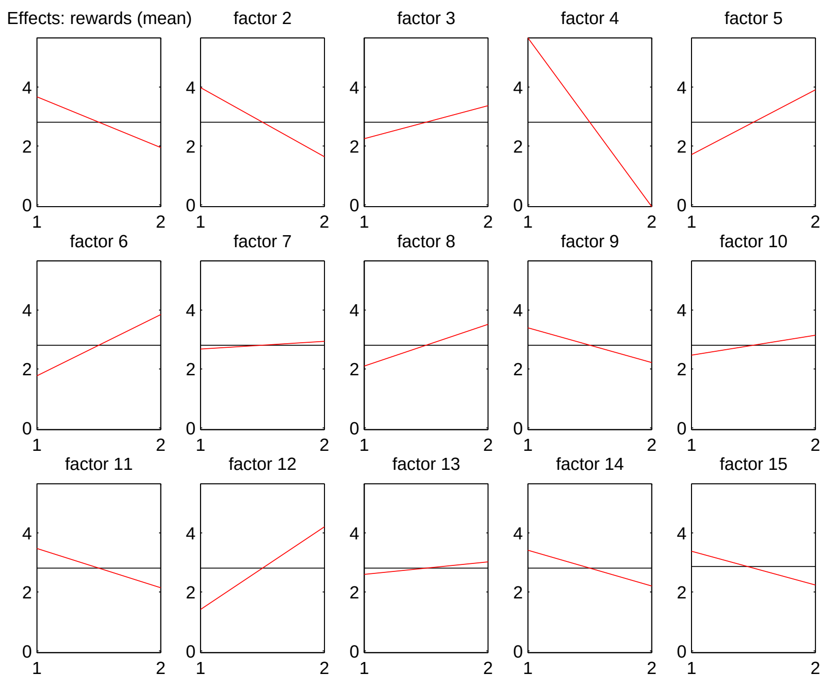

Figure 5.2: Effects plot of the mean (for the cumulative reward).

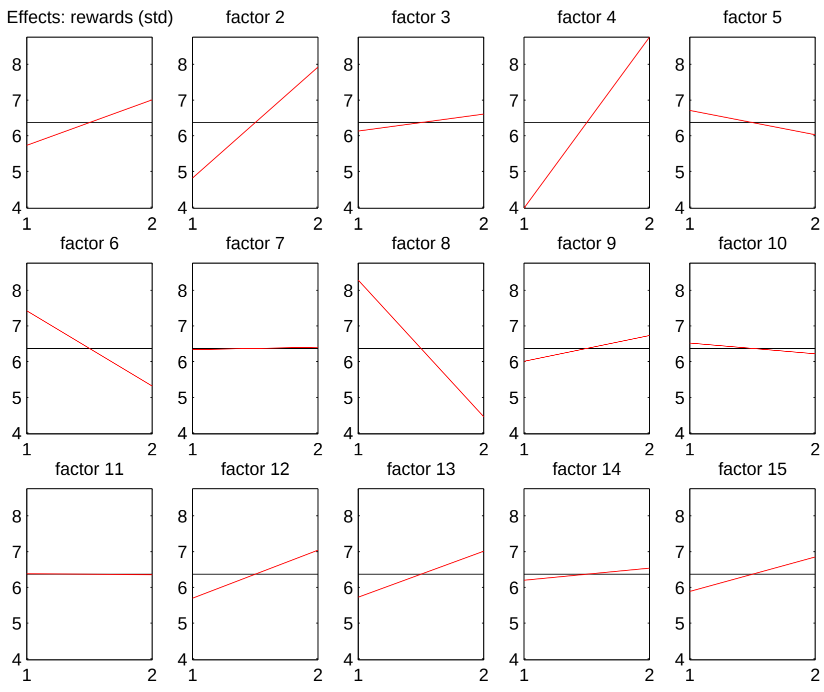

Figure 5.3: Effects plot of the standard deviation (for the cumulative reward).

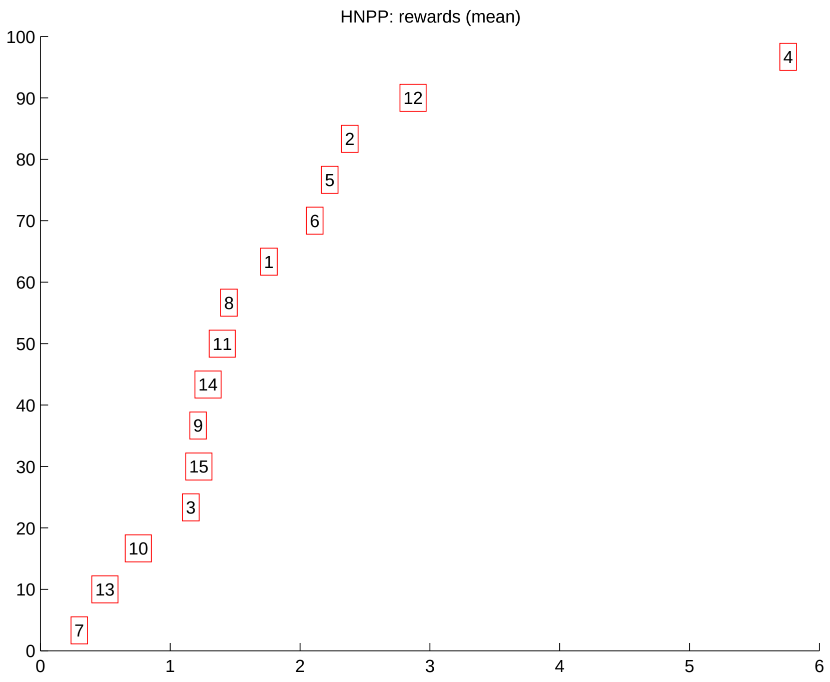

Figure 5.4: Hnpp of the mean (for the cumulative reward).

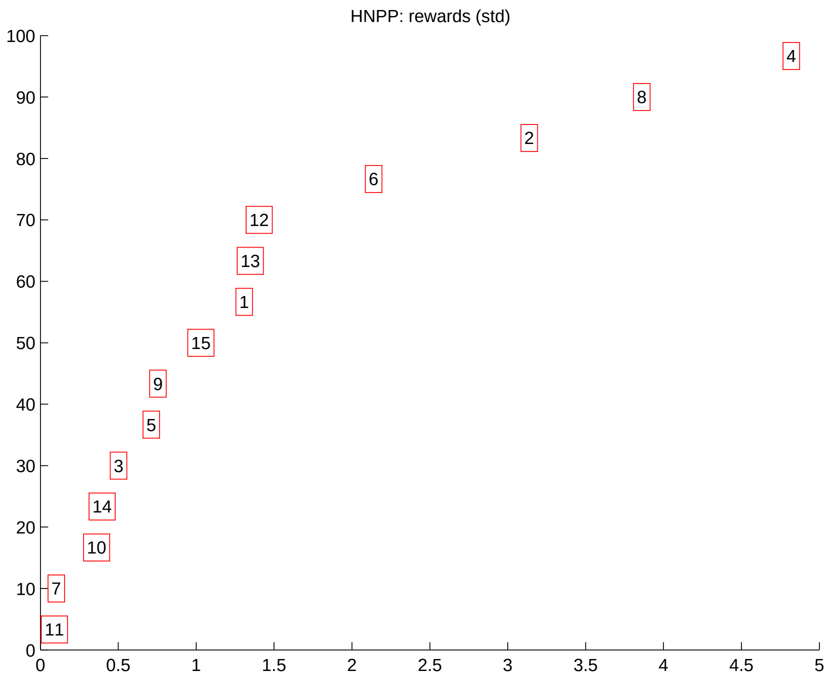

Figure 5.5: Hnpp of the standard deviation (for the cumulative reward).

policy. Therefore, looking too far ahead might be disadvantageous, since it becomes increasingly more difficult to abstract from the policy.

Finally, column 2 is the last significant effect for the standard deviation. The use of 5 neurons is favored above 10 neurons (column 2). Both the mean and standard deviation plots support this claim. This is logical with respect to the standard deviation, since less neurons implies more generalization of the function to approximate. For the mean however, we expected better performance for 10 neurons, since this is a more powerful FA.

The other effects are less significant. There is a slight advantage to use a smaller learning rate of 0.05 (column 1), both in terms of the mean and the standard deviation. This makes sense, since generally it is safer to use lower learning rates: it may be slower, but the result is usually more stable.

According to both plots, the effect of eligibility traces (column 11) is only marginal. The mean has a slight preference for $\lambda = 0$, whereas the standard deviation has a slight preference for $\lambda = 0.3$. This marginal effect makes sense, since the aim of eligibility traces is to relief the temporal-differences problem. This might result in a speed-up in the training phase. The plots show the performance in the testing phase, of agents that are already trained. The influence of this control factor is mentioned again when we discuss the number of episodes in the training phase.

We already discussed the effect of the interaction of column 8. If we look at the effect of the discount factor alone, the mean has a slight preference for $\gamma = 0.98$, whereas the standard deviation has a preference for $\gamma = 0.95$. This makes sense, since the more the agent plans ahead, the better its actions decision (potentially) can be, leading to a higher mean. But the more it plans ahead, the more variances can be expected in the agent's predictions.

However, not all effects are as important as we expected beforehand. We expected more interaction between the learning rate and the number of neurons, i.e. column 3. In general, we expect that the learning rate may increase, if there are less weights (i.e. less neurons). This interaction has an almost negligible effect on the standard deviation. Column 3 has a higher mean at its higher level. That means that one of the following cases is to be preferred: the learning rate is 0.05 and 10 neurons, or 0.10 and 5 neurons. This confirms our belief, but is not significant here.

Optimal settings. Now we derive the (statistically determined) optimal settings. The optimal settings are determined by starting at the most significant effects, and fixing the control factors one by one.

We use two quality characteristics, and may therefore give contradicting results about the settings. One has to decide whether a higher mean is more important, or that a lower standard deviation is more important. Or to find a nice trade-off. Taguchi's Nominal-The-Best (NTB) expresses such a trade-off [24]. However, this equation implicitly gives an equal weight to the squared mean and the variance. This does not necessarily give the trade-off we are after.

Table 5.6 shows the ranking in estimated effects, derived from the half-normal probability plots shown in Appendix C. When the factor in the table is

rank	mean	interaction	std	interaction
1	4		4	
2	12		8	4, 12
3	2		2	
4	5	1, 4	6	2, 4
5	6	2, 4	12	
6	1		13	1, 12
7	8	4, 12	1	
8	11		15	4, 11

Table 5.6: Ranking according to the hnpp of the mean and std (refer to Appendix C).

an interaction, we show the related control factors.

As Table 5.6 shows, both the mean and the standard deviation indicate that factor 4 is the most important factor. The two effect plots agree that fixing factor 4 to its lower level leads to the best result. That is, *SARSA* is to be preferred. The next important factor is column 12 for the mean, and column 8 for the standard deviation. Since column 8 affects column 12, we first fix column 8. Again, both effect plots agree that column 8 should be assigned the higher level. We have fixed columns 8 and 4, and therefore column 12 is fixed to its higher level, i.e. $\gamma = 0.98$. After that, both rankings denote factor 2 as most significant. Fortunately, again, both effect plots want to fix factor 2 to its lower level, i.e. *5 neurons*. The standard deviation ranks factor 6, but since 2 and 4 are already determined, 6 is fixed to its lower level. The mean ranks factor 5. Once again, the effect plots agree that 5 is best at its higher level. Since 4 and 5 are fixed, 1 is fixed to its higher level, i.e. the learning rate $\beta = 0.05$. The last control factor, column 11, is not really important, since this factor and any related interaction occurs low in the ranking. The mean wants factor 11 to be the lower level, whereas the standard deviation shows no influence. Therefore, the use of eligibility traces is disabled, i.e. $\lambda = 0$. This setting is affirmed by factor 15. This factor should be the lower level, for both effect plots. That means that either 4 and 11 both have the lower level, or that 4 and 11 both have the higher level. Since SARSA is chosen and $\lambda = 0$, this wish is complied with.

To conclude, we derived that the combination of $\beta = 0.05$, 5 neurons, SARSA, $\lambda = 0$ and $\gamma = 0.98$ statistically leads to the best trade-off between a high average reward and a low standard deviation. Actually, looking at the effects plot of the mean, every single control factor has its optimal setting. The effects plot of the standard deviation reveals that all settings but factor 12 are also chosen optimal.

The last step in the Taguchi method is to verify if these settings indeed perform best. The optimal settings that are determined, are not part of the experimental set up (recall there are 32 possible trial of which we only perform 16). Therefore, we perform a new trial with these settings.

Note that the derived optimal settings are similar to trial 10, with the exception that $\beta = 0.05$ is used. This makes sense, since we explained there was a slight advantage using this value. Table 5.7 shows the results for the eight runs. Indeed most of the runs show relative high means, but the standard deviation is a bit higher than at trial 10 for example. The settings perform better than most of the trials. However, trial 10 outperforms the 'optimal settings'. The reason that these statistical optimal settings do not yield the highest results, is mainly due that we only used two levels per control factor, and that the values of these levels are very distinct. In the effects plot, we linearly connected both levels, and this is just a simplified assumption.

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
6.85	5.46	0.98	8.43	8.77	8.96	3.82	10.25

Table 5.7: Cumulative rewards for derived optimal settings results.

Actions and switches. The percentage of episodes that reaches a goal state is nearly 100% (refer to Appendix C). Therefore, the average cumulative reward is almost completely determined by the average number of actions and switches.

The average number of actions and switches is shown in Appendix C.2. All agents have an average of at least 2.5 switches per episode. It seems that this is a lowerbound for the number of switches that is necessary. Therefore, punishing a switch even more will not make much difference, i.e. the trade-off between actions and switches will remain the same.

On the other hand, if we lower the punishment on a switch, the trade-off between speed and smoothness changes. The number of actions decreases and the number of switches increases. However, as we already mentioned, the transferability might decrease.

Training time. An interesting question is how many episodes the different trials need in the training phase.

Refer to Figure 5.6. This is the effects plot of the mean of the episodes. Most effects are not significant. The factor that shows the most effect is factor 12, i.e. the discount factor γ. It shows that $\gamma = 0.95$ has a higher average of training episodes than $\gamma = 0.98$. This is an unexpected result, since in general higher discount factors need exponentially more episodes. However, in this case it can be explained. Namely, the average number of actions needed is about 20 (see the results in Appendix C.2). For movements that need more than 20 actions, $\gamma = 0.95$ does not take enough actions into account.

An interesting effect occurs at factor 11. It shows that the use of eligibility traces ($\lambda = 0.3$) leads to slightly more episodes. This might be due to the high exploration rate during the training phase.

5.4.3 All goal angles with 50,000 episodes

One might wonder if some slower learning agents might outperform faster agents after a while, if only they would have enough time. We investigated this matter by continuing with the first three runs from the previous subsection, until all agents reached 50,000 episodes. Taking into consideration that running 20,000 episodes only takes about 10 minutes in simulation time, the answer to the above question is relevant for practical purposes as well.

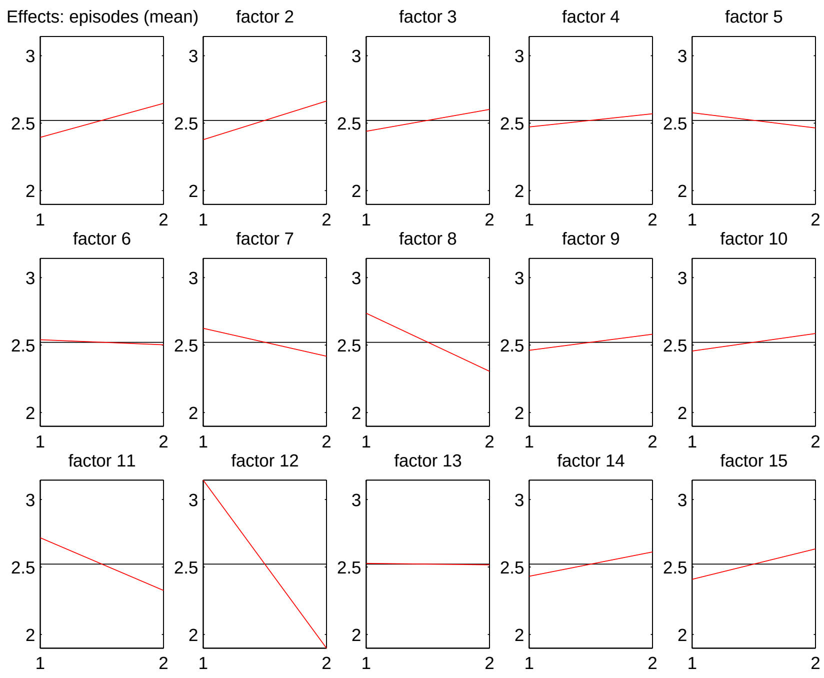

Figure 5.6: Effects plot of the mean of the episodes (the units on the y-axis are expressed in 10,000 episodes).

Agents that ended the training phase rather quickly have an advantage in that they have more episodes at their disposal for optimization purposes. In Table 5.8 we show the cumulative reward after $50,000$ episodes. In the same table, also the differences with respect to Table 5.5 are shown.

Good local optimum. Earlier we concluded that trial 10 has the best results. This trial uses about 17,000 episodes on every run, so effectively 33,000 episodes are used to improve. However, Table 5.8 shows that this trial has barely improved. Compared to the other trials, trial 10 is still performing best in terms of mean and standard deviation. This trial seems to be around a good local optimum. It is not known whether this is also a global optimum.

SARSA vs. Q-learning. In this paragraph we focus on the *differences* of the results after 50,000 episodes, compared with the previous results of Table 5.5. We give the effect plots of the differences and discuss the important factors.

Table 5.9 shows the ranking in estimated effects, derived from the half-normal probability plots shown in Appendix C. The increments and decrements in the mean are very small, at most 1.5 reward units. However, the standard deviation shows some more important changes. Again, factor 4 is the most significant one with an advantage for SARSA.

It is striking to see that the standard deviation for Q-learning is rather big. If we look at Table 5.8, we see that SARSA indeed performs quite stable. The

trial	run 1	Δ run 1	run 2	Δ run 2	run 3	Δ run 3
1	8.46	3.72	9.00	-0.34	7.08	6.53
2	10.30	7.40	9.08	1.27	5.61	6.96
3	-1.59	-11.83	11.53	1.43	12.21	12.35
4	10.62	10.45	11.01	6.95	6.97	4.12
5	10.03	1.77	8.56	3.97	3.65	3.66
6	12.64	0.34	11.80	2.62	9.57	0.72
7	12.90	4.90	13.24	2.58	8.75	15.03
8	1.23	0.13	4.69	12.87	10.66	1.00
9	7.09	8.62	6.50	-0.08	8.69	6.02
10	12.24	1.24	10.96	1.14	12.30	0.20
11	7.70	5.61	10.27	1.05	8.02	6.79
12	9.16	0.52	7.78	5.39	6.24	8.35
13	11.05	2.47	11.63	1.03	10.26	3.45
14	10.22	4.56	7.63	-0.75	9.51	4.25
15	8.29	10.30	-38.07	-1.37	9.27	2.44
16	7.86	8.04	-22.12	-13.40	7.70	8.13

Table 5.8: Average cumulative reward after 50,000 episodes.

rank	mean	interaction	std	interaction
1	11		4	
2	14	2, 12	10	1, 11
3	7	11, 12	13	1, 12
4	10	1, 11	3	1, 2
5	4		6	2, 4
6	9	2, 11	9	2, 11
7	1		15	4, 11

Table 5.9: Ranking according to the hnpp of the mean and std after 50,000 episodes.

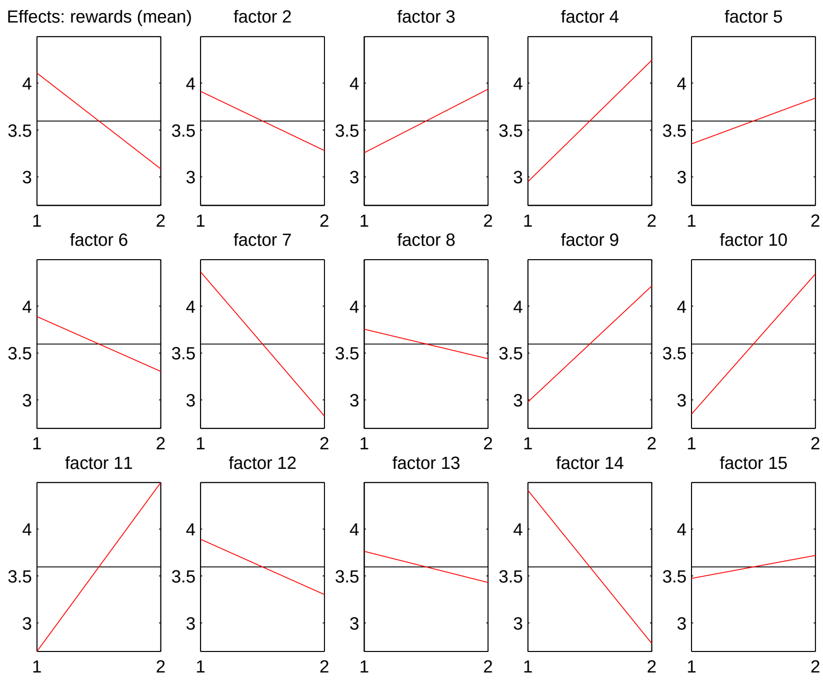

Figure 5.7: Effects plot of increments and decrements in mean until 50,000 training episodes are used.

results are improved in almost all trials, but not with many units. Q-learning improved in some trials a lot (trial 7 from run 3 even 15 units). However, in some other trials Q-learning decremented the result significantly (trial 16 from run 2 decreased 13 units). The learning process of both SARSA and Q-learning is stochastically. There are ups and downs in the performance, and if we would have measured 10,000 episodes later the results are likely to be different. However, the results of Q-learning seem to fluctuate significantly more than SARSA's.

Note that this decrement of results is *not* the same as the divergence that we mentioned in Section 4.2.6. The possibility of divergence refers to NN weights that go to infinity.

5.4.4 No shaping

In this subsection we investigate the merits of applying the shaping strategy. We fix the exploration rate $\epsilon = 0.09$ and the temperature $T = 10$. The results are shown in Appendix C.5. Here we only discuss the conclusions.

In all trials, the agent solves the problem. This result is remarkable, because in prior experiments we found that the agent was not able to reach the complete deviation at all. The explanation is straightforward. The robot's construction has been changed, but more important is the replacement of the weary potmeter. The simulation is based on transitions on the robot arm, when the potmeter was still new and not weary. The result that shaping is not necessary for reaching the goal states is a great plus for the learning method. It shows

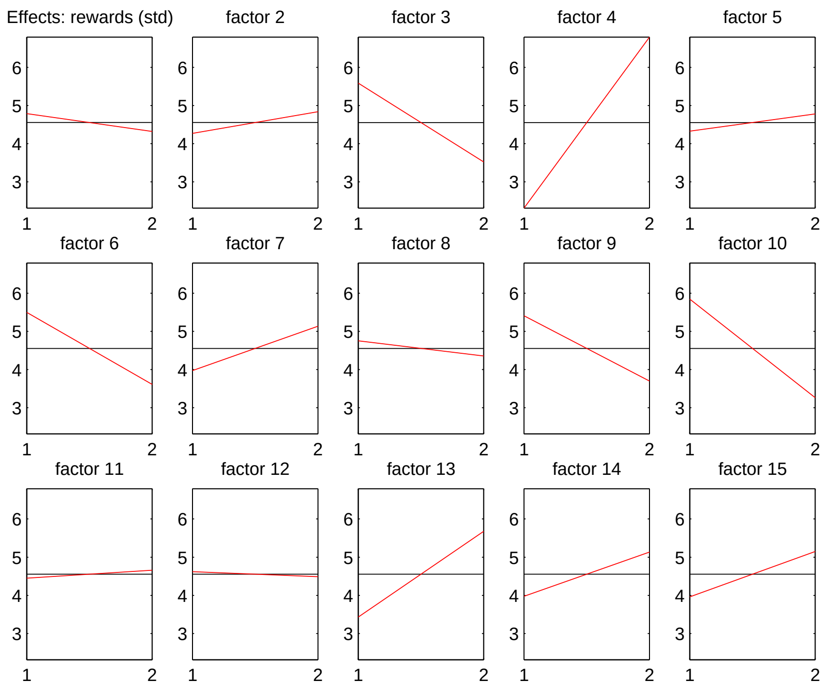

Figure 5.8: Effects plot of increments and decrements in std until 50,000 training episodes are used.

that the goal states can be reached without biasing the agent by guiding it.

The results show that fewer episodes are needed than with shaping. This can be explained from the fact that the shaping strategy only allows to adapt the deviation every 500 episodes. This number is determined empirically, based on experience with other simulations (with a weary potmeter and a worse construction). Setting this number lower would imply that fewer episodes are needed. We did not experiment with this parameter thoroughly, since the amount of training episodes is not a bottleneck: at most it might save a few minutes in simulation time.

However, shaping does indeed improve the solution. The average cumulative reward without shaping is invariably low, and shows a high standard deviation. The use of shaping thus makes the probability of efficient solutions with a smaller standard deviation higher. Moreover, a high exploration rate is desirable initially, to let the agent explore the environment. Setting the exploration rate and temperature high in the beginning, and decrease them gradually is also a form of shaping.

We conclude that shaping improves the agent's results, but that (fortunately) it is not indispensable to obtain successful episodes. It remains to be seen how transferable the results without shaping are. The use of many switches means the movement is not smooth, and is an indication that the solution is not efficient.

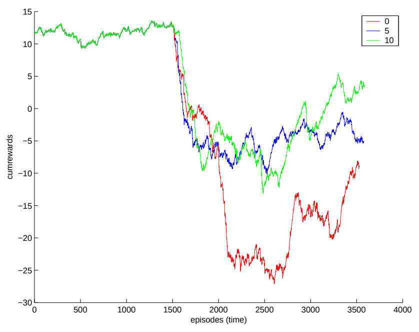

Figure 5.9: Cumulative reward on the robot arm.

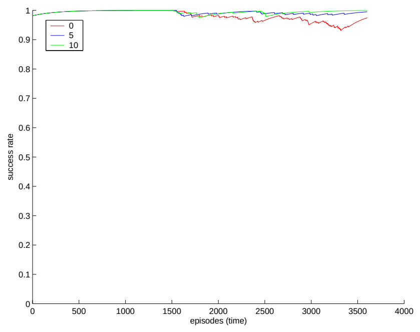

Figure 5.10: Success rate on the robot arm.

5.5 Transfer experiments

In this section the transfer problem is investigated. In Chapter 3 the *transfer problem* is defined as all issues related to the gap between simulation and real system.

Experiments on the real robot arm take much more time than in simulation. A good controller can complete about 10 episodes per minute on the robot arm. This includes the deflation of the muscles, and inflation of the muscles to go to a new random state. However, during the training phase many episodes are unsuccessful, and at least 15,000 episodes are necessary. The overhead of deflating the muscles (two seconds), inflating again (uniformly chosen with a maximum of 1.5 second, so 0.75 seconds on average) and waiting to damp oscillations (one second) together takes about 3.75 seconds. An unsuccessful episode consists of 100 steps at 30 Hz, i.e. 3.3 seconds. This would require $15000 \cdot 7.05 \approx 106,000$ seconds, i.e. approximately 29 hours of continuous training. This approximation should be considered an estimation of the order of magnitude: not all episodes are unsuccessful and $20,000$ episodes is not always enough. Based on this estimation, we conclude that training on the real robot arm would be an option in practice if no simulation is available (in simulation it takes about ten minutes).

An exhaustive benchmarking experiment like we did in Section 5.4.2 is impossible in view of time. We choose the best performing agent in simulation, and investigate the transfer problems. For the experiments, we take the trained agent of trial 10 from run 3. Note that we take the agent after it first ended the training phase and not the agent that trained 50,000 episodes.

5.5.1 Transfer results

Reality gap. Figure 5.9 shows the cumulative reward for three agents on the robot arm. The agents differ in that they use different learning rates: 0 (red line), 0.05 (blue line) and 0.10 (green line). To allow a comparison, the first $1,500$ episodes belong to the testing phase in simulation. In this paragraph we discuss the performance drop just after simulation. The differences between the three agents after the initial period are discussed in the next paragraph.

Figure 5.10 shows that the episodes remain successful. However, in all three cases the cumulative reward drops more than 15 units after transfer. This

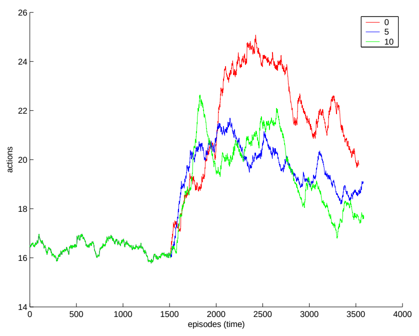

Figure 5.11: Average number of actions on the robot arm (successful episodes).

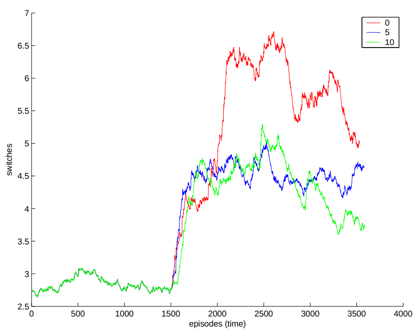

Figure 5.12: Average number of switches on the robot arm (successful episodes).

suggests that this is due to the gap between transfer and simulation. However, this is only partly true, as we explain in the next paragraphs. Figure 5.11 shows the average number of actions and Figure 5.12 plots the average number of switches per episode. Both figures do only take the successful episodes into account. This is done because an unsuccessful episode would imply a very big punishment. Moreover, this punishment is dependent on the number of actions that is allowed before an episode is cut-off. As can be seen in Figure 5.10, the agents with learning rates 0.05 and 0.10 have a very small number of unsuccessful episodes.

The number of switches rises with 2. For the agent with a learning rate of 0, the number of switches even rises with 4. The number of actions rises from 16 to about 20 actions, i.e. with about 25%. However, 20 actions is still only $\frac{2}{3}$ second. This is not too bad. To make conclusions about the transferability in general, we study how the behavior of the different agents evolves after transfer.

Learning rate. In this paragraph we discuss the differences in performance between the different learning rates.

In all three cases, the number of actions and switches rises slightly in the beginning. However, all three agents improve after some time. An average movement takes 20 steps (i.e. 0.66 seconds) plus the overhead of 3.75 seconds. It takes about 750 episodes before real progress is observed, so effectively about an hour. It is remarkable that even the agent with learning rate 0 improves. In the next paragraph we explain how this is possible.

We did no experiments overnight. The reason is that we did not want to let the system work many hours unsupervised (with respect to possible damages). Therefore, we did not investigate how far the agent optimizes its performance. We expect that it will not be significantly better than in simulation. The agent cannot derive significantly more information from the state representation than the simulation's function approximator. Therefore, we do not know what an optimal controller can achieve.

Behavior of the muscles. In Chapter 2 we discussed the difficulties of controlling the robot arm. The muscles do not only expose a highly non-linear behavior, but the hysteresis properties of the muscles cause troubles as well.

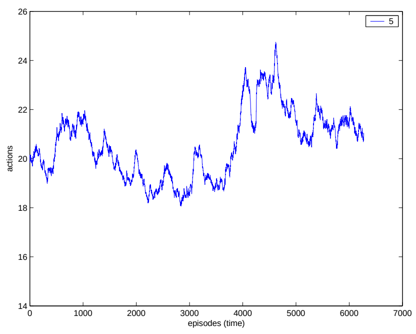

Figure 5.13: Average number of actions, with a hitch at 4000 episodes.

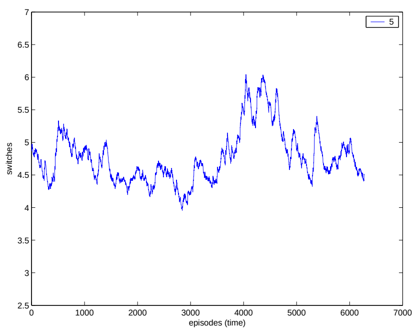

Figure 5.14: Average number of switches, with a hitch at 4000 episodes.

Refer to Figure 5.13 and Figure 5.14. These figures shows the average number of actions and switches on the robot arm, for an agent with a learning rate of 0.05. After 4000 episodes, the performance suddenly drops. The reason is that the robot arm has not been used for 14 hours. The system's behavior has changed overnight. The different behavior can be due to temperature differences, but also because of the hysteresis property. Similar effects occur when a weight is placed on the arm for a few hours. The arm stays in the same position after the weight is removed. This kind of behavior stimulates the call for adaptive algorithms. This behavior is one of the reasons of the performance drop right after simulation, since the simulation is not trained on data from a 'cold' robot arm.

After some 2000 episodes (about 3 hours) of learning, the controller is again at its old level of performance. An interesting question is now, whether the performance grows because the robot arm is changing back to its previous behavior, or because the agent adapts itself to the arm. Or a combination of both.

To answer this question, Figure 5.9 serves again. The three agents start with the same Q values. The experiments were started in the morning, after a night of rest for the arm. The figure reveals that even the agent without learning improves after a while. This is because the robot arm changes back to the behavior it had been trained on (in simulation). However, the other two agents improve faster and better. We conclude that the learning agents indeed adapt to the system.

5.5.2 General performance

In this subsection we discuss the performance in order to make a comparison to other controllers possible.

Angular displacement. The eventual goal is a fast controller. A way to benchmark the RL agent's performance with another controller, is to compare the time that is needed to complete point-to-point movements. The time depends on the angular distance between the two points, but also on the current pressures of the muscles.

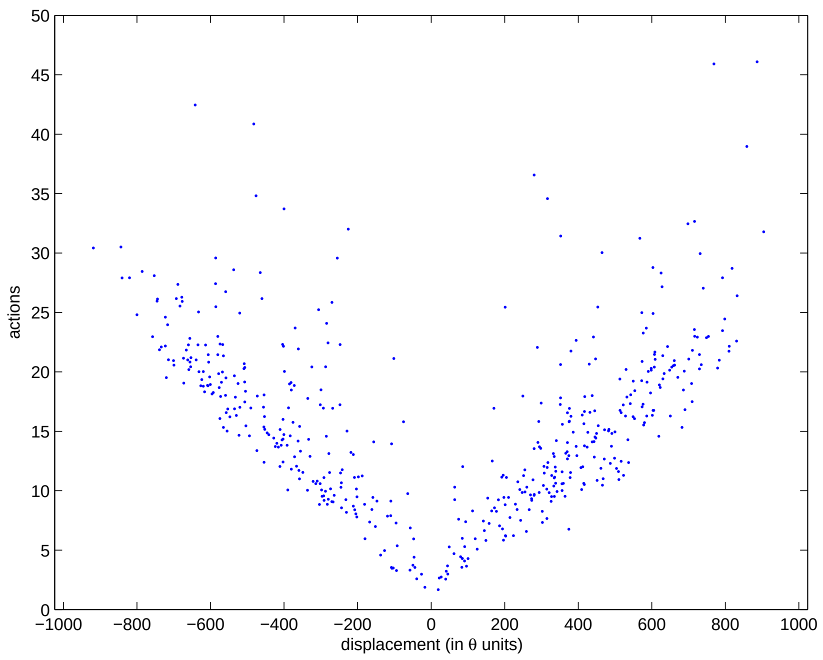

Figure 5.15: Number of actions plotted against angular displacement (in θ units).

Figure 5.15 plots the number of actions against the angular displacement. Every point is associated with an episode of the agent with learning rate 0.10. These episodes are taken from the phase where the agent has adapted to the robot arm, i.e. episodes 2500 to 3500 from Figure 5.11.

As can be seen, there is a clear relation between the angular displacement and the number of actions that is used. This is exactly the assumption of the shaping strategy, discussed in Section 4.3. The spread comes from the fact that the muscles have different pressures at the beginning of an episode. Also, the arm is a bit stochastic.

A similar plot is made for the number of switches, shown in Figure 5.16. As can be seen, the relation between the number of switches and the angular displacement is less significant. The figures shows that only a few episodes use six or seven switches, the rest uses fewer. Note that some episodes use zero switches (i.e. only one type of action is executed).

The figures together show an interesting observation. In many cases, the number of switches for small displacements is about the same as for big displacements. One might think that there is a minimal number of switches needed for every movement. Because an ideal movement always follows the following steps: accelerate, move at maximum speed, de-accelerate.

However, there is another possibility. The figures reveal that the switch-to-action ratio is bigger for smaller displacements. This implies that short movements are less smooth. This is indeed what we observed empirically. The controller seems to have bigger trouble with small displacements than with

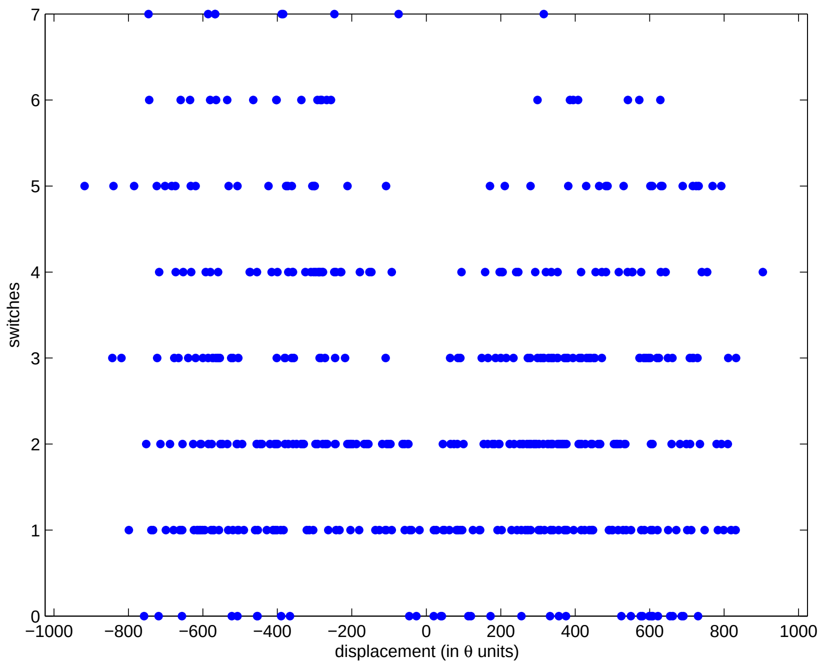

Figure 5.16: Number of switches plotted against angular displacement (in θ units).

bigger movements. The effect of a single action may be too big, and thus has to be compensated for. For example, if air is let out of a fully inflated muscle, the force of the muscle changes significantly.
In the next paragraph we give examples of typical movements.

Typical movements. In this paragraph we show and discuss a few typical movements. The plots that we show are screenshots of a scope. The scope monitors the output signal of the potmeter. The unit on the x-axis is 200 ms. The unit on the y-axis is 1 V. As we explained in Section 2.3.3, 1 V equals 200 θ units, i.e. approximately 20°.

Figure 5.17 shows a displacement of 30°. Unfortunately, we did not plot the valve signals to see when the control stops. We estimate that the first damping amplitude after the goal is reached is about 5°. This is quite much. The reason is that the pressures in the muscles was very low. So, after the goal state is reached, the compliancy of the muscles is the cause that the damping amplitude is significant. The only way to overcome this, is to make the muscles more stiff, i.e. increase the pressure. Another solution is to use tracking control, as we discuss in the next section.

A typical behavior is obtained when the desired displacement is very small. Figure 5.18 shows a movement where the desired angle is about 5° lower than

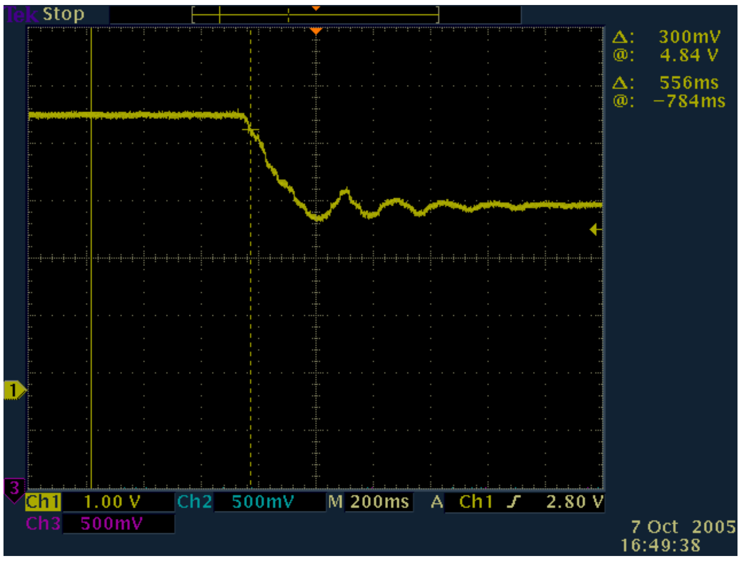

Figure 5.17: Displacement of 30° with low pressure in muscles.

the current angle. As can be seen, the angle decreases about 20° before it is brought back to its desired level. The reason is that the controller puts more air into both muscles. During the change of air pressure, the forces of both muscles change temporarily. This causes the big displacement that is made. This behavior is due to the overactuation of the joint, i.e. that two muscles are used for only one degree of freedom.

Lastly, Figure 5.19 shows a typical movement of 70°. As the figure shows, the movement is smooth, and the controller slows the arm down before the goal is reached. Note that the line itself shows irregularities. These are caused by noise and by the weary potmeter.

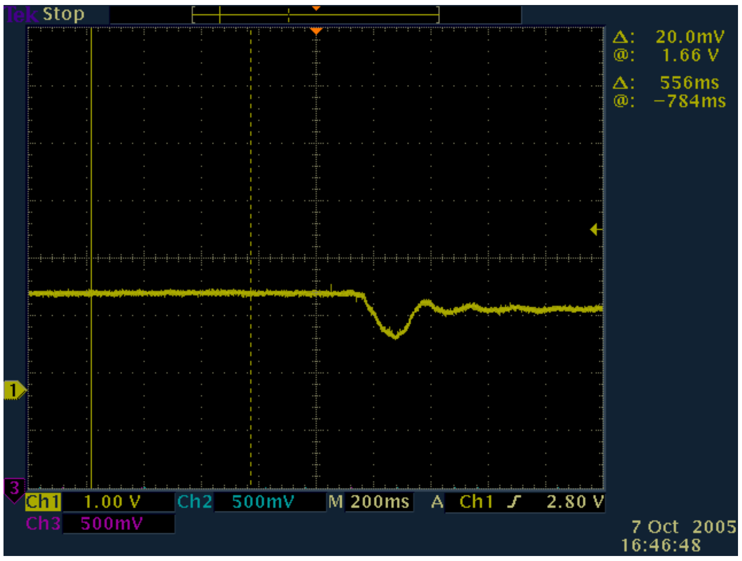

Figure 5.18: Displacement of 5° with pressure adaption.

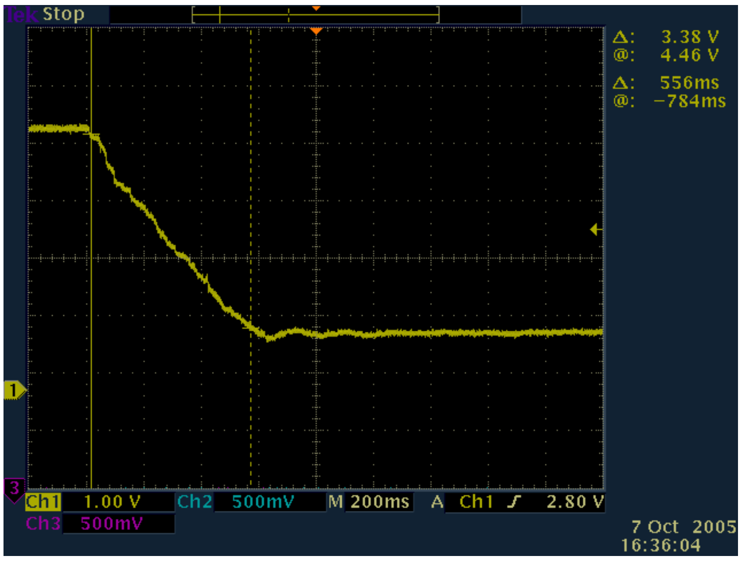

Figure 5.19: Displacement of 70°.

5.6 Discussion and conclusion

In this section we evaluate the results from the experiments. We also reflect to the research question about the merits of AI.

The basic idea was that another student would tackle the problem in the classical way. The vacancy has not been filled in time to allow a comparison in this report. However, in general we can say that the results of the experiments are satisfying. On the average, the control is fast and smooth. Moreover, the agent adapts to the changing behavior of the system. We conclude that the transfer to the robot arm is good. Moreover, the agent may be able to learn to control the robot arm without a simulation. We estimated that this would take about 30 hours.

In the next subsections we discuss the results of the experiments.

5.6.1 Practical application

In this subsection we discuss how suitable RL is for practical applications. The issue about scalability is discussed in Chapter 6.

For a practical application, the use of a *tracking controller* (refer to Section 4.4) is more useable than the definition of goal states. We already discussed how this can be implemented and that the problem of estimating ω becomes obsolete. We belief that RL is able to learn a tracking controller.

A more fundamental problem with respect to practical application, is that no performance guarantees can be given (refer to Section 4.4). Sometimes the agent makes (seemingly) unnecessary switches and actions. As a minimum requirement, there should be a guarantee that the agent always reaches the desired goal angle.

The current solution has a success rate of almost 100%. Empirically we found that almost always a goal state can be reached, even after an unsuccessful episode. The trick is to impose a delay of about two seconds, and start a new episode in that state. The two seconds decrease the rotational velocity. However, in order to use the controller on a real system, this ‘guarantee’ will not be sufficient.

We discussed that the use of Lyapunov function (refer to 4.4) might give such guarantees. However, the design of such a function is not straightforward and it implies that system knowledge is used.

The assumption that the state should have the Markov-property is a drawback. An RL agent that uses feedforward neural networks is not able to detect hidden states. In practice the system never has the Markov property completely. This need not be a problem, since we have shown that RL works well. However, we believe it can be greatly beneficial to investigate the use of recurrent neural networks, like Enforced Sub-Populations uses (refer to Section 4.5).

A last drawback is that knowledge about RL and NNs (or any other function approximator) is necessary for a successful application. The problem of designing a controller in the classical way is translated to the problem of designing an

RL agent.

5.6.2 Merits of RL

In this subsection we discuss the merits of RL with respect to classical control theory.

One of the most important advantages is that the agent *adapts* to the system. Because of temperature changes and the hysteresis properties of the muscles, the arm behaves very differently after some hours of operation. We showed that the agent adapts to the system. In the current solution, it took about one hour before real progress was observed. However, we expect that the adaptation can be made faster. For example, by re-using experiences and a higher learning rate. Further experiments have to be done in order to investigate how well the system adapts.

The learning procedure with the use of the empirical simulation is extremely fast. The database has enough transitions after two hours. In about twenty minutes, a simulation is made and a controller has been trained in simulation. Without simulation, we estimate it takes about thirty hours to learn on the real system (refer to Section 5.5).

Another very important advantage with respect to classical control theory, is that the controller does not make *linear assumptions*. McKibben actuators are notorious for their non-linear behaviour (refer to Section 2). We showed that RL is able to cope well with these muscles.

Classical control theory assumes a continuous output signal is possible. A continuous air flow can be simulated using Pulse Width Modulation (PWM) [20] (refer to Section 2). The solution of RL is naturally suited for discrete actions.

Almost no system knowledge is used. The only system knowledge that is necessary is to derive the state variables. However, some knowledge remains necessary in order to reveal and resolve system errors (refer to Appendix A).

A last interesting advantage is that the algorithm is generally applicable. Every McKibben actuator has a distinct behavior. If a muscle is replaced, the whole procedure remains exactly the same: fill the database, make a new simulation and learn a new controller. If a model would have to be designed, the new muscle needs to be tested and parameters need to be fit in order to predict the behavior of this muscle. During the project, the robot arm has had some major changes in construction, but the procedure was left unchanged.

Chapter 6

Scalability

In this chapter the scalability issues are discussed. Section one discusses the influence of the second joint on all steps in the learning process. Section two discusses a hierarchial approach to tackle the problem. We end with conclusions in section three.

6.1 Second joint

In Chapter 2 we discussed the construction of the robot arm. Refer to Figure 2.2. We designed the construction in such a way that maximal interaction between both joints occurs. That means that the dynamics of one joint heavily influences the position of the other. Note that one normally tries to make every Degree of Freedom (DoF) as independent as possible.

In this chapter, we denote the angle of the first joint with θ_1 and the angle of the second joint with θ_2. The stroke of the second joint is $90°$ (like the first joint). This means that the inverse kinematics problem does not play a role, since every position of the arm has a unique combination of angles (θ_1, θ_2).

The inclusion of an additional joint makes the problem more complex. For example, the state space is doubled in its dimensions. Furthermore, with one DoF it is not possible to 'miss' an angle if the whole stroke is used. With two DoF, the goal angles lie on a surface, and the goals are more easily missed.

6.1.1 Simulation

In this subsection we discuss how the simulation can be extended with the second joint.

The collection procedure can be applied individually to each joint. However, since the state space is doubled, the worst case scenario would be that the number of transitions needs to be squared.

The filtering procedure can be extended in a similar way. The $5D$ grid from Section 3.2.3 is replaced by a $10D$ grid. Because of the interaction effects between the joints, states that were previously marked as 'unsound' might occur in the transition set.

The function fit remains the same. The only change is that the state contains more inputs, and that six NNs are used instead of three. The NNs have no problem in training on a very big transition set, with many input dimensions. Only the training algorithm should be reconsidered, since the Levenberg-Marquardt algorithm has high memory requirements.

We conclude that the transition acquisition procedure is the major bottleneck. If another way is found to obtain a representative transition set, the empirical simulation may indeed be scalable.

6.1.2 Controller

The use of only one RL agent is not practically possible for two joints, since there are 81 combined actions. This number is far too high to obtain a Q^π estimation for each individual action. An exploration rate of 0.50 would imply that every action is explored with $\frac{0.50}{81} \approx 0.6\%$ exploration probability per action.

Another option is to use one RL agent per joint. The idea is that the controllers together converge. An assumption of RL is that the state should have the Markov property. Therefore, all sensor values (at multiple timesteps) should be included in the state of both agents. However, since the action selection procedure of the other agent is not known, the state is not Markov anymore.

Goal states. If two controllers are used, there are two options with respect to the goal states. Theoretically, the best results are possible if the controllers must reach a common goal together. That can be an (x, y) coordinate in the Euclidean space, or two goal angles (θ_1, θ_2). That means that the two joints obtain the same rewards. If one controller reaches the right angle, but the other one not, the goal is not met and both get punished. Another option is to let the controllers focus on their own goal angles. That means every controller has its own reward function. In our experiments (discussed below) we let the controllers reach a common goal.

6.1.3 Shaping strategy

In this subsection we discuss the changes in the shaping strategy.

The shaping strategy is extended in a straightforward way. Every joint gets an individual allowed deviation. However, a difficulty is imposed by the effect of exploration steps.

Exploration. The fact that the controller does not know the next action of the other one, makes converging together difficult. This becomes even more pronounced with respect to exploration steps. Especially in the beginning, when many exploration steps are performed, this makes the problem harder.

Call ϵ_i the probability of an exploration action for controller i. The probability that at least one of the two controllers performs an exploration action is $\epsilon_1\epsilon_2 + \epsilon_1(1-\epsilon_2) + (1-\epsilon_1)\epsilon_2$. If $\epsilon = \epsilon_1 = \epsilon_2$, then the probability is $2\epsilon - \epsilon^2$.

That means, that if we apply the same shaping strategy with an initial exploration rate of 0.5, the probability of at least one exploration action is 0.75. This is a very high exploration rate. A way to overcome these problems, is to lower the exploration rate, or let the criterium for the performance be less demanding.

6.1.4 Preliminary experience

In this paragraph we discuss the preliminary experiences we have with respect to including the second joint.

For one joint we used $200,000$ transitions in the database. This number might decrease, since we did not thoroughly determine the minimum size of the database. For two joints, about the double (i.e. $400,000$) proved to be enough.

We decreased the grid size of the filtering procedure by changing the filtering to $(|\theta_1|, |\omega_1|, |\rho_{1,1}|, |\rho_{1,2}|, |\alpha_1|, |\theta_2|, |\omega_2|, |\rho_{2,1}|, |\rho_{2,2}|, |\alpha_2|) = 5 \times 5 \times 5 \times 5 \times 9 \times 5 \times 5 \times 5 \times 5 \times 9$ grid elements. It is important to keep a distinction to all 9 actions per joint.

The eventual simulation is a bit worse than the simulation for one single joint, i.e. the standard deviations are a bit bigger. However, preliminary experiments showed that it indeed can be used to train a controller. About 100,000 episodes were used to learn a single goal (θ_1, θ_2). We believe that this is proof that all goal states can be learned. Because if RL can solve a single goal (i.e. one Q function), it can (in principle) solve any other goal as well. The question whether all these goals can be learned simultaneously is dependent on the function approximator.

6.2 Hierarchical learning

The use of two controller that should co-operate together, without knowledge of each other, triggers the idea of introducing a higher level agent. This is called *hierarchical reinforcement learning.*

A successful example of hierarchical reinforcement learning is explained in [15]. Their problem is the acquisition of stand-up behavior of a real robot. The problem is split in two levels. The action of the higher level is to give the desired position. This position is a subgoal for the lower level controllers, that control the actual actuators. The lower level agents get an additional reward when a subgoal is reached. One of their conclusions is that the high level planner should get a very low-dimensional state representation. The lower levels get the full state information. The use of subgoals was helpful for efficient exploration. They obtained faster learning and more robust results than a plain RL architecture.

We might try to use the same idea, by introducing a higher level that plans the general strategy. For example, that the joints should head for a certain angle without specifying the pressures that are needed. The lower levels controllers take care of the valve control. The lower level agents get a reward when their subgoal is reached, and an additional reward when the overall goal is reached.

The use of a hierarchial approach is a way of exploiting system knowledge. The problem is split up in levels, where the highest level expresses the general idea. Every lower level solves a less abstract problem until the actual actuators are controlled.

6.3 Conclusion

In this chapter we discussed that the inclusion of a second joint leads to a more complex problem. If two agents are used, they have to co-operate. The state representation is doubled and is not Markov anymore.

We have shown that the acquisition of transitions is the major bottleneck to scale the simulation. With respect to Reinforcement Learning, the use of a hierarchical approach might be better suited.

Chapter 7

Conclusions and Recommendations

In this final chapter we give an overview of conclusions and recommendations. In section one, the contributions of this research project are summarized. Section two gives conclusions and reflects to the central research question. Finally, section three gives recommendations for future work.

7.1 Contributions

In this section we discuss our contributions.

First of all, with the help of many Philips engineers we built a robot arm that may serve for future experiments with control algorithms. The chip software is adapted to the robot and ready to use. Also, we wrote C code to control the robot arm. All major errors in chip, sensors and construction have been identified and resolved.

Second, we developed and implemented a procedure to learn a controller in a time span of a few hours. We proposed a way to make a simulation that is purely based on empirical data. We developed tools to analyze the quality of the simulation. The analysis shows if the state definition contains enough information to make a good prediction possible. If this is not possible, it is strong evidence that the state representation is too poor for the RL agent to reliably base its action decisions on. Furthermore, we investigated how to tackle the transfer problem to the robot arm.

Finally, we applied Reinforcement Learning on a real robot arm in order to investigate its potential. We performed experiments to investigate the influence of the parameters. We discussed the influence of the parameters and showed that the initial weights of the neural networks are very important to the eventual performance.

7.2 Conclusions

The central research question is as follows:

> Investigate the merits of some Artificial Intelligence methods in their ability to control a difficult control system.

Since we mainly investigated Reinforcement Learning, the following conclusions hold for that approach.

Compared to classical control theory, the merits of Reinforcement Learning are as follows. First, the method adapts to the system. Next, the algorithm is able to cope with non-linear behavior: no linear assumptions are made. In classical control theory, the difference between linear and non-linear problems is very important. Third, the method is naturally suited for discrete actions. No additional efforts are needed to simulate a continuous control signal. The fourth merit is that the learning procedure is fast. We showed that it takes only a few hours to learn a controlling strategy. Fifth, almost no system knowledge is necessary. Only some basic knowledge is used in order to derive the state variables, and to reveal and resolve system errors. Another merit is that no model is necessary. We showed that it will approximately take 30 hours to learn on the robot arm. A last advantage is that the approach is generally applicable. When the construction is changed, or a muscle is replaced, the procedure remains the same.

The major drawback is that no performance guarantees can be given. We cannot even be sure that a goal state is reached at all. We discussed that Lyapunov theory might improve this matter. Second, the assumption that the state should have the Markov property restricts the application, since every influence of the system should be included in the state representation. A last drawback is that knowledge about RL and NNs (or any other function approximator) is necessary for a successful application. The problem of designing a controller in the classical way is translated to the problem of designing an RL agent.

We conclude that RL has interesting features that justify further research about the subject. We expect that classical control theory will still outperform the current controller. However, one should keep in mind that the field of control theory is a mature area, whereas the field of Artificial Intelligence is still at an early stage with respect to practical applications.

7.3 Recommendations

In this section we give recommendations for future work.

First, more research is necessary to be able to give performance guarantees. As an example we discussed Lyapunov theory. Next, the use of a controller that continuously tracks the robot arm is more useful than the use of goal states. It also allows for a better comparison with classical control theory. Second, we recommend to investigate Enforced Sub-Populations (EPS). ESP is a novel approach and might outperform RL on difficult control problems. ESP might be more suitable for multi-agents tasks (for two joints for example) and might perform better on non-Markov problems. Source code of ESP is readily available [10]. We adapted the source code of ESP to work with our simulation, but have not performed elaborate experiments.

Third, with respect to the scalability of the procedure, we advise to look into a hierarchical RL approach. For the simulation, a more efficient way of gathering relevant transitions (with sound states) should be found. Fourth, to improve the state representation and the estimation of ω, we advise to read the sensors at a higher frequency than the system is controlled. This can be done during the idle time of the chip. Alternatively, velocity or acceleration sensors could be used.

Fifth, in order to speed up the learning of the RL agent, the use of an internal world model or the application of an inverse simulation might be considered. Finally, some interesting issues are related to the merging of classical control theory and artificial intelligence. If a way would be found to translate RL solutions to its classical equivalent and vice versa, this could be beneficial for both fields. Also, it might be wise to drop the assumption that no system knowledge may be used. This might improve the learning process significantly.

Chapter 8

List of notation

General:	
k	timestamp
Δt	time between two timestamps
σ_k	state at timestep k
θ_k	angle of joint 1 at timestep k
$\rho_{1,k}$	pressure in muscle 1 of joint 1 at timestep k
$\rho_{2,k}$	pressure in muscle 2 of joint 1 at timestep k
ω_k	rotational velocity of joint 1 at timestep k
$\dot{\omega}_k$	rotational acceleration of joint 1 at timestep k
o_k	$(\theta_k, \omega_k, \rho_{1,k}, \rho_{2,k})$
θ_g	goal angle
σ_g	goal state according to Equation 2.4
α_k	one of the nine possible actions, executed at timestep k
RL:	
β	learning rate of neural networks
γ	discount factor
λ	decay rate for eligibility traces
r_k	reward obtained at timestep k
R_k	discounted rewards starting from timestep k
π	policy that maps a state σ to an action α
$Q^\pi(\sigma, \alpha)$	action-value function
$V^\pi(\sigma)$	state-value function

Table 8.1: List of notation.

Appendix A

System errors

An essential part of the project had been to identify and resolve system errors. If any one of these errors had not been discovered, the method could not have been applied successful. Therefore, we show some important errors. We discuss how we identified them and our solution.

In Section 3.3 four items are given that might be the cause that the data does not contain enough information. We repeat the item here for convenience:

- An error in the sensors: too noisy or a systematic error.
- A state variable is lacking.
- The sensors and state variables are right, but information from previous timesteps is necessary (history).
- The issue about opening or closing the valves, as explained in Section 2.4.3.

Indeed we found errors in our sensor data, using residue plots. These errors did not reveal themselves easily straight from the database. In the next section we discuss some key errors that our analysis revealed and how we solved them. However, even with the corrected sensors the isoplot remained much the same. Apparently, the function approximators did a good job in predicting the sensor values, including their errors. This is proof that the isoplots alone are not sufficient and that isoplots and residues are complementary.

Next, another cause might be that a state variable is lacking. This cause is the most tricky one to solve, since it requires more system knowledge and additional hardware (sensors). In our analysis on the state variables (Section 2.3.2) we already derived the main components of the system. A temperature sensor might improve the results. Furthermore, an explicit velocity sensor and an acceleration sensor might improve the results even further. However, we make no changes here and try to stick to the sensors we have at our disposal.

The third cause suggests that we need to include more timesteps. Indeed, including the observations o_{t-2}, o_{t-1} and o_t and the observed actions α_{t-1} and α_{t-1} that were taken in these timesteps improved the simulation results a great deal. Including an additional timestep, o_{t-3} did not significantly improve the results even further. We also experimented with the inclusion of the time difference between timesteps. However, since both the reading of sensors and the calculation time is very constant it did not have a significant influence.

To conclude, we define our state σ_t as follows:

$$\sigma_t = (o_{t-2}, \alpha_{t-2}, o_{t-1}, \alpha_{t-1}, o_t) \tag{A.1}$$

The inclusion of multiple timesteps is due to the delay in the system. For example, pressure drops and pressure rises are not captured in only the most recent sensor values.

Finally we discuss the valve issue. After the previous items were resolved, the extrapolation of the sensor values turned out to be possible. Therefore, the valves remain open during the calculation and delay.

A.1 Sensors

A.1.1 Pressure sensors

Connection of sensors. At one time, the isoplots and residues of ρ_1 and ρ_2 were very poor, whereas the results for θ were fine. A look in the database learned that enormous leaps in the pressures occurred, in a time interval of about 30 ms. The reason could be either due to the sensors, or by unwanted phenomena with the air flow. The use of a oscilloscope clarified matters. After an inlet of air, the pressure immediately increased rapidly. See Figure A.1. Only after 10 ms after the closure of the valve, the pressure dropped to a stabilized value, due to the stabilization of the air flow inside the muscle. A possible solution would be to measure the air flow at the other side of the muscle, to prevent from pressure drop and pressure build close to the valve. Fortunately, this proved to be easy with the muscles: the closings of the muscles are merely screws. One closing has an inlet. By using an inlet of a spare muscle we create an additional air inlet. Figure A.2 shows the new measurement.

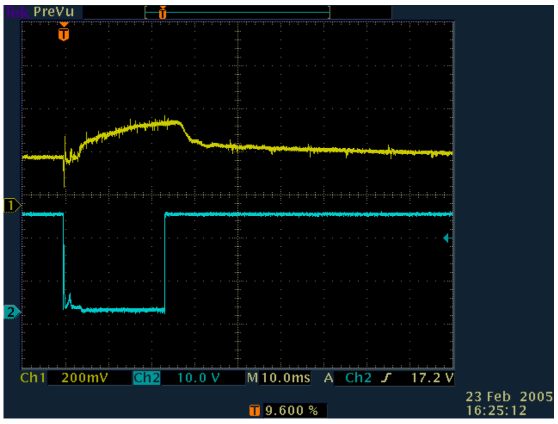

Figure A.1: Old pressure measurement.

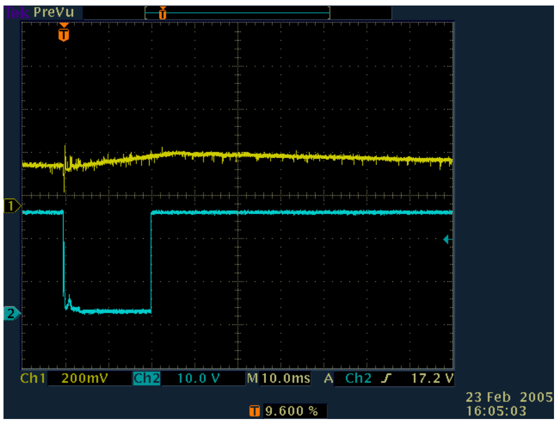

Figure A.2: New pressure measurement.

Problems with pressure. Another problem with the pressure was that the pressure from the source was not stable. The pressure fluctuates between 5 and 6 bar, depending on other consumers in the building.

To maintain a constant pressure source, we added a buffer. The content of the buffer is 10 liter. The reducer is placed between the source pressure and the buffer.

However, the reducer has a small leak. A very small air flow goes to the buffer. If no air is consumed (i.e. the valves do not led air out,), and if the leak in the pipes and muscles is not bigger than this air flow, a pressure build up occurs. This may result in a pressure of 6 bar in the buffer. We experienced this when we started again: 6 bar was put in the muscle, where only 3 bar is allowed. Fortunately, the muscle only deformed, but did not burst. For safety purposes, we therefore installed a safety valve on the buffer, that leads air out if the pressure exceeds 2 bar.

A.1.2 Potmeter

The first potmeter we received turned out to be a logarithmic potmeter. The next potmeter showed increasingly bad isoplots of θ, and as a result, in ω. The potmeter heavily suffers from wear. This is a result of the installation of the potmeter. Its axis is fixed on the axis of the joint, making its movement overdetermined. The potmeter's axis is therefore continuously pressed on.

A.1.3 Noise

Figure A.2 shows that the noise of the output signal is relatively high. The units of the y-axis are 200 mV. Looking at the figure, the tops and downs of the noise can be as much as $\frac{1}{5}$ of a unit, i.e. 20 mV.

The A/D conversion converts 1 V to 200 values. Therefore, the noise of maximal 20 mV is converted to $200 \cdot 0.020 = 4$ values. That means the bandwidth of the noise is as much as 4 units in the isoplots. That is relatively high, since the bandwidth of the isoplots (at that time) was also 4 units. Therefore it was important to find a way to temper the influence of noise.

We have chosen to read the sensor values multiple times, and to return the median. Using the median is a way to get rid of extreme values (unlike the average would do). But in order to use multiple observations, we need independent measurements, i.e. the sensor values should be uncorrelated. However, a too long delay between those observations is not possible, since we want to minimize the time of reading the sensors. Not only for real-time purposes, but also because the sensor values change significantly the observations are taken with a too big time difference.

We want to calculate the covariance between the consecutive sensor values. We need to find the minimal time difference Δt between observations that has a covariance of zero. To get good data, we first inflate the muscles completely, and then close the valves. From the moment the valves are closed, we store the scope data. The only change in pressure that occurs now is due to the leakage of the pipes and muscles. To discard this influence, we *detrend* this data to time. This makes the data stationary.

The Power Spectral Density (PSD) function is the Fourier transform of the covariance function [31]. Therefore, we can calculate the covariance function by taking the inverse Fourier transform of the PSD. The result shows that we deal with white noise. So the correlation between observations is less than a few microseconds. Therefore, we can observe the sensor values with almost no additional delay. We wrote a special function on the chip, that reads the sensors 5, 7 or 9 times (can be set with another function). This function reads the observations, and with a Bubblesort algorithm the values are sorted. Next, the middle index is returned (the median).

A.1.4 Chip

Real-time requirements. In Section 2.3.3 we explained that the chip is adjusted to return the sensor values in the minimum time possible. We used a minimal operating system on the chip, where are ACK's and NAK's are removed (acknowledges and not-acknowledges), to save time. We implemented a function that read all sensors at once. For every sensor, the noise is filtered by applying the above description of the noise. All values are written to the chip's buffer.

Driver bugs. The driver contains an error. First we developed a workaround for this problem, later we discovered where this error resides. The problem is that the driver sometimes claims to have read the buffer, when it actually has not. The driver returns that the buffer is empty, whereas in reality the chip did place the answer in its buffer.

We developed two ways to circumvent there errors. One is the use of a timestamp. The chip has its own timer, but it is not used in the standard manufacturer's software. We added a function that resets the timer. We add the timestamp to the answer, just before the chip places the whole package in its buffer. Another way to circumvent the driver's problem is via the use of headers. With every answer of the chip, also the number of bytes of the answer is used, together with the function name that put the request, and an additional checksum. This prevents problems when the answer is processed at the PC.

Appendix B

Derivations

B.1 Gaussian noise in moving average

In this section we show the relation between the moving average and the standard deviation of a parameter.

Suppose we calculate the moving average ma of the last n values from θ:

$$ma(k+1) = \frac{n-1}{n} * ma(k) + \frac{1}{n} * \theta(k) \tag{B.1}$$

The goal is to calculate the standard deviation of $std_{ma(k+1)}$. Now assume the values of θ come from a Gaussian probability distribution with mean 0 and standard deviation std_θ. We know that if we add two Gaussian distributed variables, the result is again a Gaussian distributed variable with a different variance:

$$std^2_{new} = std^2_1 + std^2_2 \tag{B.2}$$

So we know that the moving average ma is Gaussian distributed, but with a different variance. However, we cannot simply use Equation B.2, since Equation B.1 uses the factors $\frac{n-1}{n}$ and $\frac{1}{n}$.

What happens if we take a value with a Gaussian probability distribution and multiply this value by a factor f? The standard deviation is then also multiplied by f. With this observation, we indeed can use Equation B.2:

$$\begin{aligned}
std^2_{ma(k+1)} &= \left(std_{ma(k)} * \frac{n-1}{n}\right)^2 + \left(std_{\theta(k)} * \frac{1}{n}\right)^2 && \text{(B.3)}\\
std^2_{ma} &= \left(std_{ma} * \frac{n-1}{n}\right)^2 + \left(std_{\theta} * \frac{1}{n}\right)^2 && \text{(B.4)}\\
std^2_{ma} &= (std_{ma})^2 * \left(\frac{n-1}{n}\right)^2 + (std_\theta)^2 * \left(\frac{1}{n}\right)^2 && \text{(B.5)}\\
std^2_{ma}\left(1 - \left(\frac{n-1}{n}\right)^2\right) &= (std_\theta)^2 * \left(\frac{1}{n}\right)^2 && \text{(B.6)}\\
std^2_{ma}\left(\frac{2}{n} - \frac{1}{n^2}\right) &= (std_\theta)^2 * \left(\frac{1}{n}\right)^2 && \text{(B.7)}\\
std^2_{ma}\left(\frac{\frac{2}{n} - \frac{1}{n^2}}{\frac{1}{n^2}}\right) &= (std_\theta)^2 && \text{(B.8)}\\
std^2_{ma}(2n-1) &= (std_\theta)^2 && \text{(B.9)}
\end{aligned}$$

So:

$$std_\theta = \sqrt{2n-1}(std_{ma}) \tag{B.10}$$

And hence,

$$std_{ma} = \frac{std_\theta}{\sqrt{2n-1}} \tag{B.11}$$

Appendix C

Experiments

C.1 Punish a switch

Table C.1 shows the five control factors and their corresponding levels. Three rounds are performed. The goal is to learn a single angle $\theta = 650$ (this angle is chosen randomly). Refer to Table C.2 to Table C.5 for the results.

Control factor	Label	Level -1	Level +1
learning rate	a	0.05	0.10
number of neurons	b	5	10
TD-method	c	SARSA	Q-learning
eligibility traces	d	0	0.3
punishment on switch	e	-8	0

Table C.1: Five control factors with their levels.

C.2 All goal states

Table C.6 shows the five control factors and their corresponding levels. This table is a copy of Table 5.4. Eight rounds are performed. The goal is to learn all goal angles.

trial	run 1	run 2	run 3
1	12.60	13.93	14.11
2	37.54	35.94	35.93
3	38.24	35.76	36.21
4	-1.05	13.81	10.60
5	7.93	12.75	10.98
6	34.90	35.84	33.81
7	35.54	35.94	33.89
8	10.50	13.83	12.24
9	11.73	12.84	11.78
10	35.89	35.76	33.60
11	35.64	36.16	34.01
12	12.81	14.07	11.82
13	13.37	14.05	12.44
14	35.51	35.78	33.81
15	35.73	36.27	34.10
16	12.76	13.81	11.55

Table C.2: Average cumulative reward. Keep in mind that the lower level of factor e punishes a switch.

trial	run 1	run 2	run 3
1	2.91	2.64	2.63
2	5.26	7.09	7.31
3	5.18	7.93	8.15
4	3.82	2.65	2.68
5	3.06	2.75	2.68
6	7.09	7.17	7.79
7	8.37	8.37	8.74
8	2.90	2.61	2.56
9	2.79	2.75	2.63
10	7.05	7.32	7.85
11	8.38	8.19	8.86
12	2.68	2.65	2.61
13	2.68	2.64	2.56
14	7.38	7.43	7.65
15	8.20	7.92	8.08
16	2.76	2.65	2.63

Table C.3: Average number of switches.

trial	run 1	run 2	run 3
1	15.12	15.91	15.84
2	13.45	15.06	15.07
3	12.76	15.23	14.78
4	21.44	15.98	18.94
5	18.60	16.23	18.57
6	16.10	15.16	17.19
7	15.46	15.06	17.11
8	17.28	16.29	18.25
9	16.96	16.14	18.19
10	15.11	15.24	17.40
11	15.36	14.84	16.99
12	16.77	15.70	18.26
13	16.20	15.82	18.06
14	15.49	15.22	17.19
15	15.27	14.73	16.90
16	16.17	15.98	18.45

Table C.4: Average number of actions.

trial	run 1	run 2	run 3
1	13500	14000	13500
2	30000	22000	19000
3	15500	20500	15500
4	13500	14000	13000
5	14000	13000	14000
6	21000	14000	19500
7	16000	20000	14000
8	14000	14000	13500
9	13500	14000	14000
10	18000	27500	20500
11	14000	17000	17000
12	13500	13500	13500
13	13500	13500	14000
14	16000	18500	14000
15	15000	13500	15000
16	14000	13500	14000

Table C.5: Average number of episodes.

Control factor	Label	Level -1	Level +1
learning rate	a	0.05	0.10
number of neurons	b	5	10
TD-method	c	SARSA	Q-learning
eligibility traces	d	0	0.3
discount factor	e	0.95	0.98

Table C.6: Five control factors with their levels.

trial	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
1	4.75	9.34	0.55	3.59	4.56	2.52	5.09	6.83
2	2.90	7.81	-1.35	7.03	4.80	6.69	4.76	8.15
3	10.24	10.10	-0.14	9.12	-5.89	10.43	11.30	-10.62
4	0.17	4.06	2.84	3.06	3.29	4.51	3.08	3.41
5	8.26	4.60	-0.01	3.07	5.50	8.62	-6.53	-1.46
6	12.30	9.18	8.86	11.42	0.67	10.60	8.86	11.92
7	8.00	10.66	-6.27	9.74	6.23	8.13	-25.85	9.12
8	1.10	-8.18	9.66	2.10	-8.11	-0.90	-17.39	3.90
9	-1.54	6.581	2.67	-11.17	8.23	7.24	5.40	8.81
10	11.00	9.81	12.10	11.54	9.27	8.21	10.53	8.98
11	2.10	9.22	1.23	9.15	-8.74	-10.12	-14.69	7.69
12	8.64	2.39	-2.12	4.20	4.26	5.78	-1.20	-7.74
13	8.57	10.60	6.80	-4.40	2.80	8.36	2.39	-6.64
14	5.66	8.37	5.26	8.57	6.88	1.00	7.27	9.70
15	-2.01	-36.70	6.84	3.82	8.48	-10.58	-2.53	2.89
16	-0.19	-8.72	-0.43	6.84	-3.99	-15.81	-22.67	0.31

Table C.7: Average cumulative reward per episode in the testing phase.

trial	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
1	19.16	18.66	19.31	18.96	18.97	19.68	19.18	18.85
2	19.45	17.29	19.92	17.76	18.38	17.72	18.31	17.09
3	17.62	17.32	21.02	18.42	24.75	17.86	16.74	25.96
4	20.29	18.98	19.29	20.52	19.48	18.23	18.37	18.62
5	18.36	19.08	19.17	18.58	18.36	18.23	22.27	22.55
6	17.05	18.17	17.68	16.19	19.91	16.98	17.38	16.38
7	18.10	17.37	25.26	17.34	18.74	18.75	30.39	18.39
8	21.70	25.20	17.48	19.35	24.16	21.53	30.36	19.10
9	20.56	17.15	19.31	28.84	17.72	17.72	18.46	17.25
10	17.23	17.38	16.61	16.69	16.91	18.16	17.18	17.80
11	20.84	18.22	20.82	17.68	24.33	25.81	27.43	19.09
12	18.29	19.01	22.12	18.96	18.72	17.52	20.69	25.12
13	18.14	17.51	18.76	22.12	19.42	17.83	20.78	21.80
14	18.31	17.49	18.83	17.33	18.50	19.74	18.03	17.11
15	22.01	36.38	19.17	19.97	17.29	26.15	22.57	20.96
16	21.77	25.69	19.92	17.74	24.43	27.69	30.97	19.97

Table C.8: Average number of actions per episode.

trial	run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
1	3.38	2.87	3.89	3.55	3.43	3.60	3.34	3.16
2	3.58	3.24	4.05	3.28	3.48	3.32	3.49	3.22
3	2.89	2.95	3.59	2.93	3.87	2.84	2.87	4.44
4	3.78	3.48	3.61	3.43	3.53	3.53	3.69	3.59
5	3.05	3.42	3.98	3.67	3.39	3.02	4.33	3.72
6	2.71	2.96	3.06	2.92	3.80	2.93	3.10	2.84
7	3.11	2.87	3.99	2.99	3.25	3.01	5.38	2.94
8	3.51	4.18	2.98	3.68	4.34	3.76	4.41	3.50
9	3.99	3.41	3.62	3.89	3.13	3.25	3.39	3.12
10	2.85	2.98	2.79	2.85	3.10	3.08	2.91	3.03
11	3.51	2.95	3.54	3.02	4.39	4.39	4.64	3.03
12	3.01	3.70	3.82	3.45	3.50	3.46	3.93	4.17
13	3.04	2.86	3.18	4.15	3.59	3.10	3.47	4.45
14	3.38	3.14	3.36	3.14	3.20	3.78	3.21	3.03
15	3.84	6.22	3.12	3.40	3.15	4.37	3.85	3.34
16	3.61	4.17	3.91	3.30	3.82	4.57	5.00	3.84

Table C.9: Average number of switches per episode.

trial	round 1	round 2	round 3
1	33500	30500	28500
2	15000	15500	16000
3	19000	17500	22000
4	22000	22500	24500
5	34000	22500	41000
6	15000	14500	18500
7	17500	16000	22000
8	40000	28000	39000
9	29500	48000	28500
10	18500	17000	16500
11	21000	32000	17000
12	41000	35000	21000
13	44000	36000	38000
14	15500	16500	15500
15	20000	47500	24000
16	30000	23500	30000

Table C.10: Number of episodes used for training phase.

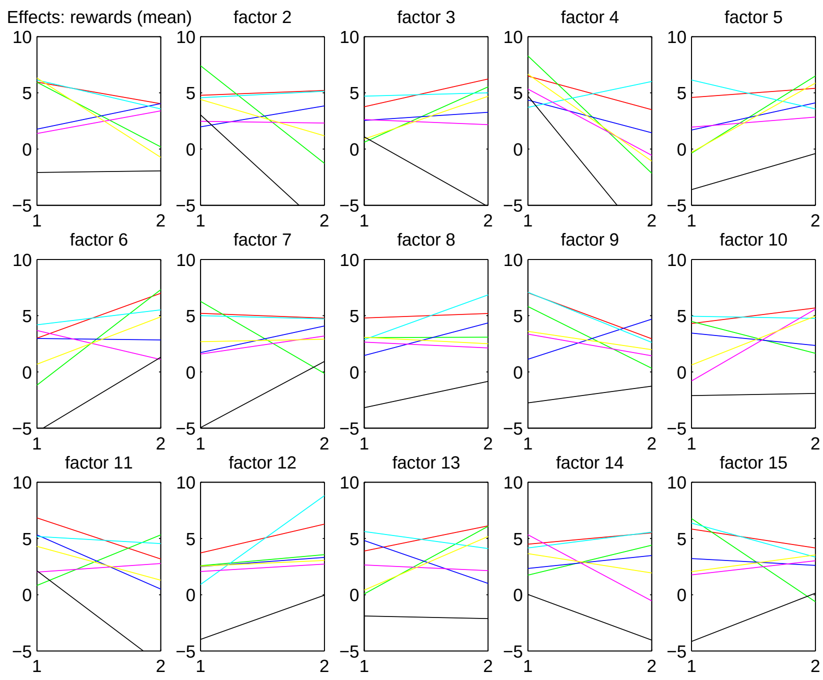

Figure C.1: Effects plot: mean cumulative reward for 7 runs separately.

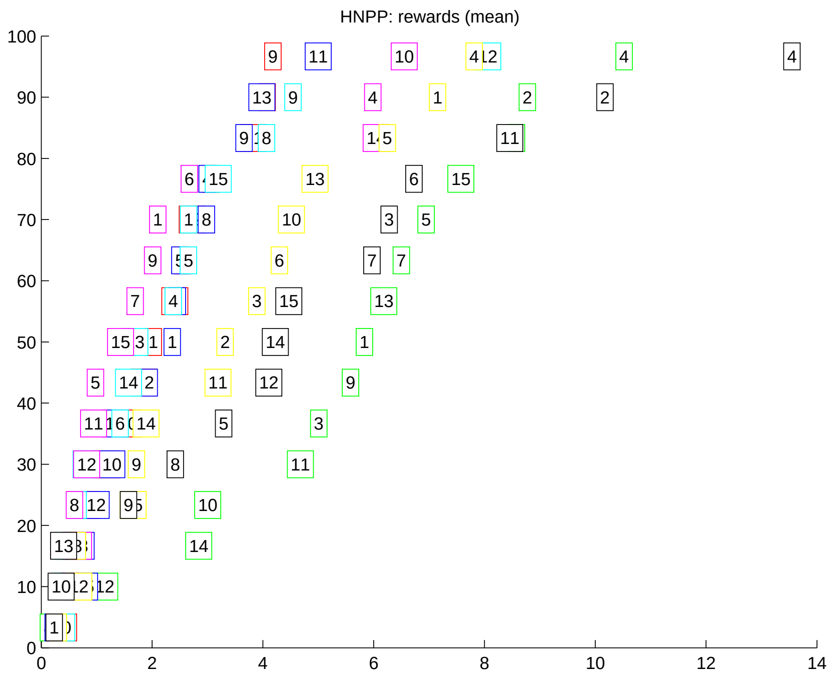

Figure C.2: Hnpp: mean cumulative reward for 7 runs separately.

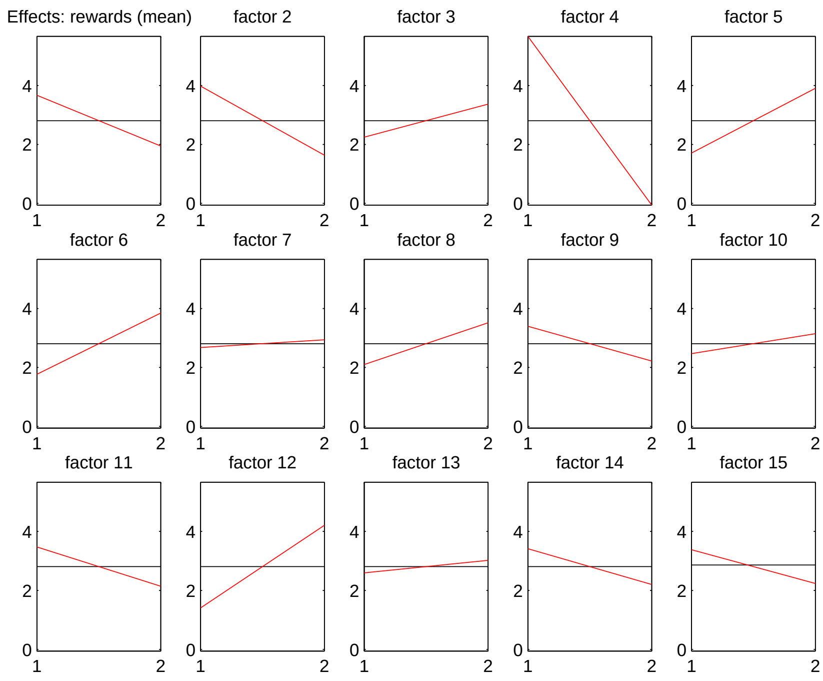

Figure C.3: Effects: mean cumulative reward.

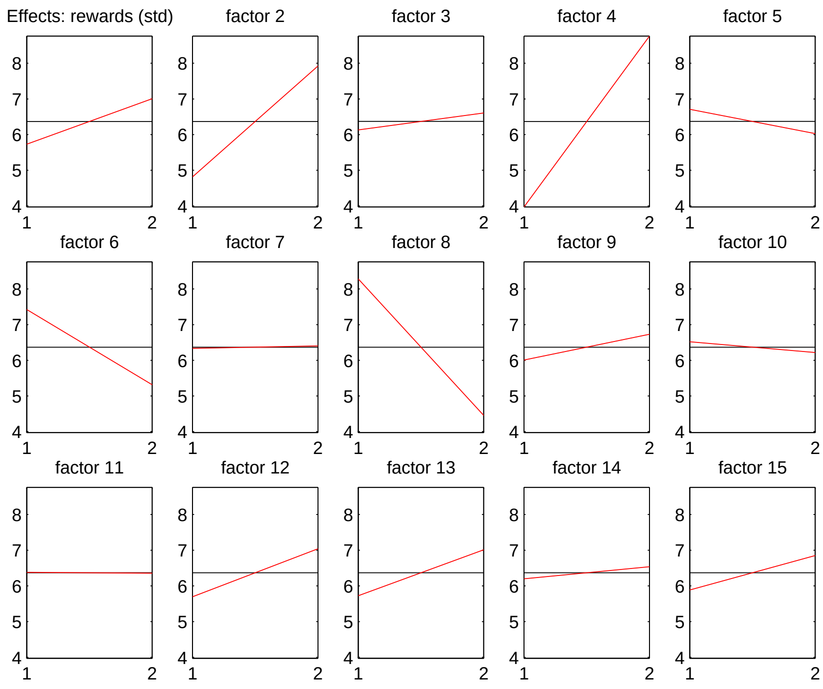

Figure C.4: Effects: std of cumulative reward.

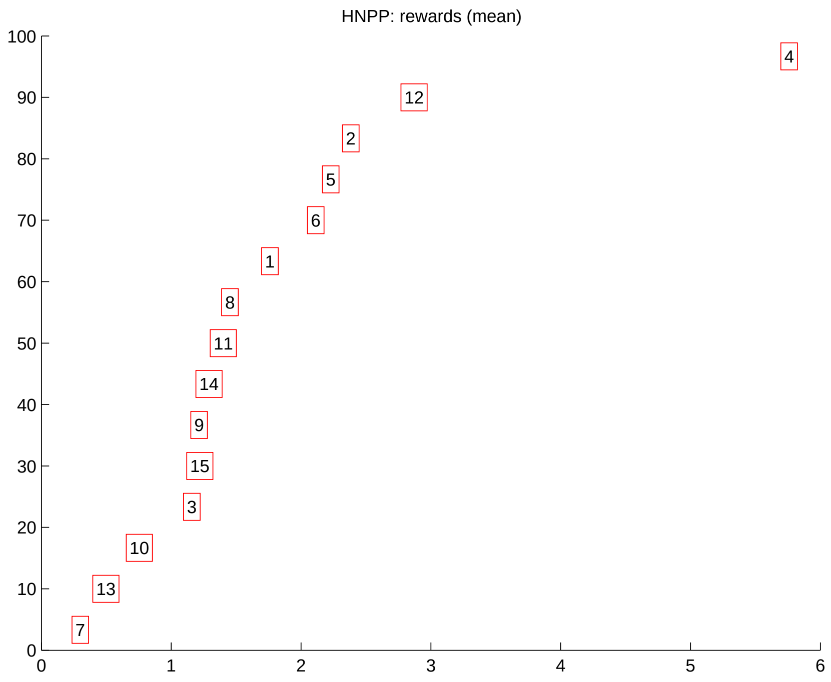

Figure C.5: Hnpp: mean cumulative reward.

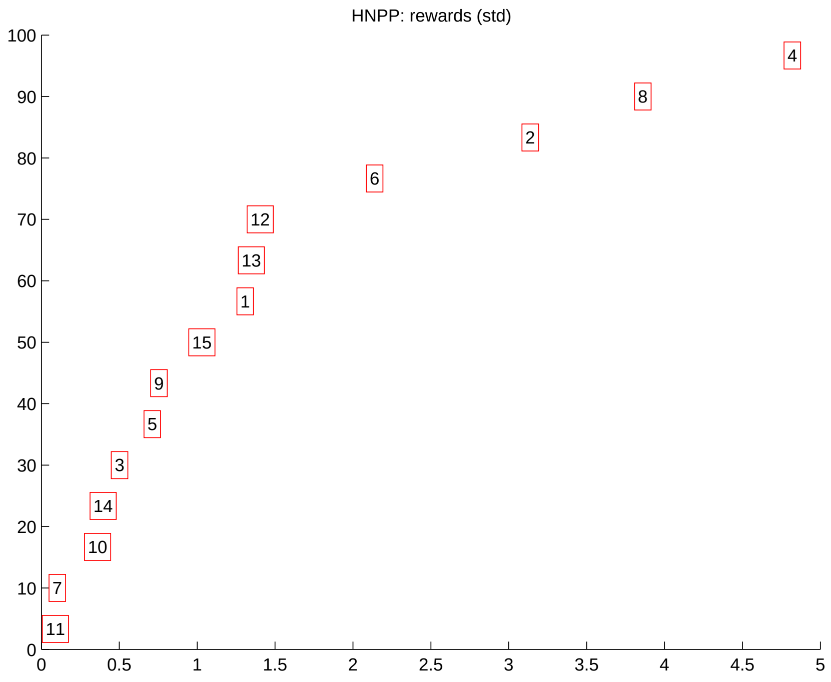

Figure C.6: Hnpp: std of cumulative reward.

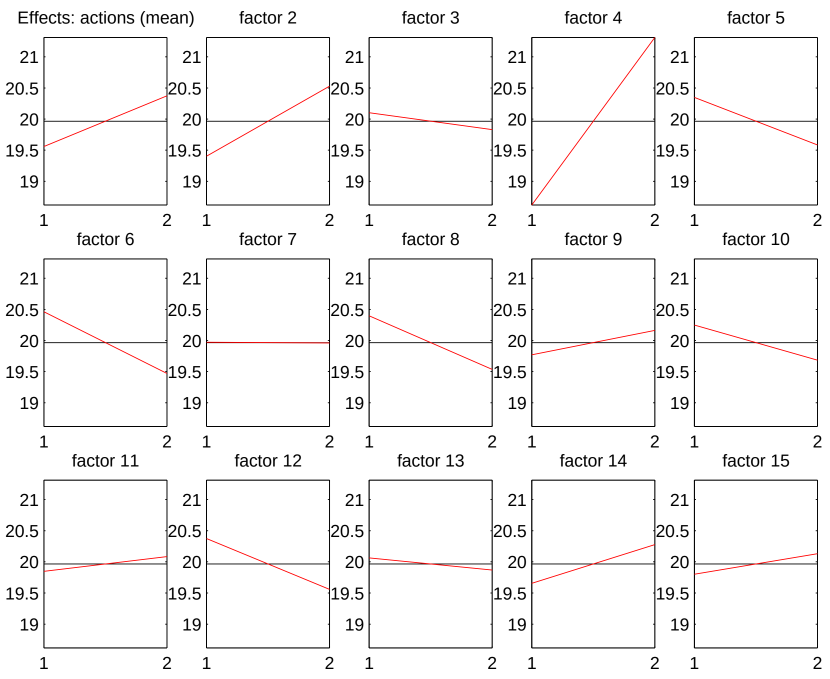

Figure C.7: Effects: mean number of actions.

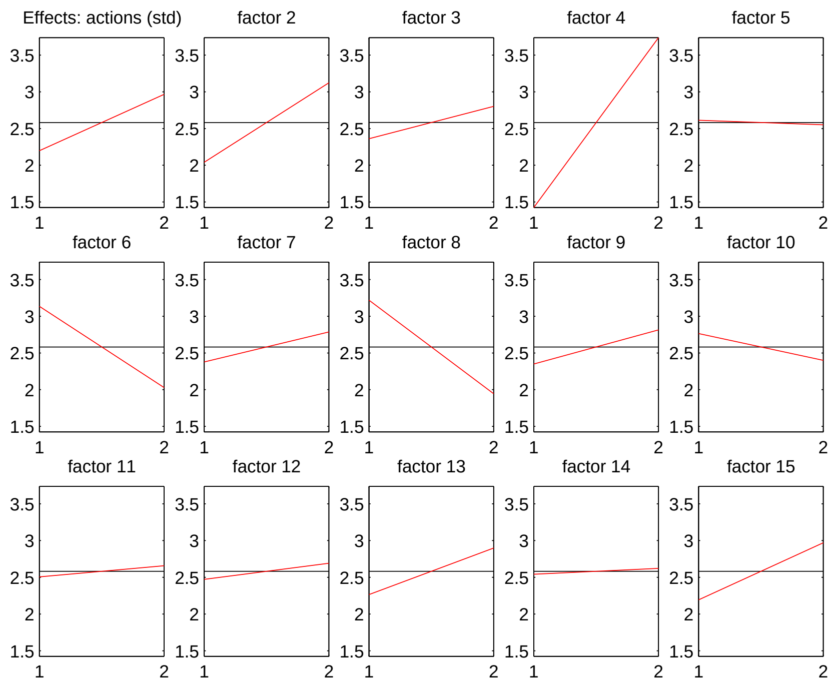

Figure C.8: Effects: std actions.

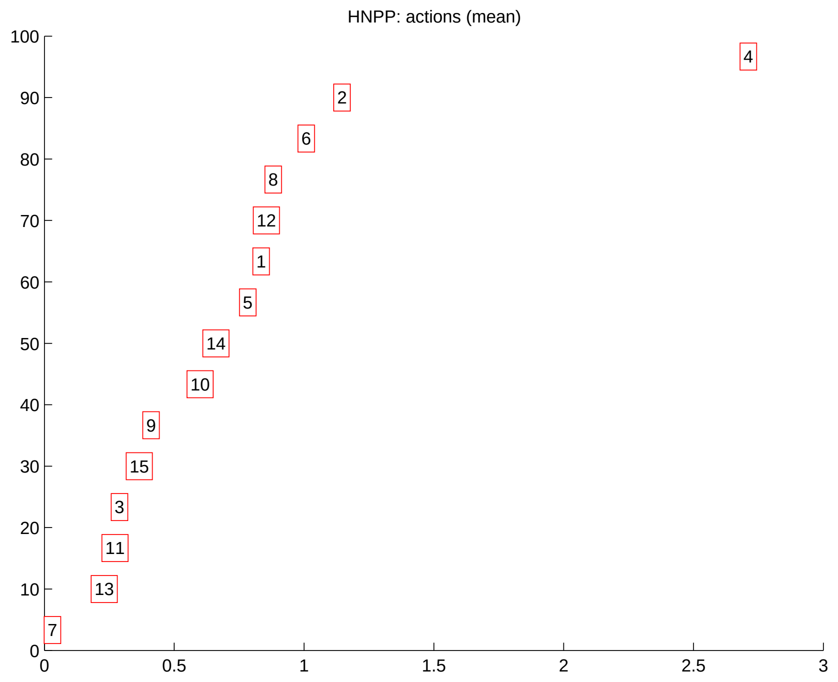

Figure C.9: Hnpp: mean number of actions.

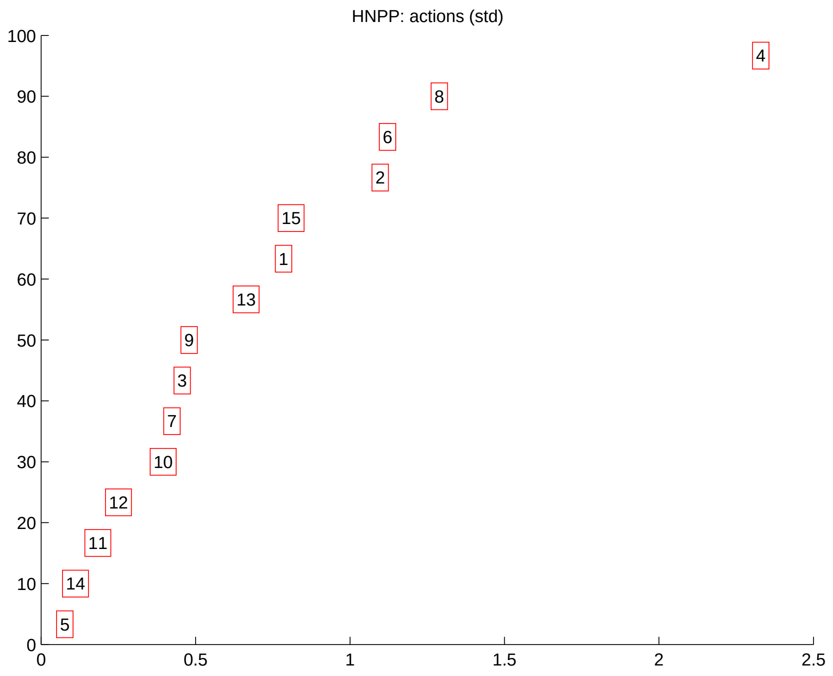

Figure C.10: Hnpp: std actions.

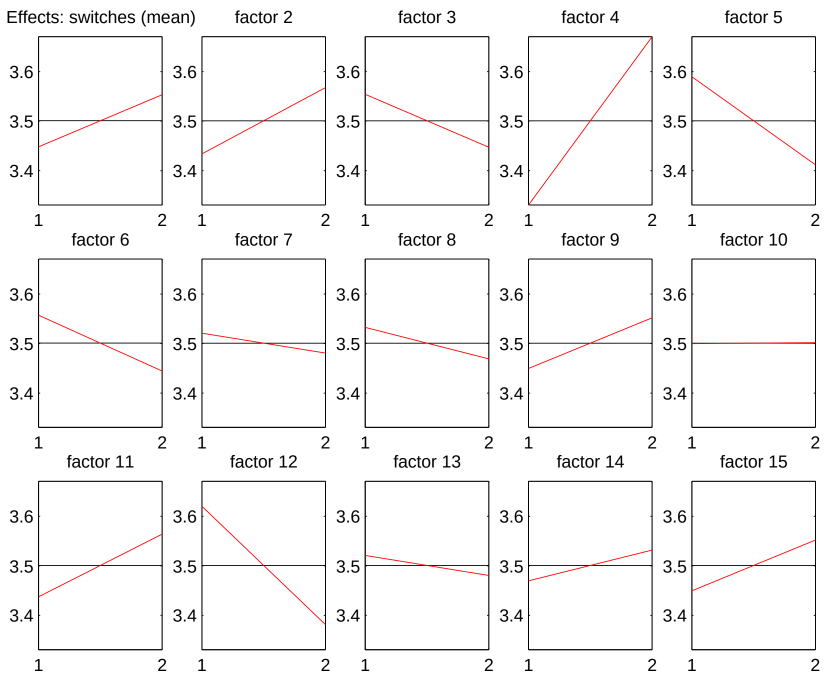

Figure C.11: Effects: mean number of switches.

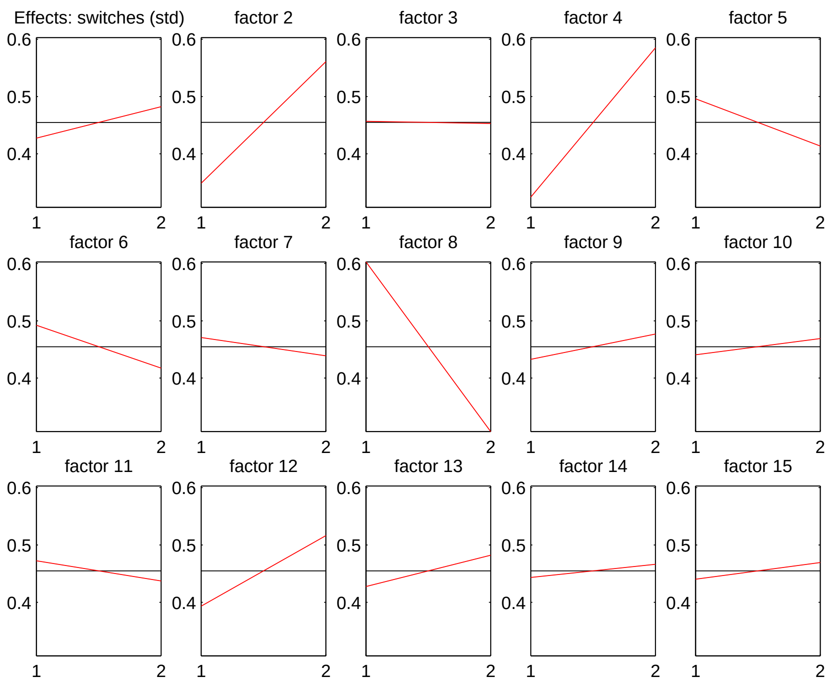

Figure C.12: Effects: std switches.

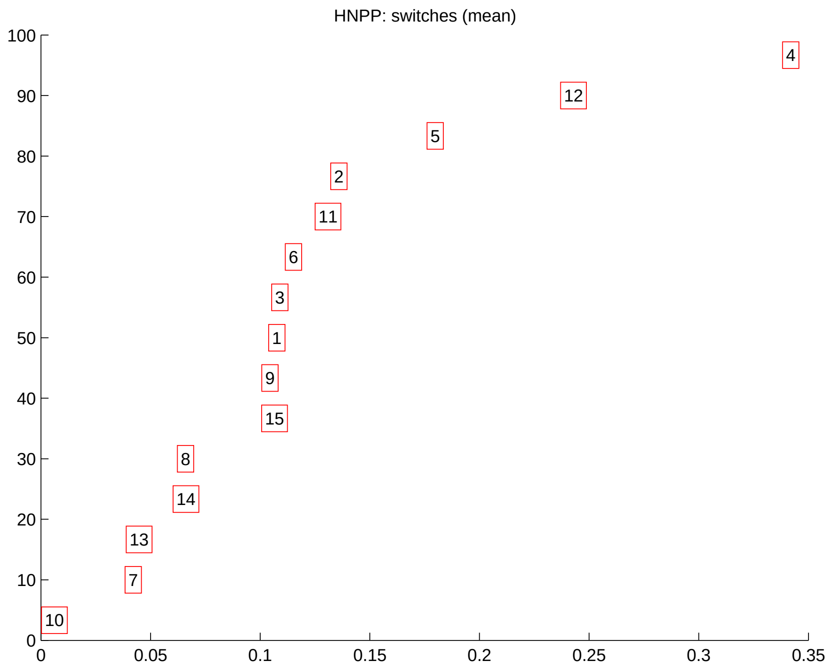

Figure C.13: Hnpp: mean number of switches.

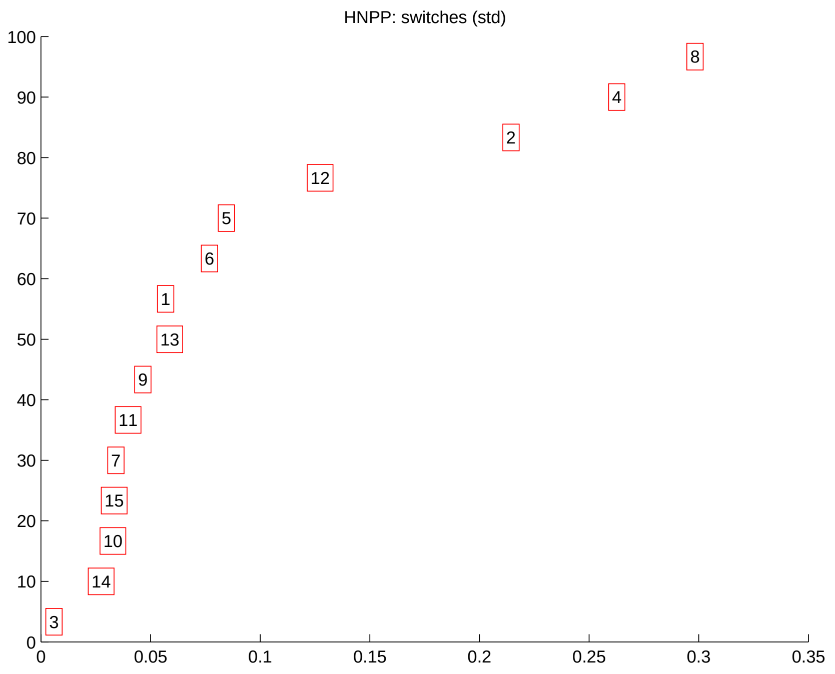

Figure C.14: Hnpp: std switches.

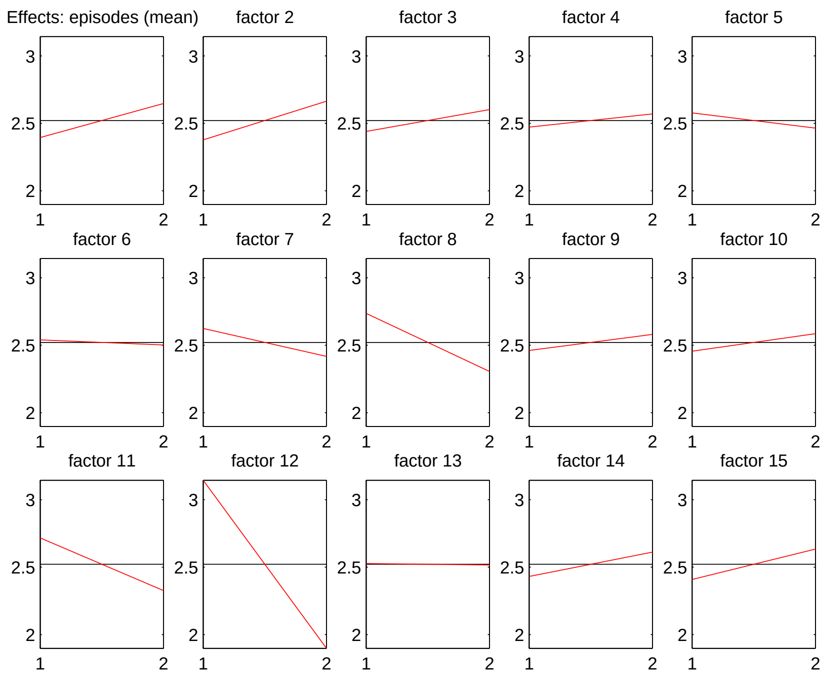

Figure C.15: Effects: mean number of episodes.

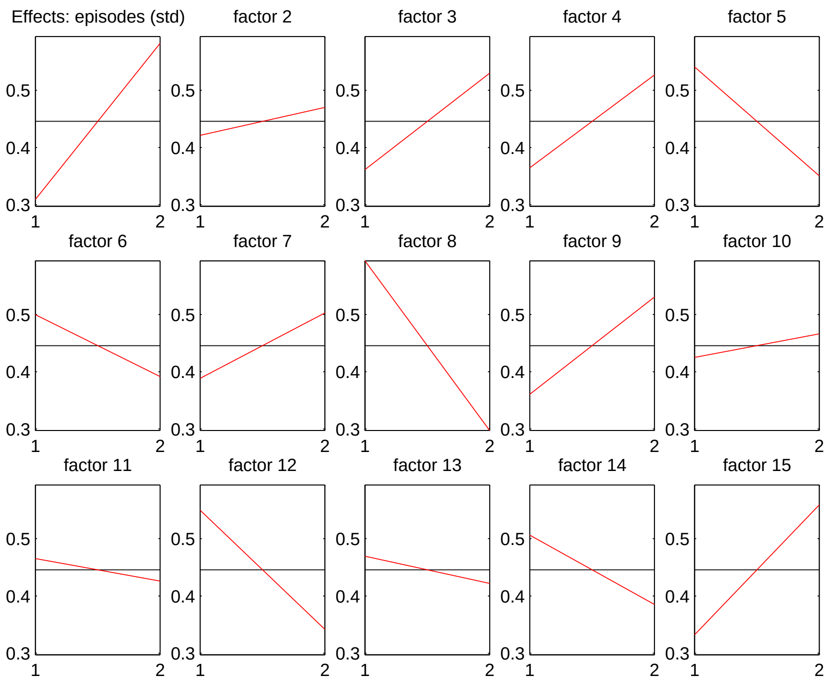

Figure C.16: Effects: std episodes.

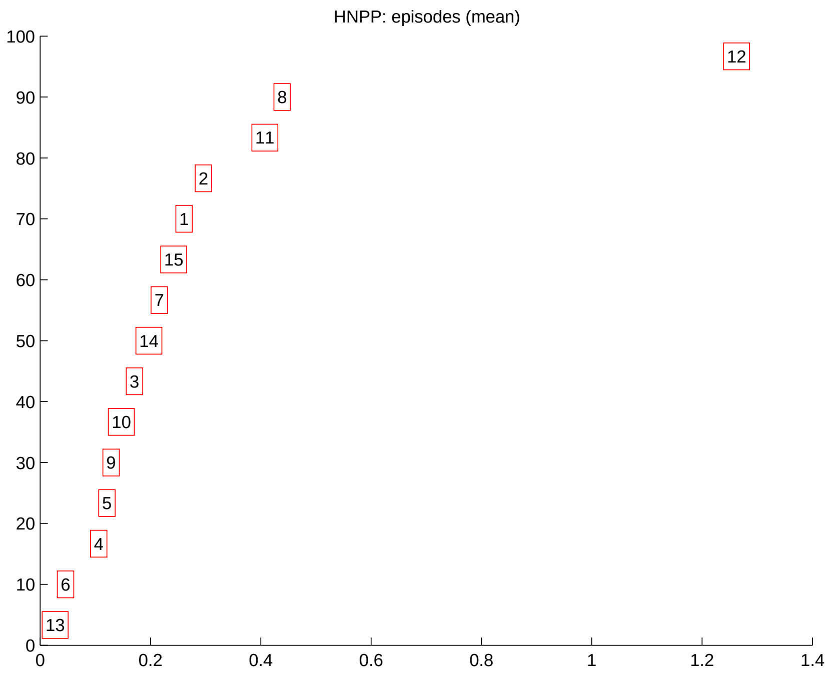

Figure C.17: Hnpp: mean number of episodes.

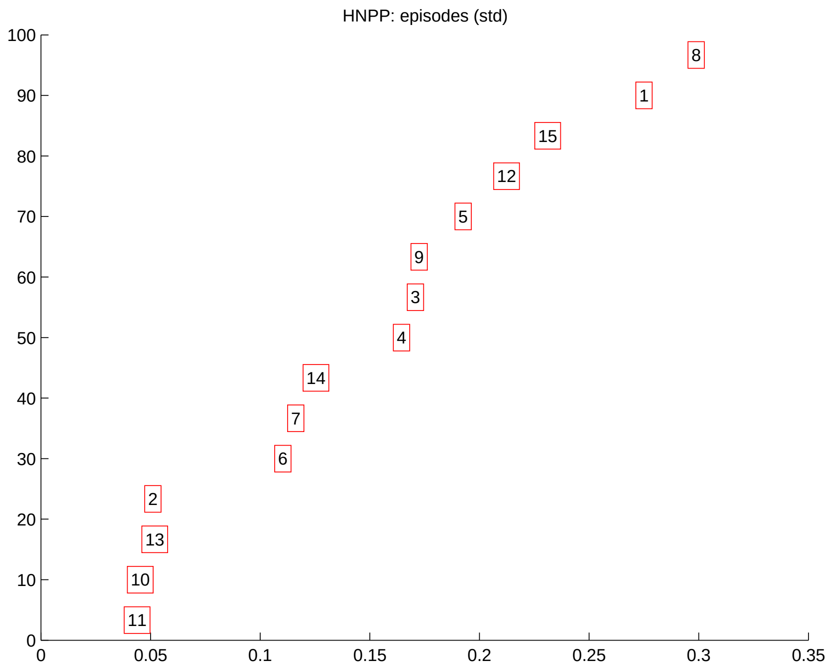

Figure C.18: Hnpp: std episodes.

C.3 All goals after 50,000 episodes

trial	run 1	Δ run 1	run 2	Δ run 2	run 3	Δ run 3
1	8.46	3.72	9.00	-0.34	7.08	6.53
2	10.30	7.40	9.08	1.27	5.61	6.96
3	-1.59	-11.83	11.53	1.43	12.21	12.35
4	10.62	10.45	11.01	6.95	6.97	4.12
5	10.03	1.77	8.56	3.97	3.65	3.66
6	12.64	0.34	11.80	2.62	9.57	0.72
7	12.90	4.90	13.24	2.58	8.75	15.03
8	1.23	0.13	4.69	12.87	10.66	1.00
9	7.09	8.62	6.50	-0.08	8.69	6.02
10	12.24	1.24	10.96	1.14	12.30	0.20
11	7.70	5.61	10.27	1.05	8.02	6.79
12	9.16	0.52	7.78	5.39	6.24	8.35
13	11.05	2.47	11.63	1.03	10.26	3.45
14	10.22	4.56	7.63	-0.75	9.51	4.25
15	8.29	10.30	-38.07	-1.37	9.27	2.44
16	7.86	8.04	-22.12	-13.40	7.70	8.13

Table C.11: Average cumulative reward after 50,000 episodes.

trial	run 1	run 2	run 3
1	17.89	17.73	17.94
2	17.22	16.68	18.53
3	22.99	17.05	16.47
4	16.86	16.29	17.71
5	17.37	17.84	18.45
6	16.83	17.27	17.25
7	16.47	16.38	17.83
8	21.27	19.14	16.89
9	17..94	17.23	17.83
10	16.77	16.54	16.28
11	18.55	17.38	18.32
12	17.39	17.22	19.21
13	17.08	17.14	17.19
14	16.90	17.36	17.68
15	18.69	36.59	18.02
16	17.68	31.48	16.78

Table C.12: Average number of actions after 50,000 episodes.

trial	run 1	run 2	run 3
1	3.08	3.03	3.25
2	2.94	3.16	3.36
3	3.70	2.80	2.79
4	2.94	2.96	3.29
5	2.95	3.07	3.61
6	2.69	2.74	3.02
7	2.70	2.67	3.05
8	3.56	3.40	2.93
9	3.24	3.41	3.06
10	2.75	2.94	2.80
11	3.09	2.92	3.08
12	3.06	3.25	3.19
13	2.86	2.78	2.94
14	2.99	3.25	2.98
15	3.00	6.31	2.96
16	3.18	4.63	3.31

Table C.13: Average number of switches after 50,000 episodes.

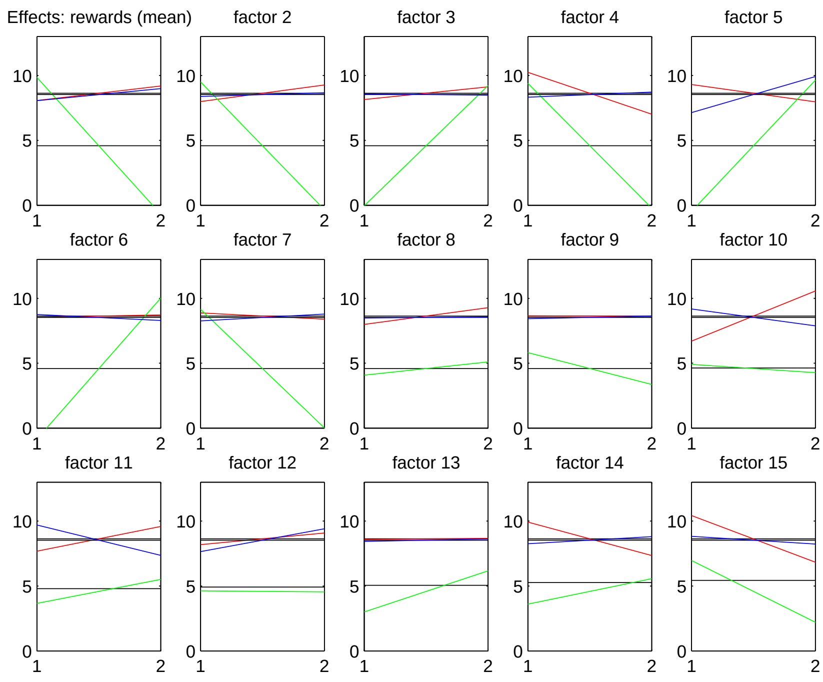

Figure C.19: Effects plot: mean cumulative reward for 3 runs separately.

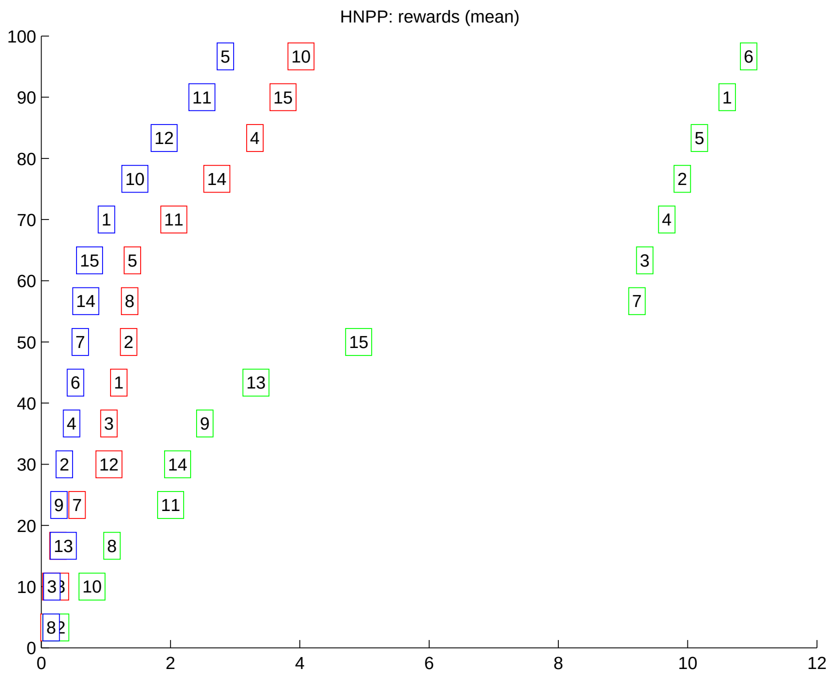

Figure C.20: Hnpp: mean cumulative reward for 3 runs separately.

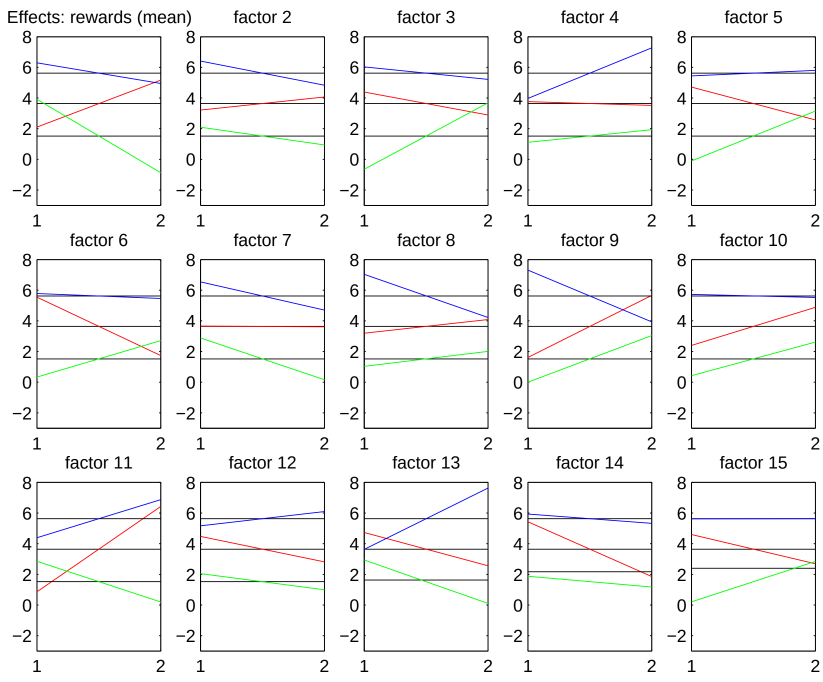

Figure C.21: Effects plot: mean cumulative reward for 3 runs separately of differences (between previous result and 50,000 episodes).

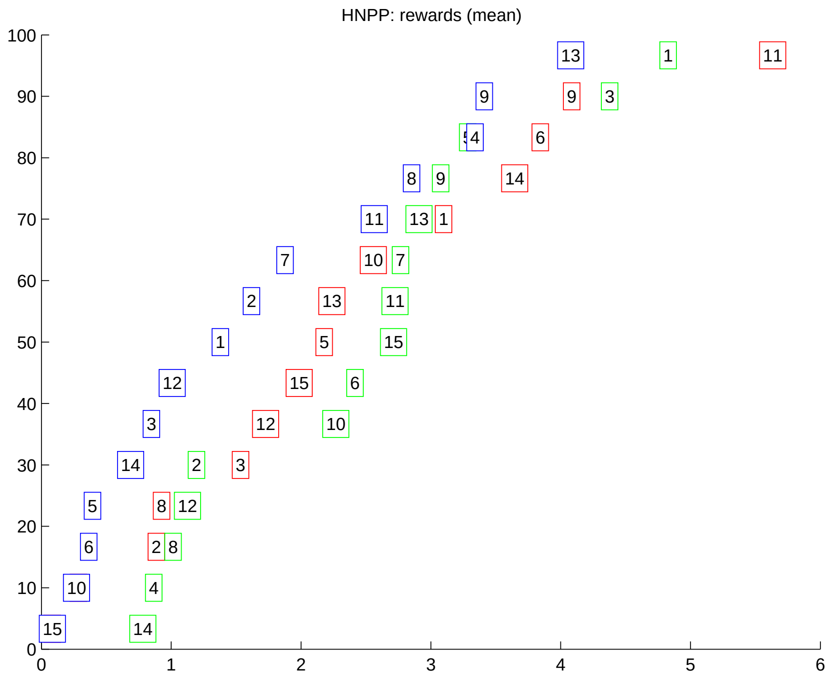

Figure C.22: Hnpp: mean cumulative reward for 3 runs separately of differences (between previous result and 50,000 episodes).

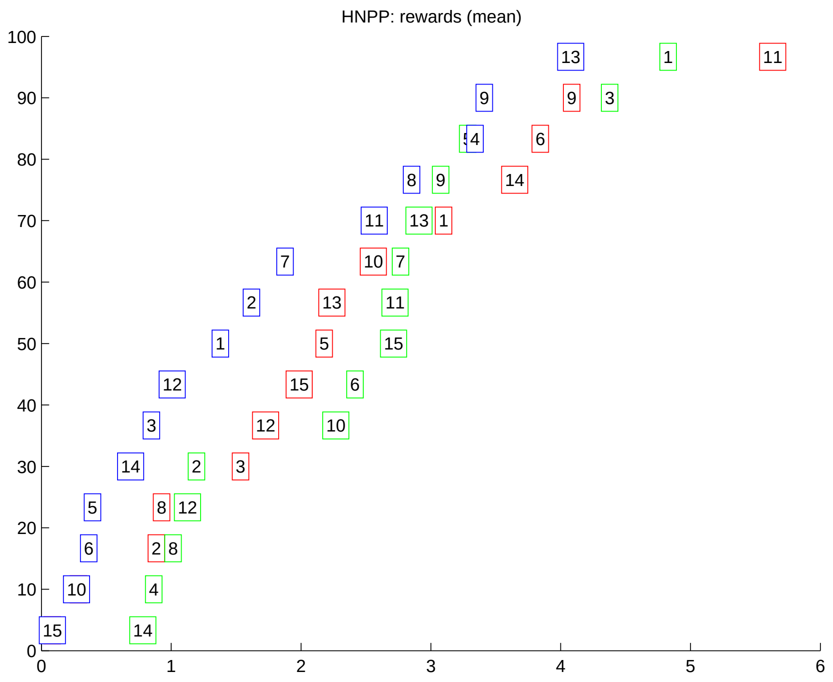

Figure C.23: Hnpp: mean cumulative reward for 3 runs separately of differences (between previous result and 50,000 episodes).

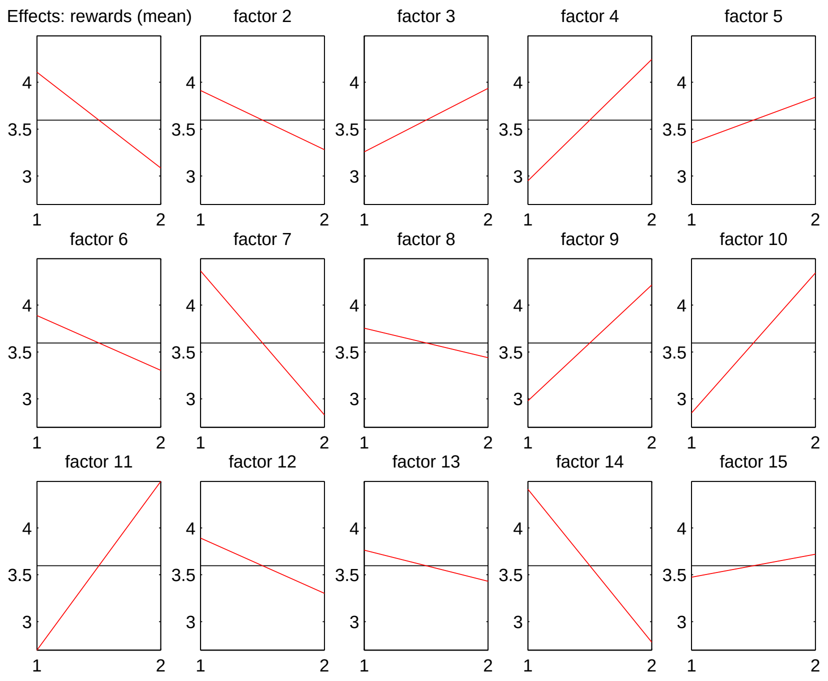

Figure C.24: Effects: mean cumulative reward of differences (between previous result and 50,000 episodes).

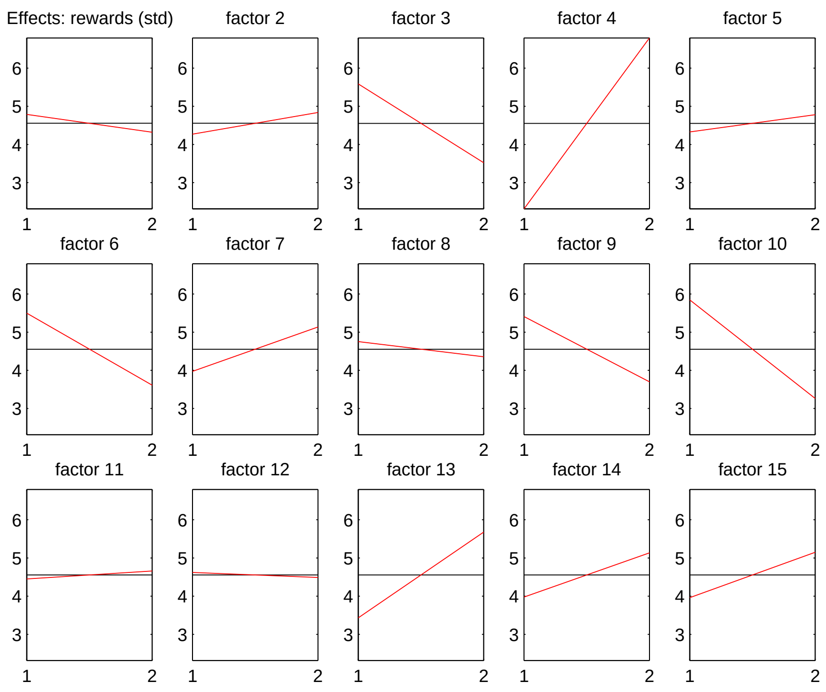

Figure C.25: Effects: std of cumulative reward of differences (between previous result and 50,000 episodes).

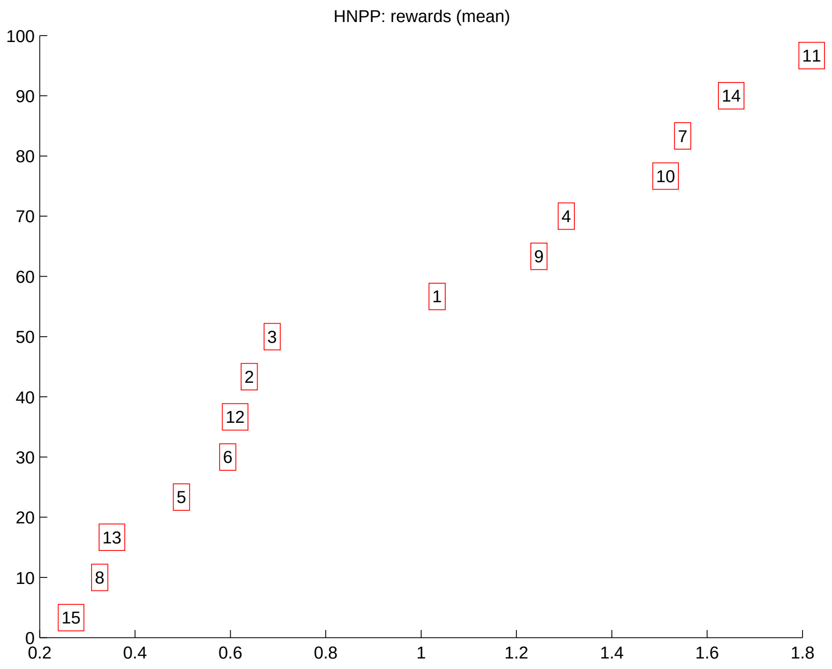

Figure C.26: Hnpp: mean cumulative reward of differences (between previous result and 50,000 episodes).

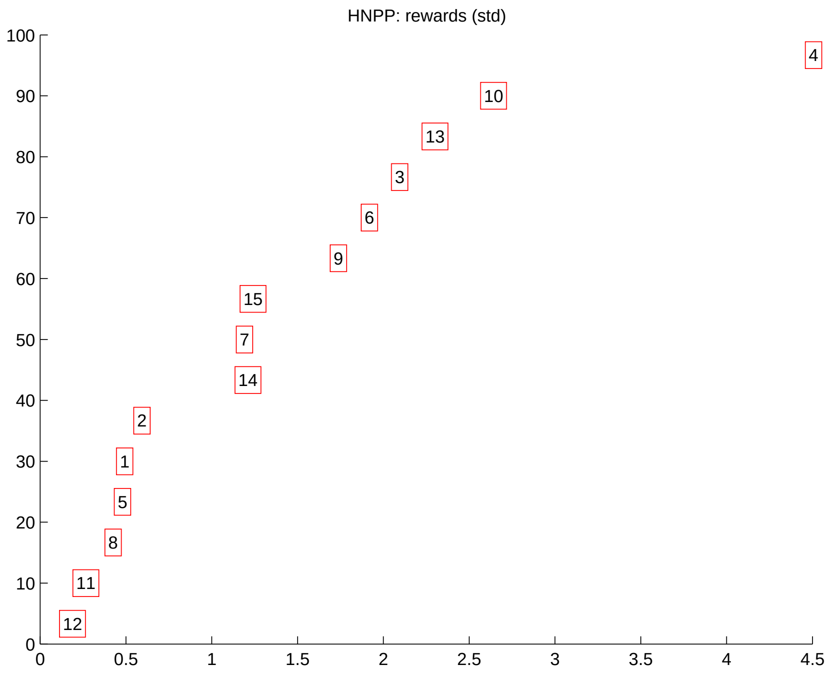

Figure C.27: Hnpp: std of cumulative reward of differences (between previous result and 50,000 episodes).

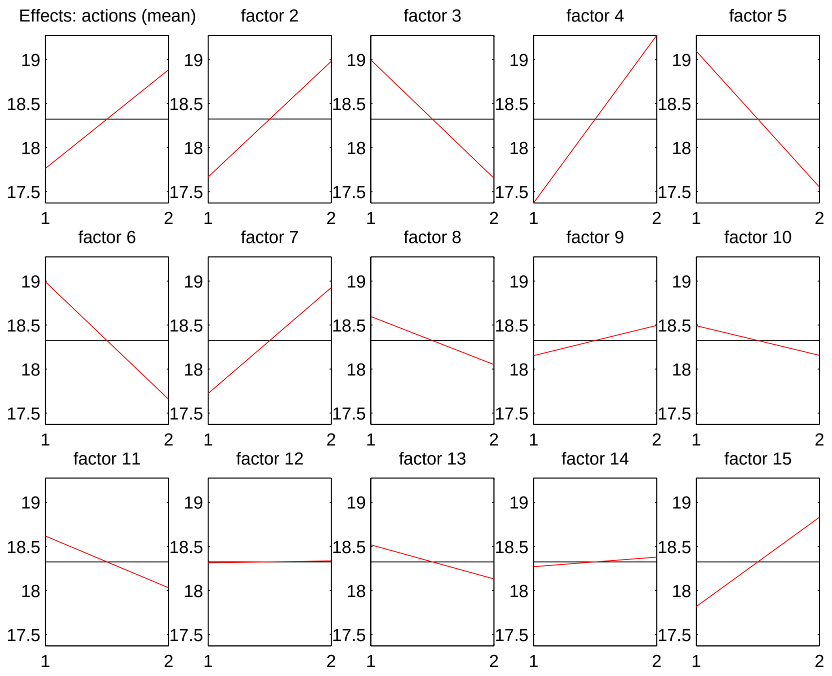

Figure C.28: Effects: mean number of actions.

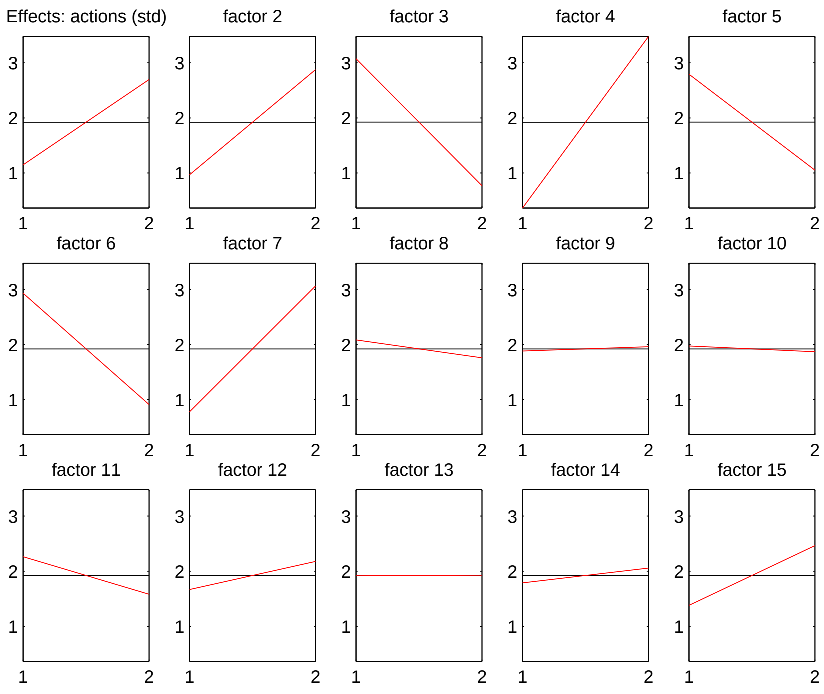

Figure C.29: Effects: std actions of differences.

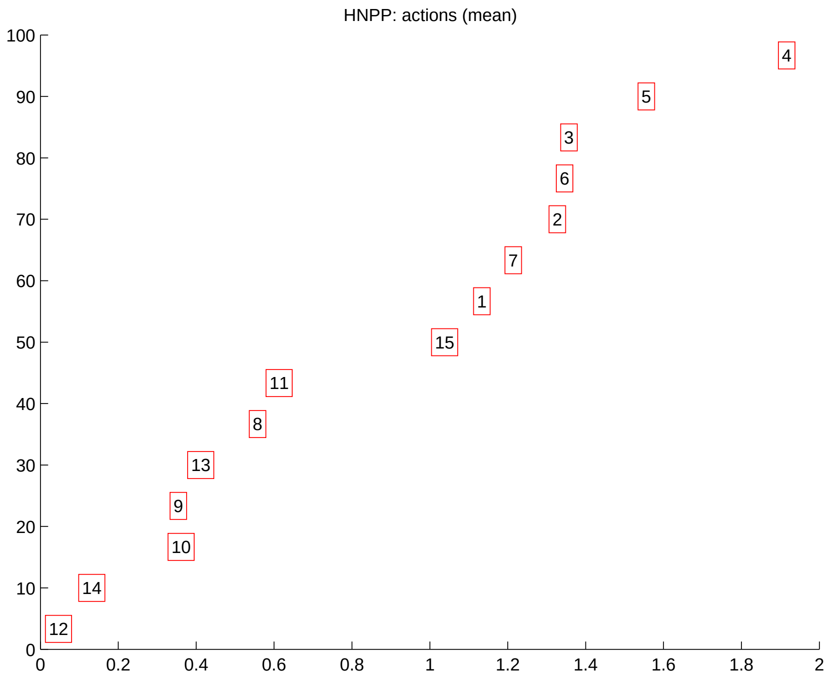

Figure C.30: Hnpp: mean number of actions.

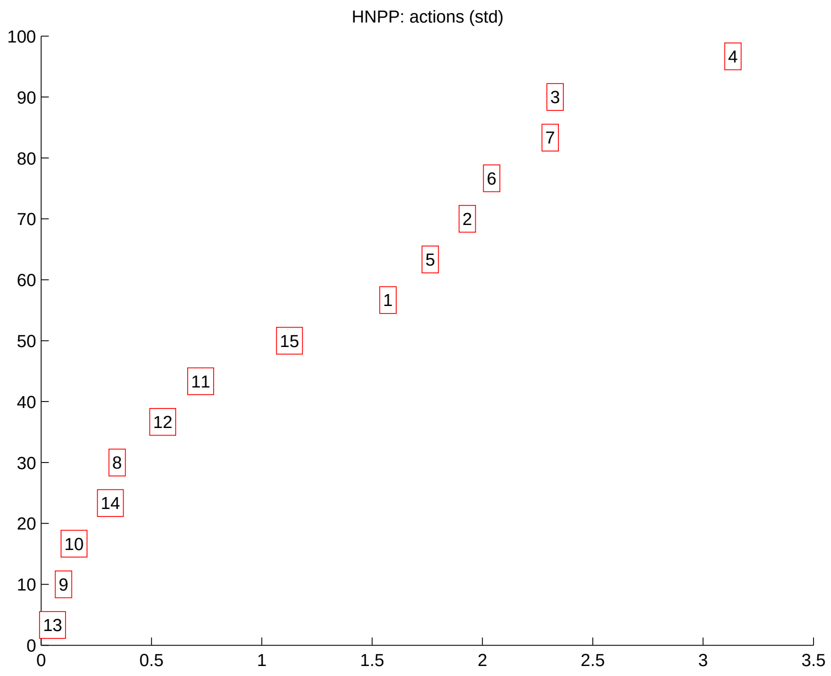

Figure C.31: Hnpp: std action of differences.

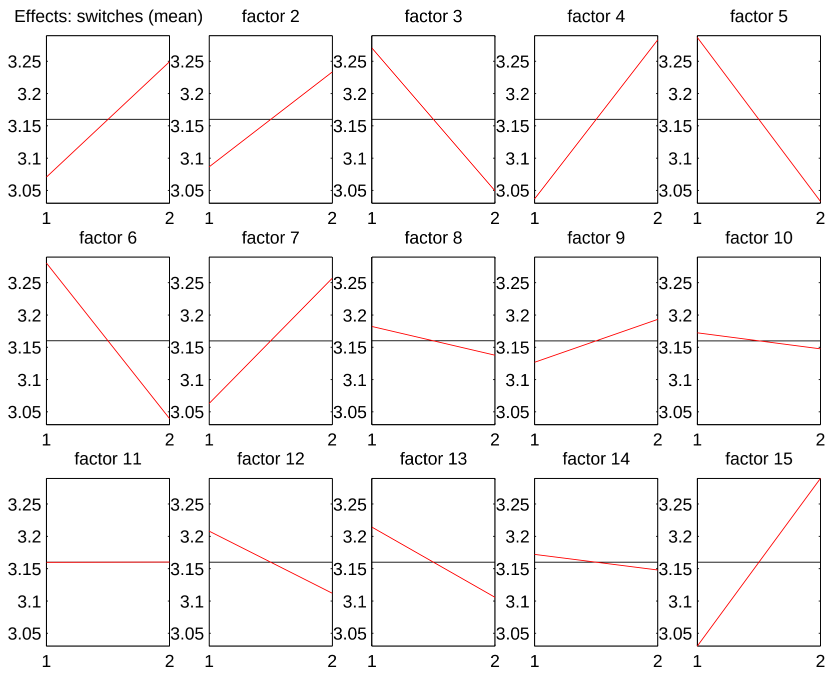

Figure C.32: Effects: mean number of switches.

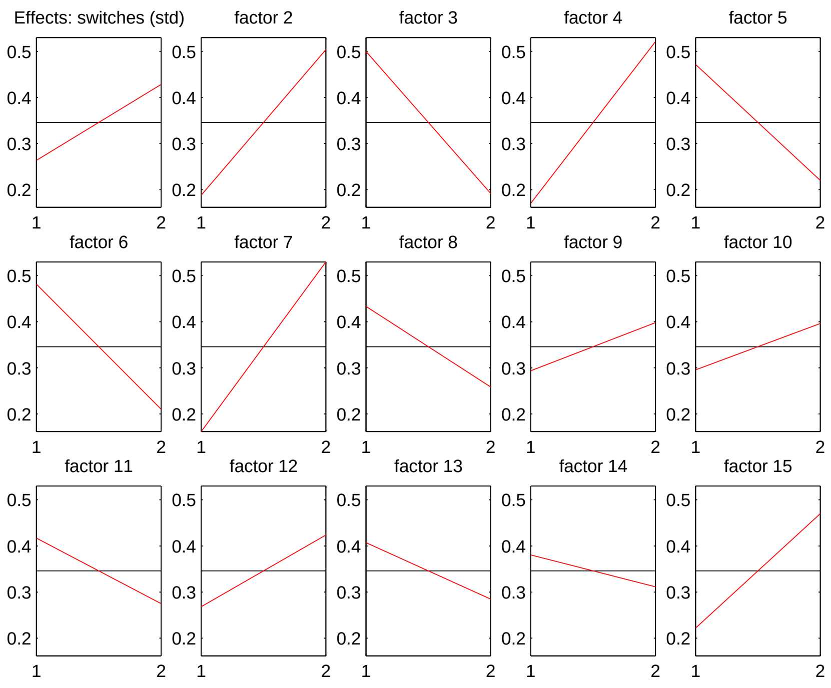

Figure C.33: Effects: std switches of differences.

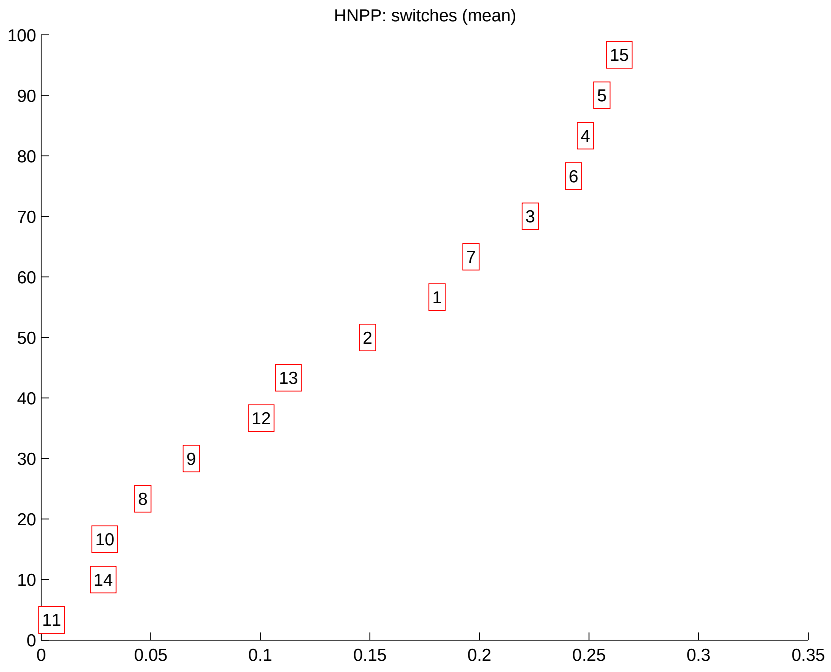

Figure C.34: Hnpp: mean number of switches.

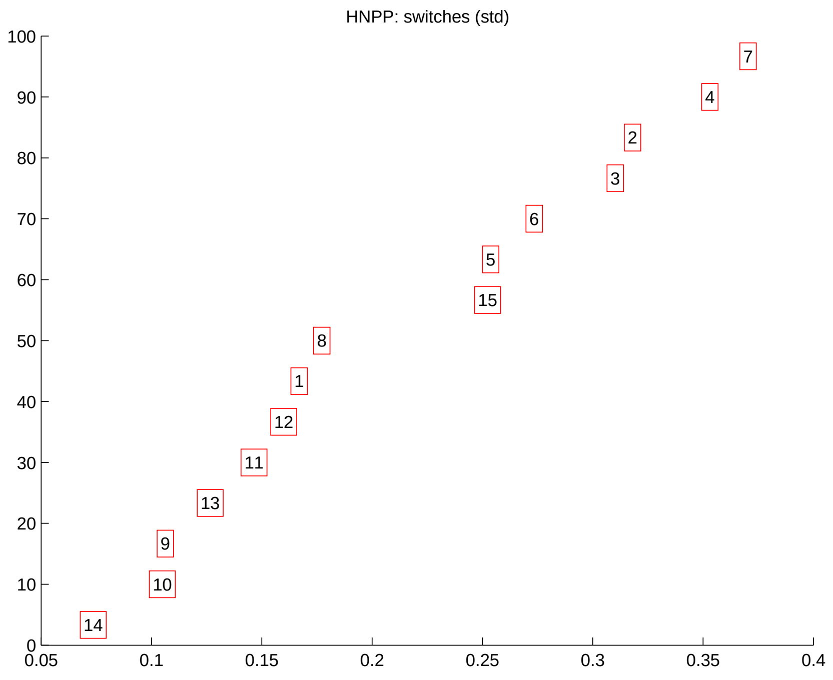

Figure C.35: Hnpp: std switches.

C.4 Optimal settings

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
6.85	5.46	0.98	8.43	8.77	8.96	3.82	10.25

Table C.14: Cumulative rewards for derived optimal settings results.

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
18.05	19.25	19.69	18.00	17.43	17.70	18.34	16.69

Table C.15: Number of actions for derived optimal settings results.

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
18.05	19.25	19.69	18.00	17.43	17.70	18.34	16.69

Table C.16: Number of actions for derived optimal settings results.

C.5 No shaping

Note that trial 10 actually gets the lowest rewards without shaping (whereas trial 10 performed best in the other experiments). We have no explanation.

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
3.26	3.29	3.79	3.07	3.10	3.04	3.61	3.01

Table C.17: Number of switches for derived optimal settings results.

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8
15000	19500	17000	17000	16000	16500	16500	17500

Table C.18: Number of episodes for derived optimal settings results.

trial	run 1	run 2	run 3
1	19.83	21.61	21.84
2	21.77	19.70	19.11
3	17.87	44.240	23.71
4	20.77	22.74	26.56
5	19.51	20.99	21.40
6	20.44	20.75	19.05
7	27.12	20.70	21.68
8	21.76	22.54	21.01
9	21.15	18.96	20.71
10	19.31	19.32	20.08
11	55.61	36.48	21.14
12	34.93	21.50	22.21
13	18.97	20.29	19.47
14	18.89	18.64	20.47
15	44.85	19.52	22.18
16	28.38	22.08	20.87

Table C.19: Number of actions without shaping.

trial	run 1	run 2	run 3
1	3.93	4.10	4.02
2	4.41	3.67	3.53
3	3.57	7.33	3.98
4	3.96	3.83	4.65
5	3.75	4.27	4.07
6	4.36	3.97	3.70
7	4.92	4.20	3.70
8	3.80	3.91	3.96
9	4.43	3.48	4.20
10	3.91	4.08	4.32
11	7.77	5.39	4.27
12	4.59	3.60	3.82
13	3.22	4.00	3.97
14	3.41	3.79	3.77
15	6.10	3.97	3.65
16	4.47	4.12	3.62

Table C.20: Number of switches without shaping.

trial	run 1	run 2	run 3
1	-0.37	-3.83	-3.29
2	-6.57	1.95	3.67
3	4.56	-60.72	-4.55
4	-1.88	-2.86	-14.67
5	1.49	-4.33	-3.22
6	-4.40	-1.64	2.31
7	-18.64	-4.39	-0.35
8	-1.46	-3.09	-1.80
9	-5.76	4.14	-3.39
10	0.37	-1.10	-3.76
11	-86.34	-38.22	-5.16
12	-27.73	0.46	-1.89
13	6.26	-1.41	-0.27
14	4.81	2.02	0.35
15	-54.96	-0.33	-0.41
16	-16.24	-4.23	1.10

Table C.21: Cumulative reward without shaping.

trial	run 1	run 2	run 3
1	16434	15115	16010
2	8335	10504	10550
3	19910	19515	29946
4	16476	16180	20330
5	18396	12526	15874
6	13227	11449	11086
7	27331	26427	27452
8	16265	13263	18752
9	17184	12848	13993
10	11017	12231	11205
11	14420	19739	28112
12	11428	11146	17349
13	16863	14278	16842
14	8833	11165	7094
15	22676	24189	17443
16	22025	10966	13465

Table C.22: Number of episodes in the training phase without shaping.

Bibliography

[1] Jiju Antony and Frenie Jiju Antony. Teaching the taguchi method to industrial engineers. *Work Study*, 50:141–149, 2001.

[2] Bram Bakker, Viktor Zhumatiy, Gabriel Gruener, and Jürgen Schmidhuber. A robot that reinforcement-learns to identify and memorize important previous observations. Technical report, CSEM Microrobotics, 2002.

[3] Mike Bax. Investigations into novel actuators. Master's thesis, Hogeschool Utrecht, 2003.

[4] Dennis J.H. Bruijnen. Structural and parametric optimisation of (non)linear dynamical systems using genetic programming. Master's thesis, Eindhoven University of Technology, August 2003.

[5] Melanie Coggan. Exploration and exploitation in reinforcement learning. Technical report, McGill University, 2004.

[6] The Shadow Robot Company. Shadow air muscle 30 mm. Datasheet.

[7] Frank Daerden and Dirk Lefeber. Pneumatic artificial muscles: actuators for robotics and automation. Technical report, Vrije Universiteit Brussel, Department of Mechanical Engineering, 2000.

[8] Howard Demuth, Mark Beale, and Martin Hagan. *Neural Network Toolbox. For Use with* MATLAB, 4 edition.

[9] Festo. Festo homepage. http://www.festo.com.

[10] Faustino J. Gomez. *Robust Non-Linear Control through Neuroevolution.* PhD thesis, University of Texas at Austin, August 2003.

[11] Simon Haykin. *Neural Networks. A comprehensive foundation.* Prentice Hall International, Inc., 2 edition, 1999.

[12] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[13] A.A.J. Marley. *International Encyclopedia of the Social & Behavioral Sciences.* Pergamon Press, 2000.

[14] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers.* John Wiley & Sons, Inc., 605 Third Avenue, Net York, USA, 2 edition, 1999.

[15] Jun Morimoto and Kenji Doya. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*, 36:37–51, 2001.

[16] R.T. Pack, M. Iskarous, and K. Kawamura. Comparison of fuzzy and nonlinear control techniques for a flexible rubbertuator-based robot joint.

[17] Alex Paikin. Hitequest, July 2001. http://www.hitequest.com/Kiss/usb.htm.

[18] Theodore J. Perkins and Andrew G. Barto. Lyapunov design for safe reinforcement learning. *Robotics and Autonomous Systems*, 36:37–51, 2001.

[19] Enno Peters, Dennis Bos, and Thom Warmerdam. Ai-techniques in (home-) robotics. a theoretical and practical study. Master's Thesis not completed.

[20] PowerDesigners. Pulse width modulation (pwm) basics. http://www.powerdesigners.com/InfoWeb/design_center/articles/PWM/pwm.shtm.

[21] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms, 1998.

[22] Numerical Recipes Software. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[23] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[24] Genichi Taguchi. *Introduction to Quality Engineering.* Nordica International Limited, Minato-ku, Tokyo 107, Japan, 6 edition, 1986.

[25] H.M.M. ten Eikelder and E.H.L. Aarts. Neural networks. Lecture notes, November 2002.

[26] Gerald Tesauro. Td-gammon, a self teaching backgammon program, achieves master-level play. In *Neural Computing*, volume 6, pages 215–219. Publisher MIT Press, 1994.

[27] Cor van de Klooster. Electrical schema for robot arm.

[28] Patrick van der Smagt. *Visual Robot Arm Guidance using Neural Networks.* PhD thesis, University of Amsterdam, 1995.

[29] Marco Wiering. *Explorations in Efficient Reinforcement Learning.* PhD thesis, University of Amsterdam, February 1999.

[30] Marco Wiering. Convergence and divergence in standard and averaging reinforcement learning. In *Proceedings of the 15th European Conference on Machine Learning (ECML'04)*, pages 477–488. Springer-Verlag Berlin Heidelberg, 2004.

[31] Wolfram. Power spectral density function. http://documents.wolfram.com/applications/timeseries/UsersGuidetoTimeSeries/1.8.1.html.

[32] Clemens C. Wüst, Liesbeth Steffens, Reinder J. Bril, and Wim F.J. Verhaegh. Qos control strategies for high-quality video processing. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, 2004.

[33] Alper Yaman. Load independent trajectory control for an artificial muscle. Master's thesis, Bogazici University, 2000.