NTNU
Innovation and Creativity

Simulations of imitative learning

Firas Risnes Barakat

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Pinar Öztürk, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Continuation of the work done in the depth study titled "Learning by Imitation". A simulator will be chosen and similar experiments to the one done in the depth study will be simulated by using this simulator and MATLAB. The first step of the Master thesis is to create an interface between the selected simulator and MATLAB. Then sensorimotor control architectures realising imitation will be designed, implemented, and analyzed using the simulator and MATLAB.

Assignment given: 20. January 2006
Supervisor: Pinar Öztürk, IDI

Abstract

This Master thesis presents simulations within the field of imitative learning. The thesis starts with a review of the work done in my depth study [2], looking at imitative learning in general. Further, forward and inverse models are studied, and a case study of a Wolpert *et al* article [29] is done. An architecture using the recurrent neural network with parametric bias (RNNPB) and a PID-controller by Tani *et al* [21] is presented, and later simulated using MATLAB and the breve simulation environment. It is tested if the RNNPB is suitable for imitative learning. The first experiment was quite successful, and interesting results were discovered. The second experiment was less successful. Generally, it was confirmed that RNNPB is able to reproduce actions, interact with the environment, and indicate situations using the parametric bias (PB). It was also observed that the PB values tend to reflect common characteristics in similar training patterns. A comparison between the forward and inverse model and the RNNPB model was done. The former appears to be more modular and a predictor of consequence of actions, while the latter predicts sequences and is able to represent the situation it is in. The work done to connect MATLAB and breve is also presented.

Preface

This report is the result of the master thesis, which is the final project (TDT4900) for the Master degree in Computer Science at the Norwegian University of Science and Technology (NTNU). The thesis is titled *Simulations of Imitative Learning*. The defined goal for the project goes as follows:

> *Continuation of the work done in the depth study titled "Learning by Imitation". A simulator will be chosen and similar experiments to the one done in the depth study will be simulated by using this simulator and MATLAB. The first step of the Master thesis is to create an interface between the selected simulator and MATLAB. Then sensorimotor control architectures realizing imitation will be designed, implemented, and analyzed using the simulator and MATLAB.*

breve Simulation Environment was selected as the mentioned robot simulator.

During the work with connecting MATLAB and breve, I cooperated with Axel Tidemann, a PHD student working on the same topic. Ole-Marius Moe-Helgesen, a Master student working with neural networks, assisted us in the beginning

Finally, I wish to thank my supervisor, Pinar Östürk, for helping me select this project, as well as giving valuable input and feedback. I would also like to thank Axel Tideman for the cooperation and discussions in the technical work with breve and MATLAB.

Trondheim, June 22, 2006

Firas Risnes Barakat

Contents

1 Introduction 9
1.1 The Goal of this Project 10
1.2 Imitative Learning 10
1.3 Summary 12

2 Forward and Inverse Models 13
2.1 Introduction 13
2.2 Architecture 13
2.3 Case Study: Wolpert and Kawato 17
2.3.1 Architecture 17
2.3.2 Discussion 20
2.3.3 Conlcusion 21

3 RNNPB and PID-controller 22
3.1 Introduction 22
3.2 Architecture 22
3.2.1 RNNPB 23
3.2.2 PID-controller 27
3.3 Summary 28

4 MATLAB and breve 29
4.1 Mathworks MATLAB 29
4.1.1 What is MATLAB? 29
4.1.2 Plugin Support 30
4.1.3 Communication Capabilities 30
4.2 breve Simulation Environment 31
4.2.1 What is breve? 31
4.2.2 Plugin Support 31
4.2.3 Communication Capabilities 32
4.3 Connecting MATLAB and breve 32
4.3.1 breve as a Plugin in MATLAB 32

4.3.2 Communicating through Sockets 33
4.3.3 Direct Communication 33
4.3.4 Our solution - Modified Direct Communication 34
4.4 Interaction between MATLAB and breve 34

5 Simulation 36
5.1 Simulation Setup 36
5.1.1 Simple Pseudo Code 39
5.2 The Experiments 42
5.2.1 Simulation 1: Separate Arm Movement 42
5.2.2 Simulation 2: Capturing Movement 47
5.2.3 Other Simulations 52
5.3 Summary and Discussion 53

6 Comparison between the Forward and Inverse Models and the RNNPB architecture 55
6.1 Training 55
6.2 Architectural Comparison 56
6.3 Summary and Discussion 57

7 Program Documentation 59
7.1 File Description 59
7.1.1 MATLAB Files 59
7.1.2 Java Files 61
7.1.3 breve Files 62
7.2 Diagrams 62

8 Conclusion 64

A Terminology 66

B Running the Simulations 68
B.1 Readying breve and MATLAB 68
B.2 Using Trained Weights 69
B.3 Recording Actions 69
B.4 Training the RNNPB 70

C Videos 71
C.1 Simulation 1 - Training 71
C.2 Simulation 1 - Interaction 71
C.3 Simulation 2 - Training 72
C.4 Simulation 2 - Interaction 72

D Source Code **73**
D.1 MATLAB Files 73
D.1.1 General Files 73
D.1.2 Training Phase 75
D.1.3 Interaction Phase 87
D.2 Java Files 91
D.3 breve Files 93

List of Figures

2.1 Simple forward and inverse model 15
2.2 Hierarchical forward and inverse model 16
2.3 The Wolpert and Kawato Architecture 19

3.1 The RNNPB Training Phase 23
3.2 The RNNPB Interaction Phase 24
3.3 Simple recurrent neural network 24
3.4 RNNPB model 25

4.1 The interaction between MATLAB and breve 35

5.1 The Simulated Robot 37
5.2 The Sensors and Field of Vision 38
5.3 RNNPB and PID-controller training phase, with MATLAB and breve coloring 40
5.4 RNNPB and PID-controller interaction phase, with MATLAB and breve coloring 41
5.5 Robot in the first simulation 43
5.6 PB plot of the first simulation 46
5.7 Robot in the second simulation 48
5.8 PB plots of the second simulation 51

7.1 File Relationship During Training 62
7.2 File Relationship During Interaction 63

Chapter 1

Introduction

This project is a continuation of the work done in my depth study, *Learning by Imitation* [2]. In the depth study imitative learning was studied in detail; in particularly the uses of imitative learning. A small simulation using a neural network with parametric bias (RNNPB) [14, 21, 15, 27] was also done to investigate if the RNNPB was suitable for further simulations. A conclusion was made that even though the RNNPB is difficult to train, it can be used in the Master thesis.

This chapter begins with an explanation the goals of this project, before a quick summary is made of what imitative learning is. In the next chapter the *forward and inverse* model is studied; both a view of the architecture and study of a paper by Wolpert *et al* [29]. Chapter 3 consists of a description on how the architecture using an RNNPB and a PID controller is, mainly based on an article by Tani *et al* [21]. The following chapter describes the work done to connect MATLAB and the breve simulat

ion environment. In Chapter 5 I describe and discuss the simulations done in this project. In Chapter 6 a brief comparison between the architectures of the forward and inverse models and the RNNPB is done. Chapter 7 contains a documentation of the code written for this project. Finally, a conclusion is made in Chapter 8. In the Appendix there is a brief terminology for imitative learning, a description of videos from simulations, and the source code for the files fpr this project.

1.1 The Goal of this Project

The main aim of this project is to make simulations of imitative learning. A robot simulator, the *breve Simulation Environment* [19], is used to simulate a humanoid robot. MATLAB [22] is used for the implementation of the RNNPB architecture. One of the goals is to connect MATLAB and breve together. The forward and inverse model and the RNNPB are two alternative ways of emulating human and monkey imitative systems. In this project, the RNNPB is used for simulations. In addition, the forward and inverse models are studied, so that a comparison between the two architectures can be made.

There are some specific goals for the simulations in this project. In my depth study, it was concluded that the RNNPB holds a great potential for use in imitative learning. The main hypothesis for the simulations is that the RNNPB can indeed be used for imitative learning. In order for this to hold, a number of conditions must be met.

First, the robot must be able to learn and reproduce different sequences of actions. This was proved in a small experiment in my depth study, and should be confirmed further. Second, sensory input should be able to affect action selection. In other words, interaction with the outside world must be possible. The robot must be able to act upon a sensory state. Third, the robot should also be able to see the sensory input in sequential context. It should be able to act upon a *sequence* of sensor states.

There are also some results which seem to be recurring in Tani's articles [14, 21, 15, 27]. His results show that after training the RNNPB, there is one parametric bias for each pattern. This means that the parametric bias can be used to indicate what situation the robot is in. How this works exactly will be described in Chapter 3. It is also discovered that after training similar patterns, common characteristics emerge in the PB vectors. During my own simulations, these claims are studied further.

1.2 Imitative Learning

What is imitative learning? Imitative learning within artificial intelligence is a learning method based on how humans and some monkeys learn. Instead of programming the robot what to do or to use reinforcement learning to punish and reward the robot, we simply *show* it how to act. In my depth study [2], it was discussed how this could be done in several ways, and in which areas imitative learning can be used.

The simplest example was direct demonstration by manually remote controlling the robot while letting the robot record the movement. After the demonstration, the robot would be able to replicate the movement. An industrial manipulator arm was used as an example. However, this form of demonstration is only sufficient to teach primitive movements, or just movements with no goal. In order for true imitation to occur, we need the robot to not only replicate the movement, but also replicate the *action*. For example, if we show how the robot how to move a cup, we want it to still be able to move it even if the cup is at a different position. We want the robot to "understand" that the action is moving the cup, and not simply moving the arm forward and grasping.

We investigated how this "understanding of goals" could achieved using a hierarchical approach [3, 25, 16]. Primitive actions could be built to achieve simple actions; simple actions could be built to achieve complex actions; complex actions could be built to achieve advanced behaviors; and so on. For example, a primitive actions might be `move forward`, while a more complex action could be `give a handshake`. We also looked at how imitative learning might be used to teach navigation through a maze [8].

Another topic that was discussed, was how imitative learning could be connected to linguistics. We saw how imitative learning might be the route to solve, or rather *avoid*, the symbol grounding problem [1] [11]. At a certain level, imitation requires the ability to predict or "imagine" events. If an event is somehow connected to interaction with an object, this interaction might be internally represented by the robot's A.I. and connected to a word.

Mirror Neurons An important topic within imitative learning is the *mirror neurons* [4, 20]. These are recently discovered neurons in both monkey and human brains. The mirror neurons is very similar to the motor neurons. However, instead of just firing when the monkey or human acts, like the motor neurons do, the mirror neurons also appear to fire when action is being observed. What is even more remarkable is that the mirror neurons also appear to fire when action is being *imagined*. Research shows that autism might be caused by defective mirror neurons, which might mean that they might also connected to empathetic

[1]The symbol grounding problem was presented by Harnad in 1990 [27]. It is the problem of giving internal meaning to a symbol. For example, when a robot has a symbol for "cup", it should have some kind of internal representation of what a cup *is*. The problem is that in order to explain what a cup is, other "ungrounded" symbols have to be used. Harnad compares it to trying to learn Chinese only by using a Chinese-Chinese dictionary.

abilities [20].

Because of the higher cognition properties of the mirror neurons, it is no surprise that scientists want to emulate them in artificial intelligence The *forward and inverse model* and the recurrent neural network with parametric bias (RNNPB), both of which will be discussed further in the coming chapters, are attempts to model the mirror neurons. So far, the different studies by Demiris *et al* [16, 9, 10], Wolpert *et al* [29], and Tani *et al* [14, 27, 15] have been quite encouraging.

1.3 Summary

The goal of this project is mainly to study and compare the two most important architectures on the route to imitative learning, the *recurrent neural network with parametric bias (RNNPB)* and *forward and inverse models*. As with my depth study, the RNNPB will be studied closer by doing simulations with it. MATLAB and the breve simulation environment will be used for this.

In this chapter, we also reviewed the most important notes from my depth study: imitative learning in general and the mirror neurons. Imitative learning holds a great promise for artificial intelligence. If successful, many problems can be solved by imitative learning. In addition, emulation of the mirror neurons might be a more true emulation of the human intelligence. This makes the ability to imitate a sign of higher cognition.

Chapter 2

Forward and Inverse Models

2.1 Introduction

In my depth study [2], forward and inverse models were not discussed. The primary focus was on Tani's work and the recurrent neural network with parametric bias [14, 27, 15]. However, Demiris *et al* [16, 10] and Wolpert *et al* [29] have done work with multiple paired forward and inverse models for motor control. This architecture can be seen as a mirror system [10], which makes them well suited for imitative learning. The model may both be implemented using symbolic AI, like Demiris [16, 10] has done, or by using sub symbolic methods, like Wolpert [29].

I will begin this chapter by explaining how the inverse and forward models works, before moving on to how they can be connected to emulate a motor system. Finally, I will present an article written by Wolpert and Kawato [29].

2.2 Architecture

Inverse Model An inverse model takes as input the current state of the system and the desired goal state. As output, the model gives the commands necessary to achieve the desired state.

There are many ways to implement such a model. The easiest one though, is using a PID-controller. As described in Section 3.2.2, a PID-controller can be implemented as an equation that receives a current state and a desired state, and outputs the strength that needs to be applied. For example, the current and desired states may be temperatures in an oven, while the output is the heating power. An other implementation of an inverse model can be a neural network using error learning propagation, as will be described in Section 2.3.

Forward Model A forward model takes as input the current state of the system and the command acting on it. It then outputs a prediction of the next state. The most sensible sub symbolic implementation of this would be a recurrent neural network (RNN) which learns to predict temporal sequences.

Relationship Next, we will look at how those two models can be connected to each other for motor control. We assume that we have fully trained or programmed forward and inverse models, meaning that we only describe the interaction. The main idea is that we need a forward model to make predictions of what would happen if certain commands was applied at certain states. In other words, the model would display a behavior. For each behavior, we need an inverse model which can find out which command should be performed to go from one state to another [29].

When the inverse model receives the current and desired states, it sends a motor command to both the robot and to the forward model. The forward model then outputs the next predicted state and sends it back to the inverse model. The robot executes the received command and sends its new current state to the inverse model.

Figure 2.1 shows a model of the relationship between the forward and inverse models, and the robot. As we will discover after seeing the model in the case study in Section 2.3, the model in Figure 2.1 is a quite simple model that is sufficient for only one behavior. If we need more than one behavior, we need one such model for each behavior. And as we will see, they can be connected so that the robot can switch between them.

The “behavior” and the states that we speak of can be on different levels. For example, the behavior can be a primitive action, like moving an arm forward. In this case each state would then be the position of the arm in each actual time step. If the behavior is on a higher level which consists of several primitive actions, each time step would have to be larger than an actual time step. The state could then be the position after each primitive action has been performed.

To illustrate, we can look at an example of a higher level action, like lifting a cup. Let us say that this behavior consists of five primitive actions for the

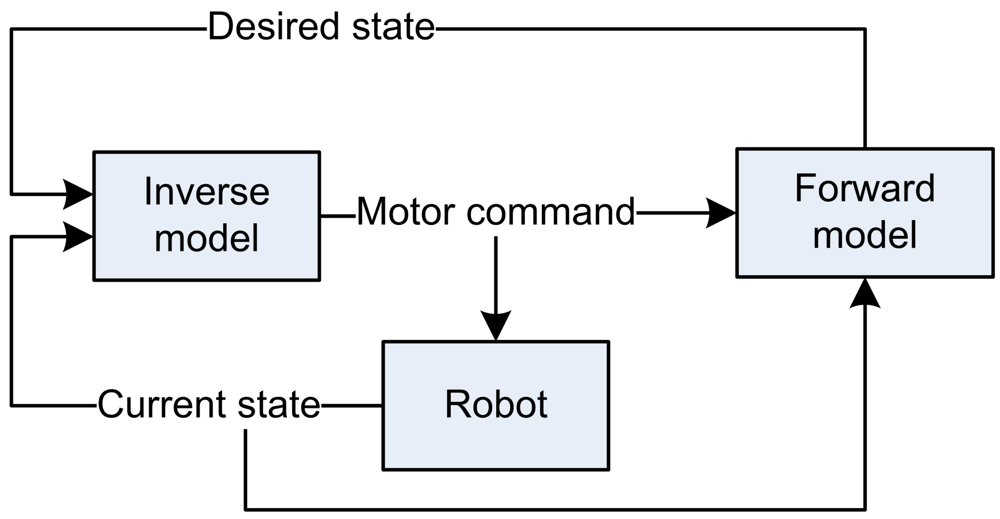

Figure 2.1: Simple forward and inverse model based on [16, 10, 29]. This simple model shows how the current state and desired state is received by the inverse model, which outputs the command that makes the change. The command is both sent to the robot and to the forward model. The forward also receives the current state, and then predicts the next desired state and sends it to the inverse model.

arm: `raise`, `lower`, `move forward`, `move back`, and `grasp`. The order of execution could be: `move forward` - `lower` - `grasp` - `raise` - `move back`. The first current state is then the start position while the first desired state is the position achieved after moving the arm forward. These two states is fed to the inverse model. The inverse model then outputs `move forward` as the solution to achieving the desired state. This output is fed to the robot which actually performs the action, and to the forward model which outputs the next desired state. In this case, the next desired state would be the one after lowering the arm. The output is fed back to the inverse model, which will then output `lower`. This looped process will continue until the whole action has been performed.

Discussion In this example primitive actions were used to achieve desired states. To apply the model on one single primitive action, for example moving the arm forward, setting the velocity in a direction could be an action to achieve a desired state. What becomes obvious is that we can build a hierarchy of forward and inverse models. At the lowest level we would have the primitive actions, which could be built to achieve higher level actions [16]. Figure 2.2 shows how the model might look like.

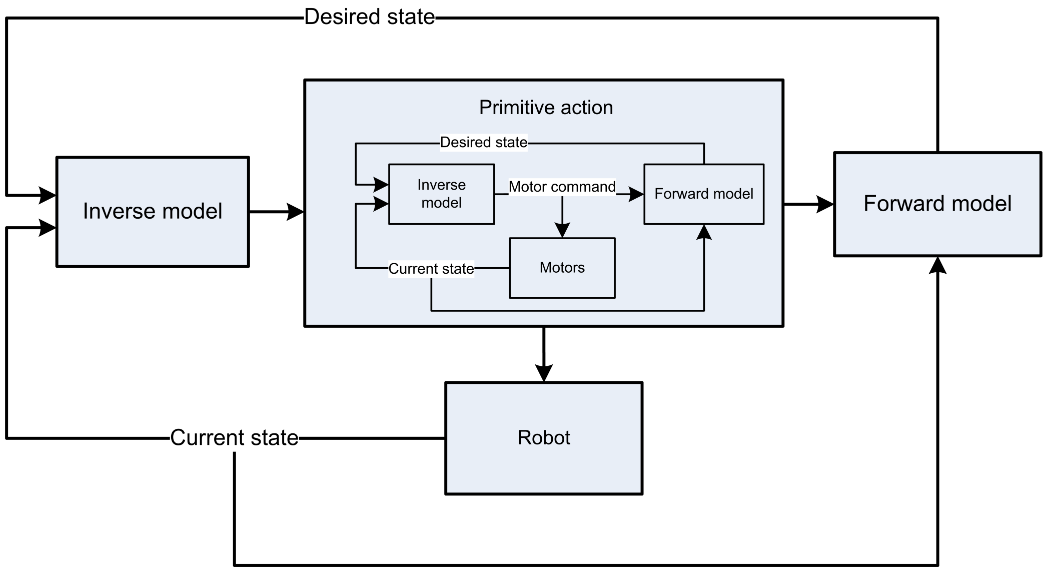

Figure 2.2: Hierarchical forward and inverse model (based on descriptions in [16, 10]. As we can see, the execution of a primitive action can be done at a lower level forward and inverse model.

The main focus of the forward and inverse model is the output of the inverse model. This output is the action that must be taken in order to achieve a desired state. It is fed both to the forward model and to the robot. As the model on Figure 2.2 shows, this output might as well be lower level actions. This opens possibilities to build more complex systems with several levels. For example, the lowest level could be, as we discussed, `move forward`. The next level could be `pick up cup`, which again could be a primitive action for a larger goal, like `fill cup with tea`. At the top level, the action might be a response to the command "`bring me some tea`".

It is easy to see that this hierarchal approach makes sense. And it is from here that we can look at *learning to imitate* and *learning by imitation*. As was discussed in my depth study, when a robot learns new actions by observing a teacher, we want to the robot to understand the *intentions* of the teacher. Using the "tea-example": when we show a robot how to bring tea we want it to understand that bringing tea is a set of actions that is again divided into primitive actions, rather than seeing the whole action of bringing tea as one single atomic action.

2.3 Case Study: Wolpert and Kawato

In this section, a brief review of Wolpert's and Kawatos's article, *Multiple paired forward and inverse models for motor control* [29], is done. The aim of this review is to see a study of the forward and inverse model, as well as providing something which can be used for comparison with the RNNPB and PID-controller model in Chapter 6.

Wolpert and Kawato present an architecture built of multiple paired forward and inverse models. Their goal is to look at a system which is capable at controlling the motors of a robot learning and interacting using both external and internal information. They specifically look at learning multiple actions, and they use a human's ability to interact with a great variety of different environments as an example. The architecture they look at is modular; it uses one module for each action. This, however, creates a module selection problem which must be solved.

The model is designed so that it is able to predict (or "imagine") actions before executing them. It can use these predictions in order to select the correct action that leads to its desired goal. Wolpert and Kawato are loosely modeling the human motor system. Though it is not specifically mentioned in the article, the proposed model does, to some extent, emulate the mirror neurons [4, 20] discussed in Section 1.2, as it is able to make predictions without necessarily executing the actions.

2.3.1 Architecture

The architecture of this model uses paired forward and inverse models, similar to the one in Figure 2.1, as basis. One such paired model is used for each action. In addition, for each paired forward and inverse model, a *responsibility predictor* is added to help each pair decide "how correct" it is. All the paired models are connected to a *responsibility estimator*, which makes sure that the most correct model is weighted most, so that the correct action is executed. Figure 2.3 shows the architecture.

The Components

As mentioned, the architecture consists of several components. We will now take a small look at each component.

Forward Model This model can, as described before, receive a current state and a motor command, and return a predicted next state. Wolpert and Kawato mentions simple supervised learning for this model. A normal backpropagtion neural network or a recurrent neural network would work.

Inverse Model This model also works as previously discussed. It is a component that receives a current state and a desired state, and returns the motor command necessary for the change. Though they do not go into detail, Wolpert and Kawato mention three different methods to adapt an inverse model: direct inverse modeling [23], distal supervised learning [17], and feedback-error-learning [18].

One pair of the two described models is able to learn one single action. Therefore the architecture needs multiple such pairs to be able to perform different actions. A switching mechanism is needed so that the system might select the appropriate action for different situations. Another component is added to each pair:

Responsibility Predictor This component receives sensor information as input. It uses the input for comparison with an internal responsibility model, and decides how relevant the sensory input is to the current "responsibility". It outputs an error that tells the difference between the internal model and the actual sensor input. This model can be a simple backpropgation network that learns to output how much error there is in a given situation.

As we begin to see, when this component is added to the paired forward and inverse models, each "module" will be able to perform a single action as well as deciding how correct this action is in the current context. However, yet one more component is needed; a single component that connects all the multiple modules:

Responsibility Estimator This component receives the error signal from each module's responsibility predictor. It uses the error to decide which module should be selected, and sends "weights" to each module's forward and inverse models. These weights is multiplied to each module's motor output. Thus, the responsibility estimator selects which action is to be executed based on the error it receives from the responsibility predictor.

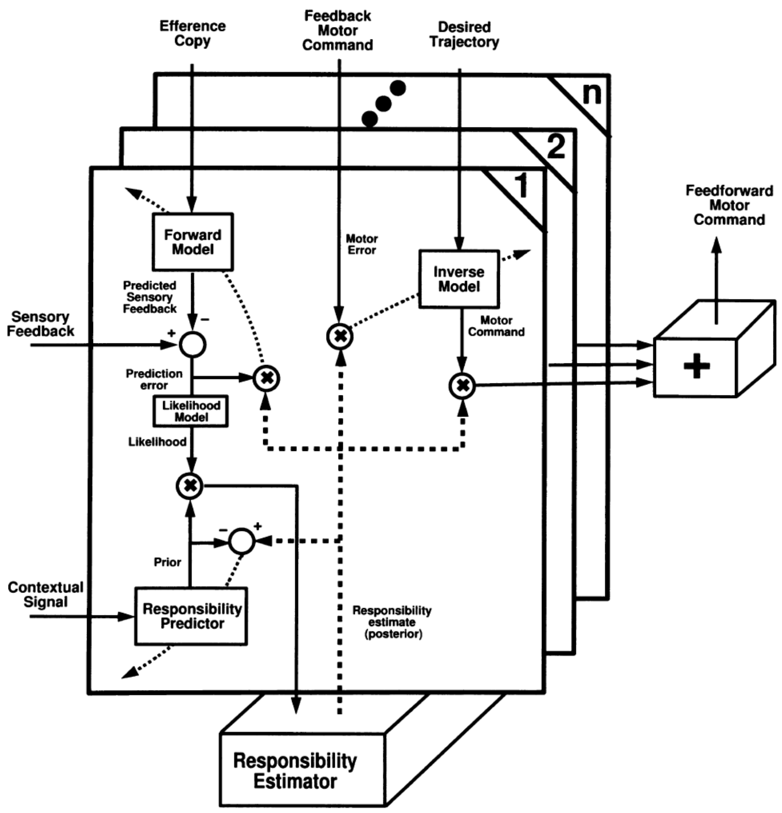

Figure 2.3: A single module within the multiple paired internal model. The thick dashed line shows the central role of the responsibility estimators signals. Dotted lines passing through models are training signals for learning. The exponential transform of the errors has been replaced by a more general likelihood model. [29]

Modularity

We now have a system which is able to learn and perform actions based on the situation it is in. An important feature of this architecture is *modularity*. First, each model (forward model, inverse model, and responsibility predictor) can be trained or programmed separately. The only important thing is that each model function as it should, be it a neural network, PID-controller, or any other traditional or sub symbolic implementation.

Second, each paired model is a module that can easily be added or removed. After adding a new module, only the responsibility estimator (and the new module) needs to be trained, without having to retrain the whole system. This particular modular feature has several benefits. Wolpert and Kawato mentions the advantage of adding different modules along the way in order for the system to be able to handle new situations. Another benefit is that motor behaviors can be trained separately without affecting each other. This makes training of each behavior more clean and efficient. In addition, many new situations may be combinations of previously learned situations. An architecture with the ability to switch between different behaviors will have a enormous repertoire of combinations.

2.3.2 Discussion

The study Wolpert and Kawato makes is quite optimistic. Though they do not produce any test results, the article is quite detailed in how their architecture should be implemented. Also, a number of untested statements is made about the architecture. For example, it is talked about how the behavior switching is controlled by the prediction error rather than the performance error. This means that behavior is selected based on how well the current behavior fits with the robot's predictions, rather being selected based on how well the behavior solves a task. This is interesting, as it shows that the robot's "imagination" is of importance.

Another interesting note is the possibilities of mixing modules. As we remember, the responsibility estimator assigns weights to the different module outputs. This means that more than one module may be active at once. Wolpert and Kawato have a theory that if there are two modules for two different situations, the two modules might be able to handle a linear combination of the two situations.

It is also discussed that the model seems to be neurophysiologically plausible. In particular, it is suggested that multiple paired forward and inverse models exists in the human cerebellum. For example, it is stated that imagination of motor actions activates the cerebellum [7]. This has been demonstrated using fMRI scans [13]. They also describe how to make further studies of the cerebellum in order to prove that multiple paired forward and inverse models are plausible.

Comparison to Hierarchical Approach The architecture of Wolpert and Kawato differs somewhat from the hierarchical approach that we discussed in Section 2.2. Demiris *et al* [16, 10], with the HAMMER architectures, follows a hierarchal approach as we described. The main difference is that in the current architecture, all primitive behaviors are parallel, and selection is handled by the responsibility estimator. In a hierarchical architecture, primitive behaviors are in parallel; though the selection is being handled by another paired forward and inverse model. More correctly, the primitive behavior is an output from a higher level inverse model. Of course, the two different approaches each has their advantages. Wolpert's and Kawato's approach is obviously well suited for using primitive behaviors in dynamic environments which requires modularity. However, a hierarchical model gives better high level abstraction possibilities, and thus a seemingly higher level of cognition.

2.3.3 Conlcusion

All in all, the model proposed by Wolpert and Kawato appears to be a good emulator of the human motor/mirror system. Motor execution is based on prediction; thus giving the robot a form of "imagination". The modularity of this architecture makes it well suited for practical implementation. It can learn new behaviors without interfering with previously learned behaviors. This learning can also occur during interaction while previously learned behaviors are active. The lack of hierarchy might make it difficult to learn actions and goals with a higher abstraction level. However, this is not discussed in the article, and further study might show that the architecture can be modified and used hierarchically.

Chapter 3

RNNPB and PID-controller

3.1 Introduction

In this chapter, the recurrent neural network with parametric bias (RNNPB) [14, 27, 15] and PID-controller model is discussed. The model is based on the one in the article by Tani, Ito, Noda and Hoshino, *Dynamic and interactive generation of object handling behaviors by a small humanoid robot using a dynamic neural network model* [21]. In this article, the RNNPB is used along with a PID-controller to study a robot's interaction with an object. In the current chapter, the architecture of the model is only described. In Chapter 5 simulations using this architecture is presented.

3.2 Architecture

The "RNNPB and PID-controller"-architecture consists of two phases, a *training phase* and an *interaction phase*.

In the *training phase* (Figure 3.2), the robot's motors are controlled manually by a human teacher, while the state in each time step is recorded and stored so that it can be used later. The state of the robot can be joint angles of arms, sensor input, and any other values which the robot can read about itself. The RNNPB is then trained using the stored states of the robot. The output is the predicted next state. The prediction is compared to the actual next state which was recorded. The error from this comparison is used for adjusting the weights and PB values.

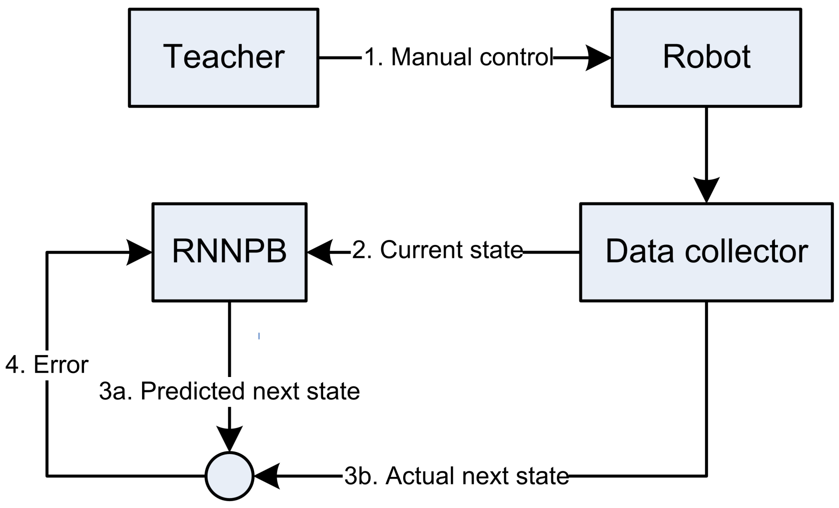

Figure 3.1: Training phase. As we can see, 1) the robot is controlled manually. After the data has been collected, off line training is done. For each iteration during training, 2) the current state is sent to the RNNPB. The RNNPB tries to 3a) predict the next sate; while 3b) the actual next state is already known. 4) The error is calculated in order to update the RNNPB weights and parametric bias.

In the *interaction phase* (Figure 3.1), the RNNPB's receives sensor input and the current state, and outputs the next state. In this case, however, the next state is not only a prediction, but it is also a *desired* state. The current state and the desired state is then sent to the PID-controller, which compares the states and applies the motor commands which are necessary to achieve the desired change. How the PID-controller does this is described in Section 3.2.2. Figures 3.1 and 3.2 show models of these phases.

3.2.1 RNNPB

The recurrent neural network with parametric bias (RNNPB) [14, 21, 15, 27] was explained in my depth study [2]. The RNNPB is an extended version of the simple recurrent neural network (SRN) [5]. The SRN is a predictive neural network, which uses context nodes to predict the next pattern in a sequence. Figure 3.3 shows a simple illustration of the SRN.

The context nodes helps the network recognize change in the context, even though the input patterns might be the same. Without the context nodes, the SRN would be a simple backpropagation network. It would not be able to

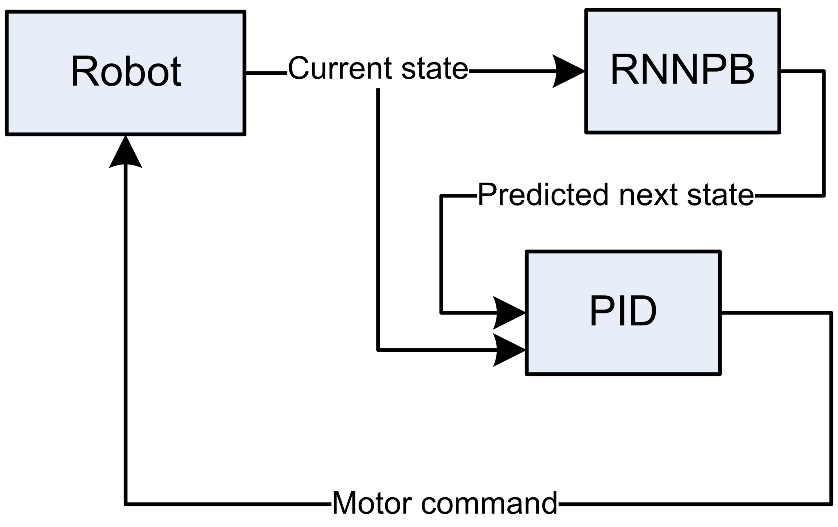

Figure 3.2: Interaction phase. The current state is sent to the RNNPB, which predicts the next state and sends it to the PID. The PID receives the current state from the robot and the next state from the RNNPB, and sends the command that needs to be executed to the robot.

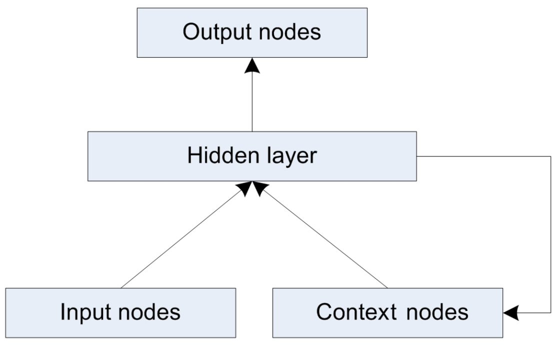

Figure 3.3: A simple recurrent network. This is a simple recurrent network during training. Each new input is obtained from a "correct" data set. During interaction, the output would have been fed back as the next input.

distinguish two identical patterns, even though they come in different *context*. In other words, the context nodes enable the network to take into account previous patterns in a sequence.

In the RNNPB, we add the parametric bias (PB) nodes as input along with the input and contexts nodes. PB nodes usually starts as a small random number, and are adjusted during training using a similar learning rule as the one for the weights. The same error that is backpropagatated to the weights is used to calculate the change in the PB nodes. The goal is that the PB values converge into one value (or vector) for each sequence. The effect is that we end up with as many different PB vectors as there is different sequences. Each PB vector will then be representing a different sequence. Figure 3.4 shows how the nodes are connected.

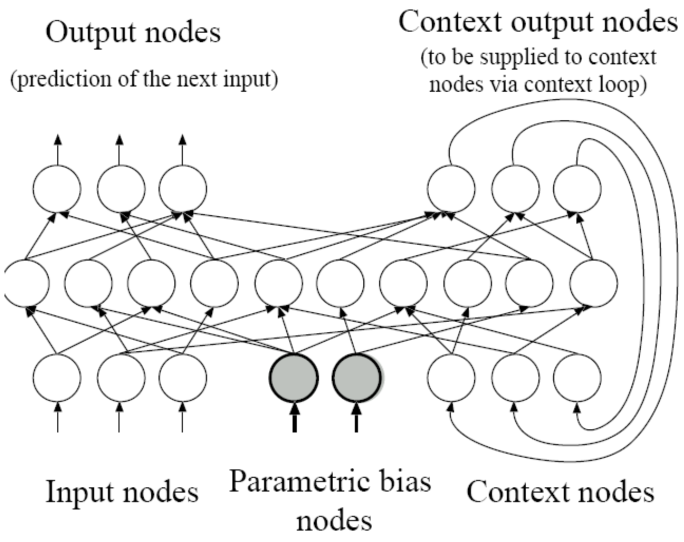
Output nodes
(prediction of the next input)
Context output nodes
(to be supplied to context nodes via context loop)
Input nodes
Parametric bias nodes
Context nodes

Figure 3.4: A model of an RNNPB during training [27]. As we can see, there is an input and an output. The output is a prediction of the next input. The context nodes is calculated and fed back for each step, while the PB nodes remains the same for each sequence. Since this model is for training, “correct” training data is fed as input, instead of feeding back the output as input, as the case is for the RNNPB during interaction.

The parametric bias can have several functions. In my depth study [2], we discussed how it can be used to connect two or more seperate neural networks together by giving corresponding pattern sequences the same PB values. If one network was a recognition network and the other was a motor execution network, we could let the recognition network recognize something and then inversely calculate its own PB values. Those PB values could then be used in the motor execution network to produce the proper motor execution.

This function is with no doubt helpful for imitative learning. As discussed in the depth study, the RNNPB can be used to connect observation and execution of primitive motor actions. To learn new actions, the robot would use the RNNPB to follow a teacher while storing the PB values as it goes along. The sequence of these PBs will represent a more complex action. Another RNNPB can then be used to connect this sequence of PBs to a verbal word describing the action. Thus, the robot learns new actions with corresponding words by imitating.

In my depth study, I did a small simulation showing how the PB values can be used to connect linguistic sequences with motor sequences. In the current report, however, I shall look at another function the PB has. In the article by Tani *et al* [21], it is stated that the PB nodes work as situated context nodes. While the "regular" context nodes indicates the sequential context, the PB nodes indicates the situation. This is due to the nature of the PB nodes. While the input and context nodes is under continuous change, the PB nodes only changes when the situation changes. This happens because during training, the PB values converge into one PB vector for each different pattern sequence (or situation).

To illustrate, I will present a simple example. Let us say that we have an RNNPB with one input node, one context node, and one PB node. If we have the input sequence $[0.1, 0.1, 0.3, 0.5...]$, the context nodes for this sequence might be $[0.5, 0.2, 0.3, 0.9...]$ after training. The PB value would have converged to a single value for the whole sequence after training, for example $[0.1, 0.1, 0.1, 0.1...]$. During interaction, the PB value will start by being faulty, but will quickly converge to the proper value. The convergence due to the use of Equations 3.1 to 3.3. The value $[0.1]$ is now indicating the situation. Should there be any small interference in the input (compared to during training), the PB value will be slow to change. The slow change helps the system remain stable in one situation even though the interaction input might have small differences compared to the training input. Should there be a complete switch to another learned sequence, the PB will change and stabilize at a different value.

During training, the following learning rule (Equations 3.1, 3.2, and 3.3) is used for updating the PB values. The same rule is used for inversely calculating the PB values during the interaction phase.

$$\Delta PB = -\sum hW \cdot (\delta H \cdot hW \cdot (1 - hW)) \qquad (3.1)$$

$$\partial PB = PB_{t-1} + \Delta PB \tag{3.2}$$

$$PB_t = \frac{1}{1 + e^{-\partial PB}} \tag{3.3}$$

hW - The weights connected from the input to the hidden layer

δH - The backpropagated error

ΔPB - The change of the PB value

∂PB - New PB value before sigmoid function

PB_{t-1} - Previous PB value

PB_t - New PB value after sigmoid function.

The sigmoid function (Equation 3.3) is used to keep the PB values between 0 and 1.

3.2.2 PID-controller

PID is a control algorithm that is often used in industrial control. It is an algorithm with various parameters which controls a variable in order to achieve a desired measurement. For example, it may be used in the cruise control of a car. The current and desired measurements are values for speed, and the controlled variable is the gas pressure. A PID-controller would give the best gas pressure in order to achieve the desired speed as fast as possible without "over shooting". When the current speed and desired speed are close to each other, the gas pressure needs to decrease so that the speed does not go beyond the desired point. Therefore, it is good to use a PID-controller in systems where delays may occur after the input is changed, before the measurement is changed.

In my experiment, the PID-controller is applied to the motors. The current and desired measurements are the states of the angles of joints, while the controlled variable (the output of the PID) is the velocity of each joint. This makes the PID-controller to work as an *inverse model*, from Chapter 2.

The parameters that are used in the formula decide the *proportional gain*, *integral gain* and *derivative gain*. There are several ways to find the near-optimal parameters for a PID controller in a given system. Neural networks, genetic algorithms [26] and tuning algorithms [30] are examples of methods that are applied to real life PID controllers. Since there already exists much literature about PID-controllers and parameter tuning [1], I will not go into too much detail on how the PID controller works or what the parameters represent.

My implementation of the PID controller is quite simple. It is a Java class which is created using four parameters. To get an output from the PID, a method is called from the PID object with the error, as well as a desired minimum and maximum for the output. The output is calculated by Equation 3.4, which is a discrete version of the PID algorithm [28].

$$output_t = output_{t-1} + P\left[(e_t - e_{t-1}) + \left(\frac{S}{I}e_t\right)\frac{D}{S}(e_t - 2e_{t-1} + e_{t-2})\right] \tag{3.4}$$

$output_t$ - Current output

$output_{t-1}$ - Output from the previous calculation

P, I, and D - Parameters for proportional, integral, and derivative gain

S - Sampling interval

e_t - Current error

e_{t-1} and e_{t-2} - Errors from the two last calculations

3.3 Summary

We have now looked at how the "RNNPB and PID-controller"-architecture works. The structure of the RNNPB was discussed, and we saw how the parametric bias can influence a recurrent neural network. The PID-controller is well known in control theory, and is only briefly described here. The architecture which we have studied appears to be suited for imitative learning. We can see that it might work with the hypotheses and conditions that we discussed in Chapter 1. This will be studied more closely during the simulation in Chapter 5, after we have studied MATLAB and breve in the next chapter.

Chapter 4

MATLAB and breve

In this chapter, I will describe the process and effort that has been put into setting up MATLAB and the breve simulation environment for the experiments. The main goal was to get MATLAB and breve to communicate with each other properly. This has been largely done by Axel Tidemann and myself, with some help from Ole-Marius Moe-Helgesen. From previous work, I was fairly familiar with MATLAB. However, breve was completely new to me, and some time was spent getting to know the environment and learning the programming language. In addition, there was no prior interface between MATLAB and breve, nor was there any clear documentation on how to set up communication between the two programs. This meant that we had to spend much time doing most of the groundwork by ourselves.

I will begin by describing MATLAB and breve, and their plugin and communication capabilities. In Section 4.3 I will describe the possibilities of connecting the two applications, and the solution we chose. Finally, in Section 4.4, the interaction between the two programs will be described more closely.

4.1 Mathworks MATLAB

4.1.1 What is MATLAB?

MATLAB is an interactive high-level programming environment that enables the user to perform mathematical computing tasks faster and easier than with a traditional programming languages [22]. It is optimized for matrix operations, which makes it useful for neural networks.

4.1.2 Plugin Support

Being an interactive programming environment, MATLAB has real good support for creating own functions. MATLAB's own scripting language for creating .m-files is easy to use and requires only basic programming knowledge in order to use it. However, it is a high-level language which cannot be used for low-level operations like communication with other programs. Fortunately, MATLAB also supports both C/C++ and Java. C/C++-files can be compiled inside MATLAB and used as regular MATLAB functions. Java objects can be used "as they are", simply by first defining the path to the files in MATLAB.

The simplicity of creating own plugins, or *functions* as they are called, enables us to easily create an interface between MATLAB and another program, provided that the other program has good plugin capabilities or a good way to communicate.

4.1.3 Communication Capabilities

We have considered several ways of communication. The simplest form of communication in MATLAB is networking. MATLAB has own features, like *urlread*, which reads data from an URL. In addition, custom Java and C functions using TCP/IP and sockets may be created.

Another way of communication might be writing and reading to a file or a memory address. The most "proper" way of making MATLAB communicate would be to recompile the other application as a plugin inside MATLAB. MATLAB would then run as usual, with the other application as a function or a plugin. However, this highly depends on the other application. As we shall see in Section 4.3, our solution is based on a Java class communicating using TCP/IP.

4.2 breve Simulation Environment

4.2.1 What is breve?

breve [19] is a free open source 3D simulation environment. It lets users create custom agents in an environment with a realistic physics engine which can simulate friction, gravity, and collision detection. breve is based on OpenGL, which makes it possible to make good visualizations of the simulation. The simulation environment is especially tailored for multi-agent systems and artificial life, which makes it suitable for biologically inspired learning.

The Structure of a breve Simulation

A breve simulation is written in a language called **steve**. The language is object oriented, and it has a different syntax from other "known" languages. In every simulation, there has to be a *controller object* which controls the entire simulation. In the breve documentation [19] the controller object is compared to the main function in C/C++. The controller initiates and starts other objects and global variables.

The objects in breve might be abstract, like in any other language, or they can be with a "physical" appearance in the breve simulation world. Like in any other language, an object can have methods and variables.

When a simulation runs, a method inside the controller superclass *Control.tz*, *iterate*, will be called with a certain time interval. This method makes the simulation move one step forward. In any object, one can create such an *iterate* method. The method will then be called each time there is an iteration in the superclass. This feature makes it very easy for a programmer to give an object a certain behavior without having to deal with threads.

The structure of a breve simulation can be studied further in Appendix D.3 or in the samples that comes with the breve installation.

4.2.2 Plugin Support

Writing plugins for breve is somewhat complicated. The plugin must be written in C, and so-called "wrapper functions" communicating with a breve class must be created. After some time testing, we decided that creating a plugin in breve was a bit too complicated. We managed to get communication working using the web-interface feature existing in breve, as will be described later.

4.2.3 Communication Capabilities

As with MATLAB, communication in breve can be done in several ways. Network communication is the simplest. breve supports networking to another breve client, as well as networking using a web-interface. The web-interface is the most relevant to our purpose. By enabling the feature in the code in breve, the simulator can receive commands from and return values to any web browser. If an application supports reading web pages, like MATLAB does, this feature can be used to interact with breve.

The problem, however, is that this feature is not suited for continuous data transfer. Each time the web-interface is used, a network socket is opened and closed. Unfortunately, the only solution to this is to use breve's plugin capabilities, and create an own custom plugin. The plugin would have to enable communication by opening and closing the network socket only once. This is possible using standard network socket communication in C.

As we shall see in the next section, we ended up using breve's own web-interface.

4.3 Connecting MATLAB and breve

As mentioned, the work done connecting MATLAB and breve was mostly done by Axel Tidemann and myself, with some input from Ole-Marius Moe-Helgesen. This was a somewhat difficult process, as our initial plan was to make a quite tight connection between the two programs. Though we ended up with a simple solution, we spent some time studying other solutions. I will begin by describing some of the solutions we looked at, before I go into detail with the solution we ended up with.

4.3.1 breve as a Plugin in MATLAB

The most "proper" solution would be to make breve work as a plugin in MATLAB. Our idea was that we would be able to start and use the breve engine from within MATLAB. There is some documentation on how to make an application frontend for breve at the breve website [19]. The way this would be done is that we would make a frontend and recompile the breve code. Once we had this frontend working, we would need to create an *application* frontend. This frondend would have to be running inside MATLAB.

The reason why we abandoned this solution was that it would have required too much work. After trying to compile the breve source code "as it is" in Linux, we discovered that the compilation required several extra packages and libraries. In addition, the documentation on how to create the custom frontends was not detailed enough, meaning we would probably have ended up spending much time figuring things out by ourselves.

4.3.2 Communicating through Sockets

One possible solution which we started implementing, but ended up abandoning, was creating one plugin for breve and one plugin for MATLAB. The idea was that the plugins could talk to each other while the two programs were running at the same time. The plugin for breve would be written in C/C++, while the plugin for MATLAB would be in either C/C++ or Java. The method of communication would be sockets.

The main problem we encountered with this method was that there was some difficulty compiling the plugin for breve. Adding libraries for socket communication made things a bit more complicated. Solving this would probably not be too difficult, though it would be more time consuming than we imagined. Another reason why we dropped this solution was that socket programming in C/C++ is platform specific. This meant that not only would we have to recompile the source for different operating systems, but we would also have to rewrite some of the code.

4.3.3 Direct Communication

The simplest solution of all would be to use the web-interface that is already integrated into breve. This feature enables any web client to enter a specific URL, and invoke methods in breve code. If the method returns a value, this value is sent back to the web client. MATLAB, in turn, has a function called *urlread*. This functions lets MATLAB read the contents of a web page. Thus, we can simply use MATLAB's *urlread* with an address to a method in breve. This solution requires both breve and MATLAB to be running at the same time.

The problem with this solution is that it is not very “elegant”. The connection is opened and closed every time communication occurs. This means that there will be some unnecessary slowdown and delay.

4.3.4 Our solution - Modified Direct Communication

The solution we ended up with was similar to the one described in the previous section. During testing, we discovered that the MATLAB function *urlread* seemed to have some extra latency. Axel Tidemann did some tests, and found out that performance could be improved by writing a small Java class that read the breve URL. Since MATLAB can run Java classes, we decided to use Axel's Java class instead of MATLAB's *urlread*.

Axel's Java class is quite simple. As input, it takes two strings and an integer: the host where breve is located, the method call, and the port number. As output, it returns the response from the server. In breve, the only thing that has to be done is enabling networking, and defining the breve controller as a network server with a port number. All the methods inside the breve controller can then be invoked using a web-interface, for example Axel's Java class.

4.4 Interaction between MATLAB and breve

Now that we are able to send method calls to breve from MATLAB, there are several ways how the two programs can interact with each other.

The most simple way would be to let breve run as usual while MATLAB sends messages to breve when only when it needs data or needs to give command. This is the most realistic way. breve would act as the “real world” with a robot, while MATLAB would be the computer which reads and writes to the real world. However, there is one problem with this method. The speed of a breve simulation is not constant on all computers. It highly depends on the system specifications, and especially the processing power of the computer. This means that if we, for example, set MATLAB to read motor positions every 0.5 seconds, we would get a different resolution of the sampling rate depending if the simulation runs on a slow or a fast computer.

How can we solve this problem? One option is to slow down the iteration steps in breve. This will make the simulation always run slow, but we still do not have any guarantee that it will not go slower. So the best solution is to control each iteration directly from MATLAB. This ensures that the two programs are synchronized. By modifying the *iterate* method in the controller superclass which we mentioned in section 4.2.1, we can make the whole breve simulation iterate only when MATLAB sends a specific command. Our solution was simply to rename the *iterate* method to *manual-iterate*. We then created a method in our simulation controller which calls *manual-iterate* only when it is invoked from MATLAB.

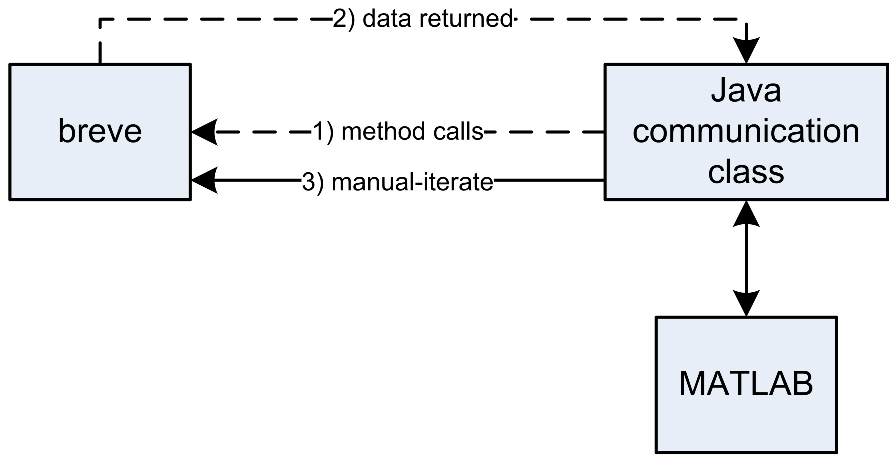

Figure 4.1: Here we can see how MATLAB and breve interacts. 1) If MATLAB needs to give special instructions, for example setting a certain velocity on a robot joint, or asking for joint orientations, a method call is sent to breve. 2) If MATLAB asked for data, breve sends back the data right away. 3) Finally, as long as we want the simulation to continue, MATLAB needs to call the *manual-iterate* method.

Chapter 5

Simulation

This simulation is mainly based on an article by Tani, Ito, Noda and Hoshino, *Dynamic and interactive generation of object handling behaviors by a small humanoid robot using a dynamic neural network model* [21]. In this article, the previously discussed RNNPB [14, 27, 15, 2] is used for learning object handling patterns. However, in my experiment, the focus is on sensory input that can be used for imitation, rather than sensory input for handling an object. The source code can be found in Appendix D.

5.1 Simulation Setup

The simulated robot in breve used for this experiment is Axel Tidemann's *Tiny Dancer*. The robot is a simplified human-shaped robot. It has a torso, two legs, two arms and a head. The legs each consists of three connected rectangular parts: the upper and lower legs and the foot. Similarly, the arms consists of two rectangular parts: the upper and lower arms, and one round part: the hand. In addition, I have added two light sensors on the head.

In this experiment, only the arms and the light sensors are used. The rest of the body is just there for the visual presentation. To simulate a real human, the shoulder joints are as defined in breve, "ball joints". This means that the shoulders can be manipulated in three different degrees of freedom. The elbow joints are "revolute joints", which can be manipulated in one degree of freedom.

The light sensors detect light and outputs values between 0 and 1, depending on the intensity of the light. In practice, they measure the distance to an object

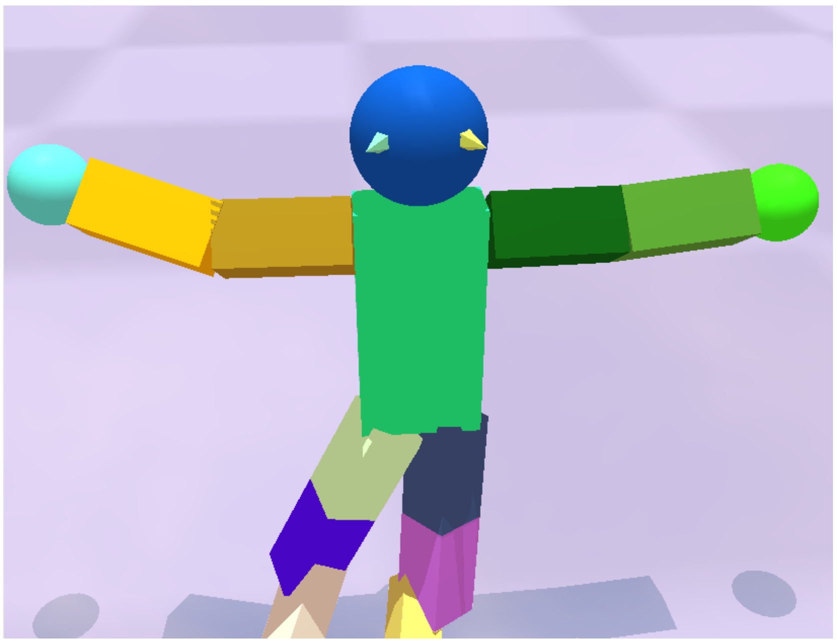

Figure 5.1: The simulated robot in breve.

which is a simulated light. The sensors are placed horizontally so that if the light moves horizontally, one sensor value will be greater than the other. This means that the robot is only able to sense a horizontal two-dimensional change in the light's position. The two axes in the two-dimensional plane is given implicitly in the sensors; the depth axis by the avarage of the two sensor values, and the length axis by the proportion of the two sensor values. Figure 5.2 illustrates this.

As was described in Chapter 3, the simulations run in two phases: a training phase and an interaction phase. In the training phase the data is first collected, before it is used to train the RNNPB. In the interaction phase the RNNPB keeps receiving information about the current state, while outputting the next state to the PID-controller. The PID-controller then sends the necessary commands to the robot. The data is collected by invoking methods in breve that returns sensor values and joint angles, while commands are given by invoking methods that set joint velocities. As described in Chapter 4, those methods can be invoked from MATLAB.

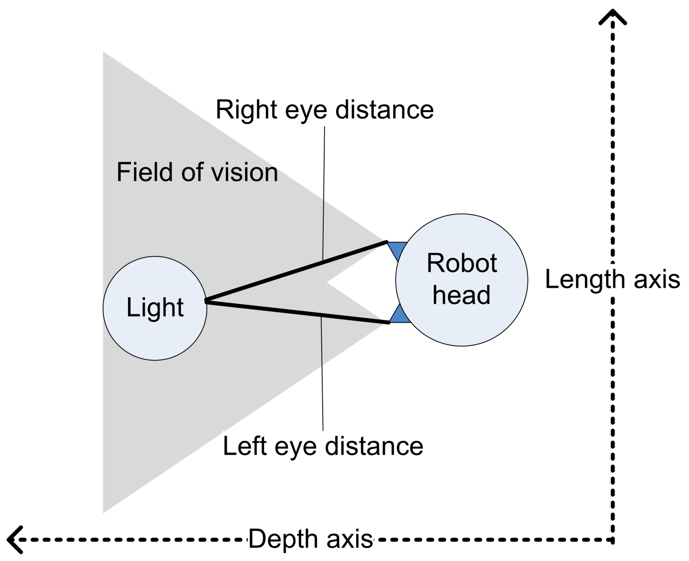

Figure 5.2: As we can see here, each "eye" measures the distance to the light. To calculate the distance from the head to the light (the depth axis), the mean of the two distances can be taken. To calculate the position of light on the length axis, the proportion of the two distances can be used. Note that this is not done explicitly in the simulation. The position is only given implicitly by the two values.

5.1.1 Simple Pseudo Code

To show how the experiment is set up, a simple pseudo code is set up for each of the two phases.

Training Phase

1. Main function asks breve for current state.
2. breve returns current state (sensor values and joint angles). The current state is also stored for later training.
3. Send predefined desired state to main function.
4. Main function finds difference between current and desired states, and send the error to the PID-controller.
5. The PID-controller calculates and returns the motor command necessary to go to decrease the error.
6. Main function sends motor command to breve.
7. breve executes command.
8. Return to 1. until all predefined desired states have been achieved.
9. Use stored states to train the RNNPB.

Interaction Phase

1. Main function asks breve for current state.
2. breve returns current state (sensor values and joint angles).
3. Main function sends current state to RNNPB.
4. RNNPB calculates the next desired state and sends it to the main function.
5. Main function finds difference between current and desired states, and send the error to the PID-controller.
6. The PID-controller calculates and returns the motor command necessary to go to decrease the error.

7. Main function sends motor command to breve.

8. breve executes command.

9. Return to 1.

Illustration of the Experiment Setup

Figures 5.3 and 5.4 illustrates the experiment setup. As we can see, MATLAB components are colored blue, Java classes are colored green, and breve components are colored light orange/red.

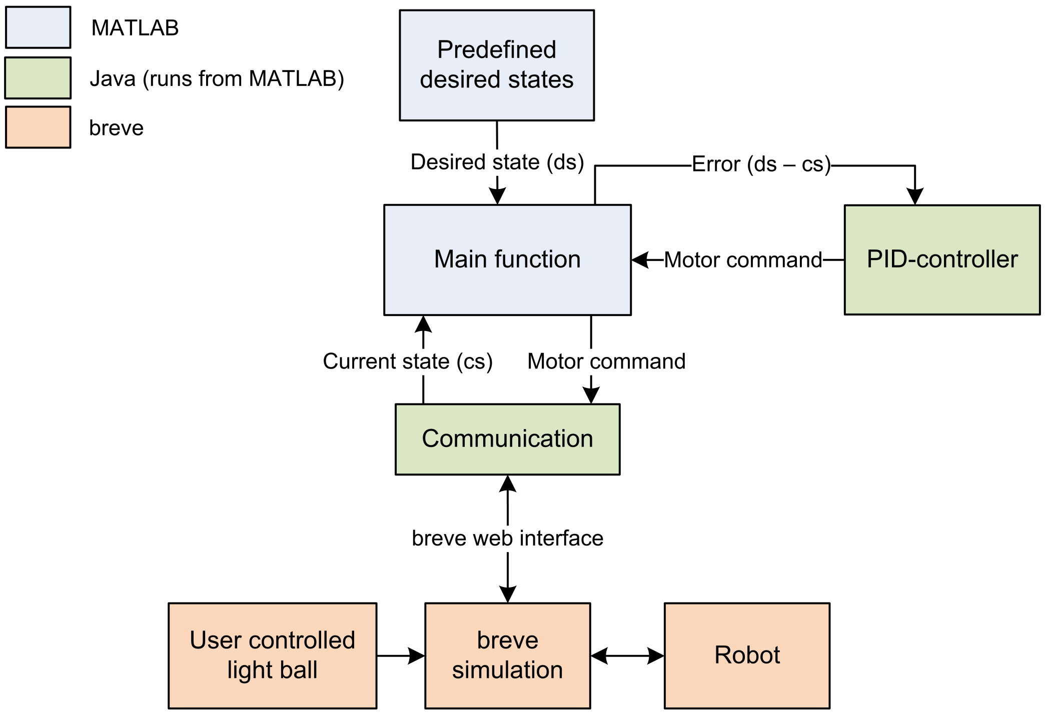

Figure 5.3: The PID-controller receives the error (difference) between the predefined desired state and the current state, and sends a motor command back to the main function. The motor command is then passed on to breve and executed. breve sends back the current state (both the sensor values detecting the light ball and the robot's angles). The current state is then compared to the next desired state, and the new error is sent to the PID-controller. The main function stores the current state for each time step. This data is then used to train the RNNPB.

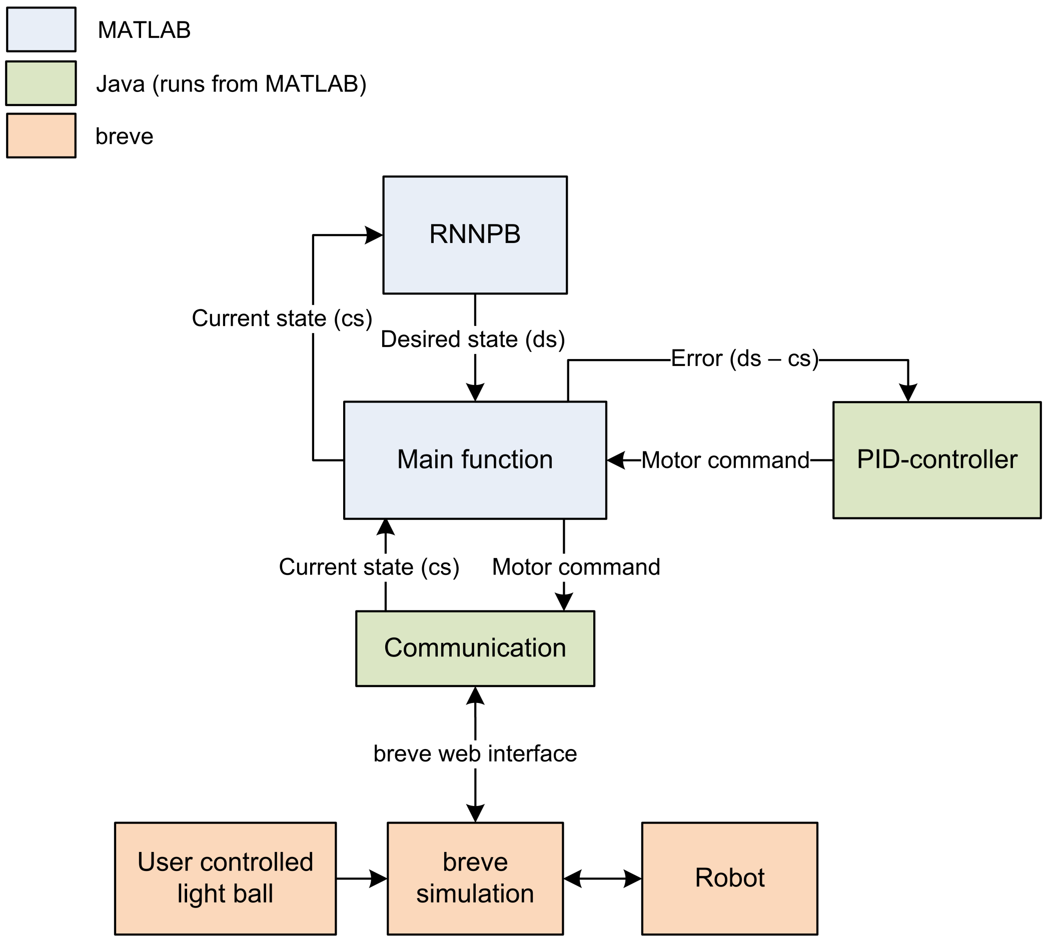

Figure 5.4: The RNNPB receives the current state and predicts the next desired state. An error between the current and desired states is calculated and sent to the PID-controller, which returns a motor command. The motor command is sent to breve, which executes it. breve sends the current state (both the sensor values detecting the light ball and the robot's angles), which is used by the RNNPB for deciding the next desired state.

5.2 The Experiments

During the simulations of the RNNPB with the PID-controller, several different experiments have been done. The variations of the experiments were in number of input sequences, number of joints used, and different kinds of sequences. During the simulations, interaction with the outside world (the light ball) was an important part. This was important, since the results of these experiments would show how the RNNPB works and how it can be used for imitative learning. Appendix C shows the list of simulations with corresponding filenames for videos showing the simulations.

The training data for each sequence was obtained by preprogramming the PID-controller to move the robot's joints into certain goal states, while MATLAB recorded all the angles and sensor input.

5.2.1 Simulation 1: Separate Arm Movement

Setup

In the first simulation a very simple form of "imitation" is learned. When the light ball is moved to the side of the robot, the robot should move the corresponding arm up. If the ball moves right, the right arm should move, and if the ball moves left, the left arm should move. If the ball is moved to the center, any arm that is raised should move down. Figure 5.5 shows hows the robot lifting its arm.

For this simulation, 25 time steps is used for each sequence (movement). 4 sequences are taught during the training phase: 2 sequences for lifting each arm while the light ball is moved to the sides, and 2 sequences for lowering each arm while the ball is moved to the center. One aim of this experiment is to see that the PB vectors converge differently for each sequence. Of course, another goal for this experiment is to see that during interaction the robot responds to the ball's position and movement the same way as during training. It is important to see that the RNNPB indeed works as it should with more "realistic" sequences than in the simpler experiment in my depth study [2]. As was mentioned in Chapter 1.1, this is one of the conditions that should be met in order to have successful imitation. The RNNPB should be able to interact with the outer world.

Before the training, the PID-controller is preprogrammed to raise and lower each arm, by giving it a list of desired angles it should be in. Because the PID-controller

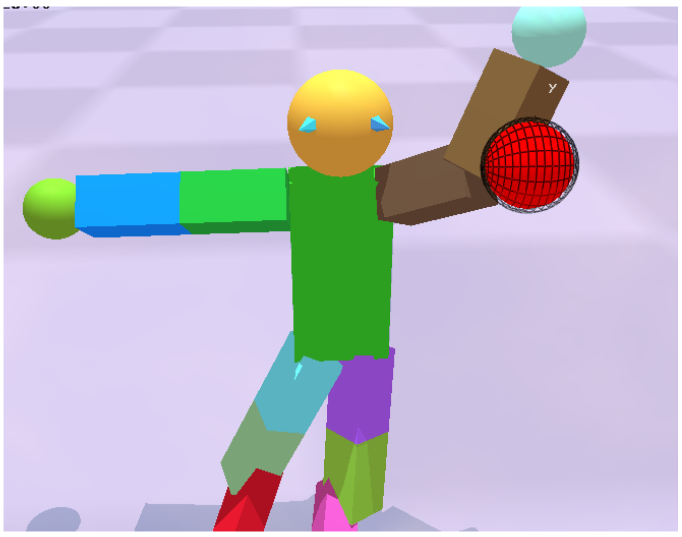

Figure 5.5: As we can see, the robot is lifting up the arm when the ball is placed on the side. For movies showing the robot performing this action see Appendices C.1 and C.2.

is an inverse model, it will move the robot's arms into the desired angles. At the same time MATLAB records the state (angles and sensor input) in each time step. The light ball is controlled manually with the mouse by using the **select** feature in the breve GUI. An alternative would have been to program the ball's trajectory. There are several reasons why manual mouse control was chosen. Firstly, it is easier to move the ball to the desired position. Secondly, controlling the ball by hand gives a more realistic and noisy data set. It is a known fact that noise can improve a neural network's generalization abilities [12].

After the recording of the states is done, training of the RNNPB is performed. Afterward, during interaction, the PID-controller receives the desired states from the RNNPB. The RNNPB uses the current state (angles and sensor input), and outputs the predicted next state to the PID-controller, as explained in Chapter 3.2.1 and Figure 3.2. Again, the light ball is controlled by mouse. The idea is that there should be a certain level of noise.

Results

Several trials with slight variations where done with this experiment in order to find good parameters for learning rates, number of nodes, ect. The best results were achieved using 10 hidden nodes, 20 context nodes, and 3 PB nodes. A learning rate and momentum of 0.002 and 0.8 were used for the weights, and a learning rate of 0.002 was used for the PB nodes. In addition, only 2 joint angles in each arm is used for this experiment: the elbow joint, and one axis in the shoulder joint. This brings the total number of input nodes to 29 (4 joint angles + 2 light sensor values + 3 PB nodes + 20 context nodes = 29). One training session takes around 30 minutes, depending on the load on the computer.

After training, during the interaction phase, the robot is able to lift the correct arm when the ball is moved to one of the sides. It keeps the arm up as long as the ball is there. When the ball is moved to the center, the arm is lowered again. It is important to note that the network seemed to generalize the ball's position in addition to the ball's movement. Had it only captured the movement and not the position, the arms would have been lowered whenever the ball stopped moving, not matter what the position was. In the next simulation, in Section 5.2.2, the goal is to capture only movement.

Several interesting facts were discovered during the current simulation. Before the experiment, it was expected that each individual sensor's strength and the proportion between the two sensors' values would affect the output of the RNNPB more or less equally. In the current experiment, however, it was discovered that it was the proportion between the two sensor values that matters most. [1]

This was discovered by the fact that during interaction, the ball had to be placed closer to the robot's head in order to see any effect from movement and position. At first, a difference in the sensor scaling during training and interaction was thought to be the reason for this. But after ensuring that the sensor values was approximately the same, the ball still needed to be closed to the head during interaction. This is due to the fact that the network seemed to learn to mostly use the proportion of the two sensor values, rather than the values themselves. When the ball is closer to the head, movement and position are more easily detected as the relative distance between the two sensors becomes larger. In other words, the proportion changes more dramatically. This is the same as with human eyes. The closer an object is, the smaller the movement has to be in order for it to be detected.

Even after several trainings and adjustment of parameters, the results are not always perfect. There is especially one particular error which seems to be consistent, though in different degrees over several trainings. During interaction, sometimes one arm is raised when the ball is far away from the sensors, though still in the middle. The same happens when the ball is removed completely from

[1]It should be noted that the proportion is not calculated and given explicitly. The proportion mentioned is only given implicitly by the two different sensor values.

the view of the robot.

Several things may be the cause of the error described above. Though after studying the PB values, it can be concluded that the reason this happens is simply because there is no specific sequence where the ball is still in the middle. The information is only given implicitly by lowering the arms when the ball is moved toward the center. Of course, one solution would be to train the network with one more sequence: keeping the arms lowered while the ball is still. I feel, however, that this would be "cheating", as the network should be able to use the implicit information.

PB Analysis

Studying the PB values reveals that the two sequences for lowering the arms is perhaps not learned properly. While the PB values for raising the arms seems to converge into stable vectors, the PB values for *lowering* the arms is somewhat unstable. In other words, they do not seem to converge as easily. After training, these are the resulting PB vectors:

Raising right arm [1 0 1]

Lowering right arm [0.59 0.47 1]

Raising left arm [0 0 0]

Lowering left arm [0 1 0.31]

Further, we can examine the plot in Figure 5.6. As we can see, the values for raising the arms seems to "firmly" converge into 0s and 1s. The values for lowering the arms, seems to settle into values in the middle. However, as we can see on the plot, the values do converge into values that "makes sense". The points can easily be separated into two groups; one group for each arm, and one group for each action. Of course, this could purely be a coincidence, but similar PB values were obtained after several training runs. Another point is that Tani *et al* [21] mentions that when there are certain common characteristics among the training patterns, the PB space tends to reflect that. When the RNNPB can not find any common characteristics, the PB space can be distorted and seem random [21]. Comparing this to the current simulation, it shows that my results may not be merely a coincidence.

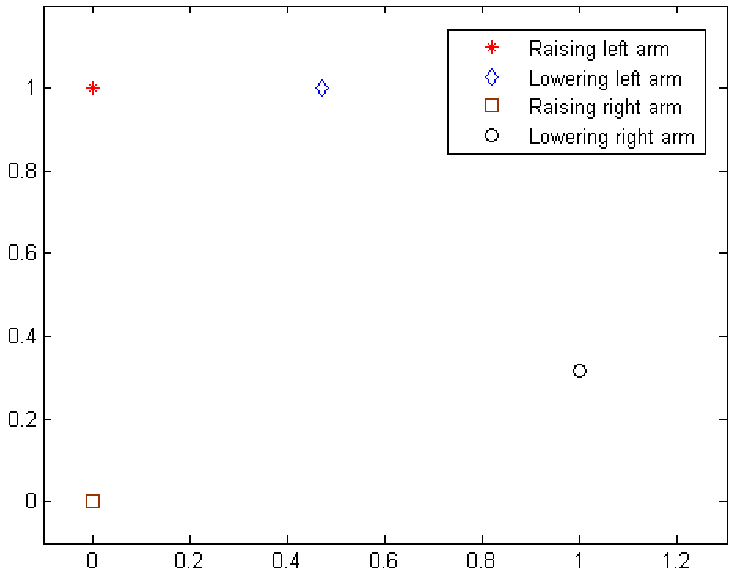

Figure 5.6: This is a plot of the PB values for this simulation. The two last values in each PB vector is plotted against each other. As we can see, an interesting feature appears. The values for raising an arm is in the lower X-axis, while the values for lowering is higher. On the Y-axis, we can see that the values for the left arm are high, while the values for the low arm are low. This suggests that during training, the PB values is distributed in such a separable manner.

Summary

The results in this simulation show that at least the two first conditions from our main hypothesis Chapter 1.1 are met. The RNNPB is able to learn patterns (actions) and reproduce them. It is also able use outside information, the position of the light ball, to determine which action to execute. Tani's claims which we discussed in Section 1.1 also seems to hold. Different PB values is selected for different situations. The PB values can thus indicate which situation the robot is currently in. In addition common characteristics in different patterns appear to be found and represented in the PB space. However, the third condition for the main hypothesis is not met. Only the position of the ball was learned, rather than sequences of positions. This means that the RNNPB was not able to learn to associate movement with action.

The results of this experiment appears to be promising. All but one of the conditions are satisfied, and the two claims by Tani seem to be holding.

5.2.2 Simulation 2: Capturing Movement

Setup

In this simulation, the focus is on capturing the *movement* of the light ball rather than the position, as turned out to be the case in the first simulation. This is the third condition in the main hypothesis from Chapter 1.1. The robot should learn to act based on a sequence of states (joint angles and sensor values).

In this experiment, instead of moving the arms separately, both arms are raised when the ball is moving and lowered when the ball is still. Thus, the biggest difference from the first simulation is that the arms are raised *only* when the ball moves. The reason why both arms are raised simultaneously, no matter which direction the ball moves in, is so that features in the PB values for movement might appear without interference from features discussed in the previous simulation.

For this simulation, 40 time steps is used for each sequence. 3 sequences are used: 1 sequence where the ball is moving, and 2 sequences where the ball is still. In the first sequence, the ball is moving fast back and forth (sideways) while both arms are being raised. In the second sequence, the ball is stopped while both arms are being lowered. Finally, in the third sequence, the ball is still and the arms are in a lowered position. Even though the main goal of this simulation is to capture movement, it is also important to study the convergence of the PB

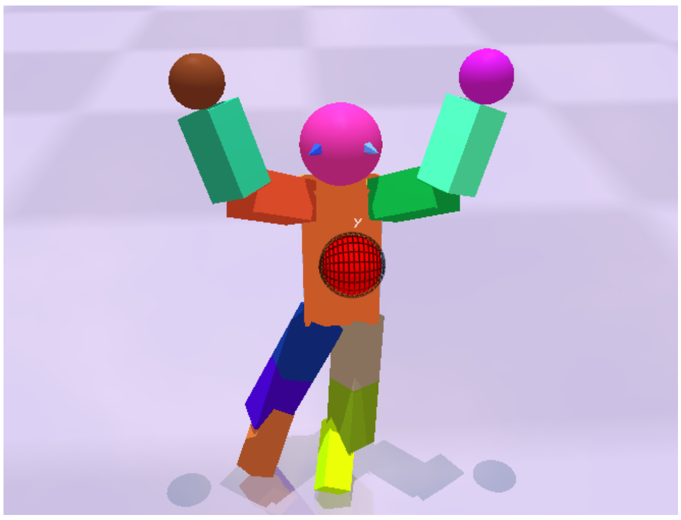

Figure 5.7: The robot is lifting up both arms when the ball is moving. For movies showing the robot performing this see Appendices C.3 and C.4.

values.

The same procedure as for the previous simulation is followed for training and interaction. Before training, the PID controller is programmed to raise both arms at once and then lower them. The light ball is controlled manually using the **select** feature in the breve GUI and mouse control. MATLAB is recording states. After using the data to train the RNNPB, the robot can be interacted with by controlling the ball manually. The trained RNNPB gets "control" of the PID-controller, feeding it with a desired state for each iteration.

Results

In this experiment, a considerable amount of time was spent training the RNNPB and tuning parameters. Even after much time spent, it is still difficult to get a "good" training. For the sequential properties to be learned, a great deal of effort is required to find the correct parameters. One problem is that each training takes about 30 minutes, which makes it time consuming to do small changes in the parameters and rerun training.

The best results for this simulation were achieved using 10 hidden nodes, 20 context nodes, and 2 PB nodes. Learning rate and momentum was 0.002 and 0.8 for the weights, and a learning rate of 0.002 was used for the PB nodes. As with the first simulation, only 2 joint angles in each arm is used for this experiment: the elbow joint, and one axis in the shoulder joint. The total number of input nodes is 28 (4 joint angles + 2 light sensor values +3 PB nodes + 20 context nodes = 29). In interaction phase after training, the robot is not always doing as it was trained to. There is a sense of randomness of what the robot does. However, after several trials with different parameters for training, some common traits are discovered. The goal is that the robot should raise the arms when the ball is moving and lower them when the ball is still. This appears to be working only when the ball is moving to and from certain positions. For example, on some occasions the arms are only raised when the ball is moving back and forth at one side of the robot. The arms remains lowered if the ball is moved on the other side. Appendix C.4 refers to the movie file showing the robot in interaction. As the movie shows, when there is a reaction from the robot, it is quite weak.

In the first simulation it was discovered that the proportion of the two sensor values affects the output more than the sensor values themselves. This appears to be true for this simulation as well. As with the first simulation, when the ball is closer to the head, moving the ball generates a more dramatic change than when the ball is far away. It should be mentioned that the achieved change is not always the desired change. The robot's behavior is quite unstable, and sometimes only one arm is raised; this is an action that was not introduced during training. It appears that the third condition from Chapter 1.1 does not hold; at least not fully.

PB analysis

Looking at the PB values after training in this simulation, we notice something quite interesting. These are the PB values after a “good” training:

Raising both arms (ball moving) [1 0.36 0.05]

Lowering both arms (ball still) [0 1 0.61]

Arms lowered (ball still) [0 0 0.69]

Now, after seeing those values, it is quite tempting to claim that the PB space appears to capture the common characteristics of the patterns. Let us speculate on what the PB value might mean.

It appears that the first and third values (in each vector) represents the perception of the ball. In the first pattern, the ball is moving and the values are 1 and 0.05.

In the two other patterns, the ball is still and the values are 0 and around 0.65. As we can see, the ball moving and being still are two opposite perceptions, and thus they have values far from each other. Similarly, the second values in each vector appears to represent the movement of the arms. In the first pattern, the arms starts lowered and are raised. The value for this is 0.36. In the second pattern, the arms are lowered down from a raised position. The PB value for this action has converged to 1. Finally, in the third pattern, the arms are not moving at all, and the converged value is 0. We have three different actions, and we get three different values. This is an initial speculation based on just looking at the PB vectors. Looking at the plots in Figure 5.8, we can see the common characteristics more closely.

Of course, this analysis of the PB vectors is pure speculation. It is easy to look at the PB vectors and draw conclusions based on seemingly common features. However, the fact that the PB values do not appear to be completely random, might be an indication that common characteristics are indeed captured in the PB space. Though as long as there is no direct proof, we can only theorize what the PB values mean.

Summary

The results in this simulation are a bit unclear. The goal was to test the third condition from Chapter 1.1 and see if the RNNPB managed to learn to detect different sequences. The results during interaction is too unstable, and we can conclude that the condition does not hold in this experiment. In theory, the sequence of changing sensor values should change the PB values, which in turn would generate a change of action. It appears, though, that the effect of constantly changing sensor values is too weak to induce a change in the PB vectors. It may be assumed that the reason why the results in the experiment are so unstable is that further tweaking of the parameters should be done.

Tani's claims about PB vectors seem to hold. Each pattern (or situation) gets its own unique PB vector. In addition, it appears that common characteristics are reflected in the PB space. Of course, the latter is only an observation that cannot be proved.

Though this experiment was less successful than the first, some interesting PB values were discovered. The condition we wanted to test did not hold. As mentioned, however, correct behavior appears occasionally, and my theory is that correct parameters for training the RNNPB needs to be found.

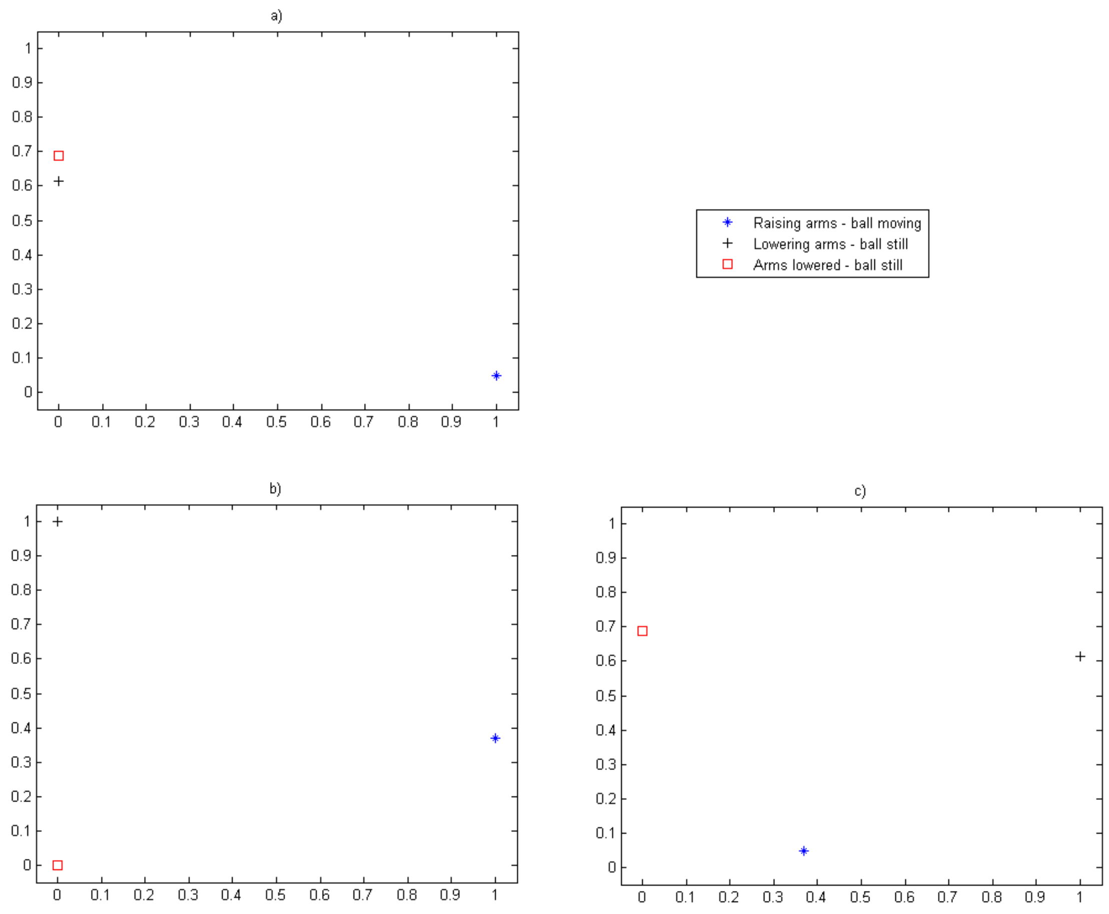

Figure 5.8: All three values in the vector for each pattern are plotted against each others. **a)** *First value plotted against second value.* As we can see, the two patterns where the ball is still is clustered together far from the pattern where the ball moves. This might mean that the first and third values contains most of the information about the perception of the ball. **b)** *First value plotted against third value.* In this plot, we can see that all the patterns are far from each other. However, the PB vectors for patterns where the ball is still is closer to each other (on the X-axis). This corresponds with our theory that at least the first value represents the perception of the ball. **c)** *Second value plotted against third value.* In this plot too, all the patterns are far from each other. But if we look more closely, well notice that the PB vectors for raising and lowering the arms are closer to each other, while the vector where the arms are still is further away. Perhaps this plot shows how much movement there is in the arms.

5.2.3 Other Simulations

The two simulations described in the previous sections where the two simulations I spent most time setting up and analyzing. A few other experiments were done in addition. However, because finding good training parameters is a time consuming process, only a quick review has been done for the following experiments. In the end of this section, a few general observations have been made.

Movement without a light ball

Some early tests were done without the light ball. This was to verify that the RNNPB managed to learn sequences. Simple tests were done without thorough testing. Generally these tests were successful, though only simple sequences were tried. This showed that the first condition for the hypothesis from Chapter 1.1 holds.

Separate Arm Movement with Always Moving Ball

A variation of "Simulation 1" were done. Instead of stopping the ball when the arms are still, the ball was moved slightly back and forth in the middle. The aim was to keep the proportion of the two sensor values constant, while the value strengths varied. This was to verify the discovery about the proportion being more important in the first simulation.

As expected, the results for this simulation was very similar to the ones in "Simulation 1". There was a slight less success rate during interaction, though this might due to the fact that the exact same training parameters as for the first simulation was used. A few tweaks might have made interaction more successful.

Capturing Movement 2

This was an extended version of the second simulation. However, instead of using only three patterns, four patterns were used:

Pattern 1 Ball moving back and forth, while arms being raised

Pattern 2 Ball still in the middle, while arms being lowered

Pattern 3 Ball still at the left side, while arms lowered

Pattern 4 Ball still at the right side, while arms lowered

The idea was that the RNNPB would be trained to lower the arms no matter where the ball was, as long as it was not moving. However, this did not work too well during interaction. The arms ended up being lowered at all times, even when the ball was moving. The RNNPB seemed to generalize everything into lowering the arms. A solution to this might be finding better training parameters; for example increasing the number of hidden nodes.

Observations

During the simulations, a few general observations are worth noting. These are only observations, and it can not be proved that they will always occur.

- Situations which are dependent on a single state, for example the position of the light ball, are easily recognized during interaction.
- Generally, the PB vectors always converge into different values for each situation.
- Very often, the PB values "makes sense". Patterns with common characteristics get PB vectors that reflects this.

5.3 Summary and Discussion

Looking back at the goals in Chapter 1.1, a conclusion can be made that the conditions in the main hypothesis partially hold. Clearly, the first condition holds in all experiments. The robot manages to learn to reproduce sequences in both of the discussed simulations, as well as in additional "test simulations". The second condition also holds, as the robot's behavior can always be affected by sensor input. However, the third condition does not fully hold. In the second simulation, the goal was to specifically test this condition. The results showed that the robot is successful only occasionally. It appears to be very difficult to get the robot to learn to recognize a sequence of perceptions.

It was expected that two first conditions in the hypothesis from Chapter 1.1 would hold. These are general features that are expected to be present in any simple recurrent neural network (SRN). However, the third condition is not necessarily expected to hold in an SRN. The SRN is supposed to reproduce sequences, though not necessarily *recognize* sequences. It was hoped that the RNNPB would be able to so. In theory, the PB vectors will change when a sequence of changing sensor values is introduced, which again will generate a change of action. As we can see in Appendix C.4, the robot responds weakly to the movement of the ball.

There might be several reasons why the changing sensor values are not properly inducing a change in the PB vectors. As mentioned, it might simply be that the training parameters (number of nodes, learning rates, ect) need to be tweaked further. However, there might be another reason. During my depth study, it was discovered that on some occasions, the inverse calculation of the PB vectors during interaction failed. This happened even if the PB vectors had properly converged during training. This error might be reappearing here. The features in the two first conditions are not very dependent on the PB vectors, as they are general SRN features. The feature in the third condition, on the other and, is dependent on the PB vectors. This makes "Simulation 2" more easily affected by a faulty inverse PB calculation. The solution to this problem might simply be in tweaking the training parameters further, or it might be to review the formula for PB (inverse) calculations.

When looking at the two claims by Tani presented in Chapter 1.1, we can say that they appear to be holding. The first claim is an expected feature. In fact, it is the main feature of the RNNPB. During training, the PB vectors are supposed to converge differently for each situation (or pattern), so that they may be used to reproduce corresponding patterns later. The second claim, on the other hand, is more of an interesting observation rather than a useful feature. Common characteristics appear to be reflected in the PB space. This cannot be properly proved, other than observing numerous occurrences. However, more thorough tests can be done than the ones done in this project. For example, one could introduce more patterns with combinations of common characteristics, and see if they are reflected in the PB space as one would expect. This might be considered for any future research done with the RNNPB.

All in all, the simulations have yielded some interesting results. The main hypothesis does not currently hold fully. However, it appears that the problem with the third condition may be solved with more work. Therefore, my opinion is that RNNPB certainly does hold a great potential for imitative learning. The RNNPB is difficult to train properly though, and it probably needs much work and research before true imitative learning can be implemented with it.

Chapter 6

Comparison between the Forward and Inverse Models and the RNNPB architecture

In this chapter, a comparison between the two models is made. The training of each is done in different ways. Both architectures consists of similar components, though connected differently. In the comparison, it is assumed that some or most of the components in each model is neural networks.

6.1 Training

One clear difference between is the training and interaction phases. The training of the forward and inverse model has to happen on line during the interaction, while the training of the RNNPB can be done after the data has been collected in a training phase. This poses some challenges for training forward and inverse models, as the training of the neural networks occurs while the data is being collected. Training becomes quite time consuming, especially if one is using a real robot which moves in real time. In the case of using a robot simulator, the simulation might be sped up in order to collect data faster. However, the simulator usually needs to process information, which will still lead to long training times. In addition, the forward and inverse architecture makes the robot "try out" different actions while training; something which also leads to long training time.

The RNNPB model, on the other hand, is different. The data for the actions that are to be learned is first recorded during a simulation. Afterward, the RNNPB is trained off line. This means that the neural network can be trained without having to interact with the robot (or the robot simulator). Of course this cuts down training time considerably. As a comparison, we can look at Axel Tidemann's forward and inverse implementation, and my own RNNPB implementation. While Axel's implementation uses many hours for each training, my own uses around 30 minutes. Of course, there are differences in the implementations. Axel's implementation uses a set of many smaller neural networks, while my own is one large neural network (the RNNPB). However, the difference in the training times gives an indication on how different they are.

There is a great advantage with the training method of the multiple paired forward and inverse models: new actions can be learned during interaction. All that is needed is adding another paired forward and inverse model. Only this new "module" need to be trained to handle the new action. The RNNPB, however, has to be taken off line. Data for the new action has to be added, before the *whole* network has to be retrained. This obviously makes the forward and inverse model more modular than the RNNPB.

6.2 Architectural Comparison

As we have established, the RNNPB has two phases; the training phase and the interaction phase. The forward and inverse model is trained during interaction, and makes no distinction between the two phases. In order for the comparison to make most sense, it is the interaction phase architecture from the RNNPB that will be compare to the forward inverse model.

If we begin by looking at the similarities in Figure 2.1 and Figure 3.2, we can see that they share very similar components. The "robot"-component is practically identical; the robot always receives some motor command, and it always outputs the current state. Having previously studied the other components, we know that the inverse model and the PID are also identical components. In fact, a PID can be used an inverse model and vica versa. The input is a current state and a desired state, and the output is the command necessary to achieve the change. This leaves us with the forward model and the RNNPB. Those components are similar in some ways. Both can predict. The forward model *can* be an RNN, and the RNNPB is an extended RNN.

The main difference between the two, however, is that the forward model uses the command from the inverse model and the current state to predict what the next state will be. It is like saying: *This is the current state, and this is the* *command I am applying. What will the consequences be?* The RNNPB on the other hand, only takes as input the current state, and outputs the prediction of what the next state will be. Instead of using the applied command, it obtains the predicted state by implicitly looking at previous states. In addition, the parametric bias helps it "keep in mind" which action it is currently performing.

Of course, a great difference of the overall architectures, is that the forward and inverse architecture needs one pair of forward and inverse model for each action, the while the RNNPB architecture requires only one neural network.

It is also important to note how the two models differs in what they are applied to. The forward and inverse model is for predicting consequences of actions. In the case of the experiments we have studied, this has largely been own motor commands. In the RNNPB model, the focus is on making an A.I. that interacts with outside objects. Even though both models are well suited for imitative learning, and are emulators of mirror neurons, they have different approaches to the problem.

6.3 Summary and Discussion

As we have seen, one of the main differences of the two architectures is that the RNNPB is split in two, a training phase and an interaction phase, while the multiple paired forward and inverse models can interact while learning. The components in the two architectures are quite similar. They are built differently though. In addition, the RNNPB architecture is much less modular. It consists of only one large neural network for all the behaviors, in addition to the PID-controller. The multiple paired forward and inverse models on the other hand, is built of a set of modules that each can handle one behavior.

The overall difference between the two architectures though, is that the RNNPB is simply a predictor of sequences of states. In addition, the PB vectors gives it the ability to indicate which behavior (or situation) it is currently in. The forward and inverse model, on the other hand, is a predictor that looks at consequences of commands applied to a current state. Of course, the behavior which the forward and inverse model is currently in can easily be found by seeing which module is active.

It is difficult to say which of the two models are the best emulator of mirror neurons. It can be said though, that the forward and inverse model has a higher level of cognition, as it can predict consequences of actions applied to a state / situation. However, it may be possible to let the RNNPB include actions (or commands) in its input, making it a predictor of consequences as well. This has not been tested in this project though. The RNNPB also seems to have the ability to generalize and represent common characteristics between patterns in the PB vectors. If this is true, then a form of internal representation is made and connected to a "symbol" (the PB vector). This may perhaps be a route to solve or avoid the symbol grounding problem.

Both architectures should be studied and developed further. It appears that both can be used as emulators of mirror neurons. Both have the ability to predict or "imagine" sequences of states. It will be interesting to see how the development continues in the future, and if true imitation is achieved using one of the architectures.

Chapter 7

Program Documentation

In this chapter, the source code is documented. In the first section there is a short description of each file. In the second section diagrams show how the files are connected to each other.

7.1 File Description

In this section, a short description of each file is done. The next section will illustrate how the files are connected to each other. The files that has a filename containing a # are scripts specific for one simulation. The # stands for the number of the simulation.

7.1.1 MATLAB Files

rnnpb.m

Function for constructing an neural network with parametric bias (RNNPB). As input, this file receives variables that specify size and learning rates for the network. The file returns an RNNPB with one hidden layer.

rnnpb_activate.m

This function is for activating an RNNPB. Activation means that a pattern is sent through the network. The final layer of the network is the output that the network produces based on the received pattern. The activation function receives the RNNPB, the input pattern, and the PB vector as input. It returns the activated network.

rnnpb_train.m

Function for updating the weights and PB vectors in an RNNPB (for one iteration). The input for this function is the RNNPB, the error (desired state - current state), and the number of the pattern. The number of the patterns tells where the pattern is located in the input layer of the network. This is so that the correct PB vector is updated.

run_training.m

Function for running a whole training. This function basically creates a new RNNPB and runs rnnpb_train.m in a number of iterations. As input, it receives the variable containing the patterns to be learned, the size of the network, learning rate, and the number of epochs. It returns a fully trained RNNPB.

trainSim#.m

This file is a script for starting run_training.m for a specific simulation. In this script, the size, learning rate and number of epochs can be set.

scaleSim#.m

This file is a script for scaling down the data variables recorded from breve for a specific simulation.

runSequencesSimulation#.m

Script for recording data for specific simulation. This a "preprogram" that tells the PID-controller how to move the arms. Simultaneously, the user must control the light ball in breve.

oneSequence.m

This function is for running one preprogrammed sequence. As input, it receives the goal state (joint angles), and the number of time steps it should go on. It then uses the PID-controller to move the joints in the direction of the goal for the number of defined steps. As output, it returns a data variable containing a recording of the states (sensor values and joint angles) in each time step.

run_trained_rnnpb.m

This script runs iterates breve using a trained RNNPB for 500 iterations.

7.1.2 Java Files

Breve.java

This class is created by Axel Tidemann, and is for communication between MATLAB and breve. As input, it takes the host (IP address where breve is running), port number, and the message to be sent to breve. As output, it returns the response from breve.

PID.java

This class works as a discrete PID-controller, based on the formula in Chapter 3.2.2. When constructed, it receives the parameters that defines how the PID-controller should act. The class also contains a method called *getNewVelocity*. This method gets the error as input, and returns the velocity needed to decrease the error.

7.1.3 breve Files

Tiny Dancer.tz

This file is the file that runs in breve for my simulations. Builds the robot, controls the joints, and is able to return all necessary data. The file was originally created by Axel Tidemann, though this version contains modifications to suit my own project. The biggest modifications are the way breve is iterated, and the light ball along with the light sensors.

7.2 Diagrams

We have now seen what each file does. However, in order for it to make more sense two diagrams showing how the files are connected to each others. Figure 7.1 is for the training phase, and Figure 7.2 is for the interaction phase.

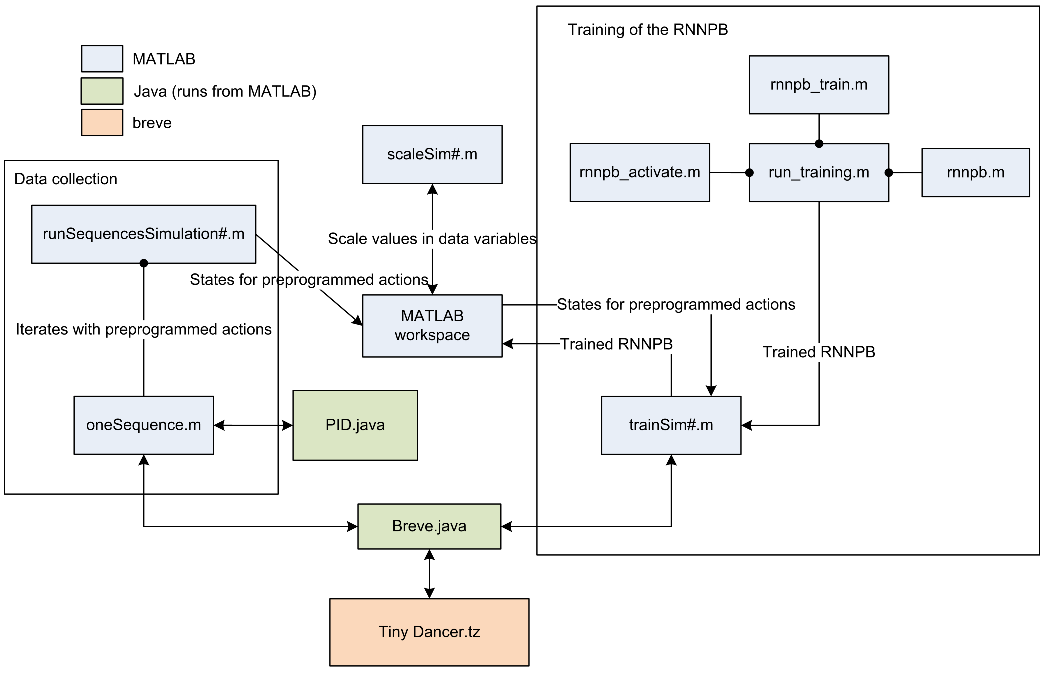

Figure 7.1: Here we can see how the files for the training phase are connected to each other. The training process is in two parts. First the data needs to be collected using a script that tells the robot how to move. Then the collected data may be used to

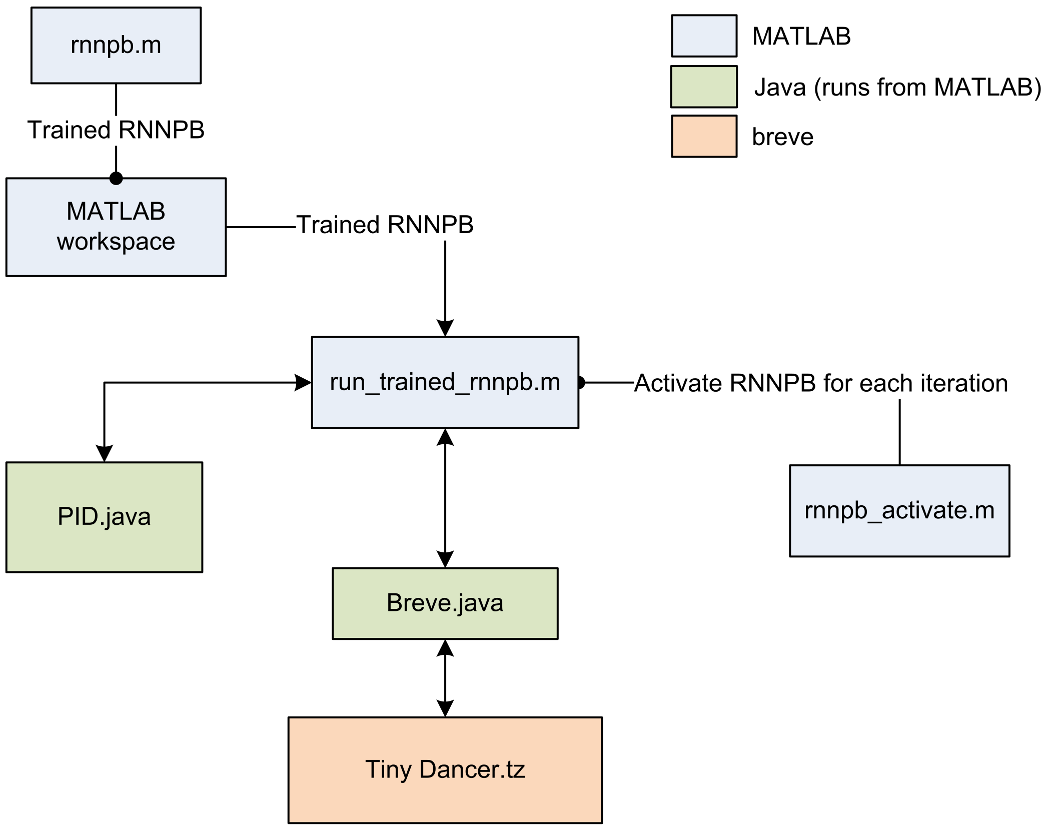

Figure 7.2: Diagram showing the relationship between the files during interaction.

Chapter 8

Conclusion

The initial goal of this project was to connect MATLAB and breve, and to do simulations within the field of imitative learning. However, more work was done beside those two main tasks.

I started by defining the main goals and introducing imitative learning and mirror neurons. I then continued to describe two promising architectures for learning motor behavior: multiple paired forward and inverse models, and the recurrent neural network with parametric bias (RNNPB). One of the greatest features of the former is modularity using a hierarchical or parallel approach. On the other hand, the RNNPB is capable of using the PB vectors to distinguish learned sequences.

Connecting MATLAB and breve was also a big part of the project. This was mainly done in cooperation with Axel Tidemann, with some input from Ole-Maruis Moe-Helgesen. After some work, we decided to go for a simple solution, and ended up using breve's web-interface, along with a custom Java communication class for MATLAB.

Simulations were done using the RNNPB and a PID-controller, and quite interesting results were achieved. The first of the two main simulations was successful, while the other had mixed results. Both, however, gave quite interesting results. In particular, common characteristics of patterns appear to be reflected in the PB space.

As for the goals that was discussed in Chapter 1.1, they have been largely achieved. A successful connection between MATLAB and breve is made, and I did some simulations testing our hypothesis on the RNNPB. However, more work seems to be necessary to completely verify if the hypothesis holds. Future work with the RNNPB should certainly include simulations where recognition of movement is central.

More comparison between the forward and inverse model and the RNNPB is also necessary. Perhaps identical simulations can be run on both architectures to see which performs best. Other work with the two models could include making a new architecture using elements from both. It would be very interesting to see an architecture that can share the advantages from both models.

All in all, this has been a quite interesting project. I learned a lot about the two discussed architectures. Connecting MATLAB and breve also gave a technical aspect to the project. Finally, it was very interesting to run simulations and "interact" with a robot using an RNNPB.

Appendix A

Terminology

This is a short description of some of the usual terms within imitative learning. This is a direct copy from my depth study [2].

Imitation To copy an action by observing how someone else is doing the action. This action may simply be an action, or it might also be something more goal-oriented.

Goal emulation [3] To produce the same results as the teacher by observation. This is especially applicable if machine that is being taught has a different form than the teacher. For example, a human that teaches a robot that only has an arm with two joints.

Learning to imitate A problem with machine imitation is to let the machine know what it is supposed to imitate. The machine has to know what and who it is supposed to learn from. If the imitation is based on visual information, a suggestion is to use a special color on whatever the robot is supposed to follow. However, there is still the problem of making it know that it *is* supposed to follow. Also, there is a problem of deciding which stimuli are relevant [8].

Learning by imitation To learn to perform an action or reaching goals by observing someone else. The essence of learning by imitation is that the robot should have only a basic system of primitives in the bottom, and that all new actions and goal-reaching abilities are learned by having someone show the robot how to put those primitive parts together. There are many advantages to this kind of learning. Humans can *show* a robot how to do something, instead of programming it manually. Also, teaching between robots might be useful. Copying one robot's program to another is difficult if the robots differs from each other [8].

Prediction In research, many of the proposed architectures utilize prediction in order to have the robot properly imitate. For example, Demiris and Hayes [9] uses a forward model that predicts the next state of the robot. This state is then compared to the demonstrators next state, and an error value is generated. This error value can then be used to correct the prediction of the next state. The result is that after learning, the robot will be able to "imagine" movements by prediction, without having to actually see the demonstrator do the whole movement. Parisi [24] explains that prediction is an important prerequisite for reaching a higher level of cognition. Demiris and Hayes talks about active imitation, where the imitator uses prediction to "understand" the actions.

Appendix B

Running the Simulations

Instructions on how to run simulations is listed here. It is assumed that the reader has some experience with MATLAB. In addition the reader must download and install breve. A link is provided in [19]. Since the work on the simulation started, a new version of breve has been released. The files has not been tested properly on the new version, and it is recommended that breve 2.3 is used.

It must be noted that testing the simulation on Windows gave poor results. breve uses too much processing power, and Windows is not able to share the resources properly. A brief test has been done on Mac OSX, where results was better. However, for the simulations to run most smoothly, they should be run on Linux. The Ubuntu [6] distribution works well.

B.1 Readying breve and MATLAB

The first thing that needs to be done is making breve and MATLAB ready.

breve

1. Copy the file *Control.tz* from */breveFiles* to your own breve classes folder, usually *breveIDE*$_2.3/lib/classes$.
2. Start up breve, and open the file *Tiny Dancer.tz*.
3. Press the play button in breve's GUI to start the simulation.

MATLAB

1. Start up MATLAB.
2. Run the command ``javaaddpath *<javapath>*'', where *<javapath>* is the path where the Java file for this project is located.

B.2 Using Trained Weights

Along with the source code, I have included two files containing MATLAB workspaces. Those to files contains trained RNNPBs and can be used for interaction right away.

1. Make sure MATLAB and breve is running by following the instruction above.
2. In MATLAB, open the file *sim1.mat* or *sim2.mat* (depending on which simulation you want to use). When asked, confirm that all variables should be imported.
3. In MATLAB open and run the scrip *run_trained_rnnpb.m*.
4. If you switch to breve, the robot should now be starting to move. Press the **Select**-button, and select the light ball. You can now move the light ball as you wish. Use the **Rotate** and **Zoom** buttons to get a better view.

B.3 Recording Actions

To record the actions for the simulations, the following must be done.

1. Make sure MATLAB and breve is set up correctly.
2. In MATLAB, open the script *runSequencesSimulation1.m* or *runSequencesSimulation2.m* (depending on which simulation you want to use).
3. Press run. The robot will start moving in breve in 2 seconds.
4. If you switch to breve, the robot should now be starting to move. Press the **Select**-button, and select the light ball. You can now move the light ball as you wish. Use the **Rotate** and **Zoom** buttons to get a better view.

5. When the recording is done, the sensor values needs to be scaled. To do so, you must examine the recorded variables (refer to the script you opened in 2. to see the names of the data matrices). The two first columns are the sensor values. You must chose a scaling value (that will divide the sensor values) that makes sure no value is above 1.
6. Open *scaleSim1.m* or *scaleSim2.m* and enter the scaling value where it is appropriate. A "sensible" value has already been chosen, but you need to make sure that the value will scale all sensor values to be below 1.
7. Run the script *scaleSim1.m* or *scaleSim2.m*.

B.4 Training the RNNPB

We now have a data set which we can train our RNNPB on.

1. Make sure that none of the sensor values (first two columns) in the data matrices are between 0 and 1.
2. Open the script *trainSim1.m* or *trainSim1.m*. Here you can modify parameters such as number of epochs, network size, learning rates, ect. "Good" values has already been selected.
3. Run the script to start the training. This should take around 30 minutes, depending on your computer.

We now have a fully trained RNNPB. To run interaction, follow the instruction in Section B.2, though ignore 2.

Appendix C

Videos

In this chapter, the files containing videos from the two simulations from Chapter 5 is described. The files are normal MPG files, and should be readable from any media player. It should be noted, however, that the video recording feature in breve makes the video based on iteration steps rather than the real time simulation. This means that the videos seem very fast compared to how the simulations actually runs when controlled from MATLAB. In addition, the videos looses resolution during recording. Unfortunately, this gives the videos a poor quality.

All four videos is located in the */videos* folder.

C.1 Simulation 1 - Training

simulation1Training.mpg This file shows the preprogrammed actions for the first simulation, while the ball is manually controlled.

C.2 Simulation 1 - Interaction

simulation1Training.mpg This file shows the interaction with the robot after training. As we can see, the robot does not behave exactly as during training. However, it follows the same behavior. When the ball is moved to the side, it raises the corresponding arm.

C.3 Simulation 2 - Training

simulation2Training.mpg This file shows the preprogrammed actions for the second simulation, while the ball is manually controlled.

C.4 Simulation 2 - Interaction

simulation2Training.mpg This file shows the interaction with the robot after training. The video shows that the robot's behavior is somewhat unstable. We can also see that the effect of moving the ball is quite small.

Appendix D

Source Code

D.1 MATLAB Files

D.1.1 General Files

rnnpb.m

```
% Creates a recurrent neural network with parametric bias (RNNPB
   , Tani)
% with 3 layers - output, input and hidden.
%
% Takes as input: size of the network, learning rate and
   momentum for the
% weights, and learning rate the PBs

function N = rnnpb(size, learning_rate, momentum,
   pb_learning_rate);

% size - the number of different nodes in the network
% size{1} is number of input/output
% size{2} is number of hidden nodes
% size{3} is number of context nodes
% size{4} is number of different sequences
% size{5} is number of PB nodes

% Store the values in the rnnpb object
N.size = size;
N.lr = learning_rate;
```

```
N.m = momentum;
N.lr_pb = pb_learning_rate;
N.nPB = size{5};


% Initialize weights
N.w = cell(1,2);

N.w{1} = randn(N.size{2}, N.size{1}); % Weights from hidden to
    output
N.w{2} = randn(N.size{1} + N.nPB + N.size{3}, N.size{2}); %
    Weights from input to hidden

% Initialize variables that holds the weight changes
N.dw{1} = zeros(N.size{2}, N.size{1});
N.dw{2} = zeros(N.size{1} + N.nPB + N.size{3}, N.size{2});

% Initialize contex weights
N.contextWeights =  0.7*randn(N.size{2}, N.size{3});

% Initialize PB vectors
N.PB = 0.5*ones(1, N.nPB, N.size{4});
N.aPB = N.PB;

% Initialize previous PB errors
N.ePB = zeros(1, N.nPB, N.size{4});
N.ePB_1 = zeros(1, N.nPB, N.size{4});
N.ePB_2 = zeros(1, N.nPB, N.size{4});

% Initialize change in PB
N.dPB = zeros(1, N.nPB, N.size{4});

% Create cell with three layers (i.e. the nodes)
N.L = cell(1,3);

% Create the context nodes
N.contextNodes = 0.5*ones(1, N.size{3});
```

rnnpb_activate.m

```
% Activation function for the recurrent neural network with
    parametric bias
% Input:    net - the neural network
%           X - the input pattern that is to be used for
    activation
```

```
% Returns activated network

function net = rnnpb_activate(net, X, PB);

% First layer: input, PBs and context nodes
net.L{1} = [X, PB, net.contextNodes];

% Activate and insert into hidden layer
act1 = net.L{1}*net.w{2};
net.L{2} = 1./(1 + exp(-act1));

% Activate and insert into output layer
act2 = net.L{2}*net.w{1};
net.L{3} = 1./(1 + exp(-act2));

% Activate and update the context nodes
cnAct = net.L{2} * net.contextWeights;
net.contextNodes = 1./(1 + exp(-cnAct));
```

D.1.2 Training Phase

rnnpb_train.m

```
% Training function for the recurrent neural network with
   parametric bias
% Takes as input:   net - the neural network (has to be an
   rnnpb.m-object)
%                   err - the error (target - output)
%                   patternNr - number (id) of the current
   sequence; this
%                               is used so that the correct PB
   vector is
%                               updated.
%
% Returns the network after one correction (one training
   iteration)

function net = rnnpb_train(net, err, patternNr);

% Output error
oError = net.L{3}.*(1 - net.L{3}).*err;

% Hidden error
hError = oError * net.w{1}' .* net.L{2} .* (1-net.L{2});
```

```
% The weight change
net.dw{1} = net.lr*(net.L{2}' * oError) + net.m*net.dw{1};
net.dw{2} = net.lr*(net.L{1}' * hError) + net.m*net.dw{2};

% Update weights
net.w{1} = net.w{1} + net.dw{1};
net.w{2} = net.w{2} + net.dw{2};

% Find the error in the PB vectors
pbError = zeros(net.size{5}, net.size{2});
for pbnr=1:net.size{5},
    pbError(pbnr,:) = oError * net.w{1}'.*net.w{2}(net.size{1}+
        pbnr,:) .* (1-net.w{2}(net.size{1}+pbnr,:));
end;

net.ePB(:,:,patternNr) = net.ePB(:,:,patternNr) - net.lr_pb .*
    sum((net.w{2}(net.size{1}+1:net.size{1}+net.size{5},:) .*
    pbError)');

% Use Simpson's rule to find the integral of the error
% (Important! These variables must be reset for each new
    sequence)
net.dPB = net.ePB + 4*net.ePB_1 + net.ePB_2;
net.ePB_2 = net.ePB_1;
net.ePB_1 = net.ePB;

% Update PB vectors
net.aPB = net.aPB + net.dPB;
```

run_training.m

```
% Function that runs a training
% Input:    patterns1 - all the patterns (sequences)
%           nPatterns - number of different sequence patterns
%           nHidden1 - number of hidden nodes
%           nCN1 - number of context nodes
%           nPB1 - number of PB nodes
%           epochs - number of iterations to run the whole set
%           n1 - learning rate for the weights
%           a1 - momentum for the weights
%           m1 - learning rate for the PBs
%
% Returns a fully trained network
```

```
function net = run_training(patterns1, nPatterns, nHidden1, nCN1
    , nPB, epochs, n1, a1, m1)

sizes = cell(1,5); % Empty cell with place for 6 numbers of
    nodes

% nInput1 = size(patterns1); % Get number of input nodes

% Fill in the number of nodes in the size
sizes{1} = numel(patterns1(1,:,1));
sizes{2} = nHidden1;
sizes{3} = nCN1;
sizes{4} = nPatterns;
sizes{5} = nPB;

% Create the neural network
net = rnnpb(sizes, n1, a1, m1);

% Number of "chunks" in each pattern
% (A chunk is a single set of inputs in a sequence.  For example
    , if we
% have the sequence [0 1; 0 0; 1 1], [0 1] would be the first "
    chunk", ect)
nChunks1 = size(patterns1);
nChunks1 = nChunks1(1,1);


% Start training
epoch = 0;
while epoch < epochs,

    % Reset weight changes
    net.dw{1} = net.dw{1}.*0;
    net.dw{2} = net.dw{2}.*0;

    % Reset PB errors
    net.dPB = zeros(1, sizes{5}, sizes{4});
    net.ePB = zeros(1, sizes{5}, sizes{4});
    net.ePB_1 = zeros(1, sizes{5}, sizes{4});
    net.ePB_2 = zeros(1, sizes{5}, sizes{4});

    % Go through each different sequence
    for pattern=1:sizes{4},
        % Initialize contex nodes
        contexNodes1 = 0.5*ones(1, nCN1);

        % Go through each "chunk" in each sequence.
        for chunk=1:nChunks1-1,
            % Set input
            X = patterns1(chunk, :, pattern);

            % Activate
            net = rnnpb_activate(net, X, net.PB(:,:,pattern));

            % Target matrix
            T = [patterns1(chunk+1, :, pattern)];

            % Calculate error and update error sum
            net = rnnpb_train(net, T−net.L{3}, pattern);
        end;
    end;

    % Sigmoid function on the PB vector
    net.PB = 1./(1 + exp(−net.aPB));

    epoch = epoch + 1;
end;
```

trainSim1.m

```
% Script that creates and trains an RNNPB with the patterns for
   the
% first simulation.  It uses the variables with recorded data
   already in
% the MATLAB workspace.
%
% IMPORTANT: Remember to scale the patterns first (as desrcibed
   in the
% instruction in the appendix of the report)

% Copy patterns
patterns1(:,:,1) = moveLeftArm1;
patterns1(:,:,2) = moveLeftArm2;
patterns1(:,:,3) = moveRightArm1;
patterns1(:,:,4) = moveRightArm2;

% Number of patterns (sequences)
nPatterns = 4;

% Number of hidden nodes in each network
nHidden1 = 10;

% Number of contex nodes in each network
nCN1 = 20;
```

```
% Number of PB nodes
nPB = 3;

% Number of epochs to run
epochs = 50000;

% Learning rate and momentum
n1 = 0.002;
a1 = 0.8;


% PB update rate
m1 = 0.002;


% Begin training
net = run_training(patterns1, nPatterns, nHidden1, nCN1, nPB,
    epochs, n1, a1, m1);
```

trainSim2.m

```
% Functions that creates and trains an RNNPB with the patterns
    for the
% first simulation.  It uses the variables with recorded data
    already in
% the MATLAB workspace.
%
% IMPORTANT: Remember to scale the patterns first (as desrcibed
    in the
% instruction in the appendix of the report)

% Copy patterns
patterns1(:,:,1) = bothArmsOut;
patterns1(:,:,2) = bothArmsIn;
patterns1(:,:,3) = bothArmsStill;

% Number of patterns (sequences)
nPatterns = 3;

% Number of hidden nodes in each network
nHidden1 = 10;

% Number of contex nodes in each network
nCN1 = 20;
```

```
% Number of PB nodes
nPB = 3;

% Number of epochs to run
epochs = 50000;

% Learning rate and momentum
n1 = 0.002;
a1 = 0.8;


% PB update rate
m1 = 0.002;


% Begin training
net = run_training(patterns1, nPatterns, nHidden1, nCN1, nPB,
    epochs, n1, a1, m1);
```

scaleSim1.m

```
% This function scales the sequences, and leaves the updated
    scaled
% variables in the MATLAB workspace.
%
% Input: sensorScale: the value which scales

% This value must be changed after checking each pattern
% Make sure that the scale is as the highest sensor value.
    Should there be
% a value that is much higher than the others (that is not
    intended, you
% can lower that value so that the scale does not become too
    high)
sensorScale = 1.9;

% A known good scale (should not be changed)
jointScale = 1.5;

moveLeftArm1(:,5) = moveLeftArm1(:,5)/jointScale;
moveLeftArm2(:,5) = moveLeftArm2(:,5)/jointScale;

moveLeftArm1(:,1:2) = moveLeftArm1(:,1:2)/sensorScale;
moveLeftArm2(:,1:2) = moveLeftArm2(:,1:2)/sensorScale;
```

```
moveRightArm1(:,5) = moveRightArm1(:,5)/jointScale;
moveRightArm2(:,5) = moveRightArm2(:,5)/jointScale;

moveRightArm1(:,1:2) = moveRightArm1(:,1:2)/sensorScale;
moveRightArm2(:,1:2) = moveRightArm2(:,1:2)/sensorScale;

moveLeftArm1(:,3) = moveLeftArm1(:,3)/jointScale;
moveLeftArm2(:,3) = moveLeftArm2(:,3)/jointScale;

moveRightArm1(:,3) = moveRightArm1(:,3)/jointScale;
moveRightArm2(:,3) = moveRightArm2(:,3)/jointScale;
bothArms(:,3) = bothArms(:,3)/jointScale;

moveLeftArm1(:,6) = -moveLeftArm1(:,6);
moveLeftArm2(:,6) = -moveLeftArm2(:,6);
moveRightArm1(:,6) = -moveRightArm1(:,6);
moveRightArm2(:,6) = -moveRightArm2(:,6);
```

scaleSim2.m

```
% This script scales the sequences, and leaves the updated
   scaled
% variables in the MATLAB workspace.
%
% Input: sensorScale: the value which scale


% This value must be changed after checking each pattern
% Make sure that the scale is as the highest sensor value.
   Should there be
% a value that is much higher than the others (that is not
   intended, you
% can lower that value so that the scale does not become too
   high)
sensorScale = 1.9;

jointScale = jointScale;

bothArmsOut(:,5) = bothArmsOut(:,5)/jointScale;
bothArmsIn(:,5) = bothArmsIn(:,5)/jointScale;
bothArmsStill(:,5) = bothArmsStill(:,5)/jointScale

bothArmsOut(:,3) = bothArmsOut(:,3)/jointScale;
bothArmsIn(:,3) = bothArmsIn(:,3)/jointScale;
```

```
bothArmsStill(:,3) = bothArmsStill(:,3)/jointScale;

bothArmsOut(:,1:2) = bothArmsOut(:,1:2)/sensorScale;e;
bothArmsIn(:,1:2) = bothArmsIn(:,1:2)/sensorScale;
bothArmsStill(:,1:2) = bothArmsStill(:,1:2)/sensorScale;

bothArmsOut(:,6) = −bothArmsOut(:,6)/sensorScale;
bothArmsIn(:,6) = −bothArmsIn(:,6)/sensorScale;
bothArmsStill(:,6) = −bothArmsStill(:,6)/sensorScale;
```

runSequencesSimulation1.m

```
% This script is a program to run and collect data for the four
    sequences
% used in the first simulation.  The sensor and angle data for
    each sensor
% is stored in seperate variables.
%
% moveLeftArm1: raise the left arm
% moveLeftArm2: lower the left arm
% moveRightArm1: raise the right arm
% moveRightArm2: lower the right arm

test = javaObject('Breve');
timeSteps = 30;

% Set start position
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 −0.8 −1];
goalShoulderR = [1.5 0.8 1];
oneSequence(goalElbowL, goalElbowR, goalShoulderL, goalShoulderR
    , timeSteps)

% Set light ball to start position
lightBall = [−2 4.5 0];
test.read('localhost', 7776, sprintf('move−light␣%f␣%f␣%f',
    lightBall(1), lightBall(2), lightBall(3)));

% Pause for two seconds so that the user may get ready to move
    the ball
% manually
pause(2);

% Raise left arm
```

```
goalElbowL = 1;
goalElbowR = 0;
goalShoulderL = [0 -0.8 -1];
goalShoulderR = [1.5 0.8 1];
moveLeftArm1 = oneSequence(goalElbowL, goalElbowR, goalShoulderL
    , goalShoulderR, timeSteps);

% Lower left arm
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 -0.8 -1];
goalShoulderR = [1.5 0.8 1];
moveLeftArm2 = oneSequence(goalElbowL, goalElbowR, goalShoulderL
    , goalShoulderR, timeSteps);

% Reset ball's position to the center
lightBall = [-2 4.5 0];
test.read('localhost', 7776, sprintf('move-light_%f_%f_%f',
    lightBall(1), lightBall(2), lightBall(3)));

% Raise right arm
goalElbowL = 0;
goalElbowR = -1;
goalShoulderL = [1.5 -0.8 -1];
goalShoulderR = [0 0.8 1];
moveRightArm1 = oneSequence(goalElbowL, goalElbowR,
    goalShoulderL, goalShoulderR, timeSteps);

% Lower right arm
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 -0.8 -1];
goalShoulderR = [1.5 0.8 1];
moveRightArm2 = oneSequence(goalElbowL, goalElbowR,
    goalShoulderL, goalShoulderR, timeSteps);
```

runSequencesSimulation2.m

```
% This script is a program to run and collect data for the three
     sequences
% used in the second simulation.  The sensor and angle data for
    each sensor
% is stored in seperate variables.
%
% bothArmsOut: raise both arms
% bothArmsIn: lower both arms
% bothArmsStill: arms lowered

test = javaObject('Breve');
timeSteps = 30;

% Set start position
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 −0.8 −1];
goalShoulderR = [1.5 0.8 1];
oneSequence(goalElbowL, goalElbowR, goalShoulderL, goalShoulderR
    , timeSteps)

% Set light ball to start position
lightBall = [−2 4.5 0];
test.read('localhost', 7776, sprintf('move−light_%f_%f_%f',
    lightBall(1), lightBall(2), lightBall(3)));

% Pause for two seconds so that the user may get ready to move
    the ball
% manually
pause(2);

% Raise both arms
goalElbowL = 1;
goalElbowR = −1;
goalShoulderL = [0 −0.8 −1];
goalShoulderR = [0 0.8 1];
bothArmsOut = oneSequence(goalElbowL, goalElbowR, goalShoulderL,
     goalShoulderR, timeSteps);

% Lower both arms
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 −0.8 −1];
goalShoulderR = [1.5 0.8 1];
bothArmsIn = oneSequence(goalElbowL, goalElbowR, goalShoulderL,
    goalShoulderR, timeSteps);

% Keep both arms lowered
goalElbowL = 0;
goalElbowR = 0;
goalShoulderL = [1.5 −0.8 −1];
goalShoulderR = [1.5 0.8 1];
bothArmsIn = oneSequence(goalElbowL, goalElbowR, goalShoulderL,
    goalShoulderR, timeSteps);
```

oneSequence.m

```
% Function that uses the PID—controller to move the joints into
    desired
% angles.  It also supports automatic moving of the light ball
    in the X
% axis.
%
% Input:     — Goal angles for each joint
%            — Number of timesteps
%
% Output:    A matrix containing sensor values and angles of the
    joints for
%            each timestep in the run.

function dataRead = oneSequence(goalElbowL, goalElbowR,
    goalShoulderL, goalShoulderR, timeSteps);

% Make a communication object
communication = javaObject('Breve');


% Make one PID—controller for each angle that need to be
    controlled
pidLe = javaObject('PID', 6, 4, 4, 3);
pidRe = javaObject('PID', 6, 4, 3, 0);
pidR1 = javaObject('PID', 6, 4, 4, 1);
pidR2 = javaObject('PID', 6, 4, 4, 1);
pidR3 = javaObject('PID', 6, 4, 4, 1);
pidL1 = javaObject('PID', 6, 4, 4, 1);
pidL2 = javaObject('PID', 6, 4, 4, 1);
pidL3 = javaObject('PID', 6, 4, 4, 1);

% Set shoulder speed to zero
shoulderV = zeros(1,3);

for i = 1:timeSteps,

    % Get current left elbow angle
    elbowL = str2double(communication.read('localhost', 7776, '
        get—left—elbow—angle'));
    % Use PID—controller to find new speed
    v = pidLe.getNewVelocity(goalElbowL—elbowL, —1, 1);

    % Get current right elbow angle
    elbowR = str2double(communication.read('localhost', 7776, '
        get—right—elbow—angle'));
    % Use PID−controller to find new speed
    v2 = pidRe.getNewVelocity(goalElbowR−elbowR, −1, 1);

    % Get current right shoulder angles
    eval(communication.read('localhost', 7776, sprintf('get−
        right−arm−angles')));
    p = [X Y Z];
    shoulderXR = X; % Store for later use
    % Use PID−controller to find new shoulder speeds
    shoulderRV(1) = pidR1.getNewVelocity(goalShoulderR(1) − p(1)
        , −1, 1);
    shoulderRV(2) = pidR2.getNewVelocity(goalShoulderR(2) − p(2)
        , −1, 1);
    shoulderRV(3) = pidR3.getNewVelocity(goalShoulderR(3) − p(3)
        , −1, 1);

    % Get current left shoulder angles
    eval(communication.read('localhost', 7776, sprintf('get−left
        −arm−angles')));
    p = [X Y Z];
    shoulderXL = X; % Store for later use
    % Use PID−controller to find new shoulder speeds
    shoulderLV(1) = pidL1.getNewVelocity(goalShoulderL(1) − p(1)
        , −1, 1);
    shoulderLV(2) = pidL2.getNewVelocity(goalShoulderL(2) − p(2)
        , −1, 1);
    shoulderLV(3) = pidL3.getNewVelocity(goalShoulderL(3) − p(3)
        , −1, 1);

    % Send the new speeds to breve
    communication.read('localhost', 7776, sprintf('set−left−
        elbow−velocity %f', v));
    communication.read('localhost', 7776, sprintf('set−right−
        elbow−velocity %f', v2));
    pause(0.03);
    communication.read('localhost', 7776, sprintf('set−left−arm−
        velocities %f %f %f', shoulderLV(1), shoulderLV(2),
        shoulderLV(3)));
    communication.read('localhost', 7776, sprintf('set−right−arm
        −velocities %f %f %f', shoulderRV(1), shoulderRV(2),
        shoulderRV(3)));

    % Tell breve to iterate
    communication.read('localhost', 7776, 'matlab−iterate');

    % Read sensors from lightball
    eyeR = str2double(communication.read('localhost', 7776,
        sprintf('get−right−eye'))) / 10;
    eyeL = str2double(communication.read('localhost', 7776,
       sprintf('get−left−eye'))) / 10;

    % Store data from sensors and joint angles for current
       timestep
    dataRead(i,:) = [eyeL eyeR shoulderXL elbowL shoulderXR
       elbowR];
    pause(0.06);

end;
```

D.1.3 Interaction Phase

run$_t rained_r nnpb.m$

```
% Main script for the interaction phase.
% − Gets current state from breve,
% − Uses the RNNPB to decide the next state
% − Uses the PID to decide new velocities for each joint
% − Sends new velocities to breve
% (− Also updates PB vectors)
%
% NOTE: This function must be run after breve has been started,
   while
% the simulation "Tiny Dancer.tz" is running.  A trained RNNPB
   named "net"
% must also be present in the workspace.

% Create a java communication object
communication = javaObject('Breve');

% Set start angles
goalElbowL = 0;
goalElbowR = 0;

goalShoulderL = [1.5 −0.8 −1];
goalShoulderR = [1.5 0.8 1];


% Create java PID objects for each joint angle

% Left and right elbow angles
pidLe = javaObject('PID', 5, 3, 3, 1);
pidRe = javaObject('PID', 5, 3, 3, 1);
```

```
% Right shoulder angles
pidR1 = javaObject('PID', 5, 3, 3, 1);
pidR2 = javaObject('PID', 5, 3, 3, 1);
pidR3 = javaObject('PID', 5, 3, 3, 1);

% Left shoulder angles
pidL1 = javaObject('PID', 5, 3, 3, 1);
pidL2 = javaObject('PID', 5, 3, 3, 1);
pidL3 = javaObject('PID', 5, 3, 3, 1);

% Initialize variables that hold PB errors
dPB1 = zeros(1, net.size{5}, net.size{4});
ePB = zeros(1,net.size{5});
ePB_1 = zeros(1,net.size{5});
ePB_2 = zeros(1,net.size{5});
dpb = zeros(1, net.size{5});
pbError = zeros(net.size{5}, net.size{2});

% Initialize PB vectors
pb = 0.5*ones(1, net.size{5});
pbused = pb;

% Initialize context nodes
contexNodes = 0.5 * ones(1, net.size{3});

% Set the light ball's start position
lightBall = [-3 4.5 0];

% Send message to breve with light ball's position
communication.read('localhost', 7776, sprintf('move-light_%f_%f_
    %f', lightBall(1), lightBall(2), lightBall(3)));

for i = 1:500,

    % Read current left elbow angle from breve
    elbowL = str2double(communication.read('localhost', 7776, '
        get-left-elbow-angle'));
    % Send error to left elbow PID, and get new velocity
    v = pidLe.getNewVelocity(goalElbowL-elbowL, -1, 1);

    % Read current right elbow angle from breve
    elbowR = str2double(communication.read('localhost', 7776, '
        get-right-elbow-angle'));
  % Send error to right elbow PID, and get new velocity
    v2 = pidRe.getNewVelocity(goalElbowR-elbowR, -1, 1);

    % Read current right shoulder angles from breve
    eval(communication.read('localhost', 7776, sprintf('get-
        right-arm-angles')));
    p = [X Y Z];
    shoulderXR = X; % Store X—angle for later use
    % Send error to each right shoulder angle PID, and get new
        velocities
    shoulderRV(1) = pidR1.getNewVelocity(goalShoulderR(1) — p(1)
        , —1, 1);
    shoulderRV(2) = pidR2.getNewVelocity(goalShoulderR(2) — p(2)
        , —1, 1);
    shoulderRV(3) = pidR3.getNewVelocity(goalShoulderR(3) — p(3)
        , —1, 1);

    % Read current left shoulder angles from breve
    eval(communication.read('localhost', 7776, sprintf('get—left
        —arm—angles')));
    p = [X Y Z];
    shoulderXL = X;% Store X—angle for later use
    % Send error to each left shoulder angle PID, and get new
        velocities
    shoulderLV(1) = pidL1.getNewVelocity(goalShoulderL(1) — p(1)
        , —1, 1);
    shoulderLV(2) = pidL2.getNewVelocity(goalShoulderL(2) — p(2)
        , —1, 1);
    shoulderLV(3) = pidL3.getNewVelocity(goalShoulderL(3) — p(3)
        , —1, 1);

    % Send new velocities to breve
    communication.read('localhost', 7776, sprintf('set—left—
        elbow—velocity_%f', v));
    communication.read('localhost', 7776, sprintf('set—right—
        elbow—velocity_%f', v2));
    communication.read('localhost', 7776, sprintf('set—left—arm—
        velocities_%f_%f_%f', shoulderLV(1), shoulderLV(2),
        shoulderLV(3)));
    communication.read('localhost', 7776, sprintf('set—right—arm
        —velocities_%f_%f_%f', shoulderRV(1), shoulderRV(2),
        shoulderRV(3)));

    % Tell breve to iterate the simulation
    communication.read('localhost', 7776, 'matlab—iterate');

    % Read sensors:
    eyeR = str2double(communication.read('localhost', 7776,
        sprintf('get—right—eye'))) / 5;
    eyeL = str2double(communication.read('localhost', 7776,
        sprintf('get—left—eye'))) / 5;

    % Pause for 0.1 seconds (this is so that breve runs smoothly
        )
    pause(0.1);
    currentPattern = [eyeL eyeR shoulderXL/1.5 elbowL shoulderXR
       /1.5 elbowR];

    % Activate — we now have a new set of desired goals
    net = rnnpb_activate(net, currentPattern, pbused);

    % Set new elbow and shoulder goals  (which will be used by
       the PID in
    % the next step)
    goalElbowL = net.L{3}(4);
    goalElbowR = —net.L{3}(6);
    goalShoulderL = [net.L{3}(3)*1.5 —0.8 —1];
    goalShoulderR = [net.L{3}(5)*1.5 0.8 1];


    % Find errors so that the PB vectors can be updated
    %
    % NOTE: net.L{3} is the third layer in the network: in other
    % words, the predicted state from the activation, which we
       now want to
    % be the target for the next step. The error is thus
       measured by
    % the difference between the target and current state.
    oError = net.L{3}.*(1 — net.L{3}).*(net.L{3} —
       currentPattern);
    hError = oError * net.w{1}' .* net.L{2} .* (1—net.L{2});

    % Find PB error
    for pbnr=1:net.size{5},
        pbError(pbnr,:) = hError .* net.w{2}(net.size{1}+pbnr,:)
            .* (1—net.w{2}(net.size{1}+pbnr,:));
    end;

    ePB = —sum((net.w{2}(net.size{1}+1:net.size{1}+net.size
       {5},:) .* pbError)');
    dpb = dpb + ePB;

    % Integrate using Simpson's rule
     pb = pb + 0.01*(dpb + 4*ePB_1 + ePB_2);
     ePB_2 = ePB_1;
     ePB_1 = dpb;

    % Sigmoid function — the PB vector used in the next step
    pbused = 1./(1 + exp(—pb))

end;
```

D.2 Java Files

Breve.java

```java
/**
 * Class for communicating with the breve simulator.
 *
 * @author Axel Tidemann
 */

import java.io.*;
import java.net.*;

public class Breve {

    /*
     * The method reads from the breve simulator by requesting a
        file on
     * breve's web server.
     */
    public static String read(String host, int port, String msg)
    {
  try{
      //connect to the server via the given IP address and port
         number
      Socket fSocket = new Socket (host, port);

      //the request
      String message = "GET /" + msg + "\n";

      //output writer
      PrintWriter pw_server_out =
    new PrintWriter(new BufferedWriter(new OutputStreamWriter (
       fSocket.getOutputStream ())), true );

      //send the request
      pw_server_out.println (message);

      //input stream
      BufferedReader server_reader =
    new BufferedReader (new InputStreamReader (
       fSocket.getInputStream () )  );

      //read the reply from breve
      String response = server_reader.readLine();
      fSocket.close();

      return response;

  } catch  ( Exception uhe) {}

  return "failure.";
    }
}
```

PID.java

```
/*
 * PID-controller
 * Upon construction, this class receives parameters for
 * a PID- controller. An error can then be sent to the
 * object, and a new velocity to reduce the error will be
    returned.
 * The two previous erros are stored in the object so that
    derivation
 * and integration of the error is possible (this is done in the
     formula).
 */

public class PID
{

  private double v;
  public double K;
  public double Ti;
  public double Ts;
  public double Td;
  public double v_1 = 0;
  public double error = 0;
  public double error_1 = 0;
  public double error_2 = 0;

  // Constructor
  public PID (double K, double Ts, double Ti, double Td)
  {
    this.K = K;
    this.Ti = Ti;
    this.Ts = Ts;
    this.Td = Td;
  }
  /*
   * Receives an error, and a max and min value.
   * Returns a velocity (in the bounds of max and min) that
      should
   * reduce the error.
   */
  public double getNewVelocity(double error, double min, double
     max)
  {
    v = v_1 + K*((error - error_1) + (Ts/Ti) * error + (Td/Ts)*(
       error - 2*error_1 + error_2));

    if (v < min)
      v = min;
    else if (v > max)
      v = max;

    v_1 = v;
    error_2 = error_1;
    error_1 = error;

    return v;
  }

}
```

D.3 breve Files

Tiny Dancer.tz

```
#
# Tiny Dancer - originally created by Axel Tidemann,
but modified for my own use.
# - Firas R. Barakat

# Imports
@use PhysicalControl.
@use Link.
@use Genome.
@use Shape.
@use Stationary.
```

```
@use MultiBody.
@use NetworkServer.
@use Braitenberg.
@use Movie.

# Create controller (called Elton)
Controller Elton.

# Controller
PhysicalControl : Elton {
    + variables:
        tiny_dancer, cloudTexture (object).
        X, Y, Z (double).
        lightBall (object).

    + to init:
        floor (object).
        myNetworkServer (object).

        #initialize network server
        myNetworkServer = new NetworkServer.
        myNetworkServer listen on-port 7776.
        print (myNetworkServer get-url).
        #end init network server

        # Initialize movie recording
       movie = (new Movie).

        # Varios initialization for visualization
        self set-random-seed-from-dev-random.
        self enable-lighting.
        self enable-smooth-drawing.
        self move-light to (0, 20, 0).
         cloudTexture = (new Image load from
"./lib/classes/images/clouds.png").

        # Create the floor
        floor = new Floor.
        floor catch-shadows.
        floor set-color to (1.0, 1.0, 1.0).
        floor set-eT to .9.
        self enable-shadow-volumes.
        self enable-reflections.

        self set-background-color to (.4, .6, .9).
        self set-background-texture-image to cloudTexture.
        # Visualization end

      # Create the Creature (the robot)
        tiny_dancer = new Creature.
        self init-position.

        # Create light ball
        lightBall = new Light.
        lightBall move to (-3, 4.5, 0).

    #this should be done by sharing instance variables of some sort.
    + to init-position:

     #   tiny_dancer move to (0, 6, 0).
        self offset-camera by (3, 13, -13).
        self watch item tiny_dancer.

    # Iterate from matlab
    + to matlab-iterate:
            self manual-iterate.

    # Returns angles for elbows and shoulders
    # (methods can be invoked from matlab)
    + to get-right-elbow-angle:
            return (tiny_dancer get-right-elbow-angle).

    + to get-left-elbow-angle:
            return (tiny_dancer get-left-elbow-angle).

    + to get-right-arm-angles:
        X = (tiny_dancer get-right-arm-angles)::x.
        Y = (tiny_dancer get-right-arm-angles)::y.
        Z = (tiny_dancer get-right-arm-angles)::z.
        return "X = $X, Y = $Y, Z = $Z".
    + to get-left-arm-angles:
        X = (tiny_dancer get-left-arm-angles)::x.
        Y = (tiny_dancer get-left-arm-angles)::y.
        Z = (tiny_dancer get-left-arm-angles)::z.
        return "X = $X, Y = $Y, Z = $Z".

    # Returns the position of the light ball
    + to get-light-position:
        X = (lightBall get-location)::x.
        Y = (lightBall get-location)::y.
        Z = (lightBall get-location)::z.
        return "X = $X, Y = $Y, Z = $Z".

    # Returns the sensor values
    + to get-left-eye:
        return (tiny_dancer get-left-eye).

    + to get-right-eye:
        return (tiny_dancer get-right-eye).

    # Sets joint velocities for elbows and shoulders
    # (methods can be invoked from matlab)
    + to set-right-elbow-velocity to-value v = 0 (double):
        tiny_dancer right-elbow velocity v.

    + to set-left-elbow-velocity to-value v = 0 (double):
        tiny_dancer left-elbow velocity v.

    + to set-right-arm-velocities to-value x (double)
to-value y(double) to-value z (double):
        tiny_dancer right-shoulder velocities (x, y, z).

    + to set-left-arm-velocities to-value x (double)
to-value y(double) to-value z (double):
        tiny_dancer left-shoulder velocities (x, y, z).


    # Move the light
    # (method can be invoked from matlab)
    + to move-light to-value x (double) to-value y(double)
to-value z (double):
        lightBall move to (x, y, z).


    # Catching keys  (For breve testing only)
    ### Right arm  start ###
    + to catch-key-y-down:
          tiny_dancer right-shoulder velocities (1, 0, 0).

    + to catch-key-y-up:
            tiny_dancer right-shoulder velocities (0, 0, 0).

    + to catch-key-h-down:
           tiny_dancer right-shoulder velocities (-1, 0, 0).

    + to catch-key-h-up:
            tiny_dancer right-shoulder velocities (0, 0, 0).

    + to catch-key-g-down:
           tiny_dancer right-shoulder velocities (0, 1, 0).

    + to catch-key-g-up:
           tiny_dancer right-shoulder velocities (0, 0, 0).

    + to catch-key-j-down:
            tiny_dancer right-shoulder velocities (0, 0, -1).

    + to catch-key-j-up:
            tiny_dancer right-shoulder velocities (0, 0, 0).

    # Right arm elbow
    + to catch-key-t-down:
            tiny_dancer right-elbow velocity 1.

    + to catch-key-t-up:
            tiny_dancer right-elbow velocity 0.

    + to catch-key-u-down:
            tiny_dancer right-elbow velocity -1.

    + to catch-key-u-up:
            tiny_dancer right-elbow velocity 0.
```

```
###  Right arm end ###

### Left arm start ###
+ to catch-key-w-down:
         tiny_dancer left-shoulder velocities (1, 0, 0).

+ to catch-key-w-up:
        tiny_dancer left-shoulder velocities (0, 0, 0).

+ to catch-key-s-down:
        tiny_dancer left-shoulder velocities (-1, 0, 0).

+ to catch-key-s-up:
        tiny_dancer left-shoulder velocities (0, 0, 0).

+ to catch-key-a-down:
        tiny_dancer left-shoulder velocities (0, 1, 0).

+ to catch-key-a-up:
        tiny_dancer left-shoulder velocities (0, 0, 0).

+ to catch-key-d-down:
        tiny_dancer left-shoulder velocities (0, -1, 0).

+ to catch-key-d-up:
        tiny_dancer left-shoulder velocities (0, 0, 0).

# Left arm elbow
+ to catch-key-q-down:
        tiny_dancer left-elbow velocity 1.

+ to catch-key-q-up:
        tiny_dancer left-elbow velocity 0.

+ to catch-key-e-down:
        tiny_dancer left-elbow velocity -1.

+ to catch-key-e-up:
        tiny_dancer left-elbow velocity 0.
### Left arm end ###
```

```
    # Record movie
    + to catch-key-m-down:
        testFile (string).
        testFile = "breveMovie.mpg".
        movie record to testFile.

}

# Create an arm
MultiBody : Arm {
    + variables:
        upperLink (object).
        elbowJoint, wristJoint, worldJoint (object).
        upperShape, lowerShape, handShape (object).
        lowerLink, handLink (object).

    + to get-root:
         return upperLink.

    + to init:

        upperShape = (new Cube init-with size (0.5, 1, 0.5)).
        lowerShape = upperShape.
        handShape = (new Sphere init-with radius 0.3).

        #create the links, give them their shape.
        upperLink = new Link.
        upperLink set-shape to upperShape.
        lowerLink = new Link.
        lowerLink set-shape to lowerShape.
        handLink = new Link.
        handLink set-shape to handShape.

        #connect the links
        elbowJoint = new RevoluteJoint.
        elbowJoint link parent upperLink to-child lowerLink
            with-normal (1,0,0)
            with-parent-point (0, 0.5, 0)
            with-child-point (0, -0.5, 0).
        wristJoint = new FixedJoint. #UniversalJoint #for now.
        wristJoint link parent lowerLink to-child handLink
            with-parent-point (0, 0.5, 0)
            with-child-point (0, -0.3, 0).

        elbowJoint set-strength-limit to 100.

       # Sets velocity
        + to elbow velocity v = 0 (double):
            elbowJoint set-joint-velocity to v.

        # Returns angle
    + to getElbowJointAngle:
        return (elbowJoint get-joint-angle).

    }

# Creates a leg
MultiBody : Leg {
    + variables:
        upperLink (object).
        kneeJoint, footJoint (object).

    + to get-root:
         return upperLink.

    + to init:
        upperShape, lowerShape, footShape (object).
        lowerLink, footLink (object).

        upperShape = (new Cube init-with size (0.5, 1, 0.5)).
        lowerShape = upperShape.
        footShape = (new Cube init-with size (1, 0.4, 0.4)).

        #create the links, give them their shape.
        upperLink = new Link.
        upperLink set-shape to upperShape.
        lowerLink = new Link.
        lowerLink set-shape to lowerShape.
        footLink = new Link.
        footLink set-shape to footShape.
        #connect the links
        kneeJoint = new RevoluteJoint.
        kneeJoint link parent upperLink to-child lowerLink
            with-normal (1,0,0)
            with-parent-point (0, 0.5, 0)
            with-child-point (0, -0.5, 0).

        footJoint = new UniversalJoint.
        footJoint link parent lowerLink to-child footLink
            with-normal (1,1,0)
            with-parent-point (0, 0.5, 0)
            with-child-point (-0.25, -0.2, 0).

        kneeJoint set-strength-limit to 100.
        footJoint set-strength-limit to 2.

    }


# Creates a head with sensors
MultiBody : Head {

    + variables:
        headShape (object).
        leftSensor, rightSensor (object).
        headLink (object).
        leftSensorJoint, rightSensorJoint (object).
        leftSensorShape (object).
        rightSensorShape (object).

    + to init:
        headShape = (new Sphere init-with radius 0.5).
        headLink = new Link.
        headLink set-shape to headShape.

            rightSensorShape = new Shape.
            rightSensorShape init-with-polygon-cone
radius .1 sides 5 height .2.
            rightSensor = new Sensor.
            rightSensor set-shape to rightSensorShape.
            leftSensorShape = new Shape.
            leftSensorShape init-with-polygon-cone
radius .1 sides 5 height .2.
            leftSensor = new Sensor.
            leftSensor set-shape to leftSensorShape.


        leftSensorJoint = new FixedJoint.

       leftSensorJoint set-relative-rotation around-axis
(0.6, 0, 1) by 1.54.
       leftSensorJoint link parent headLink to-child leftSensor
            with-parent-point (-0.4,0.1,0.3)
            with-child-point (0, 0, 0).

        leftSensorJoint set-strength-limit to 100.


        rightSensorJoint = new FixedJoint.

       rightSensorJoint set-relative-rotation around-axis
(-0.6, 0, 1) by 1.54.
       rightSensorJoint link parent headLink to-child rightSensor
            with-parent-point (-0.4,0.1,-0.3)
            with-child-point (0, 0, 0).

        rightSensorJoint set-strength-limit to 100.

    # Returns the link to the head
    + to get-root:
        return headLink.

    # Returns right sensor value
    + to get-right-eye:
        return (rightSensor get-sensor-value).

    # Returns left sensor value
    + to get-left-eye:
        return (leftSensor get-sensor-value).
```

```
}

# Creates a light ball
Mobile : Light (aka Lights) {
+ to init:
self set-shape to (new Shape init-with-sphere radius .3).
self set-color to (1, 0, 0).
}

# A slightly modified copy of the BreitenbergSensor in Breitenberg.tz
Link : Sensor {

        + variables:
            sensorShape (object).
            link (object).
            direction (vector).
            sensorAngle (float).
            sensorValue (float).
            activationObject (object).
            activationMethod (string).
            bias (float).

        + to init:
            bias = 1.0.
            direction = (0, 1, 0).
            sensorAngle = 1.2.
            self set-color to (0, 0, 0).

        + to get-root:
            return link.


        + section "Configuring the Sensor Values"

+ to set-bias to d (float):
% Sets the "bias" of this sensor.  The default bias is 1, meaning
% that the sensor has a positive influence on associated wheels
% with strength 1.  You can change this to any magnitude, positive
% or negative.

bias = d.
```

```
+ to set-sensor-angle to n (float):
% Sets the angle in which this sensor can detect light.  The default
% value of 1.5 means that the sensor can see most of everything in
% front of it.  Setting the value to be any higher leads to general
% wackiness, so I don't suggest it.

sensorAngle = n.

+ to set-activation-method to m (string) in-object o (object):
% This method specifies an activation method for the sensor.  An
% activation method is a method which takes as input the strength
% read by the sensor, and as output returns the strength of the
% signal which will travel on to the motor.
% <p>
% Your activation function should be defined as:
% <pre>
%     + to <i>activation-function-name</i> with-sensor-strength s (float):
% </pre>
% <p>
% The default activation method is linear, but more complex vehicles
% may require non-linear activation functions.
%

activationMethod = m.
activationObject = o.

        + to get-sensor-value:
                return sensorValue.

+ to iterate:
i (object).
lights (int).
total, strength, angle (float).
toLight, transDir (vector).

transDir = (self get-rotation) * direction.

foreach i in (all Lights): {
toLight = (i get-location) - (self get-location).
angle = angle(toLight, transDir).
if angle < sensorAngle: {
                strength = | (self get-location) - (i get-location) |.
strength = 1.0 / (strength * strength) .

if activationMethod && activationObject: {
strength = (activationObject call-method named
activationMethod with-arguments { strength }).
}

                if strength > 10: strength = 10.

total += strength.

lights++.
}
}

if lights != 0: total /= lights.

total = 50 * total * bias.
                sensorValue = total.
}


# Creates a Creature (the full robot)
MultiBody : Creature {
    + variables:
        bodyLink (object).
        links (list).
        joints (list).
        worldJoint (object).
        leftArm, leftArmJoint, rightArm, rightArmJoint (object).
        leftLeg, leftLegJoint, rightLeg, rightLegJoint (object).
        head (object).
        testJoint (object).
        sensorTest (object).

    + to get-root:
        return bodyLink.
```

```
+ to radians from degrees (float):
    return degrees * 3.14 / 180.

+ to init:
    #the body
    bodyShape, headLink, headJoint (object).

    bodyShape = (new Cube init-with size (0.5, 2, 1)).

    bodyLink = new Link.
    bodyLink set-shape to bodyShape.

    # the head
    head = new Head.
    headJoint = 1 new FixedJoint.
    headJoint link parent bodyLink to-child (head get-root)
        with-parent-point (0,1,0)
        with-child-point(0,-0.5,0) .

    # Lock the head to a fixed point in space
    worldJoint = new FixedJoint.
    worldJoint link parent 0 to-child (head get-root)
         with-parent-point (0, 4, 0)
         with-child-point (0, -1.0, 0).


    #the arms.
    leftArm = new Arm.
    leftArmJoint = new BallJoint.
    leftArmJoint link parent bodyLink to-child (leftArm get-root)
        with-normal (0,0,1)
        with-parent-point (0,1,0.5)
        with-child-point (0, -0.5, -0.25).

    rightArm = new Arm.
    rightArmJoint = new BallJoint.
    rightArmJoint link parent bodyLink to-child (rightArm get-root)
        with-normal (0,0,1)
        with-parent-point (0,1,-0.5)
        with-child-point (0, -0.5, 0.25).
        #the legs.
        leftLeg = new Leg.
        leftLegJoint = new BallJoint.
        leftLegJoint link parent bodyLink to-child (leftLeg get-root)
            with-normal (1,0,1)
            with-parent-point (0, -1, -0.25)
            with-child-point (0, -0.5, 0).


        rightLeg = new Leg.
        rightLegJoint = new BallJoint.
        rightLegJoint link parent bodyLink to-child (rightLeg get-root)
            with-normal (1,0,1)
            with-parent-point (0, -1, 0.25)
            with-child-point (0, -0.5, 0).

        self set-root to bodyLink.

        (self get-all-connected-links) set-color to
random[(1.0, 1.0, 1.0)].

        #self show-axis.

        #add-dependency is for archiving. not that important.

        joints set-double-spring with-strength 400
with-max .8 with-min -.8.
        joints set-strength-limit to 300.

        leftArmJoint set-strength-limit to 100.
        rightArmJoint set-strength-limit to 100.
        leftLegJoint set-strength-limit to 3.
        rightLegJoint set-strength-limit to 3.


    + to center:
        # to center the object, we set the X and Z
        # components to 0, but not the Y, otherwise
        # we would push the walker into the ground

        currentLocation (vector).
        currentLocation = (self get-location).
        self move to (0, currentLocation::y, 0).

    + to destroy:
        free links.
        free bodyLink.


    # Manipulate right arm
          #Move right shoulder joint.
    + to right-shoulder velocities v = (0, 0, 0) (vector):
        rightArmJoint set-joint-velocity to v.

    # Elbows
        #Move right elbow.
    + to right-elbow velocity v = 0 (double):
        rightArm elbow velocity v.

    # Manipulate left arm
    #Move left shoulder joint.
    + to left-shoulder velocities v = (0, 0, 0) (vector):
        leftArmJoint set-joint-velocity to v.

    # Elbows

        #Move left elbow.
    + to left-elbow velocity v = 0 (double):
        leftArm elbow velocity v.


    # Returns the angles for the arms and elbows
    + to get-left-arm-angles:
        return (leftArmJoint get-joint-angles).

    + to get-right-arm-angles:
        return (rightArmJoint get-joint-angles).

    + to get-left-elbow-angle:
        return (leftArm getElbowJointAngle).
    + to get-right-elbow-angle:
        return (rightArm getElbowJointAngle).

    # Returns the values from the light sensors (eyes)
    + to get-right-eye:
        return (head get-right-eye).

    + to get-left-eye:
        return (head get-left-eye).

}
```

Bibliography

[1] Karl Johan Åström and Tore Hägglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, North Carolina, 1995.

[2] Firas Risnes Barakat. Learning by imitation. Depth Study, 2005.

[3] Cynthia Breazeal and Brian Scassellati. Robots that imitate humans. *Trends in Cognivite Sciences*, 16(11), 2002.

[4] Martin V. Butz and Sylvian Ray. Bidirectional artmap: An artificial mirror neuron system. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, July 2003.

[5] Robert Callan. *The Essence of Neural Networks*. Prentice Hall, 1999.

[6] Canonical. Ubuntu linux. http://www.ubuntu.com/.

[7] Ryding Stenberg Decety, Sjoholm and Ingvar. The cerebellum participates in mental activity: tomographic measurements of regional cerebral blood flow. *Brain Res.*, 1990.

[8] J. Demiris and G. Hayes. A robot controller using learning by imitation, 1994.

[9] John Demiris and Gillian Hayes. Imitation as a dual-route process featuring predictive and learning components: a biologically plausible computational model. *Imitation in animals and artifacts*, pages 327–361, 2002.

[10] Yiannis Demiris and Bassam Khadhouri. Hierarchical, attentive multiple models for execution and recognition. In *Proceedings of the IEEE ICRA Workshop on Robot Programming by Demonstration*, 2005.

[11] S. Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42:335–346, 1990.

[12] Lasse Holmstrom and Petri Koistinen. Using additive noise in back-propagation training. *Neural Networks, IEEE Transactions*, 3:24–38, Jan 1992.

[13] Sasaki Y. Takino R. Putz B. Kawato M. Imamizu H., Miyauchi S. Separated modules for visuomotor control and learning in the cerebellum: a functional mri. *NeuroImage*, 1997.

[14] Masato Ito and Jun Tani. Generalization in learning multiple temporal patterns using rnnpb. In *ICONIP*, pages 592–598, 2004.

[15] Masato Ito and Jun Tani. On-line imitative interaction with a humanoid robot using a dynamic neural network model of a mirror system. *Adaptive Behavior*, 2004.

[16] Matthew Johnson and Yiannis Demiris. Abstraction in recognition to solve the correspondence problem for robot imitation. In *Proceedings of TAROS*, 2004.

[17] M. Jordan and D. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354, 1992.

[18] M. Kawato. Feedback-error-learning neural network for supervised motor learning. *Advanced Neural Computers*, 1990.

[19] Jon Klein. breve simulation environment. http://www.spiderland.org/breve/index.php.

[20] Chris Klink. The mirror in the mind: The role of mirror neurons in self-consciousness, empathy and the evolution of language.

[21] Yukiko Hoshino Masato Ito, Kuniaki Noda and Jun Tani. Dynamic and interactive generation of object handling behaviors by a small humanoid robot using a dynamic neural network model. *In press*, 2006.

[22] Mathworks. Matlab. http://www.mathworks.com/products/matlab/.

[23] W. T. Miller. Sensor-based control of robotic manipulators using a general learning algorithm. *IEEE J. Robot. and Auto.*, 3(2):157–165, 1987.

[24] Domenico Parisi. Artificial life and higher level cognition. *Brain and Cognition*, 34:160–184, 6 1997.

[25] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3:233–242, 1999.

[26] Rajeshwar Prasad Srivastava. Use of genetic algorithms for optimization in digital control of dynamic systems. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 219–224, New York, NY, USA, 1992. ACM Press.

[27] Yuuya Sugita and Jun Tani. Learning semantic combinatoriality from the interaction between linguistic and behavioral processes. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems*, 13(1):33–52, 2005.

[28] M. Tham. Discretised pid controllers, 1996-1998. Retrieved May 1, 2006 http://lorien.ncl.ac.uk/ming/digicont/digimath/dpid1.htm.

[29] D. M. Wolpert and M. Kawato. Multiple paired forward and inverse models for motor control. *Neural Networks*, 11(7-8):1317–1329, 1998.

[30] M. Zhuang and D.P. Atherton. Automatic tuning of optimum pid controllers. *Control Theory and Applications, IEE Proceedings*, 1993.