Design and Implementation of a Brain-Computer Interface System

Bastian Venthur

Design and Implementation of a Brain-Computer Interface System

Dipl.-Inform. Bastian Venthur aus Berlin

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Klaus Obermayer
1. Gutachter:	Prof. Dr. Klaus-Robert Müller
2. Gutachter:	Prof. Dr. Benjamin Blankertz
3. Gutachter:	Prof. Dr. Stefan Jähnichen
4. Gutachter:	Prof. Dr. Gerwin Schalk

Tag der wissenschaftlichen Aussprache: 18.3.2015

Berlin, 2015

D83

Abstract

In this dissertation we present the results of our effort to provide a modular, completely free- and open source brain-computer interface (BCI) system in Python.

While other BCI systems exist, they are usually monolithic and written in a low level programming language like C++. Monolithic BCI systems cover the whole BCI processing chain from data acquisition, to signal processing, and feedback- and stimulus presentation, but they also force the user to use the *whole* system. This is problematic, as a researcher who is specialized, for example, in methods development will probably already have a toolbox of algorithms in a programming language he is familiar with. Being forced to use a monolithic system, means he has to migrate the tools and algorithms to the monolithic system first, and then work in an unfamiliar environment.

Being written in C++, those BCI systems also make it unnecessary hard for non-computer scientists to add new functionality to those systems. Even among computer scientists, C++ is notorious for the skill level required to write decent C++ code. And in the field of BCI, only a small part of researchers are actually computer scientists. Those BCI systems try to minimize the need to write C++ code, by providing a large set of amplifier drivers, toolbox methods and paradigms, but since a large part in BCI research is in developing new- or improving existing methods or paradigms, writing code is inevitable.

With our BCI system we want to solve those problems altogether. We present a modular BCI system that is written in Python, an easy to learn yet powerful programming language fit for scientific computing. Our BCI system consists of three independent parts that can be used together as a complete BCI system, or independently, as a component in existing BCI systems.

In this thesis we will cover for each component the requirements, the design aspects that lead to the implementation of the component, and demonstrate how to use the component individually in their domain. At the end of this thesis, we will demonstrate how to combine all three components to a complete BCI system and perform a real-time online experiment.

Or system is free- and open source software licensed under free software licenses. The source code is freely available.

Zusammenfassung

In dieser Dissertation stellen wir die Ergebnisse unserer Arbeit vor, deren Ziel es war ein modulares und komplett freies- und quelloffenes Brain-Computer Interface (BCI) System in Python zu erstellen.

Andere BCI Systeme existieren bereits, jedoch sind diese meist monolithisch und in einer maschinennahen Programmiersprache wie C++ geschrieben. Monolithische BCI Systeme decken die komplette Datenverarbeitungskette eines BCI Systems von der Signalakquise, über Signalverarbeitung und Feedback- und Stimuluspräsentation ab, jedoch zwingen sie auch den Benutzer das *gesamte* System zu benutzen. Dies ist problematisch da ein Wissenschaftler, der zum Beispiel in der Signalverarbeitung spezialisiert ist, wahrscheinlich bereits eine Sammlung von Algorithmen in einer ihm vertrauten Programmiersprache haben wird. Gezwungen zu sein in einem monolithischen BCI System zu arbeiten, bedeutet er muss seine Algorithmen und Methoden zum System migrieren und anschließend in einer ihm unvertrauten Umgebung arbeiten.

In C++ geschriebene Systeme machen es außerdem für nicht-Informatiker unnötig schwer neue Funktionalität in diese Systeme hinzuzufügen. Selbst unter Informatikern und gut ausgebildeten Programmierern ist C++ berüchtigt für die Fähigkeiten die es einem abverlangt vernünftigen C++ Code zu schreiben, und im Feld von BCI besteht nur der kleinste Teil der Wissenschaftler aus Informatikern oder ausgebildeten Programmierern. Monolithische BCI Systeme versuchen dieses Problem zu minimieren, indem sie bereits eine große Anzahl an Treibern für Verstärker, Algorithmen, Methoden und Paradigmen bereitstellen. Da jedoch ein großer Teil der BCI Forschung gerade aus der Entwicklung und Verbesserung von Methoden und Paradigmen besteht ist, ist das Programmieren eben jener fast unausweichlich.

Mit unserem System möchten wir all diese Probleme lösen. Wir stellen ein modulares BCI System vor, das in Python geschrieben wurde. Python ist eine einfach zu lernende aber mächtige Programmiersprache, die bestens für wissenschaftliche Programmierung geeignet ist. Unser BCI System besteht aus drei unabhängigen Komponenten, die zu einem vollständigen BCI System kombiniert werden-, oder als unabhängige Komponenten in existierenden BCI System verwendet werden können.

In dieser Arbeit werden wir für jede Komponente die Anforderungen abdecken, die Designaspekte durchgehen, die zur jeweiligen Implementierung geführt haben und demonstrieren wie man die Komponente in ihrer jeweiligen Domäne benutzt. Am Ende der Arbeit werden wir außerdem alle drei Teile zu einem kompletten BCI System kombinieren, und zeigen wie wir ein Echtzeit BCI Experiment durchführen.

Unser System ist freie- und quelloffene Software und lizenziert unter freien Software Lizenzen. Der Quellcode ist frei verfügbar.

Acknowledgements

First, I want to thank Prof. Klaus-Robert Müller for encouraging me to "be brave" and start a Ph.D.. If it wasn't for him, I would probably enjoy this cold, lazy Sunday afternoon playing guitar or reading a book. Instead, I'm sitting here with a tingly feeling in my stomach, writing the last lines of my Ph.D. thesis. Not only did Klaus convince me to become a Ph.D. student, he also provided a great working environment for me and many other Ph.D. students to conduct our research. He gave us a lot of freedom and made a lot of things possible. I appreciate that, and I am very grateful for the time I had in this lab.

The next in the list is Prof. Benjamin Blankertz. In the last years, I learned a lot from Benjamin, on a professional- and personal level. The way Benjamin accepts other people and their views, and follows his own moral standards without feeling the need to push his views on others, truly amazes me. Benjamin has been a great role model in the past, and his example will continue to have an impact on my decisions in the future.

I also thank Prof. Stefan Jähnichen and Prof. Gerwin Schalk for kindly agreeing to be my referees for this thesis.

I would also like to thank the co-authors of my publications for helping me to improve my papers. Particularly, I'd like to thank my good friends Johannes Höhne and Sven Dähne for patiently explaining me the gory details some BCI methods. It has been a great pleasure to collaborate with you!

I've also met a lot of great people during the years in our lab, and some deserve a special mention. Claudia has been a great source of encouragement for me, especially in the beginning of my Ph.D. where it wasn't clear to me at all, if I was doing the right thing or not. Martijn, who be came a very good friend over the years, and with whom I spent a lot of fun times together. Maci, with whom I lived and worked together for almost three fantastic years. Javi, someone to discuss the finer points of programming in C++ or Python, being vegan or evil, and much more. Dominik, since he came to our lab, my life instantly became a lot easier. And thanks to Dom I had, hands down, the most awesome birthday parties in the last four years.

There are a lot more that deserve to be mentioned for one reason or another. Unfortunately space on this page is limited: Stefan, Mikio, Ulf, Laura, Matthias, the other Matthias, Han-Jeong, Markus, Sula, Márton, Imke, Adrien, thanks for all the fish!

I also thank my friends and family for their support, especially my mother, who always supported me in all my endeavors. Her seemingly unshakable belief in my abilities is a curious thing, but sometimes it really helped.

And last but not least: Julchen, for being such a cheerful spirit. The last years have been wonderful with you and I really can't wait for the next to come.

Contents

List of Figures

Chapter 1

Introduction

1.1 A Primer to Brain-Computer Interfacing

A Brain-Computer Interface (BCI) is a system that measures central nervous system activity and translates the data into an output, suitable for a computer to use as an input signal. Or slightly less abstract: A BCI is a communication channel that allows for direct control of a computer by the power of thoughts. Although laymen often think of BCI as being a very new field, the concept and term "Brain-Computer Interface" was introduced by Jacques J. Vidal in 1973 already [Vidal, 1973].

BCI, in a nutshell, works by measuring brain signals, analyzing and interpreting the measured data, and translating the interpretation into actions [Dornhege et al., 2007; Wolpaw and Wolpaw, 2012]. An example for a BCI system could be a player who is sitting in front of a pinball machine with the task to control the flippers of the pinball machine with his thoughts [Tangermann et al., 2008]. The players' brain activity could be measured using Electroencephalography (EEG). For that he is wearing an EEG cap with electrodes, which are measuring the current on the scalp of the subject, induced by the brain activities under the scalp. The EEG data is amplified and feed into a computer. The computer receives the continuous data stream for each EEG channel and interprets the data somehow. For now we treat this part as a black box and assume the computer is able to translate the subject's imagined left or right hand movement into a signal that says "left" or "right" whenever the subject imagines the respective hand movement. The computer is connected to the pinball machine which will move the respective flippers whenever it receives the "left" or "right" signals. The subject is able to play pinball using BCI.

This example is a good, albeit spectacular, example for a closed loop, online BCI system. If we have a closer look at the example, we can divide the BCI system into three parts: The EEG system used to measure brain activity, is the *signal acquisition*. The signal acquisition is responsible for collecting the brain signals and translating it into a data stream suitable for the computer. The second part is the black box that translates the data into the output signal. Throughout this thesis we will call this part the *signal processing* part of the BCI system. The third and last part is the pinball machine that translates the output of the signal processing into an action. In that specific setting we could call that part the *feedback*. A feedback can come in many colors and shapes: it can be the moving flipper of the pinball machine, or could it be a prosthesis [Müller-Putz and Pfurtscheller, 2008], a virtual keyboard [Farwell

and Donchin, 1988] or even the steering wheel of a car [Zhao et al., 2009]. The important feature of a feedback is, that it translates the output signal into some kind of action.

Not every BCI setup, however, requires a feedback. Often researchers are more interested in how the brain reacts to certain stimuli, and the feedback part is replaced by a *stimulus presentation*. In that case, the BCI system is not used to translate thoughts into action, but rather as a sophisticated measuring system. An example for such a setup could be an experiment where the researcher is trying to predict how well a subject can memorize, for example, Chinese characters. The subject is sitting in front of a computer monitor, wearing an EEG cap. The subject is presented with Chinese symbols and their translation and the subject's task is to memorize the symbols as good as possible while the EEG system is recording the brain activity. A few days later the subject could perform a pen and paper test without EEG, to find out which symbols were correctly memorized. The researcher can now try to correlate the (in-)correctly memorized symbols with certain patterns in the EEG signals that occurred while the subject was trying to memorize that specific symbol. If a correlation is found, one can try to predict the success of the learning while the subject is still doing it and thus provide additional feedback to the stimulus by letting the computer suggest the subject to try a bit harder. In that case the line between feedback and stimulus presentation is blurred as the application provides both, the stimuli and the feedback. Subsequently, we will call the last part of a BCI system *feedback- and stimulus presentation* throughout this thesis.

It is important to realize that a BCI system is not a mind reading device and cannot "read thoughts" whatsoever. The sobering truth is, a BCI system can only detect or classify very specific patterns of brain activity that the subject voluntarily or involuntarily produces. The two most common and successful ways to produce useful patterns for BCI are attention- and motor imagery based.

1.1.1 Attention Based BCI

Attention based BCI works by presenting the subject different stimuli. Those stimuli can be visual [Farwell and Donchin, 1988; Citi et al., 2008; Allison et al., 2010], auditory [Klobassa et al., 2009; Schreuder et al., 2010], or tactile [Müller-Putz et al., 2006], with the visual stimuli being the most commonly used ones.

Each stimulus is associated with a certain action, like the movement of a part of a prosthesis, or the selection of a letter from the alphabet. By focusing his attention on the desired stimulus and ignoring the others, the subject produces the brain patterns the BCI system needs to "interpret" his will and execute the desired action [Dornhege et al., 2007; Wolpaw and Wolpaw, 2012].

From the attention based BCIs, the visual attention based BCI is the most commonly used one, and here, two different brain patterns can be used: event-related potentials (ERP) and steady-state visually evoked potentials (SSVEP).

In ERP based BCI, the stimuli are presented successively and for a very short time. The duration of a presentation is usually a few milliseconds and the time between two stimuli about 100ms. When the stimulus the subject is focusing on appears (or gets highlighted,

intensified, etc.), the subject produces a specific brain pattern that is different from the brain patterns that arise from stimuli the subject is not focusing his attention on. The most prominent pattern is an event-related positivity in the centro-parietal areas around 300ms after the presentation of the stimulus. This phenomenon is called P300. The ERP based BCI system can (among others) recognize the P300 and thus differentiate between the "targets" (i.e. the stimuli the subject was attending to) and the "non-targets".

In contrast to ERP based BCI, where the stimuli are usually presented successively, in SSVEP based BCI they are presented continuously, all at the same time, and flickering with different frequencies between 6-30Hz. The subject chooses his target by focusing on a certain stimulus. The flickering frequency of the stimulus generates steady-state visually evoked potentials (SSVEP) with the same frequency in the visual cortex. For example, if the subject concentrates on a stimulus flickering with 23Hz, the SSVEP will also flicker with 23Hz. The BCI system can detect the stimulus the subject was attending to by comparing the frequency of the SSVEP with the ones of the stimuli.

Visual attention based BCI works very reliable across different subjects, has very high detection/classification accuracy, and allows in a speller based paradigm for a typing speed of roughly one sentence per hour. However, different visual paradigms might require the subject to have control over his eye movements in order to work properly, which is not always a given for patients, especially the ones suffering from locked-in syndrome. A solution could be gaze independent visual spellers [Treder and Blankertz, 2010], or the use of auditory- or tactile spellers which require no voluntary muscle activity. Another quite practical problem with visual spellers is that many people dislike the constant flickering on the screen and find it very exhausting to use over time.

1.1.2 Motor Imagery Based BCI

Whenever a muscle in the human body is voluntarily moved, oscillations in brain activity in the sensorimotor- and motor areas, the so called the sensorimotor rhythms (SMR), change. These changes are fairly localized, following the homuncular organization (Figure 1.1) of this cortical region [Woolsey et al., 1979]. The decrease of oscillations is called event-related desynchronization (ERD) and typically appear during movement or preparation of movement. The increase of oscillations is called event-related synchronization (ERS) and appears after movement or relaxation [Pfurtscheller and Lopes da Silva, 1999]. In fact, even *imagining* such movements produces very similar ERD/ERS patterns as the actual movements would [Pfurtscheller and Neuper, 1997].

Motor imagery based BCI works by making the subject *imagine* the movement of certain limbs, like grasping with the left- or right hand or, moving the feet and measuring the ERD/ERS patterns over the respective cortical areas. Each imagined limb movement is associated with a certain action that gets executed when imagining the respective limb movement.

Although the motor imagery paradigm seems very natural compared to the attention based BCI and the number of muscles a human can control promises complex actions that could be performed, the number of different actions that can *actually* be performed using motor

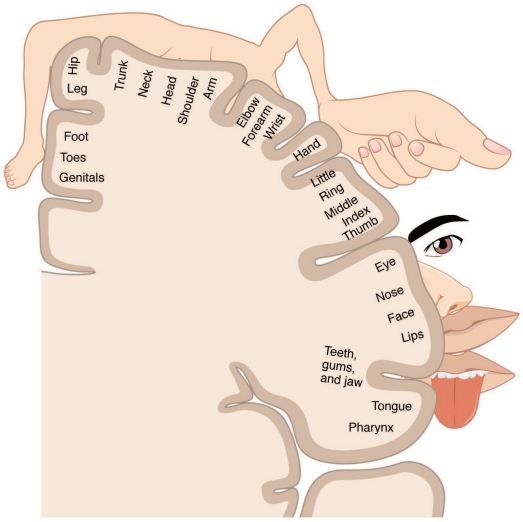

Figure 1.1: Cortical homunculus. The areas in the primary somatosensory- and motor cortex are ordered in a way that resembles a contralateral, upside-down representation of the body. (Image by OpenStax College, Creative Commons Attribution license)

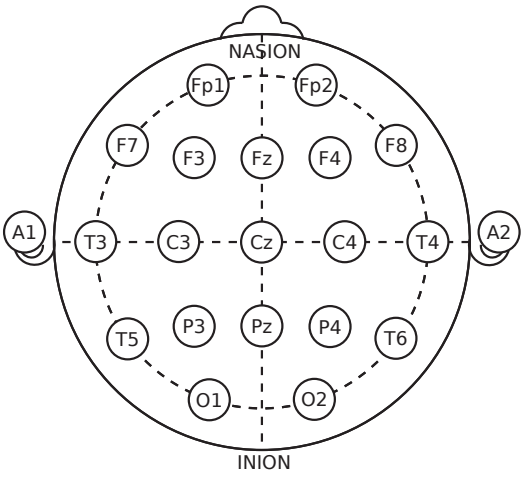

Figure 1.2: Naming and positioning of EEG electrodes according to the international 10-20 system [Jasper, 1958]. An electrode name consists of a letter and a number (or z for zero). The letters F, T, P, and O refer to the frontal-, temporal-, parietal-, and occipital lobes. Odd numbers refer to the left-, and even numbers to the right hemisphere. The numbering starts from the midline (z) and increases along the left- or right hemisphere.

imagery based BCI strongly depends on the way the brain signals are measured and not all limbs are suitable for motor imagery based BCI at all. The corresponding areas in the cortex have to be large enough to produce patterns that are distinguishable from the background EEG noise, and sufficiently far away from each other in order to make the ERD/ERS patterns distinguishable from each other. For example, when measuring brain activity using EEG, the ERD/ERS patterns for imagined left- and right hand movements are most prominent over electrode location C3 (right hand), and C4 (left hand). The cortical areas for left- and right foot on the other side, are very close (Figure 1.1) the corresponding patterns appear both on electrode location Cz (Figure 1.2). This makes them almost indistinguishable from each other using EEG. The comparatively low spatial resolution of EEG is the reason why usually only few (e.g. two or three) different actions can be used with motor imagery based BCI using EEG [McFarland et al., 2010].

Motor imagery based BCIs using invasive measurement of brain activity have shown a much higher number of actions that can be performed, with higher accuracy, and speed [Serruya et al., 2002; Velliste et al., 2008]. However, the risks associated with brain surgery, make invasive BCIs only suitable for a very small set of subjects, mostly patients that would need a brain surgery anyways.

Compared to attention based BCI, motor imagery based BCI has a higher error rate and an estimated 15% to 30% of subjects is not able to obtain control using motor imagery based BCI without training [Blankertz et al., 2010a]. Another major limitation for healthy subjects is that one cannot really use motor imagery based BCI while doing something else. As every movement the subject makes creates ERD/ERS patterns, the subject has to avoid any movement in order to use motor imagery based BCI properly.

1.2 Applications for BCI

The most important application for BCI is in assistive technology for severely physically disabled people. The classic (and sometimes the only) example, found in many BCI publications, are patients suffering from Amyotrophic Lateral Sclerosis (ALS) [Charcot, 1874] a neurodegenerative disease that slowly paralyzes the patient until he is completely locked-in in his paralyzed body. In the last stage, the patient is awake, but unable to twitch a single muscle in his body (locked-in syndrome). With BCI we can implement spellers [Farwell and Donchin, 1988; Birbaumer et al., 1999] that allow patients to communicate without moving a muscle. Those spellers are probably the most prominent application for BCI.

Apart from communication, mobility is another important application for BCI. Stroke patients might use BCI to visualize the current brain state in order learn how to suppress unwanted patterns [Daly and Wolpaw, 2008]. Or patients with spinal cord injuries can use BCI to control a prosthesis [Birbaumer and Cohen, 2007], a wheelchair [Galán et al., 2008], or even a telepresense device [Escolano et al., 2010]. However, the low information transfer rate of non-invasive BCI, which is measured in bits per *minute* and currently somewhere in the low two-digit range, is the main obstacle for complex applications using BCI. For example, a hand prosthesis that is controlled via BCI will typically not allow the patient to

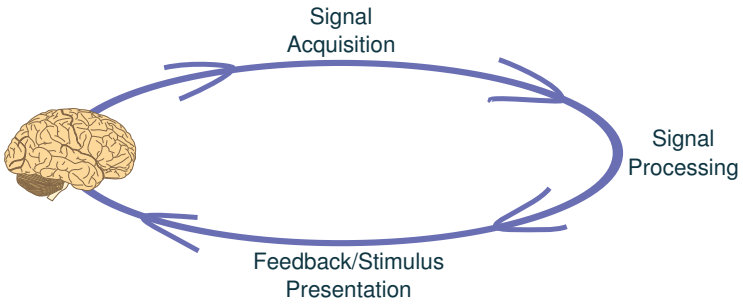

Figure 1.3: General structure of a closed loop BCI system. Brain signals (e.g. EEG data) are measured from the subject, and collected by the signal acquisition. The signal acquisition forwards the data to the signal processing where patterns are detected or classified and the result is forwarded to the feedback/stimulus presentation. When using BCI purely for stimulus presentation, the connection between signal processing and stimulus presentation is missing.

perform low-level movements like individual finger control, but only a very small set of high level operations like "grasp" and "open hand". A patient that is still able to voluntarily twitch a few muscles in his body will often be faster and more accurate using those muscles to control a device than using BCI [Mak and Wolpaw, 2009].

Apart from assistive technology, BCI has also sparked interest in many other fields [Blankertz et al., 2010b]. Studies have evaluated how BCI technology can be used as a measurement device to measure mental states like attention [Schubert et al., 2008; Haufe et al., 2011], or workload [Kohlmorgen et al., 2007; Müller et al., 2008; Venthur et al., 2010a] in order to predict and possibly prevent human failure in critical environments. BCI technology is used for quality assessment, by measuring subconscious perception of noise in auditory- or visual signals [Porbadnigk et al., 2010, 2011]. BCI can also be used for entertainment and gaming [Krepki et al., 2007; Nijholt et al., 2009] and the gaming industry has started to produce games that are "mind controlled". A field called neuromarketing [Ariely and Berns, 2010; Lee et al., 2007] studies how people respond to marketing stimuli in order to optimize marketing campaigns.

1.3 Components of a BCI System

As we saw in Section 1.1, a general BCI system consists of three parts: the signal acquisition, the signal processing, and the feedback- and stimulus presentation (Figure 1.3). In this section, we will introduce each part and their contribution to the BCI system.

1.3.1 Signal Acquisition

There are many ways to measure brain activity, but in most cases it is usually electrical or metabolic brain activities that are measured.

To the electrical measurement methods count the Electroencephalography (EEG) which measures the current on the scalp, Magnetoencephalography (MEG) which measures magnetic fields induced by electrical currents in the brain, Electrocorticography (ECoG) which measures electrical current *on* the cortex and microelectrode arrays which measure current *in* the cortex. EEG and MEG are non-invasive measurement methods, while ECoG and microelectrode arrays are invasive as they require surgery to place the electrodes on or in the brain. The Positron Emission Tomography (PET) measures chemical processes by measuring a positron-emitting radionuclide which was injected into the body of the subject. Near-Infrared Spectroscopy (NIRS) and Functional Magnetic Resonance Imaging (fMRI) measure the metabolic processes by measuring changes in the blood flow which are related to brain activity.

The above measurement methods differ largely in three important aspects: the spatial- and temporal resolution of the measurements and the size of the area of the brain that can be measured. The temporal resolution for microelectrode arrays, ECoG, EEG, and MEG is in the millisecond range, followed by NIRS and fMRI, both being in the order of seconds. PET has the worst temporal resolution of 10-100 seconds. The spatial resolution for microelectrode arrays is the best as it can measure up to single neurons, followed by ECoG, fMRI and PET which have a spatial resolution in the millimeter range. NIRS and MEG have a resolution of circa 5 millimeters, and EEG 1 centimeter and more. The coverage of the area that can be recorded is lead by EEG, PET, fMRI and NIRS which measure the whole brain, followed by MEG being able to cover large parts of the brain, ECoG which can cover small cortical areas, and micro arrays which cover a few hundred neurons.

A signal acquisition system is responsible for acquiring the brain signals and providing the data in a computer readable format. The data comes usually in a streaming format that allows for online processing, and in form of files on hard disk that allow for offline processing of the recorded data.

The signal acquisition consists of the hardware that actually measures the data (e.g. EEG cap and -amplifier), and the software that transports the data to the computer. Throughout this thesis we will be mainly concerned with the software part when talking about signal acquisition.

1.3.2 Signal Processing

The signal processing is the part of the BCI system that detects or classifies the brain patterns, that reflect the user's intend. The signal processing gets raw data from the signal acquisition and "translates" it into actionable output signals. Typically, the steps involved are: preprocessing, feature extraction, and detection or classification.

During preprocessing the data is often filtered, subsampled, and cleaned in a way that improves the signal to noise ratio. For example, one might want to ignore dead- or very

noisy channels, or epochs that contain muscle artifacts that would otherwise mask the much weaker brain signals.

The feature extraction transforms the high-dimensional data into much lower dimensional feature vectors that represent the characteristics and properties of the respective brain patterns properly and allow for detection or classification of them. The transformations necessary are highly dependent on the paradigm that is being used. For example, for motor imagery paradigms, the patterns used for BCI are found in the frequency domain [Pfurtscheller and Neuper, 1997], whereas the data from the signal acquisition is usually in the time domain and thus needs to be transformed. For ERP paradigms, where the data is already in the correct domain, it is usually not sufficient to look at the time course of the data, but one is more likely interested in the average values in specific time intervals [Blankertz et al., 2011].

After the data has been greatly reduced in dimensionality during the preprocessing and feature extraction, the signal processing can try to detect or classify certain brain states. Sometimes it is possible to simply "detect" the brain patterns by some sort of algebraic function that translates the feature vector into a scalar. More often nowadays is, however, the machine learning approach [Blankertz et al., 2002, 2007]. A classifier is trained using training data and known labels (e.g. "target", "non-target"), and the classifier learns the brain patterns that predict the labels best. After the classifier has been trained, it can decide, given new data, whether it belongs to one class or the other.

So far, we described the signal processing part of an online, closed loop BCI system where the raw brain signals are translated in real-time into actionable intentions. This is, however, only one use case. A big area of research in BCI is devoted to improving the overall BCI performance and the main lever, next to finding clever paradigms, is in the signal processing and machine learning. Researchers are constantly looking for better methods and better representations of the data in order to improve classification accuracy and information transfer rate. Therefore, a proper signal processing toolbox should also give the researcher the tools necessary to "dive" into the data. It should allow for loading data sets, playing with new methods, analyzing different aspects of the data, visualizing it, etc.. In fact, a BCI researcher will typically spend much more time on offline data analysis [Sajda et al., 2003; Blankertz et al., 2004, 2006; Tangermann et al., 2012] than on online classification.

1.3.3 Feedback- and Stimulus Presentation

The feedback- and stimulus presentation is the subject facing part of the BCI system. In the case of a closed loop, online BCI system, the feedback is the part of the BCI system that is "mind controlled" by the subject. It could be a speller program that is controlled using ERP based BCI, a game that is controlled by motor imagery, a prosthesis, etc.

In the stimulus presentation case, the subject is usually not directly controlling anything. He is receiving stimuli from the stimulus presentation while his brain activity is being recorded for later analysis. An example could be a software that lets the user perform a very complicated task in order to measure how his brain activity correlates with the error rate in his performance.

While the feedback- and stimulus presentation, in principle, could be standalone applications, they are usually embedded into a framework that allows, at least, for communication with the rest of the BCI software system. In the case of a closed loop, online system the feedback will usually receive the classification results from the signal processing in order to perform the appropriate action. Additionally, feedback- and stimulus applications will send markers to the signal acquisition soft- or hardware that give labels to specific points in time in the recording. Those markers are crucial for BCI, the signal acquisition forwards those markers along with the brain signals to the signal processing, so the signal processing knows, for example, *when* a stimulus was presented.

As mentioned before, feedback- and stimulus applications literally come in many colors and shapes. They range from simple applications running on a computer to special hardware like devices to apply tactile stimuli, multi speaker setups to apply auditory stimuli, telepresense robots, prostheses, or even complete virtual reality systems. Another big aspect of BCI is in developing new paradigms or enhance the existing ones in order to improve the information transfer rate, make BCI more usable for patients or simply to increase the knowledge about our brain.

1.4 Problem Statement

The main goal of the work described in this thesis is to provide a general purpose, complete, free- and open source BCI system, written in a modern programming language fit for scientific computing. Furthermore, the system shall run on all major operating systems.

With "complete" we mean a BCI system that covers all three parts of a BCI system: the signal acquisition, signal processing, and feedback- and stimulus presentation. "Free- and open source" means that the resulting software shall be free as in beer and speech: The software shall be freely available for everyone to use (beer) and it's source shall be free for anyone to study, modify and use as they desire (speech). Therefore, each component shall be licensed under the terms of a free software license and the code shall be made publicly available.

The target audience for our BCI system are mainly research laboratories in BCI and adjacent fields. In order to maximize the usefulness of our BCI system, the system shall be composed of three independent parts that can be either plugged into existing BCI systems or used together as a complete BCI system.

The main goals for each component of the system are the following:

- The signal acquisition shall provide a vendor agnostic interface to speak with different EEG hardware and support a reasonable number of amplifiers.
- The signal processing toolbox shall be suitable for real-time online experiments as well as for offline data analysis and visualization. The toolbox shall implement enough methods to support common BCI paradigms.

- The feedback- and stimulus framework shall provide a platform for running feedback- and stimulus applications in the context of BCI experiments. It should also provide a framework for writing feedback- and stimulus applications.

1.5 Why Python?

Development of the Python programming language started in the early nineteen-nineties. Since then, it attracted an increasing number of users and supporters in software industry and slowly gained traction in the scientific community over the last decade. Now, it has become a serious contender for the ubiquitous, and commercial Matlab. Python is currently amongst the most popular programming languages [Louden et al., 2011; Bissyandé et al., 2013] and has become an important platform for scientific computing. Open source projects like NumPy, SciPy [Oliphant, 2007], Matplotlib [Hunter, 2007], and IPython [Pérez and Granger, 2007] have become the foundation of scientific computing in Python and other projects like Scikit-learn [Pedregosa et al., 2011] for machine learning or Pandas [McKinney, 2012] for data analysis are building on top of them.

Python is free- and open source software, and runs on most platforms, which makes it attractive for research institutions and substantially lowers the entry barrier for newcomers to the field.

Stability of the language is another important aspect, especially in scientific computing, and the Python developers have shown a great sense of responsibility so far. Python's development cycle is driven by the Python Enhancement Proposal (PEP) process [Warsaw et al., 2000]. New major features are proposed in a PEP to seek the input from the Python community. PEPs are publicly discussed and commented on until a decision whether to accept or reject the proposal is made.

Python-releases come in three different types: Backwards-incompatible releases, major releases, and bugfix releases. In backwards-incompatible releases the first part of the version number is increased. Those releases happen infrequently (Python 1.0: 1994, 2.0: 2000, 3.0: 2008), and are expected to break existing code. Major releases increase the second part of the version number, introduce new features and are largely backwards compatible. Bugfix releases increase the third part of the version number and introduce no new features. Although the current Python version is 3.4.x, Python 2.7.x, the last major version of the Python 2.x branch, is supported with bugfix releases until 2020 [Peterson, 2008].

The predictable development cycle and reliable support of old Python versions gives researchers the confidence that the code base their research depends on, does not break from one day to another.

In addition to the above, we decided to use Python as the programming language of choice because we think it is an excellent general purpose programming language with a large and comprehensive standard library. Studies have shown that programming in high level languages like Python significantly shortens the amount of time needed to implement a solution [Prechelt, 2000] and leads to shorter and thus less error-prone code compared to more low level languages like C or C++ [Lipow, 1982]. This is particularly important for

software which is mostly going to be used and modified not by computer scientists, but by students and researchers from computational neuroscience, psychology and related fields.

1.6 Related Work

1.6.1 Complete BCI systems

Similar projects to create complete BCI systems exist, like BCI2000 [Schalk et al., 2004] or OpenViBE [Renard et al., 2010]. Both are complete BCI systems in a sense that they contain all three parts of a BCI system: the signal acquisition with drivers for various EEG hardware, the signal processing with many useful methods implemented, and feedback- and stimulus presentation with many paradigms. Unfortunately they are strictly coupled which means that the user cannot easily use only one part of those BCI systems and another part from another software package.

Both are written in C/C++ which gives better performance than Python, but makes it also much harder for non-computer scientists to write new toolbox methods, or feedback- and stimulus applications if the supplied ones are not sufficient. OpenViBE mitigates this by allowing for a drag- and drop like approach for visual programming of experiments and by providing Python and Matlab bindings. For BCI2000, there is BCPy2000 [Schreiner et al., 2008], which allows for writing BCI2000 modules in Python instead of C++, leveraging the infrastructure of BCI2000 without forcing the user to program in C++. PCPy2000 is free software licensed under the terms of the GPL.

BCI2000 runs only on Microsoft Windows, OpenViBE on Windows and Linux. BCI2000 is free only for non-commercial and educational usage, and OpenViBE is free software, licensed under the terms of the Affero General Public License (AGPL), a stricter variant of the GPL.

1.6.2 Signal Acquisition

When conducting BCI experiments one has to use special hardware that measures the brain activity and usually special software that acquires that data from that hardware. For example, when using EEG, one always has to use EEG amplifiers produced by commercial companies to acquire the brain signals. Those amplifiers usually come with their own proprietary software and different vendors have different formats for saving and online streaming of the EEG data. Often the software provided by the amplifier vendors only runs on Microsoft Windows systems, leaving out the Mac OS and Linux users.

Implementations of complete BCI systems, like BCI2000 and OpenViBE contain drivers for a lot of recording devices but as they are part of the respective BCI system, they cannot be easily used for recording only and processing in a different system. The closest thing to a stand-alone signal acquisition is Lab Streaming Layer (LSL) [Kothe, 2013]. LSL is a protocol to distribute time series data over network. It comes with drivers for eye

trackers, keyboards, and support for a dozen different EEG amplifiers. LSL is written in C and bindings for many programming languages are supported. LSL is free software and licensed under the terms of the MIT license.

1.6.3 Signal Processing

Many toolboxes have been developed over the years to cover the various needs and research interests. One of the oldest toolboxes is BioSig [Schlögl and Brunner, 2008] which is mainly for offline analysis of various biosignals, including EEG and ECoG data. BioSig is running in Matlab and Octave but experimental bindings for other programming languages exist. BioSig is free software licensed under the terms of the GPL. The BBCI toolbox (http://bbci.de/toolbox) is also a Matlab toolbox which has matured with age. The BBCI toolbox is suitable for online experiments and offline data analysis and has been open sourced in 2012. It allows complex online processing chains, e.g., acquiring data from several data sources that operate with different sampling rates, to use different feature extraction methods and classifiers simultaneously and to implement adaptive feature extraction and classifiers. FieldTrip [Oostenveld et al., 2011] is an open source Matlab toolbox for MEG and EEG analysis. It is relatively new but has already gained a lot of attention in the community. FieldTrip is free software licensed under the terms of the GPL. BCILAB [Kothe and Makeig, 2013] is the latest open source Matlab toolbox for BCI research. It supports offline analysis and online experiments. BCILAB is free software licensed under the terms of the GPL.

A few toolboxes exist also in the Python ecosystem. A relatively new toolbox is pySPACE [Krell et al., 2013], a signal processing and classification environment in Python. pySPACE has implemented many signal processing algorithms and allows for conducting experiments without programming by providing configuration files for each experiment. Due to its modular design it allows for implementing own algorithms as well. pySPACE is suitable for offline analysis and online classification and licensed under the terms of the GPL. MNE-Python [Gramfort et al., 2013] allows for offline analysis of MEG and EEG data and is available under the terms of the BSD license. SCoT [Billinger et al., 2014] is a special purpose toolbox for EEG source connectivity in Python licensed under the terms of the MIT license.

1.6.4 Feedback- and Stimulus Presentation

There are some comprehensive Python libraries that provide means for creating and running experimental paradigms. Vision Egg [Straw, 2008] is a high level interface to OpenGL. It was specifically designed to produce stimuli for vision research experiments. Vision Egg is free software and licensed under the terms of the LGPL. PsychoPy [Peirce, 2007] is a platform-independent experimental control system written in Python. It provides means for stimulus presentation and response collection as well as simple data analysis. PsychoPy is free software licensed under the terms of the GPL. PyEPL [Geller et al., 2007] is another Python library for object-oriented coding of psychology experiments which supports the presentation of visual and auditory stimuli as well as manual and sound

input as responses. PyEPL is free software and licensed under the terms of the LGPL. The Psychophysics Toolbox [Brainard, 1997] is a free set of Matlab and GNU/Octave functions for vision research. Being available since the 1990s, it is now a mature research tool and particularly popular among psychologists and neuroscientists. Currently, there is no principle framework to couple the Psychophysics Toolbox to a BCI system. The toolbox is open source software, which means that the source code is available, but it is unclear under which conditions one might use it.

In addition to this, there are also commercial solutions such as E-Prime (Psychology Software Tools, Inc) and Presentation (Neurobehavioral Systems).

For more in depth information on related BCI software, see [Brunner et al., 2013].

1.7 Outline of the Thesis

Our BCI system consists of three parts, and the next three chapters will introduce each part: Mushu, the signal acquisition (Chapter 2), Wyrm, the signal processing toolbox (Chapter 3), and Pyff, the feedback- and stimulus presentation framework (Chapter 4).

As each component of our system can be used independently without the others, those three chapters are relatively self contained and share a similar structure: In the beginning we will give an introduction and motivate the need for that component, we will then explain the requirements for the component, and explain the major design aspects that lead to the implementation. Since usability is an important aspect in this thesis, we demonstrate for each component how to use it individually in its specific domain. At the end of each chapter we will summarize, give an outlook for future work, and conclude.

After having introduced each component individually, in Chapter 5 we will combine all three components together to a complete BCI system and demonstrate how we used it to perform an online ERP speller experiment.

In the last chapter we will summarize the thesis and conclude.

1.8 List of Included Published Work

The following publications are in large parts included in this thesis:

1. Venthur, B., Scholler, S., Williamson, J., Dähne, S., Treder, M. S., Kramarek, M. T., Müller, K.-R., and Blankertz, B. (2010b). Pyff—A Pythonic Framework for Feedback Applications and Stimulus Presentation in Neuroscience. *Frontiers in Neuroinformatics*, 4(100)

2. Venthur, B. and Blankertz, B. (2012). Mushu, a free-and open source BCI signal acquisition, written in Python. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 1786–1788. IEEE

3. Venthur, B., Dähne, S., Höhne, J., Heller, H., and Blankertz, B. (2014). Wyrm: A Brain-Computer Interface Toolbox in Python. *Neuroinformatics*. under review

If a publication was fundamental to a chapter, it is stated in the introduction of that chapter.

1.9 List of all Publications

The following is a list of all publications (co-)authored by the author of this thesis.

Articles in Journals

1. Venthur, B., Scholler, S., Williamson, J., Dähne, S., Treder, M. S., Kramarek, M. T., Müller, K.-R., and Blankertz, B. (2010b). Pyff—A Pythonic Framework for Feedback Applications and Stimulus Presentation in Neuroscience. *Frontiers in Neuroinformatics*, 4(100)
2. Schaeff, S., Treder, M., Venthur, B., and Blankertz, B. (2012). Exploring motion VEPs for gaze-independent communication. *Journal of neural engineering*, 9(4)
3. Venthur, B., Dähne, S., Höhne, J., Heller, H., and Blankertz, B. (2014). Wyrm: A Brain-Computer Interface Toolbox in Python. *Neuroinformatics*. under review

Contributions to Conferences

1. Venthur, B. and Blankertz, B. (2008). A Platform-Independent Open-Source Feedback Framework for BCI Systems. In *Proceedings of the 4th International Brain-Computer Interface Workshop and Training Course 2008*, pages 385–389. Verlag der Technischen Universität Graz
2. Venthur, B. and Blankertz, B. (2009). A Platform-Independent Open-Source Feedback Framework for BCI Systems. 18th Annual Computational Neuroscience Meeting (CNS*2009)
3. Treder, M. and Venthur, B. (2009). (C)overt attention and P300-speller design. BCI Workshop on Advances in Neurotechnology
4. Venthur, B., Blankertz, B., Gugler, M. F., and Curio, G. (2010a). Novel Applications of BCI Technology: Psychophysiological Optimization of Working Conditions in Industry. In *Proceedings of the 2010 IEEE Conference on Systems, Man and Cybernetics (SMC2010)*, pages 417–421
5. Venthur, B. and Blankertz, B. (2010). Pyff—A Plattform-Independent Open-Source Feedback Framework for BCI Systems. 4th International Brain-Computer Interface Meeting

6. Schaeff, S., Treder, M., Venthur, B., and Blankertz, B. (2011). Motion-based ERP spellers in a covert attention paradigm. In *Neuroscience letters*, volume 500, page e11

7. Venthur, B. and Blankertz, B. (2012). Mushu, a free-and open source BCI signal acquisition, written in Python. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 1786–1788. IEEE

8. Venthur, B. and Blankertz, B. (2013). Towards a Free and Open Source BCI System written in Python. 5th International Brain-Computer Interface Meeting

9. Venthur, B. and Blankertz, B. (2014a). A Free and Open Source BCI System in Python. 12th International Conference on Cognitive Neuroscience

10. Venthur, B. and Blankertz, B. (2014b). Wyrm, A Pythonic Toolbox for Brain-Computer Interfacing. In de Buyl, P. and Varoquaux, N., editors, *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*. in press

Book Chapter

1. Brunner, C., Andreoni, G., Bianchi, L., Blankertz, B., Breitwieser, C., Kanoh, S., Kothe, C. A., Lécuyer, A., Makeig, S., Mellinger, J., Perego, P., Renard, Y., Schalk, G., Susila, I. P., Venthur, B., and Müller-Putz, G. R. (2013). BCI software platforms. In Allison, B. Z., Dunne, S., Leeb, R., Del R. Millán, J., and Nijholt, A., editors, *Towards Practical Brain-Computer Interfaces*, Biological and Medical Physics, Biomedical Engineering, pages 303–331. Springer Berlin Heidelberg

1.10 Notation

Throughout this thesis we will show a lot of code examples, mostly written in Python. Python code examples come in two different forms: they can show excerpts of Python files:

```
import time

def foo():
    """Say something and sleep a bit."""
    print 'Hello PhD thesis!'
    time.sleep(3)

if __name__ == '__main__':
    # helpful comment
    foo()
```

Or excerpts of interactive Python sessions:

```
>>> import numpy as np
>>> a = np.random.random((3, 2))
>>> a
array([[ 0.81945244,  0.34092702],
       [ 0.11069162,  0.34630389],
       [ 0.81385244,  0.43382821]])
>>> a - 1
array([[-0.18054756, -0.65907298],
       [-0.88930838, -0.65369611],
       [-0.18614756, -0.56617179]])
>>> def foo():
...     print "Hello again!"
...
>>> foo()
Hello again!
```

In interactive Python sessions, >>> marks the input prompt which is followed by a Python statement. If a statement produces an output, it is followed immediately after the statement without an input prompt as a prefix. Lines starting with ... are part of a code block that belong to the first line above that line that is prefixed by >>>.

Chapter 2

Signal Acquisition

2.1 Introduction

In this chapter we present Mushu, a vendor- and operating system agnostic signal acquisition software for EEG data.

When performing BCI experiments with EEG data, one always has to use special EEG hardware (i.e. EEG cap, and -amplifier) to acquire the brain signals. This hardware usually comes with their own software and different vendors have different formats for saving- and online streaming of the EEG data. Often the software provided by the vendors only runs on Microsoft Windows systems, leaving out the Mac OS and Linux users. Additionally to the EEG data, BCI experiments often require markers. Markers are little signals that mark a certain time point in the EEG data with a label. Currently there is no standard for sending and receiving markers. Different vendors provide different means to receive markers, sometimes requiring outdated hardware interfaces not available on modern computers anymore.

With Mushu we want to solve those problems altogether: we want a signal acquisition software that runs on all major operating systems, that supports a wide range of EEG hardware, that produces and outputs data in a standardized format independent from the amplifier used, receives markers via network interface, and that is free- and open source software.

This chapter is divided into the following parts: In the next section we describe the requirements for our signal acquisition software, in Section 2.3 we will explain the design of our solution. In the Section 2.4 we will give an overview of the supported amplifiers and in the sections Sections 2.5 and 2.6 we will show how to use Mushu as an end-user and how we reverse-engineered an amplifier in order to write a free- and open source amplifier driver for Mushu. In the last section we will summarize, give an outlook of future development directions and conclude chapter.

Parts of this chapter are based on a previous publication [Venthur and Blankertz, 2012].

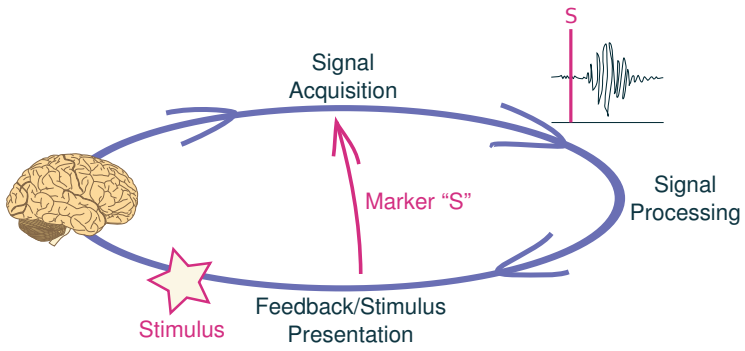

Figure 2.1: The feedback- and stimulus presentation presents a stimulus to the subject. At the same time, it sends a marker "S" to the signal acquisition. The signal acquisition marks the time when the marker was received and the marker itself in the recorded data and sends both to the signal processing.

2.2 Requirements

The most important task of a signal acquisition software is to stream the measured brain signals to the signal processing part of the BCI system in a way that allows for real-time, online processing of the data. Additionally to the streaming of data, the signal acquisition must be able to write the recorded data into a file for later processing. This is important for online experiments, where the classifier needs to be trained with pre-recorded training data, and for offline data analysis which is a major part in BCI research.

Most data acquisition software provided by amplifier vendors, streams and saves the data in their own format that is different from vendor to vendor. Consequently, parsers have to be provided or written in order to read the data. To mitigate this, Mushu shall stream and save its data in one unified format respectively, no matter what amplifier was used to record the data.

Another important feature are markers. Markers are labels for specific points in time within the EEG data (i.e. `(time=200ms, label='foo')`). They allow for finding the specific moment in the EEG time series where, for example, a stimulus was presented (Figure 2.1). The signal processing part of the BCI system uses those markers to locate the relevant part of the EEG stream for further processing. Most amplifiers support markers by providing an additional input channel and different vendors provide different means to achieve that. Some are using the USB port, others the parallel port (an ancient interface that was substituted by USB and is consequently not available on modern computers anymore), and some cheaper devices omit the possibility to receive markers completely. In order to support markers for *all* Mushu compatible devices, Mushu shall implement an *additional* way to receive markers via network interface. This will provide a unified way to send markers across all supported amplifiers and up value cheaper amplifiers that don't support markers natively.

Furthermore, Mushu shall run on all major operating systems.

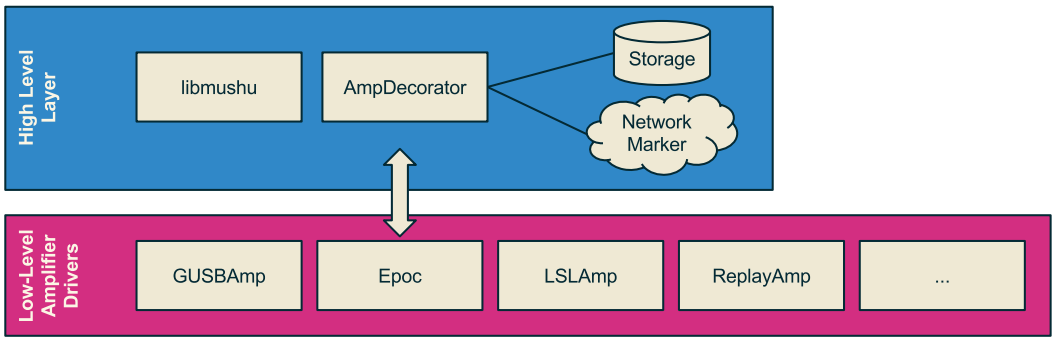

Figure 2.2: Mushu's architecture visualized in layers. At the bottom is the low level layer with the amplifier drivers, whose goal is to hide the complexities and differences of the supported amplifiers. At the top is the high level layer that adds Mushu specific features to the amplifier drivers.

2.3 Design

Mushu has a two layer architecture (Figure 2.2). The low level layer for the amplifier drivers, and a high level layer for Mushu specific features and the communication with the rest of the BCI system.

The low level layer is the realm of the `Amplifier` class. The `Amplifier` class defines the interface, a small set of methods, that is needed to use the underlying amplifier. Those methods include configuring, starting, stopping, getting data, and a few more. Device vendors can provide a way for Mushu to make use of their amplifiers by writing low level drivers in subclasses of the `Amplifier` class.

The high level layer adds the Mushu specific features to the low level amplifier drivers, like saving the data to disk or receiving additional network markers, by decorating instances of the `Amplifier` class. The high level layer also provides some convenience methods for the user to work with the amplifiers.

2.3.1 Low Level Layer

In order to support a wide range of EEG amplifiers we had to find a common interface that would make it possible to treat all amplifiers alike. What *all* amplifiers have in common is that they support a way to acquire EEG data regularly by means of polling, as this is their main purpose. *Most* amplifiers also allow for measuring the (electrical) impedance for each EEG electrode, so the experimenter can minimize the impedance before starting the actual measurements. *Some* amplifiers have built-in hardware filters than can be configured by setting the filter type, -order, and -coefficients. *Some* amplifiers allow for configuring the sampling frequency while others only support a fixed sampling frequency. *Some* amplifiers are ready to send data as soon as they are connected to the computer, others need to be explicitly put into a recording-mode.

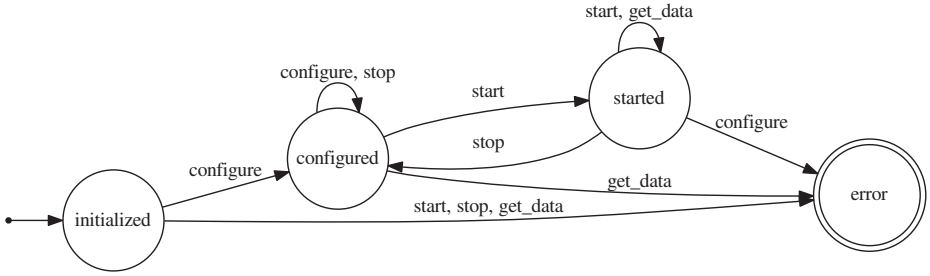

Figure 2.3: The amplifier life cycle visualized as a finite state automaton. The vertices represent the states and the edges are the method calls that lead to the state transitions.

We can generalize those use-cases in the following way:

When the amplifier is initialized it may or may not be in a *configured state*. In the *configuration* step the experimenter can set the mode (e.g. impedance, recording, etc.), sampling frequency, number of channels, etc.. The amplifier is now in a *configured state*. Within the configured state, the experimenter can *start* the amplifier to begin polling the amplifier for data. The amplifier is now in a *started state*. While in the started state, the experimenter can continuously acquire data from the amplifier. Within the started state, the experimenter can *stop* the amplifier to put it back into the configured state. Figure 2.3 shows the life cycle of an amplifier as a finite state automaton (FSA). The FSA has the states: initialized, configured, started, and error and the transitions: `configure`, `start`, `stop`, and `get_data`.

To demonstrate how actual amplifiers fit into our scheme we look at two very different amplifiers: the BrainAmp by Brain Products GmbH and the EPOC by Emotiv.

The Brain Amp is a medical grade EEG amplifier that allows for recording of up to 128 EEG channels with a sampling frequency up to 1kHz. It has two different modes: recording and impedance. During the configuration, it allows for setting sampling frequency, selection of channels, configuring filters, and more.

To measure the impedances we use the configuration step to put the amplifier into the impedance mode. Then, we start the amplifier and continuously acquire the impedance data until we finished preparing the subject and stop the amplifier. To start the recording of the EEG data, we configure the amplifier, which is still in the impedance mode, to put it into the recording mode. Then, we start the amplifier and continuously acquire EEG data until the experiment is finished and we stop the amplifier.

The EPOC amplifier is a consumer grade EEG system that supports 14 channels and has a fixed sampling frequency of 128Hz. Additionally to the 14 EEG channels, the EPOC has two channels for gyroscopes to measure head movements, a channel for the battery status, and a channel for a sample counter. The EPOC has no separate modes for impedance-

and EEG measurement, but provides the impedance for each channel along with the EEG recording in the normal recording mode. That sums up to a total of 32 channels (14 EEG, 14 impedance, 2 gyro, 1 battery, and 1 counter). The EPOC also starts to sample data immediately when it is connected to the computer and does not wait until explicitly configured and started.

Although the amplifier itself has nothing that resembles a configured- or started state, we can use the configuration step to emulate an impedance- and data mode. By setting the mode in the configuration step, the amplifier driver can switch states and only provide either impedance- or EEG data. To resolve the issue that the amplifier is always sending data no matter which state it is, we can simply ignore (i.e. throw away) all data from the amplifier when the driver is not in the started state.

Configuration issues

As we saw in the previous two examples, EEG amplifiers differ mainly in the configuration part. Some amplifiers don't need a configuration step at all, while most others differ highly in available modes and settings. One way to define a common interface for all amplifiers could be to find the lowest common denominator — a set of features that *all* amplifiers should support. This endeavor would be difficult and will likely lead to unsatisfying results, as amplifiers with many settings could only be used in a limited way, while amplifiers that have no options still don't quite fit into the configuration scheme. We decided it would be better to allow for maximum freedom in the configuration and decided to use Python's keyword arguments to allow for arbitrary options in the `configure` method.

When defining a function in Python, Python allows the function to have an arbitrary number of positional- and/or keyword arguments next to the required arguments defined by the function's parameter list. By using the `*` and `**` specifiers, the additional positional arguments will be collected in a tuple and the additional keyword arguments in a dictionary.

This kind of polymorphism is similar to function overloading found in programming languages like C++ or Java, but more flexible as it allows for a single implementation of a method to have an arbitrary number of arguments at run time.

To demonstrate that quickly, we define a function `f` that *requires* an argument `x`, and *allows* for additional positional arguments that are consumed by `args`, and additional keyword arguments consumed by `kwargs`. Our example function simply prints out the values for `x`, `args`, and `kwargs`:

```
def f(x, *args, **kwargs):
    print "x:", x
    print "args:", args
    print "kwargs:", kwargs
```

If we call `f` only with the required parameter, we see that `args` and `kwargs` are empty, as they are no additional positional- or keyword arguments:

```
>>> f(1)
x: 1
args: ()
kwargs: {}
```

If we call it with additional arguments we see that `2` and `3` are in the tuple `args`, and `foo` and `bar` in the dictionary `kwargs`:

```
>>> f(1, 2, 3, foo=21, bar=42)
x: 1
args: (2, 3)
kwargs: {'foo': 21, 'bar': 42}
```

The `configure` method of our `Amplifier` class uses only keyword arguments and thus requires no arguments at all but allows for an arbitrary number of keyword arguments:

```
def configure(**config):
    pass
```

This way we don't need to find a set of common options all amplifiers have to support, and allow the drivers to have arbitrary complex configurations.

A problem that arises, though, is that subclasses of `Amplifier` have identical interfaces, but the concrete amplifier objects may require completely different arguments and are thus not easily interchangeable anymore. A call of `configure(mode='recording', fs=1000)` might be useful for amplifier A, but could be totally useless for amplifier B.

To mitigate for the complexity of the configuration step, we introduced a `presets` attribute in the `Amplifier` class, that allows for storing the configuration parameters for some common settings. This dictionary has human readable names as keys and configuration dictionaries to pass to the `configure` method as values:

```
>>> amp.configure(amp.presets['impedance'])
>>> # or
>>> amp.configure(amp.presets['EEG 100Hz'])
```

This still does not solve the issue of blind interchangeability completely, but mitigates a bit. Graphical user interfaces (GUIs) can simply show the keys of the `presets` dictionary to the user and pass the value (i.e. the dictionary) of the selected item to the `configure` method. When exchanging `Amplifier` subclasses in an experiment script, we have to find the equivalent settings in the new amplifier and change the `configure` parameters accordingly.

This is clearly a limit to the goal to treat all amplifiers alike. We can unify the output formats and generalize the interfaces for the amplifier classes, but since different amplifiers support different features, it will never be possible to completely abstract their differences away by means of software engineering unless we are willing to offer only a common set of features that *all* amplifiers support.

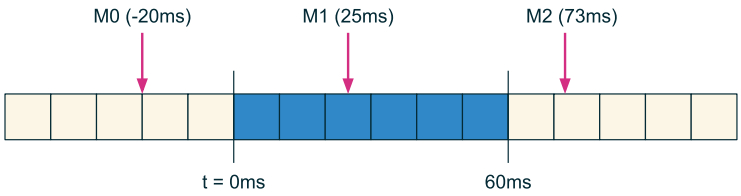

Figure 2.4: The timestamps of the markers (red) are relative to the beginning of the current block of data (blue). Markers that have been received earlier can have a negative value and markers that are received later can have a value that is bigger than the duration of the block.

We decided that it is more sensible to support all features of the expensive EEG hardware and accept the differences in the configuration step, than trying to fit all amplifiers in the same scheme by sacrificing features.

Acquiring Data

After the amplifier has been configured and put into the started state, we can regularly poll the amplifier for data.

The EEG data can be represented as a 2 dimensional array, with the first dimension corresponding to the samples, and the second one to the channels. We assume that the number of channels and the sampling frequency are constant after the amplifier has been put in the started state and we expect `get_data` to always return completely full arrays. With completely full we mean that each chunk of data contains the same amount of samples for each channel available. If the amplifier hardware does not provide the data in full chunks, it is the amplifier driver's responsibility to buffer appropriately.

The length of the data (i.e. the number of samples), can be variable between two calls of `get_data`.

The amplifier might also have received markers since the last call of `get_data`. In Mushu markers are a tuple consisting of a timestamp and a `string`. The `string` contains the label of the marker and the timestamp is the time in milliseconds relative to the onset of the block of data (Figure 2.4).

The return value of `get_data` is thus, a tuple consisting of the 2 dimensional NumPy array of EEG data and a (possible empty) list of marker-tuples.

Detecting Connected Amplifiers

Let's assume we have a large set of amplifier drivers already implemented in Mushu and want to test automatically which of the supported amplifiers are currently available (i.e. connected to the computer). Instantiating every single available `Amplifier` subclass, trying to configure-, start-, and getting some data is quite clumsy and not very efficient.

Therefore, we decided to add a class method `is_available` to the `Amplifier` class that returns a Boolean value of `True` when the amplifier is available or `False` otherwise. The default implementation of that method in the `Amplifier` class always returns `False`, subclasses can overwrite these methods to add specific tests for the respective amplifier. Being a class method, this method can be called directly on the `Amplifier` subclasses without instantiating an object from it.

The high level layer of Mushu (Section 2.3.2), builds on top of that, and provides a method called `get_available_amps` that checks all known `Amplifier` subclasses for availability and returns a list with classes for amplifiers that are currently available. This is a convenient method that can be used, for example, in GUIs that want to make the amplifier freely selectable, or for automatically picking the first amplifier that is available.

The Final Interface

Now that we discussed the functionality to be provided by the low-level drivers, we can define the list of methods and attributes for our `Amplifier` class.

- `configure(**kwargs)` configure the amplifier
- `start()` put the amplifier in the started state and begin buffering data sent by the amplifier
- `stop()` stop buffering data and put the amplifier back into the configured state
- `get_data()` return the samples and markers since the last call of `get_data`
- `get_channels()` return the list of channel names in the same order as the data from `get_data`
- `get_sampling_frequency()` return the sampling frequency
- `is_available()` static method to test whether amplifier available/connected
- `presets` dictionary with configuration presets

Amplifier drivers subclassing the `Amplifier` class will provide already basic signal acquisition functionality by implementing the above methods. They allow configuring the amplifier, acquire data and markers (using the amplifiers native marker input), and are able to stream the data in a unified data format.

In the next section we will see how we added the device independent network markers and save-to-disk features in Mushu's high level layer.

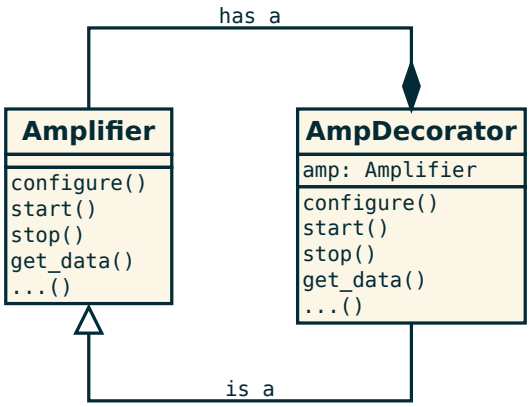

Figure 2.5: The `AmpDecorator` has an "is a"- and a "has a" relationship with the `Amplifier` class. By subclassing the `Amplifier` class, the `AmpDecorator` inherits the `Amplifier`'s interface (is a). By having an `Amplifier` instance as attribute, `AmpDecorator` can delegate its method calls to the attribute and decorate those if needed (has a).

2.3.2 High Level Layer

In this section we will show how we added the save-data-to-disk and network marker feature to all amplifiers subclassing the `Amplifier` class.

In order to add additional responsibilities to the `Amplifier` subclasses without the need to modify them or their superclass or even subclass existing `Amplifier` classes, we decided to use the Decorator design pattern [Gamma et al., 1994; Freeman et al., 2004]. The Decorator design pattern works as follows: We call the class to be decorated the "Component" and the decorator class "Decorator". We subclass the Component into the Decorator class so the Decorator inherits the Component's interface. Further, we add a Component attribute in the Decorator class and add the Component class to the Decorator's constructor parameters, so that the Decorator can initialize the Component. The Decorator has an "is a" and a "has a" relationship with the Component as visualized in the class diagram in Figure 2.5. In the Decorator we redirect all method calls we don't want to decorate to the Component attribute and override all of Component's methods we want to modify.

Our Python implementation of the Decorator pattern looks like this: The Component is our `Amplifier` class:

```
class Amplifier(object):

    def configure(self, **config):
        pass

    def start(self):
        pass
```

```
    def stop(self):
        pass

    # ...
```

The `AmpDecorator` is a subclass of `Amplifier`. It creates an `Amplifier` instance `amp` as attribute in its constructor `__init__` and delegates all undecorated method calls to this attribute:

```
class AmpDecorator(Amplifier):

    def __init__(self, ampcls):
        self.amp = ampcls()

    def start(self):
        self.amp.start()

    def stop(self):
        self.amp.stop()

    # ...
```

In order to use the decorated version of an amplifier we call `AmpDecorator` with the respective `Amplifier` class (not instance) as an argument to the constructor:

```
amp = AmpDecorator(Gusbamp)
```

The rationale behind the usage of the Decorator pattern here is to provide a mean to keep Mushu flexible without the need to touch the low level amplifier drivers. The amplifier drivers implement a minimal set of methods needed to use the amplifiers properly. Since the hardware is fixed, those drivers are unlikely to change. Mushu on the other hand might want to change the file format for saving data or provide an additional way for receiving markers. Those changes affect all supported amplifiers but should not require touching all their code. The Decorator pattern solves those requirements nicely.

Currently two additional features are implemented on top of the low level amplifier drivers using the Decorator pattern: saving recorded data to disk and receiving network markers.

Saving Data to Disk

Mushu saves three kinds of data to disk: the actual EEG data, the markers, and some metadata.

The EEG data is saved in binary form. Since the number- and ordering of EEG channels is fixed during a recording, we can write the values for each channel sample-wise to disk.

Markers are stored in text files, where each line contains the time stamp in milliseconds and the label. Since the marker's time stamps are relative to the current block of EEG data, and that information gets lost when storing all EEG data as a big block into a file, we have to make the time stamps relative to the beginning of the recording. Since the

sampling frequency is fixed during a recording we can easily translate the relative- to absolute time stamps (t_{rel} and t_{abs}), using the number of all received samples n_{r} and the sampling frequency f_{s}.

$$t_{\text{abs}} = 1000 \cdot \frac{n_{\text{r}}}{f_{\text{s}}} + t_{\text{rel}} \tag{2.1}$$

The metadata is stored as JSON files (more on JSON in Section 4.4.1), with the most crucial information being the sampling frequency, the channel names, and the order of channels in the EEG data files used in that recording.

Implementing the saving feature can be achieved by decorating the `get_data` method, in a way that the decorated `get_data` calls the Component's `get_data` and saves the data to disk before returning data and markers to the caller of the method:

```
def get_data(self):
    # get data and marker from underlying amp
    data, marker = self.amp.get_data()

    t = time.time()
    # length in sec of the new block according to #samples and fs
    block_duration = len(data) / self.amp.get_sampling_frequency()
    # abs time of start of the block
    t0 = t - block_duration
    # duration of all blocks except the current one
    duration = self.received_samples / self.amp.get_sampling_frequency()

    # save data to files
    if self.write_to_file:
        for m in marker:
            self.fh_marker.write("%f %s\n" % (1000 * duration + m[0], m[1]))
        self.fh_eeg.write(struct.pack("f"*data.size, *data.flatten()))
    self.received_samples += len(data)
    return data, marker
```

Adding Network Markers

Adding network markers can be solved similar to the save-to-disk feature, by decorating the `get_data` method of the `Amplifier` class.

This time we add a network server to our decorator class that runs concurrently in a separate process. Whenever the server receives a marker it measures the current time and stores the tuple `(timestamp, marker)`. Having the server in a second process ensures that the network server is not blocked by any long running operations of the amplifier driver and is thus capable of receiving markers as quickly as possible. Since processes, in contrast to threads, don't share memory, one process cannot directly read or write any variables of another process. In order to make two processes communicate with each

other one has to provide a mean of inter process communication (IPC). Python provides various data structures that achieve that goal, depending on the specific needs of the task at hand. We chose the `queue` data structure that allows two processes to share data in a Producer/Consumer pattern: While our network server process puts new `(timestamp, marker)`-tuples in the queue (producer), the decorated amplifier can receive those markers from the queue (consumer), convert the absolute time stamps to block-relative time stamps and merge them with the markers that might have been received directly by the amplifier (if any):

```
# ...
# merge markers
network_marker = []
while not self.marker_queue.empty():
    m = self.marker_queue.get()
    # timestamps from server are in seconds
    m[0] = (m[0] - t0) * 1000
    network_marker.append(m)
marker = sorted(marker + network_marker)
# save data to files
# ...
```

2.4 Supported Amplifiers

Currently Mushu supports two amplifiers (g.USBamp by g.tec medical engineering GmbH, and EPOC by Emotiv) directly by means of pure Python drivers. Those drivers have been written by reverse engineering- and emulating the protocols that those amplifiers use to communicate with the computer.

Mushu can also make use of the amplifier support provided by Lab Streaming Layer (LSL) [Kothe, 2013]. We implemented an `LSLAmp` that can connect to LSL streams on the network and thus supports circa a dozen different amplifiers like the BrainAmp by BrainProducs, or the Enobio by Neuroelectrics.

Mushu also provides replaying amplifier (`ReplayAmp`) that allows to load a previously recorded data set and replay it in real-time or time lapse. While replaying data in real-time is not a very common use case in BCI as the replay takes as long as the original recording, replaying data in time lapse can be useful to evaluate more complex methods, e.g., when some parameters of the feature extraction or the classifiers are continuously adapted. Furthermore, simulated online processing using the `ReplayAmp` can help the debugging, when online experiments did not work as expected from previous offline tests.

All the above amplifier drivers are platform independent and work on all major operating systems.

2.5 Using the Amplifier

In this section we will shortly demonstrate how to use Mushu as an end-user in a scenario of an online BCI experiment. We will load, configure, and start the amplifier and in the next step continuously query the amplifier for new data.

First we import the Mushu library:

```
import libmushu
```

In an interactive Python session we could query Mushu for a list of currently connected amplifiers, using the `get_available_amps` method:

```
libmushu.get_available_amps()
```

The method would return a list of strings like `['GUSBamp', 'RandomAmp', 'SinusAmp', 'ReplayAmp']`. Those strings can be used with the `get_amp` utility method provided by Mushu. `get_amp` is a factory method [Gamma et al., 1994; Freeman et al., 2004] and takes the name of the desired amplifier as argument and returns a decorated `Amplifier` object.

```
amp = libmushu.get_amp('GUSBAmp')
```

The next step is to configure the amplifier and start it.

```
amp.configure(fs=1000, mode='recording')
amp.start('testrecording')
```

When started with an optional string as parameter, the recorded data is saved to a file (in this case `testrecording`) otherwise the data is not saved.

The next step is to query the amplifier continuously for data and process it as needed. After the loop has been left, the amplifier should be stopped again.

```
while True:
    data, marker = amp.get_data()
    # further processing
    # break the loop if experiment finished
amp.stop()
```

This work flow is identical for all amplifiers supported by Mushu. The only difference is in the configuration step, where the experimenter has to adjust the specific settings for concrete amplifier being used.

2.6 Reverse-Engineering the g.USBamp USB Protocol

Using Mushu as an experimenter is Mushu's main use case. But in order to use Mushu if the desired amplifier is not in the list if supported amplifiers, one has to write the low level amplifier driver first. If the target platform is only Microsoft Windows and the vendors provide documented binary libraries (which they usually do), the task is straight forward using Python's `ctypes` library from Python's standard library (for an example of a complete implementation of an amplifier driver for Mushu, see Appendix A).

However, one of our concerns is to provide a platform independent BCI system that runs on all operating systems. For that reason we want to show briefly how we reverse engineered the g.USBamp USB protocol in order to write a platform independent, free- and open source driver for those devices in Python.

The g.USBamp is an EEG amplifier produced by g.tec medical engineering GmbH that is widely used in the BCI community. The amplifier is connected to the computer via USB and can record up to 16 EEG channels per amplifier. To record more than 16 channels, several amplifiers can be connected with a synchronization cable. g.tec provides an API and compiled libraries for Windows and Linux allowing to write own signal acquisition software. Unfortunately, those drivers have to be purchased by g.tec and cannot be redistributed along with the written software, which make those libraries unsuitable for free software. We decided to reverse engineer the USB protocol between the amplifier and the computer and implement our own platform independent and free driver.

Reverse engineering of a communication protocol for the sake of interoperability between software and a hardware device is lawful in the US and EU. However, in other regions, different laws might apply. We informed g.tec about the fact that we reverse engineered the protocol and they approved our project. However, they do not actively support it.

2.6.1 Setup

In order to observe the communication between the amplifier and the computer we connected the amplifier via USB to a Linux computer and ran a Windows XP instance inside a Virtualbox environment on the same computer. Virtualbox is a virtualization software allowing to run an arbitrary operating system (OS), like Windows XP, as a guest on a different host OS, like Linux. This happens transparently for the guest OS, which does not notice that it is running inside a virtual environment instead of real hardware.

Inside the Windows environment, we used the g.USBamp demo tool provided by g.tec which allows for reading the measured EEG data and modifying the various settings on the amplifier. For the g.USBamp tool it looked like it was communicating directly with the amplifier, but since the Windows XP system it was running on, itself ran in a virtualized environment, all communication was proxied through the Linux host system.

On the Linux host system we used a special kernel module, usbmon [Zaitcev, 2005] which allows for monitoring USB traffic between the host system and the USB device. Usbmon works analogous to network monitoring tools like tcpdump. With the help of usbmon we

were able to monitor all commands and data sent between the amplifier and the g.USBamp tool and thus had everything we needed to reverse-engineer the protocol.

2.6.2 Analysis of the USB Data

Without any prior knowledge about the protocol, we had to approach the analysis systematically. We divided the data to analyze into two categories: the commands sent to the amplifier, and data sent by the amplifier.

To analyze the commands sent to the amplifier, we used the g.USBamp tool to modify one setting at a time and recorded the data sent over USB to the amplifier. It turned out that most of the settings made in the g.USBamp tool translated into one USB request sent to the amplifier. When settings had different options (e.g. setting the sampling frequency), the parameter is either passed as a value to the USB request or inside the data buffer which is sent with every USB request. Decoding the requests was straight forward after we figured basic things like the endianness and word length of the data transferred. Describing all commands in detail in this chapter would be excessive and we kindly refer the reader to Mushu's g.USBamp driver we wrote, where every command is documented in detail.

Analyzing the data sent by the amplifier was more difficult. We expected to receive packets of fixed length as the number of channels is fixed and the amplifier is supposed to send the measured values for all available channels. Instead, we received packets of varying length with seemingly random numbers and had no clue how the data was ordered regarding the channels. Fortunately we noticed single zeros within the seemingly random data, appearing at a fixed period. It turned out that those zeros were the values for markers, and from that we could infer the ordering of the rest of the data. It turned out that the amplifier repeatedly sends its data with one measured value per channel: $ch_1, ch_2, \ldots, ch_{17}, ch_1, ch_2 \ldots$. One received packet from the amplifier contains not always exactly 17 (16 channels + marker) data points, but sometimes more and sometimes less. One has to buffer incomplete packets and concatenate the next packets, as the amplifier will always send the values in the above order. Moreover, the stream of values has no delimiter whatsoever to mark when all values for a given point in time were transferred, so one has to be careful not do drop any packages as it is not easy to determine which value belongs to which channel just by looking at the raw numbers.

After the analysis of the USB protocol between the g.USBamp and the g.USBamp tool we were able to implement our own driver for the amplifier using the Python library PyUSB, which provides USB access to the Python language.

2.7 Summary and Conclusion

In this chapter we presented Mushu, a signal acquisition software for EEG data in Python. We motivated the use for a signal acquisition that is amplifier- and operating system agnostic and then we defined the requirements for the software. We showed how we approached the problem with the design of a two layer architecture, dividing the problem

into the low level amplifier drivers and the Mushu specific features used by all amplifiers. We demonstrated how we used the decorator design pattern in order to add saving to disk and amplifier independent network markers to the amplifiers. We gave an overview of the supported amplifiers by Mushu and showed how to them as an end-user who wants to acquire EEG data for experiments. In order to help users who have to write their own drivers for Mushu, we briefly showed how we reverse-engineered the g.USBamp protocol in order to write a cross platform driver usable for Mushu.

Mushu is capable of streaming- and simultaneously saving EEG data, and is thus suitable for real-time BCI systems (we will prove that claim in Chapter 5). The data is provided and saved in a unified format that does not depend on the amplifier that is being used. Additionally to the markers supported by the respective amplifiers, Mushu also supports an additional, amplifier independent way to receive markers, which we will demonstrate in Chapter 5. Those network markers can greatly simplify the setup when using amplifiers that use outdated hardware like the parallel port for receiving markers, and they enable groups to use cheaper EEG hardware, that don't support markers natively, for BCI experiments.

Currently Mushu supports two amplifiers directly by means of pure Python drivers, a replaying pseudo amplifier, and over a dozen amplifiers by utilizing the LSL library. All those drivers are platform independent. Future work should focus on developing more drivers for Mushu by means of writing new ones or importing code from other open source projects.

Mushu is a platform independent software, which means it runs on all major operating systems, however, currently one can only use it within a Python environment, i.e. when the next part of the BCI system is also written in Python or if Mushu us used as a standalone recorder for EEG data. Mushu could be easily extended to support most other programming languages as well, by writing a thin wrapper around the Mushu library which forwards the data to a network socket. The receiving end of the connection could then use Mushu, no matter witch programming language it is written in.

Although Mushu currently only supports EEG amplifiers, it could also be used for other signal acquisition hardware, like NIRS, etc. by writing the appropriate low level drivers. The reason we only support EEG so far, is mainly because we didn't have access to other hardware than EEG in our lab.

Mushu competes mainly with commercial signal acquisition software provided by the amplifier vendors. When using those, users have the advantage that they use software that is optimized for the hardware at hand, and the whole package of soft- and hardware is often certified for medical use. Mushu does not provide those certificates, and probably never will. On the other hand, does Mushu provide a unified way to control amplifiers, and receive- and store data. On top of that, Mushu provides an interface to receive markers in a way that is independent from any external hardware.

Mushu is free- and open source software licensed under the terms of the GNU General Public License (GPL). Mushu's source is freely available at: http://github.com/venthur/mushu.

Chapter 3

Signal Processing

3.1 Introduction

In this chapter we present Wyrm, a signal processing toolbox for BCI.

The signal processing part of an online BCI system is responsible for translating the brain signals into actionable output signals by detecting certain patterns in the brain signals. In order to detect those patterns, the raw brain signals usually have to be preprocessed and specific features that represent those patterns best, have to be extracted and classified. The actual methods used to translate the raw brain signals to output signals differ highly from application to application, and a large part of BCI research is devoted to finding better methods or improving existing ones. Researchers are constantly looking for ways to improve the information transfer rate, classification accuracy, or the representation of the brain signals as feature vectors. Therefore, they spend a lot of time working with toolboxes that allow them to manipulate data, try out new methods, visualize different aspects of the data, etc..

Wyrm tries to cover both toolbox-aspects: Wyrm can be used as a toolbox for offline analysis and visualization of neurophysiological data, and in real-time settings, like an online BCI experiment. Wyrm implements dozens of different toolbox methods, which makes it applicable to a broad range of neuroscientific problems.

Since BCI researchers spend a lot of time with a toolbox, we designed the toolbox to encourage researchers to "dive" into their data and play with it: Its main data structure is very flexible yet easy to understand, and the toolbox methods follow a set of rules to keep the syntax and semantics consistent. Heavy use of unit testing for the toolbox methods gives users the confidence that the methods work as expected and the extensive documentation makes the toolbox easy learn and use. Wyrm also comes with example scripts for common BCI paradigms in online- and offline settings.

This chapter is divided into the following parts: in the next section we will explain the requirements for our signal processing toolbox. In Section 3.3 we will give a slightly technical overview of the toolbox, including the design of the main data structure, an overview of the methods, and some means of quality assurance we have taken. In Section 3.4 we show how to use Wyrm, by demonstrating how to perform the classification in two common BCI tasks: motor imagery using ECoG recordings and ERP speller using EEG recordings. In Section 3.5 we will demonstrate how to conduct a simulated online experiment using

Wyrm. In the last section we will summarize, give hints for future work, and conclude the chapter.

This chapter is based on a previous publication [Venthur et al., 2014]. The design and implementation of this toolbox was done by the author of this thesis, Hendrik Heller contributed code for the visualization as part of his bachelors thesis, Johannes Höhne and Sven Dähne helped implementing the methods for LDA-shrink, CSP, and SPoC.

3.2 Requirements

Before we start explaining our toolbox, we will have a look at its requirements. Like any proper BCI signal processing toolbox, Wyrm shall be suitable for offline data analysis of and for real-time, online BCI experiments.

Offline analysis contains tasks like post-experimental analysis of recorded data, visualization as well as experimenting with new techniques in order to improve, for example, the performance of classification algorithms. In online experiments the incoming stream of brain signals are classified in real-time manner, which requires additional data structures and modifications in some toolbox methods compared to their offline equivalent (e.g. the filter methods).

The toolbox shall implement a set of methods that make it suitable for common BCI tasks. The correctness of the methods provided by the toolbox is crucial, so all methods shall be extensively tested.

In order to make the toolbox easy to learn- and use, Wyrm shall provide extensive documentation and example scripts.

3.3 Design

In this section we will give an introduction into the technical details of the toolbox. We will explain the main data structure that is used throughout the toolbox, give an overview of the toolbox methods, discuss performance concerns, explain how we utilize unit testing as a mean of quality assurance, and how the extensive documentation is created.

3.3.1 Data Structures

In order to work efficiently with the toolbox, it is necessary to understand the toolbox' main data structure, dubbed `Data`. It is used in almost all methods of the toolbox and fortunately it is not very difficult. Before we can begin, we have to explain the terminology that is used in NumPy and thus throughout our toolbox for describing n-dimensional arrays. A NumPy array is a table of elements of the same type, indexed by a tuple of positive integers. The *dimensions* of an array are sometimes called *axes*. For example: an array with n rows and m columns has two dimensions (axes), the first dimension having the *length* n and

the second the length m. The *shape* of an array is a tuple indicating the length (or size) of each dimension. The length of the shape tuple is therefore the number of dimensions of the array. Let's assume we have an EEG recording with 1000 samples for 32 channels. We could store this data in a `[time, channel]` array. This array would have two dimensions and the shape (1000, 32). The time axis would have the length of 1000 and the channel axis the length of 32.

For the design of the data structure it is essential to take into account that the methods would deal with many kinds of data, such as continuous multi-channel EEG recordings, epoched data of different kinds, spectrograms, spectra, feature vectors, and many more. What all those types of data have in common is that they are representable as n-dimensional data. What separates them, from a data structure point of view, is merely the number of dimensions and the different names (and meanings) of their axes. We decided to create a simple data structure which has an n-dimensional array to store the data at its core, and a small set of meta information to describe the data sufficiently. Those extra attributes are: *names*, *axes*, and *units*. The `names` attribute is used to store the quantities or names for each dimension in the data. For example: A multi channel spectrogram has the dimensions: (time, frequency, channel), consequently would the `names` attribute be an array of three strings: `['time', 'frequency', 'channel']`. The order of the elements in the `names` attribute corresponds to the order of the dimensions in the `Data` object: the first element belongs to the first dimension of the data, and so on. The `axes` attribute describes the rows and columns of the data, like headers describe the rows and columns of a table. It is an array of arrays. The length of the `axes` array is equal to the number of dimensions of the data, the lengths of the arrays inside correspond to the shape of the data. For the spectrogram, the first array would contain the times, the second the frequencies and the third the channel names of the data. The last attribute, `units` contains the (preferably) physical units of the data in `axes`. For the spectrogram that array would be: `['ms', 'Hz', '#']`. Since the channel names have no physical unit, we use the hash (#) sign to indicate that the corresponding axis contains labels.

These three attributes are mandatory. It is tempting to add more meta information to describe the data even better, but more metadata also adds more complexity to the toolbox methods in order to maintain consistency. So there is a trade-off between completeness of information and complexity of the code. Since complex (or more) code is harder to understand, harder to maintain and tends to have more bugs [Lipow, 1982], we decided for a small set of obligatory metadata to describe the data sufficiently and make the toolbox pleasant to use, without the claim to provide a data structure that is completely self-explaining on its own.

Keeping the data structure simple and easy to understand was an important design decision. The rationale behind this decision was that it must be clear what is stored in the data structure, and where, to encourage scientists to not only look at the data in different ways, but also manipulate at it at will without the data structure getting in the way. It was also clear that specific experiments have specific requirements for the information being stored. Since we cannot anticipate all future use cases of the toolbox, it was important for us to allow the data structure to be easily extended, so users can add more information to the data structure if needed. Consequently, we designed all toolbox methods to ignore

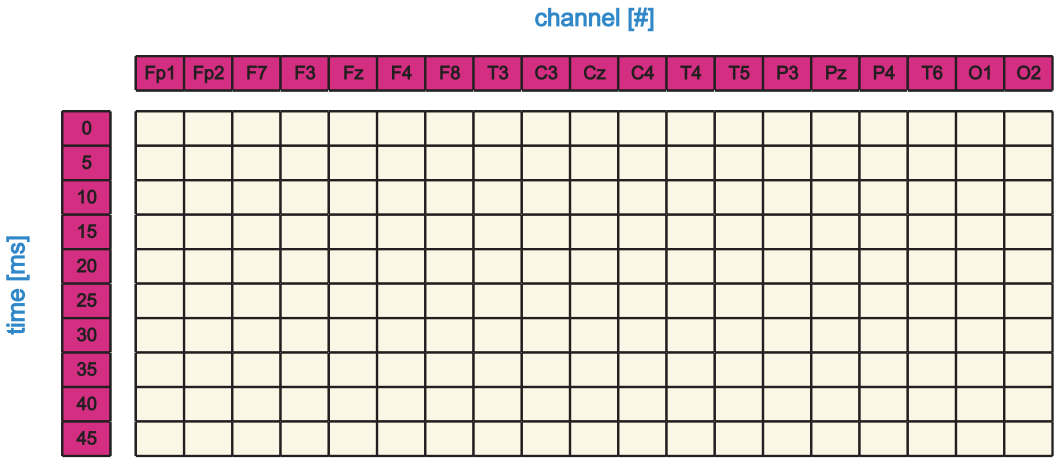

Figure 3.1: Visualization of the `Data` object and its attributes. In this example the data is two dimensional (yellow block). The `axes` (magenta) describe the rows and columns of the data and the `names` and `units` (blue) are the headings of the table.

unknown attributes and more importantly, to never remove any additional information from `Data` objects.

To summarize, Wyrm's main data structure (visualized in Figure 3.1), the `Data` class, has the following attributes: `.data`, which contains arbitrary, n-dimensional data, `.axes` which contains the headers for the columns of the data, `.names` which contains the of names the axes of `.data`, and `.units` which contains the units for the values in `.axes`. The `Data` class has some more functionality, for example built-in consistency checking to test whether the lengths of the attributes are compatible. This data structure is intentionally generic enough to contain many kinds of data, even data the author of this thesis did not anticipate during the design. Whenever additional information is needed, it can be easily added to the `Data` class by means of subclassing or by simply adding it to existing `Data` objects, thanks to the dynamic nature of Python.

Wyrm also implements two other data structures: a ring buffer and a block buffer. Those data structures are useful in online experiments and demonstrated in Section 3.5.

3.3.2 Toolbox Methods

Our toolbox implements dozens of methods, covering a broad range of aspects for offline analysis and online applications. The list of algorithms includes: channel selection, IIR filters, sub-sampling, spectrograms, spectra, baseline removal for signal processing; Common Spatial Patterns (CSP) [Ramoser et al., 2000; Blankertz et al., 2008], Source Power Co-modulation (SPoC) [Dähne et al., 2014], classwise average, jumping means, signed r^2-values for feature extraction; Linear Discriminant Analysis (LDA) with and without shrinkage for machine learning [Blankertz et al., 2011]; various plotting methods and

many more. Wyrm's `io` module also provides a few input/output methods for foreign formats. Currently supported file formats are EEG files from Brain Products and from the Mushu signal acquisition, reading data from amplifiers supported by Mushu, and two methods specifically written to load the BCI competition data sets used in Sections 3.4 and 3.5. The full list of methods and data structures implemented in Wyrm can be found in Appendix B.

It is worth mentioning that with scikit-learn [Pedregosa et al., 2011] you have a wide range of machine learning algorithms readily at your disposal. This list includes: cross validation, Support Vector Machines (SVM), k-Nearest Neighbours (KNN), Independent- and Principal Component Analysis (ICA, PCA), Gaussian Mixture Models (GMM), Kernel Regression, and many more. Our data format (Section 3.3.1) is compatible with scikit-learn and one can mostly apply the algorithms without any data conversion step at all.

Almost all methods operate on `Data` objects introduced in Section 3.3.1 and are responsible for keeping the data *and* the metadata consistent. While a few methods like `square`, `variance`, or `logarithm` are just convenient wrappers around the respective NumPy equivalents that accept `Data` object instead of NumPy arrays, the vast majority of methods implement a lot more functionality. For example the method `select_channels` requires a `Data` object and a list of strings as parameters. The strings can be channel names or regular expressions that are matched against the channel names in the `Data` object's metadata. `select_channels` will not only return a copy of the `Data` object with all channels removed that were not part of the list, it will also make sure the metadata that contains the channel names for the returned `Data` object is correctly updated. This approach is less error prone and much easier to read, than doing the equivalent operations on the data and metadata separately.

To ease the understanding of the processing methods, special attention was paid to keep syntax and semantics of the methods consistent. We also made sure that the user can rely on a set of features shared by all methods of the toolbox. For example: Methods never modify their input arguments. They create a deep copy of them and return a possibly modified version of that copy if necessary. This encourages a functional style of programming which, in our opinion, is well suited when diving into the data:

```
>>> from wyrm import processing as proc
>>> # per convention we import the processing module as proc
>>>
>>> dat2 = proc.some_function(dat)
>>> # dat is still the same and not modified
>>>
>>> dat = proc.some_other_function(dat)
>>> # dat was overwritten by the result
```

A method never touches attributes of a `Data` object which are unrelated to the functionality of that method. In particular, a method never removes additional or unknown attributes:

```
>>> # we create a new attribute on the fly:
>>> dat.some_obscure_attribute = "foo"
>>> dat2 = proc.some_function(dat)
>>> # the attribute is still present in the result of some_function
>>> dat2.some_obscure_attribute
```

```
"foo"
```

If a method operates on a specific axis of a `Data` object (Section 3.3.1), it adheres by default to our convention, but gives the option to change the index of the axis to operate on, by means of Python's default arguments. Those default arguments are clearly named as `timeaxis`, or `classaxis`, etc.:

```
>>> # create a copy of dat -> dat2
>>> dat2 = dat.copy()
>>> # swap the first two axes of dat2
>>> dat2 = proc.swap_axes(dat2, 0, 1)
>>> # channel axis is by convention the last one
>>> dat = proc.remove_channels(dat, ['Cz', 'O1', 'O2'])
>>> # if not, we have to tell `remove_channels` on which axis the channels are
>>> dat2 = proc.remove_channels(dat, ['Cz', 'O1', 'O2'], chanaxis=0)
>>> dat == dat2
True
```

In Sections 3.4 and 3.5, you will find some realistic examples of the usage of our toolbox and its methods.

3.3.3 Encapsulation

Careful readers will have noticed that our toolbox methods operate directly on a `Data` object's internal state. We effectively shifted the responsibility to keep data and metadata consistent, from the `Data` object to the methods operating on `Data`. This is a violation of the encapsulation principle: there is neither a mechanism in place that restricts the access to some of `Data`'s components, nor a language construct that bundles the methods operating on `Data` with the `Data` object. Usually this violation of the encapsulation principle would be a design error or at least a code smell. The `Data` object provides no form of contract and no class invariant other components of the software can rely on, allowing to (ab)use the `Data` object in unforeseeable ways. However, we have to keep in mind that a major use case for the toolbox is a researcher that interactively works with `Data` objects. He will experiment with modifications of the data and develop new methods that operate on `Data` objects. It lies in the very nature of this work flow, that that the researcher needs full access to all aspects of `Data`. A certain degree of freedom is required to allow the researcher to work with the toolbox effectively. Proper encapsulation would be possible, but would most likely result in a toolbox that is more cumbersome to use in the end.

In order to ensure that certain invariants are met, other mechanisms are in place. First, we have the conventions about the expected behavior of toolbox methods described in Section 3.3.2, and second, the correct behaviour of the toolbox methods is extensively tested as we will see in Section 3.3.5.

3.3.4 Speed

We realize that speed is an important factor in scientific computation, especially for online experiments, where one iteration of the main loop must not take longer than the duration of the samples being processed in that iteration. One drawback of dynamic languages like Python or Ruby is the slow execution speed compared to compiled languages like C or Java. This issue is particularly important in scientific computing, where non-trivial computations in Python can easily be in the order of two or more magnitudes slower than the equivalent implementations in C. The main reason for the slow execution speed is the dynamic type system: since variables in Python have no fixed type and can change at any time during the execution of the program, the Python interpreter has to check the types of the involved variables for compatibility before *every* single operation.

NumPy mitigates this problem by providing statically typed arrays and fast operations on them. When used properly, this allows for almost C-like execution speed in Python programs. In Wyrm all data structures use NumPy arrays internally and Wyrm's toolbox methods use NumPy or SciPy operations on those data structures. Additionally, we also carefully profiled our methods in order to find and eliminate bottlenecks in execution speed. Wyrm is thus very fast and suitable even for online experiments, as we will demonstrate in Section 3.5.

3.3.5 Unit Tests and Continuous Integration

Since the correctness of its methods is crucial for a toolbox, we used unit testing to ensure all methods work as intended. Unit testing became popular with JUnit (http://junit.org), a unit testing framework for the Java programming language, developed by Kent Beck and Erich Gamma. JUnit in turn, is a Java version of Kent Beck's testing framework for the Smalltalk programming language [Beck, 1994]. Nowadays, unit testing is a well accepted best practice in the software industry.

The concept of unit testing is to write tests for small, individual units of code (usually single functions or methods). These tests ensure that the tested function meets its design and is fit for use. Typically, a test will simply call the tested function with defined arguments and compare the returned result with the expected result. If both are equal the test passes, if not it fails. Well written tests are independent of each other and treat the tested method as a black box by not making any assumptions about how the function works, but only comparing the expected result for given inputs, with the actual one. Those tests should be organized in a way that makes it easy to run all tests at once with little effort (usually a single command). This encourages developers to run tests often. When done properly, unit tests facilitate refactoring of the code base (i.e. restructuring the code without changing its functionality), speed up development time significantly, and reduce the number of bugs.

To illustrate how unit tests work, let's assume we have a trivial method `is_odd` which tests for a given argument `n` whether it is odd or not:

```
def is_odd(n):
    return (n % 2) == 1
```

In the corresponding unit tests for this method, we test `is_odd` with some known even and odd numbers and the "corner case" for the value zero and compare the outputs with the expected results via the `assert*` methods. If the assertions are not satisfied, the corresponding test will fail:

```
import unittest

class TestIsOdd(unittest.TestCase):

    def test_odd_numbers(self):
        for i in -5, -3, -1, 1, 3, 5, 7, 9, 11:
            self.assertTrue(is_odd(i))

    def test_even_numbers(self):
        for i in -6, -4, -2, 2, 4, 6, 8, 10:
            self.assertFalse(is_odd(i))

    def test_zero(self):
        self.assertFalse(is_odd(0))
```

In our toolbox *each* method is tested respectively by a handful of test cases which ensure that the methods calculate the correct results, throw the expected errors if necessary, do not modify the input arguments, work with non-conventional ordering of axis, etc. The total amount of code for all tests is roughly 2-3 times bigger than the amount code for the toolbox methods. This is not unusual for software projects.

To automate the testing even further, we use a continuous integration (CI) service in conjunction with Wyrm's github repository. Whenever a new version is pushed to github, the CI will run the unit tests with three different Python versions (2.7, 3.3, and 3.4) to verify that all tests still pass. If and only if the unit tests pass with all three Python versions, the revision counts as passing, otherwise the developers will get a notification via mail. The whole CI process is fully automated and requires no interaction.

3.3.6 Documentation

A software toolbox would be hard to use without proper documentation. We provide documentation that consists of readable prose and extensive API documentation. The first part consists of a high level introduction to the toolbox, explaining the conventions and terminology being used, as well as tutorials how to write your own toolbox methods. The second part, the API documentation, is generated from special comments in the source code of the tool box, so called docstrings [Goodger and van Rossum, 2001]. External documentation of software tends to get outdated as the software evolves. Therefore, having documentation directly in the source code of the respective module, class, or method is an important mean to keep the documentation and the actual behavior of the code consistent. Each method of the toolbox is extensively documented. Usually a method has a short summary, a detailed description of the algorithm, a list of expected inputs, return values and exceptions, as well as cross references to related methods in- or outside the

toolbox and example code to demonstrate how to use the method. All this information is written within the docstring of the method (i.e. in the actual source code) and HTML or PDF documentation can be generated for the whole toolbox with a single command. The docstrings are also used by Python's interactive help system.

3.3.7 Python 2 versus Python 3

By the end of 2008 Python 3 was released. Python 3 was intentionally not backwards compatible with Python 2, in order to fix some longstanding design problems with Python 2. Since the porting of Python 2 software to Python 3 is not trivial for bigger projects, the adoption of Python 3 gained momentum only slowly. Although Python 2.7 is the last version of the 2.x series, it will still receive backwards compatible bug fixes and minor enhancements until 2020 [Peterson, 2008]. This is certainly a responsible decision by the Python developers, but probably one of the reasons for the slow adoption of Python 3. As of today, most of the important packages have been ported to Python 3, but there is still a bit of a divide between the Python 2 and Python 3 packages.

We decided to support both Python versions. Wyrm is mainly developed under Python 2.7, but written in a *forward compatible* way to support Python 3 as well. Our unit tests ensure that the methods provide the expected results in Python 2.7, Python 3.3, and Python 3.4.

3.4 Analyzing Experimental Data

To demonstrate the usage of our toolbox we describe the analysis and classification of two data sets from the BCI Competition III [Blankertz et al., 2006] using our toolbox. The scripts we will show are included in the `examples` directory of the Wyrm toolbox and the data sets are freely available on the BCI Competition III homepage (http://www.bbci.de/competition/iii). The reader can reproduce our results by using the scripts and the data sets.

The following code examples in this section follow our convention to import Wyrm's `processing` module as `proc`:

```
from wyrm import processing as proc
```

3.4.1 Motor Imagery in ECoG Recordings

The first data set uses Electrocorticography (ECoG) recordings, provided by the Eberhard-Karls-Universität Tübingen, and the Max-Planck-Institute for Biological Cybernetics, Tübingen, Germany, cf. [Lal et al., 2005]. The time series were recorded using a 8x8 ECoG platinum gird which was placed on the contralateral, right motor cortex. The grid covered the motor cortex completely, but also surrounding cortex areas due to is size of approximately 8x8cm. All data was recorded with a sampling frequency of 1kHz and the data was stored as μV values. During the experiment the subject had to perform imagined movements of

either the left small finger or the tongue. Each trial consisted of either an imagined finger- or tongue movement and was recorded for a duration of 3 seconds. The recordings in the data set start at 0.5 seconds after the visual cue had ended in order to avoid visual evoked potentials [Lal et al., 2005]. It is worth noting that the training- and test data were recorded on the same subject but with roughly one week between both recordings.

The data set consists of 278 trials of training data and 100 trials of test data. During the BCI Competition only the labels (finger or tongue movement) for the training data were available. The task for the competition was to use the training data and its labels to predict the 100 labels of the test data. Since the competition is over, we also had the true labels for the test data, so we could calculate and compare the accuracy of our results.

As part of the signal processing chain in this example, we employ a spatial filtering technique called Common Spatial Patterns (CSP) [Ramoser et al., 2000; Blankertz et al., 2008]. CSP spatial filters are applied to band-pass filtered data. The outputs of the spatial filters (sometimes also referred to as CSP components) are then used in subsequent processing steps. The main advantage of the CSP algorithm is that the filter coefficients are optimized to maximize the difference in variance between two classes. Trial-wise variance of band-passed filtered signals approximates the spectral power in the pass-band and thus improves the detectability of event-related (de-)synchronization (ERD/ERS), which in turn represents the basis for motor imagery BCI applications.

One formulation of the objective function that is maximized by CSP is given by

$$\mathbf{w} \quad = \quad \arg\max_{\mathbf{w}} \frac{\mathbf{w}^\top \left(\mathbf{C}_1 - \mathbf{C}_2\right) \mathbf{w}}{\mathbf{w}^\top \left(\mathbf{C}_1 + \mathbf{C}_2\right) \mathbf{w}} \,, \tag{3.1}$$

where $\mathbf{C}_1$ denotes the covariance matrix of all trials of class one, while $\mathbf{C}_2$ denotes the covariance matrix of class 2. The entire matrix of spatial filters can be obtained by means of the generalized eigenvalue decomposition of the matrices $(\mathbf{C}_1 - \mathbf{C}_2)$ and $(\mathbf{C}_1 + \mathbf{C}_2)$,

$$\left(\mathbf{C}_1 - \mathbf{C}_2\right) \mathbf{W} = \left(\mathbf{C}_1 + \mathbf{C}_2\right) \mathbf{W} \mathbf{D} \,, \tag{3.2}$$

where the diagonal matrix $\mathbf{D}$ contains the generalized eigenvalues. The first (last) columns of $\mathbf{W}$ maximize (minimize) the objective function given in Eq. (3.1) and thus they represent the spatial filters that extremize the class-wise variance difference. After the filter matrix has been obtained, the corresponding spatial patterns are computed via the relation

$$\mathbf{A} = (\mathbf{W}^\top)^{-1}. \tag{3.3}$$

The weights of the spatial filters are determined not only by the sources of interest (e.g. ERD/ERS sources) but also by task-unrelated noise contributions, which the filters aim to suppress. The spatial patterns, however, indicate how strongly the extracted source activity projects to the measurement channels. Thus, when visualizing the scalp topography of the

CSP components, one should inspect the spatial patterns (columns of **A**), rather than the spatial filters (columns of **W**) [Haufe et al., 2014].

For classification we will use Linear Discriminant Analysis (LDA) [Blankertz et al., 2011]. LDA is a simple and robust linear classification method which is frequently applied for BCI data. LDA assumes the data to follow a Gaussian distribution with all classes having the same covariance **C**. LDA seeks a linear projection **w** such that within-class variance is minimized while the between-class variance is maximized. For the binary scenario the optimal projection **w** is then given by

$$\mathbf{w} = \mathbf{C}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2), \tag{3.4}$$

$$b = \frac{1}{2}\mathbf{w}^\top (\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2) \tag{3.5}$$

For a new data point, a classifier output is computed with

$$y = \mathbf{w}^\top \mathbf{x} + b, \tag{3.6}$$

resembling the linear projection to the LDA separation hyperplane, where **w** is the normalvector of the hyperplane. Due to the bias term b, the classification boundary is set to $y = 0$. Thus, data points with $y < 0$ are classified as class 1 (e.g. target or left-hand imagery) and data point with $y > 0$ are classified as class 2 (e.g. nontarget or right-hand imagery).

After initial conversion from the epoched data in Matlab format into our `Data` format, we preprocessed both the training and test data in the following way: First the data was 13Hz low-pass- and 9Hz high-pass-filtered and subsampled to 50Hz. Note that we used the `filtfilt` method here, which implements a non-causal forward-backward filter. This is only feasible in offline analysis where the complete data set is available from the beginning. For online experiments one has to use the `lfilter` method which implements a regular IIR/FIR filter (cf. Section 3.5).

```
fs_n = dat.fs / 2
b, a = proc.signal.butter(5, [13 / fs_n], btype='low')
dat = proc.filtfilt(dat, b, a)
b, a = proc.signal.butter(5, [9 / fs_n], btype='high')
dat = proc.filtfilt(dat, b, a)
dat = proc.subsample(dat, 50)
```

After filtering and subsampling, we calculated the Common Spatial Filter (CSP) on the training set:

```
filt, pattern, _ = proc.calculate_csp(dat)
```

In the next step we perform the spatial filtering by applying the CSP filters to the training- and test data to reduce the 64-channel data down to 2 components. `apply_csp` by default uses the first and last spatial filter (i.e. columns of the `filt` argument). If more or other spatial filters are needed one can overwrite the `columns`-default argument.

```
dat = proc.apply_csp(dat, filt)
```

The last step of the preprocessing is creating the feature vectors by computing the variance along the time axis and the logarithm thereof:

```
fv = proc.variance(dat)
fv = proc.logarithm(fv)
```

Until here, the processing of training and test data is almost identical, the only difference being the calculation of the CSP filters on the training set only. In the next steps we will use `fv_train` and `fv_test` instead of `fv` to differentiate between the feature vectors of the training- and test data.

During the preprocessing we reduced the training data with the shape (278, 3000, 64) down to a feature vector with the shape (278, 2) – meaning each trial is represented by two numbers. Analogous, the test data was reduced from (100, 3000, 64) to (100, 2). After the preprocessing of training- and test-data, we can train the Linear Discriminant Analysis (LDA) classifier, using the feature vector of the training data and the class labels:

```
cfy = proc.lda_train(fv_train)
```

Applying the feature vector of the test data to the classifier, yields the projection of the test data on the hyperplane, trained by the `lda_train` method:

```
result = proc.lda_apply(fv_test, cfy)
```

The result is an array of LDA classifier outputs (i.e. one per trial), and we use the sign of each element to determine the corresponding class membership for each trial.

Analysis and Results

In Figure 3.2 we have visualized two CSP spatial patterns, which were also calculated during the computation of the CSP filter. We show the pattern for the imagined pinky movement (left pattern) as well as for the tongue movement (right pattern). Each pattern is an 8x8 grid, where each cell represents the respective electrode on the ECoG grid. The class-specific activation patterns show the spatially distinct regions that give rise to the strongest ERD/ERS during imagined movement of either the pinky or the tongue. See [Haufe et al., 2014] for a discussion about the interpretability of spatial patterns in contrast to spatial filters.

Comparing our resulting predicted labels with the true labels, showed that our method has an accuracy of 94% for that data set. The expected accuracy if classification is made by chance is 50%. That result is comparable with the results of the BCI Competition, where the first three winners reached an accuracy of 91%, 87%, and 86%. It is important to note that the goal here was not to “win” the competition, but to provide some context for the results we achieved. We had the advantage of having the true labels, which the competitors of the competition had not.

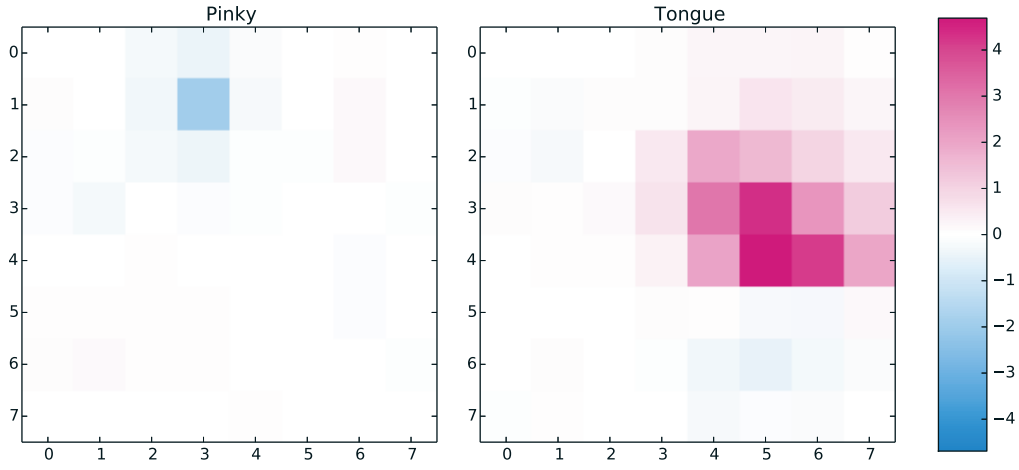

Figure 3.2: Spatial activation patterns of CSP components that show the strongest class-discriminative ERD/ERS for imagined pinky or tongue movement as measured on the 8x8 ECoG grid.

3.4.2 ERP Component Classification in EEG Recordings

The second data set uses Electroencephalography (EEG) recordings, provided by the Wadsworth Center, NYS Department of Health, USA. The data was acquired using BCI2000's Matrix Speller paradigm [Schalk et al., 2004], originally described in [Donchin et al., 2000]. The subject had to focus on one out of 36 different characters, arranged in a 6x6 matrix. The rows and columns were successively and randomly intensified. Two out of 12 intensifications contained the desired character (i.e. one row and one column). The event-related potential (ERP) components evoked by these target stimuli are different from those ERPs evoked by stimuli that did not contain the desired character. The ERPs are composed of a combination of visual and cognitive components [Brunner et al., 2010; Treder and Blankertz, 2010].

The subject's task was to focus her/his attention on characters (i.e. one at a time) in a word that was prescribed by the investigator. For each character of the word, the 12 intensifications were repeated 15 times before moving on to the next character. Any specific row or column was intensified 15 times per character and there were in total 180 intensifications per character.

The data was recorded using 64 channel EEG. The 64 channels covered the whole scalp of the subject and were aligned according to the 10-20 system. The collected signals were bandpass filtered from 0.1-60Hz and digitized at 240Hz.

The data set consists of a training set of 85 characters and a test set of 100 characters for each of the two subjects. For the training sets the labels of the characters were available. The task for this data set was to predict the labels of the test sets using the training sets and the labels.

After the initial conversion of the original data into our `Data` format, the data was available as continuous data in a `[time, channel]` fashion, with the markers describing the positions in the data stream where the intensifications took place.

In the first step, the data was 30Hz low-pass- and 0.4Hz high-pass filtered and subsampled to 60Hz:

```
fs_n = dat.fs / 2

b, a = proc.signal.butter(16, [30 / fs_n], btype='low')
dat = proc.lfilter(dat, b, a)

b, a = proc.signal.butter(5, [.4 / fs_n], btype='high')
dat = proc.lfilter(dat, b, a)

dat = proc.subsample(dat, 60)
```

In contrast to the ECoG data set, which was already in the epoched form, this data set is a continuous recording and has to be segmented into epochs. For segmentation we use the markers which define certain events (`MRK_DEF`) and "cut" the data around the time point defined by the marker and a segmentation interval (`SEG_IVAL`), in this case $[0, 700)$ ms around the respective marker onset, and assign each resulting chunk to a class defined by the marker definition. The resulting epoched data has the form `[class, time, channel]`:

```
epo = proc.segment_dat(dat, MRK_DEF, SEG_IVAL)
```

In order to receive good classification results for ERP classification tasks, it is a good strategy to calculate the means over certain time intervals (`JUMPING_MEANS_IVALS`) for each channel, instead of using the data set as is. The time intervals are highly subject specific and have to be chosen by using the classwise average, signed r^2 values or some other heuristic [Blankertz et al., 2011]. The number of intervals is usually between 3-6.

By appending the average values for each channel to a vector, we receive the feature vectors for each trial:

```
fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)
```

Now we can use again the LDA to train a classifier using the feature vectors of the training data and the labels and classify the feature vector of the testing data:

```
cfy = proc.lda_train(fv_train)
lda_out_prob = proc.lda_apply(fv_test, cfy)
```

The result is a LDA classifier output for each trial (i.e. intensification), predicting whether that intensified row or column was the one the subject was concentrating on. In order to get the actual letters the subjects wanted to spell, one has to combine the 15 classifier outputs for each row and column that the row/column has been intensified into one respectively and choose the most probable row and column. Each row-column combination defines a

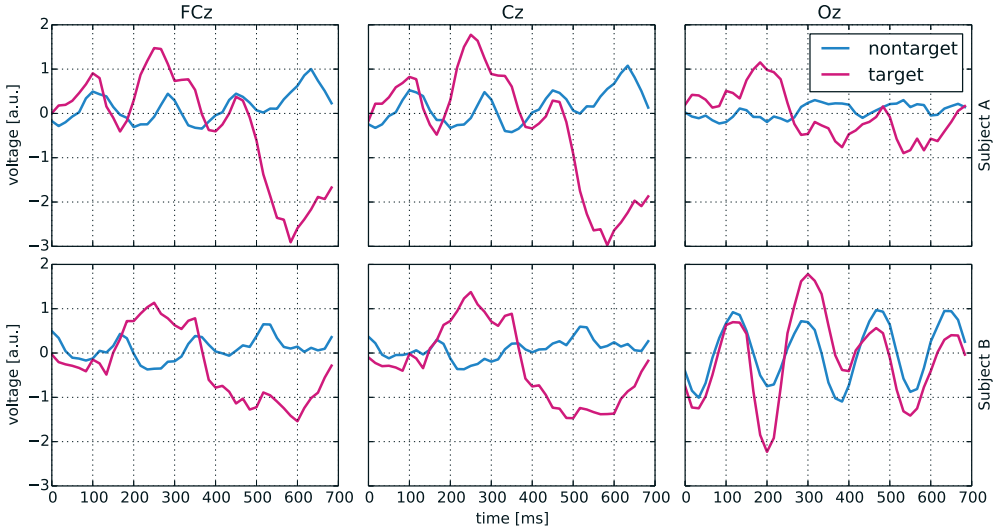

Figure 3.3: Classwise average time courses for subject A (top row) and subject B (bottom row) for three selected channels. The averages were calculated on the whole training set, t=0 ms is the onset of the stimulus.

letter which has been the one the subject was probably attending to. This "unscrambling" step has been omitted in this chapter for the sake of brevity but is available in the example script.

Analysis and Results

So far we did not explain how we choose the time intervals for the means for each subject. Figure 3.3 shows the classwise average time course for three selected channels (FCz, Cz, and Oz) and both subjects. As expected, we see for both subjects an early activation in the occipital areas around 200ms, followed by activation in the central and fronto-central areas. We also see that the kind of activation, especially in the occipital area, differs highly between the two subjects: subject A has a positive response on channel Oz around 200ms whereas subject B has a negative one. Moreover, subject B's Oz resonates much stronger at the frequency the stimuli were presented with than subject A. Not only is the inter-subject difference very large, also the variance of single time courses compared to the average is especially high for ERP experiments.

In order to quantify the discriminative information for each channel and time point, we compute the signed r^2-values, cf. [Blankertz et al., 2011]. Those r^2-values serve as univariate statistical measures for separability. The discriminative information across all channels and time points can then be visualized as a matrix (see Figure 3.4).

Comparing the signed r^2-values on Figure 3.4 between the two subjects, we see both

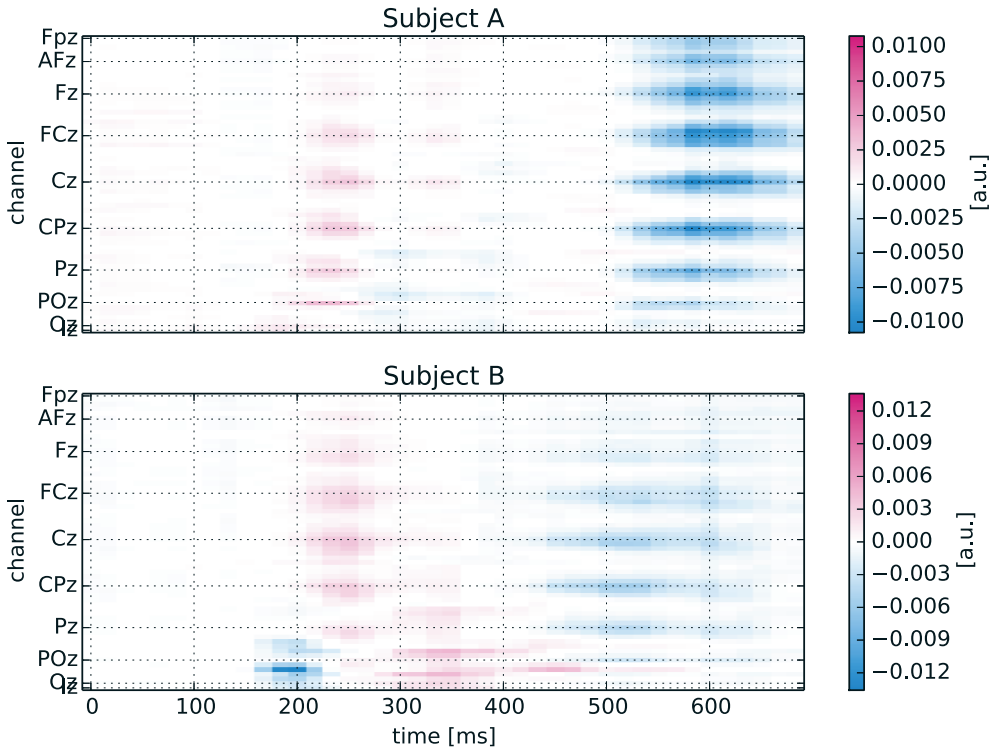

Figure 3.4: Signed r^2-values for subject A (top row) and subject B (bottom row). The channels are sorted from frontal to occipital and within each row from left to right. The blobs show the time intervals for each channel, which discriminate best against the other class.

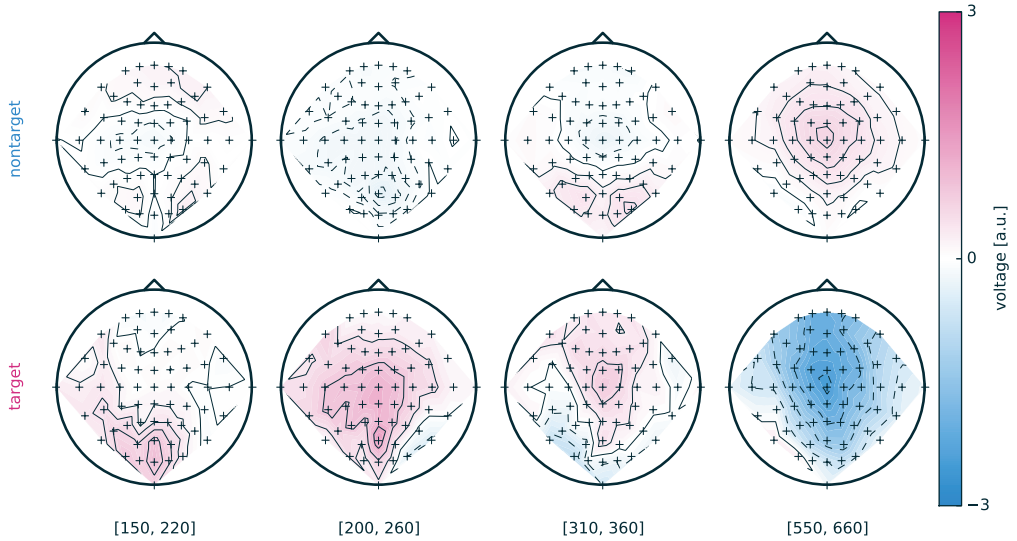

Figure 3.5: Spatial topographies of the average voltage distribution for the different time intervals used for classification for subject A. The top row shows the non-target trials, the bottom row the targets.

subjects feature a positive ERP component between 200 and 280ms after stimulus onset. This component is known as P300 component which is strongest in in the central- to frontal areas, as shown in Figure 3.3. Moreover, subject B displays a strong negative component (called N200) around 150-250ms after stimulus onset (Figure 3.6). This visual N200 component is mainly located in occipital areas.

In order to find the optimal time intervals for classification, we manually chose four intervals where the signed r^2 have their maximum or minimum and the respective other class does not change the sign on one of the other channels. For subject A, the intervals: 150-220ms, 200-260ms, 310-360ms and 550-660ms have been chosen; for subject B: 150-250ms, 200-280ms, 280-380ms and 480-610ms. The Figures 3.5 and 3.6 show the spatial topographies of the average voltage distributions in the selected time intervals we chose for classification.

Comparing the resulting letters, predicted by our classification with the real ones the subjects were supposed to spell, our implementation reaches an accuracy of 91,0% (91% for subject A and 91% for subject B) which is comparable with the results of the winners of the competition, where the first three winners reached an accuracy of 96,5%, 90,5%, and 90%. The expected accuracy if classification is made by chance is 2,8%.

Note that a much better classification can be achieved by a much simpler preprocessing method, namely: 10Hz low-pass filtering the data, subsampling down to 20Hz and just creating the feature vectors (without calculating the means over intervals):

```
B, A = proc.signal.butter(5, [10 / 120], btype='low')
```

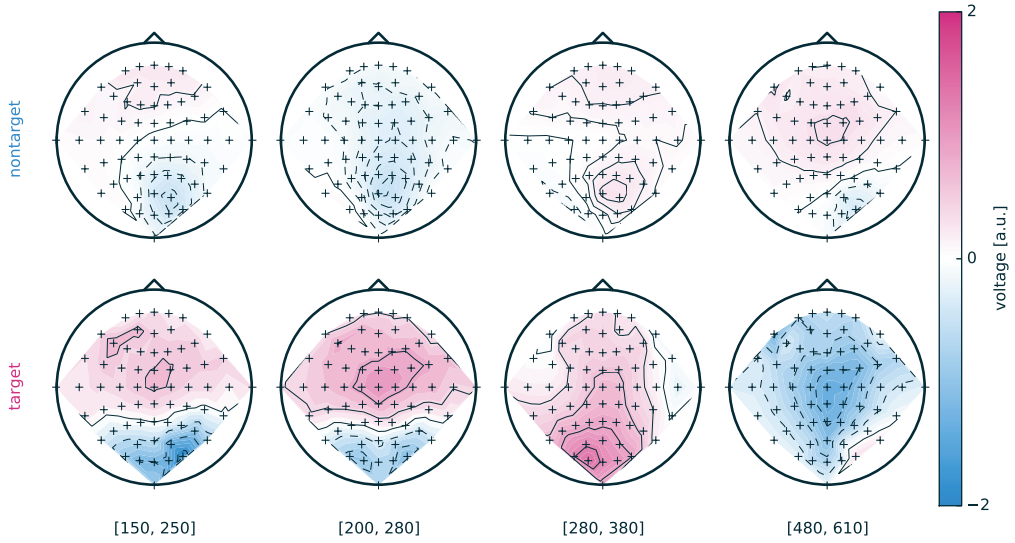

Figure 3.6: Spatial topographies of the average voltage distribution for the different time intervals used for classification for subject B.

```
dat = proc.filtfilt(dat, B, A)

dat = proc.subsample(dat, 20)
epo = proc.segment_dat(dat, MRK_DEF, SEG_IVAL)

fv = proc.create_feature_vectors(epo)
```

The results for that classification are 96% (96% for subject A and B). Due to the increased dimensionality of features, this approach requires a lot of training examples to work well and which are available in this data set. In practice, one aims at keeping the calibration short such that interval selection as explained above can be expected to work better.

3.5 Performing Online- and Simulated Online Experiments

In this section we will show how to use Wyrm to perform a simulated online experiment. To demonstrate the experiment we will use the ERP data set from Section 3.4.2, subject A and perform the classification task in an online fashion by using a software amplifier that reads data from a file and returns signals and markers in small chunks in real-time in exactly the same manner as when acquiring data from a real amplifier. This capability of realistically simulating online processing of Wyrm is not only good for demonstration but also for other purposes, see discussion in Section 3.6.

The principal processing steps and parameters for filtering, subsampling, etc., that lead to the classification are the same as in the offline experiment shown in Section 3.4.2, so we can focus here on the differences between the offline and online processing.

In order to simulate an online experiment with the available EEG data, we will use the `ReplayAmp` pseudo amplifier from the Mushu signal acquisition (Chapter 2). The pseudo amplifier can load a complete data set and its `get_data` method returns only as much data and markers as possible given the sampling frequency of the data and the time passed since the last call of `get_data`. From our toolboxes point of view this software amplifier behaves like a real amplifier. Using this `ReplayAmp` also makes the experiment reproducible for the reader as the Mushu signal acquisition, the online experiment script, as well as the data used, are freely available.

In contrast to the offline experiment, where the entire data set is available, in the online setting we have to process the incoming data chunk-wise. The chunks of data typically have a length of just a few samples (or blocks). This leads to differences in some of the processing steps:

When filtering the data chunk-wise, we have to use `lfilter` with filter delay values in order to receive the same results as if we were filtering the whole data set at once.

The subsampling from 240Hz to 60Hz internally works by returning every 4th sample from the data to be subsampled. When subsampling chunk-wise, we have to make sure that the data to be subsampled has a length of multiples of 4 samples in order to avoid losing samples between the chunks of data. For that we have to either set a block size of 4 samples (or an integer multiple of 4) in the amplifier or utilize Wyrm's implementation of a block buffer. Since most amplifiers allow for a configuration of the block size, we set the block size of 4 samples in the `ReplayAmp` as well.

If the amplifier does not support the configuration of the block size, one can use Wyrm's implementation of a block buffer. The `BlockBuffer` behaves like a queue, a first-in-first-out data structure, that is unlimited in size. The block buffer has two methods: `append` and `get`. `append` accepts a continuous `Data` object and appends it to its internal data storage. `get` returns (and internally deletes) the largest possible block of data that is divisible by `blocksize`, starting from the beginning of its internal data storage. After a `get`, the `BlockBuffer`'s internal data has at most the length `blocksize-1`. A subsequent call of `get` would return empty data, a subsequent call of `append` will append the new data to the remaining data in the internal representation and so on.

We will also utilize Wyrm's implementation of a ring buffer where we can `append` small chunks of data in each iteration of the online loop and `get` the last 5000ms of the acquired data to perform the classification on.

3.5.1 Training

The online experiment can be divided into the training part and the online part. In the first part, the training EEG data is recorded and after the recording is done, the entire training data is used for training the LDA classifier, much like in the offline setting. The signal

processing and training of the LDA classifier in the training part is identical to the signal processing and training of the LDA in the offline analysis in Section 3.4.2.

```
from wyrm import processing as proc
from wyrm import io
from wyrm.types import RingBuffer
import libmushu

cnt = io.load_bcicomp3_ds2(TRAIN_DATA)

fs_n = dat.fs / 2
b, a = proc.signal.butter(5, [30 / fs_n], btype='low')
cnt = proc.lfilter(cnt, b, a)
b, a = proc.signal.butter(5, [.4 / fs_n], btype='high')
cnt = proc.lfilter(cnt, b, a)
cnt = proc.subsample(cnt, 60)

epo = proc.segment_dat(cnt, MARKER_DEF_TRAIN, SEG_IVAL)

fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)

cfy = proc.lda_train(fv)
```

3.5.2 Online Classification

In the second part we use the classifier `cfy` obtained from the training, to classify the incoming data.

First we prepare the online loop. We load the test data set and provide it to Mushu's `ReplayAmp`. Note how we configure the amplifier to use a block size of four samples and set it into the real-time mode.

```
cnt = io.load_bcicomp3_ds2(TEST_DATA)
amp = libmushu.get_amp('replayamp')
amp.configure(data=cnt.data, marker=cnt.markers, channels=cnt.axes[-1],
              fs=cnt.fs, realtime=True, blocksize_samples=4)
```

Assuming we have an amplifier `amp`, we need to know the sampling frequency, the names of the EEG channels and the number of channels:

```
amp_fs = amp.get_sampling_frequency()
amp_channels = amp.get_channels()
n_channels = len(amp_channels)
```

Then we setup the ring buffer with a length of 5000ms.

```
rb = RingBuffer(5000)
```

We calculate the filter coefficients and the initial filter states for the low- and high-pass filters and put the amplifier into the recording mode.

```
fn = amp.get_sampling_frequency() / 2
b_low, a_low = proc.signal.butter(5, [30 / fn], btype='low')
b_high, a_high = proc.signal.butter(5, [.4 / fn], btype='high')

zi_low = proc.lfilter_zi(b_low, a_low, n_channels)
zi_high = proc.lfilter_zi(b_high, a_high, n_channels)

amp.start()
```

The actual online processing happens in a loop. At the beginning of each iteration we acquire new data from the amplifier and convert it into Wyrm's data format using the `convert_mushu_data` method provided by Wyrm's `io` module.

```
while True:
    data, markers = amp.get_data()
    cnt = io.convert_mushu_data(data, markers, amp_fs, amp_channels)
```

The remaining code samples from this section are *all* part of the loop. We removed the first level indentation from the loop for better readability.

We can filter the data using `lfilter` and the optional `zi` parameter that represents the initial conditions for the filter delays. Note how `lfilter` also returns the initial conditions for the next call of `lfilter` when called with the optional `zi` parameter:

```
cnt, zi_low = proc.lfilter(cnt, b_low, a_low, zi=zi_low)
cnt, zi_high = proc.lfilter(cnt, b_high, a_high, zi=zi_high)
```

The filtered data can now be subsampled from the initial 240Hz to 60Hz.

```
cnt = proc.subsample(cnt, 60)
```

Now we append the data to the ring buffer and query the ring buffer for the data it contains, thus we will always have the last 5000ms of acquired data. Before putting the data into the ring buffer we store the number of new samples in a variable as this number is needed later when calculating the epochs.

```
newsamples = cnt.data.shape[0]

rb.append(cnt)
cnt = rb.get()
```

In the next step we segment the 5000ms of data. Since the difference between the 5000ms of data from *this* iteration and the 5000ms from the *previous* iteration is probably only a few samples, we have to make sure that `segment` returns each epoch only once within all iterations of the loop in order to avoid classifying the same epoch more than once. For that we provide the `segment` method with the optional `newsamples` parameter. Using the

information about the number of new samples, `segment` can calculate which epochs must have already been returned in previous iterations and returns only the new epochs. Note, that `segment` has to take into account, that the interval of interest `SEG_IVAL` typically extends to poststimulus time. I.e., a segment is only returned when enough time has elapsed after a marker in order to extract the specified interval.

```
epo = proc.segment_dat(cnt, MARKER_DEF_TEST, SEG_IVAL, newsamples=newsamples)
if not epo:
    continue
```

If `segment` does not find any valid epochs, we abort this iteration and start the next one. Otherwise, `epo` contains at least one or more epochs. On these epochs we calculate the jumping means, create the feature vectors and apply it to the LDA classifier, exactly as in the offline example.

```
fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)

lda_out = proc.lda_apply(fv, cfy)
```

What happens with the output is highly application dependent. In the online experiment example script available in the `examples` directory that contains the complete script from above, we use `lda_out` to calculate the probabilities for each letter after each iteration of the loop. After 12 intensifications we select the most probable letter, reset all probabilities to zero and continue with the next letter. Running the script takes ca 50 minutes (equalling the duration of the recording since we used the setting `realtime=True` in the initialization of the `ReplayAmp`; for other options see the Section 3.6) and classification accuracy for correctly detected letters is identical with the accuracy of subject A in the offline classification (91%).

On the testing machine, a Laptop with a quad-core Intel i7 CPU at 2.8GHz, it takes a fairly constant time of 3.5ms to complete a full iteration of the main loop. This does not take into account the iterations that are aborted early because of empty epochs, those iterations are naturally even faster completed.

In ERP experiments, incoming data is usually processed with the same frequency as the stimuli are presented. For ERP experiments, 200ms is a common interval between two stimuli. In this case the time between two stimuli was 175ms, which is also the maximum time allowed to process the data per iteration. With 3.5ms, Wyrm processed the data faster by the order of two magnitudes which gives a lot of margin. 3.5ms is also well below the maximum time of 16.7ms needed to process the data block by block (if one block consists of 4 samples), and would be still faster than the 4.17ms needed to process the data sample by sample, given the sampling frequency of 240Hz.

3.6 Summary and Conclusion

In this chapter we presented Wyrm, an open source toolbox for BCI in Python.

After giving an introduction and motivating the need for a good offline and online toolbox, we defined the requirements for Wyrm. In the next section, we gave an overview of Wyrm's software architecture and design ideas, and described the fundamental data structure used throughout the toolbox. We also explained how we used unit testing and continuous integration as a mean of quality assurance.

In the next section, we showed how to use our toolbox with two very different data sets (ECoG and EEG) and two different paradigms (motor imagery and ERP). For both data sets we provide a brief analysis of key aspects of the data, typical for their respective paradigm. We also demonstrated how to complete the classification tasks, achieving classification accuracy comparable with the ones of the winners of the BCI Competition. This comparison is not meant as a fair competition, since the true labels of the evaluation data have been available to us. The purpose of the comparison was only to provide reproducible evidence that state-of-the-art classification can easily be obtained with our toolbox.

To cover the online aspect, we also showed how to use Wyrm to perform a simulated online experiment. For that we used again the ERP data set and performed the same classification task in an online fashion by replaying the data in real-time using a software amplifier. The online variant yields the exact same result and classification accuracy as the offline classification which demonstrates the consistency of offline and online processing in Wyrm.

Replaying the data in real-time, however, is not a very common use case in BCI as the replay takes as long as the original recording (in this case 50 minutes). We showed it here only to demonstrate the real-time capabilities of our toolbox. Replaying data in time lapse, however, can be useful to evaluate more complex methods, e.g., when some parameters of the feature extraction or the classifiers are continuously adapted. Furthermore, simulated online processing can help the debugging, when online experiments did not work as expected from previous offline tests. For that it is desirable to replay the data *faster* than real-time. For that we can turn the real-time mode off so the `ReplayAmp` will always return the next block of data with each call of the `get_data` call. The bigger we set the block size, the faster the data will be processed. Turning the real-time mode in the amplifier off and setting a blocksize of 40 samples (block length: 166.7ms), the whole experiment (including loading of the train- and test data sets and training of the classifier) takes a little less than 2 minutes to complete. Changing the block size to 400 samples (block length: 1.7s), takes less than 50 seconds to complete. All variations of the online experiment yield the exact same results and classification accuracy.

This shows that Wyrm is not only capable of performing offline and online experiments, but that its methods are written in a way to solve the necessary computations very efficiently.

While Wyrm does not provide a turnkey solution to run BCI experiments out of the box, it provides the user with all tools necessary to create online experiments and perform offline analyses. All methods of the toolbox are carefully tested for accuracy and profiled for speed and efficiency.

Future work on Wyrm should work on improving the plotting methods. While Wyrm provides the drawing primitives (e.g. scalp plots, signed r^2-values, time series, etc.), some composite plots, and utility methods for plotting, publication-ready plots still need some manual tweaking.

It is also important to realize, that the toolbox will never be finished in a sense that it contains all methods needed for BCI. It currently provides the user with the methods needed to solve common BCI tasks, but new methods are constantly developed in the field of BCI. Wyrm provides an excellent environment to try out (and implement) new methods as needed. Through the functional style of the methods provided, new methods can easily be added without causing side effects to the existing set of methods.

Compared to the existing toolboxes in the field of BCI (Section 1.6), Wyrm is still very young and other toolboxes may provide a larger set of methods or more sophisticated plotting methods. Some of the other toolboxes are for a special purpose, like SCoT for source connectivity, or BioSig and MNE-Python for analysis of biosignals. In those cases, Wyrm, being a general purpose BCI toolbox, offers a greater scope but at the same time lacks the special features provided by those toolboxes. Toolboxes like FieldTrip, BioSig, MNE-Python and SCoT are only for offline analysis of data, while Wyrm is able to perform offline analyses and online experiments. Regarding the scope of application and features, Wyrm is comparable to the BBCI toolbox and BCILAB. Both toolboxes provide a bigger set of toolbox methods than Wyrm but are otherwise comparable. However, both toolboxes are also written in Matlab and thus depend on commercial software, whereas Wyrm only depends on free software. All toolboxes provide extensive documentation and all, except BioSig and the BBCI toolbox, use unit testing.

Wyrm is free software, licensed under the terms of the MIT license and its source code is freely available at http://github.com/venthur/wyrm.

Chapter 4

Feedback- and Stimulus Presentation

4.1 Introduction

In this chapter we will present Pyff, a framework for developing- and running feedback- and stimulus applications.

Feedback- and stimulus applications are the third part of a BCI system, and the domain of hard- and software the subject interacts with during an BCI experiment. In the case of stimulus presentation the interaction is passive in a sense that the subject does usually not control anything with his thoughts. In the case of feedback, this is the part where the subject's intent that has been "translated" in the signal processing into an actionable signal, is implemented (or executed). Almost all BCI experiments contain a feedback- or stimulus application and those applications vary largely and range from spellers, games, prosthesis, and many more.

Running feedback- and stimulus applications – at the very least – must be able to communicate with the rest of the BCI system. They need to receive data from the signal processing, and send markers to the signal acquisition. Furthermore, they should give researchers conducting experiments some means for interaction with the running application.

But running those applications is only one aspect. Next to developing new methods in the signal processing part of the BCI system, developing new paradigms or improving existing ones is another big field in BCI research. An important aspect is, that researchers that develop new paradigms are not necessarily expert-programmers. In order to support those researchers conducting their experiments properly, a framework should make it as easy as possible to develop feedback- and stimulus applications by providing the right infrastructure.

With Pyff we try to cover both use cases. Pyff provides a platform for running feedback- and stimulus applications and interact with them, and it provides a framework for developing new applications. Pyff also provides a wide range of standard BCI paradigms, potentially eliminating the need to develop feedback- and stimulus applications altogether. The framework was designed to work with a wide range of BCI systems and enables them to interact with feedback- and stimulus applications during an experiment.

This chapter is divided as follows. In the next section, we will explain the requirements for Pyff. In Section 4.3 we will walk through the major components of Pyff's software

architecture and discuss their design. To understand how a BCI system can communicate with Pyff we will explain Pyff's communication protocols in Section 4.4. In Sections 4.5 and 4.6 we will cover Pyff's two major use cases: implementing a feedback and running a feedback application using Pyff. In Section 4.7 we will shortly present the available feedback- and stimulus applications already included in Pyff. In Section 4.8, we will explain the means of quality assurance we have taken, and in the last section, we will summarize, give outlook to further development, and conclude this chapter.

This chapter is based on a previous publication [Venthur et al., 2010b]. Design and implementation of the framework was done by the author of this thesis. The publication contained detailed introductions to feedback applications developed by co-authors. Those parts have been removed from this chapter.

Confusing Terminology

In this chapter the term "feedback" is used in two slightly different contexts and we have to clarify its usage first before we can continue. In the context of BCI, the application the subject interacts with when using BCI is called a feedback- or stimulus application. A stimulus application provides stimuli to the subject and the generated brain signals are not fed back to the application in contrast to a feedback application, like a speller, where the brain signals of the subject affect the output of the application. Since "feedback- and stimulus application" is a very bulky term, we will from now on use "feedback" as a synonym and include the stimulus applications in this term as well. As we will learn later, Pyff's class hierarchy also provides a base class for implementing feedback- and stimulus applications. This class is also called `Feedback` and includes feedback- and stimulus applications. You can distinguish the "feedback" application and the `Feedback` base class by the use of the capital F and the monospaced font for the base class.

4.2 Requirements

Before we start diving into Pyff, we will go through its requirements:

The first and foremost requirement is to provide a platform for running feedback- and stimulus applications. This goes beyond the simple launch of a feedback application. With Pyff it should be possible to interact with feedback applications. Feedbacks should be loaded, started and stopped dynamically, as well as it should be possible to inspect and modify the state of a feedback while it is running.

Pyff should make it possible to be easily integrated into existing BCI systems. Especially must Pyff not depend on the BCI system to be written in a particular programming language or to be running on a particular operating system. Pyff itself should run on all major operating systems.

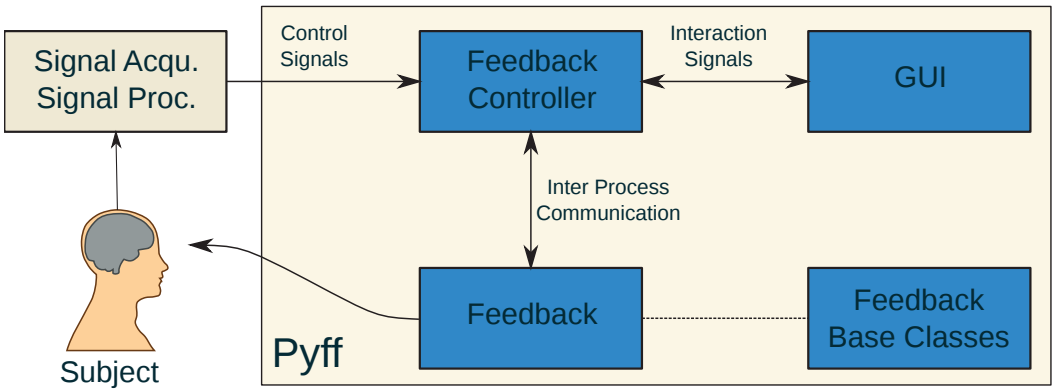

Figure 4.1: Overview of Pyff within a BCI system. The signal processing feeds control signals to the Feedback Controller. The Feedback Controller translates those signals and forwards them to the currently running feedback application using Inter Process Communication. The experimenter can interact with the feedback using the GUI which sends interaction signals to the Feedback Controller. The feedback uses modules of the `Feedback` base classes.

To assist the work of neuroscientists who are often not trained computer scientists, the programming of new feedback- and stimulus applications should be supported by an appropriate framework. Ideally, Pyff could eliminate the need to write feedback applications altogether by providing a wide range of standard paradigms.

4.3 Design

The design of Pyff focused on two aspects: the design of the software architecture, which we will discuss in this section, and the design of the protocols that enable the BCI system to communicate with Pyff, which will be discussed in Section 4.4.

The software part of Pyff consists of several components. We will now give a brief overview of Pyff's main components in order to show how these parts are connected. Later in this section, we will explain each of the main components in detail. Please note that Pyff actually consists of more components than explained here. Although those components are important for the functioning of Pyff, they are not essential for understanding how Pyff works, so we omitted them for brevity.

Figure 4.1 shows the grand overview of Pyff's main components.

Pyff as a software consists of the *Feedback Controller*, *Graphical User Interface (GUI)*, a set of *feedback- and stimulus applications*, and a set of *Feedback base classes*. The Feedback Controller is the main application that takes care of the communication with the BCI system and controls the execution of the feedback applications. The GUI is for the experimenter and allows

him to load feedbacks, start- and stop them, and inspect and manipulate their internal variables. The `Feedback` base classes are the basis for implementing own feedback applications. They provide a common interface to communicate with the Feedback Controller as well as useful methods that help to quickly develop useful feedback applications.

In order to communicate with the BCI system, Pyff also implements a protocol which allows the BCI system to communicate with Pyff over network. On a logical level, the protocol consists of *Control- and Interaction Signals*. Those signals have to be transmitted, using one of the three formats understood by Pyff: XML, JSON, and the Tobi interface C protocol.

4.3.1 The Feedback Controller

The Feedback Controller is Pyff's main application that the experimenter starts when using Pyff and it serves two purposes: first, it provides the interface for the BCI system to remote control Pyff, and second, it is responsible for controlling the execution of the stimulus- and feedback applications.

Interface for BCI Systems

Once started, the Feedback Controller acts like a server, waiting for incoming signals from the network. Signals can be encoded in various formats and Pyff currently supports: XML (Section 4.4.2), JSON (Section 4.4.3), and the TOBI interface C (TiC) protocol. The Feedback Controller converts incoming signals to message objects and, depending on the kind of signal (Section 4.4), either processes them directly or passes them to the currently running feedback. Using this protocol, a BCI system can query Pyff for a list of available feedback- and stimulus applications, start- and stop them, and is able to query and manipulate their internal variables while those applications are running.

Providing an interface to communicate with Pyff over network decouples Pyff from the rest of the BCI system and allows a great variety of BCI systems to make use of Pyff regardless of the programming language they are written. As long as they can send the appropriate signals over network, they can remote control Pyff. Please note that "over the network" implies that the BCI system and Pyff can run on the same computer as well as on different computers.

Execution of Feedback Applications

Apart from the communication aspect, the Feedback Controller also manages the execution of the feedback- and stimulus applications that run concurrently to the Feedback Controller.

When writing concurrent applications, there are usually two possible options to achieve concurrency: threads or processes. Processes are heavyweight compared to threads. Processes have their own address space and some form of inter-process communication (IPC) is needed to allow two processes to interact with each other. Threads on the other hand are

lightweight. There can be more than one thread running in the same process and threads of the same process share their address space, meaning that a modification of a variable within one thread is immediately visible to all other threads of this process as well. An exhaustive explanation of processes, threads and inter-process communication (IPC) can be found in [Tanenbaum, 2001].

Since there is no need for an IPC when using threads, threads are often more desirable than processes and a sufficient solution for many problems asking for a concurrent solution. In Python, however, things are a bit different. First, and most importantly: in Python two threads of a process never run truly concurrently but sequentially, in a time-sliced manner. The reason is Python's Global Interpreter Lock (GIL) which ensures that only one Python thread can run in the interpreter at a time [Lutz, 2006]. The consequence is that Python programs cannot make use of multiple processors by using Python's threads. In order to use real concurrency one has to use processes, effectively sidestepping Python's GIL. The second important point is that almost all graphical Python libraries like Pygame or PyQT need to run in the main (i.e. the first) thread of their Python process.

In a threaded version of Pyff, this would make things very complicated. The Feedback Controller, as a server, runs during the whole lifetime of the experiment, while feedbacks (where those graphical libraries are used) usually get loaded, unloaded, started and stopped several times during an experiment. The natural way to implement this in a threaded way would be to run the Feedback Controller in the main thread and let it spawn feedback threads as needed. But this is not possible as the feedbacks need to run in the main thread of the process, which is already occupied by the Feedback Controller. Doing it the other way round by reserving the main thread for the feedback and letting the Feedback Controller insert feedbacks into the first thread on demand, would be very complicated and error prone.

For this reason, we decided to use processes instead of threads. The feedback applications have much more resources running in their own process, while keeping the programming of the Feedback Controller's logic relatively simple and easy to maintain. Using processes, the feedbacks are also isolated from the Feedback Controller which has the advantage that a crashing or otherwise misbehaving feedback application does not directly affect the Feedback Controller as it would do if we had used threads. Since this framework also aims to be a workbench for easy feedback- and stimulus application *development*, misbehaving applications can be quite common, especially in the beginning of the application development.

The communication between two processes, however, is a lot more complicated than between two threads, since two processes do not share the same address space. In our framework we solved this issue with an inter-process communication mechanism using sockets to pass message objects back and forth between the Feedback Controller and a running feedback. We decided to use a socket based IPC since it works well on all major operating systems. The basic idea is that two processes establish a TCP connection and send messages to communicate. If a peer wants to send a message to the other one, it writes the message to his end of the connection (i.e. the socket) while the other peer can receive the message by reading from his socket. The messages itself are message objects which cannot be transferred over network directly, but have to be translated into a streamable format using a technique called serialization. The receiving end of the message has to deserialize

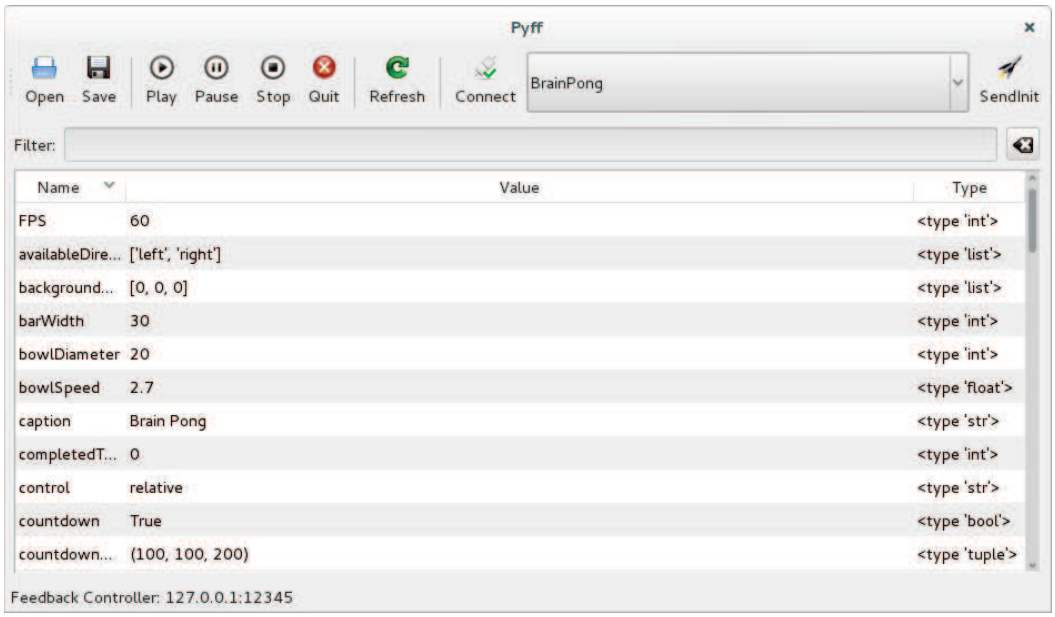

Figure 4.2: The graphical user interface of the framework. Within the GUI the experimenter can select a feedback, start, pause and stop it and he can inspect and manipulate the feedback's internal variables.

the data back into a message object in order to use it. Since both the Feedback Controller and the feedback applications are written in Python, serialization and deserialization is done using Python's `pickle` library.

4.3.2 The Graphical User Interface

Pyff provides a graphical user interface (GUI) which enables easy access to the Feedback Controller's main functions and allows for remote controlling the Feedback Controller and the running feedback application. Within the GUI, the experimenter can select and start feedbacks as well as inspect and manipulate their variables (e.g., number of trials, position and color of visual objects, etc.). The ability to inspect the feedback application's internal variables in real-time while the application is running makes the GUI an invaluable debugging tool. Being able to modify these variables on the fly also provides a great way to explore different settings in a pilot experiment.

Figure 4.2 shows a screen shot of the GUI. The drop down menu contains a list of available feedbacks the experimenter can choose from. On the left of this list are various buttons for initializing, starting and stopping the `Feedback`. The main part of the GUI is occupied by a table which presents the name, value, and type of the variables belonging to the currently running `Feedback`. The table is editable, so that the user can modify any `Feedback` variable as desired and send it back to the `Feedback` where the change is directly applied. The load and

save buttons allow for bulk loading and saving of those variables (i.e. configurations) from and to files.

The GUI communicates with the Feedback Controller via signals over network. In fact, the GUI uses the same protocol as a BCI system would use if it wants to remote control Pyff (Section 4.4). This allows the experimenter to script Pyff from his BCI system (as we will demonstrate in Chapter 5), remote control it from within the GUI, or use both. This also means that the GUI does not need to run on the same machine as the Feedback Controller. This is convenient for experiments where the subject is in a different room than the experimenter.

4.3.3 The Feedback Base Classes

The `Feedback` base class serves two purposes: it provides an interface for the Feedback Controller to talk with feedback applications and it is the basis for a class hierarchy that allows for efficient implementation of new feedback applications.

Interface for the Feedback Controller

As mentioned in Section 4.3.1, the Feedback Controller is able to load, control and unload feedbacks dynamically. Since feedback applications can be very different in complexity, functionality, and purpose, the Feedback Controller needs to rely on a small set of methods that *every* feedback provides in order to control them properly. Those methods are defined in the `Feedback` base class.

The `Feedback` class is also the root of the `Feedback` base classes' object hierarchy, and every valid feedback application must be a (direct or indirect) subclass of the `Feedback` base class. Through inheritance, all feedback applications provide the methods defined in `Feedback` and are thus controllable by the Feedback Controller.

The methods defined in the `Feedback` class are mostly for the communication with the Feedback Controller and a few convenience methods almost all feedback applications are likely to need, like methods for sending markers, loading- and saving configurations, etc.. The methods the Feedback Controller relies on for communication with the feedback are defined as follows:

```
class Feedback(object):

    def on_init(self): pass

    def on_play(self): pass

    def on_pause(self): pass

    def on_stop(self): pass

    def on_quit(self): pass
```

```
    def on_interaction_event(self, data): pass

    def on_control_event(self, data): pass
```

Without having explained yet what control- and interaction signals are (see Section 4.4), the purpose of the methods should be intuitively clear: Those methods will be called on the derived `Feedback` by the Feedback Controller. For example: when the Feedback Controller receives a control signal, it calls the `on_control_event` method of the `Feedback`, when it receives the "play" signal, it calls the `on_play` method of the `Feedback`, and so on.

By subclassing the `Feedback` base class, the derived class inherits all methods from the base class and thus becomes a valid and ready-to-use feedback for the Feedback Controller. The feedback programmer's task is to implement the methods as needed in a derived class or just leave the unneeded methods alone.

From the developer of feedback applications point of view, this is the interface he works with when implementing feedback *applications*. However, derived `Feedback` *base classes* also might need to implement complex behaviour on certain actions, like initializing the graphics environment before `on_play` is executed or taking care in which thread a specific `on_XXX` method is called. For that we added another level of indirection. Each of the `on_XXX` methods is internally called by a `_on_XXX` method (with a leading underscore). Those methods are not supposed to be overwritten by actual feedback applications, but rather by other `Feedback` base classes:

```
class Feedback(object):

    def _on_init(self):
        # do some behind-the-curtain work and call on_init afterwards
        # to be overwritten in derived feedback base classes
        self.on_init()

    def on_init(self): pass
        # to be overwritten in derived feedback applications

    # ...
```

Feedback Class Hierarchy

Providing an interface for the Feedback Controller to talk with feedbacks is one role of the `Feedback` base class, another one is to provide the basis for a class hierarchy that allows for efficient implementation of new feedback applications.

The object oriented approach can drastically simplify the development of feedbacks when applied properly. Common code of similar feedbacks can be moved out of the actual feedback implementations into a common base class and thus greatly reduce code duplication. Figure 4.3 shows an example: on the left side, two similar feedbacks share a fair amount of code. Both feedbacks work with a main loop, which is very common for graphical applications. They have a set of common main loop related- and some feedback specific

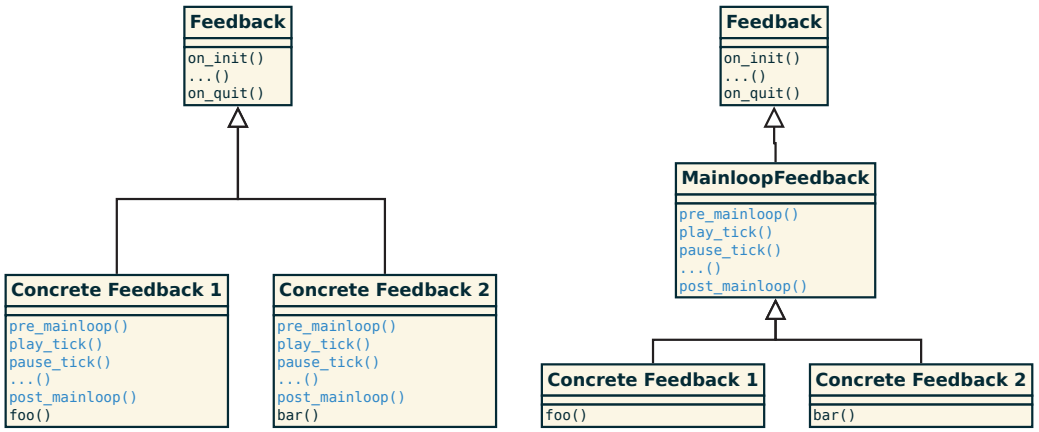

Figure 4.3: Left: Two Feedbacks sharing common code (blue). Right: The common code was moved into a single base class.

methods. Implementing a third feedback with a main loop means the programmer will likely copy the main loop logic from one of the existing feedbacks into his new feedback. This a is bad approach for several reasons, the most important one being that a bug in the main loop related logic will probably appear in all of the feedbacks using a main loop. Due to code duplication, it will then have to be fixed in every feedback using the same (copied) logic. A solution is to extract the main loop related logic into a `MainloopFeedback` base class and implement it there (Figure 4.3, right). The feedbacks using a main loop can derive from this class, inherit the logic and only need to implement the feedback specific part. The code of the new feedbacks is much shorter, less error prone and main loop related bugs can be fixed in a single location.

Pyff provides a small hierarchy of `Feedback` base classes that has been proven useful over the last years (Figure 4.4). On top of the hierarchy is the `Feedback` base class, which we already introduced a bit. The `Feedback` base class provides, next to the methods needed by the Feedback Controller, methods to send markers, load- and save variables and even provides a `logger` object, derived classes can use to output debugging information.

Next in the hierarchy are the `EventDrivenFeedback` and the `MainloopFeedback`. The idea of the later has been introduced above already, it provides methods useful when working with graphical toolkits that use a main loop (which is the vast majority). It also takes care of a particularly difficult problem when dealing with a main loop of a graphical toolkit: ensuring that the main loop of the toolkit runs in the main (i.e. first) thread of the feedback process and all other `on_XXX` methods are called from another (i.e. not-main) thread.

The `EventDrivenFeedback` base class supports a more imperative style of feedback programming like: "react on event X for 20 seconds; pause for 5 seconds; etc."

Derived from the `MainloopFeedback` class are the `PygameFeedback` and `VisionEggFeedback` classes. Pygame and VisionEgg are graphical toolkits which make use of an internal event loop (i.e. their main loop) that has to be polled regularly. Both base classes are derived from the

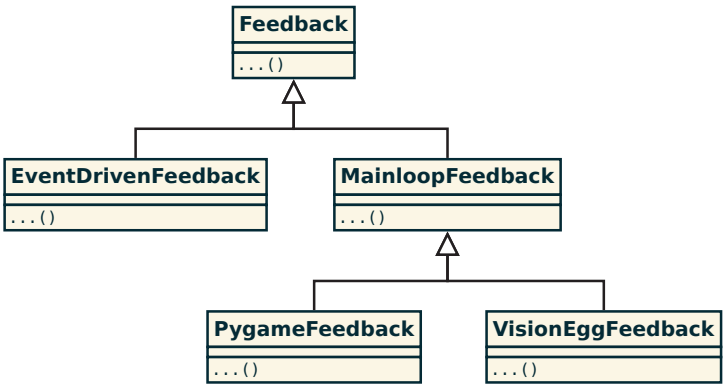

Figure 4.4: The class hierarchy of the `Feedback` base classes currently supported by Pyff. The specific methods the classes introduce have been omitted for brevity.

`MainloopFeedback` class and implement the event loop handling for their respective toolkit. Additionally, both base classes provide useful helper methods for screen initialization, positioning of the application window, drawing primitives on the screen, etc..

If a researcher needs to implement a feedback, he can use one of the base classes provided, but if none of them quite fits, he is encouraged to derive own base classes to provide the functionality needed.

4.4 Control- and Interaction Signals

In this section we will explain the protocols used in Pyff to communicate with the BCI system.

The Feedback Controller receives two different kinds of signals: *Control Signals* and *Interaction Signals*.

Control signals are signals sent from the BCI system to the feedback in order to update the feedback. Usually the control signal contains the classifier output calculated by the signal processing of the BCI system or similar data which the feedback application uses to update, for example, what is shown on the screen. Control signals can only contain data.

Interaction signals are signals that are meant to control the execution of the feedback application, for example a command to start- or stop the feedback application. Interaction signals are usually sent from the GUI or from the part of the BCI system that remote controls Pyff (see Chapter 5 for an example). Interaction signals can contain commands and data.

Control- and interaction signals are handled slightly different when received by the Feedback Controller. An interaction signal containing a command, causes this command to be executed in the Feedback Controller (e.g. the command “getfeedbacks”) or the currently running feedback (e.g. “play”). Data that is sent via an interaction signal is applied to the currently running feedback first and then the feedback is informed about the change via

`on_interaction_event()`. For example: an incoming interaction signal with the command "play" and the data `foo=42`, triggers that the Feedback Controller sets the feedbacks's variable `foo` to the value of `42` or, if it did not exist, creates `foo` and assigns the value `42`. The Feedback Controller then informs the feedback that this value has been changed by calling `on_interaction_event({"foo": 42})` followed by an `on_play()`.

On the other hand, if the Feedback Controller receives a control signal with the data `bar=23`, the Feedback Controller changes the feedbacks internal dictionary `_data` by adding or modifying the value for the key `"bar"` to `23` and then informs the feedback by calling the `on_control_event({"bar": 23})` method. What happens with this information depends on the implementation of the feedback's `on_control_event` method or whether the feedback regularly checks the contents of his `_data` dictionary.

Please note how differently data is handled in control- and interaction signals. While data in interaction signals directly overwrite existing instance variables in the feedback, data in control signals have only limited access to one special variable in the feedback. To better understand the difference between control- and interaction signals it is helpful to think of interaction signals as a mean to interact with a feedback and modify its state as an object, whereas control signals are only intended to send BCI related data (e.g. classifier output) to the feedback.

Table 4.1 shows the complete list of available commands in interaction signals and Figure 4.5 shows a sequence diagram illustrating the interaction between the BCI system, the GUI, the Feedback Controller and the feedback. All signals coming from the GUI are interaction signals, the signals coming from the BCI system are control signals. Once the GUI is started, it tries to automatically connect to a running Feedback Controller on the machine where the GUI is running. If that fails (e.g., the Feedback Controller is running on a different machine on the network) the experimenter can connect it manually by providing the host name or IP address of the target machine. Upon a successful connection with the Feedback Controller, the Feedback Controller replies with a list of available feedbacks which is then shown in a drop down menu in the GUI. The experimenter can now select a feedback and click the Init Button which sends the interaction signal with the command "init" to the Feedback Controller, telling it to load the desired feedback. The Feedback Controller loads the feedback and requests the feedback's object variables which it sends back to the GUI. The GUI then shows them in a table where the experimenter can inspect and manipulate them. Within the GUI, the experimenter can also Start, Pause, Stop and Quit the feedback.

4.4.1 Serializing Control- and Interaction Signals

We already explained that the Feedback Controller accepts control- and interaction signals over a network socket, which decouples Pyff from the BCI system in a way that both can be different programs, written in different programming languages. But so far, those signals have been abstract in a sense that we didn't specify a specific format for them. In this section we will change this and introduce two protocols we developed to enable an arbitrary BCI system to "speak" with Pyff by serializing control- and interaction signals in a programming language agnostic way.

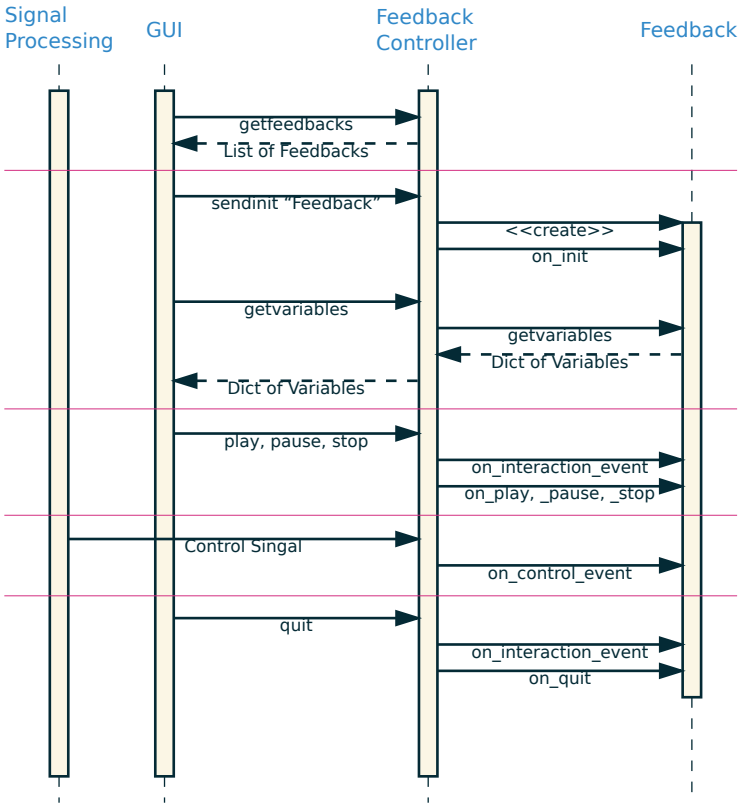

Figure 4.5: The different kinds of signals and the events they cause in the feedback. (1) The GUI connects to the Feedback Controller; (2) The user selects and initializes feedback; (3) The user starts, pauses, or stops the feedback; (4) The BCI system provides classifier output; (5): The user quits the feedback.

Command	Meaning
getfeedbacks	Return a list of available feedbacks
getvariables	Return a dictionary of the feedback's variables
sendinit	Load a feedback
play	Start the feedback
pause	Pause the feedback
stop	Stop the feedback
quit	Unload the feedback

Table 4.1: List of available interaction signal commands.

Back when the development of Pyff started, the canonical answer to that problem was to translate messages using Extensible Markup Language (XML) [Bray et al., 1998]. XML is a very flexible text format and well suited to represent documents and data structures. XML, however, is also very verbose and without a supporting library, parsing of XML can be daunting. With the raise of JavaScript in the 2010s, an alternative standard format, the JavaScript Object Notation (JSON) [Crockford, 2006], emerged. Although originally derived from JavaScript, JSON is now a language-independent data format that provides a lightweight alternative to XML in many use cases.

Pyff currently supports three protocols: XML, JSON and the TOBI interface C protocol (which we will not explain here). XML was the first protocol supported by Pyff in a time when JSON was still exotic. Years later, when JSON became a well accepted standard and JSON support even moved into Python's standard library, JSON was added as a protocol as well. Pyff supports both protocols for now, but the goal is to phase out XML in favour to JSON.

4.4.2 Encoding Control- and Interaction Signals in XML

The root element of our XML scheme is the `bci-signal`-node which contains an attribute `version`, defining the bci-signal version, and one child node which can be either of the type `interaction-` or `control-signal`.

```
<?xml version="1.0" ?>
<bci-signal version="1.0">
  <interaction-signal>
    <!-- content of the signal -->
  </interaction-signal>
</bci-signal>
```

The content of the `control-` and `interaction-signal`-nodes can be nodes containing data and additionally for `interaction-signals`, `command`-nodes containing commands and their parameters.

Data

Data is encoded in form of variables. Very much like in most programming languages, a variable is defined by the triple: `(datatype, name, value)` and the representation in our XML scheme is straightforward:

```
<datatype name="varname" value="varvalue" />
```

For example: an Integer with the variable name "foo" and the value 42 can be represented as:

```
<integer name="foo" value="42" />
```

Some data types like lists or dictionaries are containers for other variables. For example, a list which contains three integers:

```
>>> mylist = [1, 2, 3]
```

The representation of this list in our XML scheme is the following (please note how the integers in the list have no names):

```
<list name="mylist">
  <integer value="1"/>
  <integer value="2"/>
  <integer value="3"/>
</list>
```

Containers can also contain other containers. The following example shows a list containing two integers and a list which also contains two integers:

```
>>> mylist2 = [1, 2, [3, 4]]
```

The representation in XML:

```
<list name="mylist2">
  <integer value="1"/>
  <integer value="2"/>
  <list>
    <integer value="3"/>
    <integer value="4"/>
  </list>
</list>
```

Our XML schema puts no restrictions upon the depth of the nested variables and allows for arbitrary deep nesting of containers.

Special care has to be taken using dictionaries. Dictionaries are not simply collections of variables, but mappings from keys to values. The keys are strings and the values arbitrary variables (including dictionaries). Those mappings are expressed through tuples `(string, variable)` and consequently, a dictionary is represented as a collection of tuples in our XML scheme. The following example shows a dictionary containing three integers:

```
>>> mydict = {'foo': 1, 'bar': 2, 'baz': 3}
```

In XML:

```
<dict name="mydict">
  <tuple>
    <s value="foo"/>
    <i value="1"/>
  </tuple>
  <tuple>
    <s value="bar"/>
```

Type	Type in XML	Example Values	Nestable
Boolean	boolean, bool, b	"True", "true", "1"	No
Integer	integer, int, i	"1"	No
Float	float, f	"1.0"	No
Long	long, l	"1"	No
Complex	complex, cmplx, c	"(1+0j)", "(1+0i)"	No
String	string, str, s	"foo"	No
List	list		Yes
Tuple	tuple		Yes
Set	set		Yes
Frozenset	frozenset		Yes
Dictionary	dict		Yes
None	none		No

Table 4.2: Data types supported by our XML scheme.

```
    <i value="2"/>
  </tuple>
  <tuple>
    <s value="baz"/>
    <i value="3"/>
  </tuple>
</dict>
```

Our XML scheme supports all variable types supported by Python. Sometimes there is more than one way to express the data type of a variable. For example, a boolean variable can be expressed in our XML scheme via `<boolean ... />`, `<bool ... />`, and `<b ... />`. All alternatives are equivalent and exist merely for convenience. Table 4.2 shows a complete listing of all supported data types in our XML scheme.

Commands

Commands are only allowed in interaction signals and have the following form:

```
<command value="commandname"/>
```

where `"commandname"` is a member of the set of supported commands. Supported commands are: `"getfeedbacks"`, `"play"`, `"pause"`, `"stop"`, `"quit"`, `"sendinit"`, `"getvariables"` (the meaning of the commands is explained in Section 4.4). Only one command is allowed per interaction signal and an interaction signal can contain a command and several variables.

Commands can also have arguments. Arguments are represented as dictionaries, with the keys being the argument names and the values the values of the arguments. Commands with arguments have the following form:

```
<command value="commandname">
    <dict>
        <tuple>
            <string value="name"/>
            <integer value="value"/>
        </tuple>
    </dict>
</command>
```

A complete example of a valid interaction signal encoded in XML looks like the following:

```
<?xml version="1.0" ?>
<bci-signal version="1.0">
  <interaction-signal>

    <!-- play command (has no parameters) -->
    <command value="play"/>

    <!-- some variables -->
    <string name="somestring" value="foo"/>
    <float name="somefloat" value="0.69"/>
    <list name="somelist">
      <integer value="1"/>
      <integer value="2"/>
      <integer value="3"/>
    </list>

  </interaction-signal>
</bci-signal>
```

This small XML scheme provides a well defined and clean protocol for other systems to communicate with Pyff. Every BCI system that is able to produce control- and interaction-signals encoded in this XML format and sent them over network, is able to fully remote control Pyff and its feedbacks.

4.4.3 Encoding Control- and Interaction Signals in JSON

A signal encoded in JSON is a JSON object with 3 attributes: `"type"`, `"commands"`, and `"data"`. The value in `"type"` determines whether the signal is a control- or interaction signal, `"commands"` is a list with two elements, the first being a string with the command name, and the second a dictionary with the arguments for that command. The `"data"` attribute is for the data and contains a dictionary with arbitrarily nested variables.

For comparison with the XML scheme, we have reproduced the last example in the XML section in the equivalent JSON encoding:

```
{
    "type": "control-signal",
```

```
    "commands": ["play", {}],
    "data": {
        "somestring": "foo",
        "somefloat": 0.69
        "somelist": [1, 2, 3],
    }
}
```

The JSON encoding is almost as expressive as the XML encoding with the difference that only a subset of the Python data types is supported for the variables, namely: floats, strings, Booleans, lists, dictionaries and `None`. However, since those or equivalent data types are supported by almost all programming languages, libraries exist for many programming languages that allow for *direct* translation from the variables into JSON and back, which makes serialization very easy.

For example to produce the above JSON encoded interaction signal in Python, one only has to create a Python dictionary with the 3 keys `"type"`, `"commands"`, and `"data"` with valid values for them, and serialize them using Python's `json` module from the standard library:

```
>>> import json
>>>
>>> signal = {
...     'type': 'interaction-signal',
...     'commands' : ['play', dict()],
...     'data' : {
...         'somestring': 'foo',
...         'somefloat': 0.69,
...         'somelist': [1, 2, 3],
...     }
... }
>>>
>>> json.dumps(signal)
{"data": {"somelist": [1, 2, 3], "somestring": "foo", "somefloat": 0.69},
"commands": ["play", {}], "type": "interaction-signal"}
```

Similar JSON libraries exist for many programming languages, which is a huge improvement over the XML protocol for this use case. There are also many libraries for reading and writing XML files, but since XML is a very generic language, purposely designed to support any kind of document or data, one also often has to write his own library for reading and writing a specific XML scheme.

4.5 Implementing a Simple Pong Game

In this section we will demonstrate one of the two important use cases for Pyff, and show how to implement a trivial Pong game using Pyff.

The game we are going to implement resembles a game of squash: a ball moves in a straight line through the game window and bounces off the walls. The player has a bar at the

bottom of the screen that can be moved left and right to prevent the ball from touching the bottom end of the screen. In order to keep this demonstration simple, we did not implement points for hitting the ball or penalties for letting the ball touch the bottom of the window. The ball just keeps bouncing off the walls or the bar.

The bar will be controlled using motor imagery and we assume that the BCI system will provide us continuously with a classification output value in the range of `[-1..1]` via control signals.

We will use Pygame as the graphical toolkit as it is well suited for simple game like applications, and supported by Pyff via the `PygameFeedback` base class.

The first step is to derive our feedback class from the `PygameFeedback` base class. The `PygameFeedback` base class takes care of many things that have to be dealt with when implementing a Pygame application, like initializing the graphics, setting the screen size- and position, etc.. Game like applications almost always have some sort of main loop, that iterates (i.e. "ticks") several times per second. In each tick the positions of the elements on the screen are updated and re-drawn. Since the `PygameFeedback` class itself is derived from the `MainloopFeedback` class, it also implements a main loop that is connected to Pygame's internal main loop and provides us with a `play_tick` method we can overwrite to implement the game logic.

The first method we will implement is `init`. This method gets called automatically when the feedback is initialized. We will call the `init` method of the parent `PygameFeedback` class first, in order to let it set its default values for screen size, position, etc., and initialize a few variables ourselves.

```
import os
import pygame
from FeedbackBase.PygameFeedback import PygameFeedback

class TrivialPong(PygameFeedback):

    def init(self):
        PygameFeedback.init(self)
        self.caption = "Trivial Pong"
        # set the initial value for the classifier output
        self.val = 0.0
        # set the initial speeds for ball and bar
        self.barspeed = [3, 0]
        self.speed = [2, 2]
```

Now we will implement the `init_graphics` method, inherited from the `PygameFeedback` class. This method gets automatically called when the feedback is started. When this method is called, the Pygame graphics-system has been initialized already, but the main loop has not been started yet. In this method we simply load the graphics for the ball and the bar and create bar- and ball objects that we will later move around the screen:

```
    def init_graphics(self):
        # load graphics
        path = os.path.dirname(globals()["__file__"])
```

```
        self.ball = pygame.image.load(os.path.join(path, "ball.png"))
        self.ballrect = self.ball.get_rect()
        self.bar = pygame.image.load(os.path.join(path, "bar.png"))
        self.barrect = self.bar.get_rect()
```

The next step is to implement the `on_control_event` method which is inherited from the `Feedback` base class and called whenever the Feedback Controller receives new data from the BCI system. By overwriting this method and reading the `data` that was sent, we can update our variable `val` which holds the current value of the classification result:

```
    def on_control_event(self, data):
        self.val = data["clout"]
```

The last method of our feedback is the `play_tick` method. It gets continuously called several times per second after the feedback has been started until the feedback stops. The exact number of times per second this method gets called depends on the feedbacks `FPS` variable, which is inherited from the `PygameFeedback` class and has a default value of `30`.

Our `play_tick` method implements the whole logic of the game: we update the position of the ball and the bar depending on the current value of the `var` variable, take care of the collision detection (i.e. the "bouncing") and finally update the screen with the new positions of ball and bar:

```
    def play_tick(self):
        width, height = self.screenSize
        w_half = width / 2.
        # move bar and ball
        pos = w_half + w_half * self.val
        self.barrect.center = pos, height - 20
        self.ballrect = self.ballrect.move(self.speed)
        # collision detection walls
        if self.ballrect.left < 0 or self.ballrect.right > width:
            self.speed[0] = -self.speed[0]
        if self.ballrect.top < 0 or self.ballrect.bottom > height:
            self.speed[1] = -self.speed[1]
        if self.barrect.left < 0 or self.barrect.right > width:
            self.barspeed[0] = -self.barspeed[0]
        if self.barrect.top < 0 or self.barrect.bottom > height:
            self.barspeed[1] = -self.barspeed[1]
        # collision detection for bar vs ball
        if self.barrect.colliderect(self.ballrect):
            self.speed[0] = -self.speed[0]
            self.speed[1] = -self.speed[1]
        # update the screen
        self.screen.fill(self.backgroundColor)
        self.screen.blit(self.ball, self.ballrect)
        self.screen.blit(self.bar, self.barrect)
        pygame.display.flip()
```

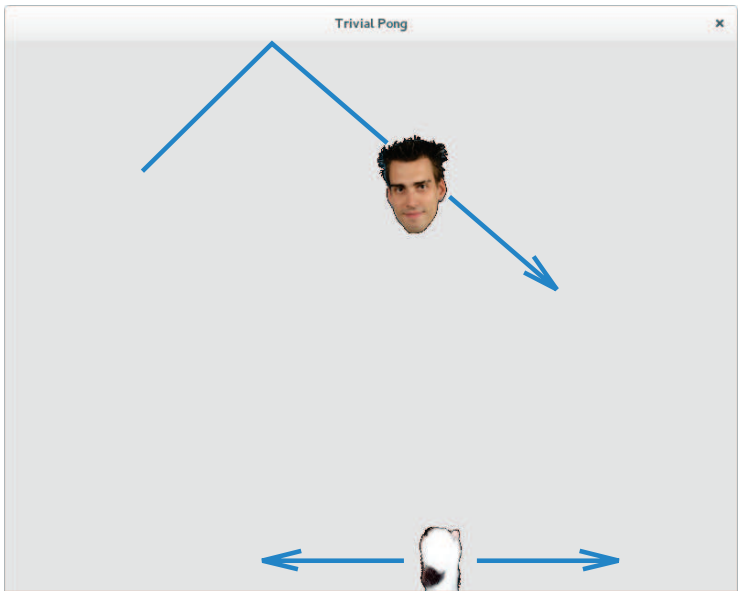

Figure 4.6: Screenshot of the Pong feedback. The ball and bar look suspiciously like the author of this thesis and paw of his cat. The blue arrows show the directions of the ball and the bar.

The resulting feedback looks like the one in Figure 4.6 and is included in the Pyff distribution. If we would start Pyff, load the feedback and connect a properly setup BCI system with a subject to Pyff, that continuously sends classification output in the range of `[-1..1]`, we could use the feedback to play a game of "Trivial Pong" using motor imagery.

Although a few things are still missing to make the game more challenging, we hope this example could demonstrate how easy it can be to implement a feedback using Pyff and its base classes.

4.6 Running a Feedback Application

In this section we will demonstrate Pyff's second major use case and go step by step through all the steps the experimenter has to take in order to run a feedback application with Pyff.

Starting Pyff

First we start the Feedback Controller by executing the file `FeedbackController.py` from Pyff's `src` directory. The Feedback Controller has many command line options for configuration and a complete list with explanations can be found when calling the Feedback Controller with the `--help` argument:

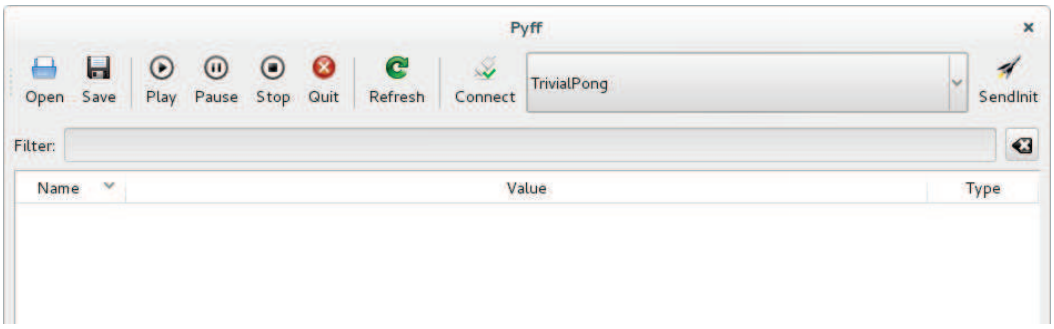

Figure 4.7: Screenshot of the GUI after it has been started.

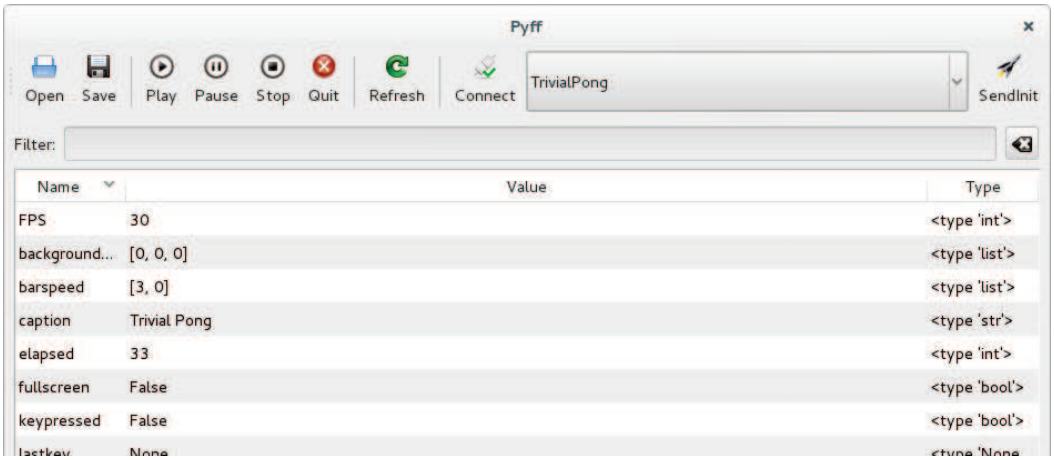

Figure 4.8: Screenshot of the GUI after a feedback has been initialized.

```
python FeedbackController.py --help
```

Starting a Feedback

Once the Feedback Controller has started, the GUI will automatically start as well, waiting for commands from the experimenter. The next step is to select the desired feedback- or stimulus application from the dropdown menu on the top right (Figure 4.7) and clicking the "SendInit" button next to it to initialize the feedback.

Inspecting and Modifying the Feedback's Internal State

After the feedback has been initialized, the feedback's internal variables appear in the table of the GUI (Figure 4.8). The experimenter can select variables in the table and change the

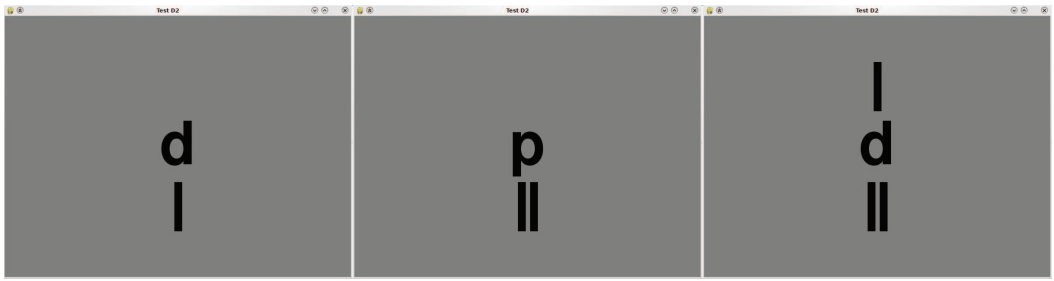

Figure 4.9: Screenshots of the stimuli presented by the feedback. A stimulus is presented until the subject presses one of two available buttons to decide if he sees a d2 or not. After the key is pressed the next stimulus is presented immediately.

values as needed. Any changes are applied immediately to the feedback.

Alternatively, the experimenter can load a complete configuration from a configuration file by using the "Open" button. A configuration file contains the values for several variables of the feedback and when the configuration file has been loaded, the variables are applied to the feedback all at once. By using the "Save" button, the experimenter can save the current variables of the feedback into a configuration file for later use.

Starting, Pausing, Stopping, and Quitting

When the experimenter is ready, he can start the feedback by clicking the "Play" button. Now that the feedback is running, the experimenter can still evaluate and modify the feedback's internal variables via the GUI.

While the feedback is running, the experimenter can pause and stop the feedback, by clicking the appropriate buttons. When the experimenter wants to quit Pyff, he can simply close the GUI. This will automatically quit any running feedbacks and the Feedback Controller as well.

In Chapter 5 we will demonstrate how we can achieve the same work flow without using the GUI, but by sending the interaction signals from the signal processing.

4.7 Included Feedbacks

Pyff provides a variety of standard BCI feedback- and stimulus applications that have been developed over the years in our lab. Most of them have been used for real experiments that resulted in publications. We provide them in the hope that other groups might find them useful. Groups that don't focus on developing own paradigms but rather work with existing paradigms can re-use them without the need to implement the applications themselves. Most applications can be customized via interaction signals and should work in without many modifications required.

The following list briefly introduces the available applications. Most of those applications have *not* been developed by the author of this thesis.

Hex-o-Speller The Hex-o-Spell paradigm, allows for writing sentences and is available in three different variations: motor imagery-, event- related potential-, and steady state visual evoked potentials-based.

Matrix Speller The matrix speller allows for writing words and sentences using ERP based BCI [Farwell and Donchin, 1988].

Gaze Independent Speller Pyff includes two ERP-based, gaze independent spellers: the Cake- and the Center Speller [Treder et al., 2011].

Motor Imagery Pyff includes a few games using motor imagery: Brain Pong, Goal Keeper and Cursor Arrow.

Oddball Paradigm A versatile implementation of the oddball paradigm using visual, auditory, or tactile stimuli.

Test D2 A computerized version of the d2 test [Brickenkamp, 1972], a Psychological pen and paper test to assess concentration- and performance ability of a subject (Figure 4.9).

N-Back A parametric paradigm to induce workload. Symbols are presented in a chronological sequence and upon each presentation participants are required to match the current symbol with the n-th preceding symbol [Gevins and Smith, 2000].

Boring Clock A computerized version of the Mackworth clock test [Mackworth, 1948], a task to investigate long-term vigilance and sustained attention. Participants monitor a virtual clock whose pointer makes rare jumps of two steps and they press a button upon detecting such an event.

4.8 Tests, Documentation, and Examples

To ensure the correct functioning of Pyff and its internal modules, various unit tests cover the critical aspects like serialization of the control- and interaction signals or the correct functioning of the inter process communication.

Part of the framework is also a complete documentation of the system and its interfaces. All modules, classes and methods are also extensively documented in form of Python docstrings. Those docstrings are used by Python's internal help system and Integrated Development Environments (IDEs). Furthermore, a complete html or pdf documentation can be generated with a single command.

In order to teach users how to implement own feedback- and stimulus applications, Pyff also provides a tutorial explaining every aspect of feedback development step-by-step.

4.9 Summary and Conclusion

In this chapter we presented Pyff, an open source framework for feedback- and stimulus presentation, and rapid feedback development.

The chapter started by motivating the need for such a framework and discussing its requirements. We then went on to the design aspect, were we investigated the main components of Pyff's software architecture. Later we examined the protocols used to communicate with the BCI system and gave an overview of the feedback- and stimulus applications already included with Pyff. We also demonstrated the two main use cases of Pyff: developing a feedback application and running it.

Pyff provides a platform for writing high quality stimulus- and feedback applications with minimal effort, even for non-computer scientists. Pyff's concept of `Feedback` base classes allows for rapid feedback and stimulus application development, e.g., oddball paradigms, ERP-based typewriters, Pygame-based applications, etc.. Moreover, Pyff already includes a variety of feedback- and stimulus applications which are ready to be used instantly, or with minimal modifications. This list is ever growing as we constantly develop new ones and other groups will hopefully join the effort.

Pyff can be used as a standalone application, for instance providing stimulus presentation in psychophysics experiments, or within a closed loop such as in BCI- or biofeedback.

By providing an interface utilizing well known standard protocols and formats, Pyff is adaptable to most existing BCI systems. Furthermore, Pyff also implements a few foreign standards to talk with TOBI interface C compatible BCI systems or BCI2000. This creates the unique opportunity of exchanging feedback applications and stimuli between different groups, even if individual systems are used for signal acquisition, processing and classification. Having such a general, open source framework will help foster a fruitful exchange of experimental paradigms between research groups. In particular, it will decrease the need for reprogramming standard paradigms, ease the reproducibility of published results, and naturally entail some standardization of stimulus presentation.

Currently the set of `Feedback` base classes focuses mainly on the interaction with graphical toolkits. While those base classes do a good job in taking care of threading issues, graphics initialization and generally, reducing boiler plate code that is not related to the functionality of the actual feedbacks, there is still room for improvement. A lot of feedbacks still share similar functionality that is duplicated among the different implementations. For example, matrix- and similar spellers need to highlight specific elements in a pseudo random order that follows a few rules like: in each iteration every element has to be intensified exactly once, the first element of a sequence must not be the same as the last one from the previous one, avoid two neighboring elements to be intensified successively, etc.. Those common behavioral design patterns should be provided by libraries in Pyff and used by implementations of feedbacks by using, for example, the strategy pattern.

Usually the signal processing part of a BCI system is the "driver" of the whole software setup, and therefore should be able to communicate with the signal acquisition and the feedback- and stimulus presentation. While we provide the specification of the protocol to enable other groups to implement a library that enables their BCI system to talk with Pyff,

and provide such a library for Python, we could provide those libraries for other common programming languages used in BCI ourselves, to ease the migration to Pyff.

Since Pyff was developed, it replaced the Matlab based solution and became the standard framework for developing feedback- and stimulus applications in our lab. Over the last years, Pyff has been successfully used in numerous experiments, that lead to publications in- and outside our group [Treder et al., 2011; Höhne et al., 2011; Acqualagna and Blankertz, 2011; Quek et al., 2011; Bahramisharif, 2012; Quek, 2013; An et al., 2014]. Pyff has also been used in the TOBI project, a large European project which aimed for developing practical technology for BCI to improve quality of life of disabled people and the effectiveness of rehabilitation [Schreuder et al., 2012, 2013].

Pyff is free software, licensed under the terms of the GNU General Public License (GPL) and its source code is freely available at http://github.com/venthur/pyff.

Chapter 5

Putting it all together

5.1 Introduction

In the last three chapters we introduced the three parts of our BCI system and showed, among other things, how to use them in their typical use cases of their domain. In this chapter we will go one step further and combine all three components to form a complete BCI system. Furthermore, we will demonstrate how Mushu, Wyrm, and Pyff work together by performing a very typical BCI task: an online BCI experiment using an ERP based speller paradigm.

We will start this chapter by showing how the Cake Speller paradigm used in this experiment works and explaining the experimental setup. In the main part of this chapter we will walk the reader through the script that runs the online experiment and explain the main phases of the experiment. In the last sections, we will show the results, summarize, and conclude the chapter.

The Cake Speller feedback used in this experiment has been implemented by Laurens R. Krol.

5.2 The Cake Speller

In this experiment, our subject's task is to spell a specific set of words using an ERP based speller, similar to the matrix speller from Section 3.4.2. Instead of a matrix speller, we will use the gaze independent Cake Speller paradigm [Treder et al., 2011].

In the Cake Speller, the letters of the alphabet are grouped into six groups (i.e. cake segments), and choosing a letter is a two-step process (Figure 5.1). In the first step, the subject chooses the group containing the desired letter, in the second step, the subject chooses the desired letter from the six elements of the previously selected group. Each possible group in the second step has a special "back" character (caret) which the user can choose to get back to the first level, in case the group he is currently in, is not the one he wanted.

The actual selection of the cake segments happens by random intensifications of the segments. During a sequence, all segments are intensified consecutively in random order. The

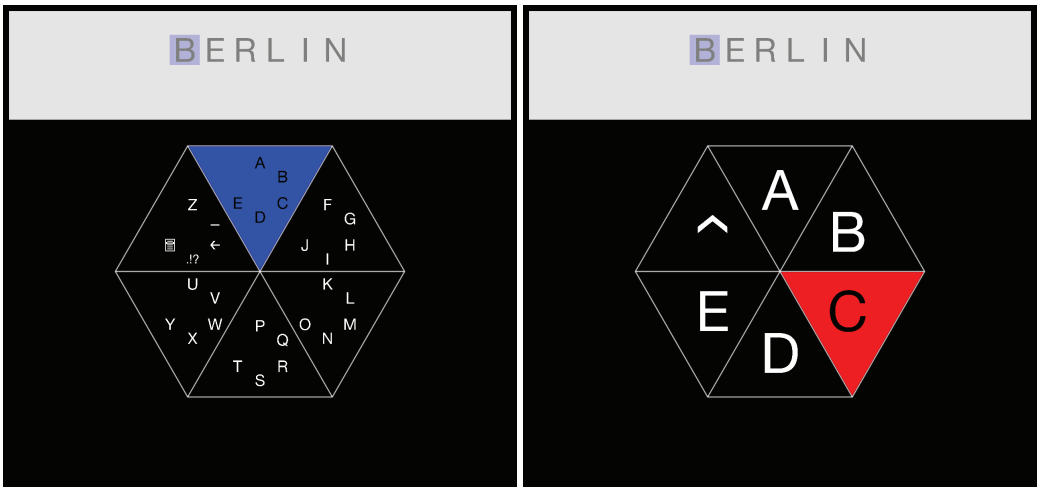

Figure 5.1: Screenshots of the Cake Speller used in the experiment. The image on the left shows the first level, where the subject selects a group of letters. After the selection has been made, the subjects gets to the second level (image on the right), where he can select the letter he wants to spell. The caret character between the A and the E is the "back" character. The word in the gray area is the word the subject is supposed to spell, with the current letter highlighted in blue.

segment the subject is concentrating on (i.e. the target) will have a different ERP response in the subject than the other ones (i.e. the non-targets). Those differences will be used in the classification to retrieve the segment the subject wanted to select. In this experiment we used 8 intensifications per segment and sequence.

For this experiment we used the Cake Speller in copy spelling mode. In contrast to the free spelling mode, were the subject is free to spell whatever he wants, in copy spelling mode the user has to spell predefined words that are visible in the top area of the feedback with the current letter highlighted. The copy spelling mode is useful in settings like this, where we want to quantify the accuracy of our classification method.

5.3 Experimental Setup

In this online experiment we used two computers which were connected via the university network. One computer was running Microsoft Windows, the other laptop was running Debian GNU/Linux (Figure 5.2).

The amplifier, a BrainAmp DC by Brain Products GmbH, was connected to the Windows PC, where the data was acquired using the BrainVision Recorder by Brain Products. Further, a Lab Streaming Layer (LSL) server [Kothe, 2013] was started on that machine that uses Remote Data Access (RDA) to get the data from the BrainVision Recorder and provide it

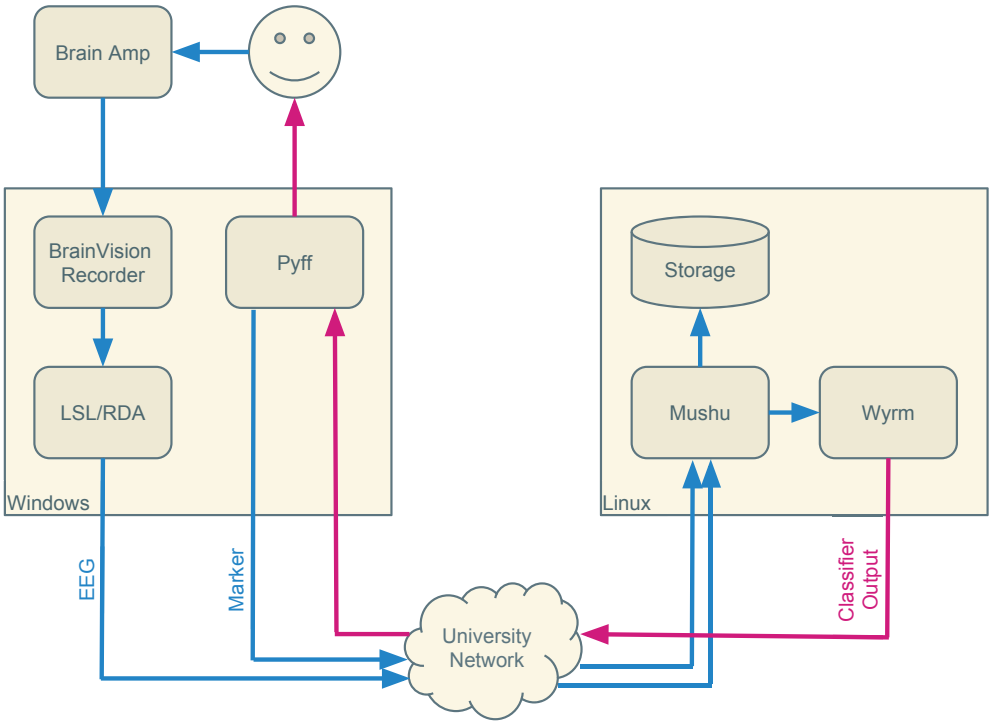

Figure 5.2: Setup of the experiment. The blue arrows show the flow of the EEG data and markers from the Windows computer over the university network to the Linux laptop; the red arrows show the flow of the classifier output from the Linux laptop over the university network to the Windows computer.

as an LSL stream to the network. The reason why we used the BrainVision Recorder to collect the data and forward it via LSL to Mushu was that the project in which context the experiment was conducted, required the use of the BrainAmp amplifier. Unfortunately we do not have a driver for that amplifier in Mushu so we had to take the detour over LSL to use the required amplifier *and* Mushu.

To showcase Mushu's network marker feature and prove that it works reliable enough even for online experiments when used over a network, Pyff was also running on the Windows machine to present the Cake Speller feedback to the subject and send markers via User Datagram Protocol (UDP) over the university network to Mushu.

On the Linux laptop we used Mushu's LSL amplifier to collect the EEG data from the LSL stream and Pyff's UDP markers over the university network. Mushu also saved the EEG data and markers to disk. Also running on the Linux laptop, was Wyrm to perform the signal processing and classification, as well as to remote control Pyff.

The recording was performed using an actiCAP by Brain Products GmbH, with a standard 32 channel setup and a sampling frequency of 1kHz.

5.4 Running the Experiment

The online BCI experiment consists of two parts. In the first part, training data is recorded and used to train the classifier, in the second part the online classification takes place. We will now walk through the whole experiment script step by step and see how Mushu, Wyrm, and Pyff work together.

The script starts with the import statements, where we import Python's `time` module, `libmushu` for access to the amplifier, Wyrm's `processing` module for signal processing and classification, Wyrm's `io` module for communication with Pyff, as well as Wyrm's `RingBuffer` and `BlockBuffer` implementations.

```
import time

import libmushu
import wyrm.processing as proc
from wyrm import io
from wyrm.types import RingBuffer, BlockBuffer
```

The original script contains the definitions of some global constants after the imports, which we omitted here for brevity. The constants are written in capital letters, and we will explain their content when they appear the first time in the script.

5.4.1 Recording Training Data

In the training step, we need to acquire training data for the classifier so it can learn the different brain patterns for targets and non-targets. For that, we use the Cake Speller and

show the subject a few words he has to spell. The Cake Speller then runs for each letter of the words through all sequences of intensifications while the subject is concentrating on the correct segment of the cake respectively. At each intensification of a segment, Pyff sends a marker to Mushu, that contains the segment number and the information whether the current segment is a target or a non-target. This will allow us later to find the parts of the EEG signal that contained target- or non-target patterns.

The script for the training is straightforward: we use Wyrm's `PyffComm` object to remote control Pyff, which is running on a different computer (i.e. `PYFF_HOST`). Via `PyffComm` we initialize the `'SpellCake'` feedback and set a few feedback specific variables. We then initialize and configure Mushus LSL-amplifier:

```
pyff = io.PyffComm(PYFF_HOST)
pyff.send_init('SpellCake')
time.sleep(10)
pyff.set_variables({'useOutputModule' : True, 'autoSelect' : True,
                    'allowBackspace' : False, 'pause' : False})
time.sleep(5)

amp = libmushu.get_amp('lslamp')
amp.configure()
```

After everything has been set up, we start the amplifier and the feedback. The training begins and we collect data from the amplifier in a loop until we received the `MARKER_END` marker, which Pyff sends when the training has finished. The amplifier automatically saves the collected EEG data and markers into a file `filename`, that was given as parameter on `amp.start()`. When the training sequences are finished, and the recording loop has been left, the amplifier is stopped and the Cake Speller feedback in Pyff is closed.

```
amp.start(filename)
pyff.play()

while True:
    _, markers = amp.get_data()
    if MARKER_END in [m for _, m in markers]:
        break

amp.stop()
pyff.quit()
```

5.4.2 Training the Classifier

After the training data has been recorded, we train the classifier. For that we load the previously recorded data using Wyrm's `load_mushu_data` method. We then calculate the filter coefficients for a 45Hz low- and .4Hz high pass filter of order 5 respectively, apply the filters to the data and subsample the filtered data from 1kHz to 100Hz:

```
cnt = io.load_mushu_data(filename)

fs_n = cnt.fs / 2
b_l, a_l = proc.signal.butter(5, [45 / fs_n], btype='low')
b_h, a_h = proc.signal.butter(5, [.4 / fs_n], btype='high')

cnt = proc.lfilter(cnt, b_l, a_l)
cnt = proc.lfilter(cnt, b_h, a_h)

cnt = proc.subsample(cnt, 100)
```

Now we need to split the EEG data into epochs of target- and non-target EEG segments by cutting the EEG data around the marker positions. The `SEG_IVAL` defines the position of the segments relative to the markers and is defined as $[0, 700)$ milliseconds after the onset of a marker. The mapping whether a segment belongs to the target- or non-target class is defined in `MARKER_DEF_TRAIN`.

```
epo = proc.segment_dat(cnt, MARKER_DEF_TRAIN, SEG_IVAL)
```

We then calculate the means over time intervals stored in `JUMPING_MEANS_IVALS`, which are continuous 40ms intervals starting from 120ms until 320ms, and continuous 60ms intervals from 320ms until 620ms. From those jumping means we crate the feature vectors and train the classifier using LDA with shrinkage:

```
fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)

cfy = proc.lda_train(fv, shrink=True)
```

The classifier is stored in `cfy` and will be used in the online step.

5.4.3 Running the Online Experiment

In the online step the subject is using the Cake Speller in copy spelling mode, which means he has to spell predefined words. For each letter of the words the Cake Speller runs through all sequences of intensifications and sends for each intensification a marker with the segment number. We will use the classifier that has been trained in the previous step to classify each epoch whether it was the target or a non-target. After a sequence is finished we will send the segment number with the highest probability of being the desired target back to Pyff, so the Cake Speller can present the subject with the result for that sequence (i.e. the selected group or letter).

Like in the test data recording we start by preparing the `PyffComm` object and Mushu's LSL amplifier:

```
pyff = io.PyffComm(PYFF_HOST)
pyff.send_init('SpellCake')
time.sleep(10)
pyff.set_variables({'useOutputModule' : True, 'autoSelect' : False,
                    'allowBackspace' : False, 'pause' : False})
time.sleep(5)

amp = libmushu.get_amp('lslamp')
amp.configure()
```

The online signal processing of the incoming EEG data is very similar to the processing described in the simulated online experiment in Chapter 3.5, so we will comment most of the remaining code only briefly and focus on the new or different things.

In the next step we prepare the low- and high pass filters, and their initial filter delay values.

```
amp_fs = amp.get_sampling_frequency()
amp_channels = amp.get_channels()

fs_n = amp_fs / 2
b_l, a_l = proc.signal.butter(5, [45 / fs_n], btype='low')
b_h, a_h = proc.signal.butter(5, [.4 / fs_n], btype='high')
zi_l = proc.lfilter_zi(b_l, a_l, len(amp_channels))
zi_h = proc.lfilter_zi(b_h, a_h, len(amp_channels))
```

During the processing we will subsample the data from 1kHz to 100Hz. The subsampling internally works by taking, in this case, every 10th sample. Since Mushu's LSL amplifier has no configuration for a block size that would guarantee that `get_data` returns data in multiples of a 10 samples, we have to use Wyrm's `BlockBuffer`. We initialize the `BlockBuffer` with a block length of 10 samples and the `RingBuffer` with a length of 5000 milliseconds. The results of the classifications within each sequence will be stored in the `sequence_result` dictionary, where the keys are the markers and the values the sum of the LDA outputs for that marker.

```
blockbuf = BlockBuffer(10)
ringbuf = RingBuffer(5000)

sequence_result = {}
```

After the preparation is done, we start the amplifier and the Cake Speller, and enter the main loop. Within each iteration of the loop we acquire new data and markers from the amplifier and convert it from Mushu's format into Wyrm's format.

```
amp.start(filename)
pyff.play()
running = True
while running:
```

```
    data, markers = amp.get_data()
    cnt = io.convert_mushu_data(data, markers, amp_fs, amp_channels)
```

Now, we use the `BlockBuffer` to make sure that the data further down the processing line has a length of multiples of 10 samples. For that we simply `append` the new data to the `BlockBuffer` and call `get` for as much data as possible in the `BlockBuffer` that meets the length requirements. Any remaining data (here, at most 9 samples) will stay in the `BlockBuffer` until the next iteration of the loop.

If the `BlockBuffer` did not contain at least one block of data we abort this iteration and go back to query the amplifier for new data.

```
    blockbuf.append(cnt)
    cnt = blockbuf.get()
    if not cnt:
        continue
```

The next steps are the same as described in Section 3.5. We filter the data in an online fashion, subsample it down to 100Hz, put the data into the ring buffer, segment into epochs, calculate the jumping means and create the feature vectors:

```
    cnt, zi_l = proc.lfilter(cnt, b_l, a_l, zi=zi_l)
    cnt, zi_h = proc.lfilter(cnt, b_h, a_h, zi=zi_h)

    cnt = proc.subsample(cnt, 100)
    newsamples = cnt.data.shape[0]

    ringbuf.append(cnt)
    cnt = ringbuf.get()

    epo = proc.segment_dat(cnt, MARKER_DEF_ONLINE, SEG_IVAL, newsamples=newsamples)
    if not epo:
        continue

    fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
    fv = proc.create_feature_vectors(fv)
```

Now, we classify each epoch and map the resulting LDA outputs to their respective markers. In our experiment the targets will have a positive LDA output and the non-targets a negative one. In order to determine the winner of a sequence we can simply add the LDA outputs for each marker and chose the marker with the biggest sum at the end of the sequence.

```
    lda_out = proc.lda_apply(fv, cfy)
    markers = [fv.class_names[cls_idx] for cls_idx in fv.axes[0]]
    result = zip(markers, lda_out)
```

In the last step, we iterate through each element of the marker-output mapping and decide what to do depending on the marker. If the marker signals the end of a sequence, we

determine the winner and send the result back to Pyff, so Pyff can present the subject with the result of the decision.

If we receive a marker signaling the start of a new sequence, we simply reset the scores for each markers by emptying the `sequence_result` dictionary.

If we receive the marker signaling the end of the experiment we set the `running` variable to false which will cause the main loop to stop after this iteration.

In every other case we received a marker for an intensification of a cake element. In those cases we add the value of the LDA output for a marker to its current sum of LDA outputs. If the current marker has not yet a value in `sequence_result` we simply set it the current output value.

```
    for m, score in result:
        if m == MARKER_END_SEQUENCE:
            winner = sorted(sequence_result.items(), key=lambda x: x[1])[-1][0]
            pyff.send_control_signal({'cl_out' : int(winner)})
        elif m == MARKER_START_SEQUENCE:
            sequence_result = {}
        elif m == MARKER_END:
            running = False
        else:
            sequence_result[m] = sequence_result.get(m, 0) + score
```

After the experiment is finished we clean up by closing the connection to Pyff and stopping the amplifier.

```
pyff.quit()
amp.stop()
```

5.5 Results

The subject's task was to spell the words "Computer", "Berlin", "Interface", and "Brain" using the Cake Speller. The resulting words the subject wrote were "Computer", "Berlin", "Interface", and "Brain". All characters were selected correctly, so the character accuracy was 100%.

Since each character in a word needs at least two decisions, one for the group selection and the second for the selection of the character, we can also calculate the decision accuracy over all decisions. The decision accuracy can be lower than the character accuracy, for example when the subject selects the wrong group in the group selection and chooses the "back" character in the second step, to get one level back in order to choose the right group. Figure 5.3 shows the result for each decision that was made during our experiment. As we can see, the classifier chose for all but one cases the correct option. All characters took two correct decisions, except the "B" from the word "Brain" which took one correct-, one incorrect-, and then two correct decisions. We can easily reproduce what happened: The first decision was correct which means the subject chose the correct group that contained

Figure 5.3: This plot shows the result for each decision made by the classifier during the experiment. On the top row are the correct decisions, on the bottom row the incorrect ones. The onsets for the characters to be spelled are marked on the x-axis.

the letter "B", in the second step he chose not the letter "B" but another one. Since there were two other decisions and the speller did not move on to the next letter, he must have taken the "back" button to get back to the group selection, where he finally selected the correct group and the correct letter.

With 57 correct- out of 58 possible choices we get a decision accuracy of 98,2%. However, the subject indicated that after all those correct choices he has made during the experiment, the subject was curious if we were really using his brain patterns to spell the words or just always present him with the correct answer. So after selecting the correct group for the "B" in "Brain", he chose *on purpose* the "back" character, got get back to the group selection, where he finally chose the correct group and letter. If we look at the data this way we could say the classifier achieved a decision accuracy of 100%.

Analysis of the network markers have shown that latency between sending and receiving of markers over the network is in the sub-millisecond- and jitter in the low (i.e. 1-2) millisecond range. This is good enough for almost all BCI applications. Since in most of the cases all three components of the BCI system will run on the same machine, latency and jitter will be, according to our measurements, both in the microseconds range and are thus negligible. We are thus very convinced that Mushu's network markers are a viable and easy to access alternative to whatever special hardware the EEG vendor might require for sending markers. However, we did not perform a systematic measurement that tests all combinations of Operating Systems the individual components of our BCI systems could run on (e.g. Wyrm on Linux, Mushu on Windows, etc.), therefore we cannot make any hard guarantees for the expected marker latency and -jitter. It would be clearly desirable to have those guarantees or to get at least some reliable statements about the expected marker performance given the individual setup at hand. In order to provide this information, we will have to write an integration test that uses all three components of our BCI system, performs the appropriate measurements, and provides the user with the specific latency and jitter values for the given setup.

5.6 Summary and Conclusion

In this chapter we demonstrated that Mushu, Wyrm, and Pyff combined, form a fully functional BCI system capable of successfully running real-time, online BCI experiments.

We started the chapter by explaining the Cake Speller paradigm used in this experiment, and describing in detail the experimental setup of the online experiment. In the main part, we walked the reader through the Wyrm script that orchestrated the components of the BCI system through the training- and online phase of the experiment. The result of the experiment was the best possible one as we achieved a classification accuracy of 100%. Since the classification accuracy in ERP experiments using visual speller paradigms in general are very high and values over 95% can be achieved quite easily [Treder and Blankertz, 2010; Treder et al., 2011], it was here more important to demonstrate that our system performs in that range as well, as a lower accuracy might have hinted at some error in the BCI system. We can thus say with confidence that all three components of our BCI system not only work as expected on their own, but also together as a complete BCI system.

It is worth mentioning, that our experimental setup was relatively complicated, using two computers, sharing data and markers over network. Usually the whole software stack of a BCI experiment runs on one computer. But with this complicated setup we could demonstrate two things: first, we could show that Mushu's network marker work reliably even over network, and second, we could demonstrate how Wyrm can remote control Pyff, that was running on a different computer.

Chapter 6

Summary and Conclusion

In this thesis we presented our completely free- and open source BCI system in Python. Our system consists of three independent components that can be used together as a complete BCI system, or individually in different BCI systems.

We started the thesis with an introduction into brain-computer interfacing. We explained two common ways to translate brain patterns into interpretable output and gave an overview of applications for BCI. We then divided a BCI system into three parts: the signal acquisition, signal processing, and feedback- and stimulus presentation and introduced each part and their contribution to the BCI system. After the reader learned a bit about BCI and the components of a BCI system, we stated the problem of this thesis and gave an overview of related work in this field.

In the following three chapters we presented our solution for each component of our BCI system. Those three chapters have a similar structure: first we introduce the component, then we discuss the requirements and explain the design that lead to the implementation of the component. We then go on explaining how to use the component individually in their domain, summarize the chapter, give some outlook for future work, and conclude.

Mushu is our signal acquisition software, responsible for collecting EEG data from various amplifiers and providing the data to the later part of the BCI system in a unified format. We motivated the need for an operating system- and vendor agnostic signal acquisition, and discussed the requirements. We then explained Mushu's two-layer design for the low level amplifier drivers and the Mushu specific features, like the vendor independent network markers. After we gave an overview of the currently supported amplifiers, we showcased how to use Mushu from two points of views: as an experimenter using Mushu to acquire data, and as a developer writing an amplifier driver for Mushu.

Wyrm is our signal processing toolbox suitable for offline analysis and online experiments. After we discussed the requirements, we explained in depth Wyrm's main data structure that is used in almost all methods of the toolbox and the design principles behind the functional programming style used in this toolbox. We explained how we made Wyrm fast enough to make it suitable for online experiments and we showed how we utilized unit tests and continuous integration to ensure that all methods work as expected. To demonstrate how to use Wyrm for real BCI tasks, we demonstrated the offline classification and analysis for two common BCI tasks: an ERP matrix speller and a motor imagery classification. We also demonstrated a simulated online experiment using the ERP data set.

Pyff is our framework for running and developing feedback- and stimulus applications. We started by explaining the requirements for such a framework and explained in the design section how we solved the task by explaining Pyff's main software components as well as the protocols used to communicate with the rest of the BCI system. We then, again, showed how to use Pyff from two points of views: as an experimenter who wants to run an experiment using Pyff, and as a developer who wants to implement a feedback application himself.

In each of the three chapters describing a component of our system, we explained how to use the component in an independent setting. In Chapter 5 we put all three components together to form a complete BCI system. Furthermore, we demonstrated how to use it to perform a real online experiment. We explained the paradigm used in this experiment, and the experimental setup. We then went through the Wyrm script that ran the whole experiment using Mushu for acquiring the data, its Wyrm's own methods for classification, and Pyff for feedback presentation to the subject. We then discussed the results and summarized the chapter.

All three parts of our BCI system are written in Python, which allows our system to run on all major operating systems. Comparing our system to BCI systems like BCI2000 or OpenViBE, which are both written in C++, our system has the advantage that it is relatively easy to add new functionality by writing Python code. While BCI2000 and OpenViBE mitigate this by providing large sets of amplifier drivers, toolbox methods, and paradigms, a big part of BCI research is in developing new methods or paradigms, or improving the existing ones. This often requires writing new- or modifying existing code, which is a lot more complicated in C++, especially for non computer scientists, than in Python. While C++ is a very efficient and fast language, it is also notorious for the skill level required to write decent C++ code. Python on the other side, being a higher level language than C++, is relatively easy to learn and has a learning curve similar to Matlab, while being also an excellent programming language for scientific computing. BCI2000 and OpenViBE, however, do provide experimental setups for standard paradigms that can be used out of the box. Our BCI system does not provide a turnkey solution for BCI experiments yet, but rather the building blocks to create them.

Another advantage of our system is, that all its components are written as independent components that can also be used in arbitrary BCI systems to replace the respective part of the BCI system. If a group, for example, uses an internally developed BCI system that was created in that lab, and is unsatisfied with their Matlab toolbox because Mathworks introduced a backward incompatible change in Matlab that broke a lot of existing code in their BCI toolbox, they can easily replace that part of their BCI system with Wyrm, without the need to use Mushu and Pyff as well.

The adoption of our system in the BCI community so far is not very wide spread. This is mainly because Mushu and Wyrm are still relatively new. Nevertheless, Brain Products GmbH, a prominent vendor of EEG systems, has already showed interest in using our BCI system in a new product for ALS patients as a result of a joint project between Technische Universität Berlin, Universität Tübingen, and Brain Products GmbH. The proposed product aims to bring the first home-BCI system to ALS-patients. Pyff, the component of our BCI system that exists for a few years already, has also found adoption. In our lab, it has become the de facto standard for feedback- and stimulus applications, it is used by researchers

outside our lab, as well as in the TOBI project, a large European project which aimed for developing practical technology for BCI to improve quality of life of disabled people and the effectiveness of rehabilitation.

We acknowledge that all code, especially code that is developed for research, is always in a flux and new methods are added or existing ones need to be improved. For that reason, all three components are published under free software licenses and their code is freely available on the web-based hosting service GitHub. Furthermore, all three components are well documented, provide examples and are tested by unit tests, giving users of the system not only a realistic chance to use it properly, but also encouraging them to make modifications as needed without the fear of breaking existing functionality. As the code is published on a web-based hosting service that provides, next to the code, issue trackers, wikis, and more, we encourage users not only to use our code, but also to be part of a community that, in the spirit of free- and open source software, contributes new methods or bug fixes back to the projects and thus improves the software for all its users.

Appendix A

Implementation of the LSLAmp

This listing contains the implementation of the `LSLAmp`, a pseudo amplifier, that allows for reading LSL streams from the network.

```
from __future__ import division

import time
import logging

import numpy as np
import pylsl

from libmushu.amplifier import Amplifier


logger = logging.getLogger(__name__)
logger.info('Logger started.')


class LSLAmp(Amplifier):
    """Pseudo Amplifier for lab streaming layer (lsl).

    This amplifier connects to an arbitrary EEG device that is
    publishing its data via lsl to the network. With this class you can
    use this amplifier like a normal `mushu` amplifier.

    https://code.google.com/p/labstreaminglayer/

    Examples
    --------

    >>> amp = libmushu.get_amp('lslamp')
    >>> amp.configure()
    >>> amp.start()
    >>> while True:
    ...     data, marker = amp.get_data()
    ...     # do something with data and/or break the loop
    >>> amp.stop()
```

```
    """

    def configure(self, **kwargs):
        """Configure the lsl device.

        This method looks for open lsl streams and picks the first `EEG`
        and `Markers` streams and opens lsl inlets for them.

        Note that lsl amplifiers cannot be configured via lsl, as the
        protocol was not designed for that. You can only connect (i.e.
        subscribe) to devices that connected (publishing) via the lsl
        protocol.

        """
        # lsl defined
        self.max_samples = 1024
        # open EEG stream
        logger.debug('Opening EEG stream...')
        streams = pylsl.resolve_stream('type', 'EEG')
        if len(streams) > 1:
            logger.warning('Number of EEG streams is > 0, picking the first one.')
        self.lsl_inlet = pylsl.StreamInlet(streams[0])
        # open marker stream
        logger.debug('Opening Marker stream...')
        streams = pylsl.resolve_stream('type', 'Markers')
        if len(streams) > 1:
            logger.warning('Number of Marker streams is > 0, picking the first one.')
        self.lsl_marker_inlet = pylsl.StreamInlet(streams[0])
        info = self.lsl_inlet.info()
        self.n_channels = info.channel_count()
        self.channels = ['Ch %i' % i for i in range(self.n_channels)]
        self.fs = info.nominal_srate()
        logger.debug('Initializing time correction...')
        self.lsl_marker_inlet.time_correction()
        self.lsl_inlet.time_correction()
        logger.debug('Configuration done.')

    def start(self):
        """Open the lsl inlets.

        """
        logger.debug('Opening lsl streams.')
        self.lsl_inlet.open_stream()
        self.lsl_marker_inlet.open_stream()

    def stop(self):
        """Close the lsl inlets.

        """
        logger.debug('Closing lsl streams.')
        self.lsl_inlet.close_stream()
```

```
        self.lsl_marker_inlet.close_stream()

    def get_data(self):
        """Receive a chunk of data an markers.

        Returns
        -------
        chunk, markers: Markers is time in ms since relative to the
        first sample of that block.

        """
        tc_m = self.lsl_marker_inlet.time_correction()
        tc_s = self.lsl_inlet.time_correction()

        markers, m_timestamps = self.lsl_marker_inlet.pull_chunk(timeout=0.0,
            max_samples=self.max_samples)
        # flatten the output of the lsl markers, which has the form
        # [[m1], [m2]], and convert to string
        markers = [str(i) for sublist in markers for i in sublist]

        # block until we actually have data
        samples, timestamps = self.lsl_inlet.pull_chunk(timeout=pylsl.FOREVER,
            max_samples=self.max_samples)
        samples = np.array(samples).reshape(-1, self.n_channels)

        t0 = timestamps[0] + tc_s
        m_timestamps = [(i + tc_m - t0) * 1000 for i in m_timestamps]

        return samples, zip(m_timestamps, markers)

    def get_channels(self):
        """Get channel names.

        """
        return self.channels

    def get_sampling_frequency(self):
        """Get the sampling frequency of the lsl device.

        """
        return self.fs

    @staticmethod
    def is_available():
        """Check if an lsl stream is available on the network.

        Returns
        -------

        ok : Boolean
            True if there is a lsl stream, False otherwise
```

```
        """
        # Return True only if at least one lsl stream can be found on
        # the network
        if pylsl.resolve_streams():
            return True
        return False
```

Appendix B

List of Wyrm's Methods and Data Structures

This chapter lists the data structures and methods of Wyrm's modules as of December 2014. The lists are sorted alphabetically, and show the signature and the short description.

Types Module

The `types` module implements the data types that are used throughout the toolbox.

- `BlockBuffer([samples])` A buffer that returns data chunks in multiples of a block length.
- `Data(data, axes, names, units)` Generic, self-describing data container.
- `RingBuffer(length_ms)` Circular Buffer implementation.

Processing Module

The `processing` module implements the main toolbox methods.

- `append(dat, dat2[, axis, extra])` Append two data objects.
- `append_cnt(dat, dat2[, timeaxis, extra])` Append two continuous data objects.
- `append_epo(dat, dat2[, classaxis, extra])` Append two epoched data objects.
- `apply_csp(epo, filt[, columns])` Apply the CSP filter.
- `calculate_classwise_average(dat[, classaxis])`Calculate the classwise average.
- `calculate_csp(epo[, classes])`Calculate the Common Spatial Pattern (CSP) for two classes.
- `calculate_signed_r_square(dat[, classaxis])`Calculate the signed r^2 values.
- `calculate_spoc(epo)`Compute source power co-modulation analysis (SPoC)

- `clear_markers(dat[, timeaxis])`Remove markers that are outside of the `dat` time interval.
- `correct_for_baseline(dat, ival[, timeaxis])` Subtract the baseline.
- `create_feature_vectors(dat[, classaxis])` Create feature vectors from epoched data.
- `filtfilt(dat, b, a[, timeaxis])` A forward-backward filter.
- `jumping_means(dat, ivals[, timeaxis])` Calculate the jumping means.
- `lda_apply(fv, clf)` Apply feature vector to LDA classifier.
- `lda_train(fv[, shrink])` Train the LDA classifier.
- `lfilter(dat, b, a[, zi, timeaxis])` Filter data using the filter defined by the filter coefficients.
- `lfilter_zi(b, a[, n])` Compute an initial state `zi` for the `lfilter()` function.
- `logarithm(dat)` Computes the element wise natural logarithm of `dat.data`.
- `rectify_channels(dat)` Calculate the absolute values in `dat.data`.
- `remove_channels(*args, **kwargs)` Remove channels from data object.
- `remove_classes(*args, **kwargs)` Remove classes from an epoched Data object.
- `remove_epochs(*args, **kwargs)` Remove epochs from an epoched Data object.
- `segment_dat(dat, marker_def, ival[, ...])` Convert a continuous data object to an epoched one.
- `select_channels(dat, regexp_list[, invert, ...])` Select channels from data object.
- `select_classes(dat, indices[, invert, classaxis])` Select classes from an epoched data object.
- `select_epochs(dat, indices[, invert, classaxis])` Select epochs from an epoched data object.
- `select_ival(dat, ival[, timeaxis])` Select interval from data.
- `sort_channels(dat[, chanaxis])` Sort channels.
- `spectrogram(cnt)` Calculate the spectrogram of a continuous data object.
- `spectrum(dat[, timeaxis])` Calculate the spectrum of a data object.
- `square(dat)` Computes the element wise square of `dat.data`.
- `stft(x, width)` Short time fourier transform of a real sequence.
- `subsample(dat, freq[, timeaxis])` Subsample the data.
- `swapaxes(dat, ax1, ax2)` Swap axes of a Data object.
- `variance(dat[, timeaxis])` Compute the variance along the timeaxis of a data object.

IO Module

Apart from implementing the `PyffComm` object, for communication with Pyff, the `io` module, implements various methods for input/output.

- `convert_mushu_data(data, markers, fs, channels)` Convert mushu data into wyrm's Data format.
- `load(filename)` Load a Data object from a file.
- `load_bcicomp3_ds1(dirname)` Load the BCI Competition III Data Set 1.
- `load_bcicomp3_ds2(filename)` Load the BCI Competition III Data Set 2.
- `load_brain_vision_data(vhdr)` Load Brain Vision data from a file.
- `load_mushu_data(meta)` Load saved EEG data in Mushu's format.
- `save(dat, filename)` Save a Data object into a NumPy `.npy` file.

Plot Module

The `plot` module provides methods for visualization.

- `ax_colorbar(vmin, vmax[, ax, label, ticks])` Draw a color bar
- `ax_scalp(v, channels[, ax, annotate, vmin, vmax])` Draw a scalp plot.
- `beautify()` Set reasonable defaults matplotlib.
- `calc_centered_grid(cols_list[, hpad, vpad])` Calculates a centered grid of Rectangles and their positions.
- `get_channelpos(channame)` Return the x/y position of a channel.
- `plot_channels(dat[, chanaxis, otheraxis])` Plot all channels for a continuous data object.
- `plot_scalp(v, channels[, levels, norm, ...])` Plots the values 'v' for channels 'channels' on a scalp.
- `plot_scalp_ti(v, channels, data, interval[, ...])` Plots a scalp with channels on top
- `plot_spatio_temporal_r2_values(dat)` Calculate the signed r^2 values and plot them in a heatmap.
- `plot_spectrogram(spectrogram, freqs)` Plot a spectrogram.
- `plot_tenten(data[, highlights, hcolors, ...])` Plots channels on a grid system.
- `plot_timeinterval(data[, r_square, ...])` Plots a simple time interval.
- `set_highlights(highlights[, hcolors, set_axes])`

Bibliography

Acqualagna, L. and Blankertz, B. (2011). A gaze independent spelling based on rapid serial visual presentation. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 4560–4563. IEEE.

Allison, B., Luth, T., Valbuena, D., Teymourian, A., Volosyak, I., and Graser, A. (2010). BCI demographics: How many (and what kinds of) people can use an SSVEP BCI? *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 18(2):107–116.

An, X., Ming, D., Sterling, D., Qi, H., and Blankertz, B. (2014). Optimizing visual-to-auditory delay for multimodal BCI speller. In *Engineering in Medicine and Biology Society (EMBC), 2014 36th Annual International Conference of the IEEE*, pages 1226–1229. IEEE.

Ariely, D. and Berns, G. S. (2010). Neuromarketing: the hope and hype of neuroimaging in business. *Nature Reviews Neuroscience*, 11(4):284–292.

Bahramisharif, A. (2012). *Covert visual spatial attention: a robust paradigm for brain-computer interfacing*. PhD thesis, Radboud Universiteit Nijmegen.

Beck, K. (1994). Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18.

Billinger, M., Brunner, C., and Müller-Putz, G. (2014). SCoT: A Python Toolbox for EEG Source Connectivity. *Frontiers in Neuroinformatics*, 8(22).

Birbaumer, N. and Cohen, L. G. (2007). Brain–computer interfaces: communication and restoration of movement in paralysis. *The Journal of physiology*, 579(3):621–636.

Birbaumer, N., Ghanayim, N., Hinterberger, T., Iversen, I., Kotchoubey, B., Kübler, A., Perelmouter, J., Taub, E., and Flor, H. (1999). A spelling device for the paralysed. *Nature*, 398(6725):297–298.

Bissyandé, T. F., Thung, F., Lo, D., Jiang, L., and Réveillère, L. (2013). Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. In *Proceedings of the 37th Annual International Computer Software & Applications Conference (COMPSAC 2013)*, pages 1–10, Kyoto, Japon.

Blankertz, B., Curio, G., and Müller, K.-R. (2002). Classifying single trial EEG: Towards brain computer interfacing. *Advances in neural information processing systems*, 1:157–164.

Blankertz, B., Dornhege, G., Krauledat, M., Müller, K.-R., and Curio, G. (2007). The non-invasive Berlin Brain–Computer Interface: fast acquisition of effective performance in untrained subjects. *NeuroImage*, 37(2):539–550.

Blankertz, B., Lemm, S., Treder, M. S., Haufe, S., and Müller, K.-R. (2011). Single-trial analysis and classification of ERP components – a tutorial. *NeuroImage*, 56:814–825.

Blankertz, B., Müller, K.-R., Krusienski, D., Schalk, G., Wolpaw, J. R., Schlögl, A., Pfurtscheller, G., del R. Millán, J., Schröder, M., and Birbaumer, N. (2006). The BCI competition III: Validating alternative approachs to actual BCI problems. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 14(2):153–159.

Blankertz, B., Müller, K., Curio, G., Vaughan, T. M., Schalk, G., Wolpaw, J. R., Schlogl, A., Neuper, C., Pfurtscheller, G., Hinterberger, T., et al. (2004). The BCI competition 2003: progress and perspectives in detection and discrimination of EEG single trials. *Biomedical Engineering, IEEE Transactions on*, 51(6):1044–1051.

Blankertz, B., Sannelli, C., Halder, S., Hammer, E. M., Kübler, A., Müller, K.-R., Curio, G., and Dickhaus, T. (2010a). Neurophysiological predictor of SMR-based BCI performance. *Neuroimage*, 51(4):1303–1309.

Blankertz, B., Tangermann, M., Vidaurre, C., Fazli, S., Sannelli, C., Haufe, S., Maeder, C., Ramsey, L., Sturm, I., Curio, G., et al. (2010b). The Berlin Brain–Computer Interface: non-medical uses of BCI technology. *Frontiers in neuroscience*, 4.

Blankertz, B., Tomioka, R., Lemm, S., Kawanabe, M., and Müller, K.-R. (2008). Optimizing spatial filters for robust EEG single-trial analysis. *IEEE Signal Processing Magazine*, 25(1):41–56.

Brainard, D. (1997). The psychophysics toolbox. *Spatial vision*, 10(4):433–436.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (1998). Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*.

Brickenkamp, R. (1972). *Test d2*. Verlag für Psychologie, Hogrefe.

Brunner, C., Andreoni, G., Bianchi, L., Blankertz, B., Breitwieser, C., Kanoh, S., Kothe, C. A., Lécuyer, A., Makeig, S., Mellinger, J., Perego, P., Renard, Y., Schalk, G., Susila, I. P., Venthur, B., and Müller-Putz, G. R. (2013). BCI software platforms. In Allison, B. Z., Dunne, S., Leeb, R., Del R. Millán, J., and Nijholt, A., editors, *Towards Practical Brain-Computer Interfaces*, Biological and Medical Physics, Biomedical Engineering, pages 303–331. Springer Berlin Heidelberg.

Brunner, P., Joshi, S., Briskin, S., Wolpaw, J. R., Bischof, H., and Schalk, G. (2010). Does the "P300" speller depend on eye gaze? *Journal of neural engineering*, 7:056013.

Charcot, J. (1874). De la sclérose latérale amyotrophique. *Prog Med*, 2(325):341–453.

Citi, L., Poli, R., Cinel, C., and Sepulveda, F. (2008). P300-based bci mouse with genetically-optimized analogue control. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 16(1):51–61.

Crockford, D. (2006). The application/json media type for javascript object notation (json).

Daly, J. J. and Wolpaw, J. R. (2008). Brain–Computer Interfaces in neurological rehabilitation. *The Lancet Neurology*, 7(11):1032–1043.

Donchin, E., Spencer, K., and Wijesinghe, R. (2000). The mental prosthesis: assessing the speed of a p300-based brain-computer interface. *Rehabilitation Engineering, IEEE Transactions on*, 8(2):174–179.

Dornhege, G., del R. Millán, J., Hinterberger, T., McFarland, D., and Müller, K.-R., editors (2007). *Toward Brain-Computer Interfacing*. MIT press, Cambridge, MA.

Dähne, S., Meinecke, F. C., Haufe, S., Höhne, J., Tangermann, M., Müller, K.-R., and Nikulin, V. V. (2014). SPoC: a novel framework for relating the amplitude of neuronal oscillations to behaviorally relevant parameters. *NeuroImage*, 86(0):111–122.

Escolano, C., Murguialday, A. R., Matuz, T., Birbaumer, N., and Minguez, J. (2010). A telepresence robotic system operated with a P300-based brain-computer interface: Initial tests with ALS patients. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 4476–4480. IEEE.

Farwell, L. A. and Donchin, E. (1988). Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. *Electroencephalography and clinical Neurophysiology*, 70(6):510–523.

Freeman, E., Robson, E., Bates, B., and Sierra, K. (2004). *Head first design patterns*. O'Reilly Media.

Galán, F., Nuttin, M., Lew, E., Ferrez, P. W., Vanacker, G., Philips, J., and Millán, J. d. R. (2008). A brain-actuated wheelchair: asynchronous and non-invasive brain–computer interfaces for continuous control of robots. *Clinical Neurophysiology*, 119(9):2159–2169.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Geller, A., Schleifer, I., Sederberg, P., Jacobs, J., and Kahana, M. (2007). PyEPL: A cross-platform experiment-programming library. *Behavior research methods*, 39(4):950.

Gevins, A. and Smith, M. (2000). Neurophysiological measures of working memory and individual differences in cognitive ability and cognitive style. *Cerebral Cortex*, 10(9):829.

Goodger, D. and van Rossum, G. (2001). Docstring conventions.

Gramfort, A., Luessi, M., Larson, E., Engemann, D. A., Strohmeier, D., Brodbeck, C., Goj, R., Jas, M., Brooks, T., Parkkonen, L., and Hämäläinen, M. (2013). MEG and EEG data analysis with MNE-Python. *Frontiers in Neuroscience*, 7(267).

Haufe, S., Meinecke, F., Görgen, K., Dähne, S., Haynes, J.-D., Blankertz, B., and Bießmann, F. (2014). On the interpretation of weight vectors of linear models in multivariate neuroimaging. *NeuroImage*, 87:96–110.

Haufe, S., Treder, M. S., Gugler, M. F., Sagebaum, M., Curio, G., and Blankertz, B. (2011). EEG potentials predict upcoming emergency brakings during simulated driving. *Journal of neural engineering*, 8(5):056001.

Höhne, J., Schreuder, M., Blankertz, B., and Tangermann, M. (2011). A novel 9-class auditory ERP paradigm driving a predictive text entry system. *Frontiers in neuroscience*, 5.

Hunter, J. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95.

Jasper, H. H. (1958). The ten twenty electrode system of the international federation. *Electroencephalography and clinical neurophysiology*, 10:371–375.

Klobassa, D. S., Vaughan, T., Brunner, P., Schwartz, N., Wolpaw, J., Neuper, C., and Sellers, E. (2009). Toward a high-throughput auditory P300-based brain–computer interface. *Clinical Neurophysiology*, 120(7):1252–1261.

Kohlmorgen, J., Dornhege, G., Braun, M., Blankertz, B., Müller, K.-R., Curio, G., Hagemann, K., Bruns, A., Schrauf, M., and Kincses, W. (2007). Improving human performance in a real operating environment through real-time mental workload detection. In Dornhege, G., del R. Millán, J., Hinterberger, T., McFarland, D., and Müller, K.-R., editors, *Toward Brain-Computer Interfacing*, pages 409–422. MIT Press, Cambridge, MA.

Kothe, C. (2013). Lab Streaming Layer (LSL).

Kothe, C. A. and Makeig, S. (2013). BCILAB: a platform for brain–computer interface development. *Journal of neural engineering*, 10(5):056014.

Krell, M. M., Straube, S., Seeland, A., Wöhrle, H., Teiwes, J., Metzen, J. H., Kirchner, E. A., and Kirchner, F. (2013). pySPACE—a signal processing and classification environment in Python. *Frontiers in Neuroinformatics*, 7.

Krepki, R., Blankertz, B., Curio, G., and Müller, K.-R. (2007). The Berlin Brain-Computer Interface (BBCI)–towards a new communication channel for online control in gaming applications. *Multimedia Tools and Applications*, 33(1):73–90.

Lal, T. N., Hinterberger, T., Widman, G., Schröder, M., Hill, N. J., Rosenstiel, W., Elger, C. E., Schölkopf, B., and Birbaumer, N. (2005). Methods towards invasive human brain computer interfaces. In Saul, L. K., Weiss, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 17*, pages 737–744. MIT Press, Cambridge, MA.

Lee, N., Broderick, A. J., and Chamberlain, L. (2007). What is 'neuromarketing'? a discussion and agenda for future research. *International Journal of Psychophysiology*, 63(2):199–204.

Lemm, S., Blankertz, B., Dickhaus, T., and Müller, K.-R. (2011). Introduction to machine learning for brain imaging. *NeuroImage*, 56(2):387 – 399. Multivariate Decoding and Brain Reading.

Lipow, M. (1982). Number of faults per line of code. *Software Engineering, IEEE Transactions on*, SE-8(4):437–439.

Louden, K. et al. (2011). *Programming languages: principles and practices*. Cengage Learning.

Lutz, M. (2006). *Programming Python*. O'Reilly Media, Inc.

Mackworth, N. (1948). The breakdown of vigilance durning prolonged visual search. *The Quarterly Journal of Experimental Psychology*, 1(1):6–21.

Mak, J. N. and Wolpaw, J. R. (2009). Clinical applications of Brain-Computer Interfaces: current state and future prospects. *Biomedical Engineering, IEEE Reviews in*, 2:187–199.

McFarland, D. J., Sarnacki, W. A., and Wolpaw, J. R. (2010). Electroencephalographic (EEG) control of three-dimensional movement. *Journal of Neural Engineering*, 7(3):036007.

McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media.

Müller, K.-R., Tangermann, M., Dornhege, G., Krauledat, M., Curio, G., and Blankertz, B. (2008). Machine learning for real-time single-trial EEG-analysis: from brain–computer interfacing to mental state monitoring. *Journal of neuroscience methods*, 167(1):82–90.

Müller, K.-R., Anderson, C., and Birch, G. (2003). Linear and nonlinear methods for brain-computer interfaces. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 11(2):165–169.

Müller-Putz, G., Scherer, R., Neuper, C., and Pfurtscheller, G. (2006). Steady-state somatosensory evoked potentials: suitable brain signals for brain-computer interfaces? *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 14(1):30–37.

Müller-Putz, G. R. and Pfurtscheller, G. (2008). Control of an electrical prosthesis with an SSVEP-based BCI. *Biomedical Engineering, IEEE Transactions on*, 55(1):361–364.

Nijholt, A., Bos, D. P.-O., and Reuderink, B. (2009). Turning shortcomings into challenges: Brain–computer interfaces for games. *Entertainment Computing*, 1(2):85–94.

Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20.

Oostenveld, R., Fries, P., Maris, E., and Schoffelen, J.-M. (2011). FieldTrip: open source software for advanced analysis of MEG, EEG, and invasive electrophysiological data. *Computational intelligence and neuroscience*, 2011:1.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Peirce, J. W. (2007). PsychoPy–Psychophysics software in Python. *Journal of Neuroscience Methods*, 162(1-2):8–13.

Pérez, F. and Granger, B. (2007). IPython: A System for Interactive Scientific Computing. *Computing in Science Engineering*, 9(3):21–29.

Peterson, B. (2008). Python 2.7 release schelude. *Python Enhancement Proposals, PEP*, 1.

Pfurtscheller, G. and Lopes da Silva, F. H. (1999). Event-related EEG/MEG synchronization and desynchronization: basic principles. *Clinical neurophysiology*, 110(11):1842–1857.

Pfurtscheller, G. and Neuper, C. (1997). Motor imagery activates primary sensorimotor area in humans. *Neuroscience letters*, 239(2):65–68.

Porbadnigk, A. K., Antons, J., Blankertz, B., Treder, M. S., Schleicher, R., Moller, S., and Curio, G. (2010). Using ERPs for assessing the (sub) conscious perception of noise. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 2690–2693. IEEE.

Porbadnigk, A. K., Scholler, S., Blankertz, B., Ritz, A., Born, M., Scholl, R., Muller, K., Curio, G., and Treder, M. S. (2011). Revealing the neural response to imperceptible peripheral flicker with machine learning. In *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, pages 3692–3695. IEEE.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, 33(10):23–29.

Quek, M. (2013). *The role of simulation in developing and designing applications for 2-class motor imagery brain-computer interfaces*. PhD thesis, University of Glasgow.

Quek, M., Boland, D., Williamson, J., Murray-Smith, R., Tavella, M., Perdikis, S., Schreuder, M., and Tangermann, M. (2011). Simulating the feel of brain-computer interfaces for design, development and social interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 25–28. ACM.

Ramoser, H., Muller-Gerking, J., and Pfurtscheller, G. (2000). Optimal spatial filtering of single trial eeg during imagined hand movement. *Rehabilitation Engineering, IEEE Transactions on*, 8(4):441–446.

Renard, Y., Lotte, F., Gibert, G., Congedo, M., Maby, E., Delannoy, V., Bertrand, O., and Lécuyer, A. (2010). OpenViBE: an open-source software platform to design, test, and use brain-computer interfaces in real and virtual environments. *Presence: teleoperators and virtual environments*, 19(1):35–53.

Sajda, P., Gerson, A., Müller, K., Blankertz, B., and Parra, L. (2003). A data analysis competition to evaluate machine learning algorithms for use in Brain-Computer Interfaces. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, 11(2):184–185.

Schaeff, S., Treder, M., Venthur, B., and Blankertz, B. (2011). Motion-based ERP spellers in a covert attention paradigm. In *Neuroscience letters*, volume 500, page e11.

Schaeff, S., Treder, M., Venthur, B., and Blankertz, B. (2012). Exploring motion VEPs for gaze-independent communication. *Journal of neural engineering*, 9(4).

Schalk, G., McFarland, D. J., Hinterberger, T., Birbaumer, N., and Wolpaw, J. R. (2004). BCI2000: a general-purpose brain-computer interface (BCI) system. *Biomedical Engineering, IEEE Transactions on*, 51(6):1034–1043.

Schlögl, A. and Brunner, C. (2008). BioSig: a free and open source software library for BCI research. *Computer*, 41(10):44–50.

Schreiner, T., Hill, N., Schreiner, T., Puzicha, C., and Farquhar, J. (2008). Development and Application of a Python Scripting Framework for BCI2000. *Master's thesis. Universität Tübingen, Tübingen*.

Schreuder, M., Blankertz, B., and Tangermann, M. (2010). A new auditory multi-class brain-computer interface paradigm: spatial hearing as an informative cue. *PloS one*, 5(4):e9813.

Schreuder, M., Riccio, A., Ramsay, A., Dähne, S., Höhne, J., Quek, M., Crossan, A., Mattia, D., Murray-Smith, R., and Tangermann, M. (2012). End user performance in a novel social BCI application: the photobrowser. In *Proceedings of the 3rd TOBI workshop*.

Schreuder, M., Riccio, A., Risetti, M., Dähne, S., Ramsay, A., Williamson, J., Mattia, D., and Tangermann, M. (2013). User-centered design in Brain–Computer Interfaces—a case study. *Artificial intelligence in medicine*, 59(2):71–80.

Schubert, R., Tangermann, M., Haufe, S., Sannelli, C., Simon, M., Schmidt, E., Kincses, W., and Curio, G. (2008). Parieto-occipital alpha power indexes distraction during simulated car driving. *International Journal of Psychophysiology*, 69(3):214.

Serruya, M. D., Hatsopoulos, N. G., Paninski, L., Fellows, M. R., and Donoghue, J. P. (2002). Brain-machine interface: Instant neural control of a movement signal. *Nature*, 416(6877):141–142.

Straw, A. D. (2008). Vision Egg: An Open-Source Library for Realtime Visual Stimulus Generation. *Frontiers in Neuroinformatics*, 2.

Tanenbaum, A. S. (2001). *Modern Operating Systems*. Prentice Hall PTR.

Tangermann, M., Krauledat, M., Grzeska, K., Sagebaum, M., Blankertz, B., Vidaurre, C., and Müller, K.-R. (2008). Playing Pinball with non-invasive BCI. In *NIPS*, pages 1641–1648.

Tangermann, M., Müller, K.-R., Aertsen, A., Birbaumer, N., Braun, C., Brunner, C., Leeb, R., Mehring, C., Miller, K. J., Müller-Putz, G. R., et al. (2012). Review of the BCI competition IV. *Frontiers in neuroscience*, 6.

Tomioka, R. and Müller, K.-R. (2010). A regularized discriminative framework for EEG analysis with application to Brain–Computer Interface. *NeuroImage*, 49(1):415–432.

Treder, M. and Venthur, B. (2009). (C)overt attention and P300-speller design. BCI Workshop on Advances in Neurotechnology.

Treder, M. S. and Blankertz, B. (2010). (C)overt attention and visual speller design in an ERP-based brain-computer interface. *Behavioral and Brain Functions*, 6:28.

Treder, M. S., Schmidt, N. M., and Blankertz, B. (2011). Gaze-independent brain–computer interfaces based on covert attention and feature attention. *Journal of neural engineering*, 8(6):066003.

Velliste, M., Perel, S., Spalding, M. C., Whitford, A. S., and Schwartz, A. B. (2008). Cortical control of a prosthetic arm for self-feeding. *Nature*, 453(7198):1098–1101.

Venthur, B. and Blankertz, B. (2008). A Platform-Independent Open-Source Feedback Framework for BCI Systems. In *Proceedings of the 4th International Brain-Computer Interface Workshop and Training Course 2008*, pages 385–389. Verlag der Technischen Universität Graz.

Venthur, B. and Blankertz, B. (2009). A Platform-Independent Open-Source Feedback Framework for BCI Systems. 18th Annual Computational Neuroscience Meeting (CNS*2009).

Venthur, B. and Blankertz, B. (2010). Pyff—A Plattform-Independent Open-Source Feedback Framework for BCI Systems. 4th International Brain-Computer Interface Meeting.

Venthur, B. and Blankertz, B. (2012). Mushu, a free-and open source BCI signal acquisition, written in Python. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 1786–1788. IEEE.

Venthur, B. and Blankertz, B. (2013). Towards a Free and Open Source BCI System written in Python. 5th International Brain-Computer Interface Meeting.

Venthur, B. and Blankertz, B. (2014a). A Free and Open Source BCI System in Python. 12th International Conference on Cognitive Neuroscience.

Venthur, B. and Blankertz, B. (2014b). Wyrm, A Pythonic Toolbox for Brain-Computer Interfacing. In de Buyl, P. and Varoquaux, N., editors, *Proceedings of the 7th European Conference on Python in Science (EuroSciPy 2014)*. in press.

Venthur, B., Blankertz, B., Gugler, M. F., and Curio, G. (2010a). Novel Applications of BCI Technology: Psychophysiological Optimization of Working Conditions in Industry. In *Proceedings of the 2010 IEEE Conference on Systems, Man and Cybernetics (SMC2010)*, pages 417–421.

Venthur, B., Dähne, S., Höhne, J., Heller, H., and Blankertz, B. (2014). Wyrm: A Brain-Computer Interface Toolbox in Python. *Neuroinformatics*. under review.

Venthur, B., Scholler, S., Williamson, J., Dähne, S., Treder, M. S., Kramarek, M. T., Müller, K.-R., and Blankertz, B. (2010b). Pyff—A Pythonic Framework for Feedback Applications and Stimulus Presentation in Neuroscience. *Frontiers in Neuroinformatics*, 4(100).

Vidal, J.-J. (1973). Toward Direct Brain-Computer Communication. *Annual review of Biophysics and Bioengineering*, 2(1):157–180.

Warsaw, B., Hylton, J., Goodger, D., and Coghlan, N. (2000). PEP purpose and guidelines. *Python Enhancement Proposals, PEP*, 1.

Wolpaw, J. R. and Wolpaw, E. W., editors (2012). *Brain-Computer Interfaces: principles and practice*. Oxford University press. ISBN-13: 978-0195388855.

Woolsey, C. N., Erickson, T. C., and Gilson, W. E. (1979). Localization in somatic sensory and motor areas of human cerebral cortex as determined by direct recording of evoked potentials and electrical stimulation. *Journal of neurosurgery*, 51(4):476–506.

Zaitcev, P. (2005). The usbmon: USB monitoring framework. In *Linux Symposium*, page 291.

Zhao, Q., Zhang, L., and Cichocki, A. (2009). EEG-based asynchronous BCI control of a car in 3D virtual reality environments. *Chinese Science Bulletin*, 54(1):78–87.