IMPLEMENTATION OF A REAL-TIME LIQUID STATE MACHINE ON SPiNNaker FOR BIOMIMETIC ROBOT CONTROL

MASTER THESIS

submitted by
Nicolas Berberich

born on: 06.02.1993

Mitterweg 21
85748 Garching
Tel.: 0160/94797463

NEUROSCIENTIFIC SYSTEM THEORY
Technische Universität München

Prof. Dr Jörg Conradt

Supervisor: Prof. Dr Jörg Conradt
Advisor: PD Dr. Florian Röhrbein
Start: 17.10.2016
Intermediate Report: 25.01.2017
Final Submission: 30.09.2017
I confirm that this master’s thesis in electrical engineering and information technology is my own work and I have documented all sources and material used.

Munich, 30.09.2017
Nicolas Berberich
Acknowledgments

First and foremost, I would like to thank my advisor, PD Dr. Florian Röhrbein for his valuable insight, guidance and support. The Neurorobotics Seminar which he organizes weekly as well as the Lecture Series in Neurorobotics have been very helpful for gaining a better grasp and overview of the field in order to situate my thesis project. I’m especially thankful to him for encouraging me to participate in the Interdisciplinary College on Creativity and Intelligence in Brains and Machines and present my work in the form of a poster.

I’m equally thankful to my supervisor and mentor within the Neuroengineering Master Program, Prof. Dr. Jörg Conradt, who sparked my interest for neuroengineering and neurorobotics several semesters ago and keeps guiding me to new exciting topics within these fields.

I would also like to acknowledge and thank the fantastic Roboy team, especially Rafael Hofstettler, the team leader, and Simon Trendel for helping me set up the FPGA communication between ROS and the Myorobotic Arm. I’m also very grateful to Nicolas Hink for his collaboration on adding tendon support to my virtual robot model. Further thanks goes to Luis Vergara, Benedikt Feldotto, Anna Sesselmann, Alona Kharchenko and Alexander Kuhn.

Furthermore, I would like to thank the Human Brain Project, especially theNeurorobotics Subproject for allowing me to participate in their great endeavor.

Special thanks goes to the Studienstiftung des deutschen Volkes not only for their financial support but also for the wonderful summer schools and community that I dearly enjoyed during my studies.

Last but certainly not least, I would like to thank my family for their love and support.
Abstract

Musculoskeletal robots promise compliant and save human-robot interaction, but suffer from being hard to control due to inherent nonlinearities. Instead of using traditional control strategies, we aim at building a closed-loop neurorobotic system inspired by the Liquid State Machine concept for computation in cortical microcircuits. A Liquid State Machine consists of a recurrent spiking neural network designed according to the principle of reservoir computing which harnesses its rich dynamics for learning a mapping between a given input to a desired output stream. Our approach is to build a system which allows us to run an Liquid State Machine on either the SpiNNaker neuromorphic hardware or on a traditional PC with NEST as back-end. We build new and refined existing interfaces between the neural input and output and the Robot Operating System, which is used for the communication with a single-joint robot based on the Myorobotics framework. Additionally, a virtual model of the robot was constructed that can also be steered by our spiking neural network. Our system offers high modularity and scalability, with a selectable level of real-time capability, based on the chosen neural simulation platform and robot model. Finally, we discuss similarities between our neurorobotic system and the human sensorimotor system.
# Contents

Acknowledgments iii

Abstract v

1 Introduction 1
   1.1 Motivation: Neurorobotics 1
   1.2 Project Overview and Objectives 2
   1.3 Thesis Overview 2

2 Spiking Neural Networks 5
   2.1 Models of Spiking Neurons 6
      2.1.1 Action Potential 8
      2.1.2 Synaptic Transfer 10
      2.1.3 Connectionism: From Artificial to Spiking Neural Networks 10
   2.2 Neural Simulators 12
      2.2.1 NEST 13
      2.2.2 PyNN 13
   2.3 Liquid State Machine 14

3 Neuromorphic Hardware 17
   3.1 Neuromorphic Engineering 17
   3.2 The SpiNNaker Project 21

4 Biomimetic Robotics 25
   4.1 Biomimetic and Musculoskeletal Robotics 25
   4.2 The Myorobotics Project 26

5 Implementation: Tools, Interfaces and Components 29
   5.1 Tools 30
      5.1.1 PyNN and NEST 31
      5.1.2 The Robot Operating System (ROS) and Gazebo 32
      5.1.3 The Neurorobotics Platform 33
      5.1.4 Github 34
Contents

5.2 Interfaces ................................................. 36
  5.2.1 ROS-NEST Interface ................................. 36
  5.2.2 ROS-SpiNNaker Interface ......................... 36
5.3 Components .............................................. 41
  5.3.1 SpiNNaker 4-Chip Board ......................... 41
  5.3.2 Myorobotics Single-joint Arm ...................... 41

6 Implementation: System Architecture 49
  6.1 Robot Model ............................................. 49
  6.2 Joint Sensor ............................................ 49
  6.3 Spike Encoding ......................................... 50
  6.4 Neural Network ......................................... 53
    6.4.1 Stability ........................................... 54
  6.5 Learning ............................................... 62
  6.6 Spike Decoding ......................................... 65
  6.7 Transfer Function ....................................... 68

7 Discussion 71
  7.1 Neuroscience: Insight and Discussion ............... 71
    7.1.1 Comparison to human motor control ............. 71
    7.1.2 The LSM as a model for computation in cortical microcolumns 73
    7.1.3 Insight into spike encoding, stable reservoirs and spike decoding 74
  7.2 Robotics: Insight and Discussion ..................... 75
    7.2.1 Modularity ......................................... 75
    7.2.2 Scalability ......................................... 75
    7.2.3 Real-Time Performance ............................. 76
    7.2.4 Further lessons learned .......................... 76

8 Conclusion ................................................. 79

List of Figures .............................................. 82

List of Tables ............................................... 84

Bibliography ................................................ 85
1 Introduction

1.1 Motivation: Neurorobotics

The field of neurorobotics lies in the center of the convergence of robotics, artificial intelligence and neuroscience. The central idea of neurorobotics is to use robots to test brain models and on the other hand to use principles from neuroscience to make robots smarter, more flexible and more energy efficient.

Neuroscientific objective  A neurorobot as a model for physically embedded brains can be seen as an emergent cognitive system. As such, due to the goal of being biologically plausible, it can’t be directly programmed to perform certain predefined tasks. Instead, their architecture should be designed in a way that allows it learn those tasks. There are different computational models of the human or animal sensorimotor system and the best way to test them is by following the synthetic methodology of implementing them in robotic systems and observing their behavior.

Robotics & engineering objective  The field of robotics is currently shifting from static, purely industrial applications towards systems capable of dynamic interactions with humans and systems with a higher degree of mobility. The design of these new types of robots necessitates very different engineering requirements. Firstly, they need to be able to learn and adapt to new environments and tasks. Secondly, they need to be inherently safe. This is achieved in musculoskeletal robots which consist of bones, muscles and tendons similar to the human or animal body and are thus additionally more accurate models for neuroscientific sensorimotor research. For reasons presented in Chapter 4 (e.g. inherent nonlinearities), the direct programming of musculoskeletal robots is extremely difficult. A possible solution is to learn and approximate the robot’s internal nonlinear model with neural networks. State-of-the-art neural networks, however, have low biological plausibility, which would harm our neuroscientific objective and are run on energy-hungry GPUs, ruining the power-efficiency requirement needed for mobile robotic systems. The alternative, which we aim to work towards, is using spiking neural networks, which are in theory computationally equivalent to traditional artificial neural networks, and can be run on energy efficient neuromorphic hardware.
1 Introduction

1.2 Project Overview and Objectives

The system that we implemented as part of this thesis is aimed at simultaneously giving justice to both these objectives. As a computational brain model we used a type of reservoir computing called Liquid State Machine (LSM) which was proposed by Wolfgang Maass, Thomas Natschläger and Henry Markram in 2002 (Maass et al., 2002). For our robotic system we used a single-joint Myorobotic arm.

Our neuroscientific goal is to test how the LSM model can be implemented as a spiking neural network. We are especially interested on how the highly recurrent reservoir consisting of excitatory and inhibitory neurons must be designed to be stable and how spike encoding and decoding interfaces between the neural network and the robot can be build. This forms a necessary basis for testing if the LSM model is capable of controlling a musculoskeletal robot like the Myorobotic arm that we used in this project.

Our engineering objective is to implement a closed-loop system that can control in real-time both the physical Myorobotic arm as well as to construct a virtual model of the arm that can be simulated and controlled by the same system. We furthermore aim at using traditional tools from the fields of neuroscience (PyNN) and robotics (Robot Operating System (ROS)) as interfaces between our subsystems to promote re-usability and to allow a future integration into the HBP Neurorobotics Platform which is currently being developed. Where the required interfaces did not exist or did not have some required features, we built and modified them ourselves (e.g. by adding population support to the ROS-SpiNNaker Interface from Stephan Reith).

Related work has been published by Christoph Richter, Sören Jentzsch and colleagues (Richter et al., 2016) and describes a control of a single-joint Myorobotic arm with a spiking neural network model of the cerebellum simulated on SpiNNaker. The main differences to our system is that we focus on modeling another part of the human motor control system, the motor cortex, as well as that our system is based on ROS and thus allows easier interfacing with other robotics hardware and simulation.

1.3 Thesis Overview

We begin in chapters 2, 3 and 4 by introducing the theoretical foundation of the topics that this thesis builds upon.

Chapter 2 will explain the concept of spiking neural networks and their relationship to neural activity in the brain as well as software packages that support their modeling and simulation. Building up on that we will introduce the Liquid State Machine as a specific type of spiking neural network.
1 Introduction

In Chapter 3 it will become clear that spiking neural networks necessitate a new type of computing hardware called neuromorphic systems like the SpiNNaker machine that was used for this thesis.

Chapter 4 switches over to the field of robotics by introducing the concepts of biomimetic robotics and embedded cognition. These give rise to the area of artificial musculoskeletal systems like the Myorobotics single-joint arm that we used in this project.

Chapter 5 presents the different software tools and components that we used for this thesis as well as the interfaces that we used and modified to connect them. We furthermore describe how we constructed both the hardware model and a simulation model of a single-joint Myorobotic arm.

In Chapter 6 we then describe our closed-feedback system which sends joint angle values encoded as spike rates to a Liquid State Machine and relays the decoded readout spike rates over a transfer function as motor commands back to the robot. For each subsystem we explain our design choices based on research in computational neuroscience.

In Chapter 7 we discuss biological evidence for our overall system and design choices and compare our implementation to related work on cerebellar control for musculoskeletal robots. We furthermore discuss engineering aspect like scalability, modularity and real-time support.

Finally, in Chapter 8 we briefly summarize our results and the conclusions that we have drawn from them. We also state why we think our research is important for the fields of neurorobotics and cognitive systems. Concluding, we give some final remarks on future work.

This thesis is accompanied by a Github repository at https://github.com/Roboy/LSM_SpiNNaker_MyoArm.git which contains our code.
2 Spiking Neural Networks

The discipline of neuroscience started in the late 19th century, when the Spaniard Ramón y Cajal investigated brain tissue using the silver staining technique which had been developed by his Italian colleague, Camillo Golgi, only a few years before. Cajal’s methodical study and detailed drawings led to the observation that the nervous tissue is a network of discrete cells, later named neurons, which are unidirectionally transmitting signals (Law of Dynamic Polarization), opposed to the more numerous glia cells which are assumed to perform support functionality.

Although many different types of neurons have been found in the brain, e.g. pyramidal cells in the cortex (Figure 2.1 A), cerebellar Purkinje cells (Figure 2.1 B) and stellate cells in the cerebral cortex (Figure 2.1 C), three distinct functional parts can be identified in most of them.

A typical neuron consists of **dendrites** which are collecting signals from other neurons, a **soma** which processes the incoming signals and an **axon** which transmits the processed signal to other neurons over a junction called **synapse**.

The notion that neurons are performing signal processing has been added to the **neural doctrine** of discrete dynamically polarized neurons as another central tenet of neuroscience. The idea that neurons process information goes back to McCulloch’s and Pitt’s seminal paper "A logical calculus of the ideas immanent in nervous activity" from 1943 where they argue that "neural events and the relation among them can be treated by means of propositional logic" (McCulloch and Pitts, 1943).

Although the neuron model of McCulloch and Pitts was rather abstract and had little physiological realism, it can, together with Lapique’s seminal paper on the integrate and fire model from 1907 (Lapique, 1907), be conceived as the starting point of computational neuroscience, a subfield of neuroscience that only recently in 1985 came to its name by Eric L. Schwartz and is build on the core assumptions that the brain performs information processing and computation and can therefore be modeled mathematically. Those mathematical models can furthermore be simulated and analyzed on a computer.

According to the computational paradigm, information is encoded in temporal patterns of action potentials, or spikes, in the nervous system and computation is done in the neurons when incoming action potentials lead to new spikes. To better understand the information processing in brains it is therefore important to take a
2.1 Models of Spiking Neurons

An important idea for neuron modeling is that neurons are in essence electrical devices and can therefore be modeled by electrical equivalence circuits. The electrical property of interest in neurons is the potential difference between their outside and their inside. This potential difference is generated neurophysiologically by ion pumps in the cell membrane. Those protein-based ion pumps maintain gradients of ion concentrations across the membrane. Besides the ion pumps there are also ion selective resting channels in the membrane, most of them only permeable to $K^+$ ions which have a high concentration on the inside and a small concentration on the outside.

As depicted in Figure 2.2, the resting channels allow the $K^+$ ions to diffuse across the membrane along their concentration gradient which results in a net positive charge...
The formula for computing this equilibrium potential is the **Nernst equation**:

\[ u_{\text{equi}} = \frac{RT}{zF} \ln \frac{[K^{+}]_{\text{out}}}{[K^{+}]_{\text{in}}} \]  \hspace{1cm} (2.1)

With \( R \) being the ideal gas constant, \( T \) being the temperature in Kelvin and \( F \) being Faraday’s constant. The rectangular brackets denote chemical concentrations of ions.

Figure 2.2: Physiological basis for the neuron’s membrane potential (Kandel et al., 2013, p. 130)

This positive charge results in an electrical force opposing the diffusion flow of the ions. When there are enough ions outside of the cell, so that their electrical force equals the chemical driving force, then the movement of ions stops and a stable electrical potential difference across the membrane is established.

The same happens with every ion type that the membrane is permeable for (although the diffusion direction differs depending on the ion type). The superposition of all equilibrium electrical potentials is called resting potential of the neuron and is computed with the **Goldman-Hodgkin-Katz equation**:

\[ u_{\text{rest}} = \frac{RT}{F} \ln \left( \frac{P_{\text{Na}}[Na^{+}]_{\text{out}} + P_{K}[K^{+}]_{\text{out}} + P_{Cl}[Cl^{-}]_{\text{in}}}{P_{\text{Na}}[Na^{+}]_{\text{in}} + P_{K}[K^{+}]_{\text{in}} + P_{Cl}[Cl^{-}]_{\text{out}}} \right) \]  \hspace{1cm} (2.2)

\( P_{\text{ion}} \) stands for the permeability of a specific ion in meters per second.

The resting potential of a typical neuron is (depending on the temperature) at around \(-65mV\). Electrical models of the neuron describe what happens to the membrane potential when there is an input current \( I(t) \) as depicted in figure 2.3. The membrane can acts like a capacitor due to its insulating property. When there is no input current,
The differential equation for this equivalent circuit is given by:

$$\tau_m \frac{du}{dt} = -[u(t) - u_{rest}] + RI(t) \quad (2.3)$$

With the membrane time constant $\tau_m = RC$

then the voltage across this capacitor is the resting potential of the membrane $u_{rest}$ which is modeled by a battery in parallel to the capacitor. The membrane is not a perfect insulator which is why the incoming current will slowly leak through it, which can be characterized by a leak resistance $R$.

### 2.1.1 Action Potential

The capacitor integrates the input current by accumulating the incoming positive electrical charges on the inside of the cell $q = \int I(t') dt'$ while some of it is being leaked to the outside of the cell. This leads to the depolarization of the membrane potential. When a certain voltage threshold has been passed, voltage gated $Na^+$ channels open which leads to sudden influx of positively charged $Na^+$ ions into the inside of the cell. Shortly afterwards the $Na^+$ channels are inactivated again and $K^+$ channels open which drives the potential back to resting potential. This process of rapid depolarization after the voltage threshold has been passed followed by a slower reversal to the resting potential is called action potential or spike due to its characteristic form in the voltage time course as depicted in figure 2.4. Because this model incorporates leak current,
has a capacitor that integrates the input current and it spikes when the membrane potential passes the threshold, it was named **leaky integrate and fire (LIF) model**. Besides the LIF neuron model there are many other models with varying degrees of biological plausibility, for example the Hodgkin-Huxley model which also models the conductivity of the individual voltage-gated ion channels.

![Figure 2.4: Time course of an action potential and its constituting ion conductances (Kandel et al., 2013, p. 158)](image)

Action potentials are all-or-nothing processes and consequently have roughly the same shape everywhere in the brain. They typically have a duration of 1-2 ms and their amplitude is about 100 mV (Gerstner et al., 2014, p. 5). As a consequence, the form of an action potential does not carry any information, only the temporal sequence of them does. Therefore, the information in neural networks is usually modeled by time series of discrete binary events called spike trains. The understanding of the physiological basis of these spike trains remains important, as the physiological parameters shape the form of the spike trains. A low voltage threshold with a lower time constant, for example, can result in more and faster spikes. Very important is furthermore whether the neuron model includes an absolute refractory period. Biological neurons have an upper bound on how fast they can fire, because after each action potential the sodium channels cannot be immediately opened again, because they must be reactivated first, which generally takes about one millisecond.
2.1.2 Synaptic Transfer

Action potentials typically appear first at the neuron’s soma as a response to the incoming synaptic excitation from other neurons. The action potential then travels down the axon to the synapses where they lead to neurotransmitter release which elicit a post-synaptic potential (PSP) at the dendrites or the soma of the following neuron. According to Dale’s Principle the chemical processes that take place in the synaptic connections are roughly identical everywhere (although it has been shown that there are differences, e.g. with respect to the neurotransmitter), regardless of the type of target cell. Another version of Dale’s Principles states that a neuron can either be excitatory or inhibitory in its effects to other neurons and can thus be characterized as either excitatory or inhibitory neurons. Excitatory neurons induce an excitatory post-synaptic potential (EPSP) at the target neuron which corresponds to a depolarization of its membrane potential, whereas inhibitory neurons lead to an inhibitory post-synaptic potential (IPSP) which hyperpolarizes the membrane and thereby inhibits firing. Synapses can be modeled as having weights which represent how much the presynaptic neuron’s activity influences the postsynaptic neuron. Neurophysiological research has shown that synaptic weights are adaptive, mainly by increasing or decreasing the number of postsynaptic receptors. These synaptic weight changes are thought to be the neural basis for learning.

2.1.3 Connectionism: From Artificial to Spiking Neural Networks

Another paradigm that is widely accepted amongst neuroscientists is the connectionist belief that cognition is an emergent property that arises from the processes in complex interconnected networks of neurons. After the term connectionism was introduced by Donald Hebb in the 1940s, its immanent ideas quickly spread into other disciplines and can be found in artificial intelligence, neuroscience and philosophy of mind. While neurons are still believed to be the fundamental units of information processing in the neural system, connectionists emphasize the importance of the network topology and the adaptive properties of the network due to synaptic plasticity. The most commonly used connectionist topology are feedforward multilayer perceptrons. A perceptron is an extension of the McCulloch and Pitts neurons which was introduced in 1958 by Frank Rosenblatt (Rosenblatt, 1958). Besides the firing threshold, perceptrons incorporate adaptable weights that allow it to perform supervised learning tasks. Networks of perceptrons are called Artificial Neural Networks (ANNs).

Spiking Neural Networks (SNNs) in contrast are models of neuronal systems that are biologically more plausible than traditional artificial neural networks. Instead of working on a propagation cycle like multi-layer perceptrons, they use asynchronous
event-driven processing, thus incorporating time into their computations. Neurons used in SNNs are modeled, as presented for the LIF model, by differential equations that describe the dynamical behavior of the voltage across the neuron’s membrane in response to an external stimulus, e.g., a current.
2.2 Neural Simulators

The mathematical model described by equation 2.3 is a differential equation that can be solved numerically and thus be simulated. Figure 2.5 shows the model’s response to input currents of different magnitudes and durations. The first two input currents represented by the green graph are not strong enough to evoke an action potential. Instead, they depolarize the membrane until the input current is switched off, which leads to an exponential decline towards the resting potential. From 10 ms onwards the input current is high enough to surpass the voltage threshold and thus elicits continuous spiking.

While it is possible to implement the neuron models by oneself as was done for the Leaky Integrate and Fire model from Figure 2.5, this reinventing-the-wheel approach gets rather tedious for complex networks with different neuron models.

Neural simulators are software packages which offer functionality that makes the construction of large networks of spiking neurons easier and introduces a higher level of standardization. Standardization is highly important for every field of science as it facilitates the comparison and reproducibility of results. Furthermore, the methods are numerically optimized and therefore lead to better performance, especially when working with large networks.

Figure 2.5: Response of a LIF neuron to input currents of different magnitudes (Jupyter notebook can be found in Thesis/LIF_model.ipynb in our thesis code repository)
2.2.1 NEST

One of the most popular neural simulators is the Neural Simulation Tool (NEST) which was developed by the *Neural Simulation Technology Initiative* and was first presented by Markus Diesmann and Marc-Oliver Gewaltig in 2001 (Diesmann and Gewaltig, 2001). An up-to-date description of the NEST simulator can be found in (Plesser et al., 2015). It is free to use under a GNU Public Licence and is available from www.nest-initiative.org (T. N. Initiative, 2016). NEST is a software framework for the simulation of large systems of biologically realistic (spiking) neurons. It includes a wide range of neuron and synapse models that can be defined and connected with high-level commands in a Python-based interface. For the purpose of simulating large networks, NEST uses optimized numerical solvers and supports hybrid parallelization. Another major characteristic of NEST is its close correspondence to electrophysiological experiments.

2.2.2 PyNN

In recent years a diversity of neural simulators has been produced, each with its own programming or configuration language. While this has brought about the possibility of cross-checking between different simulators as well as the opportunity for computational neuroscientists to use the most appropriate and best optimized simulator for their specific task, it has also lead to considerable problems. As a consequence of the diversity of simulators, the communication, comparison and reproduction of scientific results is impeded.

The Python package for Neural Networks specification (PyNN) is a Python-based programming interface for a wide range of neuron simulators that aims at "reducing the problems of simulator diversity while retaining the benefits." (Davison et al., 2009). PyNN enables computational neuroscientists to write a network model and simulation script once and run it with (almost) no changes on several supported simulators including NEST, NEURON and Brian as well as on neuromorphic hardware like SpiNNaker and BrainScaleS. For running spiking neural networks on the SpiNNaker board, one can use the PACMAN software layer which is a set of algorithms that translate a model into machine-executable code (Furber et al., 2014). Instead of having to write network models both in NEST and with PACMAN separately, using the PyNN interface one only needs to implement them once.

The open-source PyNN software package is available from: http://neuralensemble.org/PyNN/
2.3 Liquid State Machine

The network topology of cortical microcircuits in the brain consists of randomly and sparsely connected neurons that communicate irregularly in time with a low average activity (Paugam-Moisy, 2006). In contrast to traditional computational models such as Turing machines, biological networks run in real-time on continuous streams of input and output data. One computational model that incorporates those properties is the Liquid State Machine (LSM) which was proposed by Wolfgang Maass, Thomas Natschlaeger and Henry Markram in 2002 (Maass et al., 2002).

![Figure 2.6: Structure of a Liquid State Machine (LSM) (Maass, 2010)](image)

An LSM consists of a reservoir of randomly and recurrently connected spiking neurons and several readout neurons. The dynamics of the reservoir project the input stream \( u(s) \) for all \( s \leq t \) into a high-dimensional liquid state \( x^M(t) \) from which a memoryless readout function \( f^M \) can be learned. The liquid reservoir thus serves as pre-processor for the readout neurons, similar to a kernel in supervised machine learning. Important to note is that the reservoir is not a static representation of the current input data, but a temporal integration of the input stream. Learning in an LSM only occurs between the reservoir neurons and the readout neurons. Traditionally, the readout stage is assumed to have no temporal integration capability of its own and can thus only learn a static map from reservoir states to readout states. This serves the purpose of allowing the use of simple learning algorithms like linear regression. In this context, Maass and colleagues state, that a primary goal of LSMs is ‘to make the
learning as fast and robust as possible’ (Maass, 2010).

The mathematical model of the LSM can be described by the following set of equa-
tions:

\[ x(t) = (L^M u)(t) \]  \hspace{1cm} (2.4)

\[ y(t) = f^M (x(t)) \]  \hspace{1cm} (2.5)

Maass et al. have shown that an LSM is capable of carrying out several real-time
computations in parallel within the same circuitry with different readout neurons
extracting different ‘summaries’ of information from the same transient liquid states.
Maass assumes that the liquid provides computational preprocessing for a large range
of possible tasks of different readouts (Maass, 2010).

Maass succinctly summarizes the major properties of Liquid State Machines in a
review paper from 2010 (Maass, 2010):

1. An LSM is a model for adaptive computational systems.
2. An LSM provides a method for employing randomly connected circuits for
meaningful computations.
3. An LSM provides a theoretical context where heterogeneous, rather than stereo-
typical, local gates or processors increase the computational power of a circuit.
4. An LSM provides a method for multiplexing different computations (on a com-
mon input) within the same circuit.
3 Neuromorphic Hardware

3.1 Neuromorphic Engineering

The brain is often described as a biological computer, but on closer look this analogy has serious flaws. The great mathematician and computer pioneer John von Neumann was one of the first to state the differences in his last work which was posthumously published in book form as "The Computer and the Brain" (Von Neumann, 1958). Although some differences have changed since the 1950s, e.g. transistors have become considerably smaller and are now similar in size to the membrane thickness of a neuron, certain discrepancies have stayed the same, most notably the brain’s superiority in power efficiency and its reliance on event-based signaling and massively parallel information processing. Artificial systems on the other hand exhibit faster processing speed by orders of magnitude. Von Neumann already pointed out that because of those differences it is unlikely that serial computers could run the same procedures as the parallel brain circuits without hardware changes (Von Neumann, 1958, p. 51f).

Low Power Consumption While the brain consumes only about 20 Watts, robotic systems with less performance require much more for their perception systems alone. For example, the winning robotic car in the DARPA Grand Challenge in 2005 from the Stanford Racing Team carried sensors and computers for the task of autonomous driving that had a combined power consumption of over a Kilowatt (Thrun et al., 2007). One might argue that this discrepancy originates from the use of completely different computational procedures. However, it has become clear that running neural simulations on supercomputers does not result in better energy efficiency. In 2009 the IBM group around Dharmendra Modha reported that they successfully simulated a thalamocortical model with 1.6 billion spiking neurons (more than in a cat’s brain) on a Blue Gene/P supercomputer (Ananthanarayanan et al., 2009). Even though the model consisted of less than 2% of the amount of neurons in the human brain while running 2-3 times lower than real-time, it consumed about 2 MW (Liu et al., 2015), (Ananthanarayanan et al., 2009). This results in a difference of seven orders of magnitude in contrast to the human brain. Even more recently, in 2013, a team from the Institute of Neuroscience and Medicine at Jülich led by Markus Diesmann simulated a network of 1.73 billion neurons using NEST on the Japanese K supercomputer.
Neuromorphic Hardware

(Diesmann and Program, 2013). This simulation ran at less than a thousands of the speed of biological real-time and consumed 10 billion times as much energy as equally many neurons in the brain would have (Meier, 2017b).

Together with the size of modern supercomputers this leads to the conclusion, that different hardware must be used for mimicking the human brain as well as for neuro-inspired autonomous systems.

Event-driven Another major difference between computations in brains and computers is that the former are driven by spike-events that can be evoked on continuous time at irregular intervals. Digital computers on the other hand run on an internal clock, which is the maximum frequency of the systems performance set by the processor. It has been proposed that the event-driven characteristics of brain computations are contributing to their high energy efficiency (Maass, 2015).

Parallel Simulating large networks of spiking neurons is computationally extremely expensive as it requires the numerical solving of the differential equations for every neuron in the network. Fueled by the gaming industry, graphical processing units (GPU) have become very powerful. Recently they have been adopted by the artificial neural networks community (deep learning). However, their level of parallelization still pales compared to the brain and they consume high amounts of energy. Therefore, GPUs might become a suitable platform for high-performance computing, but they are less promising for embedded cognitive systems like neurorobots.

These characteristic differences between human brains and digital computers have given the motivation for a new field of inquiry called neuromorphic engineering. The concept of neuromorphic engineering was developed in the 1980s by the US scientist and engineer Carver Mead, who hypothesized that electronic circuits could be used to mimic neuronal systems (Mead, 1990). However, those electronic circuits should, according to Mead, be designed according to principles that are quite different from standard digital chips. One principle is to use elementary physical phenomena as computational primitives to greatly enhance the efficiency of the system. Going back to the LIF neural model in Chapter 2 this could mean to model the neuron by building an electronic system according to the electrical equivalent circuit of the neuron and use the voltages as analog representations of the membrane potential. Mead made the observation, that semiconductor devices can, when operated in a certain mode (the subthreshold mode in transistors), resemble the physical processes happening in neurons. This, of course, would be orders of magnitude faster and more energy efficient than working with digital representations and solving the respective
differential equations. These considerations bring us to Mead’s next design principle, namely the use of analog systems. The brain however is not a purely analog system. The subthreshold accumulation of electric charge across the membrane is surely is an analog process, but the phenomenon of action potentials introduces a binary digital communication system called spike transmission. Therefore it would be more accurate to describe the brain as a mixed-signal system.

It is important to emphasize here that it is not the computational substance that makes traditional computers so power hungry. The reason lies not in the difference between silicon and biological wet tissue - in fact, a switching operation of a MOS transistor consumes about 20 times less energy than a synaptic transmission (0.5 fJ for transistor switching in contrast to 10 fJ for a synaptic transmission) (Meier, 2017b). Instead of the material, the architecture of modern computers is at fault. Von Neumann introduced the separation of memory and processing unit which does not exist in brains and leads to high computational cost when data has to be transferred frequently between the two subsystems.

Although von Neumann was one of the first to point out that completely different computational concepts would be needed for reverse engineering the brain, the history of neuromorphic systems is often presented as a demarcation from his ideas by alluding to the difference from the computer architecture named after him.

In their recent book *Event-based Neuromorphic Systems* Shih-Chii Liu and her colleagues from the *Institute of Neuroinformatics* in Zurich have added robustness, portability and a development interface for applications to the list of requirements for neuromorphic systems (Liu et al., 2015, p. 5). Furthermore they advocate the asynchronous event-driven protocol called the Address-Event Representation (AER).
Neuromorphic hardware is becoming even more popular as mainstream computing is beginning to face limitations in component density and clock speed, which results in the desire for more distributed processing models. Because of its energy efficiency and high level of parallelization, neuromorphic hardware is even considered for use in high performance computing.

A modern supercomputer consumes as much energy as a small city. The most powerful supercomputer at the time of this writing, the Chinese Sunway TaihuLight, has a power consumption of over 15MW for reaching its maximum of 93,000TFlops/s (TOP500.org, 2017). As has been shown earlier, the brain’s efficiency is many orders of magnitude higher than the one of classical von Neumann computers (at least when running brain-related procedures). While neuromorphic systems have not yet reached the energy efficiency of human brains, they promise to reduce the power consumption of HPC systems immensely, thus allowing further scaling. Horst Simon, one of the authors of the TOP500 project, which lists the world’s most powerful supercomputers, and Deputy Director of Lawrence Berkeley National Laboratory, recently answered a question about the future of HPC by stating that “Computing technology will become more diverse and we may use very different architectures, such as neuromorphic computers” (Sieslack and Simon, 2015).

Power consumption is not only a bottleneck for the design of the next generation of supercomputers, but also for mobile electronic devices and mobile robotic systems like autonomous cars.

Another advantage of neuromorphic chips over traditional von Neumann machines for neurorobotic research is that their event-based communication architectures favor the integration of neuromorphic sensors like silicon retinas. Systems like the Dynamic Vision Sensor (DVS) (Lichtsteiner et al., 2008) are capable of running at thousands of frames per second allowing fast reaction times in uncertain environments. Instead of sending entire images at fixed frame rates, they record only local pixel changes and communicate them in the address-event representation. This not only leads to lower power and memory requirements but also positions them perfectly for integration with neuromorphic chips.
3 Neuromorphic Hardware

3.2 The SpiNNaker Project

Nowadays, there are several different neuromorphic systems available as research platforms. The most well-know are the SpiNNaker project from the University of Manchester, IBM’s TrueNorth, Stanford university’s Neurogrid and the BrainScaleS project at the University of Heidelberg.

The first two systems are digital, while the later are based on analog or mixed-signal circuitry. At present, digital systems are far better scalable than their analog counterparts due to their underlying CMOS-technology. Therefore, at least at the moment, it seems permissible to break Mead’s analog systems rule.

The SpiNNaker and BrainScaleS systems are part of the Human Brain Project (HBP) within the Neuromorphic Computing Platform subproject.

For this project, the SpiNNaker architecture was chosen because of its scalability, easy-to-use interface due to the support of PyNN as well as the comparatively good support. Furthermore this choice will facilitate a possible future integration of the project presented in this thesis into the HBP Neurorobotics Platform (NRP) which aims at supporting SpiNNaker as well. Another advantage of the SpiNNaker system is that it is capable of running real-time simulations of neural networks which is of utmost importance for neurorobotic systems.

Figure 3.2: SpiNNaker 48-chip development board (Workshop, 2013)

The Spiking Neural Network Architecture (SpiNNaker) (Spiking Neural Network Architecture) many core computer architecture has been developed by Steve Furber’s
Advanced Processing Technologies Reasearch Group at the University of Manchester. Each SpiNNaker chip consists of 18 ARM 968 cores running at 200 MHz clock speed, a router as well as a 128 MB SDRAM. One core is always reserved for monitoring, e.g. controlling communications across the machine. Therefore users can only use 17 cores for their neuron models (see Figure 3.3).

The ARM cores are classical off-the-shelf computers which makes the SpiNNaker system comparatively inexpensive, which is valuable as Furber’s group is planning to install a supercomputer system out of 1 million SpiNNaker ARM cores within the Human Brain Project. This appears to be achievable as the current implementation already features 500,000 cores.

What makes SpiNNaker into a neuromorphic hardware is therefore not its processors but instead its specialized communication system. Spikes simulated on the cores are sent asynchronously in the address-event representation to other cores which simulate the receiving neurons. Thus whenever a spike occurs, a packet containing its source neuron is sent out and either routed to one of chip’s 18 cores or to one of the six neighboring chips/nodes. This hexagonal connectivity forms a topological torus as shown in Figure 3.4.

This lightweight multicast packet-routing mechanism is the key innovation of SpiNNaker as it supports the very high connectivity found in biological brains and is the basis for the high scalability of the system (Furber et al., 2014).

The Manchester APT group is currently working on upgrading the SpiNNaker system to SpiNNaker-2 of which working prototypes already exist at the time of writing (2017).
Figure 3.4: SpiNNaker’s communication by multicasting forms a two-dimensional torus where each node/chip is directly connected to six neighbors (Meier, 2017a) and which is expected to be available as an operational system by the year 2020. SpiNNaker-2 will be extremely energy efficient, with only 25 GIPS/W (gigainstructions-per-second per watt) in contrast to a peak performance of 3.96 GIPS/W of the current system (Painkras et al., 2013) and will support true random numbers as well as floating point precision (Karlheinz Meier: The era of cognitive computing 2017). These changes will further strengthen the area where neuromorphic hardware is strongest - low-power neural computation.

The characteristics of low power consumption, spatial compactness and the capability to run large-scale models in biological real-time make neuromorphic chips like SpiNNaker a core requirement for advances in the field of neurorobotics. Another emergent field in neurorobotics is the use of biomimetic hardware modules, which will be presented in the next chapter.
4 Biomimetic Robotics

4.1 Biomimetic and Musculoskeletal Robotics

Robotics has always been inspired by biological systems, especially by humans. The first use of the term robot was in Karel Čapek’s science fiction play R.U.R. (Rossum’s Universal Robots) from 1920 where he described artificial humans.

A special type of bio-inspired engineering systems are biomimetic systems, which go further than only adapting functional principles by additionally imitating biological systems morphologically. The system that roboticists are particularly interested in is the animal and human motion apparatus. Despite all the advances in robotics during the last decades, even the most sophisticated robotic systems don’t even come close to the level of dexterity that humans and animals possess. Bipedal locomotion and grasping has been a notoriously difficult task for decades and, as the DARPA Robotics Challenge in 2015 (DARPA Robotics Challenge (DRC) 2015) illustrated, the flexibility and robustness of human gait and manipulation is still far out of reach. The control task and its accompanying computational complexity seem too complex. This has led to the idea of morphological computing, which proposes that biological systems take advantage of their shape, material properties and physical dynamics to conduct computations needed for successful interaction with their environment (Füchslin et al., 2013). This assumption plus the fact that robotic systems are supposed to conduct similar interactions with the environment as humans do, leads to the conclusion that designing robots with a morphology similar to ours might enable us to achieve a satisfactory level of dexterity and flexibility. On a biomechanical perspective, having a similar morphology to humans and animals means consisting mainly of muscles, bones and tendons that are correctly arranged - in short: being a musculoskeletal system. The main difference to traditional robots which typically have motors in each joint which apply torques is that musculoskeletal robots are instead tendon-driving. This means that the main actuators which pull the tendons can be kept at rest (in humanoid musculoskeletal robots near the torso), resulting in low-inertia and lightweight end-effectors. This together with their built-in compliance leads to a high level of safety.

Musculoskeletal systems become even safer by using flexible and soft materials as substitute for biological flesh and skin. This area of research is called soft robotics.

Besides these practical advantages of musculoskeletal systems, namely flexibility,
dexterity and safety, there are also deeply philosophical reasons for taking this approach towards robotics. The concept of embodiment (e.g. as presented in Rolf Pfeifer’s seminal book “How the body shapes the way we think (Pfeifer and Bongard, 2006)) dissolves the Cartesian division between the physical and the mental world. Instead it proposes a tight coupling between the brain, the body and the environment.

The late philosopher and neuroscientist Francisco Varela succinctly described the meaning of embodiment in his book “The Embodied Mind - Cognitive Science and Human Experience”:

"By using the term embodied we mean to highlight two points: first that cognition depends upon the kinds of experience that come from having a body with various sensorimotor capacities, and second, that these individual sensorimotor capacities are themselves embedded in a more encompassing biological, psychological and cultural context.” (Varella et al., 1991)

If the body really shapes the way we think, as the title of Pfeifer’s book suggests, and is necessary for any kind of real intelligence, as the embodiment thesis proposes, then robots need to have human-like morphology to be able to acquire human-like intelligence and the synthetic methodology must become a major part of neuroscience. According to this paradigm, neurorobotics is the way to go in artificial intelligence as well as neuroscience research.

In summary, the reason for doing research in the field of biomimetic robotics is to achieve higher flexibility, dexterity, compliance, safety and possible in the future human-like intelligence by mimicking the musculoskeletal system of humans and animals.

### 4.2 The Myorobotics Project

In the last section, several arguments for doing research on musculoskeletal robots have been presented. However, in the past, roboticists had to construct their own hardware, which was not only time consuming but also reduced the reproducibility and comparison of results by other roboticists. To solve this issue, the Myorobotics project (Myorobotics - A framework for musculoskeletal robot development 2017) was launched in 2012 by the European Union’s Research and Innovation funding programme (FP7-ICT-2011.2.1 Cognitive Systems and Robotics) as a collaboration between the Chair for Robotics and Embedded Systems at TUM, the Bio-Inspired Robotics Lab at ETH Zürich, the Fraunhofer IPA and the Bristol Robotics Laboratory. The project's main feature is its modular design: it consists of primitives (Myo-Muscles, Myo-Joints and Myo-Bones)
that can be freely interconnected to produce different robotic systems. Examples are an anthropomorphic arm, a hopper and even a cheetah (see Figure 4.1).

![Figure 4.1: The Myorobotics project (Myorobotics - A framework for musculoskeletal robot development 2017)](image)

One major characteristic of the Myorobotics toolbox are their actuators which are called Myo-Muscles. These consist of a geared DC motor which winds up a string called tendon on a reel. Before leaving the actuator, the tendon goes through a pulley which itself is connected to a linear compression spring. The combined system reproduces the characteristics of a non-linear progressive spring, making the Myo-Muscle a variable stiffness actuator.

![Figure 4.2: The Myo-Muscle (Myorobotics - A framework for musculoskeletal robot development 2017)](image)

Variable stiffness actuators (VSA) are a subgroup of variable impedance actuators (VIA) that don’t include dedicated damping elements (Wolf et al., 2016). Since their
links are decoupled from the motor inertia by the adjustable spring, they allow the absorption of shock, e.g. in case of collision with a human. This is a major advantage over stiff robots, where the link and motor are directly over a gearbox, which can’t, even with control, reduce the contact forces (Wolf et al., 2016).

Figure 4.3: Our robot setup as an implementation of the antagonistic VSA design principle (Wolf et al., 2016)

Another use case for VSAs are cyclic movements due to their ability to store potential energy in their springs. Since locomotion and also certain arm movements can be modeled as limit cycles, this results in the potential to build energy efficient systems, an important feature of modern robotics as stated before.

As depicted in Figure 4.3 our robot setup reflects the antagonistic VSA design principle. When the motors move in the same direction the link is being moved. When they pull at the link through their tendons in different directions the result is a change in the system's stiffness.

The drawback of the VSA-characteristic of the Myo-Muscles is that they are inherently nonlinear, which makes them hard to control with traditional control methods. As indicated in the introduction, this gives rise to our idea of building an LSM-based system that can learn how to deal with the nonlinearities in order to control the arm. The implementation of this system will be presented in the next two chapters.
5 Implementation: Tools, Interfaces and Components

This chapter presents the different software tools and components that we used for this thesis project. The software tools include NEST and PyNN for our neural simulation and the ROS for the communication with the robot. For constructing a closed-loop sensorimotor system it became necessary to integrate these software packages, which is why we will present our approaches on interfacing ROS with NEST and ROS with SpiNNaker. We furthermore describe how we constructed both the hardware setup and a simulation model of a single-joint Myorobotic arm.

As illustrated in Figure 5.1 it was our goal to construct a system where the simulation and physical realization of the robot could be freely combined with different spiking neural networks running either with NEST or on SpiNNaker.

![Figure 5.1: System components and their interfaces](image_url)
5 Implementation: Tools, Interfaces and Components

<table>
<thead>
<tr>
<th>Software tool</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyNN</td>
<td>0.7.5</td>
</tr>
<tr>
<td>NEST</td>
<td>2.2.1</td>
</tr>
<tr>
<td>sPyNNaker</td>
<td>3.0.0</td>
</tr>
<tr>
<td>ROS</td>
<td>Kinetic</td>
</tr>
<tr>
<td>Gazebo</td>
<td>7.0.0</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>16.04 (Xenial)</td>
</tr>
</tbody>
</table>

Table 5.1: Overview of employed software tools

5.1 Tools

We focused on using state-of-the-art software tools which are also routinely used by researchers in the disciplines that are making up the field of neurorobotics. For modeling and simulating the neural network we therefore chose NEST, which is heavily used in computational neuroscience, with PyNN as an interface, allowing us to run the same code almost without any changes on SpiNNaker.

For the communication with and the control of our robot we used the Robot Operating System (ROS) which, in the last years, has become the standard tool for robotics research. ROS natively supports the virtual robot simulator Gazebo. Due to its modular design it is hence possible to control either a physical or a simulated robot with the same code, given that they support the same ROS-messages.

For reproducibility purposes we have listed all software tools and their versions in Figure 5.1. The reasoning behind our decision to employ these specific version will be presented in the respective sections.

Subsequently, the HBP Neurorobotics Platform (NRP) is presented, which is a large-scale effort towards integrating these neuroscientific and robotic software tools in order to offer the young neurorobotics research community a high-quality integrated software platform for their in silico experiments. Due to the Neurorobotics Platform still being in active development, it was not used in this thesis, but we designed our project with compatibility in mind, so that it can be added to the NRP in the future.

While NEST and ROS can be combined by repeatedly simulating short time intervals of a neural network model within a ROS-node, the communication between SpiNNaker and ROS is more difficult to achieve. Fortunately, a basic communication interface has recently been developed that we were able to adapt to our project.

We decided to share all of our code on the repository hosting website Github to allow other researchers to benefit from it.
5 Implementation: Tools, Interfaces and Components

<table>
<thead>
<tr>
<th>PyNN</th>
<th>NEST</th>
<th>Features</th>
<th>sPyNNaker support</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7.5</td>
<td>2.2.1</td>
<td>Numpy data handling</td>
<td>yes</td>
</tr>
<tr>
<td>0.8.1</td>
<td>2.10.0</td>
<td>Neo; Izhikevich &amp; Poisson models</td>
<td>not yet</td>
</tr>
</tbody>
</table>

Table 5.2: Overview of the software packages’ interdependencies

5.1.1 PyNN and NEST

For specifying our LSM-based neural network, we chose to use PyNN as it allows to run our code with almost no changes either on the SpiNNaker board or with the NEST neural simulator on a traditional PC. Due to the multitude of employed Python packages, it was very important to take care of their interdependencies. For each PyNN version there is for example only one NEST version that is supported, all older versions are not.

For running PyNN scripts on SpiNNaker it is necessary to also install the sPyNNaker front end interface from the Manchester SpiNNaker group’s Github repository page (SpiNNaker Manchester Github 2017). At the time of this writing the PyNN-SpiNNaker software interface only supports PyNN 0.7.5, although support for PyNN 0.8 is supposed to be released in the near future. Unfortunately, PyNN 0.7.5 is only compatible with NEST version 2.2.1 while PyNN 0.8 supports NEST version 2.10.0. For our purposes the main differences between these version pairings is the introduction of the Izhikevich neuron model, Poisson spike sources as well as a new way of data handling compared to PyNN 0.8. The newer versions use Neo (N. Initiative, 2017) for representing the data that is being recorded from the simulated neural network (e.g. spike trains or membrane voltages). This increases the usability of PyNN for neuroscientists considerably as Neo is a Python package for representing electrophysiological data, from intracellular, extracellular and even EEG recordings. Thus, this change allows neuroscientists to use the same data representation tool in in vivo experiments as well as in in silico experiments which raises their comparability. Since Neo objects behave like NumPy arrays, they can be intuitively used for computations as well as for visualization with Matplotlib.

We tested using sPyNNaker with PyNN 0.7.5 in a Python virtual environment and PyNN 0.8 with the latest NEST version globally, so that we could switch between these configurations. However, because of the presented changes (Neo etc.), scripts could not be run with either SpiNNaker or NEST as back end without changes, defeating the main motivation of using PyNN in the first place. Therefore, we decided to primarily use PyNN 0.7.5 in this thesis. The newer version was only used for neuron-level experiments as shown in 6.2 and 6.3 where the superior data representation capability of Neo presented a considerable advantage. We were able to solve the problem that
neither the Izhikevich neuron model nor poisson spike sources are directly supported in PyNN 0.7.5. by importing them as native cell types from NEST.

As soon as the Manchester SpiNNaker group releases the new version of sPyNNaker with its PyNN 0.8 support, these problems will become obsolete.

5.1.2 The Robot Operating System (ROS) and Gazebo

**ROS** The Robot Operating System ROS has been developed in the robotics incubator Willow Garage since 2007 and was released in 2010 under an open source BSD licence. After the closure of Willow Garage in 2014 the project has been continued by the Open Source Robotics Foundation. In the last years the number of robots operating on ROS has been steadily increasing.

![Figure 5.2: Number of robots running ROS](image)

‘In 2015 alone, more than $150 million in VC funding was invested in businesses that utilize ROS’ ([ROS, the Robot Operating System, Is Growing Faster Than Ever, Celebrates 8 Years 2017](#)). At its core ROS is a middleware which connects different robot hardware and facilitates the programming of application software by providing different communication protocols named topics (asynchronous), services (synchronous) and actions (asynchronous). In addition, ROS provides a multitude of useful plumbing tools which enable roboticists to gain a better understanding of what is happening inside their complex system by listening to its internal communication and visualizing the organizational structure of its modules, called nodes.

The parts of our system run as ROS-nodes which send and receive messages. There is, for example, a node which sends the joint sensor encodings as messages which are then received by another node which translates them into both degrees and radians. These are then also sent as messages to a different topic from which the ROS-node which contains the neural network takes them to inject them as input spikes.

Besides the modularity that results from this design, the philosophy behind ROS also
includes support for multiple programming languages including C++, Python, LISP, Java, MATLAB and more. In principle, every ROS-node can be written in a different language. For our project, the nodes containing the neural network as well as its signal processing periphery were written in Python while C++ has been used by the Roboy team for the nodes that are communicating with the robot hardware.

ROS is only supported for Linux machines with special focus on the Ubuntu operating system. For this project we used ROS Kinetic Kame, the tenth ROS distribution release, because it was the most recent version with long term support, together with Ubuntu 16.04 (Xenial).

With the integration of visualization software and simulators like Gazebo as well as efficient motion planning capabilities through MoveIt!, the ROS ecosystem has become the de facto standard set of tools in robotics research worldwide.

Gazebo  Gazebo (Koenig and Howard, n.d.) is an open-source 3D dynamic multi-robot simulation environment that has been used for more than a decade. Especially its tight integration with ROS as well as the incorporation of multiple high-performance physics engines (ODE, Bullet etc.) and advanced OGRE-based 3D graphics have made it very popular.

Our decision to add the capability of interfacing with a virtual model of the Myorobotic arm to our system is based on the following reasons: In robotics research, simulation plays an important role, especially for quick and efficient prototyping and testing. While hardware is expensive, often messy and has to be repaired, simulations can be run cheaply and without much setup time. Furthermore, simulation allows other researchers to test and verify (or falsify according to Karl Popper) theories and experimental results, a feature that lies at the core of philosophy of science. Simulation also democratizes research as it allows groups with smaller budgets to participate. Thus it also scales up research: Instead of one group working on their private robot, simulation allows multiple research groups all over the world to contribute and collaborate.

5.1.3 The Neurorobotics Platform

These philosophical motivations also form the basis for the Neurorobotics Platform (NRP), a subproject of the European Human Brain Project (HBP).

The NRP is a simulation environment that allows to combine brain models based on spiking neural networks and virtual robotic embodiments as well as a rich and dynamic environments (Falotico et al., 2017).

Another challenge for the NRP is that it is inherently an interdisciplinary project as it has to cater for the needs of both roboticists and neuroscientists. For intuitive and fast adoption of the platform as well as to uphold the quality of constituents, the NRP does
not reinvent the wheel but instead integrates the standard tools of roboticists (ROS and Gazebo) with the tools of neuroscientists (NEST). By combining these tools in a software infrastructure that can be accessed via a web interface (though it can also be run locally), the NRP allows researchers to conduct in silico neurorobotics experiments.

Even though the NRP was not used for this thesis, due to it still being at the time in its developmental phase, it played a central role in many design choices. We expect the NRP to become a widely used tool in the neurorobotics community. In anticipation of this, we chose to exclusively use the tools supported by it for this master thesis project and used compatible simulation parameters (e.g. having a cycle time of 20ms when running a model in NEST). We hope that these considerations will allow our results not only to be reproducible by other researchers, but also to be incorporated into the platform (e.g. our virtual robot model and possibly the ROS-SpiNNaker interface).

5.1.4 Github

We decided to use the version control and repository hosting service GitHub for maintaining and publishing our code. The advantage of using version control is, that as it saves a historical record of changes in the code base, a rollback to a previous state becomes possible, in case a complex-to-solve error has been made. Furthermore, the Git version control system allows to easily work on the same code base on different computers and more importantly to collaborate with others. Since our project was embedded in the much larger Roboy student team project, the latter was very important. Git allowed us to include several repositories from the Roboy team for managing the communication with the Myorobotics parts into our own repository as submodules (namely common_utilities, roboy_communication, roboy_plexus, and roboy_rqt_plugins). This allowed us to develop and update them independently from our own project.

Furthermore, by publishing our code openly and free for everyone we aim to follow the open science movement. We believe it to be important for the general advancement of science that results in the form of data, code and methodology are shared. This is crucial as it allows confirmation or falsification of scientific theories and in synthetic engineering science it renders the reinvention of the wheel unnecessary. Besides, this seem only fair, as all the tools we used, from NEST, PyNN to ROS and Gazebo are open-source as well.

You can find the repository for this thesis project online at:

https://github.com/Roboy/LSM_SpiNNaker_MyoArm.

For downloading the repository, you can type the following into your Unix shell:

$ git clone https://github.com/Roboy/LSM_SpiNNaker_MyoArm
Our repository has a BSD 3-clause license, allowing modification, distribution, private use and even commercial use. The license however does not include liability nor do we provide any warranty.
5 Implementation: Tools, Interfaces and Components

5.2 Interfaces

5.2.1 ROS-NEST Interface

We were able to combine ROS and NEST by designing a ROS-node in Python which imports `pyNN.nest`, subscribes to the joint-angle ROS-topic and initializes a publisher for sending the readout rates to other nodes. Whenever a new message is received, a callback function is processed which unpacks the incoming ROS-message and uses it to set the Poisson spiking rate of our input neurons. The callback function also contains our network specification which it runs for 20 ms while recording the spikes occurring at the readout neurons. We decode these resulting readout spike trains twice, once by calculating the mean firing rate during the simulated time window and a second time by weighting the spike times with an alpha function. This approach is presented in more detail in Chapter 6, when we introduce spike decoding. The two results are then published on the topics `/mean_readout_rates` and `/alpha_readout_rates` respectively, using our custom `Pop_List` ROS-message type.

The PyNN `run()` function supports being called multiple times during a simulation as well as the modification of neuron and synapse parameters in-between runs.

5.2.2 ROS-SpiNNaker Interface

Another design choice was to connect the SpiNNaker chip, on which our neural model was run in real time, with ROS, thus allowing communication between our neuromorphic hardware and the robotics middleware. This has a decisive advantage in usability over the approach of previous work in our group, where a custom-built hardware interface (SpiNN-IO) was used to translate sensor data into SpiNNaker packages and vice versa SpiNNaker packages into motor commands (Richter et al., 2016). However, as will be discussed later, using the current version of ROS has the disadvantage of not being hard real-time.

For this purpose of integrating SpiNNaker and ROS, we used the ROS-SpiNNaker Interface, which is a Python-based software package that was developed by Stephan Reith as part of his bachelor thesis at the Technical University of Munich.

This interface uses the `sPyNNakerExternalDevicesPlugin` which provides the `SpynnakerLiveSpikesConnection` functionality. This function allows to define the local port for communication. Furthermore, the `sPyNNakerExternalDevicesPlugin` includes the `SpikeInjector` cell class for defining a PyNN input population.

It also provides the `activate_live_output_for` method, which outputs spikes from a given population from SpiNNaker as they occur in the simulation.

The combination of these functions allows to inject spikes into the neural model running on SpiNNaker as well as to record spikes during runtime. The ROS-SpiNNaker
Interface builds on top of this and implements methods that put the recorded spikes into ROS-messages and receive ROS-messages, unpack their content and send it as spikes to SpiNNaker.

In the following, we will briefly describe how this is achieved.

The central component of the interface is the ROS_SpiNNaker_Interface class, which can be used in one’s PyNN code. There, it can either function as input interface from ROS to SpiNNaker or as output interface from SpiNNaker to ROS (or as both at the same time).

**ROS to SpiNNaker**  As show in Figure 5.3, when using the interface for spike injection, one has to define the number of input neurons, the spike source class as well as the name of the ROS-topic from which the input should be received. The interface then initializes a new ROS-node (given that the ROS-master has already been started by the user; otherwise an exception is caught, which prints the respective error message) of the name 'spinnaker_ros_interfaceX', where X is the number of the interface, and subscribes to the topic that was passed as an attribute. The message type of this subscriber is Int64. When a new ROS-message is received on the subscribed topic, its data field is unpacked and put on a queue (data type implemented in the Python multiprocessing library).

This queue is given as an attribute to a spike source of the type with the number of neurons that were defined as attributes of the ROS_SpiNNaker_Interface. This spike source can be chosen as either SpikeSourceConstantRate, SpikeSourcePoisson or SpikeSourceNumNeurons (all defined in the transfer_functions.py script) that all inherit from a generic BasicSpikeSource class (defined in the population.py script). The BasicSpikeSource takes the first ROS-value from the queue and for each neuron sends a spike to SpiNNaker with the send_spike function from the SpynnakerLiveSpikesConnection class which is imported from sPyNNakerExternalDevicesPlugin. Then an interval counter is set until the next spike is sent for this specific neuron, based on the received ROS-value. The value of this interval counter depends on the chosen spike source class and is computed there.

For our project we used the SpikeSourcePoisson type. The reasons for choosing this model of spike generation as well as a more detailed explanation on how it works will be presented in section 6.3.

**SpiNNaker to ROS**  For defining the ROS_Spinnaker_Interface in the output direction, so that it records spikes from the neurons in the network running on SpiNNaker and sends them as ROS-messages to an output topic, it must be given a spike sink class, the name of the output topic and a PyNN population of the output neurons. The
input_interface = ROS_Spinneraker_Interface(
    n_neurons_source=n_input_neurons,
    Spike_Source_Class=SpikeSourcePoisson,
    ros_topic_send='to_spinnaker',
    clk_rate=1000,
    ros_output_rate=10)

output_interface = ROS_Spinneraker_Interface(
    Spike_Sink_Class=SpikeSinkMultipleReadoutsConvolution,
    ros_topic_recv='from_spinnaker',
    output_population=readout_neurons,
    clk_rate=1000,
    ros_output_rate=10)

Figure 5.3: How to use the ROS-SpiNNaker Interface

interface then initializes a ROS-node with a ROS-publisher, that sends Int64 messages to the given topic. Furthermore, a spike sink class of the passed type which is given the output PyNN population as an attribute. Another attribute is a queue on which recorded spikes are set. These spikes are tuples that consist of a label, a time stamp and the ID of the neuron where the spike occurred. Analogous to the spike source classes, there are several spike sink classes defined in the transfer_function.py script which inherit from a BasicSpikeSink class defined in the population.py script. In the BasicSpikeSink class, the first spike from the queue is taken and appended to a list containing tuples of neuron IDs and a spike times array. Furthermore, the spike time as well as the spiking neuron’s ID and the last ROS value are given as arguments to the on_spike method which is implemented in the specific spike sink classes, where it computes and returns the next ROS-value which is then packed into a ROS-message and published to the output topic.

Support for neural populations  The original ROS-SpiNNaker interface only supports the injection and recording of single neurons with an Int64 ROS-message. Since our LSM was designed to control two antagonistic motors, it was necessary to be able to record the neural activity of at least two neurons. Furthermore, the focus in computational neuroscience, especially in the area of motor control, has shifted to studying the properties and behavior of whole populations of neurons instead of single neurons (e.g. in (Churchland et al., 2012)). Instead of only adding support for two output neurons, which would have been enough for our project, we therefore decided to directly add
support for arbitrarily large neural populations, so that the revised ROS-SpiNNaker Interface will be usable by other researchers for other projects. Furthermore, it is our goal to prepare it for integration into the Neurorobotics Platform, which requires generic interfaces.

Therefore, we implemented a new ROS-message type called \texttt{Pop\_List}, which can hold an arbitrarily long array of int32 values. Each of the values of this array is supposed to represent the activity of one neuron of the output population.

The revised code detects whether the output population is a single neuron or a neuron population. In the latter case, it sets a flag to True and changes the ROS-message type of the topic on which the interface publishes to \texttt{Pop\_List}.

This way, we maintained the compatibility to code written for the original ROS-SpiNNaker interface.

Furthermore, in the case of an output population the size of the ROS-message is automatically set to the number of output neurons.

Next, we also changed the code of the \texttt{BasicSpikeSink} class in the \texttt{population.py} script. Here, we again added a flag which is set to True if we are trying to record from more than a single neuron. In this case, a 1-D Numpy array \texttt{new\_ros\_value} with rows equal to the number of neurons in the output population is initialized. The value returned by the \texttt{on\_spike} method is then written to the appropriate element of the \texttt{new\_ros\_value} array and represents the current spike rate of the respective neuron.

For the computation of the spike rates we added a new transfer function in the script \texttt{transfer\_functions.py} of the name \texttt{SpikeSinkMultipleReadoutsConvolution}.

In contrast to the other spike sink classes, the \texttt{on\_update} method from \texttt{SpikeSinkMultipleReadoutsConvolution} requires a forth attribute. Besides the \texttt{spike\_time} of the current spike, the \texttt{neuron\_id} of the neuron that spiked and the last ROS value, it also requires the number of output neurons.

Inside the \texttt{SpikeSinkMultipleReadoutsConvolution} class, we first defined the convolution function as an alpha window of the form \( f(x) = 0.1 \times x \times e^{2-x} \). In section 6.6 we’ll explain in more detail why this convolution filter was chosen.

Then, an array of the name \texttt{pop\_last\_spike\_times} is initialized, which has as many rows as the output population has neurons and 10 columns. Whenever a spike occurs and thus the spike sink class’s \texttt{on\_update} function is called, the time when the spike occurred, will be added to the row of the respective neuron by replacing the spike time that lies the furthest in the past. This is achieved by treating each row of the array as a \textbf{first in, first out (FIFO)} data structure. Then, the row of the spiking neuron, containing the neuron’s ten most recent spike times, is weighted with the filter defined above. Due to the shape of the filter, this means, that more recent spikes have a higher contribution to the momentary spike rate than spikes that occurred further in the past.
For our project, only the recording of neuron populations was required, but not
the injection of spikes into a neuron population as our feedback value (joint angle)
was 1-dimensional and could therefore be represented by the rate of a single neuron.
This is why we focused on implementing population support for live spike recording
only. However, we are already working on implementing population support for spike
injection.

You can install the ROS-SpINNaker interface with neural population support with
the command:

$ pip install git+https://github.com/reiths/ros_spinnaker_interface

If you installed the sPyNNaker interface within a Python virtual environment, be sure
to activate it first by typing:

$ source venv/bin/activate
5.3 Components

5.3.1 SpiNNaker 4-Chip Board

For this thesis we used the SpiNN-3 board which comprises four chips with 72 cores in total.

The SpiNN-3 board can be connected to a PC over Ethernet by setting the board’s IP address in the network configuration and by updating the ‘.spynnaker.cfg’ configuration file in the home directory.

![SpiNN-3 board](image)

Figure 5.5: SpiNN-3 board

5.3.2 Myorobotics Single-joint Arm

**Hardware**

The structural composition of the Myorobotic 1-joint arm that we used in this project consists of two Myo-Bones connected by a Myo-Joint and driven by two antagonistic Myo-Motors as depicted in figure 5.6.

Since the robot arm had been used and modified in the HackRoboy hackathon, it had to be disassembled and rebuild first.

The 1-joint arm consists of five structural parts: a Myo-Bone as the upper arm, a Myo-Bone as the lower arm, a Myo-Hinge-joint functioning as elbow joint and two Myo-Muscles, one at each side.

**Myo-Bone** As depicted in figure 5.7 we used the Type 1 Myo-Bone which are constructed according to the "parallel assemblies" design principle. These bones consist of elongated structural profiles called *fibres* which are hold together by transverse *spacers*. The spacers have the function of increasing the assembly’s strength and stiffness while
being light-weight. On the end of the bones are end-spacers which provide an interface to other Myorobotic parts, e.g. other bones or Myo-Joints. The upper arm Myo-Bone was firmly fixed in a vertical position on a block of wood which formed the base of the robot arm. The lower arm Myo-Bone was then connected via the hinge joint and therefore forms the movable part of the system.

![Figure 5.6: Single-joint Myorobotic Arm](image1)

**Myo-Joint** Most crucial was the correct assembly of the hinge joint which connects the bones of the robot’s upper and lower arms. The joint consists of an upper and a lower joint fork which provide structural bonds to the upper and lower arm and are flexibly connected by a joint axle. The upper fork has a cable guidance and cable attachments on both sides for the tendons coming from the Myo-Motors. Thus, when tendons are pulled, a force is applied to the upper fork which thence rotates around the joint axle.

Since the old joint angle sensor was broken and could not be used with our FPGA, a new sensor had to be integrated into the robot. The sensor that was chosen is the ‘AM4096 12 Bit Rotary Magnetic Sensor IC’ (RLS, 2017a) integrated in the ‘RMK4
5 Implementation: Tools, Interfaces and Components

Evaluation Board’ (RLS, 2017b) as shown in Figure 5.9. The sensor chip contains an array of Hall sensors which produces a voltage representation of the magnetic field distribution when the diametrically polarized magnet which is placed on top of the sensor rotates. This voltage representation is then converted to a 12 bit encoder value which is representing the joint angle.

The exploded assembly drawing on the right side of Figure 5.8 as well as Figure 5.10 give an impression on how the sensor board was integrated into the hinge joint. After soldering the power supply cables and the clock and data line connections to the joint sensor PCB, the chip was then fit into the inside of the hinge joint and fixated with a hot-melt gun.
Unfortunately, the soldering connection of one of the cables broke after having fixated the chip and therefore had to be reattached while the chip was inside the hinge. This proved to be extremely difficult as the soldering iron could only be inserted into the joint in a flat angle and the connector was partly occluded by the other cables and additionally because the power supply connectors where extremely close, resulting in a short circuit if too much solder was used. Once the connections were successfully reattached, the cables were glued to the hinge to prevent future detachment.

For reduced resistance while moving the joint, ball bearings were inserted into the two moving parts of the hinge on each side on top of the sensor IC (see Figure 5.8, number 1). The drill hole diameter was slightly smaller than the diameter of the ball bearings, therefore they had to be cooled down first before inserting them in the hole leading to an interference fit. Inside the ball bearings were holes in which the shafts of the hinge joint (Figure 5.8, number 10 and 12) were put in a tight fit after colling down. One attachment had a cavity on its tip where the magnet could be inserted. In the final assembly the magnet would then always stay stationary as it was connected to the fixed link of the robot while the sensor board would move with the moving part of the hinge joint resulting in Hall sensor readings proportional to the angle. Finally, the power and data line cables from the joint sensor were connected to the FPGA.
Communication For the communication between the Myorobotic arm and the PC running ROS, an Field-Programmable Gate Array (FPGA) was used. The FPGA is connected with Ethernet to the same network as the PC which is running the ROS-master. One can then use a secure shell (SSH) to access the FPGA from the PC and add the PC’s IP address as ROS_Master_URI to the FPGA’s .bashrc. This establishes a connection between the two and ROS-messages send from the FPGA can be received on the PC. Once the ./roboy_plexus node is started on the FPGA it will start sending messages to a range of different topics, including MotorStatus and JointStatus. Since the joint states that the FPGA sends are encoder values, we implemented another node convert_joint_angle which converts these values to degrees and radians and publishes those. The roboy_plexus node also subscribes to the MotorCommand topic from which it extracts the ID’s of the controlled motors and their respective set points. The motors are then driven to these points by PID-control. This ROS-interface was developed by the Roboy team at TUM and can be generically applied to different Myorobotic systems. Our single-joint setup was used for testing and debugging.

![Figure 5.11: ROS-node converting 12-bit joint encodings into joint degrees and radians](image)

For easier usability we build an GUI (Figure 5.12) based on the Roboy rqt-plugins for motor control and motor status plotting. Additionally, the interface plots the robot’s joint angles in degrees by using the messages published by our convert_joint_angle node.

Simulation

In robotics it is extremely useful to have a virtual model of the robot to work with in simulation. Physical robot are often unavailable if several people are working on them or when some part of their hardware is broken. Furthermore, it takes time to set up the connection to the physical robot and this can obviously only be done in physical vicinity of the robot.

For this thesis Gazebo (OSRF, 2017) was chosen as robot simulator, not only because it is the simulator that is most tightly integrated with ROS (since they are both developed by the Open Source Robotics Foundation), but also because it is used in the HBP Neurorobotics Platform (NRP) (Falotico et al., 2017). A central guiding principle of this thesis was to design everything in a way that it is compatible with the NRP.
The reason why Gazebo was used directly instead of the NRP was that at the time of this work the NRP had not yet reliably integrated the option to run the brain models on a SpiNNaker machine. Furthermore by virtue of the NRP still being in the development process, continuous access to the platform was not guaranteed.

However, it is to be expected that in the future the NRP will become an extremely useful simulation and research environment for neurorobotics.

Before constructing an accurate model of the 1-joint Myorobotic arm, a simplified model was built with the purpose of quickly testing the connection between a ROS-driven robot in Gazebo and a spiking neural network running in NEST (Figure 5.13, left image). It uses the ros_control plugin for URDF files which allows to send desired joint angle positions over ROS and to receive the current joint angle.

Next, we set out to construct a more detailed and accurate model of our Myorobotic arm with tendon support. In a first step, the CAD files of the Myorobotics primitives were used in the construction of a CAD model of the 1-joint Arm, including the definition of rigid groups and the rotational joint (Software: Autodesk Fusion360). Equivalent to the physical robot, the virtual model was designed with a solid rectangular base on which the first link was rigidly attached. This static link was then connected to a second, moving, link via a hinge joint. Lastly, two Myo-Muscles were attached on both sides of the static link.

Then, the SDFexporter plugin, which had been developed by the Roboy student...
5 Implementation: Tools, Interfaces and Components

team, was used to convert this CAD model into an SDF file, which could be imported in Gazebo. In order to increase the speed of the simulation it was necessary to reduce the number of polygons in the STL-models of the arm’s links. To achieve this, the decimate modifier in Blender was used with a reduction factor of 90%. The resulting model is depicted on the right side of Figure 5.13. As illustrated, the upper link can be rotated around the joint axis.

![Simple 1-joint arm](image1)
![Myorobotic 1-joint arm](image2)

Figure 5.13: Simulation models

**MyoMuscle Plugin**  Gazebo does not support the simulation of tendons natively, however, it does allow to create custom plugins. The Roboy team developed a prototype of a tendon plugin and had Nicolas Hink debug, improve and generalize it. For this task, he needed a simple robot model as test platform, while we wished to make our simulation model more realistic by adding tendon support. Thus, we collaborated and were able succeed. The main idea was to model the effects of the motor-pulled tendons on the movable link of the arm by introducing external forces equal to the forces that tendons would apply. Within the plugin-tag in our SDF-model, we added two Myo-Muscles and defined the attachment points of their tendons on the motor and the movable link.

However, as shown on the left side of Figure 5.14, the simulated tendons would go through the joint. Besides its non-realism, this effect lead to the muscles not being able to drive the movable link back if its angle was too high. Therefore, Nicolas Hink introduced tendon wrapping to the plugin which we added to our SDF model. This was done by defining an invisible cylinder within the joint as well as the constraint that tendons could not intersect this cylinder but would instead wrap. The result is shown in Figure 5.14 on the right side.

The simulation can be run by cloning the repository from
https://github.com/Erzcoder/roboy-control-learning.git to your local ROS workspace, building and sourcing it an then typing the following command into your terminal:

```bash
$ roslaunch roboy_control_learning single_joint_myoarm.launch
```

This ROS-launch file will start Gazebo and spawn the SDF-model of our Myorobotic arm. Additionally, it visualizes the robot and the tendon forces in Rviz.

Figure 5.14: Effect of adding tendon wrapping to the robot model
6 Implementation: System Architecture

This chapter presents our approach to implementing a system that is capable of running a Liquid State Machine as a spiking neural network on either a SpiNNaker board or on a traditional PC with NEST. The LSM’s reservoir is connected to the joint sensor measurements of a musculoskeletal arm and the readout neurons’ activities is transferred into motor commands, enabling closed-loop neural control.

6.1 Robot Model

The physical robot as described in the last chapter, can be modeled as a subsystem or module that takes motor commands as input and outputs a new joint configuration. The resulting joint configuration is not explicitly expressed, but must be measured in the next module.

6.2 Joint Sensor

The joint sensor, including our transformation ROS-node, is modeled as a subsystem that takes in a new joint configuration of the real or virtual robot and outputs the respective joint angle value in degrees. While this works nicely with the real robot, the SDF model of the virtual robot must first converted to the URDF format, which supports the ros_control plugin, which is necessary for reading joint angles.
6.3 Spike Encoding

The spike encoding subsystem takes the ROS-value of the joint angle as input and encodes this information into spiking activity of the LSM’s input neurons. Thus, it serves as a stimulus-response mapping.

Origins of spike time variability Neurophysiological recordings have shown that the timing of spikes is not deterministic but underlies variability. Especially in the cortex, the Inter-spike Interval (ISI) is highly irregular (Heeger, 2000).

According to Wulfram Gerstner, Werner Kistler, Richard Naud and Liam Paninski (Gerstner et al., 2014, p. 168 ff), this variability comes from several sources, one of which is intrinsic noise in the ion channels. Their opening and closing behaves stochastically depending on the number of ion channels and the temperature. Another, much stronger source for spike variability in the brain is that biological neural networks have non-uniform connection lengths, in contrast to artificial neural networks where the output from the neurons of one layer is assumed to arrive simultaneously at the inputs of the neurons of the next layer. Corresponding to these mechanistic causes, there are two main hypothesized coding schemes. Rate coding treats stochasticity as an intrinsic property of the brain. Thus, the exact timing of individual spikes conveys little information. Instead, the theory’s proponents claim, one should focus on the firing rate of the neurons. The other main neural coding scheme is called temporal coding and assert, as the name suggests, that the precise timing of spikes and their intervals conveys information.

In this thesis we follow the rate coding scheme and model spike trains as temporal Poisson processes. This is a fair assumption if one assumes that the generation of spikes is a random process driven by an underlying continuous signal \( r(t) \) and independent of all other spikes, called the independent spike hypothesis (Heeger, 2000). The disadvantage is that we loose the biological detail of the absolute refractory period as it would represent a lower bound to the spike intervals and thus render the spike generation dependent on previous spikes.

In the following, we will show how to construct a Poisson process for modeling rate encoding (after (Heeger, 2000)): We introduce the neural response function \( \rho(t) \) which is modeled as a spike train with each spike being a Dirac impulse \( \delta(t) \):

\[
\rho(t) = k \sum_{i=1}^{k} \delta(t - t_i)
\]

where \( k \) is the number of spikes and \( t_i \) are the spike times.

For this neural response function to be a Poisson process, the number of spikes in each time interval of length \( \Delta t \) must be Poisson distributed according to the Poisson
probability density function:

\[ P(n \text{ spikes during } \Delta t) = e^{r\Delta t} \frac{(r\Delta t)^n}{n!} \]  

(6.2)

where \( r \) is the mean firing rate. The probability that at any time \( t_0 \) the next spike occurs after the time \( \Delta t \) is the same as the probability that no spikes occur until \( t_0 + \Delta t \), which we can compute by plugging \( n = 0 \) into equation 6.2. To get the probability that the next spike occurs before \( \Delta t \) we subtract this from 1:

\[ P(\text{next spike before } \Delta t) = 1 - e^{r\Delta t} \]  

(6.3)

The probability for the waiting time \( \Delta t \) until the next spike is the derivative of this cumulative distribution:

\[ P(\Delta t) = \frac{d}{dt}(1 - e^{r\Delta t}) = re^{-r\Delta t} \]  

(6.4)

This exponential distribution represents the distribution of the inter-spike intervals.

We used this insight to generate a Poisson spike train for experimental purposes. You can find the implementation as an interactive Jupyter notebook in `Archive/spike_coding.ipynb`. There, in the section *Generating a spike train. Method 1*, we draw repeatedly from an exponential inter-spike interval distribution, with \( r = 300 \) and use the returned values to construct a Poisson spike train. Afterwards we estimate the rate by binning and counting the spike, resulting in a decoded rate of about 270, rather close to the true rate of 300. Unfortunately, for higher spike rates, this difference increases. Our analysis has shown two sources of errors in this method. The first one is, that the returned inter-spike intervals are cast to integer, which always rounds down, therefore, the average ISI is smaller and the encoded spike rate larger. Obviously this leads to the opposite effect of what was observed and can thus not explain it alone. The integer-casting also has the effect that many returned ISIs are set to 0 (more often as the spike rate increases). Since there can’t be two spikes at the same point in time, these spikes are then discarded which decreases the encoded spike rate.

The notebook also presents an alternative approach to Poisson spike train generation, which results in a more accurate spike rate.

For the SpiNNaker-PyNN code we used the ROS-SpiNNaker Interface’s `SpikeSourcePoisson` class to inject Poisson distributed spikes into our neural reservoir running on SpiNNaker. On every update step, this type of spike source class returns an interval value defining the time until the occurrence of the next spike by randomly drawing a number from an exponential distribution whose mean is the desired average inter-spike interval, thus the inverted spike rate (see method 1 from above). This is done by using the `expovariate` function from the Python
random-library. This function applies the inverse CDF (cumulative distribution function) method to the $P(\Delta t)$ PDF (probability density function) which results in calculating:

$$\text{poisson\_interval} = -\frac{\log(\text{random}())}{r}$$  \hspace{1cm} (6.5)

Thanks to the analysis and experimental implementation above, we were able to find and correct a logical error in the original ROS-SpiNNaker interface, where the expovariate function was given the argument $1.0/\text{lambda}$ (note: lambda is used here, because $\lambda$ is a reserved word in Python). The documentation of random.expovariate reads:

random.expovariate(lambd):
Exponential distribution. lambd is 1.0 divided by the desired mean.

(Foundation, 2017).

The desired output mean is not the spike rate but the average inter-spike interval. Therefore, lambd should be the spike rate instead of its inverse. Furthermore, according to the analysis above, we try to solve the problem of generating a too low spike rate due to the integer-casting, by setting zero-intervals to 1, so that the spikes will not be discarded.

Due to the analysis above, in the future, another spike source class could be added for Poisson neurons, which implements the more accurate second method. Especially for high spike rates this would be very beneficial.

Additionally, we added a SpikeSourcePoissonRefrac class, where an absolute refractory period can be set. The next inter-spike interval is drawn from the exponential distribution until a value is found that is larger than the refractory period. It has to be noted, that this addition results in a more regular firing and defiles the independent spike hypothesis, rendering the spiking behavior of the neuron a non-true Poisson process. Furthermore, the introduction of the absolute refractory period results in higher inter-spike intervals on average. Thus, the spike rate should be increased proportional to the chosen absolute refractory period.

For the NEST-PyNN code, we imported the NEST poisson_generator model as a native neuron model into PyNN as it was not yet supported in PyNN 0.7.5. We then used it as a cell type to define our input population. The NEST poisson_generator model allows to define the mean firing rate as well as a start and stop time of firing. However, it is not possible to set a refractory period. Alternatively, NEST also features a poisson_generator_ps model which supports an absolute refractory period.
6.4 Neural Network

In complex emergent systems like neural networks it is important to completely understand their single building blocks even if this knowledge is not enough to understand the whole network. Therefore, the first task was to construct the most simple network possible consisting of two neurons, one as input and one as output. The input neuron was modeled as a spike source that would spike at times 3, 6, 9, 12, 20, 20.5, 28 and 30 and was connected with a static synapse of weight 40 to the output neuron. The spike trains of both neurons were recorded and visualized (Figure 6.2 first and second row) as well as the membrane potential time course of the output neuron (Figure 6.2 third row). The results for different neuron models for the output neuron are presented in (a) to (d) in Figure 6.2 (Note: for this analysis we used PyNN 0.8.5 with the Neo neurophysiological data handling). Subplot (a) shows several important characteristics of neural networks. First of all a delay of about 1.5 ms between the spikes of the input neuron and the spikes of the output neuron can be seen. This is a reasonable value as the synaptic delay was set to 1 ms. The two spikes at times 20 and 20.5 do not result in two spikes but only in one as they are temporally too near to each other so that the second spike is incoming when the output neuron is still in its absolute refractory period.

Figure 6.3 shows the same network with a second output neuron using the Izhikevich model. Instead of having an incoming synaptic weight of 40 like the first output neuron, the second output neuron has a synaptic weight of only 10. The second output neurons spike train is shown in the third row and its membrane voltage time course is presented in the fifth row. As can be seen, with the reduced input weight a single spike in the pre-synaptic neuron is no longer able to put the post-synaptic membrane potential over the threshold to evoke an action potential. Instead the pre-synaptic action potential leads to an excitatory post-synaptic potential (EPSP) that slowly decays over time. Only the summation of several EPSPs can push the membrane potential over the threshold to evoke a spike.

In contrast to the LIF model, the Izhikevich’s model is an approximation of the Hodgkin-Huxley model and is thus biologically more plausible. It consists of a system of two differential equations. The variable $v$ represents the membrane potential of the neuron and the variable $u$ represents the membrane recovery:

\[
\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{6.6}
\]

\[
\frac{du}{dt} = a(bv - u) \tag{6.7}
\]

Additionally, the resetting of the membrane potential after a spike has occurred is
6 Implementation: System Architecture

(a) Izhikevich neuron model  
(b) Hodgkin Huxley model

(c) Leaky Integrate and Fire model  
(d) AdEx model

Figure 6.2: Simple network of two neurons with varying neuron models

modeled by

\[
\text{if } v \geq 30, \text{ then } \begin{cases} 
v \leftarrow c \\
u \leftarrow u + d
\end{cases}
\] (6.8)

Figure 6.4 shows that different sets of the parameters a, b, c and d lead to different modes of spiking behavior that match the rich dynamic behavior of neuron recordings in the brain (Izhikevich, 2003). The implementation with the parameter tuples responsible for the different behaviors can be found as a Jupyter notebook in our repository under Thesis/neuron_models/Izhikevich_model_experiments.ipynb.

6.4.1 Stability

A necessary property of any neural reservoir is stability, which means that a short period of input activity results in a finite period of reservoir activity. If instead an small input to the neural network leads to sustained and increasingly powerful spiking
behavior, we call it unstable or chaotic. Such a network could not be used as a reservoir for our Liquid State Machine.

The standard way of analyzing the stability of networks is to study the properties of their connection matrices. In neuroscience the synaptic weights are not uniform and are instead modeled as random samples from a weight distribution. Therefore, insights from random matrix theory can be successfully applied. A spectrum of eigenvalues of a random weight matrix with large real parts (several that are larger than 1) indicates an unstable network. To avoid this, Girko’s circle law (Girko, 1985) can be applied, which states for a network of size $N$ (where $N$ is large), that if one draws the weights from a distribution with zero mean and a variance of $1/N$, the eigenvalues of the $N \times N$ random weight matrix lie uniformly within the unit circle.

Neural networks with a high degree of biological plausibility obey the constraint of Dale’s principle which states that synapses can have either excitatory (E) or inhibitory (I) neurotransmitters, but not both and are furthermore unable to change their type of neurotransmitter.

As Kanaka Rajan and L.F. Abbott note in their seminal paper "Eigenvalue Spectra of Random Matrices for Neural Networks" from 2006, this biological constraint exacts that "the elements of the synaptic matrix must be drawn from two distributions with different means and perhaps different variances" (Rajan and L. Abbott, 2006).

In the cortex the observed ratio between excitatory and inhibitory neurons is approxi-
mately $f = 0.8$. This has the effect that the density of eigenvalues is higher in the center of the complex plane than towards the circle’s border, in contrast to Girko’s circle law which predicts uniform eigenvalue distribution.

Figure shows the eigenvalue spectrum of a random network of 1000 neurons with connectivity of 0.05 and $f = 0.5$. On the right side is the same network with $f = 0.8$.

Considering that eigenvalues at the edge of the eigenvalue circle represent slowly oscillating and long-lasting modes, these results are intuitive. Fewer inhibitory neurons (as for $f=0.8$) result in more eigenvalues in the center of the complex plane and thus more fast oscillations.

The experiments in Figure 6.13 (at the end of this chapter) show that as long as the eigenvalues of the reservoir weight matrix lie almost exclusively within the unit circle the spiking activity of the excitatory neurons is stable. The neurons start firing shortly after the onset of the Poisson distributed input spike trains and cease firing only a few milliseconds after the input has been stopped after 40 ms. The short delays between the start/finish of the input activity and the start/finish of the reservoir activity are
(a) Eigenvalue spectrum with E-I ratio $f=0.5$  
(b) Eigenvalue spectrum with E-I ratio $f=0.8$

Figure 6.5: Relation of eigenvalue spectrum and E-I ratio

due to the synaptic delays of the neural projections. According to (Rajan and L. Abbott, 2006) the radius of the circle in which the eigenvalues of the synaptic strength matrix lie can be computed with:

$$r = \sqrt{1 + f \mu_E^2 + (1 - f) \mu_I^2}$$  \hspace{1cm} (6.9)

with $f$ being the ratio between excitatory and inhibitory neurons in the network (in our case 0.8), $\mu_E$ being the mean excitatory weight and $\mu_I$ being the mean inhibitory weight.

Once the eigenvalues of the weight matrix grow beyond the unit circle, some having real parts larger than 1 due to choosing a higher mean of the excitatory weight distribution, the network becomes increasingly unstable as shown in row 3. The last row illustrates the effect of an even larger mean of the excitatory weights which leads to many eigenvalues of the weight matrix lying outside the unit circle. This leads to unstable and chaotic behavior of the reservoir as depicted on the right side.

The code with all parameter values that were used for these experiments can be found within the LSM_SpiNNaker_MyoArm repository under Thesis/NetworkStability/network_stability_1.ipynb as an interactive Jupyter notebook.

Inhibition stabilization Until now, only the network projections between excitatory cells have been studied due to our focus on the effect of excitatory weights and their eigenvalue spectrum on the reservoir stability. While we increased the excitatory mean weights in 0.6 nA steps from 0.2 nA to 2.0 nA, we left the inhibitory weights at a constant mean weight of 0.8 nA. Now, we would like to investigate the effects of increasing the inhibitory weights on the network stability, known as inhibition stabilization.

When changing the inhibitory weights, obviously the eigenvalue spectrum of the excitatory to excitatory weight matrix does not change. Therefore first of all, we needed
to construct a combined weight matrix $J$ of all reservoir sub-projections as shown in Figure 6.1.

Rajan and Abbott showed that the balance condition, namely that the strengths of the synapses to each neuron (column-sum of $J$) sum up to 0, is a condition for network stability. In this case, if all neurons are equally active, excitation and inhibition cancel each other for each neuron. Mathematically this is represented by $J^T u = 0$ with $u_i = 1$. In contrast to their paper, we use the matrix transpose due to the format that PyNN uses for projection weights. PyNN does not offer a way to implement this condition while drawing the weights for E- and I-neurons from different distributions, as the combined matrix $J$ consists of multiple projections whose weights can only be chosen in separation. Instead we approximate the balance condition by multiplying the mean excitatory weight by $\frac{f}{f_I}$ to get our mean inhibitory weight. In our case there are four times as many excitatory neurons as inhibitory neurons. Therefore, on average each neuron gets inputs from four times as many excitatory neurons as inhibitory neurons. Multiplying the mean weight of these inhibitory inputs will then result in net zero input on average, thus balancing the network.

Therefore, in our case the balance condition is only approximately met, but not strictly.

Our experiments show that the approximation of the balance condition still helps to stabilize the network. Using the network From figure 6.13, row 4, with mean excitatory weight of 2.0 nA we changed the mean inhibitory weight from 2.0 nA to 8.0 nA (four-times the excitatory weight for balancing out the 4-fold superiority in numbers of excitatory neurons). The network response to the same input stimulus is shown in Figure 6.6, where one can see that the network has been decisively stabilized compared to its behavior in Figure 6.13 row 4. Most neurons stop firing shortly after the end of the input stimulus at 40 ms and by the end of the simulation time of 100 ms all of them have gone back to a quiescent state.
Scaling stabilization Given a fixed connection probability within the network, the number of synaptic inputs that each neuron receives increases when a larger network size is chosen. This has the potential to destabilize the network as it leads to larger variances in input to each neuron which can push the incoming depolarization over the membrane potential threshold voltage more frequently. To counteract this scaling destabilization effect, Rajan and Abbott suggest drawing excitatory weights from a distribution with mean $\mu_E \sqrt{N} > 0$ and variance $r/N$ and drawing the inhibitory weights respectively from a distribution with mean $\mu_I \sqrt{N} < 0$ and variance $r/N$.

Figure 6.7 (a) shows the spiking behavior of the same network as in 6.13 row 1, however with $N = 2000$ instead of $N = 200$. While the increase in the number of neurons in the reservoir has not resulted in instability, it now takes more time to stop firing after the input stimulus has been discontinued at $t = 40 ms$. Figure 6.7 (b) shows the effects of scaling the variances of the distributions from which the weights are drawn by the network size and (c) shows the effects of scaling the distributions’ means, both according the Rajan’s and Abbott’s suggestion. While variance scaling has little to no effect, mean scaling results in a more stable firing behavior.

(a) Without mean and variance scaling  (b) Only variance scaling  (c) Only mean scaling

Figure 6.7: Scaling stabilization of the reservoir

Lastly, we combine variance and mean scaling in Figure 6.8. It shows the effect of
increasing the number of neurons in the reservoir to 2000. While one can see that the firing frequency has grown, the network is still stable, returning to quiescent behavior shortly after the end of the input stimulus.

While these results seem positive with regards to the aim of stabilizing the network against scaling, it must be noted that our results differ from the results obtained in (Rajan and L. Abbott, 2006). For them “modifying the mean strengths of excitatory and inhibitory synapses has no effect on stability [...] under balanced conditions” and “the key elements in determining the spontaneous dynamics of networks [...] are the widths of the distributions of excitatory and inhibitory synaptic strengths.”. The differences in our results are likely to be caused by our non-perfect compliance with the balance condition, despite the approximation thereof. This points to the sensitivity as well as to the importance of the balance condition for the stabilization of randomly connected neural networks, which is further supported by experimental data (Okun and Lampl, 2008). Vogels and colleagues suggest that the balance of excitatory and inhibitory synaptic currents in biological neurons is maintained by inhibitory plasticity (Vogels et al., 2011). A comprehensive read on this topic is Guillaume Hennequin’s PhD thesis from 2013 (Hennequin, 2013).

In Thesis/balance_condition.ipynb we implemented the balance condition, meaning that the inputs to every neuron in the reservoir cancel each other out and sum up to zero. To achieve this, we only drew the excitatory weights from a random distribution and chose the inhibitory weights so that the column sums were zero (resembling the inhibitory plasticity). We have not yet repeated the above stability analysis for this new balanced network.

The increase in firing rate in figure 6.8 compared to figure 6.13 row 1 is likely an artifact due to a fixed input connectivity. By reducing either the input weights or the input connectivity with respect to the increase in reservoir size (and therefore the number of input connections), we could decrease the firing rate of the 2000-neuron network to a similar magnitude as observed in the 200-neuron network.

Figure 6.9 shows the results of different approaches to input scaling in order to counteract the effects of a reservoir growing in size. These spike plots correspond to a
reservoir of size $N = 2000$, which should be compared to the spike plot of the stable network with size $N = 200$ shown in figure 6.13, row 1 as well as the spike plot in figure 6.7 where the size is also $N = 2000$ but no input scaling has been implemented.

Row 1 of figure 6.9 shows the effects of input weight scaling, while having a constant input connectivity between the Poisson source population and the reservoir neurons. On the left side we scaled the input weights directly with the reservoir size ($w/N$) whereas on the right side we scaled it with the square root of the reservoir size ($w/\sqrt{N}$).

Row 2 shows the effects of input connectivity scaling, while having constant input weights. On the left side we used direct scaling ($inp\_conn/N$) while on the right side we used the square root of $N$ for scaling ($inp\_conn/\sqrt{N}$).

As can be clearly seen, direct scaling (left side) leads to a too low firing rate compared to figure 6.13 row 1, whereas square root scaling (right side) leads to similar firing rates (albeit a little too high). These empirical results should however be taken with caution - more research has to be done first to find the correct scaling factor.

Interestingly a comparison of the right side spike plots of figure 6.9 suggests that high weights (unscaled as in row 2) have a higher contribution to a quick onset of firing as high input connectivity (unscaled as in row 1).

![Pics](Direct input weight scaling) ![Pics](Square root input weight scaling)

![Pics](Direct input connectivity scaling) ![Pics](Square root input connectivity scaling)

**Figure 6.9: Input scaling for large reservoirs**

It has to be noted that stability analysis of random neural networks is an ongoing research field within computational neuroscience. Especially the application of random matrix theory and mean-field methods are promising.

Our network analysis has resulted in the following design of the LSM’s liquid reservoir:
6 Implementation: System Architecture

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of input neurons</td>
<td>$N = 10$</td>
</tr>
<tr>
<td>Number of reservoir neurons</td>
<td>$N = 2000$</td>
</tr>
<tr>
<td>Number of readout neurons</td>
<td>$N = 2$</td>
</tr>
<tr>
<td>Input connectivity</td>
<td>$\text{inp_conn} = \frac{0.3\sqrt{2N}}{\sqrt{N}}$</td>
</tr>
<tr>
<td>Input weight</td>
<td>3 nA</td>
</tr>
<tr>
<td>Reservoir connectivity</td>
<td>$\text{res_conn} = 0.05$</td>
</tr>
<tr>
<td>E-I ratio</td>
<td>$f = 0.8$</td>
</tr>
<tr>
<td>Excitatory weight distribution</td>
<td>$\mu_E = \frac{2\sqrt{2N}}{\sqrt{N}}, \sigma_E = \frac{1}{N}$</td>
</tr>
<tr>
<td>Inhibitory weight distribution</td>
<td>$\mu_I = \frac{-11.312\sqrt{2N}}{\sqrt{N}}, \sigma_I = \frac{1}{N}$</td>
</tr>
<tr>
<td>Synaptic delays</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

Table 6.2: Network parameters

6.5 Learning

The learning submodule of our system has not yet been finished, thence we will describe the intended functionality and the progress that we have made so far.

The paradigm that we intend to follow is Robot Learning from Demonstration (LfD), also called Robot Programming by Demonstration (PbD) (Billard and Grollman, 2013). Instead of manually programming the desired behavior of the robot, a human operator performs the task by directly moving the robot with her hands. Using the proprioceptive sensory data that is being recorded during several demonstrations, the robot then learns by generalization to perform the task itself. This bio-inspired approach to robot learning does not only have the potential of making robots more flexible due to easier reprogramming but also allows non-experts to teach the robot.

This has the advantage over just recording and replaying the trajectory that it involves learning the mapping between a dynamic world state and actions. If the robots state is being changed by an external force, the system will recognize this new world state and act accordingly. Thus, we believe that trajectory recording only works in structured and static environments, while learning leads to higher adaptability.

For testing this theory, we propose the following experiment:

The upper link of the Myorobotic single-joint arm is manually moved periodically along a sinusoidal trajectory from the left joint limit to the right joint limit and back. While doing so, the time sequence of motor displacements is recorded. Additionally, we record the sequence of reservoir states of our LSM, which we use for learning the
readout weights for this task, in the way described below. Then, we first play back the recorded trajectory. At one point in time we manually hold the upper link so that it can’t move any more, and release it again after a few seconds. Our hypothesis is, that after releasing the link, the robot will make a rapid movement to make up for the time wasted. This is because the recorded trajectory is unable to adjust to the new situation.

We then repeat the experiment with the trained LSM as controller. Our hypothesis is, that after releasing the upper link, the LSM will ‘recognize’ that the world state has not been altered and resume its original trajectory with little alterations. The difference between the manually demonstrated trajectory and the trajectory after learning when faced with external perturbations can be used as performance measure.

To allow performing this experiment, our goal is to build a system that allows to apply the LfD paradigm to our one-joint musculoskeletal robotic arm. The robot should record its internal states (motor displacements and joint angle) while it is moved along a certain trajectory as well as the reservoir’s liquid state. After the end of the demonstration, the recorded data will be used to train our LSM. The robot has to learn the sequence of motor displacements that would result in the demonstrated trajectory, i.e. for each joint position along the trajectory the robot should learn how to change the motor displacements to bring about the next joint position along the trajectory. By adjusting (learning) the readout weights of our LSM, a dynamic mapping between joint angles and motor displacements has to be learned, constituting an internal inverse kinematic model of the robot.

As Maass and colleagues note in (Maass, 2010), the learning in an LSM works by calculating the weights with a linear regression between the liquid states and the desired readout values. To achieve this, we have implemented a function (for the ROS-NEST interface) that computes a vector containing the rates of all reservoir neurons during the previous simulation window of 20 ms. For the ROS-SpiNNaker interface this can be done by using our population spike rate class for outputting the rates of the reservoir neurons.

The spike rate vectors of the reservoir neurons relate to the liquid state during the respective simulation time window. They can be transposed and stacked into a matrix $X$ of dimension $M \times N$, where $N$ is the number of reservoir neurons and $M$ is the number of recorded simulation windows. It has to be noted here, that the reservoir states are not static representations of the current input, but instead high-dimensional representations of the history of spatio-temporal input streams.

Next, we have to construct the output vector $y$ of length $M$, whose elements are the readout spike rates which relate to the desired motor displacements via the transfer function. Since we can only record the time sequence of desired motor displacement
during the LfD, we have to apply the inverse transfer function.

This gives rise to the following equation:

\[ Xw = y \]  

(6.10)

with \( w \) being the weights between the reservoir and the readout neurons. Due to the fact that our system has two motors, \( w \) and \( y \) each consist of two columns.

The learning process is the procedure of solving equation 6.10 for \( w \). The simplest method is using linear regression with the least squares estimator. In this case, only the normal equation has to be solved:

\[ w_{\text{learned}} = (X^T X)^{-1} X^T y \]

(6.11)

There are two problems with this approach of learning the readout weights. First of all, the weights are learned in one go, instead of incrementally. Therefore, what has been learned in one training phase is overwritten in the next training phase. Second, the learned weights are typically not applicable to the projection between the reservoir and the readout neurons because of Dale’s principle. There might for example be more negative weights than there are inhibitory neurons. Thus, when using classical linear regression one should model the readout neurons as perceptrons instead of spiking neurons, which lowers the biological realism.

The advantage of linear regression for learning the readout weights is that the training process cannot get stuck in local minima like the backpropagation algorithm (Maass, 2010).

In the future, we plan on investigating whether spike-timing dependent plasticity (STDP) could be applied to our system. This form of Hebbian learning would further increase the biological plausibility of our system and is compatible with Dale’s principle and spiking readout neurons. Furthermore, it is an incremental form of learning meaning that past training results are getting adjusted when faced with new data, instead of being overwritten.
6.6 Spike Decoding

The spike decoding submodule takes the spike trains of the readout neurons as inputs and maps them to spike rates.

There are different ways of doing that. The easiest way is to divide the recorded spike train into equally large windows, count the spikes and get the spike rate by dividing the spike count by the window width. This leads, however, to a non-smooth rate function with many discrete jumps.

A more elegant approach is to convolve the spike train with a kernel function, which is often called filter. Figure 6.10 illustrates this approach schematically.

![Figure 6.10: Spike train rate decoding by convolution](image)

In the Jupyter notebook in Thesis/spike_coding.ipynb we take the Poisson spike train that we generated in the section on spike encoding and apply different filters to it. The results are illustrated in Figure 6.11.

Rectangular kernel  We first convolved the spike train with a rectangular kernel which results in an edgy spike rate function (see row 1). Since the kernel is defined in the units according to the chosen time step \( dt \) (in our case milliseconds, because we chose \( dt = 0.001 \) as time step), while the output rate is supposed to be in Hertz (thus on the dimension of seconds), we define a scaling factor. This scaling factor is not only a bridge from the ms-scale to the s-scale, but also normalizes the rectangular kernel by setting its integral to 1.

Gaussian kernel  Next, we defined a symmetric time-discrete Gaussian kernel array with mean \( \mu = 10 \) and standard deviation \( \sigma = 4 \) according to

\[
g(n) = e^{-\frac{n^2}{2\sigma^2}}
\]

with the scipy.gaussian function from the SciPy Python library. This is missing the normalization constant of a typical Gaussian and is like the rectangular kernel in milliseconds, while the output rate is supposed to be in Hertz. We therefore introduce
6 Implementation: System Architecture

<table>
<thead>
<tr>
<th>Original firing rate</th>
<th>300.0 Hz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangular-decoded mean firing rate</td>
<td>314.0 Hz</td>
</tr>
<tr>
<td>Gaussian-decoded mean firing rate</td>
<td>310.4 Hz</td>
</tr>
<tr>
<td>Alpha Function-decoded mean firing rate</td>
<td>311.7 Hz</td>
</tr>
</tbody>
</table>

Table 6.3: Decoded mean firing rates

A new scaling factor which will take care of this situation by multiplying the integral under the Gaussian kernel function with the time step $dt$ and computing the inverse of the result. This scaling factor is then multiplied to the result of the convolution of the spike train with the Gaussian kernel, resulting in an array of spike rates in Hertz. The plot of this Gaussian-convoluted spike train is much smoother as depicted in Figure 6.11, row 2.

**Alpha kernel** Last, we defined an alpha kernel according to:

$$f(n) = \alpha x e^{\frac{2}{\alpha}x}$$

The graph of this function with $\alpha = 0.3$ is shown in the 3rd row in Figure 6.11.

Again, we convolved the kernel with our spike train and applied an appropriate scaling factor. The plot of the resulting rate function is depicted on the right of the alpha kernel.

**Results** All three resulting spike rate functions have roughly the same mean firing rate (MFR), close to the 300 Hz that we initially defined for the generation of our spike train (see Table 6.3).

This means, that they are all appropriate tools for extracting the mean firing rate from a spike train. At first glance, one notices, that the rectangular decoded signal is more edgy and discretized than the other two. The spike rate function produced by the Gaussian kernel is the smoothest one, while the alpha function-decoded signal appears very detailed and has many ripples. At closer inspection, one can see, that the rough shapes of the three graphs are similar, with peaks and minima at almost equal times.

**Discussion of the different kernels** Using a rectangular kernel for the decoding of the spike train is an implementation of moving average filtering. While it is computational comparatively cheap, it has the disadvantage of weighting all spikes within its window equally instead of giving more recent spikes a higher influence for determining the
1. Rectangular kernel  
2. Gaussian kernel  
3. Alpha kernel

Figure 6.11: Convolutional rate decoding with different kernels

The Gaussian filter is an ideal time domain filter, which means that it has no overshoot or ripples in response to a step function while minimizing the rise and fall time of the signal. Furthermore, its smoothing effect reduces noise. The drawback of using a Gaussian kernel is that it is non-causal, because it is symmetric about its mean. Therefore, it weights spikes that occur in the future with the same amount as it weights past spikes, while computing the instantaneous spike rate.

The disadvantage of having a non-causal filter is that the system becomes non-real-time. Furthermore, it has negative effects on the biological plausibility, as neurons can only encode past information into their firing rates and are thus inherently causal systems.

In our ROS-NEST lms_node.py we implemented both the simple mean decoding as well as alpha decoding. After each 20 ms simulation run, we compute the mean firing
rates of all neurons, in the mean_decoding function by simple spike counting and in the alpha_decoding function by weighting the spikes with the alpha function, which results in the instantaneous spike rate at the last spike event during the 20 ms run.

In a continuous real-time setting such as with the SpiNNaker board, the approach of using a non-causal Gaussian filter introduces a delay, because incoming spikes need to fill a queue-like filter window before the Gaussian filter can be applied. This would destroy the real-time property of SpiNNaker. Therefore, it makes more sense to use the causal alpha function kernel. This was already done in the original ROS-SpiNNaker Interface and is kept in the SpikeSinkMultipleReadoutsConvolution that we added.

6.7 Transfer Function

The transfer function takes the decoded spike rate as an input and produces a valid control command for the robot. Therefore, the set of output values to which the transfer function maps should exclude impossible motor positions and ideally it should also exclude pairs of motor positions that correspond to joint positions that violate the joint limits.

The ROS-graph in Figure 6.12 illustrates the structure of our system, where the robot node is a dummy for all ROS communication to either the simulated or physical robot and the lsm_node being either the ROS-NEST interface or the ROS-SpiNNaker interface which both include spike encoding, the neural network and spike decoding.

Figure 6.12: ROS-graph of the system
6 Implementation: System Architecture

1. (a) Eigenvalues of the E-E reservoir weights.
   Mean weight: 0.2 nA
   (b) Spikes of excitatory reservoir neurons

2. (a) Eigenvalues of the E-E reservoir weights.
   Mean weight: 0.8 nA
   (b) Spikes of excitatory reservoir neurons

3. (a) Eigenvalues of the E-E reservoir weights.
   Mean weight: 1.4 nA
   (b) Spikes of excitatory reservoir neurons

4. (a) Eigenvalues of the E-E reservoir weights.
   Mean weight: 2.0 nA
   (b) Spikes of excitatory reservoir neurons

Figure 6.13: Network stability analysis
7 Discussion

This thesis project had a threefold objective. As all neurorobotics research, it aimed at using robotics to better understand the working principles of the (human) brain and vice versa use neuro-inspired principles for controlling robots in a more efficient and natural way. Being an engineering project, a third objective was added, namely to build systems with future use in mind, which are generic enough to be also used by other people and in other projects. Modern scientific and engineering projects are collaborative in nature and therefore a thesis as entrance to these professional fields should not be a one-shot.

7.1 Neuroscience: Insight and Discussion

7.1.1 Comparison to human motor control

A central neuroscientific question that the field of neurorobotics works on answering is how the human (or animal) sensorimotor system works. Neuroroboticists are especially interested in computational models that they might be able to implement into physical robots. For simplicity, we will henceforth focus on the human neurobiological system and disregard the differences in animals.

Figure 7.1: Human motor control (image from (Burdet, Franklin, et al., 2013, p. 6))

Obvious candidates for the role of our system’s readout neurons are the α-motoneurons, which are located entirely within the spinal cord (Burdet, Franklin, et al., 2013). These are single neurons which each control tens to hundreds of muscle fibers,
with whom they form a \textit{motor unit}. Different to our readout neurons, who only receive input from the LSM’s reservoir, motoneurons receive feedback from sensory receptors located in the skin and muscle. Therefore, our LSM system can’t be a model of the peripheral nervous system. Besides the motoneurons, the sensory receptors also send feedback by relay to different parts in the brain. Perhaps, our system is instead a model of the Central Nervous System (CNS) with the proprioceptive sensory receptors (a special type of mechanoreceptors) as input neurons, some brain region as liquid reservoir (as discussed in the next section) and motoneurons as readout neurons. According to Burdet and colleagues the motor control signal is rate coded, where “the higher the firing rate, the greater the force produced by the muscle fibers of the motor unit” (Burdet, Franklin, et al., 2013, p. 20).

They furthermore state that the firing rate of the sensory receptors is also rate coded and that synaptic integration is stochastic (Burdet, Franklin, et al., 2013, p. 20 f.). Thus, the neural firing coming from the proprioceptive receptors and being received at different regions of the CNS can be modeled as a stochastic rate coded process, just like the Poisson spike sources that we used as a neuron model for our input population. The mechanoreceptor type that would be the best biological candidate are called \textit{Ruffini’s endings} which are located deeper in the dermal layer of the skin as shown in Figure 7.2. Burdet and colleagues write that "Ruffini’s endings are sensitive to stretch of the skin that contributes to proprioception, that is, sense of joint position" (Burdet, Franklin, et al., 2013, p. 23), which is exactly what the input neurons in our system represent. Ruffini’s ending are classified as slowly adapting receptors, meaning that they not only transiently increase their firing rate at the onset and offset of the stimulus (like rapidly adapting receptors), but sustain this elevated rate while the stimulus is sustained. This corresponds well to our system, where the ROS-node communicating with the joint sensor is continuously publishing joint angle messages.

Another biologically plausible property of our system which indicates that it might be a model of the motor cortex is the dedicated incorporation of dynamics in reservoir computing. As Churchland and colleagues write in their paper from 2012: "It therefore seems increasingly likely that motor cortex can be understood in relatively straightforward terms: as an engine of movement that uses lawful dynamics" (Churchland et al., 2012). According to them, the question of what single neurons in motor cortex code or represent is misguided, instead research should investigate behaviors and regularities at the population level with a special focus on dynamics.

Yet another connection between our system and human motor control lies in the properties of the Myo-Muscles. As variable stiffness actuators they enable stiffness variation at constant position, which is analogous to cocontraction in humans. Etienne Burdet and colleagues showed in their paper from 2001 that "the central nervous system stabilizes unstable dynamics by learning optimal impedance" (Burdet, Osu, et al., 2001).
7 Discussion

7.1.2 The LSM as a model for computation in cortical microcolumns

The concept of Liquid State Machines was initially inspired by the structure of a generic cortical microcircuit and aimed at giving a computational model for its working principles. Cortical columns are areas in the cerebral cortex which consist of six distinct layers of tightly connected neurons and are oriented perpendicular to cortical surface. Since we are interested in using an LSM for sensorimotor control, the part of the cortex that we focus on is the motor cortex. Figure 7.3 a) shows an LSM-based model of a generic cortical microcircuit based on neuroanatomical and neurophysiological data of an adult rat and cat neocortex from (Thomson et al., 2002). It shows neural subpopulations as nodes of excitatory (black) and inhibitory (red) type as well as their connection probability modeled as the width of the arrows between them. The horizontal pairs of excitatory and inhibitory subpopulation represents the neurons on different cortical layers (from layers 2/3 on top to layer 5 on the bottom. Based on ample anatomical evidence, most sensory inputs (here input stream 1) go into the neurons of layer 4 while the input from other cortical areas is connected to layer 2/3. The output streams are colorized in green and connect the excitatory neurons from layer 5 with lower cortical areas and the excitatory neurons from layers 2 and 3 to higher cortical areas.

While our LSM model is more abstract in having only one excitatory and one inhibitory subpopulation, it could be interesting to add a more complex model based on Figure 7.3 a) and compare both. However, the increased complexity would make the stability analysis much more complicated.

Figure 7.3 b) shows how the reservoir of an LSM can be seen as a model for the cortical microcircuit with two input and two output streams.

According to Gerstner and colleagues, one cubic millimeter of cortex tissue contains more than $10^4$ cell bodies and several kilometers of axonic wires (Gerstner et al., 2014),
which is on the order of magnitude as the reservoir which we are simulating. Taken together with the Maass’ statement, that “the Liquid is typically a generic recurrently connected local network of neurons, modeling for example a cortical column which spans all cortical layers and has a diameter of about 0.5 mm” (Maass, 2010) with about 100 neurons, this means, that our model simulates a mega-column of 20 times the size of a cortical column. In the future, we aim at investigating if this increase in size has an effect on the LSM’s function or if simulating instead 20 smaller sized LSMs is more beneficial.

7.1.3 Insight into spike encoding, stable reservoirs and spike decoding

Since we have already discussed our results on these subsystems in the respective sections, we only give a brief summary here: We have seen that Poisson spike sources can be used as good models for the variability in observed in biological neural networks - under the assumption that the spike rates carry most of the information. Furthermore, we have seen that inhibitory neurons play a major role for stabilizing networks with randomly chosen connections and weights.

While our approach did not result in any novel neuroscientific theory (unsurprisingly,
as the goal of neurorobotics lies more in the verification and testing of neuroscientific models coming from neurophysiological research), we experienced that our synthetic approach to neuroscience frequently made us ask important questions. For example, when designing the reservoir, its stability became of practical importance.

7.2 Robotics: Insight and Discussion

7.2.1 Modularity

Our choice of using ROS as the communication middleware greatly increased the modularity of our system. It allows us to freely combine different robot models (physical or virtual) with different ways of running our LSM-based spiking neural network (NEST or SpiNNaker). Moreover, in contrast to previous systems (e.g. in (Richter et al., 2016)), new modules can easily be added by implementing a new ROS-node and subscribing to the ROS-topics that we defined. The Dynamic Vision Sensor (DVS) that was developed at the Institute for Neuroinformatics in Zurich can also offers a ROS interface and could thus be added to our sensorimotor system, e.g. for visual guidance of the arm trajectory. While this ROS-based modularity is nice for prototyping and research, possible industrial applications might require a stronger integration for safety and efficiency reasons. Since neuromorphic sensors like the DVS also use the address-event-representation, a more efficient integration to our system would be to forgo ROS and send their events directly to SpiNNaker.

Richter and colleagues proposed in (Richter et al., 2016) that their cerebellum model might be integrated with a cortical model by sending a corrective term that adds to the cortical forward-kinematic control signal. We would like to support this proposal. A possible way of integrating the two systems could be to add a ROS-SpiNNaker interface to their neural network specification and send the ROS-output to a topic on which our `lsm_node` can subscribe.

7.2.2 Scalability

Due to the design of our system it possesses excellent scalability. Since we are using the generic Roboy communication packages for interfacing with our physical robot, we can easily add new motors, joints and links, however we would also need more FPGAs.

Scalability is a core principle behind the SpiNNaker system, as their goal is to eventually install a 1-million core machine.

As a result of our addition of population support for the ROS-SpiNNaker interface’s output, it become possible to scale the number of readout neurons. The support of input populations is currently being worked on. The current system can represent one
scalar (int64) per interface as the Poisson spike rate of arbitrary many input neurons, but does not support the encoding of vectors in a single population. This could be achieved by having an input population of neurons with different tuning curves analog to the encoding scheme used by the Nengo neural simulator (https://www.nengo.ai/).

7.2.3 Real-Time Performance

A system is capable of running in real-time if it can guarantee a response within a specified time constraint. Typically this time window is small, making ‘real-time computing’ analogous to ‘computing without delay’. For simulations real-time capability means, that the simulated clock runs at the speed as the real clock. In contrast to other neuromorphic system like BrainScaleS which runs 10,000 times faster than the real clock, SpiNNaker is capable of running in biological real-time (Furber et al., 2014). This means, that if a neuron is simulated to be spiking every 100 ms then the SpiNNaker chip will send a spike event every 100 ms. The same does not apply for our model running with the NEST neuron simulator on a PC. For our network with N = 2000 reservoir neurons it took 25 s to simulate 100 ms of neural activity (with Intel Core i5-7200 CPU with four 2.5 GHz cores). Thus our model runs 250 times slower than biological real-time. For larger networks, this factor increases further.

Another source of delay is the ROS-middleware, which is non-real-time. In this case, this term is more about determinism than about performance. ROS is perfectly capable of running control loops of several hundred Hz which would be enough for our system. The reason why it isn’t real-time is that it can’t guarantee to respond at precise times. For industrial applications (less so for neurorobotics) this failure of perfectly timed responding can be as bad as a wrong response.

Currently, ROS 2 is in development and will feature real-time capabilities. By virtue of it being designed for downwards compatibility, we expect that we’ll be able to run our system with ROS 2.0.

After the polygone reduction of our model, the Gazebo simulation was almost running in real-time on a powerful PC.

7.2.4 Further lessons learned

Finally, here are some lessons learned as well as tips that might help others.

Joint Sensor

The RMK4 sensor PCBs have the drawback of having a small distance between their connectors, making the process of soldering on connection cables rather difficult, which
is one of the reasons why a new version of the PCB is currently under development in the Roboy team.

**Gazebo models**

In Gazebo robot models can be defined either in the Universal (sometimes 'Unified') Robot Description Format (URDF) or in the Simulation Description Format (SDF). While both use XML and have similar tags, they differ with respect to what they can be used for. Furthermore certain data fields are optional in SDF but necessary in URDF and vice versa. It is important to note, that certain Gazebo plugins work only in URDF format. One example is `ros_control` which can be only used in URDF because the required `robot_description` ROS parameter can only be set there. In summary, it has to be recommended for anyone intending to use Gazebo with ROS to use the URDF instead of the SDF format, contrary to intuition, as SDF is the main Gazebo model format.

**ROS**

When something doesn’t work although there doesn’t seem to be an error in your code, try sourcing your ROS-workspace (`source devel/setup.bash`). Deleting the build and `devel` folders and rebuilding the workspace by running `catkin_make` followed by sourcing often helps as well.
8 Conclusion

In summary, we were able to construct a closed-loop system capable of steering a Myorobotic single-joint arm as well as its digital twin in simulation. We refrain from using the word 'control' here, because our learning subsystem has not been finished and tested yet. The system is based on the concept of a Liquid State Machine (LSM) which was implemented as a spiking neural network and can be either run on the neuromorphic SpiNNaker system or with NEST as back-end on any PC. Based on insight from the field of computational neuroscience, we have analyzed and designed our neural network with regard to stability. Furthermore, we have implemented mechanisms for encoding proprioceptive sensor values from the robot into spiking behavior of the LSM’s input neurons and for decoding the spiking activity of the LSM’s readout neurons as firing rates.

The integration of many different hardware and software systems of which most are research instead of consumer oriented and are still in ongoing development, was a difficult and time consuming task. This emphasizes the need for a ready-to-use platform for research in neurorobotics. The HBP Neurorobotics Platform (NRP) promises to cover this demand by allowing researchers to define their neural models in PyNN without installation requirements and the risk of having to resolve dependency issues and run them either in simulation using the NEST neuron simulator or on the SpiNNaker neuromorphic hardware.

We believe that our work will be valuable for the NRP for several reasons. First of all, we constructed the simulation model of our Myorobotic arm in a way to be compatible with the NRP and are going to include it into the platform so that other neurorobotics researchers are able to use it for testing their own brain models with a simple tendon-driven musculoskeletal robot. Additionally, we will be investigating whether the ROS-SpiNNaker interface, which was originally developed by Stephan Reith and extended with population support by us, can be added to the NRP to allow running the brain models on SpiNNaker as well, instead of being restricted to NEST. Moreover, we intend to improve the ROS-SpiNNaker interface further, e.g. normalizing the decoded spike rates to Hertz and adding population coding for input data (especially for input-vectors).

Additionally to these tasks, future work will focus on finishing the learning sub-system (ideally with STDP learning) in order to test our sensorimotor system with
programming by demonstration tasks and to answer the question whether LSM-based spiking neural networks can be used for the control of non-linear tendon-driven roots. We expect that this will necessitate an in-depth analysis of which network parameters influence the reservoir’s capability to separate different input streams in the high-dimensional liquid state. Moreover, we intend to test and verify the multiplexing property of our LMS. Once successful for our single-joint arm, we plan on applying our system, which was designed with scalability and modularity in mind, to more complex Myorobotic systems like legs and possible even the full-humanoid robot Roboy. These redundant musculoskeletal systems could eventually allow us to repeat manipulandum experiments used in the field of human motor control (e.g. the seminal experiments by Reza Shadmehr and Ferdinando Mussa-Ivaldi from 1994 (Shadmehr and Mussa-Ivaldi, 1994))
Abbreviations

AER  Address-Event Representation. 19
ANNs  Artificial Neural Networks. 10
CNS  Central Nervous System. 72
FPGA  Field-Programmable Gate Array. 45
HBP  Human Brain Project. 21, 33
ISI  Inter-spike Interval. 50
LSM  Liquid State Machine. 2, 14, 63
NEST  Neural Simulation Tool. 13
NRP  Neurorobotics Platform. 21, 33, 45
PyNN  Python package for Neural Networks specification. 13
ROS  Robot Operating System. 2, 29, 32
SNNs  Spiking Neural Networks. 10
SpiNNaker  Spiking Neural Network Architecture. 21
List of Figures

2.1 Three neuron types. A: cortical pyramidal cell B: cerebellar Purkinje cell C: stellate cell in the cerebral cortex (Dayan and L. F. Abbott, 2001, p. 5) 6
2.2 Physiological basis for the neuron’s membrane potential (Kandel et al., 2013, p. 130) ................................................................. 7
2.3 Modeling the cell membrane with an electrical circuit (Gerstner et al., 2014, p. 11) ................................................................. 8
2.4 Time course of an action potential and its constituting ion conductances (Kandel et al., 2013, p. 158) ................................................................. 9
2.5 Response of a LIF neuron to input currents of different magnitudes (Jupyter notebook can be found in Thesis/LIF_model.ipynb in our thesis code repository) ................................................................. 12
2.6 Structure of a Liquid State Machine (LSM) (Maass, 2010) ................. 14

3.1 The origins of neuromorphic computing: From von Neumann to Mead Image credits: www.thefamouspeople.com and Wikimedia Commons . 19
3.2 SpiNNaker 48-chip development board (Workshop, 2013) ................. 21
3.3 Architectural components of a SpiNNaker chip (Furber et al., 2014) .... 22
3.4 SpiNNaker’s communication by multicasting forms a two-dimensional torus where each node/chip is directly connected to six neighbors (Meier, 2017a) ................................................................. 23

4.1 The Myorobotics project (Myorobotics - A framework for musculoskeletal robot development 2017) ................................................................. 27
4.2 The Myo-Muscle (Myorobotics - A framework for musculoskeletal robot development 2017) ................................................................. 27
4.3 Our robot setup as an implementation of the antagonistic VSA design principle (Wolf et al., 2016) ................................................................. 28

5.1 System components and their interfaces ........................................... 29
5.2 Number of robots running ROS ....................................................... 32
5.3 How to use the ROS-SpiNNaker Interface ........................................... 38
5.4 Exponentially decaying decoding filter function: \( f(x) = 0.1 \cdot x \cdot e^{2-x} \) ................................................................. 40
5.5 SpiNN-3 board ................................................................. 41
List of Figures

5.6 Single-joint Myorobotic Arm ........................................... 42
5.7 Myo-Bone (Dürscheid, 2017) ........................................... 42
5.8 Hinge Joint (Dürscheid, 2017) ........................................... 43
5.9 Relation of eigenvalue spectrum and E-I ratio ......................... 43
5.10 Assembly of the hinge joint ............................................ 44
5.11 ROS-node converting 12-bit joint encodings into joint degrees and radians 45
5.12 GUI for motor control (left), motor status (middle) and joint angles (right) 46
5.13 Simulation models ....................................................... 47
5.14 Effect of adding tendon wrapping to the robot model .................. 48

6.1 System architecture ...................................................... 49
6.2 Simple network of two neurons with varying neuron models ........ 54
6.3 Two output neurons with different weights ........................... 55
6.4 Different firing behaviors of Izhikevich neurons ...................... 56
6.5 Relation of eigenvalue spectrum and E-I ratio ....................... 57
6.6 Inhibition stabilization of the reservoir ................................ 59
6.7 Scaling stabilization of the reservoir .................................. 59
6.8 Variance and mean scaling stabilization of the reservoir ............. 60
6.9 Input scaling for large reservoirs ...................................... 61
6.10 Spike train rate decoding by convolution .......................... 65
6.11 Convolutional rate decoding with different kernels ................. 67
6.12 ROS-graph of the system .............................................. 68
6.13 Network stability analysis .............................................. 70

7.1 Human motor control (image from (Burdet, Franklin, et al., 2013, p. 6)) 71
7.2 Mechanoreceptors (image from (Burdet, Franklin, et al., 2013, p. 22)) 73
7.3 LSM as a computational model for generic cortical microcircuits. Images from (Maass, 2010) ............................................. 74
List of Tables

5.1 Overview of employed the software tools ............................................. 30
5.2 Overview of the software packages’ interdependencies ......................... 31
6.1 Combined weight matrix ................................................................. 58
6.2 Network parameters ........................................................................ 62
6.3 Decoded mean firing rates ................................................................. 66


