Implementation of a Simple Asynchronous Pipeline Framework (SAPF) for construction of real-time BCI systems

Tom Andersen

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6141
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2021 by Tom Andersen. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2021

1

Abstract

This thesis attempts to implement a library in pure Python for building real-time Brain-Computer Interface (BCI) systems. The library does this by employing nodes containing data transformation methods—filtering, classification, data acquisition, and more. These nodes are linked together to create pipelines of nodes in which data flows. Due to the asynchronous nature of the data flow the library was named Simple Asynchronous Pipeline Framework (SAPF).

Moreover, a demonstration BCI was also built using SAPF. In parallel with this, a small game application was developed which the specific BCI system was used to control. At first eye-movements were considered as a user command for controlling the example BCI system. However, in a later phase this was changed to eye-blinks and jaw clenching.

Acknowledgements

First and foremost I want to thank my supervisors, Frida Heskebeck and Carolina Bergeling, for their time, comments, and optimism. During times like these it is hard to find motivation, and I don't think I would have finished the thesis without your support.

I also want to thank my family for your support; my mother and father, Anette and Toni, for supporting me and my education and work, even though you are not really sure what I am doing exactly. And I also want to thank you, my sister, Jonna, for your cheerfulness, but also for letting me borrow your camera.

Lastly, I also want to thank my flatmates, the people at Real Fighter club, and my colleagues and friends Josef and Daniel for this semester. The situation that these times brings would had a much higher negative impact on my thesis if it was not for them.

Contents

1. Abstract **3**

List of Figures **9**

List of Tables **10**

2. Introduction **11**
- 2.1 Current situation . . . 11
- 2.2 Objectives, scope and methodology of thesis . . . 11
- 2.3 Thesis outline . . . 12

3. Background **13**
- 3.1 A brief history of the electroencephalography and brain-computer interfaces . . . 13
- 3.2 Categorization of BCI systems . . . 14
- 3.3 Ethical aspects . . . 16
- 3.4 EEG acquisition . . . 16
- 3.5 State-of-the-art real-time BCI systems . . . 19
- 3.6 Ocular activity and EEG . . . 19
- 3.7 A quick review of digital filter theory . . . 20

4. Implementing a Simple Asynchronous Pipeline Framework (SAPF) **24**
- 4.1 Existing BCI platforms . . . 24
- 4.2 Definitions and figure interpretations . . . 26
- 4.3 The SAPF implementation . . . 26
- 4.4 The LSL node . . . 32
- 4.5 The pre-processing node . . . 33
- 4.6 The processing node . . . 39
- 4.7 The feature extractor node . . . 39
- 4.8 The classifier node . . . 40
- 4.9 The application interface node . . . 40
- 4.10 The stimuli node . . . 40
- 4.11 Final discussion and further work regarding SAPF . . . 50

5. Implementing a game application 52
5.1 Game world 52
5.2 Remote control of the game application 55
5.3 Player movement 57

6. Implementation of a demonstration using SAPF and the game 60
6.1 User commands 60
6.2 Hardware selection 61
6.3 The LSL node configuration 61
6.4 The pre-processing node configuration 61
6.5 The processing node configuration 62
6.6 The feature extractor node configuration 63
6.7 The stimuli node and adapter configuration 65
6.8 The classification node configuration 67
6.9 Building and running the pipelines 67
6.10 Discussion 70

7. Final remarks 72

8. Appendix 74

Bibliography 77

List of Figures

3.1 Mason and Birch's functional BCI model 15
3.2 Electrode placements according to the 10/10 system. 17
3.3 The EasyCap and Smarting amplifier. 18
3.4 The Muse S EEG cap. 19

4.1 The TOBI CIP pipeline 25
4.2 Decomposition of functionality 27
4.3 Overview of the intrinsic behaviour of a node 28
4.4 Two example SAPF pipelines 31
4.5 LSL node overview 32
4.6 Pre-processing node overview. 34
4.7 Chunk-wise filtering 34
4.8 Chunk-wise filtering, baseline correction then filtering 36
4.9 Chunk-wise filtering, filtering then baseline correction. 37
4.10 Band-pass filtering to remove drift 38
4.11 Feature extractor node overview 39
4.12 Stimulation-response mapping 41
4.13 Stimuli node application interface 42
4.14 Stimuli node and adapter communication 43
4.15 Stimuli and adapter due in the loop 45
4.16 Example of focus point selection 49

5.1 Simple game graphics 53
5.2 Updated game graphics. 54
5.3 Remote control of the game application 56
5.4 Game remote control diagram of second iteration software 58

6.1 Filter bank filters 64
6.2 Feature extraction example using filter bank 64
6.3 Collect the crowns example world 65
6.4 Limited traversability 66

6.5 The calibration pipeline 69
6.6 The classification pipeline. 70

List of Tables

4.1 Overview of SAPF nodes and their behaviour. 30

6.1 Digital filter design arguments 62
6.2 The two pipelines 67

2

Introduction

2.1 Current situation

Brain-computer interfaces (BCIs) generally aims to measure neural activity in the user's brain, extract useful information from these signals, and use this information to control application, supplement or restore human body functions, and more [Schalk and Allison, 2018]. Examples include controlling a model plane using signals from the motor complex of the user [Kryger et al., 2016], or attempting to restore limb movements in paralysed patients by means of so called bi-direcitonal BCI [Collinger et al., 2018].

As of 2018 there exists at least a handful of BCI libraries and platforms, some of them are BCI2000, BioSig, OpenVibe, TOBI, and PMW. These platforms and libraries aim to simplify the construction of BCI systems. Some of these require programming knowledge in either MATLAB, C, or C++. However, platforms such as OpenVibe allows the user to construct BCI platforms with a graphical language [Stegman et al., 2020].

2.2 Objectives, scope and methodology of thesis

It is debatable if there already exists a minimalistic open-source BCI platform written in a simple language such as Python. However, this thesis attempts to fill this potential gap by I) constructing a simple BCI library for constructing real-time BCI pipelines, and II) constructing a specific real-time BCI system, acting as a demonstration, using the library, and lastly III) implement a small tile-based game application, acting as a toy example, which the user of the BCI can control, in real-time, using user commands.

The user commands considered in this thesis will not be of purely neural origin however, instead eye movements, eye-blinks, and jaw clenching will be used. These signals are generally of great magnitude and are in many systems considered artefacts. It is common to attempt to remove them as they make EEG data analysis difficult or even impossible [Plöchl et al., 2012][Simon L. et al., 2017]. These com-

mands will therefore only act as a place holders. One should note that these easily can be exchanged for other user commands in the future.

The first real-time BCI system developed using the library used eye movements only. However, due to poor classification accuracy the user commands were later swapped with eye-blinks and jaw clenches.

Moreover, in this thesis only non-invasive electroencephalography (EEG) data acquisition will be considered.

2.3 Thesis outline

Chapter 3 introduces previous work as well as the history of the technologies used in this thesis. Chapter 4 describes the implementation of the library. Chapter 5 is dedicated to the implementational details of the game application. The final chapter, Chapter 6, described the implemented BCI system using the library.

3

Background

This chapter presents the background needed to better understand this thesis. In Section 3.1 the history of the so called electroencephalography (EEG) and Brain-Computer Interfaces (BCI) are briefly introduced. In Section 3.2 some categories and examples of BCI systems will be presented. Section 3.3 briefly touches ethical aspects of BCI. Section 3.4 introduces the reader to hardware and software related to the acquisition of user EEG data. Section 3.5 presents some already existing BCI systems. Section 3.6 introduces the reader to how ocular activities, such as eye movements and eye-blinks, affects the EEG. And finally, in Section 3.7 the fundamentals of digital signal processing is introduced. This section will also expand briefly on filtering of EEG signals.

3.1 A brief history of the electroencephalography and brain-computer interfaces

The invention of EEG

Year 1929 the German doctor Hans Berger published his *Über das Elektrenkephalogramm des Menschen*—an article on electroencephalography (EEG)—where he describes his findings of electrical potential oscillations in the measurements on the scalp of his patients. He found these signals by placing the same kind of electrodes that, at the time, was also used for electrocardiogram (ECG), in the vicinity of enlarged trephine openings—a hole drilled into the scull. By the use of a galvanometer—a device for measuring small current fluctuations—Berger could amplify the signals measured from the patient's scull enough to be displayed on so called photographic paper [Millett, 2001].

It took Berger many attempts to find any readable signals, and even after discovering the alpha and beta waves—two types of brain signals—Berger was sceptical towards the results. Not only he was sceptical, most of his work was regarded with indifference in Europe. In 1934 the leading neurophysiologist Lord Edgar Adrian even constructed an experiment to show that Berger's finding indeed were artefacts. However, Adrian's results were not as expected. His experiment showed clear

indications that the alpha and beta waves were indeed not artifacts. He immediately published his results which lead to a burst of excitement and EEG becoming the mainstream diagnostic tool in neurology and psychiatry [Kaplan, 2011][Millett, 2001]

Though Berger is considered the discoverer of the alpha and beta wave, and the EEG [Kaplan, 2011], much of Berger's work were based on previous work by other scientists like Richard Caton. Caton was an English scientist known for his recordings of electrical activity on animals. In 1875 he reported his first experiments in which he found—using a galvanometer—an increase in electrode current with sleep [Haas, 2003].

The emergence of Brain-Computer Interfaces

Berger's first recordings were, as one might expect, not nearly as sophisticated as what one might achieve today. His setup was analog to the core, and no processing was done to the EEG signals before being drawn on photographic paper [Millett, 2001]. A big milestone of the history of BCI and EEG was therefore, as we will see, the leap away from working with analog signals and the introduction of digitalization.

Thelma Estrin, an electrical engineer from the University of Wisconsin, was one of the first to create a system that was able to convert analog EEG signals into the digital domain [Kübler, 2020]. As we know today, the digitization of signal processing opens up a new world of possibilities such as more sophisticated preprocessing, data storage, and much more.

In his paper from 1973, Vidal, [Vidal, 1973], introduces the Brain Computer Interface project—a pilot project in direct brain-computer communication—conducted at the University of California. Vidal further discusses the applications of different types of EEG recordings and the general apparatus setup which was named The Computer Interface Laboratory. The paper is seen as the first attempt to systematically clarify the different concepts regarding EEG as well as their limitations. [Vidal, 1973].

Due to the lack of a standardized language in the BCI research community between the 70s to 90s, Mason and Birch attempted to generalize the existing BCI systems into models to simplify communication between research groups [Mason and Birch, 2003]. Among others, they identified one type of BCI system referred to as the functional BCI model, see Figure 3.1.

3.2 Categorization of BCI systems

There are many different types of BCI systems and [Chang S. Nam et al., 2018] suggests categorizing them according to their so called *mode of operation*, *brain signal pattern*, *stimulus modality*, and *recording method*. A further explanation will be made in this section of the different categories.

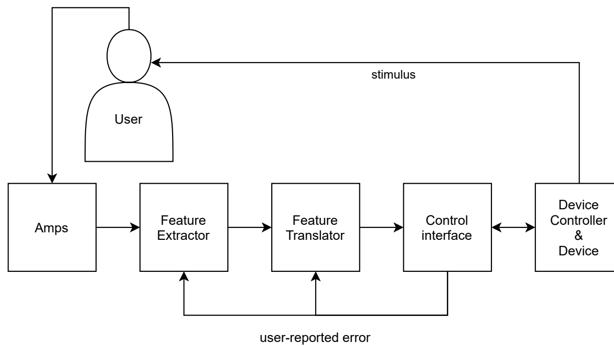


Figure 3.1: A version of Mason and Birch's functional BCI model [Mason and Birch, 2003].

Mode of operation

The mode of operation for the BCI system specifies how signals from the user are obtained. The mode of operation can either be synchronous, or asynchronous. In synchronous mode the BCI presents information to the user to obtain the brain signal in a synchronous fashion. In asynchronous mode, however, the BCI system does not expect the user to respond synchronously with the presentation of information—the user's response time is determined by the user herself.

Brain signal pattern and some of their BCI applications

The brain signal pattern refers to the type of brain signals that will be used to control, or communicate, with the BCI system.

Visual-Evoked Potential (VEP) based BCI

Visual-evoked potential (VEP) are the EEG activity observed in users when a visual stimuli—such as a flash of light—is applied in the subject's visual field [Bin et al., 2009].

There exists multiple types of VEPs, but they are all based on the same fundamental principle that a visual stimulation results in a measurable response.

A VEP-based BCI system generally consists of multiple visual targets which the user can focus attention on. An example of this is the VEP-based spelling BCIs in which a virtual keyboard of characters is displayed to the user. Each target on the screen—i.e. each character—is coded with a unique sequence of flickering. Each of these codings[1] gives rise to different unique VEPs in the brain signal produces by the subject. Based on this the BCI system can determine which target the subject focuses attention on. By looking at different characters the user can thus spell out

[1] If a single frequency of flickering is selected it is called Steady-State Visual Evoked Potential (SSVEP) [Masood et al., 2020]

words without any need of physical movement such as eye movement or physical typing [Masood et al., 2020].

Stimulus modality

The stimulus modality refers to the method used to present the user with stimuli. This could for example be visual (what VEP-based BCIs uses), tactile, auditory, or a combination of them all [Chang S. Nam et al., 2018].

Recording method

There are two types of recording methods: invasive and noninvasive.

3.3 Ethical aspects

In clinical use of BCIs, one of the most important aspects might be informed consent-related. Patients can be promised too much, and in some cases some of the patients might not be able to use the BCI system at all. This is especially true for severely disabled patients who might not be able to sufficiently operate non-invasive EEG-based BCIs. Detrimental effects could also not be excluded—due to BCI training which evokes neuroplasticity and thus changes behaviour. This must also be made clear to any user [Grübler et al., 2014].

However, as the system here developed will not be used by any wider audience this is not relevant in this thesis.

3.4 EEG acquisition

In this section different aspects of hardware and software will be presented that can be used for EEG acquisition.

Electrode placement

Defined 1958, the 10/20 system standardizes the placement of EEG electrodes. It is, as of today, the de faco standard for clinical EEG [Oostenveld and Praamstra, 2001]. There exists extensions of the 10/20 system, such as the 10/10 and 10/5 system, which allows for more electrodes [Jurcak et al., 2007]. In Figure 3.2 the electrode placement of the 10/10 system can be seen.

Lab streaming layer and time-synchronization

Lab streaming layer (LSL) is a protocol for collecting time series data in research experiments (e.g. a stream of EEG data from a EEG-cap). Relating LSL to the e.g. Mason's model (Figure 3.1), LSL handles the communication between the user, or the data source, at the first block in the BCI pipeline.

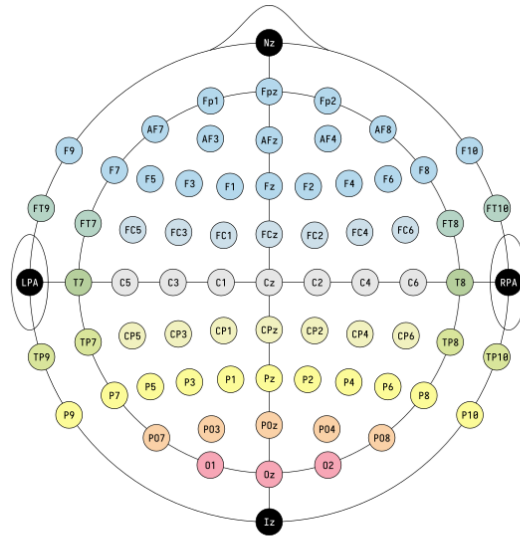


Figure 3.2: Electrode placements according to the 10/10 system.

Moreover, LSL handles time-synchronization of time series and networking. LSL is typically used to stream data over a local network from one or more applications with integrated LSL functionality, together with event marker streams, to a data acquisition software on another device in the network [Computational Neuroscience, n.d.] Event markers can be used to associate intervals in the collected time series with different external events, such as when a stimulation was presented, or when a button was pressed.

When receiving multiple streams from different devices the problem of time synchronization occurs (e.g. the event marker stream might be hosted by a personal computer, whereas the EEG data from dedicated hardware in the EEG-cap). The time-synchronization that LSL implements uses a protocol similar to the Network Time Protocol (NTP) where UDP packages are repeatedly sent to the remote device to estimate the round-trip time and the clock offset[2] of the device. The process of doing this is referred to as the Clock Filter Algorithm [Computational Neuroscience, n.d.]

LabRecorder

The default software that comes with LSL is the LabRecorder software which can be used to collect data off of the streams and save it into Extensible Data Format (xdf) [Computational Neuroscience, n.d.]

[2] Difference in starting times for the LSL devices.

LabRecorder has a built-in remote control server (RCS) that enables remote control of the software over sockets. One can thus control the LabRecorder application programmatically using e.g. Python.

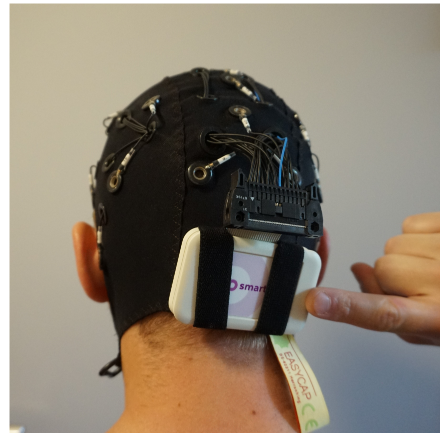
Smarting

SMARTING 24 is a battery-driven mobile EEG amplifier intended for use as a biofeedback research platform, see Figure 3.3b. It supports 24 channel EEG-caps that can be connected via an insulation-displacement connector (an electrical connector design). The amplifier hosts a LSL stream over Bluetooth that can be read by any platform that supports the LSL protocol [Smarting, n.d.]

EasyCap for Smarting

EasyCap for Smarting is a 24 electrode EEG-cap that employs the international 10/20-system, see Figure 3.3a.

(a) EasyCap EEG cap.



(b) EasyCap EEG cap. The Smarting amplifier is attached to the cap using elastic bands.

Figure 3.3: The EasyCap and Smarting amplifier.

Muse S

Muse S is an EEG-cap marketed as a tool for sleep and meditation tracking. Muse S supports 5 channels of EEG electrodes (with placements TP9, AF7, AF8, TP10, and Fpz as reference electrode according to the 10/20-system) [Muse, n.d.(a)][Krigolson et al., 2017], photoplethysmogram, pulse oxiometry, and accelerometer and gyroscope measurements [Muse, n.d.(b)]. Muse S has a sampling frequency of 256 Hz. The electrodes are of so called dry-type, which means than no conductive paste is needed between the skin and the electrode.

Figure 3.4: The Muse S amplifier attached to the EEG cap. The white strip near the ear is one of the electrodes.

3.5 State-of-the-art real-time BCI systems

There exists several BCI competitions open to researcher to commit his or hers BCI research. One of them are the Brain-Computer Interface Race[3], which is a part of Cybathlon—a multi sport event for people with physical disabilities.

During Brain-Computer Interface Race teams—consisting of developers, researchers, and a disabled person acting as the "pilot"—are given virtual cars in the game *BrainDriver* which they can control using a BCI system. The pilots compete to reach the goal line first, and have to control the virtual car left and right to not crash into objects and to optimize speed.

3.6 Ocular activity and EEG

Saccades and eye-blinks

Ocular activities—such as saccades (eye movements) and eye blinks—generate signals in the EEG measurements. Due to the relatively large amplitudes of these signals, compared to other signal patters that arise from neural activity, and due to the ocular signals' non-neural origins, they are generally considered as artifacts in most EEG-paradigms. Furthermore, the ocular-related signals are considered generated by at least three mechanisms; corneo-retinal dipole rotation, eyelid movements, and the recruitment of extra-ocular muscles [Plöchl et al., 2012][Picton et al., 2000].

The corneo-retinal rotation effect is due to the ionic gradient between the retinal pigment epithelium. In human eyes this creates a dipole whose axis closely follows

[3] www.cybathlon.ethz.ch/en/event/disciplines/bci

the direction of gaze. Eye movements, i.e. dipole rotations, thus result in potential changes which can be seen in EEG measurements containing saccades. Eye movements in the opposite direction results in a mirror reversed version of the measured signal in the EEG recording [Plöchl et al., 2012].

Interestingly, the movement of eyelids have been found to result in measurable changes in the EEG. One explanation of this effect is the change in resistance between the corneo-retinal and the recording electrodes that is accompanied by eyelid movements, which results in changes in the EEG [Chioran and Yee, 1991].

The lesson one should take from this is that, though not consisting of neural activity only, movement of the eyes and eye-lid do elicit EEG signals that are generally of great amplitude.

Fixation and involuntary eye movements

It is possible to fixate one's gaze at a target for quite some time. However, even though the intent is to keep one's gaze still, there are notable eye movements occurring. Among these movements are drift and involuntary so called micro-saccades [Rolfs, 2009].

These movements affect EEG measurements, and in settings where the corresponding EEG components are not desired the quality of the fixation target can be adjusted to minimize their magnitude. The stability of a fixation target can be measured by how much of these components are present when a certain target is used. It should be noted that external factors also affect the stability. Another factor related to the target itself is its shape and size, and these parameters have been shown to affect the spatial dispersion and rate of micro-saccades, i.e. the fixation stability. Specifically, a smaller target results in lower dispersion [Thaler et al., 2013].

3.7 A quick review of digital filter theory

A quick review of digital filter theory

Frequency filtering refers to the act of applying a digital filter onto a signal of interest. By carefully designing filter parameters specific frequency bands can be attenuated or amplified according to the designer's wishes. Attenuating specific frequency bands can be desired if it is known beforehand that these will contain noise or disturbances that will lower the signal-to-noise ratio.

There are at least two methods of applying a filter to a signal, *convolution in time domain* and *multiplication in frequency domain*.

Convolution in the time domain

The convolution in time domain involves applying the convolution operator between the filter characterizing series—called the impulse response, denoted $h(i) = h_i$. The convolution can be written as:

$$y(n) = (x * h)(n) = \sum_{i=-\infty}^{\infty} h_i x(n-i) \tag{3.1}$$

where $x(n)$ is some discrete series that is the input to the filter, $y(n)$ the filter output, and $n \in \mathbb{Z}$ denotes the time sample.

Note that equation (3.1) is a recursion equation but without any recursive terms. Sometimes filters are described on the form below where recursive terms are included

$$y(n) = \sum_{i=-\infty}^{\infty} b_i y(n-i) + \sum_{j=-\infty}^{\infty} a_j x(n-j) \tag{3.2}$$

where b_i and a_j are some constant parameters defining the filter. However, one can show that equation (3.2) can be described on the form in equation (3.1) by choosing the appropriate values of h_i. Any digital filter described on form (3.2) can be applied to a signal by using the convolution in equation (3.1).

Lastly, one should note that i.e. equation (3.1) potentially includes future samples when calculating the current sample. However, if $h_i = 0, i < 0$ then this is prevented. Filters of this type are called casual filters and they only include previous data points. By contrast, filters where $h_i \neq 0, i < 0$ are called non-casual.

Multiplication in the frequency domain

According to the convolution theorem, that states that convolutions in time domain equals multiplication in frequency domain, the multiplication in frequency domain approach can be formulated

$$Y(k) = X(k) \cdot H(k). \tag{3.3}$$

Here $Y(k)$, $X(k)$, and $H(k)$ denote the discrete-time Fourier transform (DTFT) of the series $y(n)$, $x(n)$, and $h(n)$. If one wishes to retrieve the filtered series $y(n)$, the inverse transformation can be applied to the output $Y(k)$.

It is thus possible to obtain the filter output $y(n)$ without using the convolution in equation (3.1). This is done by first converting the time series into the frequency domain, multiplying it with the filter frequency function as in (3.3), and lastly inverse transform it back into time-domain.

Filtering methods

Fast convolution — a frequency domain approach

The frequency multiplication approach might seem like a detour, first the signal of interest needs to be transformed to frequency domain, then multiplied with the filter, and then inverse transformed back into sample domain. This technique was known since the time of Fourier but was mostly ignored since it took longer to compute than using the standard time convolution approach. However, this changed

1965 when the Fast Fourier Transform (FFT) algorithm was developed [Smith, 1997].

By swapping the DTFT with the FFT the detour presented by the frequency multiplication approach can in some cases yield faster computational time than the direction convolutions approach, at least for longer filters (>40–80 samples depending on hardware) and input sequences [Lyons, n.d.]. The method involving frequency multiplication with the FFT is referred to as the FFT convolution, or Fast convolution.

An implementation of this method can e.g. be found in the SciPy package, namely `scipy.signal.fftconvolve`. According to the documentation this implementation is generally faster for longer arrays (>500 samples) [Virtanen et al., 2020].

Overlap-add method — a time based approach

The overlap-add (OA) method is a method that allows chunk-wise filtering of long sequences of input data. This can be attractive in systems with insufficient memory to store the entire input sequence, or in real-time applications where the filtering must occur on-line as data is collected [Smith, 1997].

The direct convolution OA method builds on the direct convolution approach. The data is first segmented and zeros are right-padded to the segment. The segments are then convoluted with the filter impulse response and lastly added together. Note that the padding of zeroes results in the filter output being somewhat longer than the initial segment. When the output segments are added to yield the unsegmented output there are thus some overlap between the filtered segments. The result after summing the segments are identical to the direct convolution approach without the OA [Smith, 1997].

An implementation of the fast convolution OA method can e.g. be found in the SciPy package, see `scipy.signal.oaconvolve` [Virtanen et al., 2020].

There are other methods, such as overlap-save, that builds on OA. However, they are not included here.

Filter properties

Causality

Causal filters are filters where the output depends on current and previous information only. In (3.1) this would mean that $h(n) = 0$ when $n < 0$.

Non-causal filters use future samples to calculate the current value. In a strict on-line context this is not possible.

One should note that, because the non-causal filters use future values, some future information can "leak" backwards in time. When e.g. studying onset in EEG signals non-causal filters can potentially produce results that make the onset period appear shorter. This leakage can potentially lead to a systematic underestimation of signal onset, or even erroneous interpretation of pre-stimuli phase of the filtered signal [Widmann et al., 2014].

IIR and FIR filters

Infinite impulse response (IIR) filters are filters with internal feedback, i.e. where the filter output at one sample depends on the filter output at another sample. As one can imagine this feedback can result in instabilities. In contrast, finite impulse response (FIR) filter outputs depend only on the filter input. An example of FIR filter is the moving average filter which calculates the average of a fixed number of previous samples and outputs the result. See any introductory digital signal processing book, e.g. [Proakis and Manolakis, 2006].

Though IIR filters can be designed to have approximately linear phase they never truly posses it. FIR filters on the other hand possess linear phase if the corresponding impulse response, $h(n)$, is symmetric. A linear-phase filter is a filter with a phase-delay that changes linearly with frequency. One can show that if frequency components are phase-delayed linearly with frequency, then all frequency components are delayed equally in time. A filter with the linear-phase property does thus not change *where* in time the frequency components present in the signal relative to each other (because all frequency components are changed, or delayed, equally).

Digital filtering and EEG

Non-linear phase filters

Filter with non-linear phase distorts the temporal shape of the input signal—e.g broadband signals such as ERP or signals with complex spectrum—even if its spectrum is in the filters pass-band [Widmann et al., 2014]. Depending on the application this can have more or less impact.

Cutoff frequencies and frequency content

Generally, the EEG-signals are filtered with a band-pass filter with cutoff frequencies around 1–40 Hz during the preprocessing step. In some cases wider pass-bands are used but special care has to be taken to suppress mains frequency (50 or 60 Hz depending on location). In EEG-preprocessing band-stop filters centered around 50/60 Hz are almost exclusively used [Widmann et al., 2014].

If eye movement related EEG-signals are of interest the cutoff-frequencies must be adjusted accordingly. Previous studies have shown that to retain 95 % of the spectral power of the eye-blink related signals frequencies up to 54 Hz must be considered. Similarly, frequencies up to 13 Hz must be included for saccades [Noureddin et al., 2007]. This suggests that eye-blink signals contains a wide span of frequencies compared to saccade signals.

It has been shown that it is possible to detect eye-blinks by simply applying a threshold check on the low-pass filtered EEG-signal using a cutoff frequency of 10 Hz. It is also shown that corneo-retinal dipole rotations give rise to signals with frequency content centered at around 10 Hz. Moreover, the spectral power of eye-blinks seems to be contained in frequencies mostly below 10 Hz [Keren et al., 2010].

4

Implementing a Simple Asynchronous Pipeline Framework (SAPF)

This chapter describes the implementation of the library that will later be used to build an example BCI system. Most of the library consists of *nodes*—i.e. blocks of functionality—that can be puzzled together to create data pipelines acting as a BCI system. The library will be referred to as a Simple Asynchronous Pipeline Framework (SAPF) to simplify references from hereon.

Most of the node implementations described in this section will have general and configurable behaviours, and the configurations suitable for the specific BCI system presented in this thesis will thus be presented later in Chapter 6.

Section 4.1 describes existing BCI platforms and libraries and some parallels are drawn, Section 4.2 presents definitions that will be used throughout the chapter, and, finally, Section 4.3 through Section 4.10 present the implementations of the SAPF and its SAPF nodes. Some of the sections will contain a small discussion at the end to highlight some of the findings and results. Moreover, at the end of the chapter, in Section 4.11, a final discussion regarding the SAPF as whole will be held. For more information see the source code[1].

4.1 Existing BCI platforms

Tools for BCI (TOBI) Common implementation platform (CIP)

Though not a BCI system itself, the TOBI system specifies the communication between the BCI "blocks" in the model largely based on Mason and Birch's formalizations, see figure 4.1. This allows different BCI platforms that obey the CIP to communicate. For example, a newly create a data acquisition module that follows

[1] `https://gitlab.control.lth.se/users/sign_in`

the CIP standard could easily be integrated in an already existing BCI system. This is achieved by using the standardized interfaces, namely the TOBI interface A–D.

The TOBI interface A (TiA) specifies the communication between the data acquisition module (e.g. data acquisition server) and the feature extractor. This interface is built on XML over socket (either UDP or TCP is available).

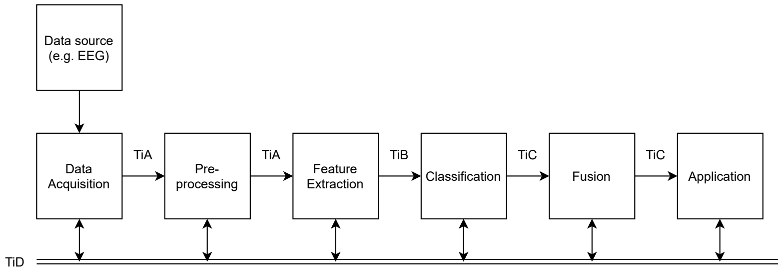


Figure 4.1: The TOBI CIP pipeline decomposition.

The TOBI interface B (TiB) is used to transmit features between the the feature extractor and the classifier. The TOBI interface C (TiC) specifies communication between the classifier and the application. Lastly, the TOBI interface D (TiD) is used to transmit event markers over XML in between the BCI blocks in a bus-like fashion.

BCI2000

BCI2000 is an open-source self-contained BCI software implemented in C++. The BCI pipeline is built on four modules; source, signal processing, user application, and operator interface, which communicate over a protocol that runs over TCP/IP. As of 2018 more than 1000 articles have been published supported by BCI2000, making it the most used BCI platform [Peter Brunner, 2018].

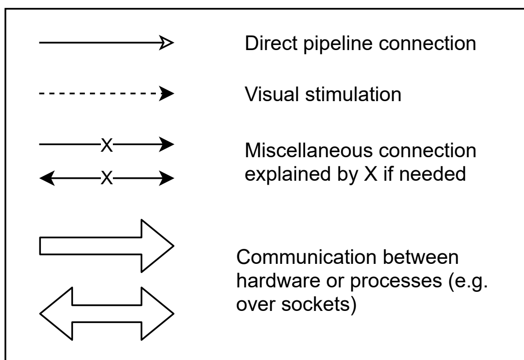
OpenVibe

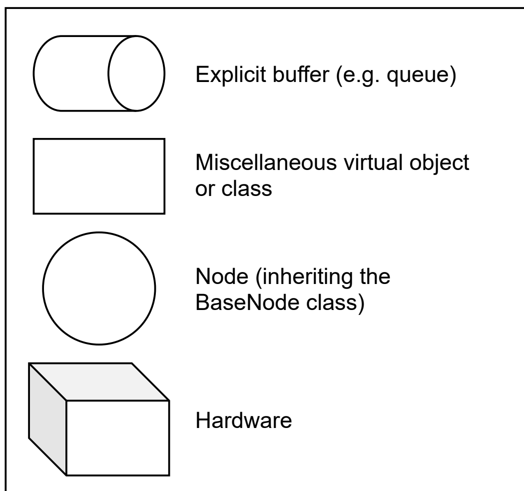
Similar to BCI2000, OpenVibe is a open-source self-contained BCI software implemented in C++. Its drag-and drop based graphical user-interface makes it easy to use for a wide array of users [Peter Brunner, 2018].

OpenVibe's architecture is based around a kernal together with what is referred to as boxes. The boxes act as a way of decomposing the functionality into discrete units with an input and output that are connected to other boxes in the pipeline. Each box runs what is referred to as a "box algorithm" which can be run either on an external trigger, a clock tick, or when data arrives at its input [OpenVibe, 2017].

4.2 Definitions and figure interpretations

To simplify the explanation of the concepts in this chapter many images are used. Much of the images contain information about the connection, and flow of data, and proper definitions of the geometries used are thus needed:





4.3 The SAPF implementation

In the beginning of the thesis the TOBI CIP implementation was considered a good candidate as a foundation for the BCI pipeline. This was mainly because it standardizes the communication between the pipeline blocks and thus offers greater flexibility than other home-made implementations. However, the TOBI comes with a couple of problems that made other solutions more attractive.

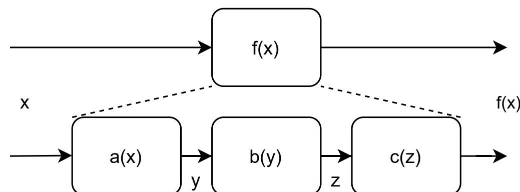


Figure 4.2: The concept of decomposition of functionality. The pipeline itself describes the overall mapping pipeline-mapping, which can be decomposed into easily interchangeable sub-functionalities.

The TOBI CIP specifies four different interfaces (the TiA–TiD), but only two of them seem to be implemented as of when this thesis was written (TiC and TiD). Also, no binaries[2] are available and must therefore be built from the source code, written in C++, which could be problematic for the non-programmer user. This also makes it harder to do any modifications to implementations, as most of the end-users are more familiar with Python than C++.

Therefore a simple pipeline library was implemented instead without the structural restrictions of TOBI CIP. The library contains mainly a pipeline framework which will be further presented in this section. The concept and implementation will be referred to as Simple Asynchronous Pipeline Framework (SAPF).

An asynchronous node-based pipeline

A pipeline can in some sense be seen as a series of transformations of the input data. The idea of decomposing the transformations into "discrete blocks" increases the code reusability, flexibility, and makes unit-testing easier as the functionality can be run on its own, see Figure 4.2. Taking a step further, a pipeline can be seen as a directed graph of these discrete blocks—from now on referred to as *nodes*—each performing some functionality on the data that flows within the graph. Some nodes generate data, and thus acts as an input to the pipeline graph, and some nodes might absorb the data and send it to any external source thus acting as an output out of the graph. Note that the functionality of a node is defined by the way it processes the inflow of data and outputs it to the connected nodes.

In the implementation, each node—implemented as a node object using object oriented programming—runs in its own thread and is connected to other nodes with thread-safe[3] `Queue.queue` objects acting as buffers. The nodes perform some action, defined by its `action(x)` method, on any data `x` that appears on its input

[2] Compiled code, in contrast to source code that is not itself executable.

[3] Thread-safe means that the only one thread can access the data at the time. This prevents race-conditions and other unpredictable behaviour.

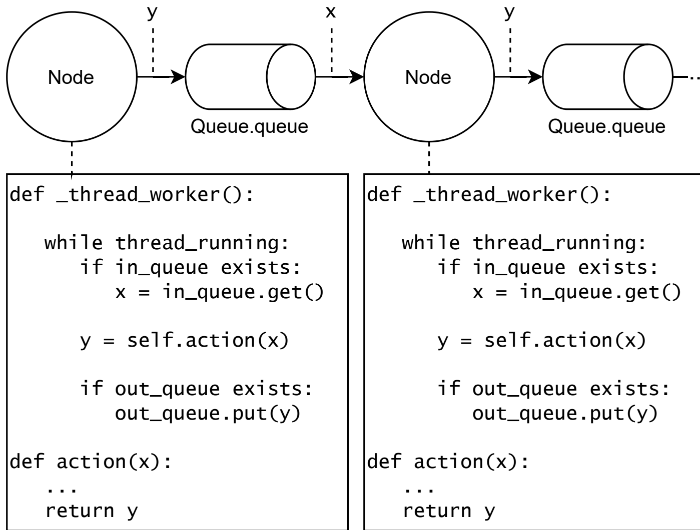


Figure 4.3: Overview of how node objects perform transformation of the data, and how the processed data is outputted to the next node via shared buffers. The `_thread_worker(.)` is run by the node thread. The node thread waits for items to appear in the input buffer. Incoming items are fetched and transformed using the node's `action(.)` method, after which the item is pushed to the output queue.

queue. The result of the `action(x)` is placed on the node's output queue for other nodes to use, see Figure 4.3.

At first, attempts where made to implement nodes that were capable of supporting directed graphs. However, due to time limitations, and the fact that the cyclic pipeline graphs (which is a subset of directed graphs) brings many problems such as instability issues, the graph was later limited to be only of the linear type, as seen in Figure 4.3. The pipelines presented in this thesis are thus nothing more than a series of decomposed functionalities.

The asynchronicity comes from the fact that the nodes' transformations are not necessarily performed in a synchronized fashion, as suggested by Figure 4.3. Objects placed in any queue might, or might not, be picked up by subsequent nodes directly. As one might expect this can result in a build-up of data in the pipeline if any node outputs data faster than subsequent nodes can process. However, throughout this thesis it is assumed that the processing delay of any node is short enough, and the pipeline graphs are small enough, so that only one piece of data is present in the pipeline at once:

ASSUMPTION 1 Zero, or one, chunk (pieces of data) is present in the pipeline graph at once. □

Data flowing in the pipeline will sometimes be referred to as a chunk. Throughout the thesis only EEG time series data will be considered, and this data contains not only one sample, but a time interval of EEG recording. This, accompanied by the fact that multiple channels of EEG data are contained in the time series, are hopefully highlighted by the word chunk.

A bit of implementational details; one should note that—according to the pseudocode in figure 4.3—nodes could potentially have no input or output buffers. These nodes are used as end or start nodes of the pipeline. The start nodes can generate data inside the `action(.)` method, and output it to the subsequent node. *Generate* here typically means loading data from a file, or a so called LSL stream, as will be discussed further later.

Moreover, the threads in Figure 4.3 can easily be changed to processes—and the buffers to e.g. processes-safe `multiprocessing.queue` buffers—if needed. Attempts were made to implement process-based nodes, but this implementation was later abandoned as it was not needed.

Another assumption that will be made, if not explicitly stated otherwise, the processing delay of the pipeline will be assumed to be zero.

ASSUMPTION 2 Placing and fetching operations of chunks from and to the *intermediate* queues, as well as transforming the chunks using the nodes `action(.)` method, are assumed to be instantaneous. Any time delays introduced by the OS in which the pipeline is run, and any time delays introduced by e.g. context switching, are assumed to be negligible. In other words, any data generated by any node is assumed to be present and readily available inside the output node instantaneously. □

This assumption is made as any potential real-time considerations and optimizations are regarded as out of scope of this thesis. The assumption might, or might not, be realistic depending on hardware, `action(.)` implementation as well as framework implementations, and which size of chunk is used in the pipeline.

The implemented SAPF nodes

A hand-full of nodes and their corresponding general functionality were implemented following approximately the decomposition of functionality that TOBI CIP presented (Figure 4.1). These nodes will be referred to the SAPF nodes as they are a part of the SAPF framework, and are meant to give the user the standard functionalities that are needed to construct any basic BCI pipeline. An overview of these nodes, and their general behaviour, is shown Table 4.1.

Table 4.1: Overview of SAPF nodes and their behaviour.

LSL node	Reasonably acting as the first node in the pipeline, the LSL node handles the connection to EEG hardware, and the acquisition of the user EEG. The node buffers the incoming LSL samples from the LSL stream and generates a chunk which is outputted to any subsequent node. The length of the buffer defines the length of the outputted chunks, and the length—in samples if not explicitly stated—will be denoted L_c.
Pre-processing node	A node that performs the pre-processing of the EEG data. This includes chunk-wise digital filtering using a `Filter` object, and any potential baseline correction of the acquired EEG data.
Processing node	An optional node that performs e.g. chunk validity check, as described later.
Feature extractor node	An optional node that is dedicated to extracting features of interest from the inputted data using a `FeatureExtractor` object.
Classification node	A node that performs the classification of the inputted time series, features, or data in general, using a `Classifier` object.
Application interface node	A node dedicated to communication between an application and the pipeline, and thus reasonably acts as an end node of the graph. The communication e.g. includes commands for changing the state of the application, which can be used to stimulate a user engaged in the application.
Stimuli node	A node that attempts to generate a labeled data set by sending stimuli commands to the application interface node. The application interface node performs the stimulation and the stimuli node saves the incoming chunks and labels them accordingly.

In the implementation all these nodes extends the `BaseNode` class. The abstract class `BaseNode` itself contains the fundamental behaviour of a node as presented in Figure 4.3. If any future projects finds that a node is missing a new node extending the `BaseNode` class can simply be introduced.

The SAPF pipeline graph – a composition of SAPF nodes

The class `PipelineGraph` was introduced to simplify the act of connecting the SAPF nodes to a graph. As previously stated, only linear graphs were allowed. On

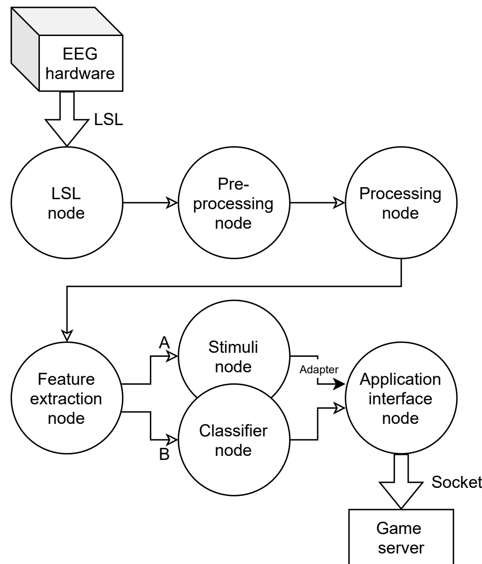


Figure 4.4: Two example pipelines. If the connect A is made the pipeline works as an acquisition pipeline, and if the B connection is made the pipeline works as a classification pipeline. In this example the application interface node is controlling a game via a socket. The decomposition of functionality is comparable with the TOBI CIP in Figure 4.1 or Mason and Birch's model in Figure 3.1.

start the `PipelineGraph` simply iterates through the supplied nodes and generates the shared buffer queues as shown in Figure 4.3. The act of sharing a queue is what corresponds to a connection in the graph. When the pipeline graph is started the graph starts each of the node's listening thread. An example pipeline is shown in Figure 4.4.

Due to time limitations, the framework cannot produce one pipeline that is able to *both* perform calibration (i.e. creating a labeled data set which is the purpose of the stimuli node), and perform classification. Instead, as Figure 4.4 suggests, *two* pipelines have to be constructed—one for collecting the data, and one for performing the actual classification. This is of course a disadvantage as the operator of the system would have to manually collect the data, train the classifier, and then run the pipeline for real-time user EEG classification.

In the following sections the different nodes, and their general behaviour, will be discussed.

4.4 The LSL node

The LSL node reasonably acts as a start node in the pipeline. The node operates—as all other nodes—in its own thread where it attempts to pull samples off from the LSL host's stream and send it—as chunks of data—to any connected node as NumPy arrays of shape:

`(No. epochs, No. samples, No. channels 1, No. channels 2)`

The number of samples in a chunk will be denoted as L_c. The reason why two dimensions are dedicated to holding the channels are mainly of historical origins and will be discussed further later. This shape will simply be referred to as the standard shape and everything referred to as a chunk is of this type and shape.

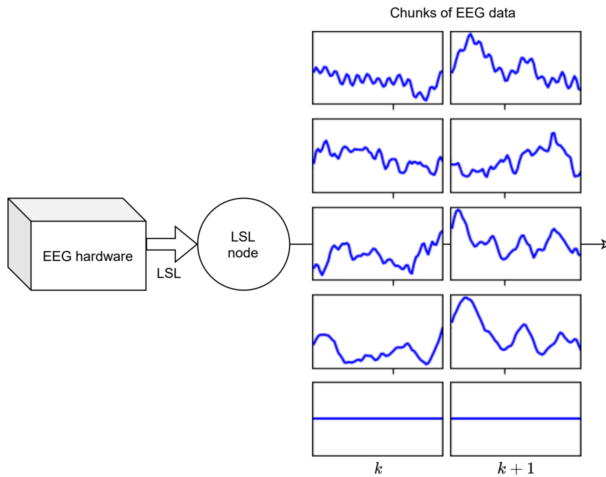


Figure 4.5: Overview of the functionality of the LSL node. The node collects samples from the LSL stream and produces chunks. Two chunks are displayed in the image with 5 channels of EEG.

It should be noted that longer chunk, i.e. bigger L_c, has the potential to contain more of the signal, but also increases the pipeline delay as the LSL node's buffer needs to be filled before a whole chunk can be constructed and sent to the next node. If the time delay produced by the buffering mechanics, τ_{LSL}, are the only delay considered—i.e. if Assumption 2 holds—a minimal expected time delay can be formulated. If a sample enters the buffer, and the sample is the last sample before a chunk can be constructed, the time delay experienced by this sample is 0. Moreover, if the sample is the first sample to enter the buffer before a chunk is constructed, then it must wait until the chunk is constructed and passed to the next node. The longest and shortest time delay experienced by any sample due to the buffering mechanics

are thus

$$\max(\tau_{LSL}) = \frac{L_c}{f_s}, \quad \min(\tau_{LSL}) = 0. \tag{4.1}$$

Assuming that τ_{LSL} is uniformly distributed in this interval one can then formulate the guaranteed minimal expected time delay, τ_{sys}, of the entire system

$$\mathrm{E}\{\tau_{sys}\} \geq \mathrm{E}\{\tau_{LSL}\} = \frac{L_c}{2f_s} \tag{4.2}$$

where L_c is the length of the outputted chunks, and f_s is the frequency of arriving samples, which is also the sampling frequency of the hardware. τ_{sys} is the delay experienced by the user of the system. Due to the assumptions made, and the fact that the OS that the BCI system is run on is not strictly speaking suitable for real-time systems (typically Windows or Linux), it is hard or impossible to guarantee an upper bound.

The lower bound presented by (4.1) and (4.2) will act as a theoretical limit of what can be achieved by a BCI system utilizing the LSL node in terms of speed. A more thoroughly motivated explanation of the equations is presented in the appendix.

4.5 The pre-processing node

The pre-processing node is dedicated to digital signal processing of any time series presented on the input. The node is also capable of dropping specified channels and applying baseline correction. The filtering was implemented using the previously presented concepts of chunk-wise filtering.

Methodology, dummy filter and dummy data

The pre-processing node was developed through an iterative process of testing different methods and evaluating the results. During the testing a moving average filter of length 11 acting as a dummy low-pass filter was used if not explicitly stated otherwise. During the final test a dummy band-pass filter was used with cutoff frequencies 40 Hz and 2 Hz. The dummy signal used throughout the experimentation was a 1 second long EEG snippet split into 4 chunks of equal length. A saccade is contained in the chunk. An online setting was mimicked by giving the pre-processor node one chunk at the time.

It should be noted that the filter and the signal are just dummy objects for developing the node, and during later usage in the BCI system any linear filter can be used.

Chunk-wise filtering

The first couple of filter output samples is affected by the initial conditions of the digital filter, this is demonstrated in Figure 4.7a below. The uninitialized filter (UF)

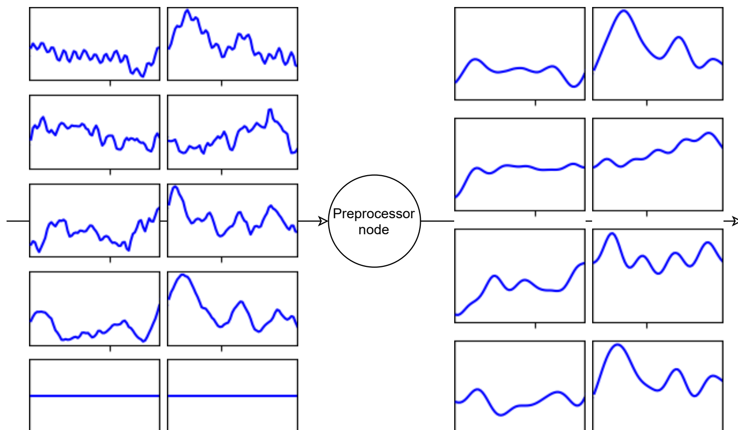
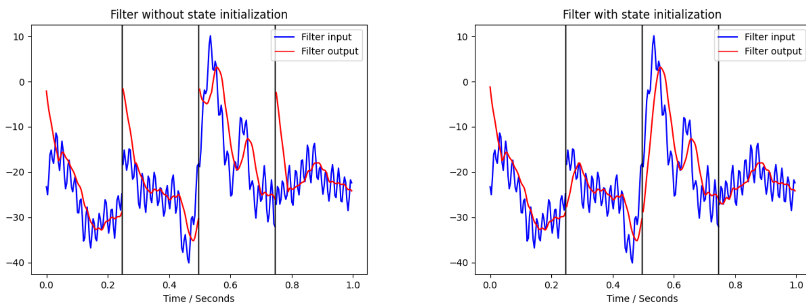


Figure 4.6: Overview of the pre-processor node functionality. In the example the last channel was dropped by the pre-processor node as it suggestively contains no information (which channels to drop are manually specified).

produces output with transients heavily affected by the initial states of zeros. This distorts the signal which is not desired.



(a) Chunk-wise filtering using UF.

(b) Chunk-wise filtering using IF.

Figure 4.7: Chunk wise filtering. The vertical black lines indicate the start of a new chunk.

To remove the transient effects one could e.g. use the OA method. At first SciPy's OA-method, such as `scipy.signal.oaconvolve`, were considered. However, this method seemed to be suitable for offline filtering only. Instead SciPy's `scipy.signal.lfilter` method was used as this allows the specification of initial states. `scipy.signal.lfilter` seems to be implemented using the convolution in

time domain approach however.

A demonstration using what will be referred to as re-initializing filter (IF) can be seen in Figure 4.7b. The output is produced using Scipy's `scipy.signal.lfilter` where the previous filter state is reused:

```
Function filter(x)
    global z, b, a ;                    // Filter state and parameters
    y, z = scipy.signal.lfilter(b, a, x, initial_state=z);
    return y;
```

One should note that the first chunk is still affected by the transient effect as no previous data is present at this point. The filter states can at this point be guessed or simply set to zero. The latter alternative was used.

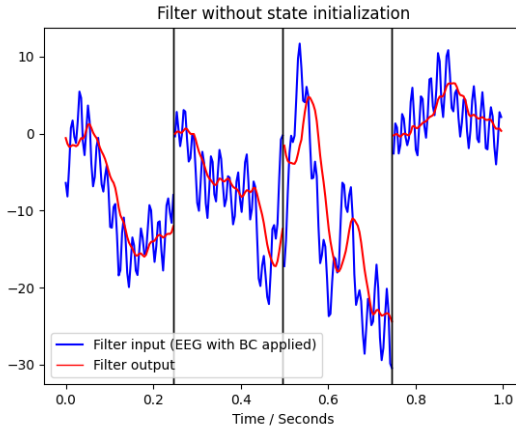
As can be seen in both plots in Figure 4.7 the signal seems to contain a constant negative component. If this component changes slowly over time the signal is said to be subjected by drift (i.e. it contains low frequency content that does not contain useful information). Different methods were considered to remove this signal component.

Removing drift

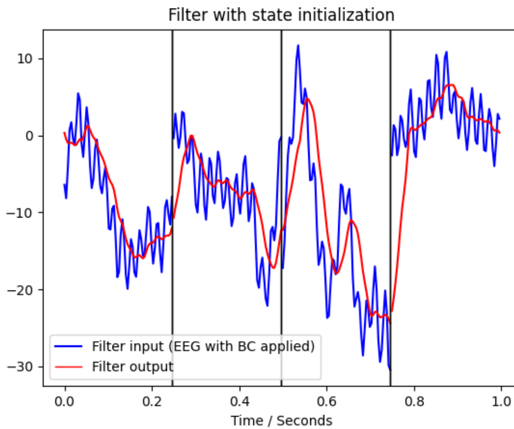
When performing offline analysis of EEG data, baseline correction (BC) can be used as a way to normalize the chunks by removing the baseline. This is one method to counter drift. Therefore, naturally, the first approach was to use the same method in the online setting. Three methods were tested: BC then filtering, filtering then BC, and removal of drift using simple band-pass filtering (the high-pass component baked into the band-pass filter is the interesting part).

BC then filtering

BC was first applied to the sample data after which the data was filtered using the dummy filter. The results—using either IF or UF—are shown in Figure 4.8.



(a) BC then UF.

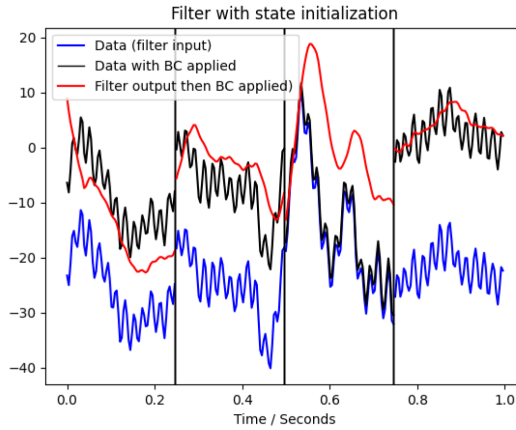


(b) BC then IF.

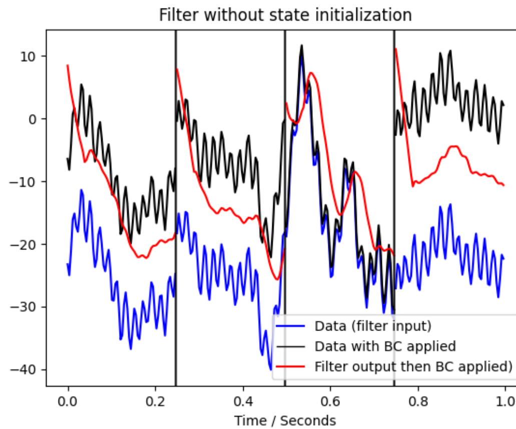
Figure 4.8: BC first applied after which filtering using UF or IF was performed.

Filtering then BC

Changing the order so that filtering—again, both UF and IF—was performed before BC was applied yielded results presented in Figure 4.9.



(a) Filtering using IF then BC.



(b) Filtering using UF then BC.

Figure 4.9: Filtering first performed after which BC was applied. Blue indicates the dummy sample and red indicates first filter then BC node output.

Band-pass filtering

The dummy low-pass filter was replaced by a FIR band-pass filter. No baseline correction was used. The filter was applied using the IF and the UF method. The UF filtering result is not included here as it yields results similar to the BC then UF approach seen in Figure 4.8a.

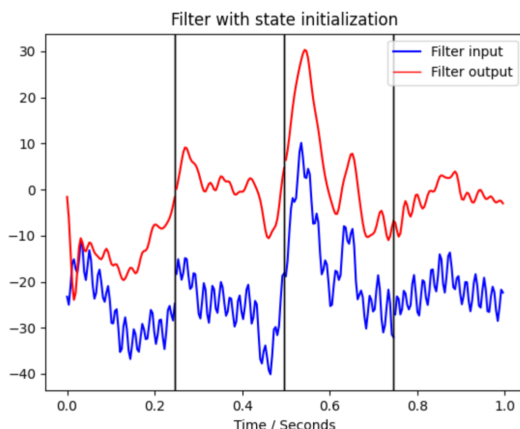


Figure 4.10: Band-pass filtering using IF.

Discussion regarding the pre-processor node

Chunk-wise filtering

It is unclear what type of filtering method is used by `scipy.signal.lfilter`. Attempts were made to understand the source code, but due to the strict time schedule, and the low demand of high-speed filtering, this was deemed as a too high hanging fruit and no further investigation was done.

Removing drift

Interestingly, during "BC then filtering" the UF performed seemingly better than IF. Given a second though, this is not strange at all. The BC introduces discontinuities larger than those already present in the EEG data. The IF filter re-initializes its states by using the previous states from the last chunk, and will therefore see these discontinuities as large steps in the filter input. In contrast, the UF filter keeps none of the previous states and therefore "starts" from zero, right in the vicinity of EEG data which too "starts" near zero (due to BC).

Less interesting is the results from using filtering then BC. The results appear less obvious, and almost chaotic. However, there is a reasonable explanation; the time lag imposed by the filter—which shifts the signal to the right in the figures—basically changes the interval which BC is applied on, and thus result in an output (red) that does not even slightly follow the input (blue).

During band-pass filtering the results are as expected, the low frequency component(s) in the signal is removed after the first chunk is filtered and the IF is properly initialized. Depending on the choice of filter this period might be longer.

The results suggest that BC is not recommended in an online setting, and that other means—such as high-pass or band-pass filtering—should be used.

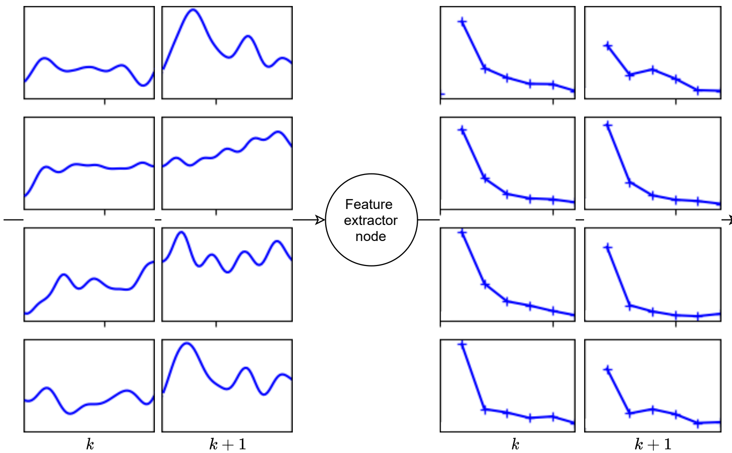


Figure 4.11: Overview of the functionality of the feature extractor node. The example is showing features based on frequency band energies for different bands. This will be further discussed in section 6.6

4.6 The processing node

The processing node was mainly added to separate any use-case specific processing and the digital filter—which is generally present in every BCI pipeline. The processing node can be used for e.g. independent component analysis (ICA) or principle component analysis (PCA) transformations or simple chunk rejection (e.g. rejecting chunks with too high variance). One should note that there are nothing that corresponds to the processing node in Masons and Birch's model, or in the TOBI CIP pipeline. Most likely ICA and PCA are considered a type of feature extraction in those models.

4.7 The feature extractor node

The feature node contains a `FeatureExtractor` object which performs channel-wise feature extraction. The `FeatureExtractor` acts as a composition of vector-valued callback functions that extract one, or multiple, features from one channel at the time. What features to use depends on the specific use-case.

One could argue that the feature extraction could be implemented in the classifier node. This is possible and can be easily done. However, if one wishes to build a data set—using the stimuli node—of features and label pairs using a separate node, performing feature extraction is more reasonable as the extraction would only be implemented once.

4.8 The classifier node

The classification node is dedicated to hold any classification method that is capable of translating user EEG or features to a command class. The class (typically an integer) is sent to the subsequent node, reasonably the application interface node. In the bare framework implementation no classifier is selected by default.

4.9 The application interface node

The application interface node was dedicated to connect the pipeline and an application. The node was supposed to act as a general end-node in the pipeline and handle communication to any application. Much of the application specific implementations—how the application is connected, and how exactly the pipeline to application communication occurs—were implemented in a separate class, the GameClient class, as will be further discussed in Chapter 5. This class acts as a sort of adapter and could easily be swapped with little change in the code, thus allowing other applications to be connected.

The general behaviour of the application interface node is that it takes a user command (extracted from the EEG by e.g. the classification node) and performs whatever is associated with said command.

4.10 The stimuli node

Implementation of the stimuli node

The stimuli node is used to generate stimuli and save the incoming responses to create a labeled data set. The node is not dependent on the choice of stimulation type—tactile, visual, etc—as will be seen later.

The implementation is based on the "black-box" idea presented in Figure 4.12. The goal was to make the stimuli node independent of any other node in the pipeline graph. In this concept, the node produces a signal—a stimulation command—that is sent forward in the pipeline. When the command reaches the appropriate node, i.e. the application interface node, that node performs what is necessary to produce the stimulation described by the stimulation command.

One should note that the node does not perform classification and is thus not capable of translating user responses to an actual user command to control the connected application. During calibration—i.e. when the stimuli node is used—no data set exists and no classification can be done. However, as will be seen later, the stimuli node can "simulate" classification by applying whatever is associated with the expected user response given a stimulation command.

Stimulation-response mapping – the concept

The stimuli node attempts to explore the mapping from *stimulation command*[4] to *response*[5], as seen in Figure 4.12. It does this by employing a strategy to stimulate the user. The resulting *response* chunk(s) is then collected by the stimuli node, and associated with the type of stimulation used, to create a labeled data set.

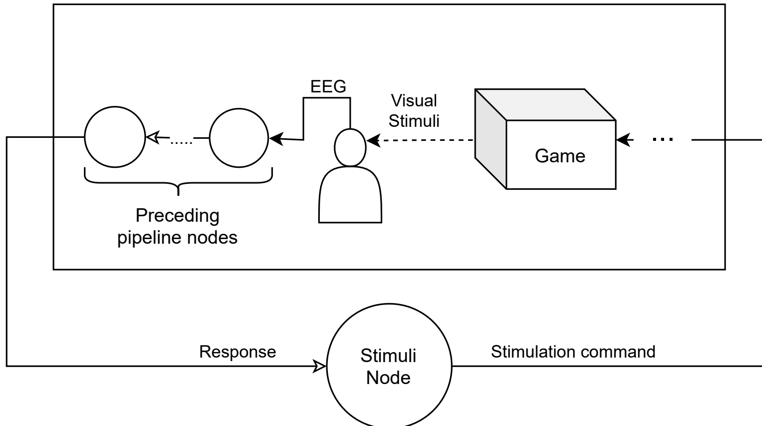


Figure 4.12: Simplified diagram of the stimuli node in a pipeline. The implementation builds around the idea that the stimuli node should select and output something—e.g. a stimulation command—and listens for the preceding node's response. In the example above, a game displays visual stimulation to a user. The loop is closed as the user's EEG is affected by the visual stimulation.

Stimulation-response mapping – the implementation

In contrast to the conceptual idea, shown in Figure 4.12, the implementation deviates slightly. This is mainly due to implementational restrictions and time limitations. The node was restricted to communication with application nodes only, and the communication was through a so called *stimuli adapter*. The generation of a stimulation—i.e., algorithmically, what should be done to produce said stimulation—is highly use-case specific. The adapter handles this and therefore encapsulates the use-case specific behaviour from the general mapping behaviour

[4] A stimulation command is a command that informs the application what stimulation should be run. In the case with visual stimulation the command can e.g. change the visual appearance of the application.

[5] A response is, in general, a chunk (containing any type of data) that appears on the input of the stimuli node directly after the stimulation command was sent. The chunk could e.g. contain pre-processed EEG data, or features.

which remained in the stimuli node. The stimuli adapter basically interfaces the stimuli node and the application interface node, see Figure 4.13.

The adapter converts the abstract stimulation commands provided by the stimuli node to actual application and or game commands that are needed to produce the desired user response. The stimuli node and adapter communication and behaviours are presented in Figure 4.14.

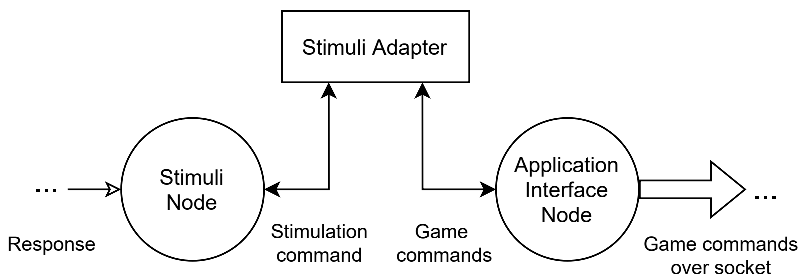


Figure 4.13: Diagram of the interface between the stimuli node and the application interface node. The stimuli adapter acts, in a sense, as a mediator. The stimuli node behaviour does not depend on the application or application interface node implementation—what exactly a stimulation involves is abstracted away by the stimuli adapter.

The stimuli node retrieves a set of possible stimuli from the adapter and picks one[6]. The stimuli node then notifies the adapter to perform the selected stimulation. The adapter sends the appropriate commands to the application interface node to change the application state in such a way that the desired user response can be expected.

When no stimulation command is sent the incoming EEG chunks are stored and labeled as containing "nothing". Note the user might still perform actions during these intervals, and the incoming chunks might erroneously be labeled as "nothing". The node-adapter communication is presented in Figure 4.14.

[6] Randomly in this case. It can be desirable to have a data set where the number of data points per class is approximately equal.

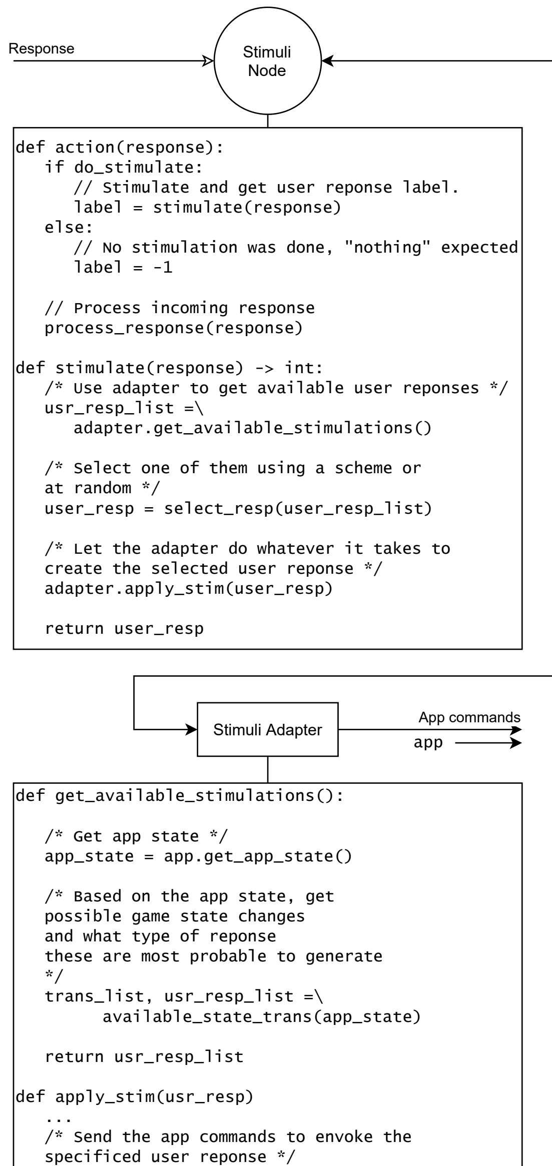


Figure 4.14: Pseudo-code describing the communication between the node and adapter. The figure shows how stimulation is selected by the stimuli node, how the adapter performs the stimulation, and lastly that the stimuli node processes the incoming chunks and labels them accordingly. The `process_response(.)` will be further discussed later.

As the reader hopefully has discovered the stimuli node-adapter duo—simply referred to as NA from hereon—acts as a tool for calibration. As mentioned, the NA is not capable of letting the user actually control the application. Fortunately, the NA implementation offers a surprisingly simple way of introducing basic gamification concepts which almost changes this. However, before these are discussed a quick summary of what exactly NA does is suitable.

1. NA investigates the application state and finds potential user evoked state transitions—what virtual button can be pressed, in which direction can a player move, etc.
2. These transitions are collected and paired with what the user has to do to evoke these transitions—typically an integer representing a user command.
3. The NA now knows what stimulation to use to evoke a user response of a specific class and can therefore select one stimulation and perform it—i.e. what should be done to make the user press that button, or move the player in a certain direction.
4. The NA saves the incoming chunks and labels them using the known user response class.

As mentioned, a stimulation here means changing the application state. In the case of visual stimulation it could be to move something on the screen. Instead of just letting the NA stimulate only, the NA can follow through and perform not only the stimulation, but also what the user might had done if exposed to the stimulation. If the user is assumed to be a fully engaged in the application—and fully susceptible or benevolent in the sense that the user's response always coincides with the expected user response—then if the NA follows through and performs the state transitions that the user would do, it would appear as if the user controlled the application herself.

An example could be if the NA puts an attractive item next to a user controlled player in a game, then it is reasonable to assume that the user would attempt to collect the item. The NA—which is incapable of actually classify the user EEG—could then simply move the player to collect the item.

The processing of chunks

This section will highlight different implementations of the `process_response(.)` method. But first some terminology will be presented.

A more abstract view of the loop previously presented (Figure 4.12) can be seen in Figure 4.15. NA denotes the stimuli node and stimuli adapter combination, and S the remaining BCI system components in the loop. The incoming response chunk is denoted X, and its true class is denoted Y. $C_{\hat{Y}}$ denotes the stimulation command selected by the stimulation node with expected user response class label $\hat{Y}$. The index k denotes the chunk index.

$C_{\hat{Y}}(k)$ is produced right after $X(k-1)$ is processed by NA, i.e. the stimulation commands are sent synchronized with chunks appearing on the input of NA. The EEG response contained in the chunk $X(k)$ thus appears to be the output of the system S when the input $C_{\hat{Y}}(k)$ is inputted (with a delay described by the equations 4.1 and 4.2 if the LSL node is present in S).

If only one chunk is presented in the pipeline S at once—i.e. Assumption 1 holds—then the first chunk read after stimulation onset, i.e. $X(k)$, will contain EEG data generated after the stimulation $C_{\hat{Y}}(k)$ was presented to the user. If the assumption does not hold modifications have to be made to the implementation, which might be more or less complicated due to the asynchronous nature of the pipeline.

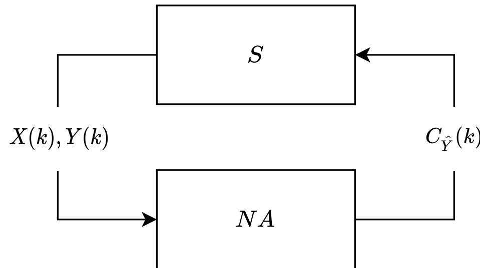


Figure 4.15: Abstract view of the NA in the BCI loop.

The predominant problems of this approach is that there are generally a user response delay, τ_{user}, before the user responds to the stimulation. Using the terminology presented in Figure 4.15, if τ_{user} is longer than the chunk length L_c in seconds, then $Y(k) \neq \hat{Y}(k)$. Maybe the user response instead is present in the second chunk, then $Y(k+1) = \hat{Y}(k)$. However, in general it is unknown in which chunk the user response resides, and frankly, if the user even responded at all.

Handling user response time delay

The following algorithms attempts to solve the issue of finding the user response in the stream of incoming chunks. This is done by implementing the `process_response(.)` presented before in Figure 4.14 in different ways. These algorithms assume that the user responds within some fixed time and the BCI system, at least during this calibration phase, is regarded as synchronous.

Assuming small user response delay The seemingly naive approach is to assume no, or a relatively short, user response delay. "Short" here means short enough so that the entire user response is contained in the first chunk of EEG data, i.e. $Y(k) = \hat{Y}(k)$. If the user response is contained in the first chunk then the mapping is trivial:

Algorithm 1: Pseudo-code of trivial mapping algorithm assuming short τ_{user}. The code is run in the NA, more precisely the stimuli node as Figure 4.14 suggests.

```
Function save(X, Y)
    ... /* Saves the chunk X with label Y to file.                */
Function read_chunk()
    ... /* Blocking read. Wait for a chunk to appear on
           input buffer queue of NA and read it.                  */
Function process_response(Yhatk)
    Xk = read_chunk();
    save(Xk, Yhatk);
```

Assuming constant user response delay A more realistic approach would be to assume that τ_{user} is longer than the chunk length, and thus not contained in the first chunk. To simplify, it is here assumed that the user response is approximately constant and known. Because it is known one can simply wait and discard the corresponding number of chunks before saving and storing the chunk containing the response.

However, the user response might be located in two chunks simultaneously[7]. To solve this, multiple chunks were stored and merged into one larger chunk. In this larger chunk, a new chunk was extracted centered around the user response (i.e. the sample corresponding to τ_{user} measured from the start of the larger chunk).

[7] If the user delay τ_{user} is approximately a multiple of the chunk length L_c, that way the onset will be just at the start, or end, of one chunk, and the response might stretch over two chunks.

Algorithm 2: Pseudo-code, run in NA, describing the process of collecting and merging chunks after which a new chunk is extracted centered (hopefully) around the user response.

```
Function process_response(Yhatk)
    tau_user = user delay in samples;
    chunk_length = chunk length in samples;

    /* Calculate number of chunks needed to capture the
       user response.                                      */
    N = ceil(tau_user / chunk_length);

    /* Save the N chunks and merge them.                   */
    X_temp = [];
    repeat N times
        X_temp[i] = read_chunk();
        i++;

    X_merged = merge_chunks(X_temp);

    /* Calculate interval to use and extract new chunk of
       original length using this interval.                */
    n0 = tau_user - chunk_length/2;
    n1 = tau_user + chunk_length/2;
    X = X_merged[n0:n1];

    save(Xk, Yhatk);
```

Assuming user response delay with small variations The assumption of constant user response delay is not always valid. If smaller chunk sizes are used this is even more important as the window in which the EEG response can be captured is smaller. To introduce a degree of time invariance the `n0` and `n1` in Algorithm 2 was allowed to vary. A placeholder function for finding the user response was selected and is denoted `find_focus_point(.)` in Algorithm 3.

Algorithm 3: Pseudo-code run in NA describing a similar process as presented in algorithm 2. In this method the focus point is allowed to vary, and it is defined by the implementation of `find_focus_point(.)`.

```
Function process_response(Yhatk)
    chunk_length = chunk length in samples;
    N = number of chunks to store;

    /* Save the N chunks and merge them.                    */
    X_temp = [];
    repeat N times
        X_temp[i] = read_chunk();
        i++;
    X_merged = merge_chunks(X_temp);

    /* The tau_user is now replaced by a calculated focus
       point.                                               */
    focus_point = find_focus_point(X_merged);
    n0 = focus_point - chunk_length/2;
    n1 = focus_point + chunk_length/2;
    X = X_merged[n0:n1];

    save(X, Yhatk);

/* Method defining the focus point.                         */
Function find_focus_point(X)
    return arg_max(abs(X));
```

The `find_focus_point(.)` method attempts to determine where in the merged epochs the user response is located. Due to time limitations, and the fact that the SAPF was designed only to implement the specific use case later discussed, the focus point was simply set as the point of maximal deviation from the mean of the chunk, and an example of this can be seen in Figure 4.16.

Finally, it should be noted that this approach assumes that the user responds within some fixed time[8]. Moreover, this approach also assumes that the user respond only once in this time window, or at least that the `find_focus_point(.)` finds the correct response.

Discussion and further work regarding the stimuli node

The stimuli node and adapter Due to time limitations very little research was done on implementational—or even conceptual—alternatives to the stimuli adapter and stimuli node. The result was therefore a somewhat "rough" solution, see e.g the filled arrow in Figure 4.4 which indicates that the shared queues concept is not

[8] Within $\frac{N \cdot \text{chunk_length}}{f_s}$ seconds to be specific, where f_s is the sampling frequency.

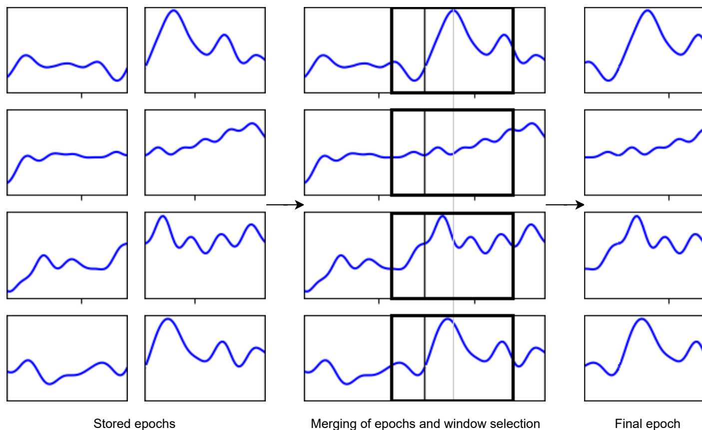


Figure 4.16: Example of focus point selection of merged EEG chunks using the max deviation focus point. In the example two EEG chunks (with 4 channels) are merged after which a focus point is generated using the `find_focus_point(.)`. A span defined by the original chunk length, and the center point (the focus point), is used to extract the final chunk as seen in the figure.

used there. Further work could therefore aim to find better ways of implementing the desired behaviour that aligns better with the SAPF implementation.

Passive mapping A potentially better alternative to the stimulation-response mapping approach would be to build the stimuli adapter around a more passive approach. The stimuli adapter could e.g. return a probability distribution of user actions based on the application state—e.g. if the application is a game, and something attractive is placed next to the player, the adapter can most likely assume that the player will collect the item, the probability of the corresponding player command is thus high. This approach was in fact in focus during the beginning of the thesis, however, due to time limitations, and the fact that it is more complex and less likely to produce a satisfying demonstration, it was discarded.

Limitations of the implementation The problem of no user response was never solved. Furthermore, the selected `find_focus_point(.)` method also assumes that the user user response has a high magnitude. However, the reader should note that the selected `find_focus_point(.)` mainly acts as a placeholder for more sophisticated methods. Previous work suggests that a sliding window calculating the Higuchi's fractal dimension—which measures the complexity of the time series—can be used to find changes in the signals stationarity [Klonowski, 2016]. This could

potentially be used to find intervals in the signal which differentiates—in statistical properties—from baseline or the rest of the EEG time series. These intervals might then be regarded as containing the user response. However, further work is needed to evaluate this method.

Another disadvantage with the approach presented in Algorithm 3 and Algorithm 2 is that it is assumed that the user responds within some time defined by N. The most obvious case is when N is too small and the user too slow, then the user response will not be captured. If N is too big an abundance of chunks are saved, some of which might contain information that is not actually associated with the stimulation response. Further work could therefore focus on developing an algorithm that, in some sense, is capable of early stopping. A check could potentially be done after the `read_chunk(.)` has been executed and one chunk is loaded—if the loaded chunk deviates too much from previously saved chunks containing "nothing", then it can be regarded as containing the user response, and the algorithm can stop storing chunks (i.e. early stopping).

4.11 Final discussion and further work regarding SAPF

Standardized data types

The pipeline nodes lack a proper type check at the input of each node. It is up to the user to make sure that each node is supplied with the correct input type.

The EEG time series are contained in `numpy` arrays. This is also the data type that is sent between most of the nodes. Also, here it is up to the user to make sure that the dimensions are setup and used correctly.

As of now the time series are contained in 4 dimensional arrays as previously mentioned, where the channels take up 2 dimensions. This is mainly due to historical reasons—two dimensions allows one to rearrange the channels according to their physical placement on the head, and previous work that this framework is built on included classifiers that used this spatial information.

One could imagine that the `mne.Epochs` and `mne.io.Raw` objects[9] would be suitable to use instead of `numpy` arrays as the data type of choice to send between the nodes. The `mne.Epochs` comes with many features that could potentially be beneficial. However, in some cases the `mne`'s data types might also provide too much functionalities and the data types were experienced as less transparent to `numpy` arrays.

Process-based nodes and plotting of node data

Attempts were made to create nodes that were able to plot the data that passes through it. This would potentially allow for much better understanding of what

[9] More information about the `mne` package and the included data types can be found on MNE's web page `www.mne.tools`

is happening in the graph. However, many python modules for plotting—such as `matplotlib`—are not completely thread-safe. One solution to the plotting problem might be to either run the `matplotlib` GUI in the main thread, and simply updating it with data coming from the node threads, or to use processes instead of threads. As mentioned, processes-based nodes were implemented but not fully tested. More work that did not align with the goals of the thesis would then be needed to fully understand and implement this, which is why this feature was not added.

Pipeline not capable of calibration and classification at the same time

Due to mainly time limitations the SAPF is not capable of producing a pipeline that can run the data collection in parallel with actual EEG or feature classification. Therefore, to train a classifier and run it one must construct two pipelines—one for collecting the data using the NA, and one for running the real time classification of user EEG.

In practice, this means that the calibration process must be done separately and before the training of the classifier and thereafter the actual controlling of the application can be done.

5

Implementing a game application

In this chapter the implementation of a game application is described. This game will act as the application to control when a BCI system is implemented—using the SAPF—in Chapter 6. The game will also act as a stimulation program for collecting a labeled data set, it will sometimes be referred to as the stimuli program. For more details see the source code[1] Most of the game was built with no connection or dependency to either the previously described SAPF, or the specific BCI system described in Chapter 6. However, the graphics of the game, and the player movement, were heavily affected by the choices made when implementing the specific BCI system.

In Section 5.1 the game world will be presented, the main focus in this section will be the graphics. Section 5.2 is dedicated to the interface between the application interface node and the game itself. And finally, Section 5.3 presents the player movements.

5.1 Game world

In order to visually present the game to the user the game was rendered using the `pygame`[2] Python module. The game is completely tile based, and each tile can either be empty or contain a game object. The game supports different world sizes and 3 different game objects; a player object, collectable objects, and wall tiles.

By sending the appropriate commands to the game program the player object can be moved in the world. Empty tiles are traversable by the player, and wall tiles are non-traversable. When the player object is moved over a collectable object, the object is collected by the player and removed from the game.

[1] `https://gitlab.control.lth.se/users/sign_in`
[2] www.pygame.org

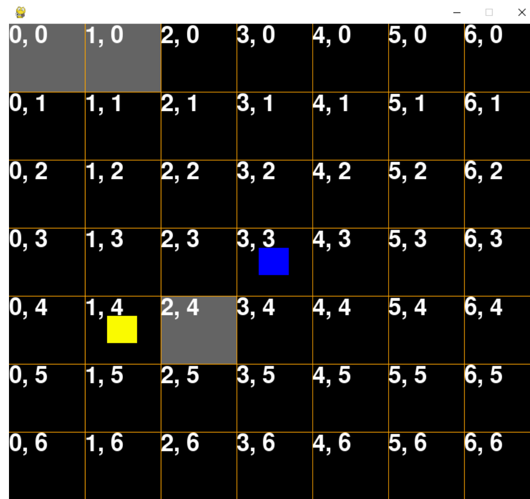


Figure 5.1: During the first iteration simple squares were used to display the game object. The yellow squares represent collectable objects, blue the player object, and gray wall objects.

Iteration 1 – simple graphics

During the first iteration of the game no special considerations on the graphics were made. Colored squares were used to represent the different game objects, see Figure 5.1 for an example world.

Discussion

As mentioned, no special considerations were made during the first graphics iteration. However, after using the game in the specific BCI system presented in Chapter 6 problems arose. As previously briefly mentioned in the introduction, the BCI system in Chapter 6 used eye movements as user commands. In that system, the user moves the player object by looking at any tiles adjacent to the player object.

The first problem that arose was related to the poor choice of fixation targets. In the situation described above the adjacent tiles, and any objects in them, act as potential fixation targets. The homogeneous coloring of both the tiles and the objects was hypothesized to result in unstable fixations. The recorded EEG chunks should then contain both the saccade component—which is the desired component as this act as the user command—and an abundance of the undesired EEG components which is a result of an unstable fixation.

The coordinates displayed in each tile were also experienced to sometimes accidentally act as an fixation targets.

Another problem related to the colors of the game objects are the fact that the player avatar and the collectable tiles look very much the same. The shape and

Figure 5.2: During the second iteration of the game the graphics were updated.

general design does not indicate what the tiles are and how to use them.

Iteration 2 – updated graphics

In the second iteration the tile coordinates were removed. The squares representing the player and the collectable coins were changed to sprites[3] resembling a person[4] and a golden crown[5] respectively, see Figure 5.2.

Due to time limitations no objective analysis was done to investigate the quality of the new fixation points.

Discussion

Compared to the previous iteration the complexity of the visualization of the game objects increased substantially. Due to time limitation no objective comparison were made between the new and old graphics, it is therefore unclear if any improvements were made at all. However, due to the heterogeneity of the new sprites the player now more easily consciously select a point in the sprite and use that as a fixation point. It is unclear how consistently these fixation points could be used.

[3] A sprite is a two dimensional image, e.g. a bitmap, that can be displayed on the screen.

[4] Made by the creator BilouMaster `www.opengameart.org/content/rpg-asset-pack`. Licensed under CC-BY 4.0. No changes were made to the original sprite. Date downloaded 2021-05-19.

[5] Made by the creator Bonsaiheldin `www.opengameart.org/content/gold-treasure-icons-16x16`. Licensed under public domain. No changes were made to the original sprite. Date downloaded 2021-05-19.

The new graphics of the game was experienced as easier to engage in, and more relaxing as the gaze could easily be put on any of the new tile details.

5.2 Remote control of the game application

To increase flexibility of the BCI system the game application was written as a stand-alone application. To allow for remote control of the stimuli program (i.e. control of the application outside the process in which it is running) a simple server was constructed using `multiprocessing`. Any external software running in another process can easily exchange commands with the server by connecting to it. The connection was based on a socket, and from hereon anything that connects to server via this socket is referred to as the client. `multiprocessing` allows for different types of so called Inter-process communication (IPC) methods, and the socket communication will therefore from hereon simply be referred to as IPC.

Game commands

The server has two states; start-up state and running state. During the start-up state the server accepts the following setup commands:

Game setup commands	**Description**
`--gamesize w h`	Set game board width and height to (w, h) number of tiles.
`--wintitle t`	Set the window title to t.
`--winsize x y`	Set the window width and height to (x, y) pixels.

As soon as the first message is received and parsed the server enters the running state in which it accepts the following commands:

Game commands	**Description**
`--move d`	Moves the player in direction d.
`--set t x y`	Se the tile at position (x, y) to the tile type t.
`--clear`	Set all tiles to empty.
`--quit`	Shutdown server.

To simplify sending and receiving these commands the GameClient class was created. In this case the GameClient was used to encapsulate the application specific communication, described above, from the application interface node. The application interface node, or any other node, can use this class to easily communicate with the game server.

Below the GameClient (Figure 5.3) in the "protocol stack" the IPCClient can be seen. This class handles the lower layer communication between processes (i.e. IPC). As mentioned before, the IPC is over sockets.

If any other application is needed the GameClient can easily be swapped, and if any other lower layer communication is needed the IPCClient and IPCServer can be swapped.

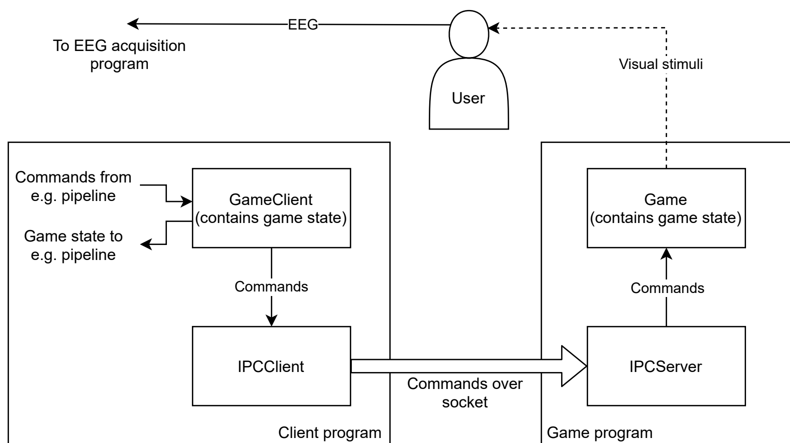


Figure 5.3: Overview showing the general structure of how a client communicates with the game program. The IPCClient and IPCServer classes act as so called mediators classes. Both the GameClient and the Game class contain a model of the game. The state of the games are kept identical by applying the same state transitions on both sides of the socket. The game state is thus directly accessible on the client side.

Sometimes it is desirable to know the state of the application. This is the case when using the stimuli adapter to create stimuli suggestions to the stimuli node. Also, if the passive mapping method suggested in Section 4.10 is implemented, the application state will be needed too. However, as the game is run in its own process, with its own separate memory, it is not possible to access the game state from outside the game process directly.

One solution to this was to allow the GameClient to fetch the game state from the server when it is requested (externally). However, another feasible solution was to store a copy of the game state inside GameClient. By applying the same state transitions on both the client side and server side the GameClient could effectively keep an up-to-date game in its own process as long as the socket connection is alive and working properly. This allows for easy access of the game state from within the client process through GameClient, see Figure 5.3. As the game state is very small

in terms of memory, this was deemed as a good enough solution.

5.3 Player movement

Iteration 1 – free movement

During the first iteration of the game the player object could be moved in all four directions—up, right, down, and left. If no commands were sent to the game the player object remained stationary.

To control the player using the BCI pipeline four distinct user commands would be needed. However, at a later phase of the thesis the 4 commands were reduced to two due to difficulties classifying the four ocular user commands. To allow the player to be able to move in the two dimensional world with only two commands, changes had to be made to the game.

Iteration 2 – snake movement

During the second iteration, the player movement commands were reduced to two which corresponded to clockwise or anti-clockwise turning. The player object was also continuously moved forward in the direction the object faced, similar to the snake in the game snake.

At this phase of the thesis the game was planned to be entirely rewritten. Therefore, the question of where to implement the continuous forward moving behaviour naturally arose.

Method 1 – contained in game mechanics, using two parallel games

In the first method the forward movement was identified as intrinsic to the game, and a part of the game update. This implementation allowed the game to handle the forward movement without any external commands or interaction. The server could thus run the game—with the forward movements—without the client.

This choice lead to the question of how to handle the two parallel copies of the game state. As mentioned, and as previously seen in Figure 5.3, the game models are updated and kept synchronized by applying the same transitions on both models. However, if the game state is changed internally from within the server without notifying the client, the game states will diverge[6].

The naive solution to this problem would be to run the same internal logic on the client side as is run on the server side. Both the client and server side thus have this internal dynamics—simply referred to as game mechanics—that updates the model without external interaction.

As one can imagine, this potentially also leads to diverging states. One aspect of this divergence is the racing condition that is intrinsic to this method. If an external

[6] Diverging here simply refers to the game models becoming less and less similar due to e.g. an external event associated with a state transition not being applied to the models at exactly the same time, or non-equal model update frequencies.

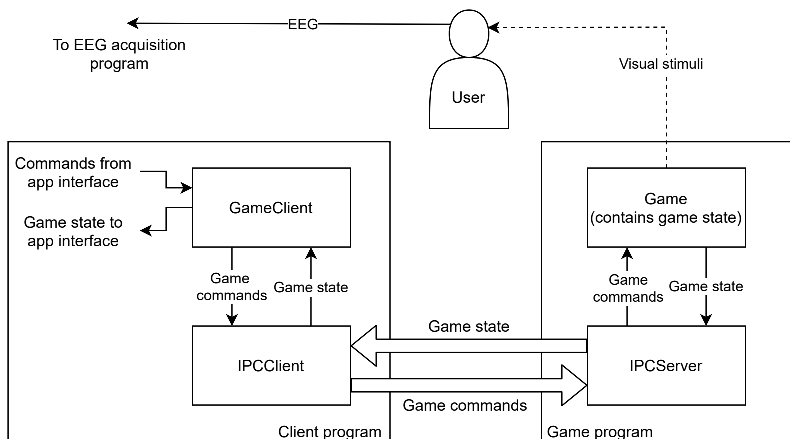


Figure 5.4: The so called game mechanics are contained on the server side (in the Game class). When needed on the server side, the game state is fetched through a request over the IPC connection. This approach is used by Method 2.

action is producing a state transition on the server side, the corresponding game command is sent over the socket to server to, hopefully, produce the same transition there. However, due to the small time delay between client and server, the server model could have transitioned to a new state before the game command arrives. In short, the game commands are constructed to invoke a specific transition at the client side, but could potentially meet a model with a different game state at the server side.

Method 2 – contained in game mechanics, using one game

A more sophisticated approach was to only run the so called game mechanics on the server and simply let the client fetch the game state when needed, see Figure 5.4 (which meant that a new game command had to be introduced). However, one should note that racing conditions are still present in the system if any action on the client side is based on the fetched game state—the fetched game state might be outdated at the time when the said action is generated.

An example of when this is obvious is during the generation of golden crown tiles next to a forward moving player. The "next to" here indicates the game state is needed to find tiles adjacent to the player. If the forward moving player makes a forward jump between the client fetch and the generation of the golden crown, the "adjacent tiles" will no longer be adjacent to the player, and the crown will not appear where it was supposed to.

Method 3 – as an external interaction

Due to the multitude of potential problems associated with the first and second method, a third less complex approach was outlined.

In contrast to the other methods described, the forward movement of the player was not viewed as a part of the game mechanics. Instead, the forward movement was produced by continuously sending appropriate movement commands from the client. When this approach was used all dynamics were eliminated from the game, and so are the racing conditions.

Similar to the first method, the client holds the game state. In contrast to the first and second method, all dynamics are contained in the game client. Because of this one can be sure that the available game state is always the most recent. There are also no issues with racing conditions and diverging games. To clarify, this method does not use two way communication, and operates as previously shown in Figure 5.3.

This approach was used in the final software.

6

Implementation of a demonstration using SAPF and the game

This chapter describes a BCI system implemented using the SAPF and the game previously presented.

Section 6.1 the selection of EEG signal to use as user command will be presented and discussed. Section 6.2 gives a discussion and motivation of the choice of hardware. In In the subsequent sections, Section 6.3–6.8, each node will be configured for the specific BCI system in this chapter. Section 6.9 will present the construction of the BCI pipeline using the configured nodes.

As will be discussed further in this chapter, the signals used as commands were at first eye movements. The outcome was, however, poor. Therefore a second attempt was made using jaw clenching and eye blinks as commands. The following sections will treat both attempts.

6.1 User commands

At first, eye movements were selected as user commands. The idea was that the user should control the player object in the game by looking at tiles adjacent to the player. The number of commands are thus 4. Problems arose when these commands were used. Due to these difficulties the commands were later swapped to eye-blinks and jaw clenching (pressing teeth together) instead.

6.2 Hardware selection

Selecting EEG hardware

At first the EasyCap EEG-cap and the Smarting amplifier were considered. However, with its 24 wet-type electrodes the system provided more data—and of higher quality—than needed in this thesis. Therefore, the MuseS was selected. Its 5 dry-type electrodes, and sampling frequency of 256 Hz, provided the right quantity (in terms of number of channels) and quality of data for the use-case in this thesis.

Moreover, the inexperience of the author in the field of experimental setup and EEG further motivates the selection of the Muse S. The inexperience would—as predicted—lead to many spontaneous experiments which further required a remissive setup in the sense that it allowed for quick and not thoroughly planned experiments. The setup time of EasyCap—which is in the vicinity of 20 minutes due to the requirement of conductive paste—does not allow quick and spontaneous experiments. In contrast, the setup time of Muse S is a couple of seconds, mainly due to the usage of dry-type electrodes.

Other hardware

To run the experiment a Bluetooth module was needed to connect to the Muse S. BLED112 was selected.

6.3 The LSL node configuration

The LSL node was configured using $L_c = 64$ samples. According to Equation (4.1) and Equation (4.2) the theoretical minimal average experienced system delay is then 125 ms, which was deemed small enough.

6.4 The pre-processing node configuration

The filter coefficients were calculated using `mne.filter.create_filter(.)` instead of more transparent methods such as `scipy.signal.firwin(.)` (which the `mne`-method is built on). The filter design arguments are shown in Table 6.1. One should note that `mne.filter.create_filter(.)` automatically determines the length of the filter based on the given requirements.

As indicated by the requirements, high-pass filtering (implied by the low-frequency cutoff) was used to remove drift as BC was deemed not suitable for online filtering in Section 4.5.

The transition bandwidth was deliberately set to relatively large number—as advised by [Widmann et al., 2014]—which resulted in "poor" attenuation of stop-band in the sense that the transition region between pass- and stop-band is relatively large. However, this should also reduce ringing artefacts. To compensate for the

Table 6.1: Filter arguments

Argument explanation	Argument name	Value
Low-pass cutoff frequency	`l_freq`	0.5 (Hz)
High-pass cutoff frequency	`h_freq`	20 (Hz)
High-frequency transition bandwidth	`h_trans_bandwidth`	15 (Hz)
Filter phase	`phase`	'minimum'
Design method	`fir_design`	'firwin'
Filter type	`method`	'fir'

large transition bandwidth the high-pass cutoff frequency was deliberately set to the low value 20 Hz to avoid the line-noise at 50 Hz.

The AUX channel of the Muse S was discarded by the pre-processing node as this does not contain any EEG.

Due to time limitations no objective evaluation were done to maximize signal-to-noise ratio, or to keep as much of the signal power as possible. The signal was simply examined—in time domain—before and after filtering, and the parameters were changed based on intuition until satisfying results were achieved.

Discussion

Much work was dedicated to understand the pre-processing and selection of filter-parameters of previous work. However, in the end, much of the filter design was based on experimentation. Also, no exclusive line-noise filtering was deemed necessary.

It was also unclear how much effect the minimum-phase argument made except introducing a delay in the filter output. It was also unclear if the selection of an IIR filter would potentially provide better filtering.

Moreover, as mentioned, phase-distortion distorts the temporal shape of the signal, which might be critical if high time precision is necessary (phase-distortion "smears" the signal). However, if features based on the spectrum of the entire chunk is used, this most likely has no or little effect as the frequencies are still present in the chunk.

The filter produced by the filter method did not possess linear phase, and the maximum time delay in the pass-band was 4 ms. This delay is relatively small compared to the LSL buffering delay, which was 125 ms.

6.5 The processing node configuration

The processing node was equipped with a channel-wise variance check. If the variance was too high the chunk was rejected (i.e. not passed further in the pipeline). The rejection threshold was set to 4000.

Discussion

Initially, the intent of the node was to reject chunks containing eye blinks when only eye movements were considered a command. However, the variance rejection features proved useful for rejecting epochs containing rapid head movements or channels with electrodes with poor connection, and were thus kept even after eye-blinks were considered an user command.

However, the combination of the processing node and nodes such as the stimuli node—which were built around the assumption that a stimulation command always result in a response as presented in Section 4.10—did not mix well. Remember, the the stimuli node stimulates the user and expects an incoming chunk to contain the response. However, what if the chunk has high variance and is rejected by the processing node? Then the mapping will fail, and therefore the processing node was later not used in the acquisition pipeline.

6.6 The feature extractor node configuration

Selecting the features

The selection of features were based on the first choice of user signals, namely eye movements. However, as will be seen later, the selected features worked well to distinguish eye-blinks from jaw clenching as well. The reader should therefore note that the reasoning in the following sections are based on eye movements as user commands.

Spectral envelope

As was seen in Section 3.7, the frequency content of saccades and eye-blinks seem to be a distinguishing feature.

Therefore different parts of the spectrum were extracted, and the sum of the frequency bins corresponding to the specific part of the spectrum were used as a feature. One could simply select the correct frequency bins, however, a filter bank was used for this instead. The filter bank (Figure 6.2) was based on 6 triangular filters.

When 4 EEG channels are used in the EEG chunks a total of 24 features are extracted from each chunk. The Python package `pyfilterbank`[1] was used to construct the filters.

[1] `www.github.com/SiggiGue/pyfilterbank`

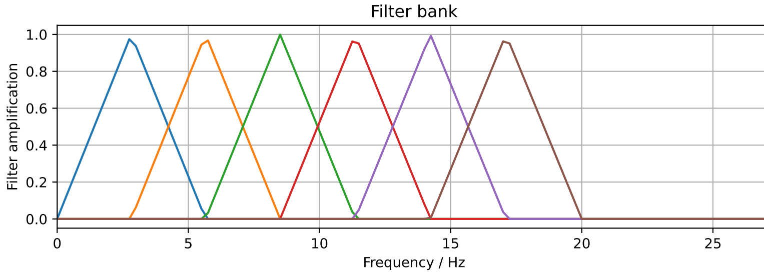


Figure 6.1: Triangular filters used in the filter bank.

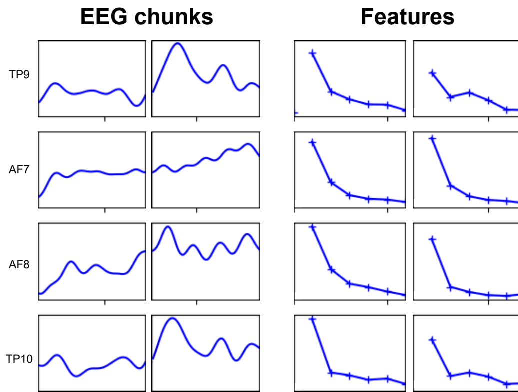


Figure 6.2: Example showing the features (right) which were extracted from the EEG chunks (left).

Discussion

At first, channel variance features seemed suitable to input to a classifier. However, a desired classifier accuracy could not be achieved using this method alone. It is unclear exactly what is the cause of this, but it was hypothesized that the variance feature itself had a high inter-class variance. This could be especially true when using the first iteration graphics of the game application—and eye-movements as user commands—which had problems with fixation points. The lack of a proper fixation point could potentially lead to a higher variety of saccade sizes, and a lower fixation stability, which potentially could increase the inter-class variance when using the variance feature. However, it should be noted that no data is available to support this claim.

In the pipeline no data normalization was done to the EEG chunks entering the feature extractor node. Using some normalization might increase the classification

accuracy.

6.7 The stimuli node and adapter configuration

When the NA is configured most of the work is done in the adapter part. This is, as mentioned before, because most of the use-case specific behaviour is located in the adapter part.

Two different stimuli strategies were considered when implementing the adapter—these are referred to as *collect the crowns*, and *limited traversability*. Note that the stimuli node expects a known user response for each possible stimulation presented by these two methods. Thus, both strategies aim to change the game state in such a way that the type of the user response can be regarded as known. However, due to time limitations, and due to the changes made in the game during the second iteration, only the "collect the crowns" strategy was implemented.

Collect the crowns

The adapter based on the collect the crowns strategy finds traversable tiles next to the player to generate a crown in, see Figure 6.3. The adapter associates each of these tile selections with a user response class. The expected user response is to collect the crown, and in Figure 6.3 this is "move left".

Figure 6.3: Example of game world during calibration session with stimuli adapter based on the collect the crowns approach.

The stimuli node was first configured using Algorithm 2 with $\tau_{user} = 1$ second. A test using Algorithm 3 was also done, then $N = 4$ which—with the LSL node's $L_c = 64$ samples and Muse S $f_s = 256$ Hz—corresponds to 1 second. To summarize, in both methods it is assumed that the user responds to the crown stimulation within 1 second. The time interval between stimuli were set to 2 seconds. Note that, as previously mentioned, the chunks collected in this period is labeled as "nothing".

Limited traversability

The limited traversability approach aims to reduce the number of reasonable player actions by reducing the traversability of the tiles surrounding the player.

Figure 6.4: Example of game world during calibration session using the limited traversability approach.

Discussion

Not much more work was dedicated to the limited traversability approach, and it is unclear if it is even possible to implement this strategy using NA—the stimuli node and stimuli adapter duo—as the NA is based around the stimulation command and response concept. During the limited traversability approach no changes are made to the game state to stimulate a user to respond. The discussion in Section 4.10 about asynchronous mapping suggests that mapping approach would have been suitable for this strategy.

Table 6.2: The two pipelines and the included nodes. These two pipelines acts as the BCI system.

Acquisition pipeline	Classification pipeline
LSL node	LSL node
Pre-processing node	Pre-processing node
	Processing node
	Feature extractor node
Stimuli node	Classification node
Application interface node	Application interface node

6.8 The classification node configuration

The classifier was a simple CNN built in Keras, see Appendix for a summary. Little time was dedicated to finding a good classifier as this was not the main objective.

6.9 Building and running the pipelines

Building the pipelines

As mentioned before, the framework was not capable of producing pipelines capable of running both a classifying pipeline and the data labeling pipeline. Therefore two pipelines were constructed, see Figure 6.2. The implementations and settings described in this chapter were used for the nodes. Note that, due to time limitations, these two pipelines are, as mentioned, not capable of running in parallel.

The reason why the processing node was not used in the data acquisition pipeline was due to the chunk rejection that it contains. The stimuli node, which is also present in the acquisition pipeline, does not cooperate well with this behaviour as it assumes that a stimulation command leads to a response chunk. This is not the case if the processing node "accidentally" rejects one or more of the chunks.

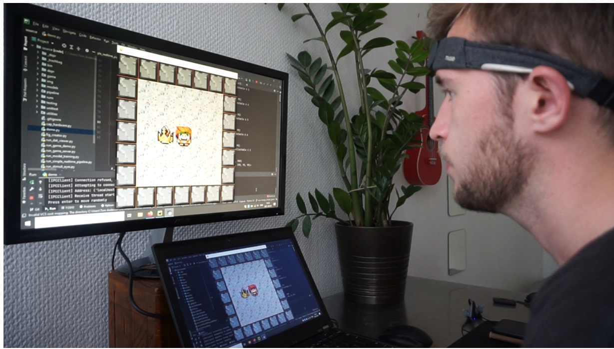
Moreover, the feature extractor node was not included in the acquisition pipeline as the `find_focus_point(.)` implementation was based on finding the user response in the EEG chunks and not chunks containing features.

Running the acquisition pipeline

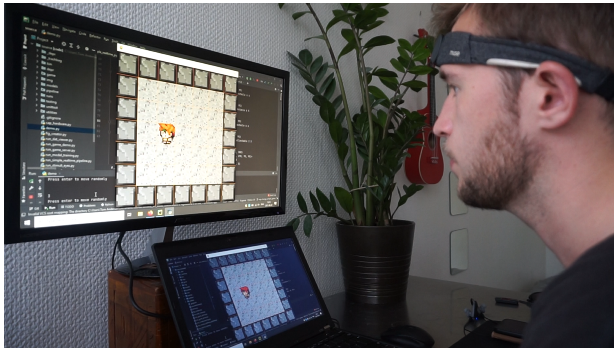
The acquisition pipeline was setup according to Table 6.2, and to the configurations described in this chapter. As mentioned, the collect the crowns method was used, which can be seen in Figure 6.5. The game server and the pipeline was started, these are automatically connected and an overview of the initial setup commands (automatically) sent by the GameClient can be seen below:

Game setup commands	**Comment**
`--gamesize 7 7`	Set game size to (7, 7)
`--winsize -1 -1`	Automatically sets window size to maximize the window size of the game, while keeping the aspect ratio of tiles 1:1.
`--wintitle " "`	
Game commands	
`--set 3 3 PLAYER`	Create a player object in the center of the game world.
`--setsquare 0 0 6 6 WALL`	Generate the walls around the borders of the map (for aesthetic purposes only)

Note, the NA takes cares of the rest of the communication needed to implement the collect the crowns approach, and is not included here.



(a) A crown appears next to the player.



(b) The user attempts to collect the crown by sending the appropriate user command, in this case by pressing teeth together slightly.

Figure 6.5: The process of stimulating the user by creating a crown next to the player object. The user responds by attempting to collect the crown, in this case by pressing the teeth together. More crowns appear either to the left or right of the player as the calibration proceeds.

As mentioned, two user commands were used—jaw clenching and eye-blinks. During the calibration the user can attempt to either move the player to the left or to the right[2]. After stimulation $N = 4$ (1 second) EEG chunks are collected after stimulation onset. Using Algorithm 2 the user response was identified among these $N = 4$ chunks, and one single chunk extracted from these chunks which was associated with the expected user response (i.e. "left" in Figure 6.5).

In total 40 crowns were collected, which corresponds to 20 data points of "left", and 20 of "right". Meanwhile, 200 data points of "nothing" were collected.

[2] Again, note, the player **does not actually control the player**. It is the NA that does that.

Processing and training of the classifier

The data set generated during the calibration phase was used. The feature extraction had to be done manually using a `FeatureExtractor` object implemented to extract the features previously described. A new data set was thus generated containing the features. This data set was used to train the classifier.

Running the classification pipeline

The `FeatureExtractor` object was inserted into the feature extractor node, and the trained `Classifier` object into the classification node. The classification pipeline described in Table 6.2 was setup.

Due to the fact that only 2 commands were used, instead of 4 as was initially planned, the free movement described in Section 5.3 could not be used if the whole game world shall be traversable by the player. Instead the snake movement was used in the GameClient. The collected "left" and "right" data during acquisition was thus, in the real time classification pipeline, regarded as turning left and right respectively.

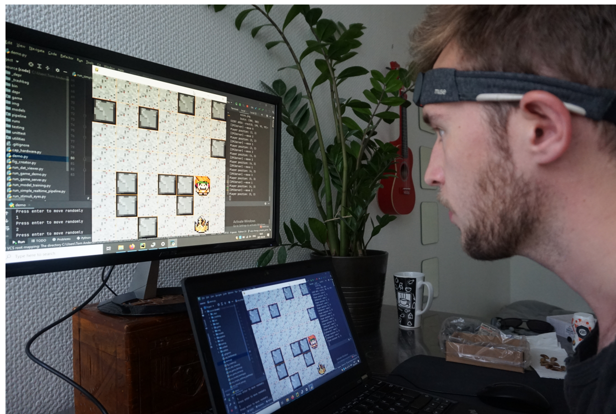


Figure 6.6: The classification pipeline running. A random game world with one crown can be seen above.

A demonstration of the process mentioned above can be found here `youtu.be/i3kGB5Ed8ww`.

6.10 Discussion

As mentioned, no objective evaluation of anything was done. However, the author experienced that it took some time to learn the control after the calibration process. After a few minutes of controlling the player object around the world the author experienced that the control was good enough to be enjoyable. It should be relatively

simple to add more game objects to the game that in turn would make the game itself more interesting. However, the final game play was left as presented in Figure 6.6.

7

Final remarks

The SAPF presented a simple way of stacking functionality together. However, it is unclear if the asynchronous multi-threading approach (the concept of nodes) presented in SAPF was a good design choice. If one chunk is present in the pipeline at once it does not matter if the pipeline works asynchronously as it will appear as if it is working synchronously anyways. The context switching that is intrinsic to the multi-threading approach might introduce unnecessary processing delays, and the approach itself might introduce unnecessary complexity. It is arguable if the node based concept makes the code more or less readable.

However, the SAPF was able to construct a BCI pipeline capable of controlling a simple game, which was two of the objectives of the thesis. The final objective was to create a simple game to control, and this was also accomplished.

Throughout this thesis many problems, bugs, and other setbacks have affected the quality and quantity of the work done. Much of the technical problems revolved around the LSL connection made between the EEG hardware and the computer on which the BCI is run. Bluetooth dongle compatibly issues also setback the thesis somewhat.

However, not only problems were to be found, throughout the many investigations and implementations done in this thesis many opportunities for further work was discovered. Some of the most interesting ones might be the passive mapping approach, presented in Section 4.10, which might be combined with the sliding window fractal presented in the same section. If the synchronous mapping approach—the stimulation command to response mapping—is further used one can attempt to implement early-stopping and improve the `find_focus_point(.)`.

Another interesting feature would be if the SAPF actually could produce pipelines capable of both calibration and classification simultaneously. Some work was dedicated to implementing other types of nodes—based on the stimuli node and the classifier node—that could produce this behaviour. These were never finished and were thus never included in the thesis.

The main reason for using Python throughout this thesis was that it is simple to learn and thus allows more potential users to make changes to the SAPF later. However, if one wish to write larger programs Python has its drawbacks. This was

especially noted when attempting to implement the more complicated nodes where no known code patterns, at least to the author, could have been used instead. Also, as it is not possible to declare the types of Python variables, mistakes are sometimes easily made which might go unnoticed as there are no compiler warnings. The question is then, is software written in 100% Python code really easier to understand than e.g. C++ or Java code?

Lastly, some of my own thoughts will be presented. The SAPF pipeline provides a solution to create asynchronous data pipelines and could most likely be used similarly in other contexts outside BCI. It is flexible in the sense that it is relatively easy to adjust nodes, change order of them, and implement new nodes. However, the implementation solutions to many problems that arose throughout the thesis are inefficient and generally regarded as "rough" solutions (code smell). Some parts of the source code of SAPF follows bad code practices and should ideally be rewritten. Also, some of the core mechanics of SAPF—i.e. the buffering using queues, and the corresponding threads—could more efficiently be implemented in e.g. C++. At this stage, however, the SAPF would indeed just be an uglier twin of OpenVibe. Therefore, if one only wish to demonstrate a certain BCI system only, OpenVibe is most certainly a better choice than SAPF.

8

Appendix

Pipeline and system delay

As the pipeline is run under a non-real time OS no guarantees can be made on the upper bound of the pipeline delay.

If one assumes that the processing speed of all nodes are 0, i.e. assumption 2 holds, and the time delay imposed by context switching is negligible—i.e. the only source of time delay is the buffering in the LSL node. Then a lower bound on the pipeline delay can be formulated by using the delay imposed by the LSL node.

If a sample enters the LSL buffer right before a chunk is created the sample will experience a time delay of 0 seconds. If a sample enters the buffer right after a chunk is created the sample is the first sample in the buffer, and it has to wait there until the buffer is full before the buffer can be flushed and a chunk created. The sample experience a time delay of L_c/f_s seconds, where L_c is the chunk length, and f_s the sampling frequency (rate of which samples appear on the buffer) which is assumed to be constant. I.e.:

$$\min(\tau_{LSL}) = 0, \quad \max(\tau_{LSL}) = \frac{L_c}{f_s}$$

Assume that the user response can be pin-pointed to one sample. As the responses of the user of the system is, in most cases, independent of the buffer status it is reasonable to assume that this sample appears (discrete) uniformly on the LSL buffer[1].

$$\tau_{samp,pipeline} = \text{unif}\left\{0, \frac{L_c}{f_s}\right\}$$

The expected delay of the user response sample in the pipeline is thus:

$$E\{\tau_{samp,pipeline}\} = \frac{L_c}{2f_s}$$

[1] This is not always the case. During the active mapping approach the stimulation and the appearance of the user response sample is highly correlated. As the stimulation is synchronized with the appearance of chunks on the stimuli node buffer (and therefore the LSL buffer if the time delay is assumed to be 0, i.e. assumption 2 holds), then user response is also synchronized with the LSL buffer. In this case the user response is most likely not uniformly distributed in the LSL buffer.

This delay only includes the delays imposed by the pipeline itself, where most of them—expect the LSL buffer delay—is assumed to be 0.

If one wishes to account for delays imposed by fetching the samples from the hardware, and sending the pipeline results to the application, a lower bound of the entire system delay can be formulated. This delay, τ_{sys}, is what is experienced by the user. The LSL buffer delay is included in this delay, then:

$$\tau_{sys} \geq \tau_{LSL}$$

and the expected delay of the user response sample in the pipeline is thus

$$E\{\tau_{sys}\} \geq E\{\tau_{samp,pipeline}\} = \frac{L_c}{2f_s}$$

.

Keras model

```
model.summary()
Model: "sequential"
_________________________________________
Layer (type)                 Output Shape
Param #
=========================================
conv3d (Conv3D)              (None, 8, 4, 1, 4)
28

_________________________________________
max_pooling3d (MaxPooling3D) (None, 4, 3, 1, 4)
0

_________________________________________
batch_normalization (BatchNo (None, 4, 3, 1, 4)
16

_________________________________________
flatten (Flatten)            (None, 48)
0

_________________________________________
dense (Dense)                (None, 8)
392

_________________________________________
dropout (Dropout)            (None, 8)
0

_________________________________________
dense_1 (Dense)              (None, 5)
45
```

```
_________________________________________________________________
dropout_1 (Dropout)          (None, 5)
0
_________________________________________________________________
dense_2 (Dense)              (None, 3)
18
=================================================================
Total params: 499
Trainable params: 491
Non-trainable params: 8
_________________________________________________________________
```

Bibliography

Bin, G., X. Gao, Y. Wang, B. Hong, and S. Gao (2009). "VEP-based brain-computer interfaces: Time, frequency, and code modulations [Research Frontier]". *IEEE Comp. Int. Mag.* **4**, pp. 22–26. DOI: 10.1109/MCI.2009.934562.

Chang S. Nam, Inchul Choi, Amy Wadeson, and Mincheol Whang (2018). "Brain–computer interface an emerging interaction technology". Ed. by F. L. Chang S. Nam Anton Nijholt, pp. 11–52.

Chioran, G. and R. Yee (1991). "Analysis of electro-oculographic artifact during vertical saccadic eye movements." *Graefe's Archive for Clinical and Experimental Ophthalmology* **229**:3, pp. 237–241. ISSN: 0721832X. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0025732468&site=eds-live&scope=site.

Collinger, J. L., R. A. Gaunt, and A. B. Schwartz (2018). "Progress towards restoring upper limb movement and sensation through intracortical brain-computer interfaces". *Current Opinion in Biomedical Engineering* **8**. Neural Engineering/ Novel Biomedical Technologies: Neuromodulation, pp. 84–92. ISSN: 2468-4511. DOI: https://doi.org/10.1016/j.cobme.2018.11.005. URL: https://www.sciencedirect.com/science/article/pii/S2468451118300369.

Computational Neuroscience, S. C. for (n.d.). *Labstreaminglayer*. URL: https://github.com/sccn/labstreaminglayer.

Grübler, G., E. Hildt, A. Al-Khodairy, R. Leeb, I. Pisotta, A. Riccio, and M. Rohm (2014). "Psychosocial and ethical aspects in non-invasive EEG-based BCI research - A survey among BCI users and BCI professionals." *Neuroethics* **7**:1, pp. 29–41. ISSN: 18745504. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-84896040602&site=eds-live&scope=site.

Haas, L. F. (2003). “Hans Berger (1873–1941), Richard Caton (1842–1926), and electroencephalography”. *Journal of Neurology, Neurosurgery & Psychiatry* **74**:1, pp. 9–9. ISSN: 0022-3050. DOI: 10.1136/jnnp.74.1.9. eprint: https://jnnp.bmj.com/content/74/1/9.full.pdf. URL: https://jnnp.bmj.com/content/74/1/9.

Jurcak, V., D. Tsuzuki, and I. Dan (2007). “10/20, 10/10, and 10/5 systems revisited: their validity as relative head-surface-based positioning systems.” *Neuroimage* **34**:4, pp. 1600–1611. ISSN: 1053-8119. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselp&AN=S1053811906009724&site=eds-live&scope=site.

Kaplan, R. M. (2011). “The mind reader: the forgotten life of Hans Berger, discoverer of the EEG”. *Australas Psychiatry* **19**:2, pp. 168–169.

Keren, A. S., S. Yuval-Greenberg, and L. Y. Deouell (2010). “Saccadic spike potentials in gamma-band EEG: Characterization, detection and suppression”. *NeuroImage* **49**:3, pp. 2248–2263. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2009.10.057. URL: https://www.sciencedirect.com/science/article/pii/S1053811909011288.

Klonowski, W. (2016). “Fractal Analysis of Electroencephalographic Time Series (EEG Signals)”. In: Di Ieva, A. (Ed.). *The Fractal Geometry of the Brain*. Springer New York, New York, NY, pp. 413–429. ISBN: 978-1-4939-3995-4. DOI: 10.1007/978-1-4939-3995-4_25. URL: https://doi.org/10.1007/978-1-4939-3995-4_25.

Krigolson, O. E., C. C. Williams, A. Norton, C. D. Hassall, and F. L. Colino (2017). “Choosing MUSE: Validation of a Low-Cost, Portable EEG System for ERP Research”. *Frontiers in Neuroscience* **11**, p. 109. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00109. URL: https://www.frontiersin.org/article/10.3389/fnins.2017.00109.

Kryger, M., B. Wester, E. A. Pohlmeyer, M. Rich, B. John, J. Beaty, M. McLoughlin, M. Boninger, and E. C. Tyler-Kabara (2016). “Flight simulation using a Brain-Computer Interface: A pilot, pilot study”. *Experimental Neurology* **287**. DOI: 10.1016/j.expneurol.2016.05.013.

Kübler, A. (2020). “The history of BCI: From a vision for the future to real support for personhood in people with locked-in syndrome.” *Neuroethics* **13**:2, pp. 163–180. ISSN: 18745504. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85066635311&site=eds-live&scope=site.

Lyons, R. G. (n.d.). *Understanding Digital Signal Processing 3rd Edition c2011 (Lyons)*. Prentice Hall. ISBN: 0-13-702741-9,978-0-13-702741-5.

Mason, S. and G. Birch (2003). "A general framework for brain-computer interface design." *IEEE Transactions on Neural Systems and Rehabilitation Engineering, Neural Systems and Rehabilitation Engineering, IEEE Transactions on, IEEE Trans. Neural Syst. Rehabil. Eng* **11**:1, pp. 70–85. ISSN: 1558-0210. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.1200910&site=eds-live&scope=site.

Masood, F., M. Hayat, T. Murtaza, and A. Irfan (2020). *2020 International Conference on Emerging Trends in Smart Technologies (ICETST), Emerging Trends in Smart Technologies (ICETST), 2020 International Conference on*. ISSN: 978-1-7281-7113-5. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.9080743&site=eds-live&scope=site.

Millett, D. (2001). "Hans Berger: From Psychic Energy to the EEG." *Perspectives in Biology and Medicine* **44**:4, pp. 522–542. ISSN: 1529-8795. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edspmu&AN=edspmu.S1529879501405229&site=eds-live&scope=site.

Muse (n.d.[a]). *Muse S*. Accessed: 2021-03-18. URL: %5Curl%7Bhttps://choosemuse.force.com/s/article/What-electrode-channels-does-Muse-use?language=en_US%7D.

Muse (n.d.[b]). *Muse S*. Accessed: 2021-03-18. URL: %5Curl%7Bhttps://choosemuse.com/muse-s/%7D.

Noureddin, B., P. Lawrence, and G. Birch (2007). "Time-frequency analysis of eye blinks and saccades in eog for eeg artifact removal". In: pp. 564–567. DOI: 10.1109/CNE.2007.369735.

Oostenveld, R. and P. Praamstra (2001). "The five percent electrode system for high-resolution EEG and ERP measurements". *Clinical Neurophysiology* **112**:4, pp. 713–719. ISSN: 1388-2457. DOI: https://doi.org/10.1016/S1388-2457(00)00527-7. URL: https://www.sciencedirect.com/science/article/pii/S1388245700005277.

OpenVibe (2017). *Openvibe software architecture*. URL: http://openvibe.inria.fr/software-architecture-130/ (visited on 2021-03-31).

Peter Brunner, G. S. (2018). "Bci software". Ed. by F. L. Chang S. Nam Anton Nijholt, pp. 323–336.

Picton, T., P. van Roon, M. Armilio, P. Berg, N. Ille, and M. Scherg (2000). "Blinks, saccades, extraocular muscles and visual evoked potentials (reply to Verleger)." *Journal of Psychophysiology* **14**:4, pp. 210–217. ISSN: 02698803. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0034458593&site=eds-live&scope=site.

Plöchl, M., J. Ossandón, and P. König (2012). "Combining EEG and eye tracking: Identification, characterization and correction of eye movement artifacts in electroencephalographic data". *Frontiers in human neuroscience* **6**, p. 278. DOI: 10.3389/fnhum.2012.00278.

Proakis, J. G. and D. K. Manolakis (2006). *Digital Signal Processing (4th Edition)*. 4th ed. Prentice Hall. ISBN: 0131873741. URL: http://www.amazon.com/Digital-Signal-Processing-John-Proakis/dp/0131873741%3FSubscriptionId%3D192BW6DQ43CK9FN0ZGG2%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0131873741.

Rolfs, M. (2009). "Microsaccades: Small steps on a long way". *Vision Research* **49**:20, pp. 2415–2441. ISSN: 0042-6989. DOI: https://doi.org/10.1016/j.visres.2009.08.010. URL: https://www.sciencedirect.com/science/article/pii/S0042698909003691.

Schalk, G. and B. Z. Allison (2018). "Chapter 26 - Noninvasive Brain–Computer Interfaces". In: Krames, E. S. et al. (Eds.). *Neuromodulation (Second Edition)*. Second Edition. Academic Press, pp. 357–377. ISBN: 978-0-12-805353-9. DOI: https://doi.org/10.1016/B978-0-12-805353-9.00026-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128053539000267.

Simon L., K., L. David, M. Danilo P., and K. Preben (2017). "Physiological artifacts in scalp eeg and ear-eeg." *BioMedical Engineering OnLine* **16**:1, pp. 1–16. ISSN: 1475-925X. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsdoj&AN=edsdoj.f7d87fc0dbcc404d928bba65476f32bf&site=eds-live&scope=site.

Smarting (n.d.). *Smarting user manual*. Accessed: 2021-03-18.

Smith, S. W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA. ISBN: 0966017633.

Stegman, P., C. S. Crawford, M. Andujar, A. Nijholt, and J. E. Gilbert (2020). "Brain–Computer Interface Software: A Review and Discussion". *IEEE Transactions on Human-Machine Systems* **50**:2, pp. 101–115. DOI: 10.1109/THMS.2020.2968411.

Thaler, L., A. Schütz, M. Goodale, and K. Gegenfurtner (2013). "What is the best fixation target? the effect of target shape on stability of fixational eye movements". *Vision Research* **76**, pp. 31–42. ISSN: 0042-6989. DOI: https://doi.org/10.1016/j.visres.2012.10.012. URL: https://www.sciencedirect.com/science/article/pii/S0042698912003380.

Vidal, J. J. (1973). "Toward Direct Brain-Computer Communication." *Annual Review of Biophysics and Bioengineering* **2**:1, pp. 157–180. ISSN: 00846589. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.

com/login.aspx?direct=true&db=edo&AN=ejs6236742&site=eds-live&scope=site.

Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, l. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. 1. 0. Contributors (2020). "Scipy 1.0: fundamental algorithms for scientific computing in python". *Nature Methods*.

Widmann, A., E. Schröger, and B. Maess (2014). "Digital filter design for electrophysiological data—a practical approach". *Journal of neuroscience methods* **250**. DOI: 10.1016/j.jneumeth.2014.08.002.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2021

Document Number
TFRT-6141

Author(s)
Tom Andersen

Supervisor
Frida Heskebeck, Dept. of Automatic Control, Lund University, Sweden
Carolina Bergeling, Dept. of Automatic Control, Lund University, Sweden
Bo Bernhardsson, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Implementation of a Simple Asynchronous Pipeline Framework (SAPF) for construction of real-time BCI systems

Abstract

This thesis attempts to implement a library in pure Python for building real-time Brain-Computer Interface (BCI) systems. The library does this by employing nodes containing data transformation methods—filtering, classification, data acquisition, and more. These nodes are linked together to create pipelines of nodes in which data flows. Due to the asynchronous nature of the data flow the library was named Simple Asynchronous Pipeline Framework (SAPF).

Moreover, a demonstration BCI was also built using SAPF. In parallel with this, a small game application was developed which the specific BCI system was used to control. At first eye-movements were considered as a user command for controlling the example BCI system. However, in a later phase this was changed to eye-blinks and jaw clenching.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-81

Recipient's notes

Security classification

http://www.control.lth.se/publications/