UNIVERSIDAD
NACIONAL
DE COLOMBIA

EEG-based BCI monitoring framework: Real-time acquisition and visualization from audiovisual stimulation paradigms

Yeison Nolberto Cardona Álvarez

Universidad Nacional de Colombia
Faculty of Engineering and Architecture
Department of Electric, Electronic and Computing Engineering
Manizales, Colombia
2022

EEG-based BCI monitoring framework: Real-time acquisition and visualization from audiovisual stimulation paradigms

Yeison Nolberto Cardona Álvarez

Dissertation submitted as a partial requirement to receive the grade of:
Master of Engineering - Automatics

Director:
Prof. Germán Castellanos-Domínguez, Ph.D.

Co-director:
Prof. Andrés Marino Álvarez-Meza, Ph.D.

Academic research group:
Signal processing and recognition group (SPRG)

Universidad Nacional de Colombia
Faculty of Engineering and Architecture
Department of Electric, Electronic and Computing Engineering
Manizales, Colombia
2022

Framework de monitoreo para BCI basado en EEG: Adquisición, visualización y procesamiento en tiempo real de paradigmas de estimulación audiovisual

Yeison Nolberto Cardona Álvarez

Disertación presentada como requisito parcial para recibir el título de:
Maestría en Ingeniería - Automatización Industrial

Director:
Prof. Germán Castellanos-Domínguez, Ph.D.

Codirector:
Prof. Andrés Marino Álvarez-Meza, Ph.D.

Grupo de investigación:
Grupo de control y procesamiento digital de señales (GCPDS)

Universidad Nacional de Colombia
Facultad de Ingeniería y Arquitecura
Departamento de Ingeniería Eléctrica, Electrónica y Computación
Manizales, Colombia
2022

ACKNOWLEDGEMENTS

A very special thanks goes out to my parents who always believed in the power of education.

I would like to express my gratitude to the professors Andrés Marino Álvarez Meza and Germán Castellanos Domínguez for their orientation during this research. Besides, I would like to thank all the Signal Processing and Recognition Group (SPRG) of the Universidad Nacional de Colombia sede Manizales for their suggestions and hours of academic discussion, to David Cardenas Peña and the Automatic research group of the Universidad Tecnológica de Pereira for allow me to work in so interesting projects. Finally, special tanks to Vanessa Cañaveral for her revisions and corrections about the redaction of this document.

Finally, I recognize that this research would not have been possible without the support given by the project: *Caracterización Morfológica de Estructuras Cerebrales por Técnicas de Imagen para el Tratamiento Mediante Implantación Quirúrgica de Neuroestimuladores en la Enfermedad de Parkinson* (*código 110180763808*), funded by MINCIENCIAS.

Yeison Nolberto Cardona Álvarez

2022

ABSTRACT

The widespread use of neurophysiological signals to develop brain-computer interface (BCI) systems has certainly varied clinical and nonclinical applications. Main implementations in medical issues include: rehabilitation, cognitive state analysis, diagnostics, assistive devices for communication, locomotion and movement. By other hand, there is a bunch of researches that approaches the BCI systems to healthy people in fields like: neuroergonomics, smart homes, neuromarketing and advertising, games, education, entertainment and even security and validation. Not all EEG acquisition systems are capable to use in BCIs systems. Even if the clinic devices are highly accurate, these implementations have a limited, or nonexistent, real-time data flow access; because they mainly use is about diagnostic and offline analysis. Recently, and because of the cheapening prototyping development, there is in the market a set of low-cost embedded systems for electroencephalography (EEG) acquisition, i.e., OpenBCI, InteraXon, Muse, NeuroSky MindWave and Emotiv. All these options usually include a high or low-level software development kit (SDK), that could be open-source or proprietary and will come with a different grade of flexibility (rigid or customizable electrode placement, multiple sampling rates, transmission protocols, wireless, etc). Many of these devices have shown capabilities to handle BCI tasks, but they need a context-specific development to boost their base benefits. Acquiring brain signals is only one task for a BCI system, also it is necessary to carry out a lot of data processing and controlled experiments,

concerning this have been specialized software for developers and researchers purpose i.e., BCI2000, Neurobehavioral Systems Presentation, Psychology Software Tools, Inc. ePrime and PsychoPy. All these systems offer greater ease of use through experimenter interfaces, but they can be costly, require high-level programming and technical skills, and usually do not support dedicated data acquisition. For this reason, the acquisition involves the implementation of third party software and drivers; consequently, losing interesting hardware features in favor to support as many devices as possible. To implement a BCI system is an interdisciplinary activity that requires a set of specific and outstanding knowledges about communication systems, signals acquisition, instrumentation, clinical protocols, experiments validation, software development, among others.

Besides, in order to perform a real-world experiment, the user must calibrate the specific set of acquisition system, stimuli delivery and data processing stages. Current software approaches try to converge multiple technologies and methodologies to provide general purpose BCI systems. The most popular is the BCI200, which comes with default paradigms but their interface has been pointed out to be not very intuitive and its operation is difficult to understand, although, it is possible to add new paradigms, this include software contributions using their own libraries and do not through a built-int development interface. Other software widely used is the OpenVIBE this one includes a graphical drag-and-drop interface to perform data analysis with an extensive set of pre-defined algorithms. Its synchronous acquisition system is known for not only occasionally frozen the computer but also for adding delays to the streaming of the signals. All these systems handle with an extensive set of compatible devices which may be good at first glance but make that some specific hardware features are not available for compatibility reasons. On the side of the open source hardware, we can find that OpenBCI a flexible option, but with some important lacks. The most important relies on the communication between the computer and the board is not always stable and their graphical user interface (GUI) does not provide the possibility of acquiring data under wich a particular BCI paradigm. Otherwise, their hardware

base and SDK features gives to this board a huge potential to implement a complete BCI system comparable with medical grade equipment.

With all these factors in mind, we aim to develop a standalone BCI system with the OpenBCI Cyton board that handles the signal acquisition and the stimuli deliver in the same interface, to reduce the needed infrastructure to perform neurophysiological experiments. Alongside a distributed platform to improve the performance, increase the scalability, and reduce the *jitter*. This software, BCI-Framework, provides the user with a built-in development environment enhanced with a custom API for data interactions, montage context, and markers generation. This environment is full compatible with any Python module and is focused in the generation of real-time visualizations, data analysis and network-based stimuli delivery for the remote presentation of audiovisual cues. This approach converges almost all needed components for BCI researches into a single standalone implementation.

In a nutshell, the introduced EEG-based BCI framework comprises the following benefits: i) A portable and cheap acquisition system (hardware) founded on the well-known OpenBCI devices. ii) This approach includes a wireless, e.g., Wi-Fi, communication protocol to couple the EEG data acquisition and event markers synchronization from audiovisual stimulation paradigms. iii) A distributed system is enhanced within this BCI framework to carry out real-time data acquisition and visualization while favoring the inclusion of conventional or user-designed EEG data processing libraries over a Python language environment. In addition, a latency-based quality assessment method is carried out.

Keywords: Brain-Computer Interface, Signals acquisition, Neurophysiological experiments, Distributed systems, Embedded systems, OpenBCI.

RESUMEN

El uso generalizado de señales neurofisiológicas para desarrollar sistemas BCI ciertamente tiene diversas aplicaciones clínicas y no clínicas. Las principales implementaciones en temas médicos incluyen: rehabilitación, análisis del estado cognitivo, diagnóstico, dispositivos de asistencia para la comunicación, locomoción y movimiento. Por otro lado, hay muchas investigaciones que acercan los sistemas BCI a personas sanas en campos como: neuroergonomía, hogares inteligentes, neuromarketing y publicidad, juegos, educación, entretenimiento e incluso seguridad y validación. No todos los sistemas de adquisición de EEG se pueden usar en los sistemas BCIs. Incluso si los dispositivos clínicos son muy precisos, estas implementaciones tienen un acceso limitado o inexistente al flujo de datos en tiempo real; debido principalmente a que se tratan sistemas enfocados al diagnóstico y análisis fuera de línea. Recientemente, y debido al abaratamiento del desarrollo de prototipos, existe en el mercado un conjunto de sistemas embebidos de bajo costo para la adquisición de EEG, algunos de ellos son: OpenBCI, InteraXon, Muse, NeuroSky MindWave y Emotiv. Todas estas opciones suelen incluir un SDK de nivel alto o bajo, que puede ser de código abierto o privativo los cuales vienen con un grado diferente de flexibilidad (disposición de electrodos rígida o personalizable, frecuencias de muestreo variable, diferentes protocolos de transmisión, conexión inalámbrica, etc). Muchos de estos dispositivos han demostrado capacidades para manejar tareas BCI, pero necesitan un desarrollo específico del contexto para aumentar sus beneficios básicos.

Adquirir señales cerebrales es sólo una tarea indiidual para un sistema completo de BCI, también es necesario llevar a cabo una gran cantidad de procesamiento de datos y experimentos controlados, con respecto a esto se ha especializado software para desarrolladores e investigadores, por ejemplo: BCI2000, Neurobehavioral Systems Presentación, Psychology Software Tools, Inc. ePrime y PsychoPy. Todos estos sistemas ofrecen una mayor facilidad de uso a través de las interfaces del sistema de experimentos, pero pueden ser costosos, requieren habilidades técnicas y de programación de alto nivel y por lo general, no admiten la adquisición de datos dedicada. Por esta razón, la adquisición de señales se basa en la implementación de software y controladores de terceros; en consecuencia, se pierden características de hardware interesantes a favor de soportar tantos dispositivos como sea posible. Implementar un sistema BCI es una actividad interdisciplinaria que requiere un conjunto de conocimientos específicos y sobresalientes sobre sistemas de comunicación, adquisición de señales, instrumentación, protocolos clínicos, validación de experimentos, desarrollo de software, entre otros.

Además, para realizar un experimento del mundo real, el usuario debe calibrar el conjunto específico de sistema de adquisición, entrega de estímulos y etapas de procesamiento de datos. Los enfoques de software actuales intentan hacer converger múltiples tecnologías y metodologías para proporcionar sistemas BCI de propósito general. El más popular es el BCI200, que incorpora paradigmas predeterminados pero se ha señalado que su interfaz es poco intuitiva y su funcionamiento es difícil de entender, aunque es posible agregar nuevos paradigmas, esto permite incluir contribuciones de software utilizando sus propias bibliotecas. y no mediante una interfaz de desarrollo integrada. Otro software ampliamente utilizado es OpenVIBE, este incluye una interfaz gráfica de arrastrar y soltar para realizar análisis de datos con un amplio conjunto de algoritmos predefinidos. Su sistema de adquisición sincrónica es conocido no sólo por congelar ocasionalmente la computadora, sino también por agregar retrasos en la transmisión de las señales. Todos estos sistemas manejan un amplio conjunto de

dispositivos compatibles que pueden ser buenos a primera vista, pero hacen que algunas características específicas del hardware no estén disponibles por razones de compatibilidad. Del lado del hardware de código abierto, podemos encontrar que OpenBCI es una opción flexible, pero con algunas carencias importantes. La más importante se basa en que la comunicación entre la computadora y la placa no siempre es estable y su GUI no brinda la posibilidad de adquirir datos bajo un paradigma BCI particular. Por otro lado, su base de hardware y las características de SDK le dan a esta placa un gran potencial para implementar un sistema BCI completo comparable con el equipo de grado médico.

Con todos estos factores en mente, nuestro objetivo es desarrollar un sistema BCI independiente con la placa OpenBCI Cyton que maneje la adquisición de señales y la entrega de estímulos en la misma interfaz, para reducir la infraestructura necesaria para realizar experimentos neurofisiológicos. Junto con una plataforma distribuida para mejorar el rendimiento, aumentar la escalabilidad y reducir el *jitter*. Este software, BCI-Framework, proporciona al usuario un entorno de desarrollo integrado mejorado con una API personalizada para interacciones de datos, selección de montaje y generación de marcadores. Este entorno es totalmente compatible con cualquier módulo de Python y se centra en la generación de visualizaciones en tiempo real, análisis de datos y entrega de estímulos a través de conexiones de red para la presentación remota de señales audiovisuales. Este enfoque reúne casi todos los componentes necesarios para la investigación de BCI

En pocas palabras, el sistema BCI basado en EEG presentado comprende los siguientes beneficios: i) Un sistema de adquisición portátil y económico (hardware) basado en los conocidos dispositivos OpenBCI. ii) Este enfoque incluye un protocolo de comunicación inalámbrico (Wi-Fi), para acoplar la adquisición de datos de EEG y la sincronización de marcadores de eventos de paradigmas de estimulación audiovisual. iii) Se implementa un sistema distribuido dentro de este entorno BCI para llevar a cabo la adquisición y visualización de datos en tiempo

real mientras se favorece la inclusión de bibliotecas de procesamiento de datos EEG convencionales o diseñadas por el usuario sobre un entorno de lenguaje Python. Además, se lleva a cabo método para la evaluación de la calidad basada en la latencia.

Palabras clave: Interfaces Cerebro-Computador, Adquisición de señales, Experimentos neurofisiológicos, Sistemas distribuidos, Sistemas embebidos, OpenBCI.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ADC analog-to-digital converter 42

API application programming interface xviii, 26, 55, 59, 87, 90

BCI brain-computer interface ix–xi, xiii–xv, xvii, xviii, 1–7, 9–16, 18, 26, 32, 55, 56, 86, 90, 118

CLI command-line interface 21, 26

ECG electrocardiography 10, 18, 89

EEG electroencephalography ix, xiii, 2–4, 6, 7, 10, 15, 16, 18, 25, 61, 89

EMG electromyography 10, 18, 89

ERP event-related potential 6, 65

FFT Fast Fourier transform 61

FPGA field-programmable gate array 15

GUI graphical user interface x, xv, 5, 56

HDF hierarchical data format 32, 33

IDE integrated development environment 16

JSON JavaScript object notation 28

LDR light-dependent resistor 85

MI Motor imagery xxii, 117–122

MQTT message queue telemetry transport 26

RPyC remote Python call 21, 40, 41, 52

RSST Reward Stop Signal Task xxii, 129

RTP real-time protocol 41

SBC single-board computer 16, 41

SDK software development kit ix, xi, xiii, xv, 5, 7, 22, 27

SPRG Signal Processing and Recognition Group 3, 91

SPS samples per second 10, 26, 41

TCP transmission control protocol 10, 23, 26

UX user experience 22

VWM Visuospatial Working Memory xxii, 123–126

CHAPTER

ONE

INTRODUCTION

1.1 Motivation

A brain-computer interface (BCI) is a hardware and software communication system that enables cerebral activity alone to control computers and external devices [1]. The widespread use of neurophysiological signals to develop BCI systems has certainly varied clinical and non-clinical applications. Main implementations in medical issues include rehabilitation, cognitive state analysis, diagnostics, and assistive devices for communication, locomotion, or movement. On the other hand, there is significant research that approaches the BCI systems to healthy people in fields like neuroergonomics [2], smart homes [3], neuromarketing and advertising [4], games [5], education [6], entertainment [7], security and validation [8].

In order to connect the brain to external devices, two types of brain activities can be monitored: electrophysiological and hemodynamics. Electrochemical

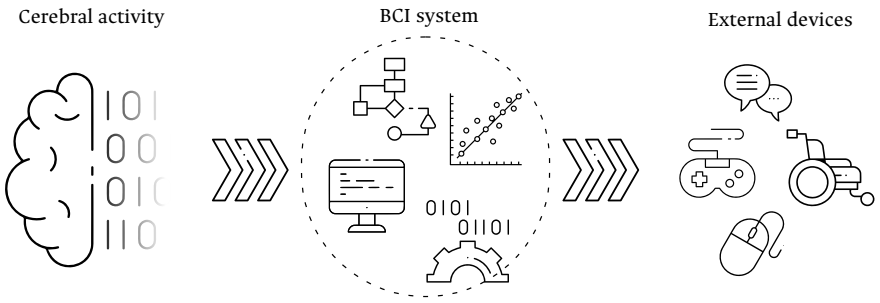


Figure **1-1**. The purposed aims in this works contribute with the implementation of a integral application, beyond that, the synergy between this characteristic allow the achievement of advance features that merge the acquisition with the stimuli delivery in a flexible development environment.

transmitters exchanging information between the neurons generate the electrophysiological signals [9]; electroencephalography (EEG), electrocorticography, magnetoencephalography, and electrical signal acquisition in single neurons are the techniques used to measure these activities. The hemodynamics response is a process in which the blood releases glucose to active neurons [10]; neuroimaging methods can quantify these changes, such as functional magnetic resonance and near-infrared spectroscopy. Yet, EEG is the most common method to get relevant information from the brain activity in BCIs systems, due to its high temporal resolution, relatively low cost, high portability, and few risks to the users [11]. Figure **1-1** shows how a BCI system works like a brain transducer.

However, although BCI systems could require high demanding computational resources [12], it is possible to build a capable system that works with EEG, skin-surface electrodes, and low-cost embedded acquisition devices. Furthermore, using free software and open-source resources, with the correct selection and integration of these components and focusing on the improvement of the signal acquisition, may consequently achieve clinical validation, effective dissemination models, and probably most importantly, increased reliability. Then, BCIs aim to become an important new technology for people with disabilities and, possibly, the general population [13].

In a local context, the Signal Processing and Recognition Group (SPRG) of the Universidad Nacional de Colombia have been working on the analysis of neurophysiological data to propose and develop machine learning methodologies for the assisted diagnosis of mental conditions [14, 15], automated analysis of human activity recognition [16], and biomedical data analysis [17]. More recently, SPRG have shown an interest in working with their own databases instead of using public domain ones in a variety of research projects (supported by Minciencias, Dirección Nacional de Investigaciones de Manizales (DIMA), and Vicerrectoría de Investigaciones de la Universidad Nacional de Colombia):

- Herramienta de apoyo al diagnóstico del TDAH en niños a partir de múltiples características de actividad eléctrica cerebral desde registros EEG.
- Desarrollo de un sistema integrado de monitoreo de actividad cerebral a partir de registros EEG en pacientes bajo anestesia general para ambientes quirúrgicos.
- Prototipo de interfaz cerebro-computador de bajo costo para la detección de patrones relevantes de actividad eléctrica cerebral relacionados con TDAH.
- Prototipo de interfaz cerebro-computador multimodal para la detección de patrones relevantes relacionados con trastornos de impulsividad.
- Interfaz cerebro-computador basada en aprendizaje de máquina y teoría de información como soporte a la detección de trastornos de déficit de atención e hiperactividad.
- Brain Music: Prototipo de interfaz interactiva para generación de piezas musicales basado en respuestas eléctricas cerebrales y técnicas de composición atonal.

The achievement of an independent BCI software with an environment for developers and researchers that integrates an interface to design their custom visualization and neurophysiological experiments, which also handles the signals acquisition, synchronizes markers, and automatically creates ready-to-use databases, would comprise a very useful tool for the SPRG. It would also greatly simplify the testing and designing of BCI systems, guarantee better repeatability, reduce failure points, and speed up the debugging process.

1.2 Problem statement

Implementing a BCI system is an interdisciplinary activity that demands specific and outstanding knowledge about communication systems, signals acquisition, instrumentation, clinical protocols, experiment validation, and software development [18]. A BCI software that functions properly can be easily adapted to different experimental situations and can facilitate the operation of entire research programs rather than the execution of an individual study. Therefore, the premise of existing open-source or commercial BCI software is to reduce this complexity, difficulty, and cost [19].

Including "open" components increases the technology acceptance [20], reduces costs, enables collaborative development, and impulses a community working around BCI to extend this machinery for the general population [13]. The main issues about BCI development systems lie around (i) the EEG acquisition system used for BCI differs from standard or medical implementations. (ii) The specialized software is hard to modify according to specific needs, and (iii) the highly variational computational cost leads to the development of unstable and difficult-to-scale applications.

1.2.1 Acquisition system requirements

Not all EEG acquisition systems are capable of using BCIs systems. Even if the clinic devices are highly accurate, these implementations have limited, or nonexistent, real-time data flow access; due to their primary use being diagnostic and offline analysis [21]. Overall, low-cost EEG headsets show greater design convenience, as the portability, for "real world" occupational use and capabilities to handle BCI tasks with varying degrees of success [22]. However, open-source software and occupational refinement may boost the potential of these systems.

Besides, there exists a need to implement a context-specific development to improve their base features [23].

As shown in Table 1-1, there is a complete set of devices, but only one of the listed hardware is "open-source"; therefore, its freedom makes it possible to modify the hardware and also access the firmware. Then, with all these options, it is possible and needed to develop custom drivers with high-level interactions. Even if the OpenBCI is the most featured option, the communication between the computer and the board is not always stable [24], and their graphical user interface (GUI) does not provide the possibility of acquiring data under a particular BCI paradigm. Otherwise, their hardware base and software development kit (SDK) features give this board an enormous potential to implement a complete BCI system comparable to medical-grade equipment [25].

1.2.2 BCI software development issues

As for the software capable of handling BCI implementation, a few of them are independent, but in almost all cases, at least a couple is needed to perform a complete BCI paradigm. Some will perform only acquisition, others will include data processing, and a few will have stimuli delivered integrated with the main interface [26]. The main reason to use over one software is that many of these tools, even when they can be helpful, are not strictly for BCI but behavioral sciences, neuroscience, psychology, psychophysics, or linguistics.

Proprietary software compromises the extensibility[1] of its tools by limiting data transmission protocols or creating a close list of compatible hardware [27, 28, 29]. Although all these software offer greater ease of use through experimenter interfaces, they can be costly. The open-source options still require high-level programming and technical skills, and usually, no one supports dedicated data acquisition [19].

[1] Extensibility is the ability of the software system to allow and accept the significant extension of its capabilities without major rewriting of code or changes in its basic architecture.

1.2.3 Computational cost

The three most important tasks of a BCI system are signal acquisition, feature extraction and classification, and command translation or mapping [30]. These tasks demand a high-performance computer running many processes under non-real-time operating systems. Distributing those three highly CPU resources consuming processes in a distributed system will reduce the computing complexity of a BCI framework, thus increasing the reliability of overall system performance [30]. In a standard EEG-based medical experiment, there are at least three components of data abstraction working simultaneously: data acquisition, signal database, and signal visualization [31, 32]. But in a close-loop BCI system, at least seven main components are required to be synchronized: data acquisition, signals database/storage, feature processing (extraction and classification), visualization (temporal or spatial), command generation for actuators, command database, and feedback acquisition [30].

BCI systems usually do not run under a real-time operating system which means that the use of the resources will affect each component of the system. Some paradigms in the event-related potential event-related potential (ERP) need high precision for marker synchronization. For this purpose, the latency must be not only at low levels but also with small variabilities. This additional measure is called jitter. Both tell about the stability of a system and the capacity to handle more processes. Under some environments like the researching, centralized systems are susceptible to being slow down because of unexpected processing costs [33, 34].

Therefore, some problems related to the agile acquisition of EEG signals under multiple paradigms remain unsolved. For this reason, the following research question arises: how to develop an independent EEG-based BCI monitoring framework that integrates real-time acquisition and visualization from audiovisual stimulation paradigms using OpenBCI?

1.3 State-of-the-art BCI systems

1.3.1 BCI hardware

Recently, and because of the cheapening prototyping development, a set of low-cost embedded systems for EEG acquisition has appeared in the market. All these options typically include an SDK which could be open-source or proprietary: this embedded device commonly adds some flexibility (rigid or customizable electrode placement, multiple sampling rates, transmission protocols, wireless, among others).

Many EEG acquisition systems have been developed in recent years; however, to name only the most significant ones, a selection criterion was taken into account to compare: portable devices that are still available in the market with a stable and substantial base of active users.

Montages and electrodes placement

Table 1-1 compiles the features of the most relevant acquisition systems. Regarding electrode placement, devices with rigid placements are related to simple neurophysiological activities like concentration, drowsiness, stress, and Approach-Withdrawal pleasantness. In most cases, a task that only needs booleans or low-frequency data transmission is provided: these systems will not return the raw EEG data, but will return trends for only a few channels (*InteraXon Inc. Muse*[2], *IMEC EEG Headset*[3], or *NeuroSky Mind Wave*[4]). Other devices concentrate their distributions in the sensory-motor brain areas to perform motor-imagery tasks (*Emotive EPOC+*[5] or *B-Alert x10*[6]). Portable devices usually support wireless

[2] https://choosemuse.com/
[3] https://www.imec-int.com/en/eeg
[4] https://store.neurosky.com/pages/mindwave
[5] https://www.emotiv.com/epoc/
[6] https://www.advancedbrainmonitoring.com/products/b-alert-x10

BCI hardware	Electrode types	Channels	Protocol and Data transfer	Sampling rate	Open hardware
Enobio	Flexible / Wet	8, 20, 32	BLE	250 Hz	No
q.DSI 10/20	Flexible / Dry	21	BLE	250 Hz - 900 Hz	No
NeXus-32	Flexible / Wet	21	BLE	2.048 KHz	No
IMEC EEG Headset	Rigid / Dry	8	BLE	????	No
InteraXon Inc. Muse	Rigid / Dry	5	BLE	220 Hz	No
Emotive EPOC+	Rigid / Wet	14	RF	128 Hz	No
Cognionic CGX MOBILE	Flexible / Dry	72, 128	BLE	500 Hz	No
Biosemi ActiveTwo	Flexible / Wet	256	USB	2 KHz - 16 KHz	No
actiCAP slim/snap	Flexible / Wet / Dry	16	USB	2 KHz - 20 KHz	No
NeuroSky Mind Wave	Rigid / Dry	1	RF	250 Hz	No
Cognionic Quick-20	Rigid / Dry	28	BLE	262 Hz	No
B-Alert x10	Rigid / Wet	9	BLE	256 Hz	No
OpenBCI	Flexible / Wet / Dry	8, 16	RF/BLE/Wi-Fi	250 Hz - 16 KHz	Yes

Table **1-1**. Popular acquisition devices used for BCI systems.

data transmission, Bluetooth (*Enobio*[7], *q.DSI 10/20*[8], *NeXus-32*[9], *Cognionic CGX MOBILE*[10]), radiofrequency, or Wi-Fi. Only the wired and the Wi-Fi ones are capable of handling data transmission over 1 kHz. These kinds of transmissions are also associated with a large number of channels (*OpenBCI*[11], Cognionic Quick-20[12], *ActiCap*[13] or *Biosemi ActiveTwo*[14]).

It is possible to conclude that, even if the rigid electrode placements can be used for some BCI experiments, the flexible electrode placement is the best option for general BCI. Also, devices that do not include a universal electrode attachment provide standard montages, like the 10-20 one, and additionally, wired systems are related to high electrode density. Lastly, few "open" devices show success in the market.

Licensing and freedoms

Licensing is one of the most important features when considering the inclusion of hardware in a real environment. There are mainly three options: first, close hardware with close restrictive licenses; in these cases, the developments can not be redistributed, commercialized, or even shared for repeatability experiments [**?**]. Second, close hardware with open licenses, which allows developers to build, modify and share a complete acquisition system [**?**]. Finally, and similar to the second case, a device that can be completely open means that there are no restrictions on the development, and all builds, configurations, and modifications can be shared.

So, a real-world solution generally includes "open" components to increase the technology acceptance [35, 36], reduce costs, enable collaborative development,

[7] https://www.neuroelectrics.com/solutions/enobio
[8] http://www.quasarusa.com/products$_dsi.htm$
[9] https://www.biofeedback-tech.com/nexus-32
[10] https://www.cgxsystems.com/mobile-128
[11] https://openbci.caom/
[12] https://www.cgxsystems.com/quick-20m
[13] https://brainvision.com/products/acticap-slim-acticap-snap
[14] https://www.biosemi.com/products.htm

and impulses a community working around BCI to extend this machinery for the general population [13].

OpenBCI acquisition system

On the open-source hardware side, we can find that *OpenBCI* is one of the most flexible options [37]. This board not only works with EEG but is also suitable for electromyography (EMG) and electrocardiography (ECG). The *OpenBCI Cyton*[15] biosensing board comprises a *PIC32MX250F128B* microcontroller, a *ChipKIT UDB32-MX2-DIP bootloader*, a LIS3DH 3-axis accelerometer, and an *ADS1299* analog-to-digital converter with 8 input channels (expandable to 16) up to a sampling rate of 16 kHz. Notably, EEG channels can be configured as monopolar and bipolar (and, consequently, sequentially) with up to five external digital inputs and three analog inputs. Also, the data flow is accessible through a Wi-Fi interface using transmission control protocol (TCP).

Table **1-2** summarizes *OpenBCI* main configurations. *RFduino*, by default, supports 250 samples per second (SPS) and 8 channels, but with the *Daisy* addition, it can expand up to 16 channels, and, with the *Wi-Fi shield*, the sample rate can increase up to 16 kHz. All channels can be configured as monopolar, bipolar, and sequential.

The *OpenBCI* Cyton board used to have Python-compatible drivers[16], but now these are deprecated in favor of a new family of drivers board agnostic, *BrainFlow*[17]. The main reason to develop board-first drivers resides in taking advantage of all low-level features and integrating them into the final drivers through high-level board configurations.

[15] https://openbci.com/
[16] https://github.com/openbci-archive/OpenBCI_Python
[17] https://brainflow.org/

OpenBCI Cyton	Channels	Digital inputs	Analog inputs	Max sample rate	Featured protocol
RFduino	8	5	3	250 Hz	Serial
RFduino + Daisy	16	5	3	250 Hz	Serial
RFduino + Wi-Fi shield	8	2	1	16 KHz	TCP (over Wi-Fi)
RFduino + Wi-Fi shield + Daisy	16	2	1	8 KHz	TCP (over Wi-Fi)

Table **1-2**. OpenBCI Cyton configurations using Daisy expansion board and Wi-Fi shield.

1.3.2 BCI software

Acquiring brain signals is only one task for a BCI system. It is also necessary to carry out a lot of data processing and controlled experiments. Therefore, a specialized software segment persists for developers and researchers that offer tools for this or similar purposes. In this case, non-BCI software refers to tools that are not designed for BCI environments, but integrate features that they can also use, usually related to neuroscience in general. Conversely, there is a set of applications that integrate useful features for BCI, such as data processing and, sometimes, close-loops.

Table **1-3** summarizes the most common systems used for BCI where we can see four big groups: (i) the ones with stimuli delivery and data analysis are designed for close loops like *BCI2000*[18] *OpenViBE*[19] and *Neurobehavioral Systems Presentation*[20]; (ii) others have the same features, but their implementation is not focused on closing the loop like *ePrime*[21], *OpenSesame*[22], and *g.BCISYS*[23]; (iii) and a few of them

[18] https://www.bci2000.org/
[19] http://openvibe.inria.fr/
[20] https://www.neurobs.com/
[21] https://pstnet.com/products/e-prime/
[22] https://osdoc.cogsci.nl/
[23] https://www.gtec.at/product/bcisystem/

that only include data analysis, typically the Matlab toolboxes like *EEGLAB*[24] and *FieldTrip*[25]; (iv) some neuropsychology tools, since this is a precision task, only have in their interfaces a stimuli delivery like *Millisecond Inquisit Lab*[26] *Pychotoolbox-3*[27], *MonkeyLogic*[28], and *PychoPy*[29]. Additionally, there exist multiple tools that are not mentioned here, except *OpenBCI GUI*[30], which like this one, only works as a demo interface for their main hardware products.

All previous systems have two additional features that share no relation with the main groups of focused users or the final implementation: the extensibility, the ability to create, modify and run custom experiments instead of the included by default, and the license distribution. However, these features are related to them. Free (as in freedom) licenses usually give the user the tools to not only configure but rebuild the software itself. A system that does not offer the user the capability for extensibility is generally a proprietary one.

Non-BCI related software

For the cases where extensibility is a needed feature, some BCI implementation uses a set of applications simultaneously. Diverse psychology software can generate a stimulus to use in BCI paradigms; certainly, there are really fast and precise tools in this field, with graphical interfaces for easy design of experiments and audiovisual stimulation. Additionally, other types of software can be used to process data in real-time, which can be done through a private scripting interface, toolboxes, or libraries. Also, some programming languages like *Matlab* [38, 39] and *Python* [40] host dedicated environments to handle the processing necessary for BCI.

[24] https://sccn.ucsd.edu/eeglab/index.php
[25] https://www.fieldtriptoolbox.org/
[26] https://www.millisecond.com/products/inquisit6/laboverview.aspx
[27] http://psychtoolbox.org/
[28] https://www.brown.edu/Research/monkeylogic/
[29] https://www.psychopy.org/
[30] https://github.com/OpenBCI/OpenBCI$_GUI$

BCI software	Stimuli delivery	Devices	Data analysis	For close-loop	Extensibility	License
BCI2000	Yes	A large set	In software	Yes	yes	GPL
OpenViBE	Yes	A large set	In software	Yes	Yes	AGPL-3
Neurobehavioral Systems Presentation	Yes	Has official list	In software	Yes	Yes	Proprietary
Psychology Software Tools, Inc. ePrime	Yes	Proprietary devices only	In software	No	Yes	Proprietary
EEGLAB	No	Determined by Matlab	System Matlab	No	-	Proprietary
PsychoPy	Yes	NO	NO	No	Yes	GPL
FieldTrip	No	NO	System Matlab	No	Yes	GPL
Millisecond Inquisit Lab	Yes	Serial and parallel devices	NO	No	No	Proprietary
Psychtoolbox-3	Yes	Determined by Matlab and Octave	NO	No	-	MIT
OpenSesame	Yes	Determined by Python	System Python	No	Yes	GPL
MonkeyLogic	Yes	Determined by Matlab	NO	No	No	Proprietary
g.BCISYS	Yes	Proprietary devices only	System Matlab	No	No	Proprietary
OpenBCI	No	Proprietary devices only	No	No	Yes	MIT

Table **1-3**. Most widely used software for implementing BCI systems.

BCI as an independent software

An independent software comprises the complete process of a BCI implementation: this goes from data acquisition to real-world command generation. A delivery feature is also included in its interfaces, regardless of whether the focus is on research or production; usually, these systems do not include the interpretation of commands in the real world and depend on third-party systems for the markers synchronization. It is also common that the interface for the acquisition is a part of the stimuli delivery. On the other hand, there are a few tools designed specifically for BCI, like *BCI2000* [41] and *OpenViBE* [42], which are available under permissive licenses and have an active community of developers and users working with both of them. It is then up to the community to manage the incorporation of new acquisition systems and custom paradigms, even if the main interface does not support extensibility.

The most popular tool focused on BCI is *BCI200* which comes with default paradigms; however, it has been pointed out that its interface is not very intuitive and its operation is difficult to understand initially [43], and although it is possible to add new paradigms, this one must include software contributions using their own libraries and not through a built-int development interface. Another software widely used is *OpenViBE*, which features a graphical drag-and-drop interface to perform data analysis with an extensive set of pre-defined algorithms; nevertheless, its synchronous acquisition system is known for occasionally freezing the computer and adding delays to the signal streaming [24]. It is easy to point out that high-level tools require a lot of computational resources.

Marker synchronization

For both cases, BCI and non-BCI implementations, the data acquisition requires to be synchronized with the stimuli delivery. Multiple techniques are described [44] to make these measures in the instances where it is possible. In most cases, specialized laboratory equipment is necessary [45, 46] to calculate latencies and

make corrections offline. This approach satisfies the database generations, and for real-time analysis, the system must be calibrated with a previously measured latency. This measurement is not constant and will depend on the operating system and may even be correlated with processing techniques, so the protocol to measure latencies must be executed after minimal system changes. Other options include creating control signals [47] and performing corrections in real-time. This choice will only work on systems that can acquire external signals in conjunction with the EEG channels.

1.3.3 Real-time and computational cost handling

At present, most existing systems use a high-quality algorithm to train the data offline and run only the classification in real-time, so much work has been done to reduce the computational cost for processing signals to use in BCI [48, 49]. A constant discussion turns around the computational cost vs. accuracy [50] because, along with new processors, new costly analysis techniques also emerge. Therefore, low-cost computational methods are always an aim [51].

Offline analysis may not represent significant challenges unless the amount of data to process is high. Although cloud-based strategies (e.g., *NeuroCAAS* [52]) are practical, they are not suitable for the real-time requirement of the BCI systems.

Real-time processing of large neural data streams has become workable thanks to advances in computer processing power, electronics such as microprocessors and field-programmable gate arrays (FPGAs), and specialized and open-source software [53]. In the neuroscience field, *RTBiomanager* [54] was designed to explore the use of real-time technology to build a set of novel experiments that combine different recording and stimulation techniques. In a similar context, purposes like *RTHybrid* [55] are not only efforts on real-time stimuli delivery precision but also standardization methods.

All tools for real-time computational in the field of neuroscience and BCI are focused on the processing instead of the acquisition, the main reason is due the separation of the specialized software. Other approaches to improve the processing are related with the optimisation instead of the distributed computing. Then, is required a framework that allows the flexible implementation of distribute computing, for the design and the development of BCI systems.

1.4 Aims

1.4.1 General aim

To develop an EEG-based BCI monitoring framework with real-time acquisition and visualization for audiovisual stimulation paradigms using *OpenBCI*, focused on designing, performing, and validating BCI systems to conduct experiments in all stages: design, testing, and production.

1.4.2 Specific aims

- To implement a cross-platform library for *OpenBCI* hardware that allows distributed functionalities, low-level board configurations, an acquisition protocol, data storage, and external inputs handler.
- To implement a distributed computing paradigm that allows managing acquisition boards, acquiring EEG signals through a network, synchronizing markers, measuring latencies, and delivering stimuli experiments into a decentralized environment using a single-board computer (SBC) scheme.
- To develop an independent interface with an integrated development environment (IDE) featured to configure the acquisition system, design real-time visualizations, execute timelock analysis, and perform stimuli delivery.

1.5 Outline and contributions

In the following, we briefly introduce the main contributions of this thesis. They are summarized in Figure 1-2

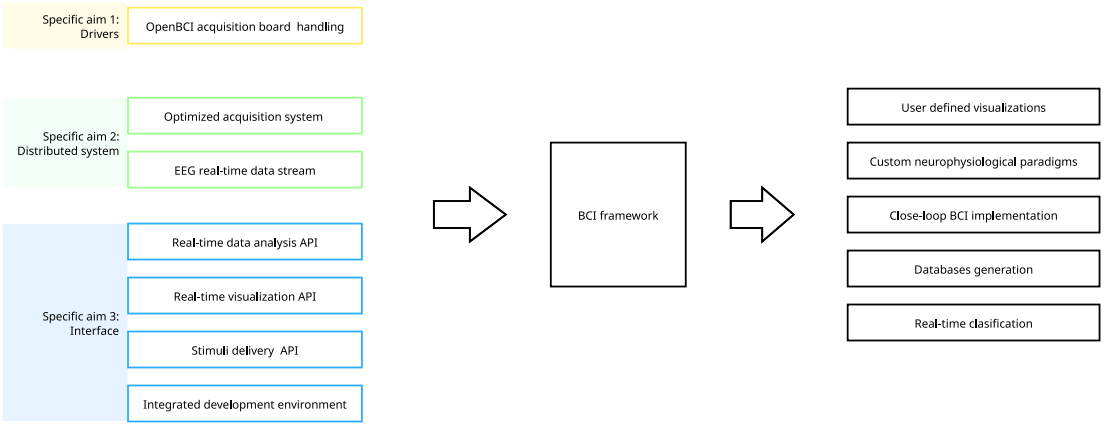


Figure **1-2**. The purposed aims in this works contribute with the implementation of a integral application, beyond that, the synergy between this characteristic allow the achievement of advance features that merge the acquisition with the stimuli delivery in a flexible development environment.

A complete source code for the drivers OpenBCI-Stream[31] and BCI-Framework[32] are available in the public research group repository.

1.5.1 High-level acquisition drivers for OpenBCI

OpenBCI Cyton is the most promising hardware to implement signals acquisition due to its low-level features and open-source design. However, it is significantly lacking in drivers, there are no dedicated tools to handle acquisition signals, and the current approach does not take advantage of all hardware capabilities. A

[31] https://github.com/UN-GCPDS/openbci-stream
[32] https://github.com/UN-GCPDS/bci-framework

qualified and research-grade tool for BCI implementations must gather a set of features related to the quality of the signal and the reliability of the data acquired; this can be done by implementing subroutines such as impedance measurement and marker synchronization, respectively.

Bearing this in mind, we proposed to develop *OpenBCI-Stream*, a high-level Python module for EEG/EMG/ECG acquisition, and distributed streaming for *OpenBCI Cyton* boards. This development is related to the first specific aim, and it is described in Chapter 2.

1.5.2 Distributed implementation

The proposed acquisition system is susceptible to be affected by the computational environment, this can affect the sampling and the real-time desired feature. Research environs needs that the computational cost of the data processing not affect the performance of neurophysiological experiments itself. To mitigate this issues, the implementation of the developed system use distributed computing approaches to facilitate the escalation and the integration with custom and third party systems.

Chapter 3 exposes the methods and resources used to built the distributed features alongside a definition of real-time used in this work, also an analysis to measure and compare latencies has been performed for some common acquisition configurations with OpenBCI.

1.5.3 BCI-Framework

The effectiveness of a BCI system, and the ability of users to learn to use it, depends on the system's ability to acquire and process signals, present stimuli in real-time,

and provide the user with consistent feedback with low latency and minimal jitter [44].

Chapter 4 presents BCI-Framework, this is the top-level software that integrates all the components developed in this work. Defines a new integrated approach to works with BCI systems, establish a dynamic and fast method to design custom paradigms, and generate visualizations. Serve to the user a clean development environment with all parameters and the data stream ready to use.

1.6 Thesis structure

The next parts of this thesis is organized as follows. In Chapter 2 we introduce a brand new drivers for OpenBCI that integrates a full features to handle the acquisition board through a Python API implementation. Chapter 3 implements the mentioned drivers under a distributed paradigm for board configuration and data acquisition. Also a set of experiments to measure the latency and performance has been included. Finally, in Chapter 4 all components developed were integrated into a single GUI software called BCI-Framework that also serve a development environment than can be used to design and build custom real-time visualizations and neurophysiological experiments.

CHAPTER

TWO

HIGH-LEVEL ACQUISITION DRIVERS FOR OPENBCI

This chapter exposes the design and development of a driver that comprises a set of scripts that deal with the configuration and connection with the board and is also compatible with both connection modes supported by *OpenBCI Cyton*: the default RFduino (through serial dongle) and Wi-Fi (with the OpenBCI Wi-Fi Shield). The drivers are a stand-alone library that can access the acquisition board from three different endpoints: (i) a command-line interface (CLI) that serves simple instructions to configure, start and stop data acquisition, debug stream status, and register events markers; (ii) a Python Module with high-level instructions and asynchronous acquisition; (iii) a remote object-proxying that uses Remote Python call (RPyC) for distributed implementations.

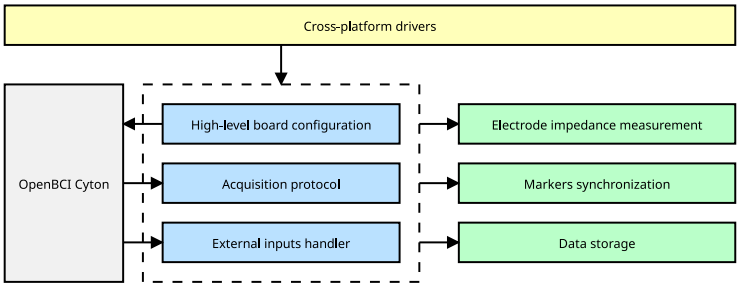


Figure **2-1**. Drivers architecture

2.1 Drivers architecture

A full-featured driver for *OpenBCI Cyton* will need solid basic features related to a high-level board configuration, acquisition protocol, and external inputs handler. These three components are essential for developing generic tasks that almost all BCI systems must implement as impedance measurement, markers synchronization, and data storage, to mention a few. Additionally, the integration of components like flexibility, scalability, and a well-designed user experience (UX) that lets the user focus on the data stream manipulation and consumption, rather than the acquisition and connection, would impact positively on the development of custom implementation that satisfies specific requirements. To achieve this, it must take full advantage of the current and official SDK[1] distributed by *OpenBCI*.

2.1.1 OpenBCI SDK

The *Cyton* board is based on an *ADS1299*, a 24-bit analog-to-digital converter designed by Texas Instruments for bio-potential measurements. *OpenBCI* has built its acquisition system over a *ChipKIT* development board with their respective firmware. The SDK defines an *instruction set* based on Unicode character exchange,

[1] docs.openbci.com/Cyton/CytonSDK/

Byte index	Description
[0]	Header, always `0xa0`
[1]	Sample number
[2:26)	EEG data, values are 24-bit signed, MSB first
[26:32)	Aux data
[32]	Footer, `0xcX` where `X` is 0-F in hex

Table **2-1**. Distribution of the 33-byte of binary package that contain a single array of 8 channels and the auxiliary data.

e. g., to turn *on* channel 1, character `'1'` (equivalent to the integer `49`) must be sent, and to turn it *off*, character `'!'` (equivalent to the integer `33`). This *instruction set* adds additional commands over Wi-Fi mode. The protocols that implement this command-based intercommunication with the board are serial for the USB dongle and TCP for the Wi-Fi interface. Additionally, some commands for the Wi-Fi module are sent to the board through plain HTTP protocol.

The main firmware version used is later than *v3.0.0*, but it is known that the impedance measurement does not work correctly on this version, and although there is a pull request that solves this issue[2], the *OpenBCI* team has done nothing about it. It is possible to use official versions, but the board must be reset every time before the impedance measurement and never change the sampling frequency on run-time; this is the only firmware-level modification, optional but recommended, to use the developed drivers.

Data formats

The type of data to send to the acquisition board is Unicode only; however, the board has three types of responses: Unicode, human-readable strings, and binary.

[2] https://github.com/OpenBCI/OpenBCI$_{C}yton_{L}ibrary/pull/95$

Footer byte [33]	Byte 26	Byte 27	Byte 28	Byte 29	Byte 30	Byte 31	Description
0xc0	AX1	AX0	AY1	AY0	AZ1	AZ0	Standar with accel
0xc1	UDF	UDF	UDF	UDF	UDF	UDF	Standar with raw aux
0xc2	UDF	UDF	UDF	UDF	UDF	UDF	User defined
0xc3	AC	AV	T3	T2	T1	T0	Timestamp *set* with accel
0xc4	AC	AV	T3	T2	T1	T0	Timestamp with accel
0xc5	UDF	UDF	T3	T2	T1	T0	Timestamp *set* with raw aux
0xc6	UDF	UDF	T3	T2	T1	T0	Timestamp with raw aux

Table **2-2**. The type of auxiliar data based on the 33-byte of the binary pakage.

Byte 26	Byte 27
0x58	AX1
0x78	AX0
0x59	AY1
0x79	AY0
0x5a	AZ1
0x7a	AZ0

Table **2-3**. If the 33-byte is 0xc3 or 0xc4 the byte 26 define the component of accelerometer value coded in the byte 27.

There are no issues about the human-readable strings, but there are usually errors and generic responses to commands. The responses on Unicode are for internal registers, i. e., responses that use the same Unicode command to report the actual configuration. The data stream, the package that contains the EEG time series along with the auxiliary data, uses a binary format of 33 bytes.

Table 2-1 shows the distribution of these 33 bytes. The header byte defines the start of the sequence, which is always `0xa0`. The second one is an incremental byte from `0x00` - `0xff`. The third set of 24 bytes contains the value for 8 channels in 24-bit signed format. The fourth set of 6 bytes encodes the auxiliary data. Finally, the footer byte defines the type of auxiliary data streamed. Table 2-2 describes the type of data that the auxiliary data could contain: they are basically accelerometers, external raw inputs, and timestamps. For the case where the timestamp is merged with accelerometer data, Table 2-3 defines the codification mode, *when*, and *where* to find the acceleration value.

The streaming uses different data formats to package the information, 24-bit signed for EEG data, 16-bit signed for accelerometer data, and 32-bit unsigned for timestamps.

2.1.2 Board interface

The board interface refers to the physical way and the protocol used to configure the board and how the data stream is accessed. *OpenBCI* supports two interfaces: Serial and Wi-Fi.

Serial setup

The default connection for *OpenBCI* is called serial because the computer will recognize the board as a serial device; however, it is wireless since the interface includes a USB-serial adapter that uses the proprietary *RFDuino* interface. This

interface is a Bluetooth-modified protocol to achieve the highest data rates. Using this interface, the maximum sample rate for 8 channels is 250. Since the computer recognizes the board as a serial device, communication is based on simple `read(bytes)` and `write(bytes)` commands.

Wi-Fi interface

It is possible to increase the SPS up to 16k with an additional board, the Wi-Fi Shield; nevertheless, these rates are not helpful for BCI purposes; instead, the rates of around 1 or 2 kSPS are easier to handle for the system. Two protocols can be used over Wi-Fi, Message queue telemetry transport (MQTT) and TCP. For simplicity, the TCP was chosen over the other since the MQTT protocol is just a reimplementation of the TCP one.

2.2 Drivers development

The development of the drivers is entirely in Python and is focused on establishing three significant bases: the configuration of the board, the data acquisition, and the external inputs handler.

2.2.1 Application programming interface (API)

The main module is called `openbci_stream` and has three main submodules: `acquisition`, to handle the data stream and the configuration board; `daemons`, to automatize the acquisition based on operating system commands; and `utils`, to accommodate the utility scripts responsible for CLI, filters, storage, and visualizations.

The Cyton board is initialized with the default configurations listed in Table 2-4, so the acquisition can be easily initialized with the following instructions:

Feature	Default
Sample rate	250
Gain	24
Input type set	ADSINPUT_NORMAL
Bias set	Include in BIAS
SRB2 set	Connect this input to SRB2
SRB1 set	Disconnect all N inputs from SRB1
Channels	All on

Table **2-4**. OpenBCI Cyton default booting configuration, all channels active on monopolar configuration.

```
from openbci_stream.acquisition import Cyton

openbci = Cyton('serial', endpoint='/dev/ttyUSB0', capture_stream=True)
openbci.stream(15)  # capture 15 seconds of data
openbci.eeg_time_series # Raw EEG is allocated in this class instance
```

In this example, we are using the *Cyton* board over a USB-serial interface. Since one of the main features of this module is to guarantee real-time acquisition, the full `timestamp` of the acquired data is also available on the `timestamp_time_series` instance:

```
openbci.timestamp_time_series  # Unix format timestamp
```

High-level board configurations

A high-level library is obtained through the automatization of low-level commands. The `CytonConstants` class collects all SDK definitions into attributes using constant notation. For example, this is the Unicode used to configure the sample rates:

```
SAMPLE_RATE_16KSPS = b'~0'
SAMPLE_RATE_8KSPS = b'~1'
SAMPLE_RATE_4KSPS = b'~2'
SAMPLE_RATE_2KSPS = b'~3'
SAMPLE_RATE_1KSPS = b'~4'
SAMPLE_RATE_500SPS = b'~5'
SAMPLE_RATE_250SPS = b'~6'
```

Additionally, the high-level methods use multiple input shapes as valid parameters, e. g., the `CytonBase.command`. They also used to write commands into the firmware and can accept the raw Unicode, the `CytonConstants`, or the name in string format. This flexibility is appreciated at the time of implementing high-level interfaces.

```
>>> CytonBase.command(b'~4')
>>> CytonBase.command(CytonConstants.SAMPLE_RATE_1KSPS)
>>> CytonBase.command('SAMPLE_RATE_1KSPS')
```

2.2.2 Data acquisition and deserialization

This task must be executed in a separate process to guarantee an efficient acquisition and, in addition, an optimal data structure based on `multiprocessing.managers.SyncManager.Queue` is implemented in this task to access the data through a different process. Regardless of the interface selected, serial or Wi-Fi, the data format transmission can be configured as *RAW*, binary or formatted, using JavaScript object notation (JSON). The *JSON* format has the advantage that the data is already deserialized, but the disadvantage is that the transmitted packages are variable in size. Conversely, *RAW* formats fix the package size problem, but a deserialization process is necessary to access the real data. This last format was selected because of the fast transmission and the detection of lost packets.

Table 2-5 shows how, in 24 bytes, the 8 channels of 24-bit signed, each one, are compressed. This conversion is expensive for *Python* since there is no native 24-bit signed format. For 8 channels, the data has no singular conditions, but for 16

EEG data index	Description
[2:5)	Data value for EEG channel 1
[5:8)	Data value for EEG channel 2
[8:11)	Data value for EEG channel 3
[11:14)	Data value for EEG channel 4
[14:17)	Data value for EEG channel 5
[17:20)	Data value for EEG channel 6
[20:23)	Data value for EEG channel 7
[23:26)	Data value for EEG channel 8

Table **2-5**. EEG data distribution, values are 24-bit signed, MSB first.

channels, a specific format is implemented to interpret the 16 channels using the same amount of data transmitted for 8 channels. Table **2-6** describes the process for unpacking 16 channels for only 8 transmitted at a time. Basically, the *Cyton* (first 8 channels) and the *Daisy* (last 8 channels) transmissions are interleaved, and the empty blocks are completed by the mean of the last two transmissions from the same board.

After acquiring the binary data, a deserialization process is necessary. This process consists of converting the bytes to values with physical units, i. e., μV for EEG and g for acceleration. Once the stream is started, a continuous flow of binary data is stored in a queue-based data structure. This data is processed to extract the EEG and the AUX data. A few steps must be implemented to deserialize the binary package: (i) select a block of binary data, (ii) prepend the offset data to the block, (iii) find the bytes header (`0xa0`) and slice the block with this byte as the first element and the following are 33 bytes (at this point, the data is a list of arrays of maximum 33 elements), (iv) crop the block of binary data to ensure that the length for all elements is 33, and store the offset data to complete the next block of binary data, (v) create

Received		Upsampled board data	Upsampled daisy data
sample(3)		avg[sample(1), sample(3)]	sample(2)
	sample(4)	sample(3)	avg[sample(2), sample(4)]
sample(5)		avg[sample(3), sample(5)]	sample(4)
	sample(6)	sample(5)	avg[sample(4), sample(6)]
sample(7)		avg[sample(5), sample(7)]	sample(6)
	sample(8)	sample(7)	avg[sample(6), sample(8)]

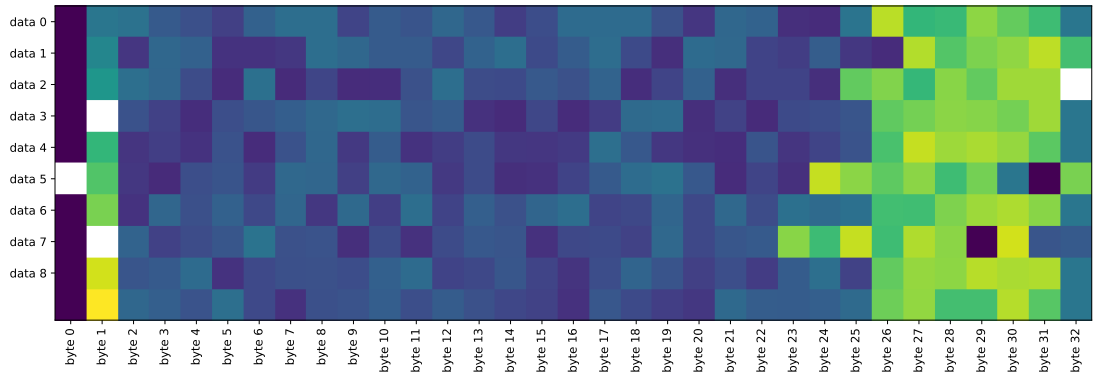
Table **2-6**. EEG data package format for 16 channels.

a matrix of shape `(33,N)`.

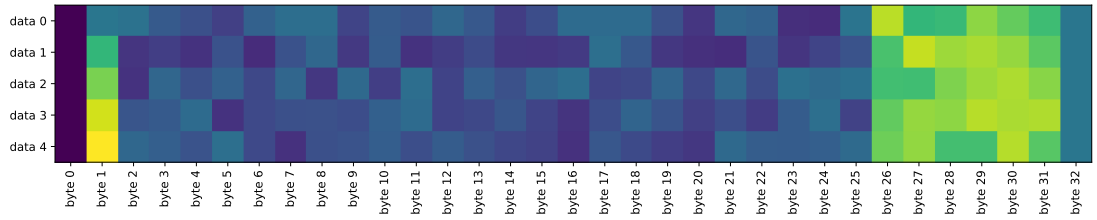
Now, the data structure on a shape `(33,N)` must meet a set of conditions: (i) all the first columns must contain the `0xa0` value, (ii) the second column must be incremental, and (iii) the last column must be in format `0xcX`, all with the same value. These conditions are described in Table **2-1**. All rows outside of these rules must be removed. Figure **2-2** shows graphically how a corrupted data set is cleaned and contextualized to deserialize the main structures.

2.2.3 External inputs acquisition (Boardmodes)

For *OpenBCI*, it is possible to configure the content of the Auxiliary data, and by default, it is the accelerometer data; nevertheless, there are different signal types: digital, analog, or marker. In marker mode, a value can be inserted programmatically into the time series; and for digital and analog mode, a set of physical input ports are available to insert signals. The external input acquisition is a significant feature since the capability to acquire signals alongside the EEG implies that the system can be used to measure latencies itself [44].



(a) Raw block of binary data with inconsistent bytes and missing ones.



(b) Block of binary data aligned and reshaped.

Figure **2-2**. Data deserialization must guarantee the data context to avoid overflow in subsequent data conversion. The fist block of columns, from 2 to 26, contain the EEG data and the columns 26 to 32 contain the Auxiliary data.

Programmatically, the boardmode can be changed at runtime through the method `command`:

```
>>> openbci.command(CytonConstants.BOARD_MODE_DEFAULT)
>>> openbci.command(CytonConstants.BOARD_MODE_DEBUG)
>>> openbci.command(CytonConstants.BOARD_MODE_ANALOG)
>>> openbci.command(CytonConstants.BOARD_MODE_MARKER)
```

Table **1-2** shows the number of analog and digital inputs available for each *OpenBCI* configuration. In default mode, the Auxiliary data will contain three-time series, one for each axis `(x,y,z)`, and for marker mode, a single time series with zeros and the value of the markers as singular points.

2.3 Data storage

The feature of storing acquired data for posterior analysis is fundamental for research. Feeding the state-of-art with public databases must be considered a community objective since collaborative efforts have brought BCI systems to the actual level.

The ideal format for data storage must meet a set of requirements: (i) it must not be RAM-based since the data appending process must read the packages, and these must be written to disk at that moment; (ii) must integrate a method to save metadata information alongside the time series, where this metadata can be used to describe things like the montage, the experiment, the engineer, and events, among others; (iii) the reading process must support partial access instead of allocating the complete archive in RAM; (iv) must be exportable to multiple formats.

hierarchical data format (HDF)

HDF was the format chosen for the default storage of EEG data, as it meets all previous requirements; however, there are some significant issues[3]. The most relevant one is related to the corruption risks, but this is a shortcoming that can be solved pragmatically with the implementation. Other reported problems are related to the performance, bugs, philosophy, and implementation, but these do not compromise the use of this specific application.

Custom data storage handler

This data handler uses *PyTables*, which is built on the top of the *HDF5* library using the *Python* language and the *NumPy* package. An important feature of *PyTables* is

[3] https://cyrille.rossant.net/moving-away-hdf5/

that it optimizes memory and disk resources, so data takes up much less space (especially, if on-flight compression is used) than other solutions such as relational or object-oriented databases [56].

The main modules to manage the proposed file format resides on the `HDF5Reader` and `HDF5Writer` classes, for example:

```
from openbci_stream.utils import HDF5Reader

reader = HDF5Reader('database.h5')
reader.eeg          # Raw EEG data (Channels x Time)
reader.markers      # Dictionary with keys as events
                    # and list of timestamps as values
reader.timestamp    # Array of timestamp time series
reader.annotations  # Arrays of events annotations
reader.close()
```

Or using *Python* context managers:

```
with HDF5Reader(filename) as reader:
    reader.eeg
    reader.markers
    reader.timestamp
    reader.annotations
```

The `HDF5Writer` is used internally to create the HDF5 file with these features. At runtime, a single script must be implemented isolated to ensure the storage from the processing and acquisition.

MNE compatibility

The `HDF5Reader` class has a method to generate *mne.EpochsArray*[4], this method needs a *tmin* (i. e., the time in seconds before the stimulus), the *duration* of the epoch, and the desired *markers*.

```
reader = HDF5Reader('sample-eeg.h5')
epochs = reader.get_epochs(tmin=-2, duration=6, markers=['RIGHT', 'LEFT'])
```

Feature	OpenBCI Python (2015) and pyOpenBCI (2015)	OpenBCI LSL (2017)	BrainFlow (2018)	OpenBCI Stream (2020)
Language	Python	Python	Python, C++, Java and C# and support for Julia, Matlab and Rust	**Python**
Acquisition boards	All OpenBCI boards	All OpenBCI boards	+10 different boards	**OpenBCI Cyton**
Distributed paradigm	No	No	No	**Yes**
Impedance measurement	No	No	No	**Yes**
Set boardmodes	No	No	No	**Yes**
Set sampling rate	No	No	No	**Yes**
Set package size	No	No	No	**Yes**
Markers syncronization	No	No	No	**Yes**
Asynchronous acquisition	No	Yes	No	**Yes**
Data storage	No	No	Yes	**Yes**
Active development	No	No	Yes	**Yes**

Table **2-7**. Acquisition drivers for OpenBCI, at the moment *BrainFlow* and *OpenBCI Stream* are the only drivers that support *OpenBCI Cython* acquisition board. The focus of the first one is to support acquisition and keep the compatibility across different hardware, *OpenBCI Stream* serve a configurable system suitable for development and research.

2.4 Summary and discussion

At the time of initiating the development of our drivers, four tools compatible with *OpenBCI Cyton* were available, as shown in Table 2-7. However, *OpenBCI Python*[5], *pyOpenBCI*[6], and *OpenBCI LSL*[7], all developed officially by *OpenBCI*, are now deprecated, but at the time, they served as references and templates for our implementation.

Currently, only two implementations are supported to work with *OpenBCI Cyton*, *BrainFlow*[8], and our drivers *OpenBCI Stream*[9]. *Brainflow* is recognized by the broadest hardware support, as more than 11 boards are now compatible. However, this can also be a disadvantage since developing device-agnostic applications to target more boards neglects the acquisition board's capabilities, for example, the features in *OpenBCI Stream* related to the Boardmodes, the sampling rate, the marker synchronization, and the Impedance measurement are implemented purely with in-depth handling of the hardware capabilities.

[4] https://mne.tools/stable/generated/mne.EpochsArray.html
[5] https://github.com/openbci-archive/OpenBCI_Python
[6] https://github.com/openbci-archive/pyOpenBCI
[7] https://github.com/openbci-archive/OpenBCI_LSL
[8] https://brainflow.org/
[9] https://openbci-stream.readthedocs.io/

CHAPTER

THREE

REAL-TIME AND DISTRIBUTED IMPLEMENTATION

In the previous chapter there is a lack bout data acquisition and transmission, this issues are solved over a distributed computing approach. In this chapter are described the implementation of the developed drivers for OpenBCI over a distributed frame. This feature use a distributed event store and stream-processing platform called *Apache Kafka* for the data transmission through the network, this same network is also used to deploy web servers and synchronize the systems. All this set defines the real-time conditions and describe the operating condition of the BCI system implemented.

3.1 Real-time

The real-time definition used to describe the purposed system is based on the sampling blocks transmission, the system developed guarantees that all EEG data blocks of duration P will be *available* for the user in a time lower that P, no

matters the duration of the block. In this case *available* refers to ready to use data in the development environment.

This definition arises from the need to compare different system configurations. Given that the obtained acquisition system is highly configurable about sampling rate, numbers of channels, protocols, and block of data transmitted. Express the latency in percentage terms simplify the comparison of the capabilities when a BCI system is designed and developed.

3.2 Kafka: Open-source distributed event streaming platform

The core of the distributing system resides in Kafka, this software is fast enough to implement real-time applications and so simple to be based on binary data transmission. The main platform run under Java but there is a wrapper called *kafka-python*[1] that able the programming inside python environments.

In addition to all the infrastructure that Kafka brings to the system, when used in the development of a system, this work only will refers to the main features related to read and send stream data: Produsers, consumers and generators scripts.

3.2.1 Topics

Kafka protocol is so simple that the identifiers consist of a single string called *topic*, this topic are categories used to organize messages. Internally they are configured by other features like, *replication factor* and *partitions*, our implementation consist only in one server, in order to speed-up the message transmission by avoiding the triggering and the redundant message allocation. The topics needs explicit definition, there can not be created on run-time.

[1] https://kafka-python.readthedocs.io/en/master/

3.2.2 Producer

From a coding perspective, the producers consist of scripts that feed the streaming with new data, then Kafka distribute this messages to all nodes connected. Since Kafka only comprise the transmission of binary data, all messages involved are serialized in producers and de-serialized in consumer using the Python standard library `pickle`.

```
from kafka import KafkaProducer
producer = KafkaProducer()

for _ in range(100):
    producer.send('my_topic', b'some_message_bytes')
```

In order to run this scripts, the Kafka service must be running in background.

3.2.3 Consumer

The scripts that implement a Kafka consumer are based on asynchronous callbacks:

```
from kafka import KafkaConsumer

consumer = KafkaConsumer()
consumer.subscribe(['my_topic'])
for msg in consumer:
    print(msg)
```

3.2.4 Transformer

The transformer are a combination of consumers and producers, this kind of scripts consume data from the stream and then stream back with a different topic. The transformers are useful to parallelize process.

3.3 Distributed system implementation

The distributed implementation is used to isolate the acquisition system, since this is not running under a real-time operating system, the latencies can turn into unstable ranges due other non related computational process running in background.

Centralized process are also susceptible to high computational demanding processing tasks, if a BCI system is under development the performance could vary in part only due the available resources, compromising the comparability of the evaluated models.

3.3.1 Remote Python Call

The RPyC Python module enable the execution through network of wrapped modules, this module require a few configurations to start serving object-proxying to overcome the physical boundaries between processes and computers, so that remote objects can be manipulated as if they were local.

Although the module requires a custom scripting in order to access to object-proxying, with the aim to simplify the implementation of the drivers a transparent configuration have been included in the development. The argument `host` refers and initialize the connection with the remote module running in the pointed server. All methods, commands and responses generated are processed in the same way that local are, them, the user will not notice any difference from use the library with a local or a remote OpenBCI device.

```
from openbci_stream.acquisition import Cyton

openbci = Cyton('serial', host='192.168.1.1', endpoint='/dev/ttyUSB0', capture_stream=True)
openbci.stream(15)  # capture 15 seconds of data
openbci.eeg_time_series # Raw EEG is allocated in this class instance
```

3.4 Isolated acquisition

Once defined the distributed system, a recommended step is move the acquisition into a isolated environment. A dedicated operating system that only executes the daemons and process related to the acquisition as well as the Kafka server. This system also can be configured as a Wi-Fi access point, needed to handle the OpenBCI through the Wi-Fi module. All this tasks can be implemented into a SBC using a operating system based on a minimalist distribution of GNU/Linux.

For the development of this work, a preconfigured environment for Raspberry Pi was developed using *Archlinux ARM*, although *Manjaro ARM Minimal* can also be used, the configuration is described in the *documentation*[2] as a simple terminal command over a fresh installation.

This distribution enable the Raspberry Pi as a *plug and play* acquisition server. Every time that the system boot this one will be configured as A real-time protocol (RTP) server and a Wi-Fi access point, the Kafka server will start to running in background, the binary deserializer daemon start listening binary data, and the EEG streamer start to listening deserialized data, and the RPyC server starts wrapping the drivers.

Although the RPyC module brings access to the EEG data, this protocol is not enough for the high rate transmissions needed for configurations of 1000 SPS and 16 channels. For this reason Kafka is used to distribute data even when there is only one client involved.

3.5 Electrode impedance measurement

A low impedance electrode-skin is always recommended because, under low ranges, the effect on the amplifications remains at low levels, even lower than the

[2] https://openbci-stream.readthedocs.io/en/latest/notebooks/A3-server-based_acquisition.html

resolution of Analog-to-digital converter (ADC). The *ADS1299*[3] in the analog-to-digital converter for biopotential measurements is implemented in *OpenBCI*: this microcontroller includes a way to perform the impedance measure using the lead-off current sources.

This method consists of injecting a small current of $6nA$ at $31.2Hz$. The signal acquired is processed to calculate the V_{RMS} voltage and then the impedance Z using Ohm's law.

```
openbci = Cyton(
    'wifi',
    '192.168.1.113',
    host='192.168.1.1',
    capture_stream=True,
    daisy=False,
)

openbci.command(cons.SAMPLE_RATE_250SPS)
openbci.command(cons.DEFAULT_CHANNELS_SETTINGS)
openbci.leadoff_impedance(
    range(1, 9),
    pchan=cons.TEST_SIGNAL_NOT_APPLIED,
    nchan=cons.TEST_SIGNAL_APPLIED,
)

openbci.stream(7)
data_raw = np.array(openbci.eeg_time_series)


band_2737 = GenericButterBand(27, 37, fs=250)
data = band_2737(data_raw)
```

The V_{RMS} can be calculated as the `std()` of the voltage array.

$$V_{RMS} = \frac{V_{pp}}{2\sqrt{2}} \approx std(V)$$

This is how our I_{RMS} can be calculated:

[3] https://www.ti.com/product/ADS1299

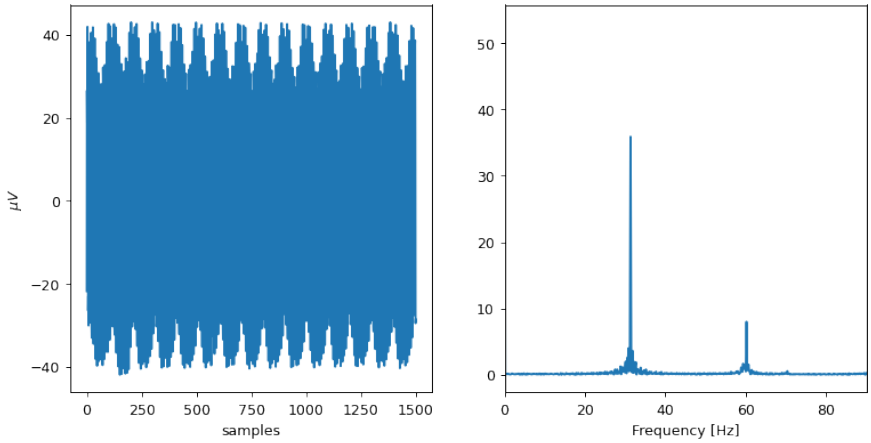


Figure **3-1**. Raw signal for the *lead-off* configuration.

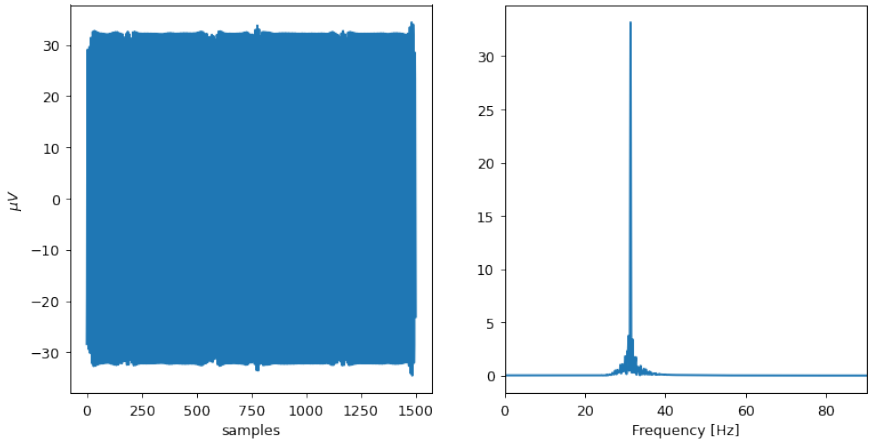


Figure **3-2**. Filtered signal for the *lead-off* configuration.

$$I_{RMS} = \frac{6nA}{\sqrt{2}}$$

Then the impedance Z is...

$$Z = \frac{V_{RMS}}{I_{RMS}}$$

Since the V_{RMS} and the $std(V)$ are by default in μV, this would be the impedance measured for a vector of data V:

$$Z = \frac{std(V) \cdot 10^{-6} \cdot \sqrt{2}}{6 \cdot 10^{-9}}\ \Omega$$

The Cyton board has a 2.2K Ohm resistors in series with each electrode, so we must remove this value in way to get the real electrode-to-head impedance.

Real time measurement

For this experiment we will use the Kafka consumer interface, and the same potentiometer. Keep in mind that this measurement uses one second signal, so, the variance will affect the real measure, in real-life the amplitude not change so drastically.

```
from openbci_stream.acquisition import OpenBCIConsumer
from openbci_stream.acquisition.cyton_base import CytonConstants as cons
from openbci_stream.utils.filters import GenericButterBand
import numpy as np
import time

def get_rms(v):
```

```
    return np.std(v)

def get_z(v):
    rms = get_rms(v)
    z = (1e-6 * rms * np.sqrt(2) / 6e-9) - 2200
    if z < 0:
        return 0
    return z

Z = []
band_2737 = GenericButterBand(27, 37, fs=250)
with OpenBCIConsumer(
    'wifi',
    '192.168.1.113',
    host='192.168.1.1',
    auto_start=False,
    streaming_package_size=250,
    daisy=False,
) as (stream, openbci):
    # with OpenBCIConsumer(host='192.168.1.1') as stream:

    openbci.command(cons.SAMPLE_RATE_250SPS)
    openbci.command(cons.DEFAULT_CHANNELS_SETTINGS)
    openbci.leadoff_impedance(
        range(1, 9),
        pchan=cons.TEST_SIGNAL_NOT_APPLIED,
        nchan=cons.TEST_SIGNAL_APPLIED,
    )
    time.sleep(1)
    openbci.start_stream()

    for i, message in enumerate(stream):
        if message.topic == 'eeg':
            eeg, aux = message.value['data']
            eeg = band_2737(eeg)
            z = get_z(eeg[0])
            Z.append(z)
            print(f'{z/1000:.2f} kOhm')

        if i >= 15:
            break
```

This measure needs a block of data to obtain a stable value. Although the method used to calculate the V_{RMS} is fast, the `std()`, at non-stationary times, such as while the electrode is fixed or manipulated, will affect the impedance measurement, so the calculated value will not be accurate. The measure impedance protocol requires rest periods before interpreting the value calculated.

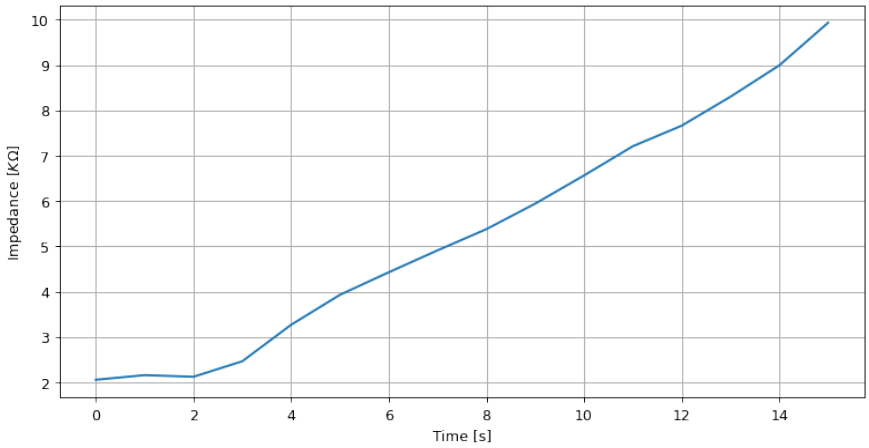


Figure **3-3**. Real-time impedance measurement of a 10 KOhm potentiometer.

Some recommendations for improving the impedance measurement can be: (i) Take shorts signals but enough, 1 second is fine. (ii) Remove the first and last segments of the filtered signal. (iii) Nonstationary signals will produce wrong measurements. (iv) A single measurement is not enough, is recommended to work with trends instead.

3.6 Latency analysis

The latency comprise the time elapsed between the acquisition of raw EEG data from the OpenBCI board and the availability in the development framework. This analysis was performed over a complete distributed system and the following conditions: (i) The OpenBCI acquisition system was isolated in a dedicated Raspberry Pi, (ii) The data was read in a remote computer using the developed drivers, (iii) The block size was fixed in 100 samples, and (iv) The samples per second was fixed in 1000. The system was designed in a way that all times are registered and streamed alongside the main EEG data. This feature able to developers to perform a latency analysis without configure a special mode, this means that use the same conditions of a EEG acquisition session.

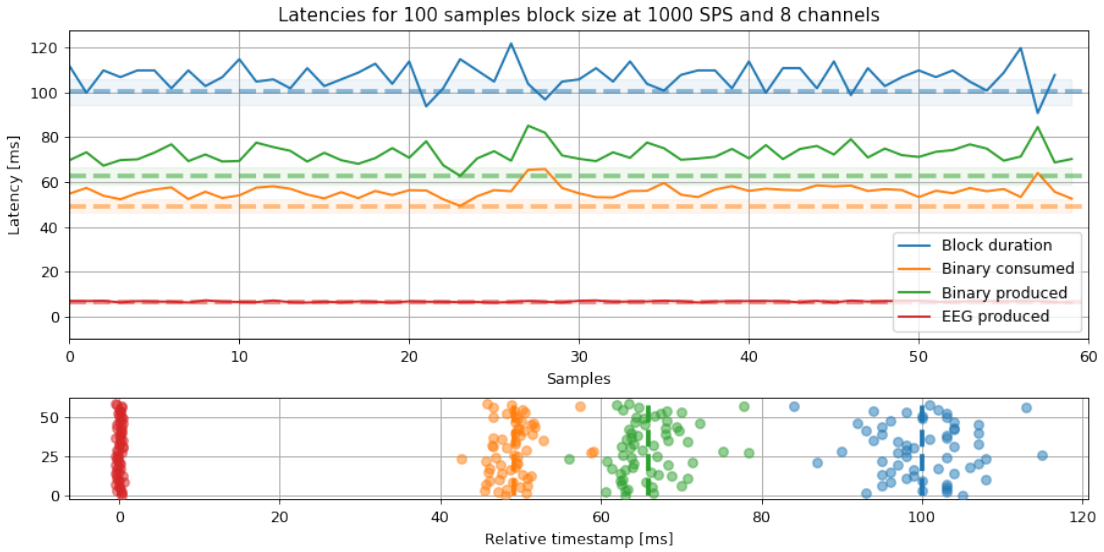


Figure **3-4**. Latencies for 100 samples block size and 1000 SPS. The latencies show the elapsed time from reading the packet to the time of packaging. The dashed line mark the minimum latency and the shade is the standard deviation for all segment.

In figure 3-4 are plotted four relative timestamps and compared with the block duration. The *Binary produced* is the time elapsed since the raw data was acquired and streamed through Kafka, *Binary consumed* is the time elapsed since the binary data was consumed, just before to the deserialization, *EEG produced* refers to the duration of the transmission, this is the time since the thee EEG was inserted in the Kafka stream and read in the final consumer. There is a few facts about this plot that needs to be mentioned, The difference between zero and *EEG produced* also includes the clock offset. The time bewteen *Binary consumed* and *EEG produced* is the time spent on the deserialization of the raw data. The time bewteen *Binary produced* and *Block duration* is the time spent on the acquisition of the EEG data, this is the latency acquisition of OpenBCI when its works over the WiFi protocole. We can conclude then, that the deserialization process is highly time expensive.

The same process was performed for six different block sizes keeping the same 1000 SPS, the results are showed in Figure 3-5, only the *Binary produced* is compared with the *Block size*. For block sizes under the 1000 samples and up to

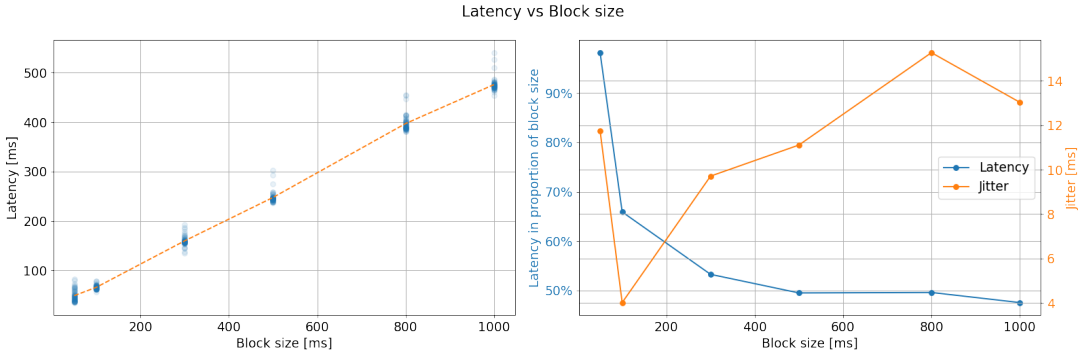


Figure **3-5**. Latency vs Block size. In the left is possible to see that the latency is proportional to the block size. In the right plot, the latency decrease for small block size but their standard deviation (jitter) increase. The preferred configuration was set up in 100 samples block size due the lower jitter.

100 the latency seems to be lineal. If the data is represented using the percentage as a comparison measure, the latency seems to stabilize in 50%, however the jitter appear to increase with block size. This results suggest that the optimal configuration for the EEG acquisition, using the developed drivers, is around 100 samples block size with only 8 ms of jitter.

Table **3-1** show a comparison between different BCI systems configurations. The wired systems has significantly lower jitter than wireless. For centralized implementations like *BCI2000 + g.USBamp* the latency depends also of the paradigm, then, the same configuration have different latencies responses.

3.7 Sampling analysis

The sampling is the process to acquire data at fixed sample rate, the acquisition process must gather the a uniform sampling rate of the signal. It is inevitable to lose data, there are some reasons, like lags in the Wi-Fi connection, in the acquisition board or in the operating system. If it is assumed that the acquisition is homogeneous, then, after a session of EEG all data will be widen to cover the

BCI system	Sample rate	Block size	Jitter	Communication	Distributed	Latency
BCI2000 + DT3003 [41]	160 Hz	6.35 ms	0.67 ms	Wired	No	**51.9 %**
BCI2000 + NI 6024E [41]	25 kHz	40 ms	0.75 ms	Wired	No	**27.5 %**
BCI2000 + g.USBamp [44]	1200 Hz	83.3 ms	5.91 ms	Wired	No	**14, 30, 48 %**
OpenViBE + TMSi Porti32 [57]	512 Hz	62.5 ms	3.07 ms	Optical MUX	No	**100.4 %**
BCI-Framework	**1000 Hz**	**100 ms**	**5.7 ms**	**Wireless**	**Yes**	**56 %**

Table **3-1**. Latencies comparison, the latency has been expressed in terms of percentage of the block size to make the possible the comparison between different systems configurations.

supposed duration of the experiment. This will drive to an incorrect partitioning of trials. OpenBCI includes a set of special features that can be used to detect this anomalies. The first one is to use the sample indexes to detect when a data is not transmitted, and the second consist of to use known test signal. Additional to this ones, the developed drivers include a timestamp annotation for each sample that can be used too.

For the following analysis, a signal of continuous 64 minutes of acquisition was processed, recorded at 250 SPS, with 100 samples per block size and 16 channels. In Figure **3-6** are compared the three features to detect the samples skips. Any of this methods can be used to locate this points. The selected one is to use the sample indexes, this method has the advantage that can be used with a normal acquisition of EEG, the timestamp also can be used, but is more easy and fast to process the sample indexes. The test signal requires to use the channels for EEG, which makes it less practical.

After to perform an algorithm to detect the sampling skips in the signals, this ones are marked as `BAD:sample`. The Figure **3-7** relates the timestamp with this markers.

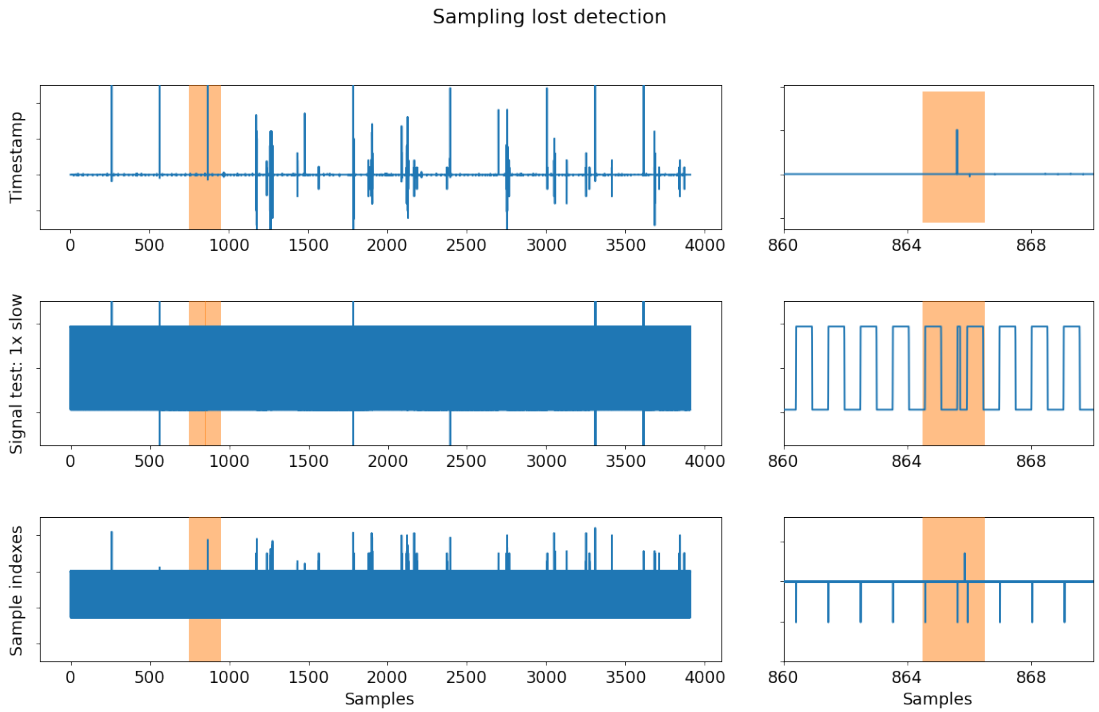


Figure **3-6**. The sampling lost can be detected by analysing the timestamp, using a test signal or analysing the sample index.

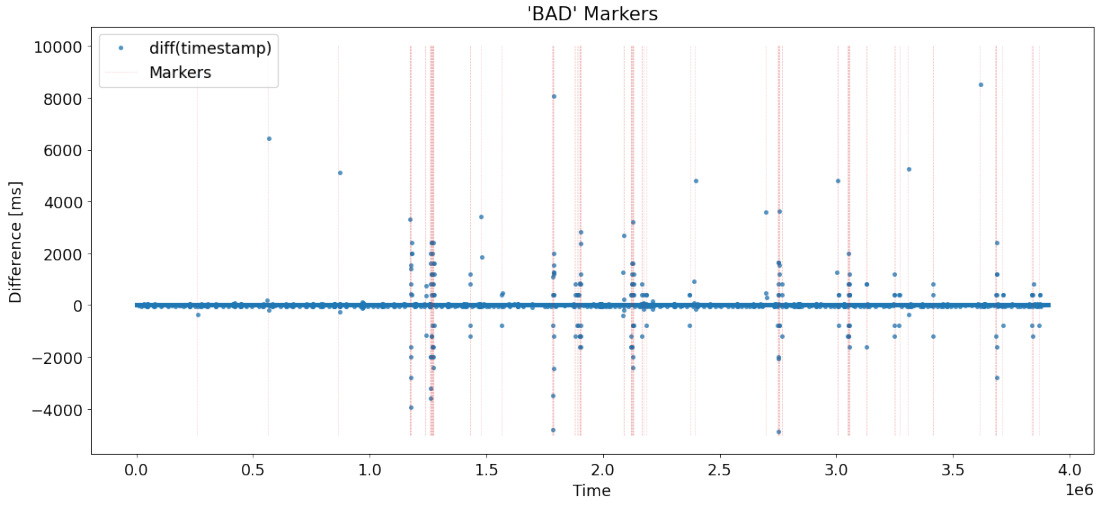


Figure **3-7**. The bad markers detection can localize the sampling lost and place markers in that locations.

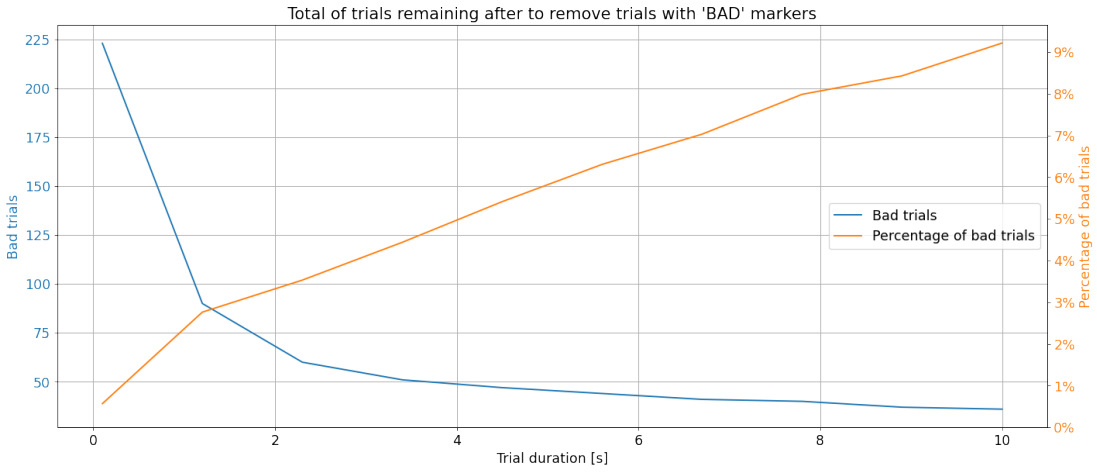


Figure **3-8**. This plot show the trials and the proportion trials remaining after to remove the trials that contain almost one 'BAD' marker for a session of 60 minutes of EEG acquisition, and with a dynamical trial duration between 100 ms to 10000 ms

The `BAD:sample` markers can be used to crop and remove the signal around this ones and to keep with the correctly sampling data.

In a real experiment is desired a set of trials free of `BAD` markers. The density of this markers can affect a number determined of trials depending of the size of the trial. Figure **3-8** show the results of an experiment of 60 minutes of duration and how many trials must be discarded for each trial duration.

Figure **3-9** compare the sampling rate around this markers. In the left without remove the skipping points, and in the right after to remove the points identified as bad samples. The right column data are more easy to interpret, the superior plot show the difference of the period in milliseconds, a solid line mark the expected $4ms$ $(1/250Hz)$, a secondary line $50ms$ equally distributed above and below. The inferior plot describe the mean of the period for each one of the segments resulting after removing the samples around the `BAD:sample`.

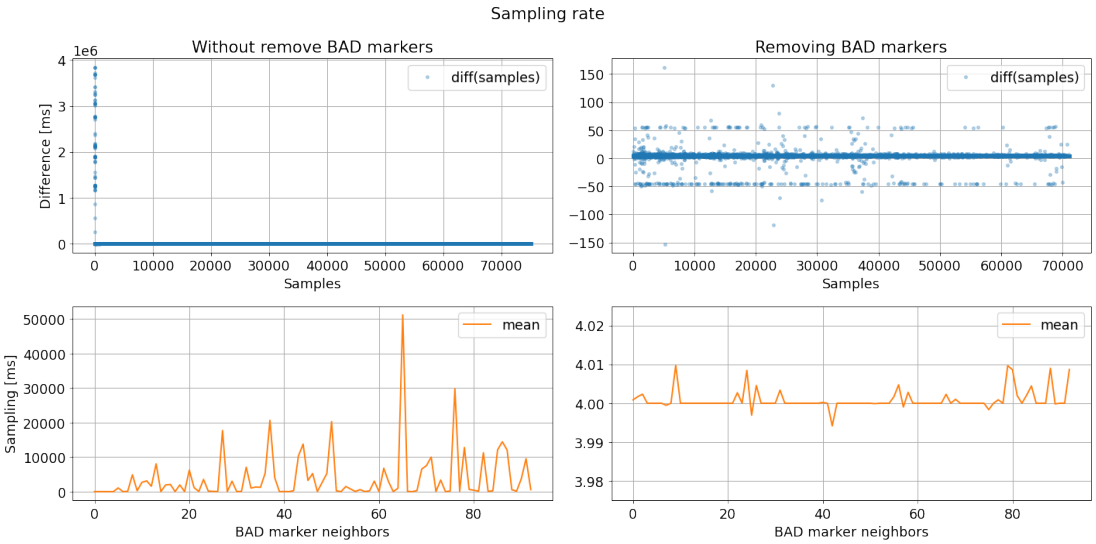


Figure **3-9**. The sampling rate analysis after remove bad markers evidence a data period acquisition close to 4 ms (250 SPS).

3.8 Summary and discussion

The real-time implementation used in this works has been fixed to guarantee the distributions of packages in a time inferior to the package duration itself. Also, the latency measures are expressed in terms of proportion of the acquisition block size, this approach facilitate the comparison with other systems and with the proposed system itself due the different acquisition configuration.

There are two resources used to implement distributed features, the first one is RPyC, this Python module create a remote proxy to access to the drivers module configuration and basic variables. This method use a protocol communication that is too slow to be used to stream EEG in real-time, then Kafka handles this task by implementing a real-time distributed streaming platform. Kafka, also brings other features to generate custom data streams.

The best configuration that integrate, for high sampling rate (1 KSPS), a low and stable latency was the one with fixed transmission block size of 100 samples. This

configuration keeps the system stable enough to recover after interruptions and still flexible to implement real-time tasks.

Detect interruptions in the acquired signal is important for the BCI implementations, the performance of the developed systems reside on the quality of the data with which they are fed. Our drivers was designed with the feature to keep and store the data necessary to make this detection and perform automatic purge of bad trials..

CHAPTER

FOUR

BCI-FRAMEWORK

In computer programming, a framework is an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing application-specific software. It supply a standard way to build and deploy applications and is a universal, reusable software environment that allow particular functionality as part of a larger software platform to facilitate the development of software applications, products and solutions. Frameworks-based software may include support programs, compilers, code libraries, toolsets, and application programming interfaces (APIs) that bring together all the different components to carry up the development of a project or system [58] without the need to looks out of the framework.

This chapter describe the implementation of a independent BCI software that consists of a distributed processing tool, stimuli delivery, psychophysiological experiments designer and real-time data visualizations for *OpenBCI*.

We purpose an open-source tool for the acquisition of EEG/EMG/ECG signals and designed to work with *OpenBCI's Cyton* board, the main core of this software lies on

OpenBCI-Stream[1] and a library designed to handle all the low-level hardware features to extend the hardware capabilities with high-level programming libraries developed in Chapter 2. *BCI-Framework* comprises a GUI with a set of individual computational processes (distributed or in a single machine), that feeds visualizations, serve stimuli delivery, handle acquisitions, storage data and/or process data in real-time. Additionally has a built-in development environment and a set of libraries that the user can implement to create their specific functionalities.

BCI-Framework as well as all BCI applications implies the participation of at least two kind of users that will be widely used in this chapter: (i) the *user* role refers to the person that manipulate the framework to execute experiments; (ii) the *patient* is the subject from whom the EEG data is being acquired; (iii) a third profile can be assigned to a *developer* or *researcher*, this subject use the framework extensibilities to create new experiments and visualizations.

4.1 Software description

BCI-Framework is a desktop application that was developed entirely using *Python* and the GUI was implemented initially on PySide2 and updated to run under the latest stable release of *PySide6*[2]. In fact all libraries implemented are free (as in freedom) or at least *Open-Source*. The software architecture is modular and designed taking into account the scalability and the configurability. Almost all components run on independent computational process and are connected to the main interface through websockets or simple *HTTP* petitions.

The major feature of *BCI-Framework* is the capability to implement a complete EEG-based BCI system in a single software application. This achievement, in part,

[1] https://openbci-stream.readthedocs.io
[2] https://wiki.qt.io/Qt_for_Python

is due the dedicated support to *OpenBCI Cyton* board and the Isolated acquisition that was described and implemented in Chapter 3. This feature leaves to the main machine, the one that run the framework interface, to keep all the resources to perform visualisations, processings and run a stimuli delivery server. And at the same time have a reliable flow of data.

BCI-Framework use a set of background services that run independent of each other, some process are initialised from inside the framework and others from outside, all process run under a distributed network. All this backend services exchange information using *Kafka* or *Websockets*, it depends on the priority level or the amount of the information transmitted.

4.1.1 Real-time visualizations backend

This feature uses the *Matplotlib*[3] backend through *FigureStream*[4] a python module developed explicitly for this framework, that serve the visualization using a simple HTTP real-time streaming. This module creates a static endpoint and the image is updated like in a video streaming, this is a synchronous process, this means that the user has the control and the responsibility to feed the streaming frame by frame.

In background, a *Flask*[5] application is running and serving the endpoint with the streaming plot for all users in the network, then, the visualizations can be both, consumed and generated from any terminal. This feature is ideal for distributed systems, since, costly real-time visualizations can be executed on a independent hardware.

The development of this backend module is described in the appendix Matplotlib-FigureStream.

[3] https://matplotlib.org/
[4] https://figurestream.readthedocs.io/
[5] https://flask.palletsprojects.com/

4.1.2 Stimuli delivery backend

This environment is based on *HTML* and *JavaScript*, is basically a dynamic web application. However, in order to standardize all user scripts into a single language, this feature use *Brython*[6], a *Python*3 implementation for client-side web programming. This tool able to the user to write web applications, however, is necessary a server running with the dependencies and the library itself. With the intention to simplify the implementation and the start up of *Brython* projects, a Python module called Brython-Radiant framework was implemented that solve this isssues. This module is a *Brython* framework for the quick development of web apps with pure *Python*/*Brython* syntax so there is no need to care about *HTML*, *CSS*, or *JavaScript*. Runs over *Tornado*[7] and include support to *Websockets*, *Local Python Scripts* and *Material Design Components*[8]. This is basically a set of scripts that allows the same script run from *Python* and *Brython*, when its running under *Python* a *Tornado* server is created and configure the local path for serving static files, and a custom *HTML* template is configured in runtime to import the same script, this time, interpreted by *Brython*.

Brython-Radiant able to the user to create with pure Python and a single script a complete web application that can interact with the native Python environment. And when run under a network-based implementation allows to be consumed from any terminal.

4.1.3 Development environment

The main interface is divided into two high-level functionalities: (i) Real-time analysis, to collect and process data, include and interface to handle custom

[6] https://brython.info/index.html
[7] https://www.tornadoweb.org
[8] https://material.io/develop/web

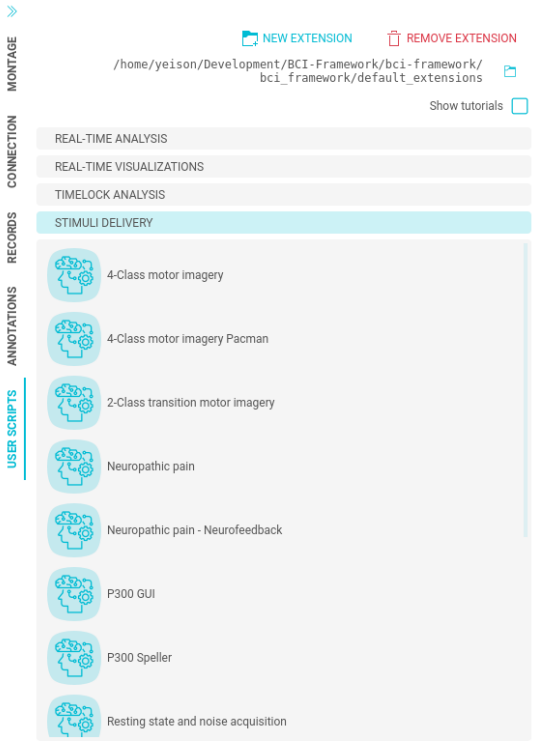


Figure **4-1**. BCI-Framework: The extensions panel is used to access to all visualizations and stimuli delivery paradigms source code, as well to create a new ones from scratch.

visualizations and serve *Kafka* generators; (ii) Stimuli delivery, to serve remote audiovisual stimuli and configure experiments. This functionalities includes a set of visualizations and basic paradigms by default, instead of being rigidly included in the interface, they are in fact fully editable and configurable, even new ones can be created from scratch as an extension. All scripts are accessible under an integrated development environment.

The development environment is one of the most outstanding features of *BCI-Framework*, this environment able to developers to implement custom data analysis, real-time visualizations and, design custom stimuli delivery experiments or paradigms without leaving the main interface. This is achieved by the development and the implementation of an API that interact with the framework

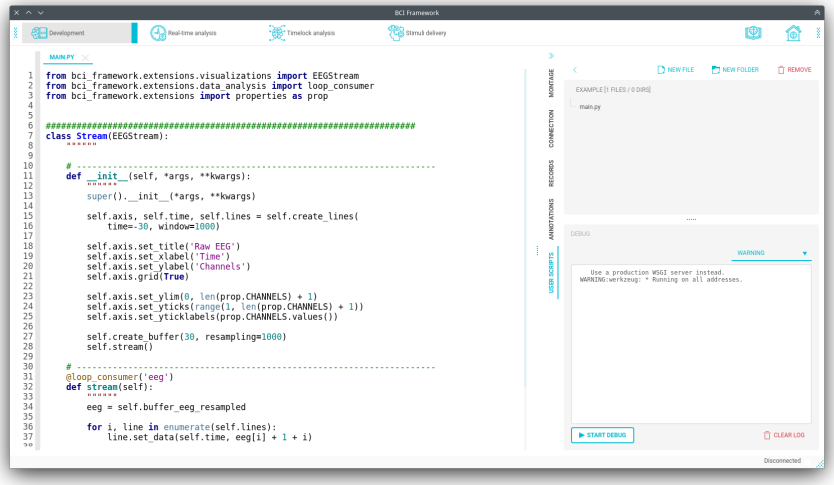


Figure **4-2**. BCI-Framework: Integrated development environment with previsualization area and debug console.

parameters, the real-time data stream and with the users. Figure **4-2** shows a capture of the integrated development environment, that consist of a full Python syntax highlighting, a previsualization area, a directory tree navigator, and a debugging console.

4.2 Real-time data analysis

The data analysis is powered by all *Python* modules and the wrapped ones. This language has showed, to be a good choice for research and develop neuroscience implementations [59]. Currently there are a bunch of modules like *MNE*[9] specifically designed for exploring, visualizing, and analyzing human neurophysiological data. Alongside, much other that can be used to implement custom analysis like *Numpy*[10] and *Scipy*[11], or for implement machine learning

[9] https://mne.tools/
[10] https://numpy.org/
[11] https://scipy.org/

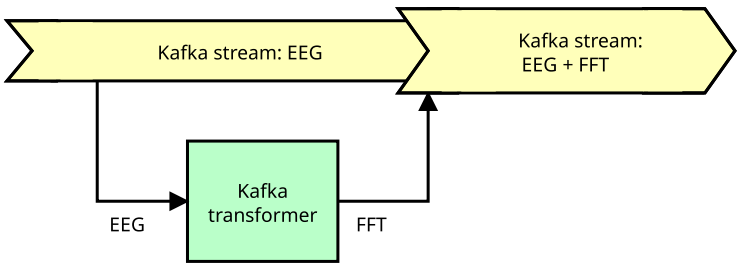


Figure **4-3**. Kafka transformer that generates a new kind of data (for example FFT) and stream it alongside the existent streams.

approaches like *Scikit-learn*[12] and *TensorFlow*[13].

BCI-Framework was designed in such a way that the user/researcher can use the environment in combination with an easy to access EEG signals and markers to build their BCI system without care about the acquisition, synchronization and distribution. This approach allows to implement the real-time analysis as a basic Kafka consumer or a Kafka transformer that connect with the stream that contain the EEG (or other signal) and consume the data to serve the user/research and then generate reports, execute local commands or send back a new kind of data to the stream.

For example, we can suppose that we need the Fast Fourier transform (FFT) of the EEG signals in real-time, then, we can use a single script that calculate the FFT and create a new topic with the new data and stream it through all consumers. This transformer is graphically represented in Figure **4-4**, this script must run as an isolated process, and could be executed on any terminal of the network, the new data stream is automatically integrated and is available for all terminals.

[12] https://scikit-learn.org/
[13] https://www.tensorflow.org/

4.2.1 Data analysis scripting

The scripting consist of configure through a custom API the environment needed to read EEG signal. A bare minimum script looks like:

```
from bci_framework.extensions.data_analysis import DataAnalysis

class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

if __name__ == '__main__':
    Analysis()
```

The `DataAnalysis` class includes a lot of useful methods to handle and configure the acquisition and manipulation of the data stream.

The API includes a custom decorator called `loop_consumer`, that is used to access asynchronously to the data stream. on every `'eeg'` and `'marker'` incoming data, this strings represents *Kafka* topics. The `stream` method only needs to be called a single time, after that, the decorator takes control of the executions, this means that entire script must finish with the call of the decorated method, in the constructor method. Is not possible, to use the decorator `loop_consumer` in more than one place, so the argument `topic` could be used to create a flow control.

```
from bci_framework.extensions.data_analysis import DataAnalysis, loop_consumer

class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.stream()

    @loop_consumer('eeg', 'marker')
    def stream(self):
        print('Incoming data...')

if __name__ == '__main__':
    Analysis()
```

The decorated method receives five optional arguments: `data`, `topic`, `frame`, `latency` and `kafka_stream`. This arguments are defined in the wrapped decorator definition that is the reason why there is no arguments in the initial caller. This arguments works like inputs, and are optional, if some one is not declared then that local variable will be not created.

```
@loop_consumer('eeg', 'aux')
def stream(self, data, topic, frame, latency):
    print(f'Incoming data #{frame}')

    match topic:
        case 'eeg':
            print(f'EEG{data.shape}')
        case 'aux':
            print(f'AUX{data.shape}')

    print(f'Topic: {topic}')
    print(f'Latency: {latency}')


@loop_consumer('eeg', 'aux', 'marker')
def stream(self, topic):
    match topic:

        case 'eeg':
            print("EEG data incomming...")

        case 'aux':
            print("AUX data incomming...")

        case 'marker':
            print("Marker incomming...")
```

This scripting define the methods to subscribe to the Kafka stream topics. This solve issue about the access to the real-time data stream, however a high-level methods are also available to process, crop and buffer data.

Buffering

For real-time analysis, additionally to subscribe and acquire data from topics, is required a method to handle the buffer of the data acquired. The `create_buffer` method is used to configure the retention of the data streamed.

```
self.create_buffer(seconds=30, fill=0)
```

The previous command will create a buffer of 30 seconds filled with 0 by default, the size of the buffer depends of the sampling rate and the number of the channels, this information is obtained from the framework parameters.

```
class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.create_buffer(seconds=30, fill=0)
        self.stream()

    @loop_consumer('eeg', 'aux')
    def stream(self):

        #  Buffer: all data from the last 30 seconds
        eeg = self.buffer_eeg
        aux = self.buffer_aux

        print(f'EEG{eeg.shape}')
        print(f'AUX{aux.shape}')
```

Notice that the previous script has not argument in the decorated method, and that the buffer is accessible through the `self.buffer_eeg` and `self.buffer_aux` instance attributes.

Resampling

A sample length fixed will be created with the argument `resampling`, by default is 1000 samples. A buffer for the auxiliary data with the same features is also generated. The buffering process is transparent for the user, once defined with `create_buffer` this will be automatically updated.

```
class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.create_buffer(seconds=30, fill=0, resampling=1000)
        self.stream()

    @loop_consumer('eeg', 'aux')
```

```
    def stream(self):

        # Resampled buffer:
        # the data from the last 30 seconds in a vector of 1000 samples
        eeg_r = self.buffer_eeg_resampled
        aux_r = self.buffer_aux_resampled

        print(f'EEG{eeg_r.shape}')
        print(f'AUX{aux_r.shape}')
```

The resampled timeseries use the `self.buffer_eeg_resampled` and `self.buffer_aux_resampled` instance attributes.

Data slicing referenced by markers

This feature works similar to `loop_consumer`, but instead of return the latest block of data, the `marker_slicing` method return a *trial*. A *trial* is localized by almost one marker, and a margin time previous and posterior to the marker.

```
from bci_framework.extensions.data_analysis import DataAnalysis, marker_slicing

class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # Needs to be greater than the duration of the slice.
        self.create_buffer(seconds=30, aux_shape=3)
        self.slicing()

    @marker_slicing(['Right', 'Left'], t0=-2, duration=6)
    def slicing(self, eeg, aux, timestamp, marker):

        print(eeg.shape)
        print(aux.shape)
        print(timestamp.shape)
        print(marker)
        print()

if __name__ == '__main__':
    Analysis()
```

In the previous script we are capturing trials of 6 seconds duration around the markers `'Right'` and `'Left'`. This feature is useful for debugging purposes, and synchronous paradigms with explicit markers presence, like ERP.

Feed the stream using Kafka producer

The data analysis can also works as a *Kafka* producer, this feature is used to generate *commands*, *annotations*, *feedbacks* and any other type of data under custom topics.

```
from bci_framework.extensions.data_analysis import DataAnalysis

class Analysis(DataAnalysis):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

if __name__ == '__main__':
    Analysis(enable_produser=True)
```

Once activate the producer, the `send_command`, `send_feedback` and `send_annotation` methods are available.

```
# Command
self.send_command('MyCommand', value=45)

# Annotation
self.send_annotation('The subject yawn', duration=5)

# Feedback
feed = {'var1': 0,
        'var2': True,
        'var3': 'Right',
       }
self.send_feedback(**feed)

# send_feedback method are keyword arguments only
self.send_feedback(a=0, b=2.3, c='Left')

# Custom topic
self.generic_produser('my_topic', my_data)
```

The `generic_produser` method can be used to stream any custom data under a user defined topic.

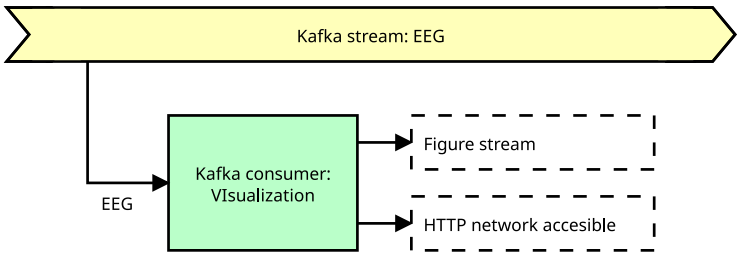


Figure **4-4**. Kafka consumer that access to the real-time EEG stream and use the data to perform individual process.

4.3 Real-time visualization

The visualizations works very similar that the Real-time analysis, with the difference that only comprise Kafka consumers due the visualization is indented to be displayed inside of the *BCI-Framework* interface and over a *HTTP* protocol instead to create a new kind of data stream.

The real-time visualizations consists of a computational process that manipulate the data in order to create static visualization and then update this one consecutively. The environment to create visualizations automatically serve the real-time EEG stream, then the user only must comply about the visualizations.

4.3.1 Data visualization scripting

Data visualization are based on Matplotlib-FigureStream, this interface inherits all features from it and extends the utilities with an specific ones. Additionally all topics explained in _analysisch4:data_analysisReal-time analysis development are valid here too, since visualizations is a special case of real-time analysis.

```
from bci_framework.extensions.visualizations import EEGStream
from bci_framework.extensions.visualizations.utils import loop_consumer
```

```
class Stream(EEGStream):  # This is `matplotlib.Figure` based class
    def __init__(self):
        super().__init__()

        self.axis = self.add_subplot(111)
        self.axis.set_title('Title')
        self.axis.set_xlabel('Time')
        self.axis.set_ylabel('Amplitude')
        self.axis.grid(True)
        self.stream()

    @loop_consumer('eeg')
    def stream(self, *args, **kwargs):
        self.feed()

if __name__ == '__main__':
    Stream()
```

Here the main difference is the class to heritage, `EEGStream`, and the `self.feed()` method. In this bare minimum example the constructor generates an empty plot using standard *matplotlib* and them the `loop_consumer` will serve the data to the developer disposition.

```
@loop_consumer('eeg')
def stream(self, data):
    eeg, aux = data

    ch0 = eeg[0]  # first eeg channel
    self.line.set_ydata(ch0)
    self.axis.set_ylim(ch0.min(), ch0.max())

    time = range(len(ch0))  # time axis
    self.line.set_xdata(time)
    self.axis.set_xlim(time[0], time[-1])

    self.feed()
```

The `loop_consumer` is used to update asynchronously the plot, since this method is executed on every new package received, this continuous update will create a plotting animation. The `self.feed()` method send an update to the plotting stream server.

4.4 Stimuli delivery

The interface for stimuli delivery is the only that interact directly with the *patient*, neurophysiological experiments requires a controlled environment with the purpose to decrease the artifacts [60, 61] in the signal as well as keep the patient concentrated on his task. This requirements suggest that the stimuli delivery must work over a remote presentation system and, in this way, separate physically the *user* from the *patient*. The method selected to develop an environment with these features was the classic web application based on HTML, CSS and JavasScript through the implementation of the Brython-Radiant framework.

Although this is a common feature in almost all neurophysiological experiments designer software, after a series of observations and experience acquiring databases for the SPGR, we propose a brand new environment for the design, implementation and configuration of audio-visual stimuli delivery. Our interface able to the user to design flexible experiments and change the parameter easy and quickly without reprogramming the paradigm. Also, since the acquisition interface is integrated into the framework the database is generated automatically with all respective metadata and synchronized markers. Then the user only has to worry about the experiment while the database is generated in second plane.

The proposal interface still requires that both, the stimuli delivery and the previsualization in the dashboard, update the screen synchronously. Then there are two kind of views, the first one is for the *patient*, this view only includes the stimuli presentation and is the responsible to create the time markers. The second view is for the *user*, additional to the stimuli preview also include a dashboard, this interface able to configure the experiment execution.

4.4.1 Stimuli delivery scripting

This interface environment use *Brython* and the *Radiant framework* as backend for do the web development in a *Python* style.

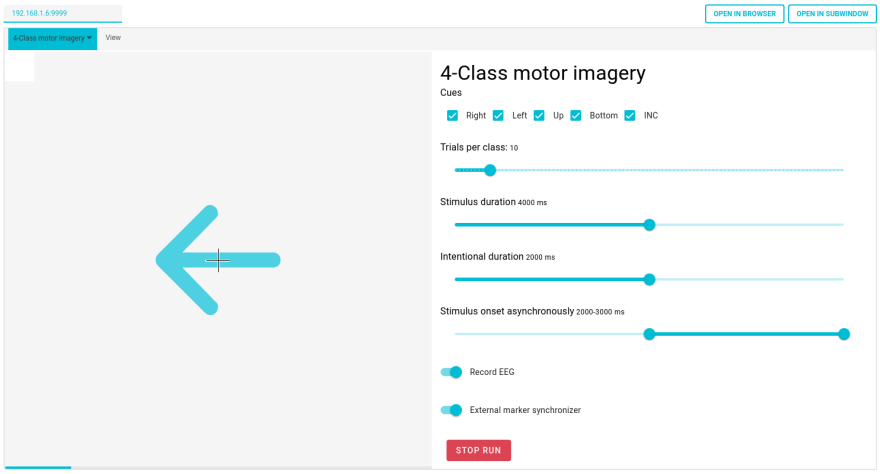


Figure 4-5. Stimuli delivery interface, in the left the live view of the stimulus presented remotely to the patient, in the right the dashboard with the main parameter of the experiment.

```
from bci_framework.extensions.stimuli_delivery import StimuliAPI

# Brython modules
from browser import document, html
from browser.widgets.dialog import InfoDialog

class StimuliDelivery(StimuliAPI):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        document.clear()

        # main brython code
        document.select_one('body') <= html.H3('Hello world')

        button = html.BUTTON('click me')
        button.bind('click', lambda evt: InfoDialog('Hello', 'world'))
        document.select_one('body') <= button

if __name__ == '__main__':
    StimuliDelivery()
```

The `StimuliAPI` includes lot of methods to simplify the interaction with the stimuli delivery environment

Stimuli area and Dashboard

One of the main features is the possibility to make configurable experiments, in favor to this philosophy, by default they are build both areas, `self.stimuli_area` and `self.dashboard` in the class constructor.

The attempt use of the stimuli area is to use it to present the stimulus, this area is automatically duplicated in the remote patient view. The right side is intended to the dashboard, this layout is only accessible for the user, and is used to configure the experiment.

The both areas are accessible thought the class attribute `self.stimuli_area` and `self.dashboard`, as instances of `browser.html.DIV`. There is a particularity in the language about the use of `<=` to nest elements.

4.4.2 Widgets

All widgets and styles are part of *Material Components Web*[14] and can be implemented with a custom module implementation designed to display widgets and get values. All widgets are available troughs the `Widgets` submodule located in the module `bci_framework.extensions.stimuli_delivery.utils`.

```
from bci_framework.extensions.stimuli_delivery.utils import Widgets as w
```

[14] https://material.io/develop/web

Typography

Not only for aesthetics, a default font was selected to make the messages easy to read, even on small screens, in combination with a set of hierarchies and contexts.

```
self.dashboard <= w.label('headline1', typo='headline1')
self.dashboard <= w.label('headline2', typo='headline2')
self.dashboard <= w.label('headline3', typo='headline3')
self.dashboard <= w.label('headline4', typo='headline4')
self.dashboard <= w.label('headline5', typo='headline5')
self.dashboard <= w.label('headline6', typo='headline6')
self.dashboard <= w.label('body1', typo='body1')
self.dashboard <= w.label('body2', typo='body2')
self.dashboard <= w.label('subtitle1', typo='subtitle1')
self.dashboard <= w.label('subtitle2', typo='subtitle2')
self.dashboard <= w.label('caption', typo='caption')
self.dashboard <= w.label('button', typo='button')
self.dashboard <= w.label('overline', typo='overline')
```

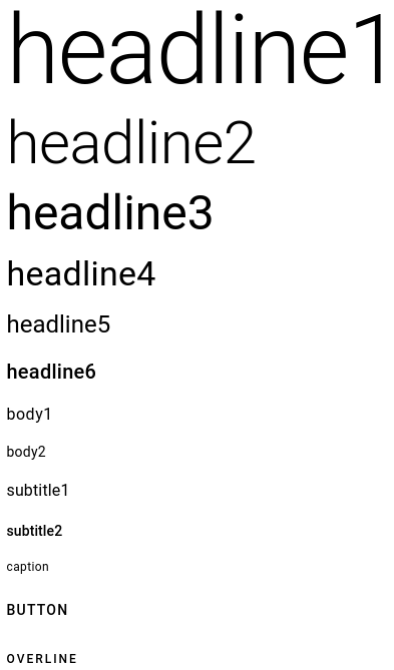


Figure **4-6**. Brython Radiant: Default typography.

Buttons

The buttons have the action `on_click` associated, this action just trigger a method, a function or an anonymous function.

```
        self.dashboard <= w.label('Buttons', typo='headline4', style=flex_title)
        self.dashboard <= w.button(
            'Button 1',
            style=flex,
            on_click=lambda: setattr(
                document.select_one('#for_button'), 'html', 'Button 1 pressed!'
            ),
        )
        self.dashboard <= w.button(
            'Button 2', style=flex, on_click=self.on_button2
        )
        self.dashboard <= w.label(
            f'', id='for_button', typo=f'body1', style=flex
        )

    def on_button2(self):
        document.select_one('#for_button').html = 'Button 2 pressed!'
```

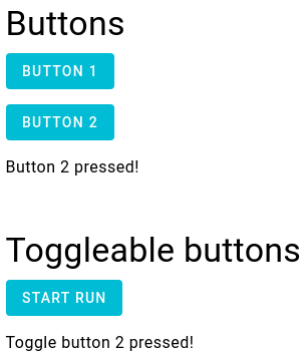


Figure **4-7**. Brython Radiant: Buttons.

Switch

The switch, like the button, have an action associated, but also, a state. The state can be getted on any moment using the `id` with the method `get_value(id)` or in the argument of the binded action.

```
        self.dashboard <= w.label('Switch', typo='headline4', style=flex_title)
        self.dashboard <= w.switch(
            'Switch 1', checked=True, on_change=self.on_switch, id='my_swicth'
        )
        self.dashboard <= w.label(
            f'', id='for_switch', typo=f'body1', style=flex
        )
```

```
def on_switch(self, value):
    # value = self.widgets.get_value('my_swicth')
    document.select_one('#for_switch').html = f'Switch Changed: {value}'
```

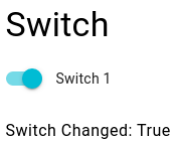


Figure **4-8**. Brython Radiant: Switch.

Checkbox

Multiple selection list items use the checkbox to interact with this list of items, the state comprise the list of selected options, and on_change is triggered on every item status changed.

```
    self.dashboard <= w.label('Checkbox', typo='headline4', style=flex_title)
    self.dashboard <= w.checkbox(
        'Checkbox',
        options=[[f'chb-{i}', False] for i in range(4)],
        on_change=self.on_checkbox,
        id='my_checkbox',
    )
    self.dashboard <= w.label(
        f'', id='for_checkbox', typo=f'body1', style=flex
    )


def on_checkbox(self):
    value = w.get_value('my_checkbox')
    document.select_one('#for_checkbox').html = f'Checkbox Changed: {value}'
```

Figure **4-9**. Brython Radiant: Checkbox.

Radios

Similar to checkboxes but, the selection is exclusive to only one item, then the state is a string.

```
        self.dashboard <= w.label('Radios', typo='headline4', style=flex_title)
        self.dashboard <= w.radios(
            'Radios',
            options=[[f'chb-{i}', f'chb-{i}'] for i in range(4)],
            on_change=self.on_radios,
            id='my_radios',
        )
        self.dashboard <= w.label(
            f'', id='for_radios', typo=f'body1', style=flex
        )


    def on_radios(self):
        value = w.get_value('my_radios')
        document.select_one('#for_radios').html = f'Radios Changed: {value}'
```

Figure **4-10**. Brython Radiant: Radios.

Select

The select component comply the same function that the radios, but the interface is displayed to the user using a different widget.

```
        self.dashboard <= w.label('Select', typo='headline4', style=flex)
        self.dashboard <= w.select(
            'Select',
            [[f'sel-{i}', f'sel-{i}'] for i in range(4)],
            on_change=self.on_select,
            id='my_select',
        )
        self.dashboard <= w.label(
            f'', id='for_select', typo=f'body1', style=flex
        )


    def on_select(self, value):
        # value = w.get_value('my_select')
        document.select_one('#for_select').html = f'Select Changed: {value}'
```

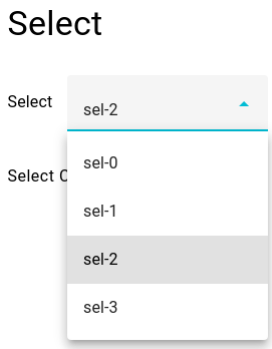


Figure **4-11**. Brython Radiant: Select.

Sliders

The slider are used to select numbers from a range or a range itself. The state is the value or the range and the trigger event is the continuous update, not the change after a release.

```
# Slider
self.dashboard <= w.label('Slider', typo='headline4', style=flex)
self.dashboard <= w.slider(
    'Slider',
    min=1,
    max=10,
    step=0.1,
    value=5,
    on_change=self.on_slider,
    id='my_slider',
)
self.dashboard <= w.label(
    f'', id='for_slider', typo=f'body1', style=flex
)

# Slider range
self.dashboard <= w.label('Slider range', typo='headline4', style=flex)
self.dashboard <= w.range_slider(
    'Slider range',
    min=0,
    max=20,
    value_lower=5,
    value_upper=15,
    step=1,
    on_change=self.on_slider_range,
    id='my_range',
```

```
        )
        self.dashboard <= w.label(f'', id='for_range', typo=f'body1', style=flex)


    def on_slider(self, value):
        # value = w.get_value('my_slider')
        document.select_one('#for_slider').html = f'Slider Changed: {value}'


    def on_slider_range(self, value):
        # value = w.get_value('my_slider')
        document.select_one('#for_range').html = f'Range Changed: {value}'
```

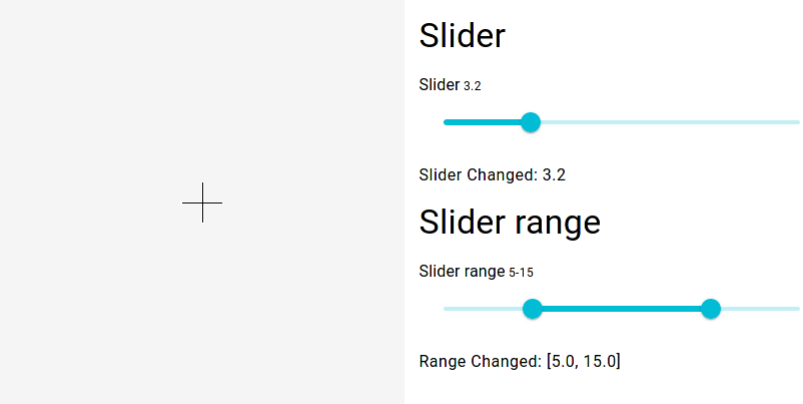


Figure **4-12**. Brython Radiant: Sliders.

4.4.3 Audiovisual stimuli

Since Brython-Radiant is basically a frontend framework, all visual stimuli can be implemented through JavaScript, HTML, and CSS. Web applications are the most featured environ to develop audivisual stimuli. Present and update views, reproduce audio or video are basic tasks in this environment.

Tones and audio

The *Tone* library allows playing single notes using the `javascript AudioContext backend`, the duration and the gain can also be configured.

```
from bci_framework.extensions.stimuli_delivery.utils import Tone as t

duration = 100
gain = 0.5

self.dashboard <= w.button(
    'f#4',
    on_click=lambda: t('f#4', duration, gain),
    style={'margin': '0 15px'},
)
self.dashboard <= w.button(
    'D#0',
    on_click=lambda: t('D#0', duration, gain),
    style={'margin': '0 15px'},
)
self.dashboard <= w.button(
    'B2',
    on_click=lambda: t('B2', duration, gain),
    style={'margin': '0 15px'},
)
```

The Audio module allows to reproduce complete audio files.

```
from bci_framework.extensions.stimuli_delivery.utils import Audio as a

a.load('rain.wav')

self.dashboard <= w.label(
    'Audio',
    'headline4',
    style={
        'margin-bottom': '15px',
        'display': 'flex',
    },
)
self.dashboard <= html.BR()

self.dashboard <= w.slider(
    'Gain',
    min=0,
    max=1,
    step=0.1,
    value=0.5,
    id='gain',
    on_change=a.set_gain,
)

self.dashboard <= w.button(
    'Start', on_click=a.play, style={'margin': '0 15px'}
```

```
)
self.dashboard <= w.button(
    'Stop', on_click=a.stop, style={'margin': '0 15px'}
)
```

Visual stimuli

A complete free HTML-based environment available to design and implement neurophysiological experiments.

```
from bci_framework.extensions.stimuli_delivery.utils import icons

self.dashboard <= icons.fa('fa-arrow-right')   # FontAwesome
self.dashboard  <= icons.bi('bi-arrow-right')  # Bootstrap icon
self.dashboard  <= icons.mi('face', size=24)   # Material icons
```

4.4.4 Stimuli delivery pipeline

Pipelines consist of the controlled execution of methods with asynchronous timeouts. We define a paradigm as a sequence of trials, and each trial is composed of a sequence of parameterized views.

For example, in the classic motor imagery paradigm with two classes, right and left. A trial is the presentation of one stimulus, each trial is composed of a sequence of views: clear screen, wait, show stimulus, clear screen. And then, the next trial with the same sequence of views but with different parameters, like the stimulus itself, or random waits.

This task can be implemented programmatically with the pipeline design pattern and simplified the implementation. First one we need to define the trials like a list of parameter used for each one of the single trial. For example, here we define 3 trials:

```
trials = [
    {'s1': 'Left',  # Trial 1
     'r1': 91,
     },

    {'s1': 'Right',  # Trial 2
     'r1': 85,
     },

    {'s1': 'Left',  # Trial 3
     'r1': 30,
     },
]
```

And the trial views consists of a list of sequential methods with a respective timeout (method, timeout), if the timeout is a number then this will indicate the milliseconds until the next method call. If the timeout is a list, then a random (with uniform distribution) number between that range will be generated on each trial.

```
view = [
    (self.view1, 500),
    (self.view2, [500, 1500]),
    (self.view3, w.get_value('slider')),
    (self.view4, w.get_value('range')),
]
```

The view configuration can used states from widgets by referencing the `id` instead of an explicit value:

```
view = [
    (self.view1, 500),          # timeout
    (self.view2, [500, 1500]),  # range
    (self.view3, 'slider'),     # from widgets id
    (self.view4, 'range'),
]
```

Then, we need to define the views methods, each method is a step needed to build a single trial. By definition, all views will share the same arguments, even if they are not used by that method.

```
def view1(self, s1, r1):
    print(f'On view1: {s1=}, {r1=}')

def view2(self, s1, r1):
    print(f'On view2: {s1=}, {r1=}')

def view3(self, s1, r1):
    print(f'On view3: {s1=}, {r1=}')

def view4(self, s1, r1):
    print(f'On view4: {s1=}, {r1=}\n')
```

Finally, our pipeline can be executed with the method `self.run_pipeline`:

```
self.run_pipeline(view, trials)
```

Here, the pipeline is running all tree trials and for each one the four views, notice that each view receive all the parameter that configure the single trial.

```
On view1: s1=Left, r1=91
On view2: s1=Left, r1=91
On view3: s1=Left, r1=91
On view4: s1=Left, r1=91

On view1: s1=Right, r1=85
On view2: s1=Right, r1=85
On view3: s1=Right, r1=85
On view4: s1=Right, r1=85

On view1: s1=Left, r1=30
On view2: s1=Left, r1=30
On view3: s1=Left, r1=30
On view4: s1=Left, r1=30
```

In addition to handle the organization and planning execution of the trials, the main purpose of the pipelines is to synchronize the remote presentation. Figure **4-13** shows the composition of a basic trial, each view needs a time to complete their execution, the pipeline system ensure that times the between $T0 - T1$, $T1 - T2$, and $T2 - T3$, remains the same for all executions, no matter the fluctuations of the view execution.

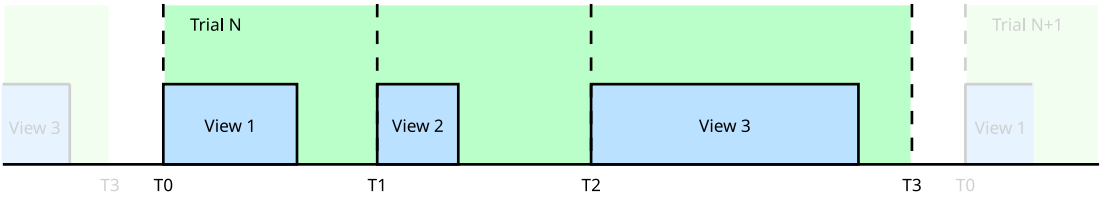


Figure **4-13**. Each trial is composed of views, the pipelines features define the asynchronous execution of each view at the precise time.

4.4.5 Hardware-based event synchronization

The stimuli delivery interface integrate a physical method to synchronize markers, this method used an small portion of the same screen that present the stimulus. In this area a sensor is attached and connected directly with the acquisition system in order to obtain a single stream with both signals, EEG and the external ones using the *analog boardmode* supported by *OpenBCI*.

4.5 Markers, commands, annotations and feedbacks

In order to integrate and intercommunicate all environments a set of custom channels for messaging was defined, this commands are functional in all environs: Data analysis, visualization and Stimuli delivery. The *markers*, *annotations*, and *feedbacks* interacts internally in the framework in order to configure the framework, intercommunicate process or create registers over the acquired signal. Unlike *commands* that consist of a simple recommendation to standardize the interaction of the framework environment with external process.

Markers

Used to define events over the EEG signal, needs a label and optionally a blink time in milliseconds.

```
self.send_marker("MARKER")
self.send_marker("MARKER", blink=200)
```

The timestamp for each marker is generated internally in order to the use the main Kafka clock reference, this strategy avoids to use multiple clocks sources and generate latencies with differents trends. Large latency with small jitter is always preferred over small latency with large jitter.

Annotations

Used to define events over the patient, needs a description and optionally a duration.

```
self.send_annotation('Data record start')
self.send_annotation('The subject yawn', duration=5)
```

This format is exported to the EDF standard annotations. Like in markers the timestamp is generated internally.

Commands

This is a private framework topic defined in Kafka used to share information with external devices, the main purpose is about to close the loop for complete BCI systems. Need a label and a value, the value can be also any Python data structure.

```
self.send_command('MyCommand', value=45)
self.send_command('MyCommand', value={'A':45, 'B':12})
```

This messages use the `'command'` topic to share the information.

Feedbacks

The feedbacks are used to communicate the Data analysis and Data visualizations with the Stimuli Delivery platform with asynchronous callbacks. For this purpose, there is a predefined stream channel called feedback. This is useful to develop Neurofeedback applications.

The asynchronous handler can be configured with the Feedback class:

```
from bci_framework.extensions.stimuli_delivery import Feedback
```

This object needs an identifier and optionally callback method to handle the input messages.

```
self.feedback = Feedback(self, 'my_feedback_id')  # ID
self.feedback.on_feedback(self.on_input_feedback)  #callback
```

The callback method will receive the input data as keyword only arguments.

```
def on_input_feedback(self, **feedback):
    ...
```

And the method `self.feedback.write` is the intented way to send feedbacks to the complementary environment.

```
self.feedback.write(**feedback)
```

This code structure must be on both endpoints, with the same *ID* in order to communicate them and exchange feedback information

Figure **4-14**. BCI-Framework: Marker synchronization real-time interface.

4.6 Latency analysis and event marker synchronization

Is possible to take advantage of the OpenBCI implementation, this board includes a set of analog and digital inputs that can be used to synchronize markers. In order to use OpenBCI, we will only need an light-dependent resistor (LDR) module connected to the pin D11 (or A5) and start the automatic latency correction system included in BCI-Framework.

BCI-Framework integrates an interface to measure latencies and synchronize markers (Figure **4-14**), it was designed to be used on distributed environments. This simple latency correction consists of a stimuli delivery with only a marker synchronization area, the LDR module is constantly sensing (the boardmode must be in analog mode) so the changes on the square signal are compared with streamed markers and then the latency is corrected. The latency correction only affects the current instance, if BCI-Framework is restarted this calibration will be lose. For hard event synchronization, is prefer to use the markers synchronization constantly during all session.

4.7 Close the loop and Neurofeedback

The BCI application with a close loop implementation consist of process in real-time the acquired data in order to execute commands in the real world. Although physiologically the close-loop signals differs from neurofeedback signals, since the nature of the data is different, the programmatically handle is exactly the same and are handled by Feedbacks. Once the feedback is defined through a method to perform asynchronous communication between them, the neurofeedback approaches can be implemented using this features. This implementation can be designed to run inside the interface, using Real-time data analysis or to run isolated in no contextualized process. In Appendix F.3 shows the implementation of a neurofeedback system.

The advantage to run process inside the framework is that this scrips share all the environment variables, and can be monitored through the main interface. On the other hand the scripts that run outside the framework only have access to the Kafka streams, this means that only can use the acquire data and are agnostics about everything else.

4.8 Summary and discussion

BCI-Framework is the convergence of a set of drivers and tools working together to serve to the *user* a full environment to acquire EEG signals and perform neurophysiological experiments with reliability and flexibility. Advance user can use the framework to develop custom visualizations and design paradigms that satisfy their needs using the integrated development environment. The main development was performed using Python, this feature brings to the framework one of the most complete computational libraries to work with data analysis and machine learning highly used to develop BCI systems today.

The integrated development environment serve a full API with the main tasks automatized. The real-time data analysis and visualization build in background a fast system to buffering, sub-sampling and slice incoming EEG data en real-time. The stimuli delivery scripting has been designed to present views to the *patient* asynchronously. In all cases the *developer* only must care about the specific implementation of the analysis, visualizations and paradigm designed.

However the system only supports one acquisition board, *OpenBCI Cyton*, although this is one of the most flexible and featured options available at the moment to write this thesis, the spreading of the framework tool could be seriously compromised. Also the main operating system focused, any based on GNU/*Linux*, could be a reason that limits the adoption of the framework by the community.

CHAPTER

FIVE

FINAL REMARKS

5.1 Conclusions and discussion

For this work were identified all components needed to get flexible, scalable, and integral BCI system. Flexibility to adapt and modify experiments, make fast changes in runtime, and process data in real-time; Scalability in the execution of data analysis distributing expensive task without affect the main acquisition process; A framework that integrate a full environment with almost all tools to develop complete research-grade BCI systems.

In order to guarantee all promised high features, a single board (*OpenBCI Cyton*) was choose to be configured and controlled in deep. Then, a brand new set of drivers was developed with the capability to take advantage of the hardware and all benefits of the *ADS1299*. The acquisition system support multiple sampling rate, packaging size, communication protocol, and free electrodes placement for use not only for EEG but ECG and EMG. Additional to this, a unique feature to synchronize markers had been included using the low levels characteristics of the acquisition board.

Unlike the centralized systems that share the resources as well the stability. The distributed systems allow the controlled execution of a set of critical process like: The acquisition, the implementation of a dedicated system to handle the interaction with the OpenBCI hardware brings to the system robustness; the stimuli delivery, that allows pull apart the rendering and audiovisual generation to be able to stream markers and annotations in accurate times; and real-time data analysis, from experimental and not debugged scripts without the worry of causing exceptions in the system. Although it is a distributed system, the real-time streaming is guaranteed, the latency and the jitter keeps in accepted ranges for closed-loop BCI systems, and acquisition methodology ensure that the bad sampling data can be labeled and processed as appropriate.

The development environment contribute with a full API and an automatic background to configure common task in the field of BCI data processing like: buffering, sub-sampling and real-time trials slicing; an easy-to-use set of widgets to build dashboards for stimuli delivery; an environment to develop and debug custom extensions; and an interface to integrate the user develops alongside other extension at the same time. Implement neurofeedback paradigms and close-the-loop represent the most demanding and interesting tasks supported already in the system.

5.2 Future work

We have presented a framework to develop BCI systems with a lot of new features that are not present in state-of-art. However, there are still many issues that can be addressed to improve the performance, acceptance and the wide spreading of our system. In particular, the following aspects could be of interest for future work:

- The electrodes density has always been one important discussion [62, 63, 64] in the field of BCI systems. The proposed acquisition method has the potential to be parallelized and multiply the number of electrodes.

- As well the selected board for this work, *OpenBCI Cyton*, is one of the hardware with best performance and configurability, there is necessary a new acquisition board that integrates the most recent technology and communication protocols in a single board.
- The SPRG has recently interest in clinic multi-modal acquisition, BCI-Framework can be turn into a new framework to acquire and process real-time philological signals from multiple sources and serve visualizations, diagnostic support or store data.
- Although the system has been proven under specific applications and some databases has been generated (Appendix: Motor imagery, Appendix: Visuospatial working memory - Change detection) there is necessary more integration and validation with the methodologies developed the group.

5.3 Academic products

5.3.1 Journal papers

Paper submitted to *SoftwareX - Journals | Elsevier* with the name "A real-time acquisition, visualization, and stimuli delivery Python-based tool for neurophysiological experiments"

5.3.2 Patents

The systems was submitted to the *Crearlo no es suficiente* summons for a *patentability search process* with the *Universidad Nacional de Colombia sede Manizales* as main beneficiary, with the title "MÉTODO Y SISTEMA PARA LA SINCRONIZACIÓN DE MARCADORES ASOCIADOS A SISTEMAS DE INTERFAZ CEREBRO-COMPUTADOR", postulation ID 343 and Application number *NC2022/0007405* from May 28, 2022.

5.3.3 Software registers

A script developed with BCI-Framework for motor imagery paradigm based on games stimulus (Pacman interface), was submitted to software register in the textitUniversidad Nacional de Colombia sede Manizales.

APPENDIX

A

PYTHON: SYSTEMD SERVICE

Description: Simple API to automate the creation of custom daemons for GNU/Linux.

License: BSD-2-clause

Latest version: `1.8`

Python: `3.8, 3.9, 3.10`

PyPi: `https://pypi.org/project/systemd-service/`

Repository: `https://github.com/UN-GCPDS/systemd-service`

Documentation: `https://systemd-service.readthedocs.io/en/latest/`

A daemon is a service process that runs in the background and supervises the system or provides functionality to other processes. Traditionally, daemons are implemented following a scheme originating in SysV Unix [65]. Modern daemons should follow a simpler yet more powerful scheme, as implemented by systemd [66].

Systemd service is a Python module to automate the creation of Python-based daemons under GNU/Linux environments.

A.1 Install

```
pip install -U systemd-service
```

A.2 Handle daemons

```
from systemd_service import Service

daemon = Service("stream_rpyc")

daemon.stop()     # Start (activate) the unit.
daemon.start()    # Stop (deactivate) the unit.
daemon.reload()   # Reload the unit.
daemon.restart()  # Start or restart the unit.

daemon.enable()   # Enable the unit.
daemon.disable()  # Disable the unit.

daemon.remove()   # Remove the file unit.
```

This commands are uquivalent to the `systemctl` calls, for example run in terminal the folowing command:

```
\$ systemctl enable stream_rpyc
```

Can be running inside a Python environment with using `systemd_service`

```
from systemd_service import Service

daemon = Service("stream_rpyc")
daemon.enable()
```

A.3 Creating services

Similar to the previous scripts, the services can be created using `systemd_service`:

```
daemon = Service("stream_rpyc")
daemon.create_service()
```

If the service must be initialized after other service

```
daemon = Service("stream_rpyc")
daemon.create_service(after='ntpd')
```

A.4 Creating timers

Defines a timer relative to when the machine was booted up:

```
daemon = Service("stream_rpyc")
daemon.create_timer(on_boot_sec=15)
```

A.5 Example

This module is useful when is combined with package scripts declaration in `setup.py` file:

```
# setup.py

scripts=[
    "cmd/stream_rpyc",
]
```

The script could looks like:

```
#!/usr/bin/env python

import sys

if sys.argv[-1] == "systemd":
    from systemd_service import Service
    daemon = Service("stream_rpyc")
    daemon.create_timer(on_boot_sec=10, after='network.target kafka.service')

else:
    from my_module.submodule import my_service
    print("Run 'stream_rpyc systemd' as superuser to create the daemon.")
    my_service()
```

Then the command can be called as a simple script but with the `systemd` argument the command will turn into a service.

```
\$ stream_rpyc
# Command executed normally


\$ stream_rpyc systemd
# Service created
```

APPENDIX

B

PYTHON: QT-MATERIAL

Description: Material inspired stylesheet for PySide2, PySide6, PyQt5 and PyQt6.

License: BSD-2-clause

Latest version: 2.12

Python: 3.8, 3.9, 3.10

PyPi: https://pypi.org/project/qt-material/

Repository: https://github.com/UN-GCPDS/qt-material

Documentation: https://qt-material.readthedocs.io/en/latest/

This is another stylesheet for PySide6, PySide2, PyQt5 and PyQt6, which looks like Material Design (close enough).

B.1 Install

```
pip install -U qt-material
```

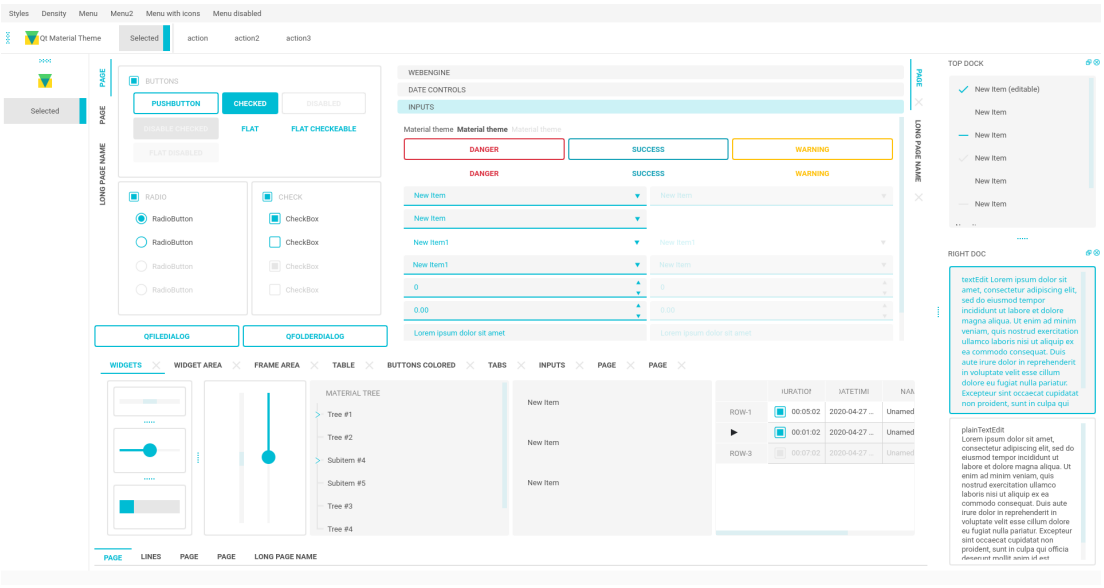


Figure **B-1**. *light_cyan_500.xml* theme for Qt-Material.

B.2 Usage

```
import sys
from PySide6 import QtWidgets
# from PySide2 import QtWidgets
# from PyQt5 import QtWidgets
from qt_material import apply_stylesheet

# create the application and the main window
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QMainWindow()

# setup stylesheet
apply_stylesheet(app, theme='dark_teal.xml')

# run
window.show()
app.exec_()
```

B.3 Themes

```
from qt_material import list_themes
```

```
list_themes()
```

```
['dark_amber.xml',
 'dark_blue.xml',
 'dark_cyan.xml',
 'dark_lightgreen.xml',
 'dark_pink.xml',
 'dark_purple.xml',
 'dark_red.xml',
 'dark_teal.xml',
 'dark_yellow.xml',
 'light_amber.xml',
 'light_blue.xml',
 'light_cyan.xml',
 'light_cyan_500.xml',
 'light_lightgreen.xml',
 'light_pink.xml',
 'light_purple.xml',
 'light_red.xml',
 'light_teal.xml',
 'light_yellow.xml']
```

B.4 Custom colors

Color Tool[1] is the best way to generate new themes, just choose colors and export as `Android XML`, the theme file must look like:

```
<!--?xml version="1.0" encoding="UTF-8"?-->
<resources>
<color name="primaryColor">#00e5ff</color>
<color name="primaryLightColor">#6effff</color>
<color name="secondaryColor">#f5f5f5</color>
<color name="secondaryLightColor">#ffffff</color>
<color name="secondaryDarkColor">#e6e6e6</color>
<color name="primaryTextColor">#000000</color>
<color name="secondaryTextColor">#000000</color>
</resources>
```

Save it as `my_theme.xml` or similar and apply the style sheet from Python.

```
apply_stylesheet(app, theme='dark_teal.xml')
```

[1] https://material.io/resources/color/!/?view.left=0view.right=0

B.5 Light themes

Light themes will need to add `invert_secondary` argument as `True`.

```
apply_stylesheet(app, theme='light_red.xml', invert_secondary=True)
```

B.6 Environ variables

There is a environ variables related to the current theme used, these variables are for consult purpose only.

Environ variable	Description	Example
`QTMATERIAL_PRIMARYCOLOR`	Primary color	`#2979ff`
`QTMATERIAL_PRIMARYLIGHTCOLOR`	A bright version of the primary color	`#75a7ff`
`QTMATERIAL_SECONDARYCOLOR`	Secondary color	`#f5f5f5`
`QTMATERIAL_SECONDARYLIGHTCOLOR`	A bright version of the secondary color	`#ffffff`
`QTMATERIAL_SECONDARYDARKCOLOR`	A dark version of the primary color	`#e6e6e6`
`QTMATERIAL_PRIMARYTEXTCOLOR`	Color for text over primary background	`#000000`
`QTMATERIAL_SECONDARYTEXTCOLOR`	Color for text over secondary background	`#000000`
`QTMATERIAL_THEME`	Name of theme used	`"light_blue.xml"`

Table **B-1**. Environ variables defined by Qt-Material.

B.7 Alternative QPushButtons and custom fonts

There is an `extra` argument for accent colors and custom fonts.

```
extra = {

    # Button colors
    'danger': '#dc3545',
    'warning': '#ffc107',
    'success': '#17a2b8',

    # Font
    'font_family': 'Roboto',
}

apply_stylesheet(app, 'light_cyan.xml', invert_secondary=True, extra=extra)
```

The accent colors are applied to `QPushButton` with the corresponding `class` property:

```
pushButton_danger.setProperty('class', 'danger')
pushButton_warning.setProperty('class', 'warning')
pushButton_success.setProperty('class', 'success')
```

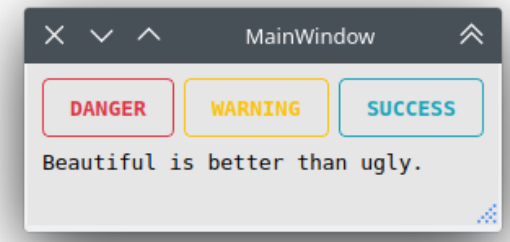


Figure **B-2**. QPushButtons stylized with class property.

B.8 Custom stylesheets

Custom changes can be performed by overwriting the stylesheets, for example:

```
QPushButton {{
  color: {QTMATERIAL_SECONDARYCOLOR};
  text-transform: none;
  background-color: {QTMATERIAL_PRIMARYCOLOR};
}}

.big_button {{
  height: 64px;
}}
```

Then, the current stylesheet can be extended just with:

```
apply_stylesheet(app, theme='light_blue.xml')

stylesheet = app.styleSheet()
with open('custom.css') as file:
    app.setStyleSheet(stylesheet + file.read().format(**os.environ))
```

And the class style can be applied with the `setProperty` method:

```
self.main.pushButton.setProperty('class', 'big_button')
```

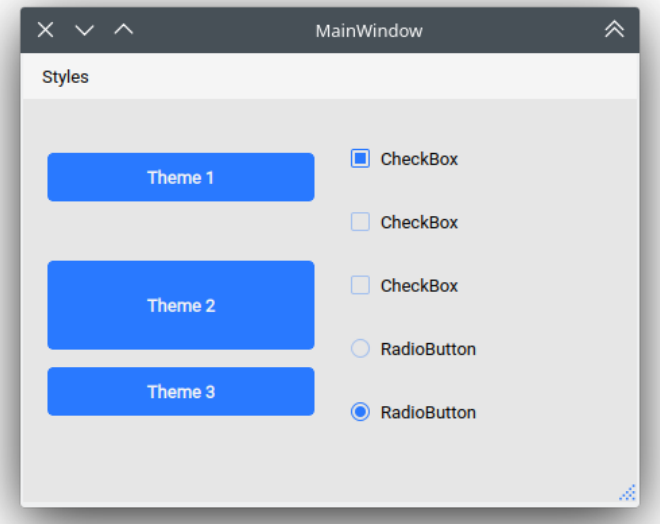


Figure **B-3**. QPushButtons stylized with user defined class property.

B.9 Run examples

A window with almost all widgets (see the previous screenshots) are available to test all themes and create new ones.

```
git clone https://github.com/UN-GCPDS/qt-material.git
cd qt-material
python setup.py install
cd examples/full_features
python main.py --pyside6
```

B.10 Change theme in runtime

There is a `qt_material.QtStyleTools` class that must be inherited along to QMainWindow for change themes in runtime using the `apply_stylesheet()` method.

```
class RuntimeStylesheets(QMainWindow, QtStyleTools):

    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        self.apply_stylesheet(self.main, 'dark_teal.xml')
        # self.apply_stylesheet(self.main, 'light_red.xml')
        # self.apply_stylesheet(self.main, 'light_blue.xml')
```

B.11 Integrate stylesheets in a menu

A custom stylesheets menu can be added to a project for switching across all default available themes.

```
class RuntimeStylesheets(QMainWindow, QtStyleTools):

    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        self.add_menu_theme(self.main, self.main.menuStyles)
```

B.12 Create new themes

A simple interface is available to modify a theme in runtime, this feature can be used to create a new theme, the theme file is created in the main directory as `my.xml`

```
class RuntimeStylesheets(QMainWindow, QtStyleTools):

    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        self.show_dock_theme(self.main)
```

A full set of examples are available in the exmaples directoryhttps://github.com/UN-GCPDS/qt-material/blob/master/examples/runtime/

B.13 Export theme

This feature able to use *Qt-Material* themes into `Qt` implementations using only local files.

```
from qt_material import export_theme

extra = {

    # Button colors
    'danger': '#dc3545',
    'warning': '#ffc107',
    'success': '#17a2b8',

    # Font
    'font_family': 'monoespace',
    'font_size': '13px',
    'line_height': '13px',

    # Density Scale
    'density_scale': '0',

    # environ
    'pyside6': True,
```

```
    'linux': True,

}

export_theme(theme='dark_teal.xml',
             qss='dark_teal.qss',
             rcc='resources.rcc',
             output='theme',
             prefix='icon:/',
             invert_secondary=False,
             extra=extra,
            )
```

This script will generate both `dark_teal.qss` and `resources.rcc` and a folder with all theme icons called `theme`.

The files generated can be integrated into a `PySide6` application just with:

```
import sys

from PySide6 import QtWidgets
from PySide6.QtCore import QDir
from __feature__ import snake_case, true_property

# Create application
app = QtWidgets.QApplication(sys.argv)

# Load styles
with open('dark_teal.qss', 'r') as file:
    app.style_sheet = file.read()

# Load icons
QDir.add_search_path('icon', 'theme')

# App
window = QtWidgets.QMainWindow()
checkbox = QtWidgets.QCheckBox(window)
checkbox.text = 'CheckBox'
window.show()
app.exec()
```

This files can also be used into non `Python` environs like `C++`.

B.14 Density scale

The `extra arguments` also include an option to set the *density scale*, by default is `0`.

```
extra = {

    # Density Scale
    'density_scale': '-2',
}

apply_stylesheet(app, 'default', invert_secondary=False, extra=extra)
```

APPENDIX

C

PYTHON: MATPLOTLIB-FIGURESTREAM

Description: A backend for serve Matplotlib animations as web streams.

License: BSD-2-clause

Latest version: 1.2.6

Python: 3.8, 3.9, 3.10

PyPi: https://pypi.org/project/figurestream/

Repository: https://github.com/UN-GCPDS/matplotlib-figurestream

Documentation: https://figurestream.readthedocs.io/en/latest/

C.1 Install

```
pip install -U figurestream
```

C.2 Bare minimum

By default, the stream serves on `http://localhost:5000`

```
# FigureStream replace any Figure object
from figurestream import FigureStream

import numpy as np
from datetime import datetime

# FigureStream can be used like any Figure object
stream = FigureStream()
sub = stream.add_subplot(111)
x = np.linspace(0, 3, 1000)

# Update animation loop
while True:
    sub.clear()  # clear the canvas

    # ------------------------------------------------------------------------
    # Any plot operation
    sub.set_title('FigureStream')
    sub.set_xlabel('Time [s]')
    sub.set_ylabel('Amplitude')
    sub.plot(x, np.sin(2 * np.pi * 2 * (x + datetime.now().timestamp())))
    sub.plot(x, np.sin(2 * np.pi * 0.5 * (x + datetime.now().timestamp())))
    # ------------------------------------------------------------------------

    stream.feed()  # push the frame into the server
```

For fast updates is recommended to use `set_data`, `set_ydata` and `set_xdata` instead of clear and draw again in each loop, also `FigureStream` can be implemented from a custom class.

```
# FigureStream replace any Figure object
from figurestream import FigureStream

import numpy as np
from datetime import datetime


class FastAnimation(FigureStream):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

```
            axis = self.add_subplot(111)
            self.x = np.linspace(0, 3, 1000)

            # ---------------------------------------------------------------
            # Single time plot configuration
            axis.set_title('FigureStream')
            axis.set_xlabel('Time [s]')
            axis.set_ylabel('Amplitude')

            axis.set_ylim(-1.2, 1.2)
            axis.set_xlim(0, 3)

            # Lines objects
            self.line1, *_ = axis.plot(self.x, np.zeros(self.x.size))
            self.line2, *_ = axis.plot(self.x, np.zeros(self.x.size))
            # ---------------------------------------------------------------

            self.anim()

        def anim(self):
            # Update animation loop
            while True:
                # -----------------------------------------------------------
                # Update only the data values is faster than update all the plot
                self.line1.set_ydata(
                    np.sin(2 * np.pi * 2 * (self.x + datetime.now().timestamp()))
                )
                self.line2.set_ydata(
                    np.sin(
                        2 * np.pi * 0.5 * (self.x + datetime.now().timestamp())
                    )
                )
                # -----------------------------------------------------------

                self.feed()  # push the frame into the server


    if __name__ == '__main__':
        FastAnimation()
```

C.3 Set host, port and endpoint

If we want to serve the stream in a different place we can use the parameters `host`, `port` and `endpoint`, for example:

```
FigureStream(host='0.0.0.0', port='5500', endpoint='figure.jpeg')
```

Now the stream will serve on `http://localhost:5500/figure.jpeg` and due the `0.0.0.0` host is accesible for any device on network. By default `host` is `localhost`, `port` is `5000` and `endpoint` is empty.

APPENDIX

D

PYTHON/BRYTHON: RADIANT FRAMEWORK

Description: A Brython Framework for Web Apps development.

License: BSD-2-clause

Latest version: `3.3.8`

Python: `3.8, 3.9, 3.10`

PyPi: `https://pypi.org/project/radiant/`

Repository: `https://github.com/UN-GCPDS/brython-radiant`

Documentation: `https://radiant-framework.readthedocs.io/en/latest/`

Radiant is a *Brython*[1] framework for the quick development of web apps with pure Python/Brython syntax so there is no need to care about (if you don't want) HTML, CSS, or Javascript. Run over *Tornado*[2] servers and include support to *Websockets*, *Python Scripts* and *MDC*.

[1] https://brython.info/
[2] https://www.tornadoweb.org/

D.1 Install

```
pip install -U radiant
```

D.2 Bare minimum

```
# Radiant modules
from radiant.server import RadiantAPI

# Brython modules
# This modules are faked after `radiant` import
from browser import document, html

# Main class inheriting RadiantAPI
class BareMinimum(RadiantAPI):

    # Constructor
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        #------------------------------------------------------------
        # Brython code (finally)
        document.select_one('body') <= html.H1('Hello World')
        #
        # ...all your brython code
        #------------------------------------------------------------

# Run server
if __name__ == '__main__':
    BareMinimum()
```

D.3 Extra options

```
# Radiant modules
# Import RadiantServer for advanced options
from radiant.server import RadiantAPI, RadiantServer
```

```
from browser import document, html

# Main class inheriting RadiantAPI
class BareMinimum(RadiantAPI):

    def __init__(self, *args, **kwargs):
        """"""
        super().__init__(*args, **kwargs)

        #--------------------------------------------------------
        # Brython code
        document.select_one('body') <= html.H1('Hello World')
        #
        # ...all your brython code
        #--------------------------------------------------------

if __name__ == '__main__':
    # Advance options
    RadiantServer('BareMinimum',
                  host='localhost',
                  port=5000,
                  brython_version='3.9.1',
                  debug_level=0,
                  )
```

D.4 How to works

This is basically a set of scripts that allows the same file run from *Python* and *Brython*, when is running under *Python* a *Tornado* server is created and configure the local path for serving static files, and a custom *HTML* template is configured in runtime to import the same script, this time under *Brython*, is very simple.

D.5 WebSockets

This WebSockets are in the Tornado side and NOT in Brython. So, is basically and *WebSocketHandler*[3] object like:

[3] https://www.tornadoweb.org/en/stable/websocket.html

```
#ws_handler.py

from tornado.websocket import WebSocketHandler

class WSHandler(WebSocketHandler):

    def open(self):
        ...

    def on_close(self):
        ...

    def on_message(self, message):
        ...
```

That can be included with the `RadiantServer` class in the `websockethandler` argument:

```
RadiantServer('MainApp', websockethandler=('ws_handler.py', 'WSHandler'))
```

This websocket will be serving on `/ws` URL.

D.6 Python scripting

This feature is to run a real Python environment through methods that return objects. make sure to inherit `PythonHandler`:

```
#python_foo.py

from radiant import PythonHandler

class MyClass(PythonHandler):

    def local_python(self):
        return "This file are running from Local Python environment"

    def pitagoras(self, a, b):
        return math.sqrt(a ** 2 + b ** 2)
```

This handler can be included with the `RadiantServer` class in the `python` argument:

```
RadiantServer('MainApp', python=('python_foo.py', 'MyClass'))
```

A full example of use could be:

```
from radiant import RadiantAPI, RadiantServer
from browser import document, html

class MainApp(RadiantAPI):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        document.select('body')[0] <= html.H1('Hello World')
        document.select('body')[0] <= html.H3(self.MyClass.local_python())

        a, b = 3, 5
        c = self.MyClass.pitagoras(a, b)
        document.select('body')[0] <= html.H3(f"Pitagoras: {a=}, {b=}, {c=:.3f}")

if __name__ == '__main__':
    RadiantServer('MainApp', python=('python_foo.py', 'MyClass'))
```

D.7 Custom themes

Material themes from MDC can be configured with *Color Tool*[4] application, just select the desired colors, save the file and add it to the `RadiantServer` class in the attribute `theme`.

```
RadiantServer('MainApp', theme='custom_theme.xml')
```

[4] https://material.io/resources/color/

APPENDIX

E

DATABASE: MOTOR IMAGERY

Description: Motor Imagery database.

Subjects: 7

Repository: https://github.com/UN-GCPDS/

Motor imagery (MI) is the process of imagining a motor action without any motor execution. During an MI task, a subject visualizes in their mind an instructed motor action, i.e., to move the right hand, without actually carrying it out. When subjects plan and execute movements, characteristic rhythms in the sensorimotor areas, typically the μ or precentral α rhythm (8–12 Hz) and the β rhythm (13–30 Hz), get activated [67]. That is to say, MI and motor execution share common sensorimotor areas, and both involve envisioning and executing the same motor plan [68]. Although, their neural mechanisms seem to have some differences [69]. Assessing and interpreting MI brain dynamics may contribute to applications like the evaluation of pathological conditions, the rehabilitation of motor functions,

and motor learning and performance [70]. Particularly, much attention has been paid in the literature to BCI systems that can decode MI-associated task patterns, usually captured through scalp EEG signals, and translate them into commands in order to control external devices [71, 67]. One the main limitations for the widespread use of such systems being that about 15–30% of users display BCI illiteracy, i.e. they do not gain enough control over the interfaces, possibly because subjects with poor control performance do not exhibit discriminative task-related changes over the modulation of sensorimotor rhythms during the interval of MI responses [68].

E.1 Paradigm

This cue-based BCI paradigm consisted of up to four different motor imagery tasks, represented by a succession of cues (arrow-shaped) and separated with an asynchronous break. This paradigm used an arrow pointing to the left right, up or bottom, which has been widely used [72, 73].

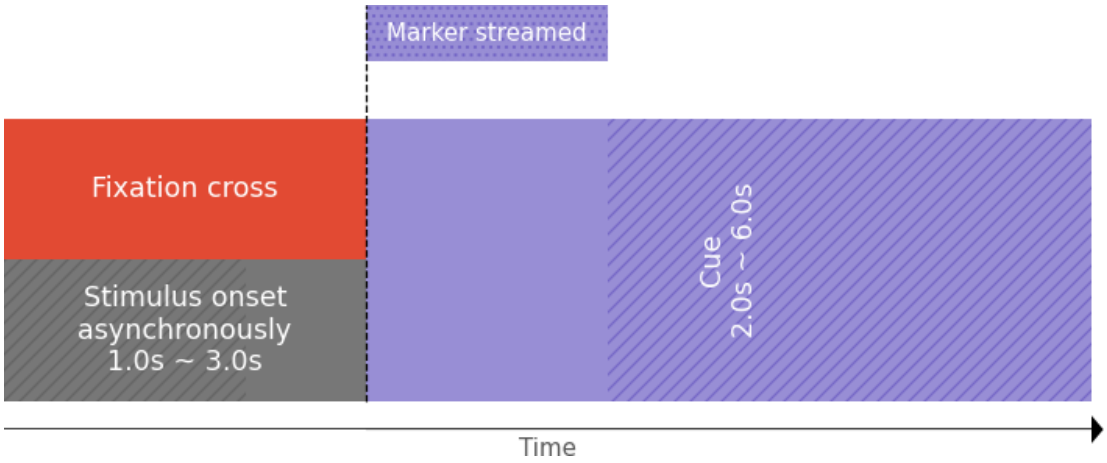


Figure **E-1**. MI paradigm implementation with markers indicators.

E.2 Stimuli presentation

There is two kind of cues for the MI stimuli delivery, the first one is based on the classic arrows and the second one use Pacman-based cues. Both of the paradigms were build with a dashboard that can be used to configure the experiment times.

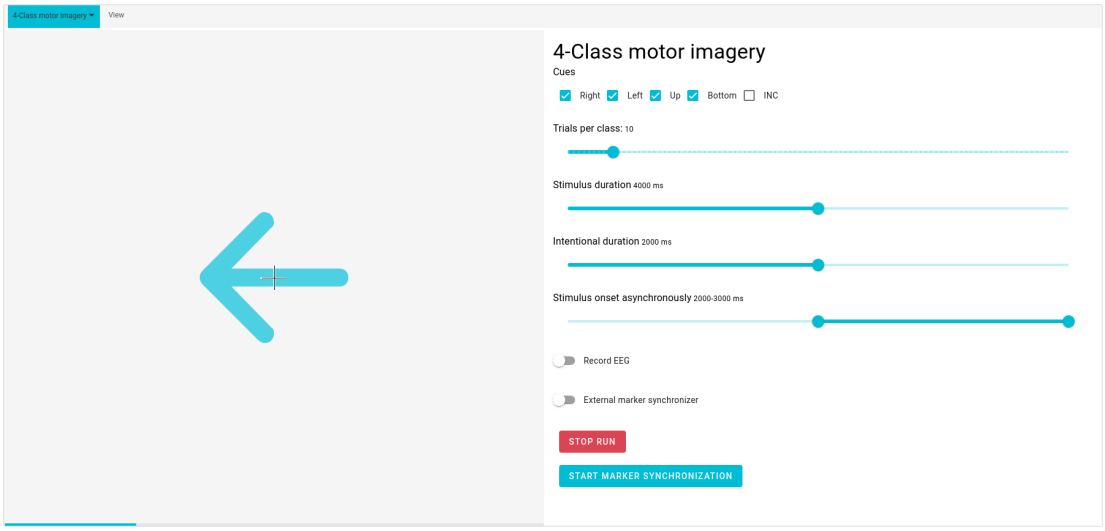


Figure **E-2**. MI stimuli delivery interface with arrow cues.

The Pacman-based cues use a clean interface in order to prevent unintentional stimulation, then all screen indicators like time, score, level and bonus were removed.

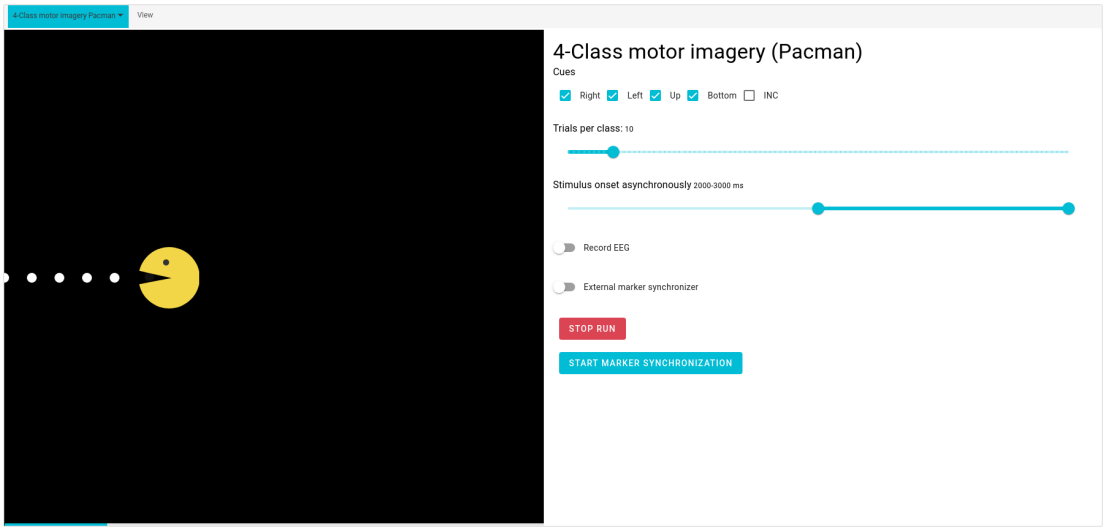


Figure **E-3**. MI stimuli delivery interface with pacman-base cues.

E.3 Intention detection

An additional feature were included into this paradigm, an intentional detection, the cue add an stimuli indicating the next task, the subject must be instructed about not perform any activity. The intentional detection, this task also has interest in the field of the motor execution.

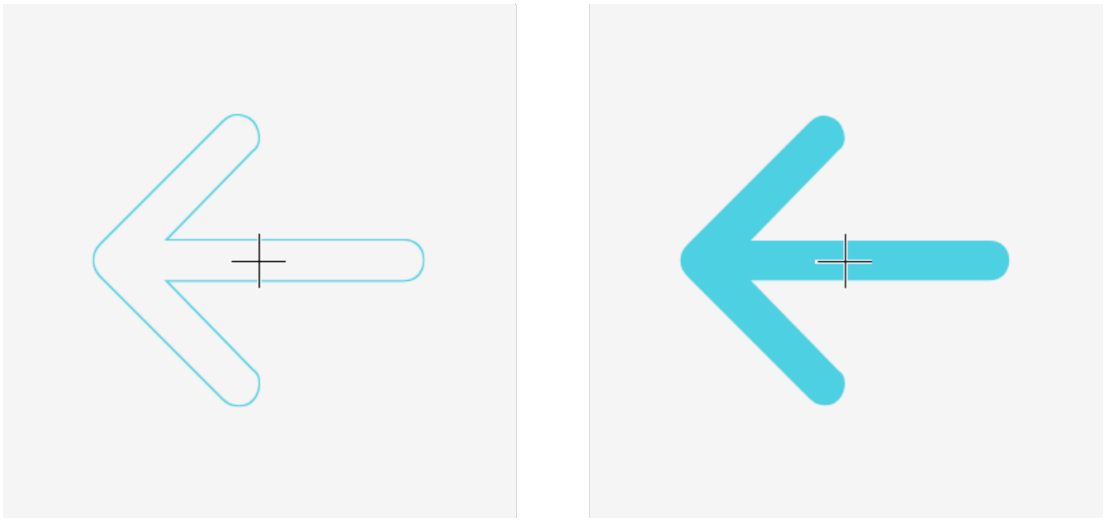


Figure **E-4**. MI with intentional detection.

E.4 Motor imagery with intentional non-control stimulus

Other experimental feature included in the stimuli delivery was the intentional non-control [74], this feature is useful to close the loop, since in real implementations there are situations were the patient do not want to perform any action. This new cue is modeled as a circle.

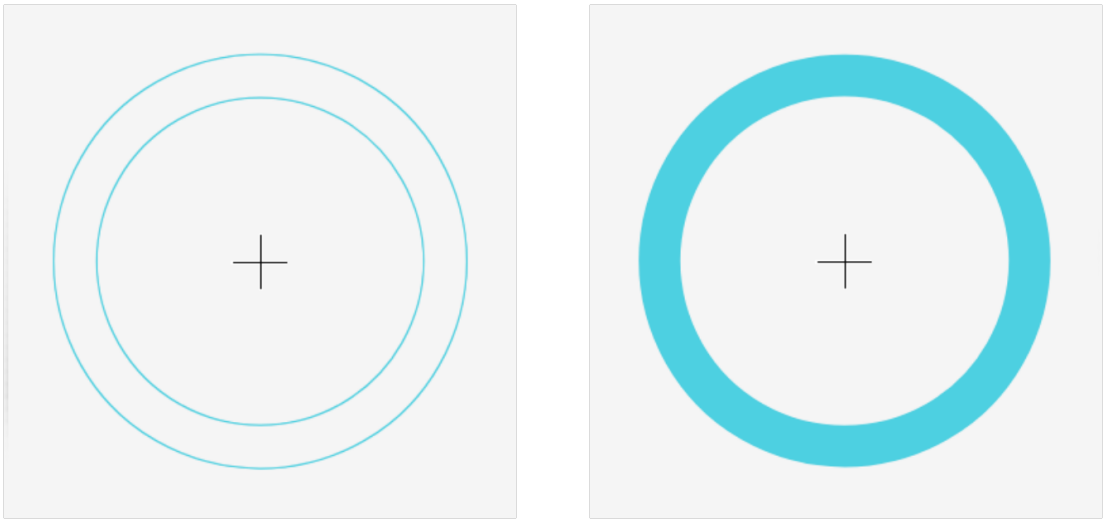


Figure **E-5**. MI with nonintentional stimulus.

APPENDIX

F

DATABASE: VISUOSPATIAL WORKING MEMORY - CHANGE DETECTION TASK

Description: Visuospatial working memory database.

Subjects: 4

Repository: https://github.com/UN-GCPDS/

Visuospatial Working Memory (VWM) is a memory system of limited capacity with the ability to store and manipulate information for a short period of time [75, 76]. It plays a key role in complex cognitive tasks such as comprehension, reasoning, planning and learning [77, 78], as well as in daily activities such as problem solving and decision-making [79]. VWM consists of three distinct stages of information processing: encoding, maintenance or retention, and retrieval [80], with the retention interval being considered as a defining component of VWM, since it differentiates it from other memory types [76].

F.1 Paradigm

The task consists in remembering the colors of a set of squares displayed on a computer screen, termed memory array, and then comparing them with the colors of a second set of squares located in the same positions, termed test array [81]. A trial of the task begins with an arrow indicating either the left or the right side of the screen for 0.2 s. Then, a memory array appears on the screen for 0.1 s. For every trial, memory arrays are displayed on both hemifields, but the subject must remember only those appearing on the side indicated by the arrow cue. Next, after a retention interval lasting 0.9 s, a test array appears for a period of 2 s. During this period the subject reports if the colors of all the items in the memory and test arrays match. The task has three levels according to the number of elements in the memory array: low memory load (one square), medium memory load (two squares), and high memory load (four squares). The subject must perform a total of 300 trials, with 100 trials for each memory load level (50 trials per hemifield). Trials from different levels are presented at random. The color of one of the squares in the test array differs from its counterpart in the memory array in 50% of the trials.

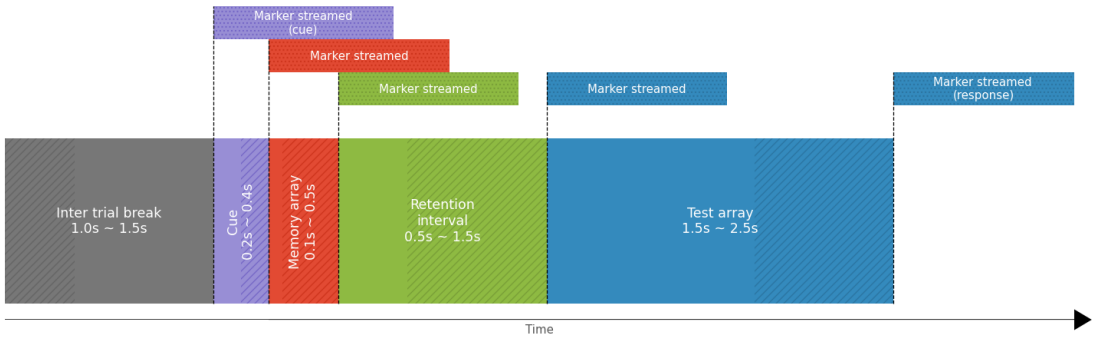


Figure **F-1**. VWM paradigm implementation with markers indicators.

F.2 Stimuli presentation

All stimuli are presented on a computer screen situated 120 cm away from the subject. The stimulus arrays appear within two $7.2° \times 13.15°$ rectangular regions that are centered $5.4°$ to the left and right of a central fixation cross on a gray background (the symbol $\circ$ stands for degrees of visual angle [82, 83]. Each colored square ($1.17° \times 1.17°$) is randomly selected from a set of seven colors (red, blue, violet, green, yellow, black and white). A given color can appear no more than twice within an array. Stimulus positions were randomized on each trial, with the constraint that the distance between squares within a hemifield was at least $3.5°$ (center to center) [84].

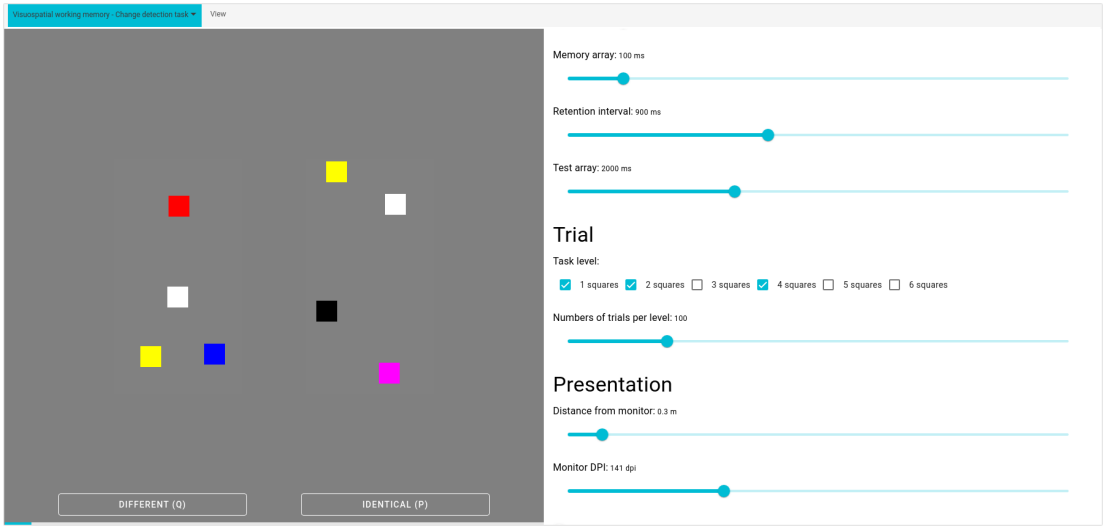


Figure **F-2**. VWM stimuli delivery interface.

F.3 Neurofeedback

Neurofeedback is attracting renewed interest as a method to self-regulate one's own brain activity to directly alter the underlying neural mechanisms of cognition and behavior. It not only promises new avenues as a method for cognitive

enhancement in healthy subjects, but also as a therapeutic tool. In the current article, we present a review tutorial discussing key aspects relevant to the development of EEG neurofeedback studies. In addition, the putative mechanisms underlying neurofeedback learning are considered. We highlight both aspects relevant for the practical application of neurofeedback as well as rather theoretical considerations related to the development of new generation protocols. Important characteristics regarding the set-up of a neurofeedback protocol are outlined in a step-by-step way. All these practical and theoretical considerations are illustrated based on a protocol and results of a frontal-midline theta up-regulation training for the improvement of executive functions. Not least, assessment criteria for the validation of neurofeedback studies as well as general guidelines for the evaluation of training efficacy are discussed [85].

Figure **F-3**. VWM neurofeedback dash board.

APPENDIX

G

PARADIGM: REWARD STOP SIGNAL TASK (RSST)

The concept of inhibitory control in human cognition can be approached from its basic motor and reflexive aspects to elaborate control processes such as planned actions and strategies [86], it can also be simply defined as the resistance to interference [87]. From a cognitive perspective, inhibitory control is not only a fundamental tool to guide behaviour towards goals accomplishment but to dynamically modify or cancel planned actions [88]. This dynamic dimension of (inhibitory) cognitive control is crucial to enable the flexibility of cognitive and behavioural control systems [89].

G.1 Paradigm

The general principle of Stop Tasks is a routine motor reaction where participants must hit a key each time they are confronted with a frequent go stimulus, and a cancellation of the ongoing action, after exposure to an infrequent stop signal. Our

visual stimuli and experimental design consist on a modified version of the SST developed by Rubia and colleagues (2003) [90], which is, in turn, a faster visual variant of the Tracking SST [91]. Main modifications reside on the introduction of monetary feedback after each successful inhibition and the suppression of punishment feedback after a failed inhibition [92].

Participants performed the Reward Stop Signal Task Paradigm (RSST) in two different groups. One group was aware of the possibility of rewards magnitudes shift but the order of rewards was not communicated (*expected specific rewards group*). In the other group (*unexpected reward group*), participants only knew that a monetary reward will appear without any mention to the reward shift and subsequently discovered (by themselves) a distinct reward magnitude only at the last block.

G.2 Stimuli presentation

The RSST was presented over 4 blocks of 4 min each. Each block has one of the three possible feedbacks: non-monetary reward (Smiley), low reward (50 COP) or high reward (500 COP). Regardless of the assigned condition or group, all participants performed exactly the same first – baseline- block, were each successful inhibition was rewarded with a Smiley. Afterwards, participants received two types of the mentioned monetary feedbacks.

To control for the effect of reward order presentation, we have built two conditions: for Increasing condition the order was Smiley, 50 COP, 50 COP, 500 COP; and for Decreasing condition, Smiley, 500 COP, 500 COP, 50 COP. Participants were randomly assigned to each condition in a counterbalanced way. Half of participants underwent the Increasing Condition and the other half, the Decreasing Condition.

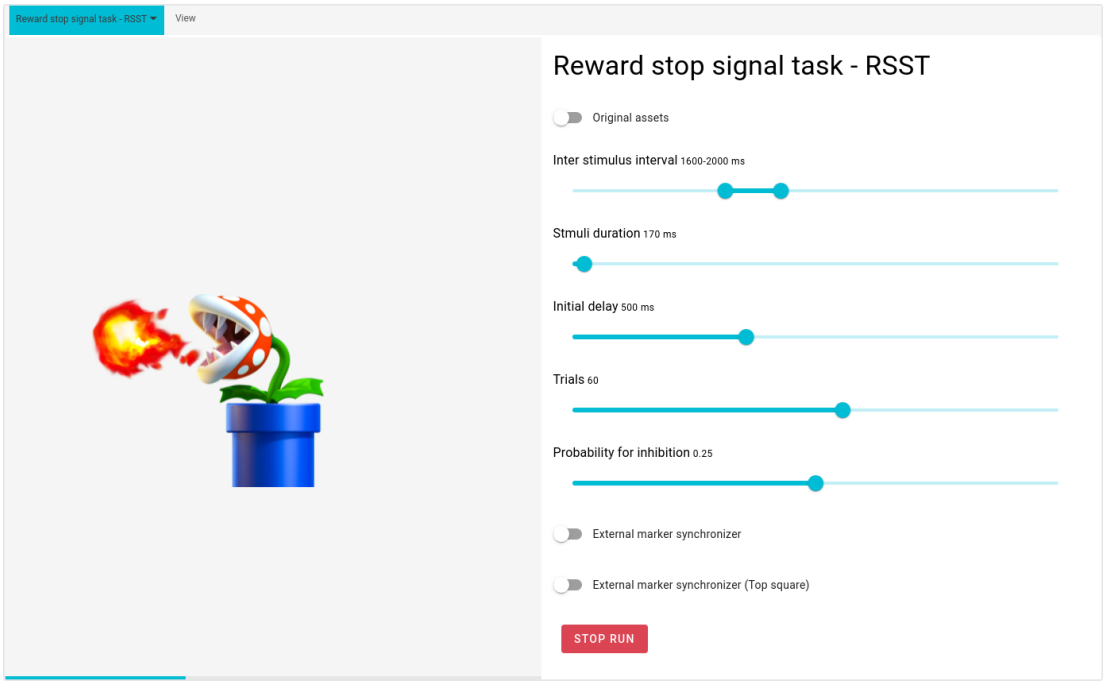


Figure **G-1**. Reward Stop Signal Task (RSST) stimuli delivery dashboard.

BIBLIOGRAPHY

[1] L. F. Nicolas-Alonso and J. Gomez-Gil, "Brain computer interfaces, a review," *sensors*, vol. 12, no. 2, pp. 1211–1279, 2012. (page 1)

[2] C. Tremmel, *Estimating Cognitive Workload in an Interactive Virtual Reality Environment Using Electrophysiological and Kinematic Activity*. PhD thesis, Old Dominion University, 2019. (page 1)

[3] M. Maleki, N. Manshouri, and T. Kayikcioglu, "Brain-computer interface systems for smart homes-a review study," *Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering)*, vol. 14, no. 2, pp. 144–155, 2021. (page 1)

[4] K. Polat, A. B. Aygun, and A. R. Kavsaoglu, "Eeg based brain-computer interface control applications: A comprehensive review," *Journal of Bionic Memory*, vol. 1, no. 1, pp. 20–33, 2021. (page 1)

[5] G. A. M. Vasiljevic and L. C. de Miranda, "Brain–computer interface games based on consumer-grade eeg devices: A systematic literature review," *International Journal of Human–Computer Interaction*, vol. 36, no. 2, pp. 105–142, 2020. (page 1)

[6] S. Taherian and T. C. Davies, "Caregiver and special education staff perspectives of a commercial brain-computer interface as access technology: a qualitative study," *Brain-Computer Interfaces*, vol. 5, no. 2-3, pp. 73–87, 2018. (page 1)

[7] S. K. Mudgal, S. K. Sharma, J. Chaturvedi, and A. Sharma, "Brain computer interface advancement in neurosciences: Applications and issues," *Interdisciplinary Neurosurgery*, vol. 20, p. 100694, 2020. (page 1)

[8] D. Bansal and R. Mahajan, *EEG-Based Brain-Computer Interfaces: Cognitive Analysis and Control Applications*. Academic Press, 2019. (page 1)

[9] S. Baillet, J. C. Mosher, and R. M. Leahy, "Electromagnetic brain mapping," *IEEE Signal processing magazine*, vol. 18, no. 6, pp. 14–30, 2001. (page 2)

[10] S. Laureys, M. Boly, and G. Tononi, "Functional neuroimaging in the neurology of consciousness: cognitive neuroscience and neuropathology," 2009. (page 2)

[11] L. F. Nicolas-Alonso and J. Gomez-Gil, "Brain Computer Interfaces, a Review," *Sensors*, vol. 12, pp. 1211–1279, Jan. 2012. (page 2)

[12] K. Kostiukevych, S. Stirenko, N. Gordienko, O. Rokovyi, O. Alienin, and Y. Gordienko, "Convolutional and recurrent neural networks for physical action forecasting by brain-computer interface," in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, pp. 973–978, IEEE, 2021. (page 2)

[13] J. Wolpaw and E. W. Wolpaw, "Brain-computer interfaces: Principles and practice," 2012. (pages 2, 4, and 10)

[14] D. Cárdenas-Peña, D. Collazos-Huertas, and G. Castellanos-Dominguez, "Enhanced data representation by kernel metric learning for dementia diagnosis," *Frontiers in neuroscience*, vol. 11, p. 413, 2017. (page 3)

[15] D. Collazos-Huertas, D. Cárdenas-Peña, and G. Castellanos-Dominguez, "Instance-based representation using multiple kernel learning for predicting conversion to alzheimer disease," *International journal of neural systems*, vol. 29, no. 02, p. 1850042, 2019. (page 3)

[16] J. D. Pulgarin-Giraldo, A. Ruales-Torres, A. M. Álvarez-Meza, and G. Castellanos-Dominguez, "Relevant kinematic feature selection to support human action recognition in mocap data," in *International Work-Conference on the Interplay Between Natural and Artificial Computation*, pp. 501–509, Springer, 2017. (page 3)

[17] J. V. Hurtado-Rincón, J. D. Martínez-Vargas, S. Rojas-Jaramillo, E. Giraldo, and G. Castellanos-Dominguez, "Identification of relevant inter-channel eeg connectivity patterns: a kernel-based supervised approach," in *International Conference on Brain Informatics*, pp. 14–23, Springer, 2016. (page 3)

[18] J. R. Wolpaw, N. Birbaumer, D. J. McFarland, G. Pfurtscheller, and T. M. Vaughan, "Brain–computer interfaces for communication and control," *Clinical neurophysiology*, vol. 113, no. 6, pp. 767–791, 2002. (page 4)

[19] C. S. Nam, A. Nijholt, and F. Lotte, *Brain–computer interfaces handbook: technological and theoretical advances*. CRC Press, 2018. (pages 4 and 5)

[20] J. R. Wessel, K. J. Gorgolewski, and P. Bellec, "Switching software in science: Motivations, challenges, and solutions," *Trends in cognitive sciences*, vol. 23, no. 4, pp. 265–267, 2019. (page 4)

[21] M. Ordikhani-Seyedlar and M. A. Lebedev, "Augmenting attention with brain–computer interfaces," in *Brain–Computer Interfaces Handbook*, pp. 549–560, CRC Press, 2018. (page 4)

[22] J.-A. Martinez-Leon, J.-M. Cano-Izquierdo, and J. Ibarrola, "Are low cost brain computer interface headsets ready for motor imagery applications?," *Expert Systems with Applications*, vol. 49, pp. 136–144, 2016. (page 4)

[23] J. LaRocco, M. D. Le, and D.-G. Paeng, "A systemic review of available low-cost eeg headsets used for drowsiness detection," *Frontiers in neuroinformatics*, vol. 14, 2020. (page 5)

[24] V. Peterson, C. Galván, H. Hernández, and R. Spies, "A feasibility study of a complete low-cost consumer-grade brain-computer interface system," *Heliyon*, vol. 6, no. 3, p. e03425, 2020. (pages 5 and 14)

[25] J. Frey, "Comparison of a consumer grade eeg amplifier with medical grade equipment in bci applications," in *International BCI meeting*, 2016. (page 5)

[26] P. Brunner and G. Schalk, "Bci software," in *Brain–Computer Interfaces Handbook*, pp. 323–340, CRC Press, 2018. (page 5)

[27] A. Lécuyer, F. Lotte, R. B. Reilly, R. Leeb, M. Hirose, and M. Slater, "Brain-computer interfaces, virtual reality, and videogames," *Computer*, vol. 41, no. 10, pp. 66–72, 2008. (page 5)

[28] M. Palaus, E. M. Marron, R. Viejo-Sobera, and D. Redolar-Ripoll, "Neural basis of video gaming: A systematic review," *Frontiers in human neuroscience*, p. 248, 2017. (page 5)

[29] M. Bassolino, M. Franza, J. Bello Ruiz, M. Pinardi, T. Schmidlin, M. Stephan, M. Solca, A. Serino, and O. Blanke, "Non-invasive brain stimulation of motor cortex induces embodiment when integrated with virtual reality feedback," *European Journal of Neuroscience*, vol. 47, no. 7, pp. 790–799, 2018. (page 5)

[30] I. Sugiarto and I. H. Putro, "Application of distributed system in neuroscience, a case study of bci framework," in *The 1st international seminar on science and technology*, 2009. (page 6)

[31] V. Alvarez and A. O. Rossetti, "Clinical use of eeg in the icu: technical setting," *Journal of clinical neurophysiology*, vol. 32, no. 6, pp. 481–485, 2015. (page 6)

[32] S. Beniczky, H. Aurlien, J. C. Brøgger, L. J. Hirsch, D. L. Schomer, E. Trinka, R. M. Pressler, R. Wennberg, G. H. Visser, M. Eisermann, *et al.*, "Standardized computer-based organized reporting of eeg: Score–second version," *Clinical Neurophysiology*, vol. 128, no. 11, pp. 2334–2346, 2017. (page 6)

[33] M. Assran, A. Aytekin, H. R. Feyzmahdavian, M. Johansson, and M. G. Rabbat, "Advances in asynchronous parallel and distributed optimization," *Proceedings of the IEEE*, vol. 108, no. 11, pp. 2013–2031, 2020. (page 6)

[34] S. Deshmukh, K. Thirupathi Rao, and M. Shabaz, "Collaborative learning based straggler prevention in large-scale distributed computing framework," *Security and communication networks*, vol. 2021, 2021. (page 6)

[35] A. Powell, "Democratizing production through open source knowledge: from open software to open hardware," *Media, Culture & Society*, vol. 34, no. 6, pp. 691–708, 2012. (page 9)

[36] H. Legenvre, P. Kauttu, M. Bos, and R. Khawand, "Is open hardware worthwhile? learning from thales' experience with risc-v," *Research-Technology Management*, vol. 63, no. 4, pp. 44–53, 2020. (page 9)

[37] F. Laport, F. J. Vazquez-Araujo, D. Iglesia, P. M. Castro, and A. Dapena, "A comparative study of low cost open source eeg devices," in *Multidisciplinary Digital Publishing Institute Proceedings*, vol. 21, p. 40, 2019. (page 10)

[38] R. Martínez-Cancino, A. Delorme, D. Truong, F. Artoni, K. Kreutz-Delgado, S. Sivagnanam, K. Yoshimoto, A. Majumdar, and S. Makeig, "The open eeglab portal interface: High-performance computing with eeglab," *NeuroImage*, vol. 224, p. 116778, 2021. (page 12)

[39] T. Choudhury, A. Tripathi, B. Arora, and A. Aggarwal, "Implementation of common spatial pattern algorithm using eeg in bcilab," in *International Conference on Recent Developments in Science, Engineering and Technology*, pp. 288–300, Springer, 2019. (page 12)

[40] A. Gramfort, M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, R. Goj, M. Jas, T. Brooks, L. Parkkonen, and M. S. Hämäläinen, "MEG and EEG data analysis with MNE-Python," *Frontiers in Neuroscience*, vol. 7, no. 267, pp. 1–13, 2013. (page 12)

[41] G. Schalk, D. J. McFarland, T. Hinterberger, N. Birbaumer, and J. R. Wolpaw, "Bci2000: a general-purpose brain-computer interface (bci) system," *IEEE Transactions on biomedical engineering*, vol. 51, no. 6, pp. 1034–1043, 2004. (pages 14 and 49)

[42] Y. Renard, F. Lotte, G. Gibert, M. Congedo, E. Maby, V. Delannoy, O. Bertrand, and A. Lécuyer, "Openvibe: An open-source software platform to design, test, and use brain–computer interfaces in real and virtual environments," *Presence: teleoperators and virtual environments*, vol. 19, no. 1, pp. 35–53, 2010. (page 14)

[43] M. Madrid Sobrino, "Brain computer interface," Master's thesis, 2014. (page 14)

[44] J. A. Wilson, J. Mellinger, G. Schalk, and J. Williams, "A procedure for measuring latencies in brain–computer interfaces," *IEEE transactions on biomedical engineering*, vol. 57, no. 7, pp. 1785–1797, 2010. (pages 14, 19, 30, and 49)

[45] S. Appelhoff and T. Stenner, "In com we trust: Feasibility of usb-based event marking," *Behavior Research Methods*, vol. 53, no. 6, pp. 2450–2455, 2021. (page 14)

[46] M. Razavi, V. Janfaza, T. Yamauchi, A. Leontyev, S. Longmire-Monford, and J. Orr, "Opensync: An open-source platform for synchronizing multiple measures in neuroscience experiments," *Journal of neuroscience methods*, vol. 369, p. 109458, 2022. (page 14)

[47] C. E. Davis, J. G. Martin, and S. J. Thorpe, "Stimulus onset hub: An open-source, low latency, and opto-isolated trigger box for neuroscientific research replicability and beyond," *Frontiers in Neuroinformatics*, vol. 14, 2020. (page 15)

[48] E. Netzer, A. Frid, and D. Feldman, "Real-time eeg classification via coresets for bci applications," *Engineering applications of artificial intelligence*, vol. 89, p. 103455, 2020. (page 15)

[49] M. A. Hasan, M. U. Khan, and D. Mishra, "A computationally efficient method for hybrid eeg-fnirs bci based on the pearson correlation," *BioMed Research International*, vol. 2020, 2020. (page 15)

[50] A. Ahmadi, O. Dehzangi, and R. Jafari, "Brain-computer interface signal processing algorithms: A computational cost vs. accuracy analysis for wearable computers," in *2012 Ninth International Conference on Wearable and Implantable Body Sensor Networks*, pp. 40–45, IEEE, 2012. (page 15)

[51] V. Changoluisa, P. Varona, and F. D. B. Rodríguez, "A low-cost computational method for characterizing event-related potentials for bci applications and beyond," *IEEE Access*, vol. 8, pp. 111089–111101, 2020. (page 15)

[52] T. Abe, I. Kinsella, S. Saxena, E. K. Buchanan, J. Couto, J. Briggs, S. L. Kitt, R. Glassman, J. Zhou, L. Paninski, *et al.*, "Neuroscience cloud analysis as a service," *bioRxiv*, pp. 2020–06, 2021. (page 15)

[53] S. M. Potter, A. El Hady, and E. E. Fetz, "Closed-loop neuroscience and neuroengineering," *Frontiers in neural circuits*, vol. 8, p. 115, 2014. (page 15)

[54] C. Muñiz, F. d. B. Rodríguez, and P. Varona, "Rtbiomanager: a software platform to expand the applications of real-time technology in neuroscience," *BMC Neuroscience*, vol. 10, no. 1, pp. 1–2, 2009. (page 15)

[55] R. Amaducci, M. Reyes-Sanchez, I. Elices, F. B. Rodriguez, and P. Varona, "Rthybrid: a standardized and open-source real-time software model library for experimental neuroscience," *Frontiers in Neuroinformatics*, vol. 13, p. 11, 2019. (page 15)

[56] "Welcome to pytables' documentation! — pytables 3.7.0 documentation." https://www.pytables.org/. (Accessed on 03/29/2022). (page 33)

[57] H. S. Kisakye, "Brain computer interfaces: Openvibe as a platform for a p300 speller," 2013. (page 49)

[58] R. K. Soni, *Full Stack AngularJS for Java Developers: Build a Full-Featured Web Application from Scratch Using AngularJS with Spring RESTful*. Apress, 2017. (page 55)

[59] E. Muller, J. A. Bednar, M. Diesmann, M.-O. Gewaltig, M. Hines, and A. P. Davison, "Python in neuroscience," *Frontiers in neuroinformatics*, vol. 9, p. 11, 2015. (page 60)

[60] F.-B. Vialatte, J. Solé-Casals, and A. Cichocki, "Eeg windowed statistical wavelet scoring for evaluation and discrimination of muscular artifacts," *Physiological Measurement*, vol. 29, no. 12, p. 1435, 2008. (page 69)

[61] G. Gómez-Herrero, W. De Clercq, H. Anwar, O. Kara, K. Egiazarian, S. Van Huffel, and W. Van Paesschen, "Automatic removal of ocular artifacts in the eeg without an eog reference channel," in *Proceedings of the 7th Nordic signal processing symposium-NORSIG 2006*, pp. 130–133, IEEE, 2006. (page 69)

[62] P. T. Wang, C. E. King, C. M. McCrimmon, J. J. Lin, M. Sazgar, F. P. Hsu, S. J. Shaw, D. E. Millet, L. A. Chui, C. Y. Liu, *et al.*, "Comparison of decoding resolution of standard and high-density electrocorticogram electrodes," *Journal of neural engineering*, vol. 13, no. 2, p. 026016, 2016. (page 90)

[63] L. Guo, "Principles of functional neural mapping using an intracortical ultra-density microelectrode array (ultra-density mea)," *Journal of Neural Engineering*, vol. 17, no. 3, p. 036018, 2020. (page 90)

[64] Q. Liu, M. Ganzetti, N. Wenderoth, and D. Mantini, "Detecting large-scale brain networks using eeg: impact of electrode density, head modeling and source localization," *Frontiers in neuroinformatics*, vol. 12, p. 4, 2018. (page 90)

[65] "daemon." `https://www.freedesktop.org/software/systemd/man/daemon.html`. (Accessed on 06/04/2022). (page 93)

[66] "systemd." https://www.freedesktop.org/software/systemd/man/systemd.html#. (Accessed on 06/04/2022). (page 93)

[67] C. Xu, C. Sun, G. Jiang, X. Chen, Q. He, and P. Xie, "Two-level multi-domain feature extraction on sparse representation for motor imagery classification," *Biomedical Signal Processing and Control*, vol. 62, p. 102160, 2020. (pages 117 and 118)

[68] D. G. García-Murillo, A. Alvarez-Meza, and G. Castellanos-Dominguez, "Single-trial kernel-based functional connectivity for enhanced feature extraction in motor-related tasks," *Sensors*, vol. 21, no. 8, p. 2750, 2021. (pages 117 and 118)

[69] M. Matsuo, N. Iso, K. Fujiwara, T. Moriuchi, D. Matsuda, W. Mitsunaga, A. Nakashima, and T. Higashi, "Comparison of cerebral activation between motor execution and motor imagery of self-feeding activity," *Neural regeneration research*, vol. 16, no. 4, p. 778, 2021. (page 117)

[70] D. F. Collazos-Huertas, A. M. Álvarez-Meza, C. D. Acosta-Medina, G. Castaño-Duque, and G. Castellanos-Domínguez, "Cnn-based framework using spatial dropping for enhanced interpretation of neural activity in motor imagery classification," *Brain Informatics*, vol. 7, no. 1, pp. 1–13, 2020. (page 118)

[71] S. Galindo-Noreña, D. Cárdenas-Peña, and Á. Orozco-Gutierrez, "Multiple kernel stein spatial patterns for the multiclass discrimination of motor imagery tasks," *Applied Sciences*, vol. 10, no. 23, p. 8628, 2020. (page 118)

[72] K. Choi, "Electroencephalography (eeg)-based neurofeedback training for brain–computer interface (bci)," *Experimental brain research*, vol. 231, no. 3, pp. 351–365, 2013. (page 118)

[73] C. Llanos, M. Rodriguez, C. Rodriguez-Sabate, I. Morales, and M. Sabate, "Mu-rhythm changes during the planning of motor and motor imagery actions," *Neuropsychologia*, vol. 51, no. 6, pp. 1019–1026, 2013. (page 118)

[74] S. Perdikis, R. Leeb, and J. d. R. Millán, "Subject-oriented training for motor imagery brain-computer interfaces," in *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 1259–1262, IEEE, 2014. (page 121)

[75] A. D. Baddeley, "Working memory: theories, models, and controversies," *Exploring Working Memory*, pp. 332–369, 2017. (page 123)

[76] Y. G. Pavlov and B. Kotchoubey, "Oscillatory brain activity and maintenance of verbal and visual working memory: A systematic review," *Psychophysiology*, vol. 59, no. 5, p. e13735, 2022. (page 123)

[77] E. L. Johnson, D. King-Stephens, P. B. Weber, K. D. Laxer, J. J. Lin, and R. T. Knight, "Spectral imprints of working memory for everyday associations in the frontoparietal network," *Frontiers in Systems Neuroscience*, vol. 12, p. 65, 2019. (page 123)

[78] D. Zhang, H. Zhao, W. Bai, and X. Tian, "Functional connectivity among multi-channel eegs when working memory load reaches the capacity," *Brain research*, vol. 1631, pp. 101–112, 2016. (page 123)

[79] Z. Dai, J. De Souza, J. Lim, P. M. Ho, Y. Chen, J. Li, N. Thakor, A. Bezerianos, and Y. Sun, "Eeg cortical connectivity analysis of working memory reveals topological reorganization in theta and alpha bands," *Frontiers in human neuroscience*, p. 237, 2017. (page 123)

[80] E. L. Johnson, J. N. Adams, A.-K. Solbakk, T. Endestad, P. G. Larsson, J. Ivanovic, T. R. Meling, J. J. Lin, and R. T. Knight, "Dynamic frontotemporal systems process space and time in working memory," *PLoS biology*, vol. 16, no. 3, p. e2004274, 2018. (page 123)

[81] E. K. Vogel and M. G. Machizawa, "Neural activity predicts individual differences in visual working memory capacity," *Nature*, vol. 428, no. 6984, pp. 748–751, 2004. (page 124)

[82] L. Newsome, "Visual angle and apparent size of objects in peripheral vision," *Perception & Psychophysics*, vol. 12, no. 3, pp. 300–304, 1972. (page 125)

[83] R. Haeuslschmid, S. Forster, K. Vierheilig, D. Buschek, and A. Butz, "Recognition of text and shapes on a large-sized head-up display," in *Proceedings of the 2017 Conference on Designing Interactive Systems*, pp. 821–831, 2017. (page 125)

[84] M. Villena-González, I. Rubio-Venegas, and V. López, "Data from brain activity during visual working memory replicates the correlation between contralateral delay activity and memory capacity," *Data in brief*, vol. 28, p. 105042, 2020. (page 125)

[85] S. Enriquez-Geppert, R. J. Huster, and C. S. Herrmann, "Eeg-neurofeedback as a tool to modulate cognition and behavior: a review tutorial," *Frontiers in human neuroscience*, vol. 11, p. 51, 2017. (page 126)

[86] A. R. Aron, P. C. Fletcher, E. T. Bullmore, B. J. Sahakian, and T. W. Robbins, "Stop-signal inhibition disrupted by damage to right inferior frontal gyrus in humans," *Nature neuroscience*, vol. 6, no. 2, pp. 115–116, 2003. (page 127)

[87] F. N. Dempster, "The rise and fall of the inhibitory mechanism: Toward a unified theory of cognitive development and aging," *Developmental review*, vol. 12, no. 1, pp. 45–75, 1992. (page 127)

[88] A. Bari and T. W. Robbins, "Inhibition and impulsivity: behavioral and neural basis of response control," *Progress in neurobiology*, vol. 108, pp. 44–79, 2013. (page 127)

[89] J. S. Ide, P. Shenoy, J. Y. Angela, and R. L. Chiang-Shan, "Bayesian prediction and evaluation in the anterior cingulate cortex," *Journal of Neuroscience*, vol. 33, no. 5, pp. 2039–2047, 2013. (page 127)

[90] K. Rubia, A. B. Smith, M. J. Brammer, and E. Taylor, "Right inferior prefrontal cortex mediates response inhibition while mesial prefrontal cortex is responsible for error detection," *Neuroimage*, vol. 20, no. 1, pp. 351–358, 2003. (page 128)

[91] G. D. Logan and W. B. Cowan, "On the ability to inhibit thought and action: A theory of an act of control.," *Psychological review*, vol. 91, no. 3, p. 295, 1984. (page 128)

[92] P. M. Herrera, A. V. Van Meerbeke, M. Speranza, C. L. Cabra, M. Bonilla, M. Canu, and T. A. Bekinschtein, "Expectation of reward differentially modulates executive inhibition," *BMC psychology*, vol. 7, no. 1, pp. 1–10, 2019. (page 128)