Brain-computer Interfacing practical course

- Optimized Calibration Procedure for Improved Classifier Training using Steady State Evoked Potentials -

Project Report

Group 5

Tommy Clausner
Robert Goß
Steven Smits
Kai Standvoß

Radboud University
Artificial Intelligence
Cognitive Neuroscience

All experiments were performed in Python 2.7 and Matlab R2016a. The game environment was programmed using Pygame and Psychopy. The BCI system was based on the Buffer BCI system which employs Fieldtrip.

Radboud University

Artificial Intelligence
Cognitive Neuroscience
Radboud University
http://www.ru.nl

Title:
Optimized Calibration Procedure for Improved Classifier Training using Steady State Visual Evoked Potentials

Theme:
Brain Computer Interfacing

Project Period:
First Semester 2017/2018

Project Group:
Group 5

Participant(s):
Tommy Clausner (s4836219)
Robert Goß (s1001785)
Steven Smits (s4232763)
Kai Standvoß (s4751744)

Supervisor(s):
Jason Farquhar
Karen Dijkstra

Copies: 1

Page Numbers: 52

Date of Completion:
February 4, 2018

Abstract:

Calibration of EEG-based brain computer interfaces is subject and application specific and therefore an integral but resource-intensive part of BCI projects. The main goal of the study was to create a universal calibration procedure for BCI systems that facilitates maximal transfer irrespective of the final application. To this end, a noisy yet meaningful background stimulus in the form of Conway's "Game of Life" was incorporated. Additionally, performance of colored (red/blue) as opposed to white SSVEP-stimuli were tested. The employed 2x2 design contrasts different calibration setups ("Game of Life" vs. neutral / colored vs white-stimuli). Two subjects trained 3 classifiers each for each condition. Application tests were performed in a "Space Invaders"-like game environment. We find that classifier perfomance is higher for white as opposed to colored SSVEP-stimuli. Due to limited amounts of data, the effect of the calibration procedure remains inconclusive.

Contents

1 Introduction

While until not long ago, controlling something in the outside world merely by the force of your will appeared as something straight out of a science-fiction novel, research in neuroscience is advancing at a pace that actually allows people to control external devices through nothing but signals from their brains and thereby perform tasks otherwise deemed impossible to them.

Generally, advances in cognitive neuroscience have led to an increasingly detailed understanding of the brain's signals and its way of processing, and communicating information. More specifically research on so-called *Brain Computer Interfaces* (BCIs) is aiming to exploit these new insights to devise means by which measured brain signals can be directly turned into commands to control an external device without the use of the peripheral nervous system [1]. Thereby users are not only enabled to control distal devices without occupying their hands, which might be desirable in industrial settings [2, 3]. More importantly, in the medical domain BCIs can return partial control to severely disabled patients like nearly locked-in ALS patients [4, 5].

One challenge to the extensive application of BCIs in real-world settings is finding suitable training and calibration procedures that allow for setting up the system in a way that is both interesting to the user as well as transferable to its actual application. BCIs draw heavily on recent advances in Machine Learning [6] for the classification of brain signals and thus good training data is crucial to the employability of the system. Specifically BCIs that make use of Electroencephalography (EEG) usually require subject specific training data since the obtained signals differ substantially between subjects [7]. Therefore, most BCI systems set up a calibration phase prior to the application of the system in which the user is provided with concrete instructions on how to control the BCI. Often the calibration procedures feature simplified versions of the actual task to be performed by the system. Since generalization and transferability are principal challenges to many machine learning algorithms [8] this classical approach to calibrating BCI systems might not be optimal. Not only might the actual problem be different in complexity, but more importantly the mental state of the user will presumably be quite distinct in a controlled setup environment as opposed to later use case scenarios. A real-world setting might involve noisier environments, different levels of arousal and higher expectations of the system's accuracy [9].

In the present paper, we aim to overcome these flaws by devising a calibration procedure more faithful to real-world application settings. We investigate our solution in the context of a "Space Invaders"-like game environment. The calibration phase was set to be different from the actual game since the specifications of the final applications were not in focus. Instead the goal was to find a calibration procedure that is as universally applicable as possible, independent of the subsequent

task. Hence we aimed to design a calibration phase that provides highly generalizable results in order to be most predictive for brain signals used in the later game. The actual challenge thereby is to create a noisy environment that still seems meaningful. This is due to the fact that the actual gaming procedure will display various visual patterns (e.g. space invaders) that are not shown during the training. This kind of "background noise" however is meaningful and hence we desired to create a calibration phase that displays something akin to random noise but in a meaningful manner. To achieve this we used an instance of the "Game of Life" [10]. Its pseudo random patterns enabled us to avoid the use of "real" meaningful stimuli like pictures or videos which might be too distracting or actually relevant in some applications. Further, the game of life does not feature familiar patterns but still appears to be meaningful in general. Utilizing this kind of noise as "background" for the calibration phase might create a highly transferable pre-calibration of the classifier without the need to train on the actual task itself.

The data was obtained using a ten electrode mobile EEG system. The stimulation took place in form of steady state visual evoked potentials (SSVEPs). SSVEPs are neural responses to visual stimulation at a specific frequency. The stimulation elicits a brain response at the same frequency, or harmonics thereof, in the visual cortex. SSVEPs were chosen in order to achieve a high signal to noise ratio (SNR) with as little training as possible [11]. Yet, this type of stimulation was chosen to be exemplary for a classical BCI setup and we assume that our findings translate to other types of EEG signals.

In order to compare different approaches for steady state stimulation we aimed for a 2 by 2 design. Presence of the "Game of Life" as a background stimulation vs. a plain black background and colored vs. white flickering stimuli. We tested colored stimuli specifically due to the high classification performance [12] demonstrated on red and blue. As reported in [13] 10 Hz stimulation has been shown to elicit strongest SSVEPs followed by the range 16-18 Hz. In order to match monitor requirements frequencies were set to 10 (blue) and 15 Hz (red) respectively.

Since the research question addresses how well calibration performances transfers to the actual application the task at hand is treated as a transfer learning problem [14]. In order to overcome intra-subject signal variabilities data from several recordings is pooled. The actual game performances are obtained during different sessions than the calibration.

We hypothesize that a colored SSVEP stimulation employing the game of life as "meaningful background noise" will enhance performance of later application to the actual game environment most.

2 Methods

2.1 Hardware specifications

Experimentation and data analysis was performed on a Laptop with Windows 10 Home, latest 7th Gen. Intel® Core™ i7 processor, latest GeForce® GTX 1060 3GB GDDR5 with desktop level performance 15.6" Full HD (1920x1080), IPS level panel.

A TMSi Mobita 10-channel water based EEG systems was used for data acquisition together with a 32-channel amplifier using a sampling rate of 250 Hz.

2.2 Software specifications

For this study, the buffer BCI framework was used to orchestrate all experiments [15]. This server is based on the FieldTrip buffer [16]. The framework allows for data acquisition, online analysis and stimuli representation in an integrated environment. Matlab2016A and Python 2.7 were used for their respective file types.

2.3 Subjects

Two healthy right-handed master students (males, age: 24, 31) with normal or corrected to normal vision served as volunteer subjects after giving informed consent.

2.4 Experimental design

A factorial 2x2 within-subject design was used, resulting in four experimental conditions. On a first level the “Game of Life” as background noise for the calibration phase was tested against a simple black background. On the second level colored flicker stimuli were contrasted to white stimuli.

Each subject performed the calibration procedure a minimum of three times in each condition during multiple sessions. All recordings for one condition were obtained during the same session. During the calibration the subject was asked to attend the stimulus indicated by an arrow at the bottom center of the screen. Stimuli were chosen at random and subjects were asked to switch attendance every 2.5 seconds. The calibration ended when the subject had attended at least 80 times to each of the two sides.

Subjects’ participations in the conditions was counterbalanced in order to avoid order effects. To that goal, the first subject did both GOL conditions first and the second subject last, while color vs. no-color conditions were interleaved.

In order to investigate the transferability of the results obtained during calibration, subjects were tested in the actual application of the BCI system — a “Space Invaders”-like game (c.f. Section 2.6). Since the research question was whether the improved calibration phase would enhance generalizability of the training data, testing in the game environment was performed during a different session than

calibration. For that, subjects played the game for 90 seconds twice for each condition. During game play, the BCI controller used a signal classifier trained only on the data of the respective condition. All data within one condition was pooled together. As an evaluation metric the achieved game score was recorded.

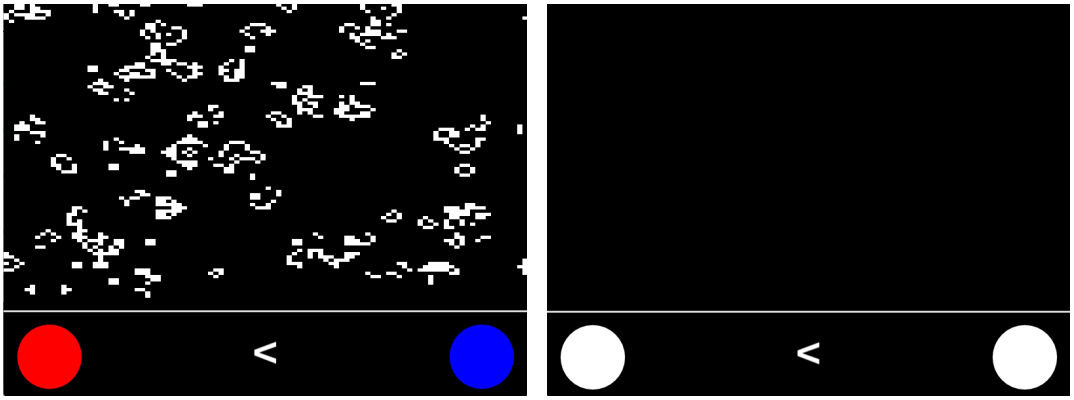
2.5 Stimuli

We used flickering circles with a diameter of 100 pixels. Depending on the condition the circles were either white or blue and red, with flicker frequencies of 10 and 15 Hz respectively. The steady state stimuli were expected to evoke steady state visual evoked potentials. In [13] 10 and 16 Hz are reported to give optimal responses. Here, frequencies were adjusted to the closest possible frequencies allowed by the monitor requirements. In all conditions 15 Hz stimulation was shown on the left side of the screen and 10 Hz on the right. The width of the calibration and game screen was set to 800 pixels resulting in the steady state stimuli being roughly 20cm apart such that peripheral interference of the non-attended stimuli was minimal.

Additionally, the noisy training conditions included an instance of Conway's "Game of Life" (GOL) [10]. The GOL defines a simple cellular automaton whose rules specify under which conditions a cell/pixel in the game is "alive". Therein, the state of a pixel is solely determined by the state of its neighbors. Different initial configurations of on- and off-pixels allow for complex periodically changing patterns. Here, the background of the calibration screen was instantiated as a GOL where each pixel corresponds to a cell in the game. Cells that were "alive" were colored white and "dead" pixels black. The game was set such that there were always active cells with different kinds of activation patterns, resulting in a primarily black background with continuous, "meaningful" patterns of activation. Example calibration screens for the different conditions can be seen in Figure 1. The game was updated at the screen refresh rate of 60Hz resulting in an almost continuous updating. Hence, we did not expect any interference with the steady state stimuli.

2.6 Game environment

The game used in the experiments is similar to the classic space invaders game. The player controls a "spaceship" on the bottom of the screen that moves on the x-axis and is able to shoot. In this particular case the shooting is automated to a rate of 5Hz. At fixed intervals of 3 seconds "aliens" spawn at pseudo random locations on the top of the screen and proceed to move downwards in a straight line while increasing in size. Further the aliens are accompanied by vertical lines extending to the edges of the screen that force the player to engage the lowest alien first. Moreover, green bonus-aliens that do not move can appear at random positions of



(a) Noise, color condition **(b)** No noise, no color condition

Figure 1: Calibration screen in different experimental conditions.

the screen with a probability of 5% every 8 seconds. At the left and right bottom of the screen steady-state stimuli in the shape of the aliens are displayed to enable the user to control the spaceship. Based on the attended stimulus, the classifier decodes the stimulation frequency from the measured EEG signals and translates it to a corresponding movement.

The game score is calculated by dividing the number of aliens hit by the total number of aliens that have appeared in the game so far. Therein the bonus-aliens are included, meaning that an accuracy of 100% is hardly achievable even at keyboard control.

An example screen of the game is shown in Figure 2. The background of the game is a static "star-field".

For the controls of the game a prediction was made every second by the classifier and sent to the game controller. In order to make the controls appear continuous the last command was repeated, causing a constant displacement of the spaceship by one pixel, until the next command was received.

In order to obtain a baseline accuracy, the average score of 10 games with random commands at the same frequency as during normal gameplay was calculated.

2.7 Classification

The signal to be classified were SSVEPs elicited by the attended steady state stimulus. Since the elicited visual potentials were expected to be strongest over the occipital lobe, a custom ten electrode EEG configuration was used that can seen in Figure 3. After standard preprocessing of the signal — detrending and rereferencing — spatial filtering via a ramped bandpass filter between 1 and 30 Hz was performed. Since the SSVEP is characterized by a clear increase in power in the frequency of stimulation the range of frequencies was deemed adequate for

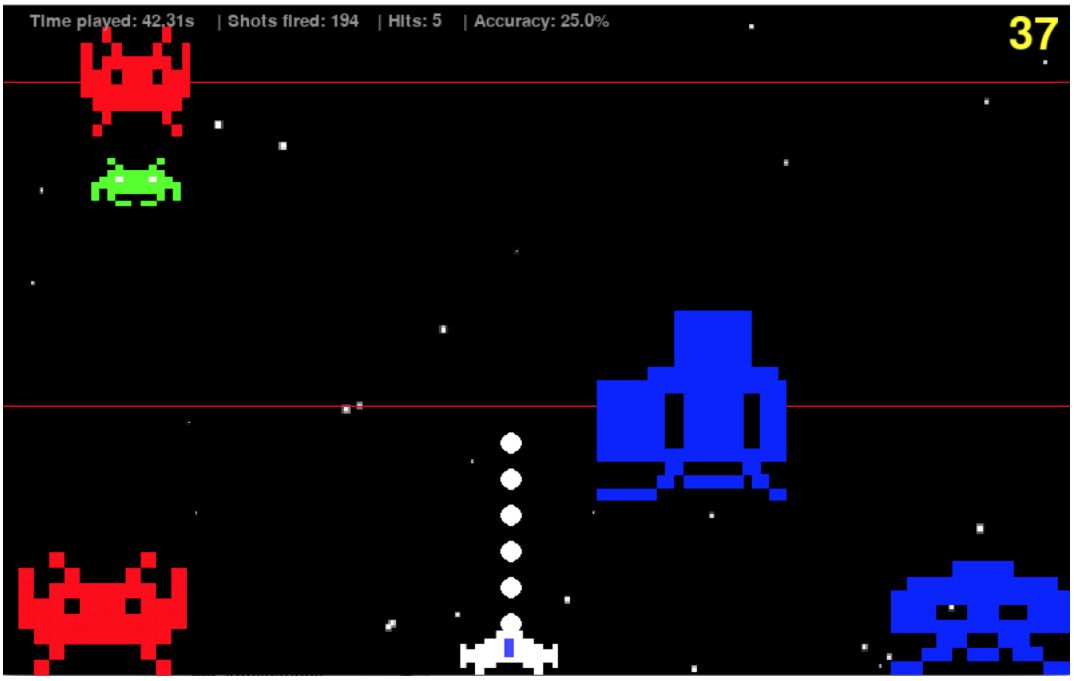


Figure 2: The application of the BCI system. The user controls a spaceship in a "Space-Invaders"-like game.

classification in the task at hand.

Subsequently the signal was fed into a simple linear classifier. Prior to the linear signal separation, a feature pre-selection in electrode and frequency space was performed based on the training data. In order to transform the data from the time to frequency domain Welch's fast Fourier transform [17] was used.

3 Results

3.1 Calibration

In order to obtain a measure of performance for the calibration phase that can be used to quantify the transferability of the results to the game application, an individual classifier was trained on all data per condition separately. The results per condition and for both subjects separately can be seen in Table 1 — **R** and **S** refer to the two subjects.

It can be seen that both subjects obtained high classification accuracies of over 70%, endorsing the good signal-to-noise ratio of SSVEPs. It can be further seen that both subjects performed better in the non-color conditions than in the color conditions. No clear trend is visible between noisy and normal training. Figure 4 visualizes the interaction between color and GOL conditions for the two subjects.

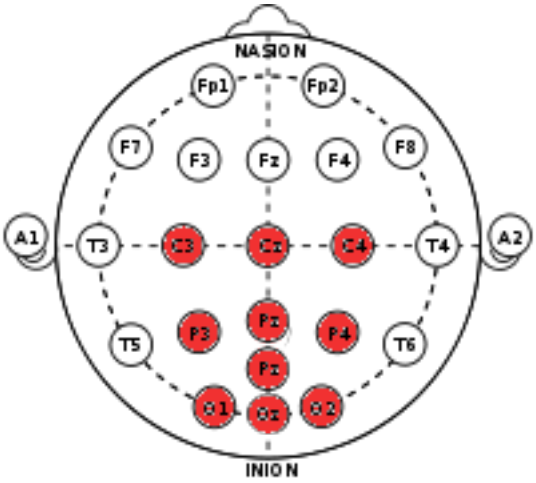


Figure 3: Custom EEG placement

No clear interaction between color and background can be seen. A Friedman's χ^2-test for repeated measures, revealed no significant interactions between the conditions (χ^2: 1.8, $p > 0.6$). However due to the small sample size the test statistic is not very informative.

Figure 5 shows sample power spectra for two electrodes of one subject in the two calibration tasks — left, 15 Hz attended vs. right, 10 Hz attended. A difference in the power spectra between the two tasks is clearly visible as a relative increase in power for the stimulation frequency as opposed to the non attended frequency. Moreover, an increase in the 20 Hz amplitude can be seen for the right attended task. This is to be expected as 20 Hz is a harmonic of the steady state frequency. The fact that no corresponding increase in 30 Hz power is visible for the left attended condition, is explained by the bandpass cutoff at 30 Hz.

In Figure 6 the frequencies selected by the classifier for "right attended" are visualized. It can be seen that the classifier indeed primarily selects frequencies around 15 Hz for most electrodes. Additionally 20 Hz is used to a lesser extend for separation of the classes.

Conditions	Color(Red/Blue)	Non-Color (White)
Noisy Training (Game of Life)	78.2(**R**), 82.2(**S**)	81.6(**R**), 88.8(**S**)
Normal (Blank)	72.5(**R**), 87.9(**S**)	87.1(**R**), 81.8(**S**)

Table 1: Classifier Accuracy per Condition

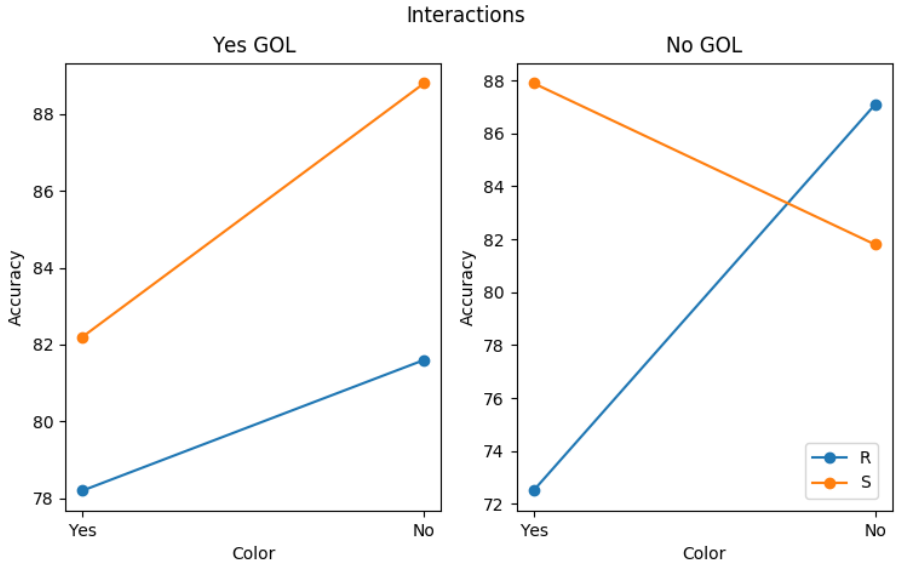


Figure 4: Interaction between game of life and color conditions for both subjects. Data points correspond to mean accuracies per subject and condition.

3.2 Testing

Accuracies from the calibration phase and the game score are not directly comparable since the former directly reflects classification accuracy while the latter is calculated based on the game performance. Nevertheless, the relative game performances give an indication on how well the learned decision boundary is transferable to the test case.

The obtained game scores can be seen in Table 2. The game accuracies are expected to be lower than the classification accuracies during calibration. The mean score for both subjects together is 59.2 (std: 7.5) and individual means are 63.75 (std: 6.5) and 54.7 (std: 5.3) for the subjects **R** and **S** respectively. The simulated

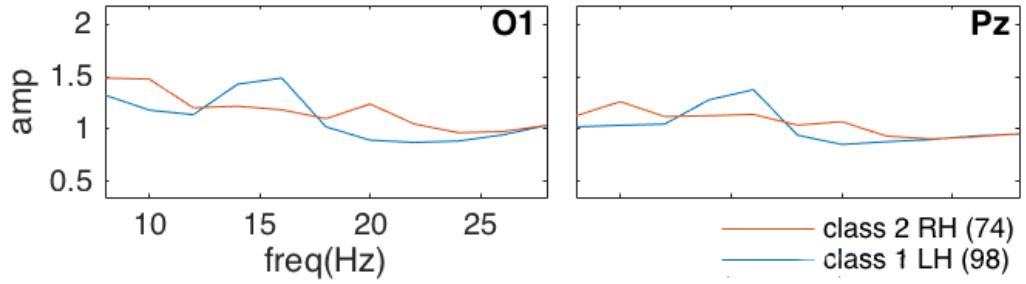


Figure 5: Signal amplitude in frequency space per condition — left vs. right attended — for electrodes **O1** and **Pz**. Differences are most visible at 10, 15, and 20 Hz.

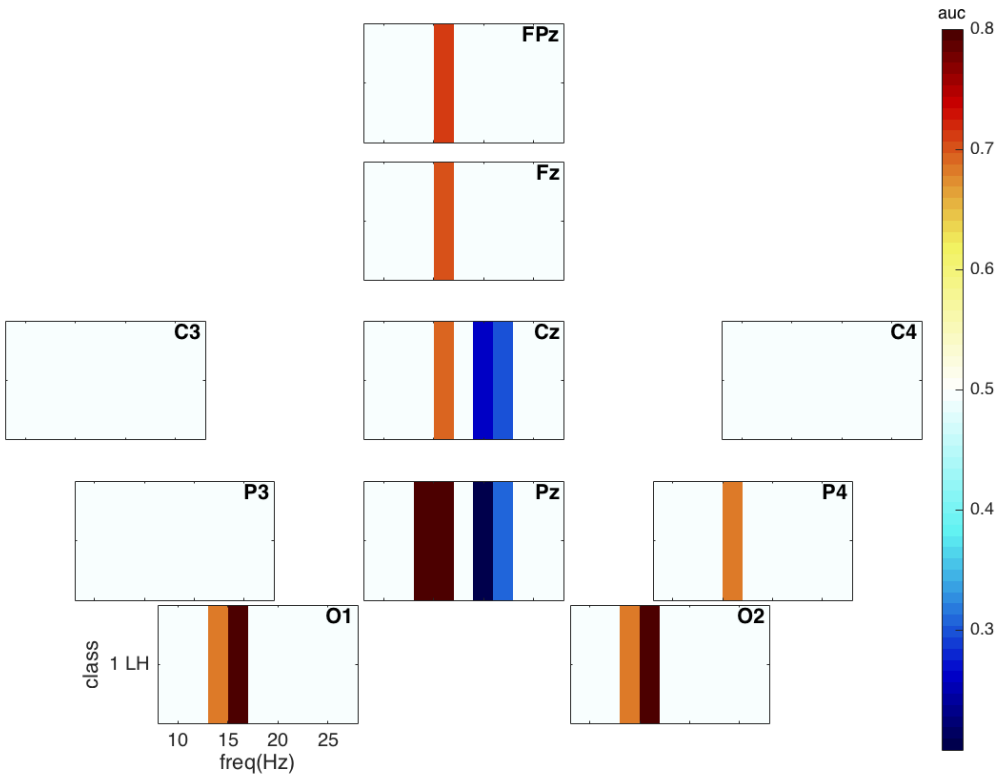


Figure 6: Selected features in electrode and frequency space. The classifier primarily uses 15 and 20 Hz to discriminate the conditions.

baseline score is 46.07 with a standard error of 6.3. Only subject **R** performed significantly above chance (t: 2.41, df: 8.24, $p < 0.05$).

No clear difference between the conditions is visible.

Figure 7 shows the transfer between calibration and game. While the absolute scores are not entirely comparable, the relative decrease is still indicative of the amount of transfer between the different sessions and environments. The figure shows more clearly, that transfer barely happened for subject **S** with most game accuracies close to chance. For subject **R** on the other hand control in the game was possible using the calibration data from the other sessions in most conditions. Yet very little difference in loss in accuracy is visible across conditions.

Conditions	Color(Red/Blue)	Non-Color (White)
Noisy Training (Game of Life)	72.5, 50 (**R**); 57.5, 42.5 (**S**)	57.5, 60 (**R**); 47.5, 57.5 (**S**)
Normal (Blank)	67.5, 52.5 (**R**); 57.5, 70 (**S**)	80, 70(**R**); 60, 45 (**S**)

Table 2: Game Score per Condition

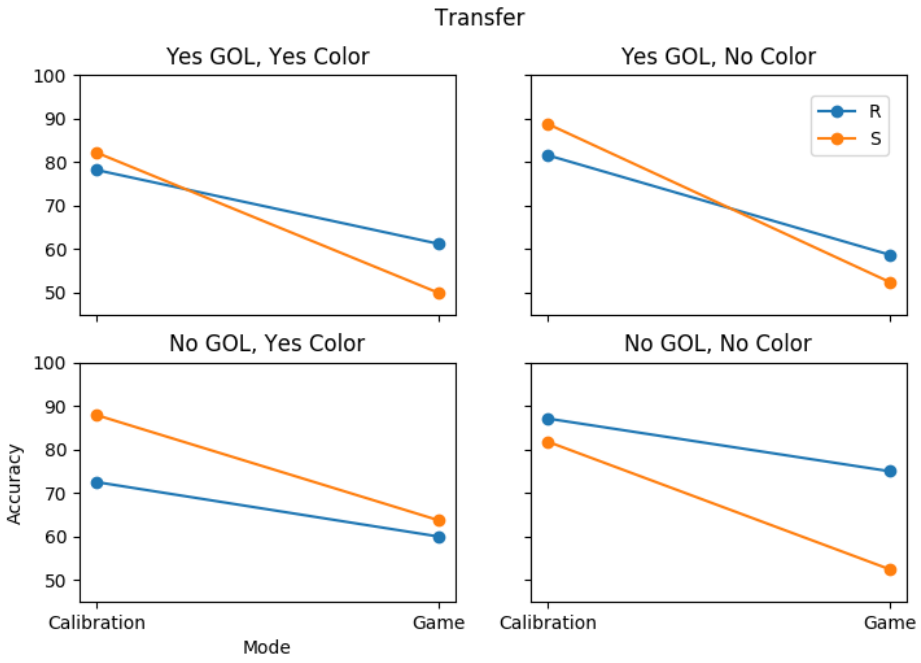


Figure 7: Transfer between calibration and game for the different conditions.

4 Discussion

This work aimed to devise an improved, universal calibration procedure for EEG-based BCI systems that enables maximal transfer to the final application of the system. To that end, a calibration setup was introduced that uses noisy, yet meaningful background stimulation in the form of Conway's "Game of Life". Furthermore, it was investigated whether colored stimuli are more suitable to elicit SSVEPs as opposed to white steady-state stimuli. It was hypothesized that the new calibration procedure using colored stimuli would lead to optimal transfer between calibration and application.

In contrast to our expectations, colored steady state stimulation led to lower classification performances during calibration than white stimuli. One possible explanation for this finding is that white light activates all three cone types in the retina, as opposed to monochromatic light, that only activates a single type as stated in [18]. Furthermore, luminance of the stimuli was not controlled for. Hence, white stimulation appeared to be brighter than colored stimuli causing a lower perceived contrast to the black background and consequently weaker SSVEPs [19].

No significant difference between the newly introduced calibration phase and the control condition was found in the classification scores. This was to be expected, as the hoped-for advantage of the noisy training was the transferability to

the later application of the BCI. In fact, slightly worse performance during calibration owing to the noisier environment is in line with our initial hypothesis as long as the relative decrease in the later game performance would be smaller than for the control condition.

Unfortunately, this was not the case as no considerable difference between the calibration conditions and the later game score was found. While transfer learning between sessions worked for one of the participants, no advantage of either modified background nor color of steady state stimuli could be seen. One reason for this is evidently the lack of a sufficiently large sample size hindering the generalizability of our findings to a larger population [20, 21]. With only one subject that showed any form of transfer between calibration and application no meaningful statistics for the effects of the different training conditions could be obtained. Another possible explanation for the obtained null-findings is that the type of noise induced by the GOL is in fact not universal enough to translate to other applications [22]. Under the hypothesis that some form of noise during the calibration procedure is beneficial for transfer learning, future research could reveal whether more application specific noise is better suited to achieve that goal, by contrasting the GOL condition to another calibration procedure in which the actual game environment is used for training. Thereby the universality of the calibration would be lost, but possibly the vast variation between different kinds of noise is not easily captured by a simple, unique kind of stimulation.

All in all, we showed that SSVEPs provide a strong signal for binary control in a general BCI setting. Further, we found that white steady state stimulation leads to higher classification accuracies than colored stimuli, not exploiting findings from previous BCI literature. We were not able to confirm our hypothesis that a noisy calibration procedure would benefit transfer between training and application. Future research should reveal whether a calibration more closely tailored for a specific application can improve results.

References

[1] Marcel Van Gerven, Jason Farquhar, Rebecca Schaefer, Rutger Vlek, Jeroen Geuze, Anton Nijholt, Nick Ramsey, Pim Haselager, Louis Vuurpijl, Stan Gielen, and others. The brain–computer interface cycle. *Journal of neural engineering*, 6(4):041001, 2009.

[2] Jan van Erp, Fabien Lotte, and Michael Tangermann. Brain-Computer Interfaces: Beyond Medical Applications. *Computer*, 45(4):26–34, April 2012.

[3] Gianluca Borghini, Laura Astolfi, Giovanni Vecchiato, Donatella Mattia, and Fabio Babiloni. Measuring neurophysiological signals in aircraft pilots and

car drivers for the assessment of mental workload, fatigue and drowsiness. *Neuroscience & Biobehavioral Reviews*, 44:58–75, July 2014.

[4] Femke Nijboer, EW Sellers, Jürgen Mellinger, Mary Ann Jordan, Tamara Matuz, Adrian Furdea, Sebastian Halder, Ursula Mochty, DJ Krusienski, TM Vaughan, and others. A P300-based brain–computer interface for people with amyotrophic lateral sclerosis. *Clinical neurophysiology*, 119(8):1909–1916, 2008.

[5] Ujwal Chaudhary, Bin Xia, Stefano Silvoni, Leonardo G Cohen, and Niels Birbaumer. Brain–computer interface–based communication in the completely locked-in state. *PLoS biology*, 15(1):e1002593, 2017.

[6] Ewan S. Nurse, Philippa J. Karoly, David B. Grayden, and Dean R. Freestone. A Generalizable Brain-Computer Interface (BCI) Using Machine Learning for Feature Discovery. *PLOS ONE*, 10(6):e0131328, June 2015.

[7] Dongrui Wu, Vernon J Lawhern, and Brent J Lance. Reducing offline bci calibration effort using weighted adaptation regularization with source domain selection. In *Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on*, pages 3209–3216. IEEE, 2015.

[8] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, pages 3320–3328, 2014.

[9] Chang S. Nam, Steve Johnson, and Yueqing Li. Environmental Noise and P300-Based Brain-Computer Interface (BCI). *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 52(12):803–807, September 2008.

[10] John Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[11] Ming Cheng, Xiaorong Gao, Shangkai Gao, and Dingfeng Xu. Design and implementation of a brain-computer interface with high transfer rates. *IEEE Transactions on Biomedical Engineering*, 49(10):1181–1186, October 2002.

[12] Lingling Yang and Howard Leung. An online BCI game based on the decoding of users' attention to color stimulus. pages 5267–5270. IEEE, July 2013.

[13] Danhua Zhu, Jordi Bieger, Gary Garcia Molina, and Ronald M. Aarts. A Survey of Stimulation Methods Used in SSVEP-Based BCIs. *Computational Intelligence and Neuroscience*, 2010:1–12, 2010.

[14] Vinay Jayaram, Morteza Alamgir, Yasemin Altun, Bernhard Scholkopf, and Moritz Grosse-Wentrup. Transfer learning in brain-computer interfaces. *IEEE Computational Intelligence Magazine*, 11(1):20–31, 2016.

[15] J Farquhar. Buffer bci.

[16] Robert Oostenveld, Pascal Fries, Eric Maris, and Jan-Mathijs Schoffelen. Fieldtrip: open source software for advanced analysis of meg, eeg, and invasive electrophysiological data. *Computational intelligence and neuroscience*, 2011:1, 2011.

[17] Peter Welch. The use of fast fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. *IEEE Transactions on audio and electroacoustics*, 15(2):70–73, 1967.

[18] Teng Cao, Feng Wan, Peng Un Mak, Pui-In Mak, Mang I Vai, and Yong Hu. Flashing color on the performance of ssvep-based brain-computer interfaces. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*, pages 1819–1822. IEEE, 2012.

[19] Wenqiang Yan, Guanghua Xu, Jun Xie, Min Li, Sicong Zhang, and Ailing Luo. Study on the effects of brightness contrast on steady-state motion visual evoked potential. In *Engineering in Medicine and Biology Society (EMBC), 2017 39th Annual International Conference of the IEEE*, pages 2263–2266. IEEE, 2017.

[20] Jacob Cohen. Things i have learned (so far). *American psychologist*, 45(12):1304, 1990.

[21] Amitav Banerjee and Suprakash Chaudhury. Statistics without tears: Populations and samples. *Industrial psychiatry journal*, 19(1):60, 2010.

[22] Hohyun Cho, Minkyu Ahn, Kiwoong Kim, and Sung Chan Jun. Increasing session-to-session transfer in a brain–computer interface with on-site background noise acquisition. *Journal of neural engineering*, 12(6):066009, 2015.

A User Manual

In order to make full use of the supplemented code, the following user guide will serve as a walk through all capabilities and the required structure.

The full code can be downloaded from https://github.com/kstandvoss/BCI17

If everything is set up correctly all subroutines will run in background. The user is able to make all necessary changes to the environment in the main menu of the provided graphical user interface (GUI). Note, that it is highly recommended to use the included reduced version of the *buffer_bci-master*. This version was reduced to essential parts and further *eeg_quickstart.bat*, *debug_quickstart.bat*, *eeg_quickstart.sh* and *debug_quickstart.sh* were modified in order to correctly output process IDs, which are stored in */external/pids.txt* and that are used to correctly kill all processes run in background.

A.1 Pre- requisites

All functions are executed either using Python2.7 or MATLAB. Both, Python and MATLAB root path must be set in the *config.txt* in the master folder.

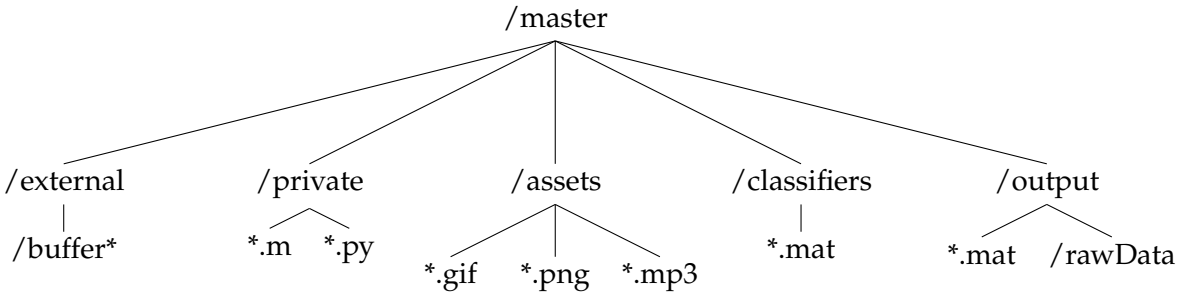
Example for Mac:

```
MATLABroot = MATLAB_RXXXXx.app/bin/matlab -nosplash -nodesktop -r
Pythonroot = /Users/user/anaconda/bin/python
```

Example for Windows:

```
MATLABroot = matlab -nosplash -nodesktop -r
Pythonroot = cmd /c  py -2.7
```

Folder structure:



All provided buffer BCI functions that are necessary to run the buffer system are located in */external*. This version was reduced to core features and wrapper

function such as *eeg_quickstart.bat* were modified in order to correctly interact with the GUI system. Functions used for presenting stimuli or data analysis are located in the folder */private*. Note that all functions related to presenting visual content are written in Python2.7; Whereas all functions related to data analyses are written for MATLAB. Additional (mostly) graphical assets are located in */assets*. Trained classifiers are stored in */classifiers* in MATLAB file format. Raw data as well as MATLAB compatible raw data is stored in */output*.
Furthermore the master folder contains several files:

- *START_for_windows.bat* to launch the GUI on a windows system
- *START_for_unix.sh* to launch the GUI on a unix-oid system
- *brainflyGUI.py*, the actual graphical user interface
- *config.txt* text file to predefine MATLAB and Python root paths
- *highscore.txt* in case you're into that

Basic Python packages are assumed to already be pre-installed. Additional Python packages that are required are *matplotlib* (https://matplotlib.org/) and *numpy* (http://www.numpy.org/). Further make sure that *PsychoPy* and *PyGame* is installed and accessible. Information about how to install PsychoPy and PyGame can be found here:
http://psychopy.org/installation.html
https://www.pygame.org/wiki/GettingStarted

A.2 Quickstart

If the above mentioned pre- requisites are met the GUI can be started by running *sh START_for_unix.sh* from the command line (Unix case) or by double clicking *START_for_windows.bat*

A.3 GUI overview

The GUI uses all buffer related functions located in */external*. Furthermore it calls scripts from */private*. Those scripts are used for stimulation (Python scripts) and data analysis (MATLAB scripts). Note, that a headless instance of MATLAB will be run in a background command window.
Once the GUI started the user has the following choices:

- use $\uparrow$ and $\downarrow$ to navigate in the main menu options
- use **e** to switch between active EEG and debug mode. In EEG mode an actual connected EEG system that outputs its data to the buffer, will be recognized

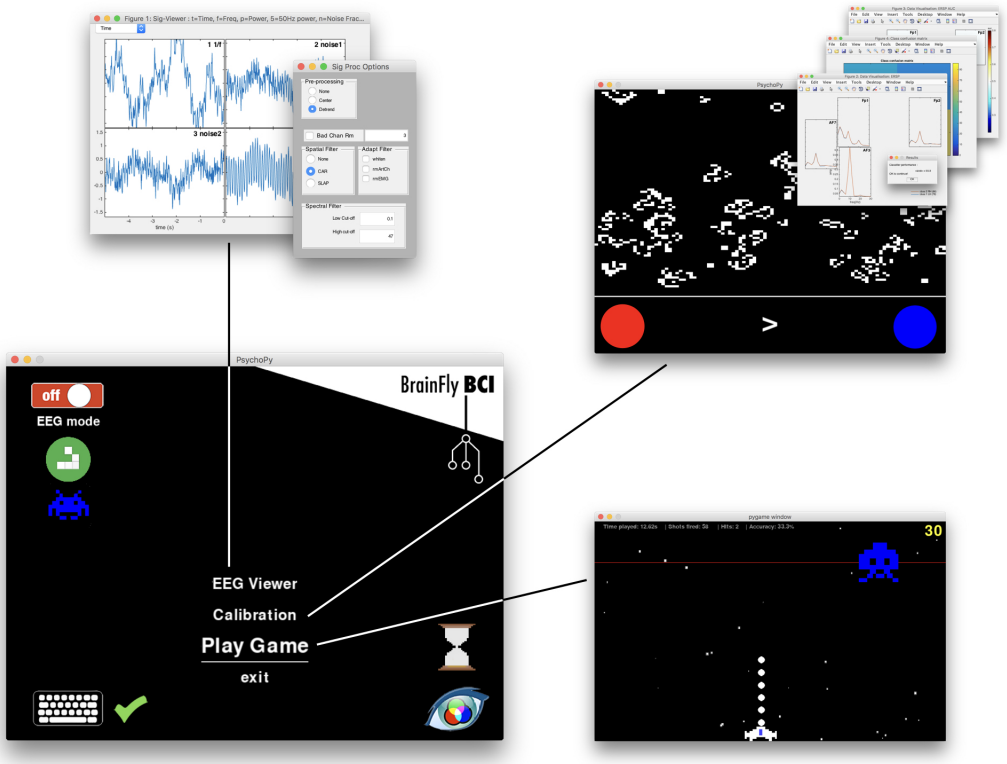


Figure 8: The Main window of the GUI connects all bits of the BrainFly BCI system. Each of the main menu options will be responsible for a different stage in using the BCI. EEG Viewer: allows for signal inspection of the EEG. Calibration: starts the calibration phase, which is used to collect data for training the classifier. Play Game: Navigate your space ship through space and kill all enemies either with your brain or keyboard control.

and used for data input. In case of selecting debug mode, random data will be generated.

- use **u** to enable (green) or disable (red) game of life during calibration. If game of life stimulation is enabled a random instance of the game of life will be rendered online in the background. After each frame update, the current state will be displayed in the background. If game of life stimulation is disabled, the respective background remains black.

- use **c** to switch between white or colored SSVEP stimulation. Colored SSVEPs will be red (left) and blue (right). If white stimulation was chosen, both sides will flicker in white color. Default steady state frequencies are 15Hz (left stimulus) and 10 Hz (right stimulus).

- use **k** to switch between EEG control or keyboard control. If keyboard control

is enabled, the user is able to control the space ship by using ← and →. When keyboard control is disabled, a MATLAB instance will be run in the background to classify input data according to a preselected classifier and EEG cap layout. Input data can be random or actual data. Classifications will be made binary (-1, 1) and outputted to the buffer, from where it is read from the game instance.

- use **t** to enable time limit for testing phase (filled hourglass). This option limits the game time to 90s.
- use **g** to enable or disable eye-candy. When eye-candy is enabled the user can dive into the full BrainFly BCI experience. Only recommended if you're hard enough!
- use **m** to switch music on / off. Within the main menu a background music provides just the right ambient for BCI. During EEG Viewer mode the music can stay enabled. However when selecting calibration or game mode, the main menu background music will be muted. If desired it can be switched back on by pressing m

Note, that depending on which processes are run in the background a file called */external/pids.txt* is created, storing the process IDs of currently running processes. This enables the GUI (but also the user if desired) to kill background processes that were launched during operation.

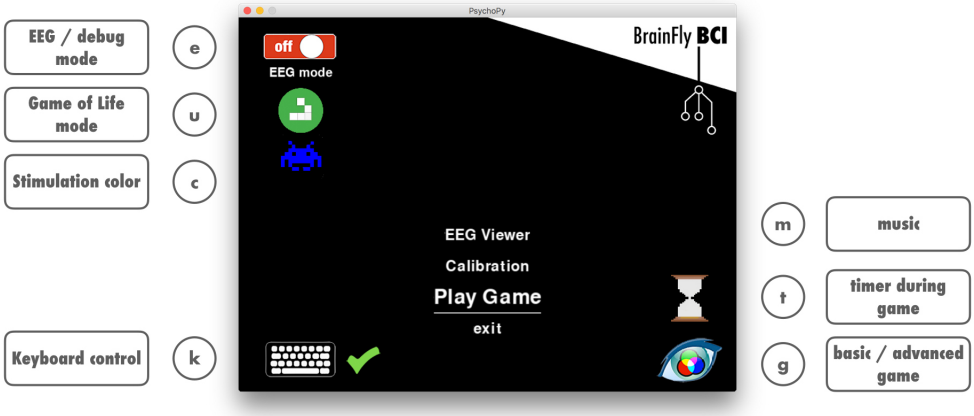


Figure 9: Options to chose for the BrainFly GUI. Each point is represented by a pictogram that reflects the current state of the respective option. Circled characters indicate the respective key stroke necessary to change the respective setting. Note that each key reflects a binary decision whether or not to activate a given setting.

A.3.1 EEG Viewer

The EEG viewer launches a MATLAB session in the background. Soon a file selection dialog will prompt, asking the user to select a EEG cap file. After selection the buffer's inherent EEG viewer will be displayed including it's full functionality. Raw signal as well as time-frequency resolved inspections can be made. Furthermore the subject can be prepared by plotting the current resistance for each electrode. The respective view mode can be changed by selecting the desired mode from the drop-down menu in the top left corner.

A.3.2 Calibration

Calibration will run according to the pre-defined settings in the main menu. If e.g. Game of Life stimulation and white SSVEP stimulation was chosen, the calibration phase will run accordingly.
The subject will be instructed (using white index arrows) to focus on the left or right flickering stimulus. Thereby eventual background stimulation has to be suppressed (i.e. ignored) as good as possible. After sampling an equal amount of trials, the calibration phase will end and the data is handed over to an ERSP classifier. Afterwards four results windows will appear, depicting classifier accuracy, feature selection per electrode, average class signal per electrode and a confusion matrix of the classifier performance. The respective classification results are stored in */classifiers*. Raw buffer data is outputted to */output* after closing the GUI window.

A.3.3 Play Game

Starts actual implementation of BrainFly according to the settings in the main menu. A dialog prompt will ask the user to input a certain classifier file, if EEG mode is used. Again, the subject has to focus on the respective flickering stimulus depending on which side the virtual space ship is requested to go. While focusing on a stimulus as control input, background stimulation must be suppressed as good as possible.

B Source code

The code below contains core part of the present project, excluding provided functions. All function below can be found in */master* or */private*. Explanations to any sub-function can be found within the comments in the respective function.

B.1 Main GUI window (brainflyGUI.py)

```
"""
  ____                _       ______ _          ____    _____ _____
 |  _ \              (_)     |  ____| |        |  _ \ / ____|_   _|
 | |_) |_ __ __ _ _ _ _ __ | |__  | |_   _  | |_) | |      | |
 |  _ <| '__/ _` | | | '_ \|  __| | | | | | |  _ <| |      | |
 | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
 |____/|_|  \__,_|_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
                                       __/ |
                                      |___/
"""

# This is the actual main window from where everything is administered
# In the appendix of the report a users manual is provided.

# import all necessary libraries
import sys,os,time
import subprocess
import signal
import pygame
from psychopy import visual, core,event
import platform
from PIL import Image
import gc
import pdb

# initializes PyGame window
def initModules():
    # presetting for some modules
    gc.collect()
    pygame.init()
    pygame.display.init()

#pdb.set_trace()
initModules()

# sets up all path dependencies needed for the GUI to run
def initPaths():

    # own path from where the script was run
    main_path=os.path.dirname(os.path.abspath(__file__))+os.sep

    # setup relative sub-directories for each component
    BCI_buff_path="external"+os.sep
    bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
    sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
    assetpath="assets"+os.sep+"GUI_grafix"
    functionspath="private"+os.sep

    # add respective paths
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+bufferpath))
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+sigProcPath))
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+assetpath))
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+functionspath))
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+BCI_buff_path))

    # get MATLAB and Python root paths from the config.txt file (Additionally the initial configuration for
        debug mode is read)
    pathsettings=open(main_path+"config.txt","r").read().split("\n")

    idx_MATLAB=int([i for i, e in enumerate([str.find(path_,"MATLABroot") for path_ in pathsettings]) if e ==
        0][0])
    idx_Python=int([i for i, e in enumerate([str.find(path_,"Pythonroot") for path_ in pathsettings]) if e ==
        0][0])
```

```
    idx_debug = int([i for i, e in enumerate([str.find(path_, "debug") for path_ in pathsettings]) if e ==
        0][0])

    # setup roots
    MATLABrootpath=pathsettings[idx_MATLAB][str.find(pathsettings[idx_MATLAB],'=')+1:]
    Pythonrootpath=pathsettings[idx_Python][str.find(pathsettings[idx_Python],'=')+1:]

    # define initial debug mode
    initdebug=bool(int((pathsettings[idx_debug][str.find(pathsettings[idx_debug], '=') + 1:]).replace('␣', ''))
        )
    print(initdebug)
    return MATLABrootpath,Pythonrootpath,main_path,BCI_buff_path,bufferpath,sigProcPath,assetpath,functionspath
        ,initdebug

# calling the function below sets up all folder dependencies necessary for the GUI to run
MATLABrootpath,Pythonrootpath,main_path,BCI_buff_path,bufferpath,sigProcPath,assetpath,functionspath,initdebug=
    initPaths()

# setting OS specific driver information. Due to various problems with display drivers and the resulting output
    ,
# it is highly recommended to keep the settings as they are
def setupOS():
    # setup video drivers
    if platform.system() == 'Windows':
        os.environ['SDL_VIDEODRIVER'] = 'windib'#'directx'
        # sleep timer is used to limit the number of button press responses to certain maximum (e.g. min 0.09 ~
            11Hz)
        sleep_timer=0.09
    else:
        sleep_timer=0.09
        os.environ['SDL_VIDEODRIVER'] = 'quartz'
    return sleep_timer

# This function is used to kill background processes that are started by the GUI. Within "pids.txt" process IDs
    are stored,
# as processes are created. Those IDs will be read and killed. Since this works different on Unix and Windows,
    the
# respective implementation varies slightly
def killbuff(BCI_buff_path_int=BCI_buff_path):

    if platform.system() == 'Windows':
        try:
            PID=open(main_path+BCI_buff_path_int+"pids.txt")
            PID=PID.read()[:-1]
            os.system("start␣cmd␣/c␣taskkill␣/PID␣"+PID)
            print('killed␣those␣processes:␣'+PID)
        except:
            pass
    else:
        killthose_pids = []
        try:
            with open(main_path+BCI_buff_path_int+"pids.txt", 'r') as pids:
                for line in pids:
                    for pid_ in line.split():
                        killthose_pids.append(pid_)

            for kill_pid in killthose_pids:
                try: os.kill(int(kill_pid), signal.SIGTERM)
                except: pass
            print('killed␣those␣processes:␣')
            print(killthose_pids)
        except: pass


# This function is used to switch between real EEG and debug mode
def debug_mode(isdebug=False):

    # Kills all buffer related processes in order to build up the new architecture corresponding to the
        respective choice
    killbuff()

    # run buffer components
    if isdebug:
        # debug_quickstart.*
        if platform.system() == 'Windows':
            os.chdir(main_path + BCI_buff_path)
            subprocess.Popen('start␣debug_quickstart.bat␣%', shell=True)
        else:
            subprocess.Popen(main_path+BCI_buff_path+'debug_quickstart.sh␣&', shell=True)
```

```
    else:
        # eeg_quickstart.*
        if platform.system() == 'Windows':
            os.chdir(main_path + BCI_buff_path)
            subprocess.Popen('start eeg_quickstart.bat %', shell=True)
        else:
            subprocess.Popen(main_path + BCI_buff_path + 'eeg_quickstart.sh &', shell=True)

def setupScripts():
    # setup scripts
    # That part is important: it uses the interpreters set in config.txt to call the respective scripts
    # the general syntax is: your_interpreter your_script

    # the order of the scripts is:
    #   (0): Signal Viewer (MATLAB)                 default: /private/sigViewer_wrapper.m
    #   (1): Calibration Signal (MATLAB)            default: /private/calib_sig.m
    #   (2): Feedback Signal (MATLAB)               default: /private/feedback_sig.m
    #   (3): Calibration Stimulus (Python)          default: /private/calib_stim.py
    #   (4): Feedback Stimulus (Python)             default: /private/brainflyTest.py
    #   (5): Feedback Stimulus v2 (Python)          default: /private/SS_Game_BCI.py

    # The scripts are called within the key listener object (class below):
    # depending on the operating system a suffix will be appended in order to make the system command not wait
        (% or &)
    # Further a boolean indicating keyboard control will be used in Feedback Stimuli

    # the lines below might seem odd, but what actually happens is that a command is created and send to the
        default
    # command line tool in the respective language (Batch or Bash). The command is composed by the respective
        call
    # function for the interpreter instance (MATLAB or Python) followed by the command to open the respective
        script.
    # In the MATLAB case the script is not called directly but wrapped inside a command. This is due to the
        fact that
    # the command line Version of MATLAB does not accept script input, but only command input. For this reason
        the script
    # was wrapped inside a "try" statement

    scripts_=[]
    scripts_.append(MATLABrootpath+' '+'"'"try;run('"+main_path+functionspath+"sigViewer_wrapper.m"');exit;end
        ;exit"'"')
    scripts_.append(MATLABrootpath+' '+'"'"try;run('"+main_path+functionspath+"calib_sig.m"');exit;end;exit"'"
        ')
    scripts_.append(MATLABrootpath+' '+'"'"try;run('"+main_path+functionspath+"feedback_sig.m"+"');exit;end;
        exit"'"')
    scripts_.append(Pythonrootpath+' '+'"'+main_path+functionspath+"calib_stim.py"+'"')
    scripts_.append(Pythonrootpath+' '+'"'+main_path+functionspath+"brainflyTest.py"+'"')
    scripts_.append(Pythonrootpath+' '+'"'+main_path+functionspath+"SS_Game_BCI.py"+'"')
    return scripts_

# Sets up the main window of the GUI
def setupMainwindow(winsize=[800,600]):

    # load assets
    icons=[]
    icons.append(Image.open(main_path+assetpath+os.sep+"EEG_off_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"EEG_on_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"set_False_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"set_True_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"keyboard_icon.png"))
    icons.append(Image.open(main_path + assetpath + os.sep + "high_res_off.png"))
    icons.append(Image.open(main_path + assetpath + os.sep + "high_res_on.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"GOL_on_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"GOL_off_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"color_on_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"color_off_icon.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"time_on.png"))
    icons.append(Image.open(main_path+assetpath+os.sep+"time_off.png"))


    # setup display
    mywin=visual.Window(winsize,units='pix',monitor='testMonitor',winType="pygame")

    # define background
    BG_=visual.ImageStim(mywin,image=Image.open(main_path+assetpath+os.sep+"GUI_background.png"))
    BG_.setAutoDraw(True)

    # EEG indicator logo
    EEG_indicator=visual.ImageStim(mywin,image=icons[int(not initdebug)],pos=[-300,250],units='pix')
```

```
    EEG_indicator.setAutoDraw(True)
    EEG_indicator_text=visual.TextStim(mywin,text='EEG mode',font='Futura',pos=[-300,210],units='pix',height
        =30)
    EEG_indicator_text.setAutoDraw(True)


    # GOL indicator logo
    GOL_indicator=visual.ImageStim(mywin,image=icons[7],pos=[-300,150],units='pix', size=(75,75))
    GOL_indicator.setAutoDraw(True)

    # Color indicator logo
    Color_indicator=visual.ImageStim(mywin,image=icons[9],pos=[-300,75],units='pix', size=(75,75))
    Color_indicator.setAutoDraw(True)

    # Time indicator logo
    Time_indicator=visual.ImageStim(mywin,image=icons[12],pos=[325,-150],units='pix', size=(75,75))
    Time_indicator.setAutoDraw(True)

    # Keyboard indicator logo
    Keyboard_=visual.ImageStim(mywin,image=icons[4],pos=[-300,-250])
    Keyboard_.setAutoDraw(True)
    Keyboard_indicator=visual.ImageStim(mywin,image=icons[3],pos=[-200,-250])
    Keyboard_indicator.setAutoDraw(True)

    # eye-candy indicator logo
    high_res_indicator = visual.ImageStim(mywin, image=icons[6], pos=[325, -250])
    high_res_indicator.setAutoDraw(True)

    # main menu - All Text based main choices within the GUI are defined here
    main_menu=[]
    main_menu.append(visual.TextStim(mywin,text='EEG Viewer',font='Helvetica',pos=[0,-50],units='pix',height
        =36))
    main_menu[0].setAutoDraw(True)

    main_menu.append(visual.TextStim(mywin,text='Calibration',font='Helvetica',pos=[0,-100],units='pix',height
        =36))
    main_menu[1].setAutoDraw(True)

    main_menu.append(visual.TextStim(mywin,text='Play Game',font='Helvetica',pos=[0,-150],units='pix',height
        =48))
    main_menu[2].setAutoDraw(True)

    main_menu.append(visual.TextStim(mywin,text='exit',font='Helvetica',pos=[0,-200],units='pix',height=36))
    main_menu[3].setAutoDraw(True)

    main_menu.append(visual.Line(mywin, start=((main_menu[2].pos[0]-main_menu[2].width/2),(main_menu[2].pos[1]-
        main_menu[2].height/2)),
                                 end=((main_menu[2].pos[0]+main_menu[2].width/2), (main_menu[2].pos[1]-main_menu
                                     [2].height/2)),units='pix'))
    main_menu[4].setAutoDraw(True)

    # pre-initialize audio
    pygame.mixer.pre_init(44100, 16, 2, 4096)
    pygame.mixer.init(44100, -16,2,2048)

    # initialize background music
    pygame.mixer.music.load(main_path+assetpath+os.sep+'backgroundmusic.wav')
    pygame.mixer.music.set_volume(1)
    pygame.mixer.music.play(-1)
    return mywin,main_menu,EEG_indicator,GOL_indicator,Color_indicator, Time_indicator, Keyboard_indicator,
        high_res_indicator,icons

# update line under the respective menu option of choice
def updateMenu():
    main_menu[4].start = ((main_menu[curr_menu_idx].pos[0] - main_menu[curr_menu_idx].width / 2),
                          (main_menu[curr_menu_idx].pos[1] - main_menu[curr_menu_idx].height / 2))

    main_menu[4].end = ((main_menu[curr_menu_idx].pos[0] + main_menu[curr_menu_idx].width / 2),
                        (main_menu[curr_menu_idx].pos[1] - main_menu[curr_menu_idx].height / 2))
    mywin.flip()
    return 0

# splash screen that is displayed when starting the GUI, while buffer functions are initialized in the
    background
# This was done to ensure that everything is loaded / initialized properly before any of the critical functions
    are called.
# This way crashes could mostly be avoided
def splashscreen_():
    #pdb.set_trace()
```

```
    mywin_splash = visual.Window([400,300], units='pix', monitor='testMonitor', winType="pygame")
    BG_ = visual.ImageStim(mywin_splash, image=Image.open(main_path + assetpath + os.sep + "splashscreen.jpg"))
    BG_.setAutoDraw(True)
    mywin_splash.flip()
    return mywin_splash

# PyGame Key-listener
# Processes all input from the user in the main window. After changing a respective setting, processes and
    commands
# that the actual change applies to will be selected. When then using one of the main GUI functions the
    respective
# settings apply to the respective choices.
class keylistener_(object):

    def __init__(self):
        # predefinition of all sorts of stuff

        self.curr_menu_idx = 2 # selection initially on "Play Game"
        self.EEG_is_on = not initdebug
        self.Keyboard_is_on = True
        self.braincontrol='␣'+str(int(not self.Keyboard_is_on))
        self.update_menu = 1 # to draw the main window for the first time
        self.done = False # for the while loop that executes the GUI
        self.music=True
        self.Stevens_version=True # set to False if you prefer the default brainFly version
        self.color_mode = True # set to False for non-color stimuli
        self.use_gol = True # set to False for calibration without the game of life
        self.use_timer = False # set to True to set time limit in game to 90 seconds

        if platform.system() == 'Windows':
            self.skip_suffix = '␣%'
        else:
            self.skip_suffix = '␣&'

    def __call__(self, ev_):
        if len(ev_) > 0: # ensure that keyboard events exist

            #ev_ = ev_[-1]  # use lates event

            # navigation in main window
            if ev_ == 'up' or ev_[pygame.K_UP]:
                self.curr_menu_idx -= 1
                self.update_menu = 1

            if ev_ == 'down' or ev_[pygame.K_DOWN]:
                self.curr_menu_idx += 1
                self.update_menu = 1

            # switching game mode
            if ev_ == 'g' or ev_[pygame.K_g]:
                self.Stevens_version=not self.Stevens_version
                high_res_indicator.image = icons[int(self.Stevens_version)+5]
                self.update_menu = 1

            # switching eeg mode
            if ev_ == 'e' or ev_[pygame.K_e]:
                self.EEG_is_on = not self.EEG_is_on
                EEG_indicator.image = icons[int(self.EEG_is_on)]
                self.update_menu = 1
                debug_mode(not self.EEG_is_on)

            # switching music on/off
            if ev_ == 'm' or ev_[pygame.K_m]:
                self.music = not self.music
                pygame.mixer.music.set_volume(int(self.music))

            # switching keyboard on/off
            if ev_ == 'k' or ev_[pygame.K_k]:
                self.Keyboard_is_on = not self.Keyboard_is_on
                Keyboard_indicator.image = icons[int(self.Keyboard_is_on) + 2]
                self.braincontrol='␣'+str(int(not self.Keyboard_is_on))
                self.update_menu = 1

            # switching color mode
            if ev_ == 'c' or ev_[pygame.K_c]:
                self.color_mode = not self.color_mode
                Color_indicator.image = icons[int(not self.color_mode)+9]
                self.update_menu = 1
```

```
            # switching game of life
            if ev_ == 'u' or ev_[pygame.K_u]:
                self.use_gol = not self.use_gol
                GOL_indicator.image = icons[int(not self.use_gol) + 7]
                self.update_menu = 1

            # switching time limit in game
            if ev_ == 't' or ev_[pygame.K_t]:
                self.use_timer = not self.use_timer
                Time_indicator.image = icons[int(not self.use_timer) + 11]
                self.update_menu = 1

        # loop the menu
        if self.curr_menu_idx < 0:
            self.curr_menu_idx = 3

        if self.curr_menu_idx > 3:
            self.curr_menu_idx = 0

        # check selection
        if ev_ == 'return' or ev_[pygame.K_ESCAPE] or ev_[pygame.K_RETURN]:

            # 'exit' was selected
            if self.curr_menu_idx == 3:
                self.done = True

            else:
                # turn off music when necessary
                if self.curr_menu_idx != 0:
                    self.music = not self.music
                    pygame.mixer.music.set_volume(0)

                # call signal Viewer
                if self.curr_menu_idx == 0:
                    try: subprocess.Popen(scripts_[0]+self.skip_suffix,shell=True);print(scripts_[0]+self.
                         skip_suffix)
                    except: pass

                # call calibration
                if self.curr_menu_idx == 1:
                    try: subprocess.Popen(scripts_[1]+self.skip_suffix,shell=True);print(scripts_[1]+self.
                         skip_suffix);time.sleep(15)
                    except: pass
                    try: subprocess.Popen(scripts_[3]+'␣'+str(int(self.color_mode))+'␣'+str(int(self.use_gol))+
                         self.skip_suffix, shell=True);print(scripts_[3]+ '␣' + str(int(self.color_mode)) +'␣'
                         +str(int(self.use_gol)) + self.skip_suffix)
                    except: pass

                # call Game
                if self.curr_menu_idx == 2:
                    if not self.Keyboard_is_on:
                        try: subprocess.Popen(scripts_[2]+self.skip_suffix, shell=True);print(scripts_[2]+self.
                             skip_suffix);time.sleep(15)
                        except: pass
                    try: subprocess.Popen(scripts_[4+int(self.Stevens_version)]+self.braincontrol + '␣' + str(
                         int(self.color_mode)) + '␣' + str(int(self.use_timer)) + self.skip_suffix, shell=True
                         );print(scripts_[4+int(self.Stevens_version)]+self.braincontrol+ '␣' + str(int(self.
                         color_mode)) + '␣' + str(int(self.use_timer))+self.skip_suffix)
                    except: pass

        return self.curr_menu_idx, self.update_menu, self.done

# show splash screen
mywin_splash=splashscreen_()

sleep_timer=setupOS()
# initial EEG mode setting -> EEG: set to False
debug_mode(initdebug)

# prepare sub-scripts
scripts_=setupScripts()

time.sleep(10)
mywin_splash.close()

# create screen
mywin,main_menu,EEG_indicator,GOL_indicator,Color_indicator, Time_indicator, Keyboard_indicator,
    high_res_indicator,icons=setupMainwindow()
```

```
# create key-listener
get_keys=keylistener_()

# run GUI
done=False
while not done:
    #ev_=event.getKeys()
    ev_ =  pygame.key.get_pressed()
    curr_menu_idx,update_menu,done=get_keys(ev_)
    if update_menu: update_menu=updateMenu()
    time.sleep(sleep_timer)

# clear screen + kill buffers
mywin.close()
if platform.system() == 'Windows':
    os.system("start cmd /c taskkill /IM cmd.exe")
killbuff()
try: os.remove(main_path+BCI_buff_path+"pids.txt")
except: pass
core.quit()
```

B.2 EEG Viewer (private/sigViewer_wrapper.m)

```
%      ____                 _       ______ _                ____    _____ _____
%     |  _ \               (_)     |  ____| |              |  _ \ / ____|_   _|
%     | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |      | |
%     |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |      | |
%     | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
%     |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
%                                         __/ |
%                                        |___/
%
% This functions is used to call the buffer's default signal viewer
% It is only a wrapper function that finds the current path structure,
% connects to the buffer and launches the signal viewer (sigViewer.m)

clc
clear
close all
try;run([pwd filesep filesep '..' filesep 'external' filesep 'matlab' filesep 'utilities' filesep 'initPaths.m'
    ]);end

buffhost='localhost';buffport=1972;

% wait for the buffer to return valid header information
hdr=[];
while ( isempty(hdr) || ~isstruct(hdr) || (hdr.nchans==0) ) % wait for the buffer to contain valid data
  try
    hdr=buffer('get_hdr',[],buffhost,buffport);
  catch
    hdr=[];
    fprintf('Invalid header info ... waiting.\n');
  end;
  pause(1);
end;
%capFitting;
sigViewer(buffhost,buffport)
exit
```

B.3 Calibration Stimulation (private/calib_stim.py)

```
"""
  ____              _        ______ _          ____   _____ _____
 |  _ \            (_)      |  ____| |        |  _ \ / ____|_   _|
 | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |      | |
 |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |      | |
 | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
 |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
                                     __/ |
                                    |___/

"""

# This function is used during the calibration phase. SSVEPs are created and paired with the selected
    background.
# In the appendix of the report a users manual is provided.

# import all necessary libraries
import sys
from time import sleep
import os
import matplotlib
import numpy as np
from threading import Timer

from psychopy import visual, core,event

matplotlib.rcParams['toolbar']='None'

# parent path from where the script was run
main_path=os.path.dirname(os.path.abspath(__file__))+os.sep+'..'+os.sep

# sets up all path dependencies needed for the function to run
BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath="assets"+os.sep+"GUI_grafix"
functionspath="private"+os.sep

# add paths
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+bufferpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+sigProcPath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+assetpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+functionspath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+BCI_buff_path))

import platform

##### BCI enable #######
if __name__ == "__main__":
    braincontrol = 1#int(sys.argv[1])
    ##### BCI enable #######
    if len(sys.argv) > 1:
        color = int(sys.argv[1])
        use_gol = int(sys.argv[2])
    else:
        color = 1
        use_gol = 1
else:
    braincontrol=0
    color = 1
    use_gol = 1

print(braincontrol)
print(color)
print(use_gol)

# if using real EEG as input
# In that case the FieldTrip buffer and bufhelp is imported and the respective paths are set
# Further the function connects to the buffer
if braincontrol:
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))
    import FieldTrip
    import bufhelp
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

    ## CONFIGURABLE VARIABLES
    # Connection options of fieldtrip, hostname and port of the computer running the fieldtrip buffer.
```

```
    hostname='localhost'
    port=1972

    ## init connection to the buffer
    timeout=5000
    (ftc,hdr) = bufhelp.connect(hostname,port)

    # Wait until the buffer connects correctly and returns a valid header
    hdr = None
    while hdr is None :
        print(('Trying to connect to buffer on %s:%i ...'%(hostname,port)))
        try:
            ftc.connect(hostname, port)
            print('\nConnected - trying to read header...')
            hdr = ftc.getHeader()
        except IOError:
            pass

        if hdr is None:
            print('Invalid Header... waiting')
            sleep(1)
        else:
            print(hdr)
            print((hdr.labels))
    fSample = hdr.fSample

    # send start even value
    def sendEvent(event_type, event_value=1, sample=-1):
        e = FieldTrip.Event()
        e.type=event_type
        e.value=event_value
        e.sample=sample
        ftc.putEvents(e)

BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath="assets"+os.sep

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

# setup video drivers
if platform.system() == 'Windows':
    os.environ['SDL_VIDEODRIVER'] = 'directx'
else:
    os.environ['SDL_VIDEODRIVER'] = 'quartz'

## CONFIGURABLE VARIABLES
# Connection options of fieldtrip, hostname and port of the computer running the fieldtrip buffer.
hostname='localhost'
port=1972

## init connection to the buffer
timeout=5000
ftc = FieldTrip.Client()
# Wait until the buffer connects correctly and returns a valid header
hdr = None
while hdr is None :
    print(('Trying to connect to buffer on %s:%i ...'%(hostname,port)))
    try:
        ftc.connect(hostname, port)
        print('\nConnected - trying to read header...')
        hdr = ftc.getHeader()
    except IOError:
        pass

    if hdr is None:
        print('Invalid Header... waiting')
        sleep(1)
    else:
        print(hdr)
        print((hdr.labels))
fSample = hdr.fSample

def sendEvent(event_type, event_value=1, sample=-1):
    e = FieldTrip.Event()
    e.type=event_type
```

```
    e.value=event_value
    e.sample=sample
    ftc.putEvents(e)

# setup Game of Life
class GOL(object):

    def __init__(self, x):
        self.gridsize_=int(x)
        self.god = 0.001
        gridsize_=self.gridsize_
        gol=np.zeros((gridsize_**2),dtype='float32')
        num_start_clusters = int(np.round(gridsize_ / 5.))
        clustersize = int(np.round(gridsize_ / 10.))
        clustercomplexity = 0.1

        # define grid structure
        c1 = [[0, 0, 0], [1, 1, 1], [2, 2, 2]], [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
        c2 = [[1], [1]]

        offsets = np.ravel_multi_index(c1, dims=(gridsize_, gridsize_), order='F') - np.ravel_multi_index(c2,
              dims=(
            gridsize_, gridsize_), order='F')

        # initialize starting clusers randomly given a certain size and complexity (added noise)
        for i in range(num_start_clusters):
            tmp=int(np.ceil((np.round(np.random.rand() * clustersize + 1))))
            randclust = np.zeros(tmp**2,dtype='float32')
            randclust[np.where(np.random.rand(np.size(randclust)) > clustercomplexity)] = 1
            size_clust = tmp
            randloc = (np.round(np.random.rand(1,2) * (gridsize_ - size_clust - 2)) + 1).astype('int16')[0]

            col_=np.repeat(range(randloc[0],randloc[0] + size_clust), size_clust, 0)
            row_ = np.repeat(np.reshape(range(randloc[1], randloc[1] + size_clust),(1,size_clust)), size_clust,
                   0).reshape(tmp**2,1).reshape(1,tmp**2)
            # get indices of cluster and update the new arrangement
            randindclust = np.ravel_multi_index([col_,row_], dims=(gridsize_,gridsize_), order='F')
            gol[randindclust[0]] = randclust

        self.gol_old=gol+0
        self.offsets=offsets.reshape(1,9)


    def __call__(self):
        # if all cells are dead, re-initialize display (like __init__)
        if self.gol_old.sum()<1:
            self.god = 0.001
            gridsize_ = self.gridsize_
            gol = np.zeros((gridsize_ ** 2))
            num_start_clusters = np.round(gridsize_ / 5)
            clustersize = np.round(gridsize_ / 10)
            clustercomplexity = 0.1

            c1 = [[0, 0, 0], [1, 1, 1], [2, 2, 2]], [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
            c2 = [[1], [1]]

            offsets = np.ravel_multi_index(c1, dims=(gridsize_, gridsize_), order='F') - np.ravel_multi_index(
                  c2, dims=(
                gridsize_, gridsize_), order='F')

            for i in range(num_start_clusters):
                tmp = int(np.ceil((np.round(np.random.rand() * clustersize + 1))))
                randclust = np.zeros(tmp ** 2)
                randclust[np.where(np.random.rand(np.size(randclust)) > clustercomplexity)] = 1
                size_clust = tmp
                randloc = (np.round(np.random.rand(1, 2) * (gridsize_ - size_clust - 2)) + 1).astype('int64')
                      [0]

                col_ = np.repeat(range(randloc[0], randloc[0] + size_clust), size_clust, 0)
                row_ = np.repeat(np.reshape(range(randloc[1], randloc[1] + size_clust), (1, size_clust)),
                       size_clust,
                                 0).reshape(tmp ** 2, 1).reshape(1, tmp ** 2)

                randindclust = np.ravel_multi_index([col_, row_], dims=(gridsize_, gridsize_), order='F')
                gol[randindclust[0]] = randclust

            gol = gol.reshape(gridsize_, gridsize_)
            self.gol_old = gol.reshape(1, gridsize_ ** 2)[0] + 0
            self.offsets = offsets.reshape(1, 9)
```

```
        # find neighbouring cells for evaluation of the rules
        gol = self.gol_old+0
        offsets=self.offsets+0
        livingidx=np.where(gol==1)[0]
        size_idx=len(livingidx)
        neigharrayidx=np.tile(livingidx, (9, 1)).transpose()+np.tile(offsets[0], (size_idx, 1))
        neigharrayidx[neigharrayidx < 0] = 0
        neigharrayidx[neigharrayidx >= (self.gridsize_**2)] = 0

        # those that have less than 3 or more than 4 neighbours die of starvation or overpopulation
        # furthermore a random fraction is killed anyways by "God"
        neigharray=gol[neigharrayidx]
        killidx=(neigharray.sum(1)<3) | (neigharray.sum(1)>4)
        randkillidx=np.random.rand(size_idx)<self.god

        # evaluating offspring
        offspridx=np.tile(livingidx, (9, 1)).transpose()[:,range(4)+range(5,9)]+np.tile(offsets[0][range(4)+
            range(5,9)], (size_idx, 1))
        offspridx=offspridx.flatten()
        offspridx[offspridx < 0] = 0
        offspridx[offspridx >= (self.gridsize_ ** 2)] = 0

        size_idx = len(offspridx)

        neigharrayidx = np.tile(offspridx, (9, 1)).transpose() + np.tile(offsets[0], (size_idx, 1))
        neigharrayidx[neigharrayidx < 0]=0
        neigharrayidx[neigharrayidx >= (self.gridsize_**2)] = 0

        neigharray = gol[neigharrayidx]

        # dead cells with exactly 3 living neighbours become alive
        # furthermore a random fraction re-born by "God"
        aliveidx = (neigharray.sum(1) == 3)
        randaliveidx = np.random.rand(size_idx) < self.god

        # update current state
        gol[livingidx[killidx+randkillidx]] = 0
        gol[offspridx[aliveidx+randaliveidx]] = 1

        self.gol_old=gol+0
        return self.gol_old

# setup stimuli for SSVEP stimulation
class stimuli_(object):
        def __init__(self, mywin, color):
            self.mywin = mywin
            win_s = self.mywin.size

            # depending on color choice in the main menu, the colors will be set to red and blue or both white
            if color:
                blue = [-1, -1, 1]
                red = [1, -1, -1]
            else:
                blue = [1, 1, 1]
                red = [1, 1, 1]

            # background
            self.pattern1 = visual.ImageStim(win=mywin, name='pattern1', units='pix',
                                             size=[win_s[0], win_s[1] - win_s[0] / 8], pos=(0, win_s[0] / 8))
            # right circle
            self.pattern2 = visual.Circle(win=self.mywin,
                                          pos=[win_s[0] / 2 - win_s[0] / 10 + 10, -win_s[1] / 2 + win_s[1] /
                                               10],
                                          radius=win_s[0] / 16, edges=32, fillColor=blue, lineColor=[-1, -1,
                                                -1],
                                          units='pix')
            # left circle
            self.pattern3 = visual.Circle(win=self.mywin,
                                          pos=[-win_s[0] / 2 + win_s[0] / 10 - 10, -win_s[1] / 2 + win_s[1] /
                                               10],
                                          radius=win_s[0] / 16, edges=32, fillColor=red, lineColor=[-1, -1,
                                                -1],
                                          units='pix')
            # screen line
            self.pattern4 = visual.ShapeStim(win=self.mywin, vertices=(
                [-win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 10)],
                [-win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 9)],
                [win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 9)],
```

```
            [win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 10)]),
                                        lineColor=[1, 1, 1], units='pix')
        # stimulation text (side indicators and fixation cross)
        self.pattern5 = visual.TextStim(win=mywin, pos=(0, -(win_s[1] / 2 - win_s[1] / 10 - 10)), text='+',
                                    color=[1, 1, 1], units='pix', height=win_s[0] / 8)
        self.pattern6 = visual.TextStim(win=mywin, pos=(0, -(win_s[1] / 2 - win_s[1] / 10 - 10)), text='',
                                    color=[1, 1, 1], units='pix', height=win_s[0] / 8)
        self.pattern7 = visual.TextStim(win=mywin, pos=(0, -(win_s[1] / 2 - win_s[1] / 10 - 10)), text='',
                                    color=[1, 1, 1], units='pix', height=win_s[0] / 8)

        # set timers for stimulation
        self.Trialclock = core.Clock()

        self.start_time1 = self.Trialclock.getTime()
        self.start_time2 = self.Trialclock.getTime()
        self.start_time3 = self.Trialclock.getTime()
        self.start_time4 = self.Trialclock.getTime()

        # set screen components to draw
        self.pattern1.setAutoDraw(True)
        self.pattern2.setAutoDraw(True)
        self.pattern3.setAutoDraw(True)
        self.pattern4.setAutoDraw(True)
        self.pattern5.setAutoDraw(True)
        self.pattern6.setAutoDraw(True)
        self.pattern7.setAutoDraw(True)

        self.instructions=['<','>']

    # screen update
    def __call__(self, numtrials_per_cond_act,freq=15, freq2=10):
        # set frequencies for SSVEPs. Note, that it is advisable to chose frquencies that are not first
            order harmonics
        # and are in accordance with the refresh rate of your screen
        dur = 1. / freq
        dur2 = 1. / freq2

        # enable stimulus (half cycle + some buffer time, hence the 0.45)
        if ((self.Trialclock.getTime() - self.start_time1) > (dur * 0.45)):
            # correct for buffer time
            if (dur * 0.45 - (self.Trialclock.getTime() - self.start_time1)) > 0:
                core.wait(dur * 0.49 - (self.Trialclock.getTime() - self.start_time1))
            self.pattern3.setAutoDraw(True)

        if ((self.Trialclock.getTime() - self.start_time2) > (dur2 * 0.45)):
            if (dur2 * 0.45 - (self.Trialclock.getTime() - self.start_time2)) > 0:
                core.wait(dur2 * 0.49 - (self.Trialclock.getTime() - self.start_time2))
            self.pattern2.setAutoDraw(True)

        # disable stimulus (half cycle + half cycle enable + some buffer time, hence the 0.95)
        if ((self.Trialclock.getTime() - self.start_time1) > (dur * 0.95)):
            # correct for buffer time
            if (dur * 0.95 - (self.Trialclock.getTime() - self.start_time1)) > 0:
                print(self.Trialclock.getTime() - self.start_time1)
                core.wait(dur * 0.99 - (self.Trialclock.getTime() - self.start_time1))
            self.start_time1 = self.Trialclock.getTime()
            self.pattern3.setAutoDraw(False)

        if ((self.Trialclock.getTime() - self.start_time2) > (dur2 * 0.95)):
            if (dur2 * 0.95 - (self.Trialclock.getTime() - self.start_time2)) > 0:
                core.wait(dur2 * 0.99 - (self.Trialclock.getTime() - self.start_time2))
            self.start_time2 = self.Trialclock.getTime()
            self.pattern2.setAutoDraw(False)

        # update trial according to trial length
        if ((Trialclock.getTime() - self.start_time3) > trialtime):
            self.start_time3 = Trialclock.getTime()
            values_ = ['1_LH','2_RH']
            idx = np.random.randint(2)
            t = Timer(rec_wait_time, sendEvent, ['stim.target', values_[idx]])

            numtrials_per_cond_act[0][idx] += 1
            self.pattern5.setPos(self.pattern2.pos)
            self.pattern5.setText(self.instructions[idx])

            self.pattern6.setPos(self.pattern3.pos)
            self.pattern6.setText(self.instructions[idx])

            self.pattern7.setText(self.instructions[idx])
```

```
            t.start()
        # return current trial count per condition
        return numtrials_per_cond_act

# Game of life presetting of the squared grid
grid_size=100
# Stimulation window size at a ratio of 4:3
window_size=800

trialtime = 2.5 # time per trial
rec_wait_time = 0.5 # wait 0.5s until target event is sent to only record last second
numtrials_per_cond=80

#### Using PsychoPy and pygame ####
mywin=visual.Window([window_size,window_size*0.75],color=(-1,-1,-1),units='pix',monitor='testMonitor',winType="
    pygame")
stim = stimuli_(mywin, color)

Trialclock = core.Clock()

numtrials_per_cond_act=np.atleast_2d(np.zeros(2))
gol=GOL(grid_size)

# sending start event that can be read by the data collector
sendEvent('stimulus.training','start')
done=False

# run calibration
while (numtrials_per_cond_act.sum(0)<numtrials_per_cond).any()==True & (not done):

    if use_gol: # obtain latest gol frame
        data_ = gol().reshape(grid_size, grid_size) + 0
        data_[data_ == 0] = -1
    else: # background is black screen
        data_ = -np.ones((grid_size, grid_size),dtype='float32')

    stim.pattern1.setImage(data_)
    numtrials_per_cond_act=stim(numtrials_per_cond_act)
    mywin.flip() # update screen
    ev_ = event.getKeys()
    if len(ev_)>0:
        ev_=ev_[-1]
        if ev_ == 'escape':
            done = True
# if calibration ends, exit screen
sendEvent('stim.training','end')
stim.pattern5.setText('end')
mywin.flip()
core.quit()
```

B.4 Calibration Classification (private/calib_sig.m)

```
%      ____                 _       ______ _                ____    _____ _____
%     |  _ \               (_)     |  ____| |              |  _ \ / ____|_   _|
%     | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |        | |
%     |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |        | |
%     | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
%     |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
%                                         __/ |
%                                        |___/
%
% This functions is used to record and process calibration data it receives
% from the buffer. It finds the current path structure,
% connects to the buffer. Afterwards a selection dialog promts fro the user
% to select the respective EEG cap. Once done it starts recording data
% according to the predefined instructions. After receiving the exit event
% a classifier is trained on that data. Output data is saved in /output and
% classifier results in /classifier
%
% default settings:
%
% trlen_ms=2000;
% spatialfilter = 'CAR'
% freqband = [0.1 1 30 31]
% badchrm = 0
% badtrrm = 0

clc
clear
close all

try;run([pwd filesep filesep '..' filesep 'external' filesep 'matlab' filesep 'utilities' filesep 'initPaths'])
    ;end

buffhost='localhost';buffport=1972;
% wait for the buffer to return valid header information
hdr=[];
while ( isempty(hdr) || ~isstruct(hdr) || (hdr.nchans==0) ) % wait for the buffer to contain valid data
  try
    hdr=buffer('get_hdr',[],buffhost,buffport);
  catch
    hdr=[];
    fprintf('Invalid␣header␣info ...␣waiting.\n');
  end;
  pause(1);
end;

CLASSIFIER_PATH = ['..' filesep 'classifiers'];
DATA_PATH = ['..' filesep 'output']

cap_=uigetfile([pwd filesep filesep '..' filesep 'external' filesep 'resources' filesep 'caps' filesep '*.txt'
    ]);
cap_=cap_(1:end-4);

% length of data that is recorded
trlen_ms=2000;

sendEvent('calib.start',1)
[data,devents,state]=buffer_waitData(buffhost,buffport,[],'startSet',{'stim.target'},'exitSet',{'stim.training'
     'end'},'trlen_ms',trlen_ms);
devents(end,:)=[]; % removing last devent entry due to the exit event
data(end,:)=[]; % removing last data entry due to the exit event

clsfr=buffer_train_ersp_clsfr(data,devents,hdr,'spatialfilter','CAR','freqband',[0.1 1 30 31],'badtrrm',0,'
    badchrm',0,'capFile',cap_,'overridechnms',1);

num = numel(dir([CLASSIFIER_PATH filesep '*.mat'])) + 1;
save([CLASSIFIER_PATH filesep 'clsfr' num2str(num)],'-struct','clsfr');

save_data.data = data;
save_data.devents = devents;
num = numel(dir([DATA_PATH filesep '*.mat'])) + 1;
save([DATA_PATH filesep 'data' num2str(num)],'-struct','save_data');

waitfor(gcf)
exit
```

B.5 Feedback Stimulation A (private/brainflyTest.py)

```
"""
  ____            _       ______ _         ____   _____ _____
 |  _ \          (_)     |  ____| |       |  _ \ / ____|_   _|
 | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |      | |
 |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |      | |
 | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
 |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
                                     __/ |
                                    |___/

"""

# This function is basic implementation of the BrainFly game. It can be played with and without keyboard
      control (EEG mode)
# In the appendix of the report a users manual is provided.

# import all necessary libraries
import pygame.locals
import sys,os
from time import sleep
import time # NEW
import matplotlib
matplotlib.rcParams['toolbar']='None'
from psychopy import visual, core
import pygame

# enable if you want animated explosions
enable_explosions=1

# enable if you want steady state stimulation
enable_stimuli=1

# fire rate of the ship in Hz
cannonfirerate=5

# spawn time of aliens in seconds
timeBeforeNextAlien= 3

# game duration in seconds
max_game_dur=90

# window size
window_size=800
deaths=0
hits=0

pygame.init()

matplotlib.rcParams['toolbar']='None'

# parent path from where the script was run
main_path=os.path.dirname(os.path.abspath(__file__))+os.sep+'..'+os.sep

# sets up all path dependencies needed for the function to run
BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath="assets"+os.sep+"GUI_grafix"
functionspath="private"+os.sep

# add paths
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+bufferpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+sigProcPath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+assetpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+functionspath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+BCI_buff_path))


import numpy as np
import platform
from PIL import Image

##### BCI enable #######
if __name__ == "__main__":
    if len(sys.argv) > 1:
        braincontrol = int(sys.argv[1])
        color = int(sys.argv[2])
```

```
    else:
        braincontrol = 0
        color = 1
else:
    braincontrol = 0
    color = 1

# if using real EEG as input
# In that case the FieldTrip buffer and bufhelp is imported and the respective paths are set
# Further the function connects to the buffer
if braincontrol:
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))
    import FieldTrip
    import bufhelp
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

    ## CONFIGURABLE VARIABLES
    # Connection options of fieldtrip, hostname and port of the computer running the fieldtrip buffer.
    hostname='localhost'
    port=1972

    ## init connection to the buffer
    timeout=5000
    (ftc,hdr) = bufhelp.connect(hostname,port)

    # Wait until the buffer connects correctly and returns a valid header
    hdr = None
    while hdr is None :
        print(('Trying to connect to buffer on %s:%i ...'%(hostname,port)))
        try:
            ftc.connect(hostname, port)
            print('\nConnected - trying to read header...')
            hdr = ftc.getHeader()
        except IOError:
            pass

        if hdr is None:
            print('Invalid Header... waiting')
            sleep(1)
        else:
            print(hdr)
            print((hdr.labels))
    fSample = hdr.fSample

    def sendEvent(event_type, event_value=1, sample=-1):
        e = FieldTrip.Event()
        e.type=event_type
        e.value=event_value
        e.sample=sample
        ftc.putEvents(e)
######################

BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath=main_path+"assets"+os.sep

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

# setup video drivers
if platform.system() == 'Windows':
    os.environ['SDL_VIDEODRIVER'] = 'directx'
else:
    os.environ['SDL_VIDEODRIVER'] = 'quartz'
# stimuli
if enable_stimuli:
    mywin=visual.Window([window_size,window_size*0.75],color=(-1,-1,-1),units='pix',monitor='testMonitor',
        winType="pygame")
    class stimuli_(object):
        def __init__(self, mywin, color):
            self.mywin = mywin
            win_s = self.mywin.size

            # depending on color choice in the main menu, the colors will be set to red and blue or both white
            if color:
                blue = [-1, -1, 1]
                red = [1, -1, -1]
            else:
```

```
            blue = [1, 1, 1]
            red = [1, 1, 1]

        # right circle
        self.pattern2 = visual.Circle(win=self.mywin,
                                      pos=[win_s[0] / 2 - win_s[0] / 10 + 10, -win_s[1] / 2 + win_s[1] /
                                          10],
                                      radius=win_s[0] / 16, edges=32, fillColor=blue, lineColor=[-1, -1,
                                          -1],
                                      units='pix')
        # left circle
        self.pattern3 = visual.Circle(win=self.mywin,
                                      pos=[-win_s[0] / 2 + win_s[0] / 10 - 10, -win_s[1] / 2 + win_s[1] /
                                          10],
                                      radius=win_s[0] / 16, edges=32, fillColor=red, lineColor=[-1, -1,
                                          -1],
                                      units='pix')
        # split line circle
        self.pattern4 = visual.ShapeStim(win=self.mywin, vertices=(
            [-win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 10)],
            [-win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 9)],
            [win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 9)],
            [win_s[0] / 2, -(win_s[1] / 2 - win_s[1] / 5 - 10)]),
                                         lineColor=[1, 1, 1], units='pix')

        # set timers for stimulation
        self.Trialclock = core.Clock()

        self.start_time1 = self.Trialclock.getTime()
        self.start_time2 = self.Trialclock.getTime()
        self.start_time3 = self.Trialclock.getTime()
        self.start_time4 = self.Trialclock.getTime()

        # set screen components to draw
        self.pattern2.setAutoDraw(True)
        self.pattern3.setAutoDraw(True)
        self.pattern4.setAutoDraw(True)

    def __call__(self, freq=15, freq2=10):
        # set frequencies for SSVEPs. Note, that it is advisable to chose frquencies that are not first
            order harmonics
        # and are in accordance with the refresh rate of your screen
        dur = 1. / freq
        dur2 = 1. / freq2

        # enable stimulus (half cycle + some buffer time, hence the 0.45)
        if ((self.Trialclock.getTime() - self.start_time1) > (dur * 0.45)):
            # correct for buffer time
            if (dur * 0.45 - (self.Trialclock.getTime() - self.start_time1)) > 0:
                core.wait(dur * 0.49 - (self.Trialclock.getTime() - self.start_time1))
            self.pattern3.setAutoDraw(True)

        if ((self.Trialclock.getTime() - self.start_time2) > (dur2 * 0.45)):
            if (dur2 * 0.45 - (self.Trialclock.getTime() - self.start_time2)) > 0:
                core.wait(dur2 * 0.49 - (self.Trialclock.getTime() - self.start_time2))
            self.pattern2.setAutoDraw(True)

        # disable stimulus (half cycle + half cycle enable + some buffer time, hence the 0.95)
        if ((self.Trialclock.getTime() - self.start_time1) > (dur * 0.95)):
            # correct for buffer time
            if (dur * 0.95 - (self.Trialclock.getTime() - self.start_time1)) > 0:
                print(self.Trialclock.getTime() - self.start_time1)
                core.wait(dur * 0.99 - (self.Trialclock.getTime() - self.start_time1))
            self.start_time1 = self.Trialclock.getTime()
            self.pattern3.setAutoDraw(False)

        if ((self.Trialclock.getTime() - self.start_time2) > (dur2 * 0.95)):
            if (dur2 * 0.95 - (self.Trialclock.getTime() - self.start_time2)) > 0:
                core.wait(dur2 * 0.99 - (self.Trialclock.getTime() - self.start_time2))
            self.start_time2 = self.Trialclock.getTime()
            self.pattern2.setAutoDraw(False)

pos_corr=0
stim = stimuli_(mywin, color)
else: # if stimuli are disabled, set up a blank screen
    pos_corr = (window_size*0.625-window_size*0.75)/2.
    mywin=visual.Window([window_size,window_size*0.625],color=(-1,-1,-1),units='pix',monitor='testMonitor',
        winType="pygame")
```

```
# setup enemies
class enemy(object):
    def __init__(self,mywin):
        self.mywin=mywin

        # Alien stimuli
        self.enemy_line=visual.Line(win=self.mywin, start=(-self.mywin.size[0]/2, self.mywin.size[1]/2+pos_corr
            ), end=(self.mywin.size[0]/2, self.mywin.size[1]/2+pos_corr),lineColor='red')
        self.enemy_line.setAutoDraw(True)
        self.enemy = visual.Circle(win=self.mywin,
                                   pos=[np.random.randint(self.mywin.size[0]*0.75) - self.mywin.size
                                       [0]*0.375, self.mywin.size[1]/2+pos_corr],
                                   radius=self.mywin.size[0]/20, edges=32, fillColor=[-1, 1, -1],
                                   units='pix')
        self.enemy.setAutoDraw(True)

    # move along the y-axis according to 'stepsize' per update
    # increase the radius of the alien by 'radincrease' per update
    def __call__(self,stepsize=1.25,radincrease=0.3):
        self.enemy.setPos([self.enemy.pos[0],self.enemy.pos[1]-stepsize])
        self.enemy_line.setPos([self.enemy_line.pos[0], self.enemy_line.pos[1] - stepsize])
        self.enemy.radius = self.enemy.radius + radincrease
        pass

# setup ship
class spaceship(object):

    def __init__(self, mywin,braincontrol=0):
        self.mywin=mywin
        self.braincontrol=braincontrol
        self.ship = visual.Rect(win=self.mywin,pos=[0, -self.mywin.size[1]/4.+pos_corr*2.],
                                   width=self.mywin.size[0]/20, height=self.mywin.size[0]/25, fillColor=[1,
                                       1, 1],
                                   units='pix')
        self.ship.setAutoDraw(True)
        self.direction=0

    def __call__(self):
        stepsize=self.mywin.size[0]/80
        # if brain control accept classifier predictions as input
        if self.braincontrol:
            stepsize = self.mywin.size[0] / 16
            events = ftc.getEvents()
            events = events[-1]
            if events.type == 'classifier.prediction':
                pred = events.value
                self.direction = pred
                sendEvent('stim.target', 1)
            else:
                self.direction = 0
        else:

            if pygame.key.get_pressed()[pygame.K_LEFT]:
                self.direction=-1
            elif pygame.key.get_pressed()[pygame.K_RIGHT]:
                self.direction = 1
            else:
                self.direction = 0
        self.ship.setPos([self.ship.pos[0] +self.direction* stepsize, self.ship.pos[1]])
        # the output of the direction choice is a -1 or 1. This is used to change the direction along the x-
            axis of the ship
        # the ship will move according to 'stepsize' per update

        # statements below ensure that the sip cannot move outside the screen
        if self.ship.pos[0]<-0.4375*self.mywin.size[0]:
            self.ship.setPos([-0.4375*self.mywin.size[0], self.ship.pos[1]])

        if self.ship.pos[0]>0.4375*self.mywin.size[0]:
            self.ship.setPos([0.4375*self.mywin.size[0], self.ship.pos[1]])

        pass

# setup cannonballs
class cannonball(object):

        def __init__(self, mywin,shippos):
            self.mywin=mywin
            self.ball = visual.Circle(win=self.mywin, pos=shippos,
                                      radius=self.mywin.size[0]/53.334, edges=32, fillColor=[1, 1, 1], lineColor
```

```
                                        =[-1, -1, -1],
                                    units='pix')
            self.ball.setAutoDraw(True)

        def __call__(self):
            stepsize = self.mywin.size[0] / 80
            self.ball.setPos([self.ball.pos[0],self.ball.pos[1]+stepsize])
            pass

# setup explosion animation
class explosionanim(object):
    def __init__(self, mywin,explosion,exppos):
        self.mywin = mywin
        self.gif=explosion
        self.expl = visual.ImageStim(self.mywin,pos=exppos)
        self.expl.setAutoDraw(False)
        self.frame_counter=0
        self.frameorder=range(7,17)+range(6)

    def __call__(self):
        self.expl.setAutoDraw(False)
        self.gif.seek(self.frameorder[self.frame_counter])
        self.expl.setImage(self.gif)
        self.expl.setAutoDraw(True)
        self.frame_counter += 1
    pass

enemies=[]
balls=[]


# create all objects
ship=spaceship(mywin,braincontrol)
enemies.append(enemy(mywin))

Timerobj=core.Clock()
enemtimer=Timerobj.getTime()
canontimer=Timerobj.getTime()

gamedur=Timerobj.getTime()
fpstimer=Timerobj.getTime()

# setup HUD
hud_string='Time␣played:␣'+str(round(Timerobj.getTime()-gamedur,1))+'␣|␣hits:␣'+str(hits)+'␣|␣deaths:␣'+str(
    deaths)

HUD_ = visual.TextStim(win=mywin,pos=[-mywin.size[0]/2,mywin.size[1]/2],text=hud_string,color=[1, 1, 1],units='
    pix',height=20,alignVert='top',alignHoriz='left')
HUD_.setAutoDraw(True)

FPS_ = visual.TextStim(win=mywin,pos=[mywin.size[0]/2,mywin.size[1]/2],text='0',color=[0, 1, 1],units='pix',
    height=40,alignVert='top',alignHoriz='right')
FPS_.setAutoDraw(True)

fps_timer_col=[]

if enable_explosions:
    explosiongif = Image.open(assetpath+"explosion2.gif")
    explo = explosionanim(mywin, explosiongif, [0,0])
    explosions=[]
    show_explosion = 0

if braincontrol:
    sendEvent('stim.target', 1)
pygame.display.init()

# run Game
while Timerobj.getTime() - gamedur<=max_game_dur:
    time.sleep(1.0/100)
    if pygame.key.get_pressed()[pygame.K_ESCAPE]:
        if braincontrol:
            sendEvent('stim.target', 0)
        core.quit()

    # create enemies
    if Timerobj.getTime()-enemtimer>timeBeforeNextAlien:
        enemtimer = Timerobj.getTime()
        enemies.append(enemy(mywin))
```

```
    # update enemy pos
    if len(enemies)>0:
        for enem in enemies:
            enem()
            if enem.enemy.pos[1]<ship.ship.pos[1]:
                enem.enemy.setAutoDraw(False)
                enem.enemy_line.setAutoDraw(False)
                enemies.remove(enem)
                deaths+=1

    # update ship pos
    ship()

    # create cannonballs
    if Timerobj.getTime() - canontimer > 1./cannonfirerate:
        canontimer = Timerobj.getTime()
        balls.append(cannonball(mywin, ship.ship.pos))

    # update cannonballs pos
    if len(balls) > 0:
        for ball in balls:

            if len(enemies) > 0:
                for enem in enemies:
                    # detect hit
                    if np.sqrt((ball.ball.pos[0] - enem.enemy.pos[0]) ** 2 + (
                                ball.ball.pos[1] - enem.enemy.pos[1]) ** 2) < enem.enemy.radius:
                        ball.ball.setAutoDraw(False)
                        balls.remove(ball)
                        if enable_explosions:
                            show_explosion = 1
                            explosion_pos=enem.enemy.pos
                        enem.enemy.setAutoDraw(False)
                        enem.enemy_line.setAutoDraw(False)
                        enemies.remove(enem)
                        hits+=1

                    else:
                        if ball.ball.pos[1] > enem.enemy.pos[1]:
                            ball.ball.setAutoDraw(False)
                            balls.remove(ball)
                        else:
                            if ball.ball.pos[1] > mywin.size[0]/2:
                                ball.ball.setAutoDraw(False)
                                balls.remove(ball)
            ball()

    # update stimuli pos
    if enable_stimuli:
        stim()

    # update explosion anim
    if enable_explosions:
        if show_explosion:
            explosions.append(explosionanim(mywin, explosiongif, explosion_pos))
            show_explosion=0

        for exp_ in explosions:
            if exp_.frame_counter>15:
                exp_.expl.setAutoDraw(False)
                explosions.remove(exp_)
            else:
                exp_()

    # update HUD
    hud_string = 'Time␣played:␣' + str(round(Timerobj.getTime() - gamedur, 1)) + '␣|␣hits:␣' + str(
        hits) + '␣|␣deaths:␣' + str(deaths)
    HUD_.setText(hud_string)
    fps_timer_col.append(1. / (Timerobj.getTime() - fpstimer))
    FPS_.setText(str(int(np.round(np.mean(fps_timer_col)))))
    if len(fps_timer_col)>120:
        del fps_timer_col[0]
    fpstimer = Timerobj.getTime()

    mywin.flip()
if braincontrol:
    sendEvent('stim.target', 0)

core.quit()
```

B.6 Feedback Stimulation B (private/SS_Game_BCI.py)

```
"""
  ____            _       ______ _         ____   _____ _____
 |  _ \          (_)     |  ____| |       |  _ \ / ____|_   _|
 | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |      | |
 |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |      | |
 | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
 |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
                                     __/ |
                                    |___/

"""

# This function is advanced implementation of the BrainFly game. It can be played with and without keyboard
      control (EEG mode)
# In the appendix of the report a users manual is provided.

# import all necessary libraries
import pygame, sys
import pygame.locals
from pygame.locals import *
from time import sleep, time
import os
import matplotlib
import re



def sorted_nicely( l ):
    """ Sorts the given iterable in the way that is expected.

    Required arguments:
    l — The iterable to be sorted.

    """
    convert = lambda text: int(text) if text.isdigit() else 0
    alphanum_key = lambda key: [ convert(c) for c in re.split('(\d+)', key) ]
    return sorted(l, key = alphanum_key)

matplotlib.rcParams['toolbar']='None'

main_path=os.path.dirname(os.path.abspath(__file__))+os.sep+'..'+os.sep


BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath="assets"+os.sep+"GUI_grafix"
functionspath="private"+os.sep

# add paths
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+bufferpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+sigProcPath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+assetpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+functionspath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),main_path+BCI_buff_path))


import time
import numpy as np
import platform
from PIL import Image

##### BCI enable #######
if __name__ == "__main__":
    if len(sys.argv) > 1:
        braincontrol = int(sys.argv[1])
        color = int(sys.argv[2])
        use_timer = int(sys.argv[3])
    else:
        braincontrol = 0
        color = 1
        use_timer = 0
else:
    braincontrol = 0
    color = 1
    use_timer = 0
print(braincontrol)
```

```
print(color)
print(use_timer)
if braincontrol:
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))
    import FieldTrip
    import bufhelp
    sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

    ## CONFIGURABLE VARIABLES
    # Connection options of fieldtrip, hostname and port of the computer running the fieldtrip buffer.
    hostname='localhost'
    port=1972

    ## init connection to the buffer
    timeout=5000
    (ftc,hdr) = bufhelp.connect(hostname,port)

    # Wait until the buffer connects correctly and returns a valid header
    hdr = None
    while hdr is None :
        print(('Trying to connect to buffer on %s:%i ...'%(hostname,port)))
        try:
            ftc.connect(hostname, port)
            print('\nConnected - trying to read header...')
            hdr = ftc.getHeader()
        except IOError:
            pass

        if hdr is None:
            print('Invalid Header... waiting')
            sleep(1)
        else:
            print(hdr)
            print((hdr.labels))
    fSample = hdr.fSample

    def sendEvent(event_type, event_value=1, sample=-1):
        e = FieldTrip.Event()
        e.type=event_type
        e.value=event_value
        e.sample=sample
        ftc.putEvents(e)
######################

BCI_buff_path="external"+os.sep
bufferpath = BCI_buff_path+"dataAcq"+os.sep+"buffer"+os.sep+"python"
sigProcPath = BCI_buff_path+"python"+os.sep+"signalProc"
assetpath=main_path+"assets"+os.sep

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufferpath))
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),sigProcPath))

class Alien(object):
    def __init__(self, screen, x, alienColor, hz, image):
        '''
        Alien
        '''
        self._x = x
        self._y = int(screen.get_size()[0] * 0.06)
        self._y_now = self._y + 0  # Moving y
        self.size = int(round(screen.get_size()[0] * 0.05)) # Ball radius
        self.R_init = self.size + 0
        self.color = alienColor
        self.start_time = time.time() + 0
        self.screen = screen
        self.speed = int(screen.get_size()[1] * 0.16)  # Pixels per seconds.
        self.Force = pygame.draw.line(self.screen, (255, 0, 0), (0, self._y_now), (screen.get_size()[0], self.
            _y_now), 1)
        self.destroy = False
        self.Growth = 0.25 # Growth per second

        self.hz = hz
        self.image = image
        self.sprite = self.screen.blit(self.image, (self._x - self.image.get_rect()[2]/2, self._y_now -  self.
            image.get_rect()[3]/2))
        self.time_hz = time.time()


    def move(self):
```

```
        '''
        Alien move down
        '''
        # Timing of movement
        time_passed = (time.time() - self.start_time)


        self._y_now = int(round(self._y + self.speed*time_passed))
        if self._y_now > screen.get_size()[1]*0.9:
            self.destroy = True

        # Growth alien
        self.size = int(self.R_init + time_passed*self.R_init*self.Growth)

        self.image = pygame.transform.scale(self.image, (int(self.screen.get_size()[0] *0.001 + self.size *2),
              int(self.screen.get_size()[1]*0.001 + self.size*2)))

        # self.Force = pygame.draw.line(self.screen, (255, 0, 0), (0, self._y_now), (screen.get_size()[0], self
              ._y_now), 1)
        pygame.draw.line(screen, (255, 0, 0), (0, self._y_now),(self._x - self.size, self._y_now), 1)
        pygame.draw.line(screen, (255, 0, 0), (self._x + self.size, self._y_now),(self.screen.get_size()[0],
              self._y_now), 1)
        self.screen.blit(self.image, (self._x - self.image.get_rect()[2]/2, self._y_now -  self.image.get_rect
              ()[3]/2))
        #if (time.time() - self.time_hz) >= (1.0 / self.hz):
            # self.sprite = pygame.draw.circle(self.screen, self.color, (self._x, self._y_now), self.size)

         #   self.time_hz = time.time()
        pass

class BonusAlien(object):
    def __init__(self, screen, x, y, image):
        '''
        Bonus Alien
        '''
        self._x = x
        self._y = y
        self._y_now = y
        self.size = int(round(screen.get_size()[0] * 0.05)) # Ball radius
        self.R_init = self.size + 0
        self.color = (0,255,0)
        self.start_time = time.time() + 0
        self.screen = screen
        self.image = image
        self.sprite = self.screen.blit(self.image, (self._x - self.image.get_rect()[2] / 2, self._y_now - self.
              image.get_rect()[3] / 2))
        self.destroy = False




    def update(self):
        '''
        Alien move down
        '''
        # Timing of movement
        time_passed = (time.time() - self.start_time)

        if time_passed >= 3:
            self.destroy = True

        # self.sprite = pygame.draw.circle(self.screen, self.color, (self._x, self._y), self.size)
        self.screen.blit(self.image, (self._x - self.image.get_rect()[2] / 2, self._y_now - self.image.get_rect
              ()[3] / 2))
        pass

class Cannonball(object):
    def __init__(self, screen, x, y):
        '''
        Constructor for cannonball.
        '''
        self._x = x
        self._y = y
        self._y_now = y + 0  # Moving y
        self.size = int(round(screen.get_size()[0] * 0.01)) # Ball radius
        self.color = (255, 255, 255)
        self.start_time = time.time() + 0
        self.screen = screen
        self.speed = int(round(screen.get_size()[1] * 0.25))  # Pixels per seconds. 16% of height
```

```
        self.sprite =[]
        self.destroy = False

    def move(self):
        '''
        initiates the cannonball to move until its past the screen
        '''
        time_passed = (time.time() - self.start_time)

        self._y_now = int(round(self._y - self.speed*time_passed))
        if self._y_now < 0:
            self.destroy = True


        self.sprite = pygame.draw.circle(self.screen, self.color, (self._x, self._y_now), self.size)
        pass

def hit_alien(Alien, Ball):
    """Checks Euclidean distance between alien and ball"""
    x_squared = (Alien._x - Ball._x) ** 2
    y_sqaured = (Alien._y_now - Ball._y_now) ** 2
    L2 = np.sqrt(x_squared + y_sqaured)
    hit = (L2 - Alien.size) <= 0
    return hit

def hit_forcefield(Alien, Ball):
    """Check if balls hit the forcefield of alien"""
    hit = Alien._y_now >= (Ball._y_now - Ball.size)
    return hit


def extractFrames(inGif, outFolder):
    frame = Image.open(inGif)
    nframes = 0
    while frame:
        frame.save( '%s/%s-%s.gif' % (outFolder, os.path.basename(inGif), nframes ) , 'GIF')
        nframes += 1
        try:
            frame.seek( nframes )
        except EOFError:
            break;
    return True

class explosionanim(object):
    def __init__(self, screen,explosion_gif,exppos):
        self.screen = screen
        self.pos = exppos
        self.gif=explosion_gif
        self.frame_counter=0
        self.frameorder = range(7, 17) + range(6)


    def __call__(self):
        self.screen.blit(self.gif[self.frameorder[self.frame_counter]], self.pos)
        self.frame_counter += 1
    pass

# GIF Explosion parser
# Loading gif from seperated images
gif_direct = assetpath
# gif_direct = "D:\OneDrive\RLI\Master Artificial Intelligence\courses\BCI Practical\Project\Tests\GIftests"
extractFrames(assetpath+"explosion2.gif", gif_direct)
explosion_gif =[]


# OS
if platform.system() == 'Windows':
    os.environ['SDL_VIDEODRIVER'] = 'directx'
else:
    os.environ['SDL_VIDEODRIVER'] = 'quartz'

# Music
pygame.mixer.pre_init(44100, 16, 2, 4096)

# Initialize screen
ScreenWidth = 800
ScreenHeight = int(0.625 * ScreenWidth)
pygame.init()
```

```
if platform.system() == 'Windows':
    screen = pygame.display.set_mode((ScreenWidth, ScreenHeight), pygame.HWSURFACE | pygame.DOUBLEBUF)
else:
    screen = pygame.display.set_mode((ScreenWidth, ScreenHeight))


clock = pygame.time.Clock()
# screen.set_alpha(None)

# GIF
for i in list(range(17)):
    file = gif_direct + os.sep+"explosion2.gif" + "-" + str(i) + ".gif"
    loaded = pygame.image.load(file).convert_alpha()
    explosion_gif.append(loaded)

explosions=[]
show_explosion = 0


# Background
background = pygame.image.load(assetpath+"space_background.png")
background = pygame.transform.scale(background, (ScreenWidth, ScreenHeight))

# Images
if color:
    alien_red = pygame.image.load(assetpath+"alien_red.png").convert_alpha()
    alien_blue = pygame.image.load(assetpath+"alien_blue.png").convert_alpha()
else:
    alien_red = pygame.image.load(assetpath+"alien_red_white.png").convert_alpha()
    alien_blue = pygame.image.load(assetpath+"alien_blue_white.png").convert_alpha()

alien_red = pygame.transform.scale(alien_red, (int(ScreenWidth * 0.15), int(ScreenHeight * 0.15)))
alien_blue = pygame.transform.scale(alien_blue, (int(ScreenWidth * 0.15), int(ScreenHeight * 0.15)))

alien_red_static = pygame.transform.scale(alien_red, (int(ScreenWidth * 0.17), int(ScreenHeight * 0.17)))
alien_blue_static = pygame.transform.scale(alien_blue, (int(ScreenWidth * 0.17), int(ScreenHeight * 0.17)))

alien_bonus = pygame.image.load(assetpath+"alien_green.png").convert_alpha()
alien_bonus = pygame.transform.scale(alien_bonus, (int(ScreenWidth * 0.1), int(ScreenHeight * 0.1)))


space_ship = pygame.image.load(assetpath+"space_ship.png").convert_alpha()
space_ship = pygame.transform.scale(space_ship, (int(ScreenWidth * 0.1), int(ScreenHeight * 0.1)))

# Init cannon
square_side = int(round(0.08 * ScreenHeight)) # Length of one side of square cannon
x = ScreenWidth /2 - square_side# Start position cannon
y = ScreenHeight - square_side # Static height cannon
colorCannon = (255, 255, 255)
MoveSpeed = 0.01 # Move with speed % pixels of ScreenWdith

# Inits balls
balls=[]
ball_time = time.time()
space_tmp  = 0 # Space bar hasn't been pushed yet

# Inits aliens
aliens=[]
alien_time = time.time()
alien_secs = 3 # 1 alien per 3 sec
alien_start = 1 # For fist alien

if color:
    alienColorRight = (0, 0, 255)
    alienColorLeft = (255, 0, 0)
else:
    alienColorRight = (255, 255, 255)
    alienColorLeft = (255, 255, 255)

# Inits bonus aliens
bonusaliens = []
bonus_time = time.time()
bonus_prob = 0.05
bonus_secs = 8 # Per second there is a bonus_prob of a bonus alien

# Count inits
start_time = time.time()
shots_fired = 0
hit_score = 0
```

```
aliens_total = 0

# Frequencies
left_hz = 15 #16
right_hz = 10 #8
clock_FPS = 43 #35 # FPS 43 works!

# Emergency looks
left_time_IC = time.time()
right_time_IC = time.time()
right_IC_color = alienColorRight
left_IC_color = alienColorLeft
IC_side = int(round(0.1 * ScreenHeight))

# Music
pygame.mixer.init(44100, -16,2,2048)
pygame.mixer.music.load(assetpath+"space.mp3")
pygame.mixer.music.set_volume(1)
pygame.mixer.music.play(-1)

# Text inits
if 'font' not in locals():
    font = pygame.font.Font(None, int(ScreenHeight * 0.04)) # Python crashes if SysFont is called > 1 XOR
        specified font
    font_FPS = pygame.font.Font(None, int(ScreenHeight * 0.1))

text_y = int(round(ScreenHeight * 0.01))
x_time = int(round(ScreenWidth * 0.025))
x_static = int(round(ScreenWidth * 0.2))
x_shots = int(round(ScreenWidth * 0.21))
x_static2 = int(round(ScreenWidth * 0.35))
x_score = int(round(ScreenWidth * 0.36))
x_static3 = int(round(ScreenWidth * 0.43))
x_accuracy = int(round(ScreenWidth * 0.44))

textColor = (150, 150, 150) # So the bullet over it don't hide the text

# Autofire
fire_hz = 5
fire_last = time.time()

# For FPS testing
test_timing = time.time()
times= []
now_FPS = 0
x_FPS = int(round(ScreenWidth * 0.94))
fps_timer = time.time()

##### BCI - Start game sign
if braincontrol:
    sendEvent('stim.target', 1)

direction=[]

## Game loop
done = False

NAME = 'STV'
WRITTEN = False
TIME_LIMIT = 90 # 90 seconds
# Check time limit reached
check = lambda t: (time.time() - start_time) < TIME_LIMIT if t else 1

while not done:

    # if time limit set, then check if time limit has been reached
    if not check(use_timer):
        if braincontrol:
            sendEvent('stim.target', 0)
        screen.fill((0,0,0))
        screen.blit(background, (0, 0))

        with open('highscore.txt','a') as f:
            if not WRITTEN:
                f.write(NAME + '   ' + str(acc) + '%\n')
                WRITTEN = True
        with open('highscore.txt','r') as f:
            highscore = f.readlines()
```

```
    sort = sorted_nicely(highscore[4:])

    center = -100
    pos = -100
    for line in highscore[:4]:
        screen.blit(pygame.font.SysFont('Consolas', 35).render(line[:-1], True, (0, 255, 0)), (ScreenWidth
            /2 + center,ScreenHeight/2 + pos))
        pos += 25

    count = 0
    for line in sort[::-1]:
        if count > 4:
            break
        screen.blit(pygame.font.SysFont('Consolas', 35).render(line[:-1], True, (0, 255, 0)), (ScreenWidth
            /2 + center,ScreenHeight/2 + pos))
        pos += 25
        count += 1

    screen.blit(pygame.font.SysFont('Consolas', 35).render(text_acc, True, (0, 255, 0)), (ScreenWidth/2 +
        center-10,ScreenHeight/2 + pos+50))
else:
    for event in pygame.event.get():
        pass

    # BCI events
    if braincontrol:
        events = ftc.getEvents()
        events = events[-1]
        if events.type == 'classifier.prediction':
            pred = events.value
            direction = pred
            sendEvent('stim.target', 1)



    pressed = pygame.key.get_pressed()
    if pressed[pygame.K_LEFT] or (direction == -1):
        if (x <= 0): x -= 0 # If window limit left reached, dont do squat.
        else: x -= int(round(ScreenWidth * MoveSpeed))

    if pressed[pygame.K_RIGHT] or (direction == 1):
        if (x >= ScreenWidth - square_side): x +=0 # If window limit right reached, don't do squat.
        else: x += int(round(ScreenWidth * MoveSpeed))
    # Check for shots
    if pressed[pygame.K_SPACE] and (space_tmp==0): space_tmp=1 # Now you can check for release space

    if (event.type == pygame.KEYUP and event.key == pygame.K_SPACE) and space_tmp: # If space pushed and
        released
        balls.append(Cannonball(screen, x + square_side/2, y + square_side/2))
        space_tmp = 0 # Now don't check for release anymore
        shots_fired += 1

    if (time.time() - fire_last) >= (1.0/fire_hz):
        balls.append(Cannonball(screen, x + square_side/2, y + square_side/2))
        space_tmp = 0 # Now don't check for release anymore
        shots_fired += 1
        fire_last = time.time()

    # Set screen dark for update
    screen.fill((0, 0, 0))
    screen.blit(background, (0, 0))

    # Draw constant emergency SSVEP's
    if ((time.time() - left_time_IC) >= (1.0 / left_hz)):
        screen.blit(alien_red_static, (0, ScreenHeight - alien_red_static.get_rect()[3]))
        left_time_IC = time.time()

    if ((time.time() - right_time_IC) >= (1.0 / right_hz)):
        screen.blit(alien_blue_static, (ScreenWidth - alien_blue_static.get_rect()[2], ScreenHeight -
            alien_blue_static.get_rect()[3]))

        right_time_IC = time.time()


    # Draw the cannon position
    screen.blit(space_ship,(int(x - space_ship.get_rect()[2]/3.5), y))


    # Create aliens
```

```
if ((time.time() - alien_time) >= 3) or alien_start:
    x_position = np.random.randint(0, ScreenWidth)
    aliens_total += 1 # For accuracy!
    if x_position >= (ScreenWidth/2.0):
        aliens.append(Alien(screen, x_position, alienColorRight, right_hz, alien_blue))
        alien_time = time.time()
        alien_start = 0
    else:
        aliens.append(Alien(screen, x_position, alienColorLeft, left_hz, alien_red))
        alien_time = time.time()
        alien_start = 0


# Create Bonus Alien
if ((time.time() - bonus_time) >= bonus_secs) and (float(np.random.uniform(0, 1, 1)) <= bonus_prob):
    aliens_total += 1 # For accuracy!
    x_position = np.random.randint(0, ScreenWidth)
    y_position = np.random.randint(0, int(ScreenHeight * 0.8))
    bonusaliens.append(BonusAlien(screen, x_position, y_position, alien_bonus))
    bonus_time = time.time()

# Update balls position and delete if hit forcefield or out of FOV
if len(balls)>0:
    for i in list(reversed(range(len(balls)))):  # Reversed because if 1 is del, index out of range..
        balls[i].move()

        if balls[i].destroy:
            del balls[i]

        if len(aliens)> 0:
            if hit_forcefield(aliens[0], balls[i]):
                del balls[i]

# Draw Bonus Alien and remove if hit
if len(bonusaliens) > 0:
    for i in list(reversed(range(len(bonusaliens)))):
        bonusaliens[i].update()

        if bonusaliens[i].destroy:
            del bonusaliens[i]

    if (len(balls) > 0) and (len(bonusaliens) > 0):  # Looks for a hit by ball
        for i in list(reversed(range(len(bonusaliens)))):
            if len(balls) > 0:
                for u in list(reversed(range(len(balls)))):
                    if len(bonusaliens) > 0 and len(balls) > 0:
                        if hit_alien(bonusaliens[i], balls[u]):
                            explosion_pos = (bonusaliens[i]._x - bonusaliens[i].size*2, bonusaliens[i].
                                _y_now - bonusaliens[i].size*3)
                            show_explosion = 1
                            del bonusaliens[i]
                            del balls[u]
                            hit_score += 1

# Move aliens and remove if hit or out of range
if len(aliens) > 0:
    for i in list(reversed(range(len(aliens)))):
        aliens[i].move()

        if aliens[i].destroy:
            del aliens[i]

    if len(balls) > 0:  # Remove alien if hit
        for i in list(reversed(range(len(balls)))):
            if len(aliens) > 0:
                if hit_alien(aliens[0], balls[i]):
                    explosion_pos = (aliens[0]._x - aliens[0].size, int(aliens[0]._y_now - aliens[0].
                        size*1.5))
                    show_explosion = 1
                    del aliens[0]
                    del balls[i]
                    hit_score += 1

# Explosion
if show_explosion:
    explosions.append(explosionanim(screen, explosion_gif, explosion_pos))
    show_explosion = 0

for exp_ in explosions:
```

```
            if exp_.frame_counter > 15:
                explosions.remove(exp_)
            else:
                exp_()

        # Text time on screen
        now_time = time.time()
        time_played = round(now_time - start_time, 2)
        text_time = 'Time played: ' + str(time_played) + 's'
        text = font.render(text_time, True, textColor)
        screen.blit(text, (x_time, text_y))

        # Text shots on screen
        text_shots = 'Shots fired: ' + str(shots_fired)
        text = font.render(text_shots, True, textColor)
        screen.blit(text, (x_shots, text_y))

        # Text score
        text_hits = 'Hits: ' + str(hit_score)
        text = font.render(text_hits, True, textColor)
        screen.blit(text, (x_score, text_y))

        # Text accuracy
        acc = round(100 * hit_score/(aliens_total +0.000001),1)
        text_acc = 'Accuracy: ' + str(acc) + '%' # aliens_total used to be shots_fired!
        text = font.render(text_acc, True, textColor)
        screen.blit(text, (x_accuracy, text_y))


        # Text FPS
        text_fps = str(now_FPS)
        text = font_FPS.render(text_fps, True, (255,255,0))
        screen.blit(text, (x_FPS, text_y))


        # Text statics
        text_static = '|'
        text = font.render(text_static, True, textColor)
        screen.blit(text, (x_static, text_y))

        text_static = '|'
        text = font.render(text_static, True, textColor)
        screen.blit(text, (x_static2, text_y))

        text_static = '|'
        text = font.render(text_static, True, textColor)
        screen.blit(text, (x_static3, text_y))


    # Update screen + check FPS
    pygame.display.flip() # Update screen
    onewhile = time.time() - test_timing
    times.append(onewhile)
    if (time.time() - fps_timer) >= 0.5:
        now_FPS = int(1.0 / onewhile)
        fps_timer = time.time()
    test_timing = time.time()
    clock.tick(clock_FPS) # Set response pygame rate fo 60fps

    # Quit without error (if this loop is at the beginning, pygame finishes the loop
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            if braincontrol:
                sendEvent('stim.target', 0)
            done = True



#if use_timer:
    # Set screen dark for update
 #   screen.fill((0,0,0))
  #  screen.blit(background, (0, 0))
    #screen.blit(pygame.font.SysFont('Arial', 25).render('Accuracy: ' + text_acc, True, (0, 255, 0)), (0,0))
   # pygame.display.flip()



pygame.quit() # Close window
```

```
### BCI End game sign
if braincontrol:
    sendEvent('stim.target', 0)


# Just checking FPS
FPS = [1/float(i) for i in times]
sumFPS =0
for i in range(len(FPS)):
    sumFPS += FPS[i]
meanFPS = sumFPS / len(FPS)
print("Mean FPS: " + str(meanFPS))
```

B.7 Feedback Classification (private/feedback_sig.m)

```
%      ____                 _       ______ _                 ____   _____ _____
%     |  _ \               (_)     |  ____| |               |  _ \ / ____|_   _|
%     | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |       | |
%     |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |       | |
%     | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
%     |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
%                                         __/ |
%                                        |___/
%
% This functions is used to record and process online data it receives
% from the buffer. It generates a class label (-1 or 1) as a prediction
% from the classifier specified. It finds the current path structure,
% connects to the buffer. Afterwards a selection dialog promts fro the user
% to select a pre-trained classifier. Once done it starts recording data
% according to the predefined instructions. After receiving the exit event
% (as soon as data is available in that case) a classifier calssifies the
% input data and generates a label. Those labels are directly used to
% control the space ship for which reason they are translated to -1 or 1 in
% order to modify the ships x-axis movement.
%
% default settings:
%
% trlen_samp = 250; which corresponds to 1s of data in the present
%                                         case (sampling rate of 250Hz)

clc
clear
close all

try;run([pwd filesep filesep '..' filesep 'external' filesep 'matlab' filesep 'utilities' filesep 'initPaths'])
      ;end

buffhost='localhost';buffport=1972;
% wait for the buffer to return valid header information
hdr=[];
while ( isempty(hdr) || ~isstruct(hdr) || (hdr.nchans==0) ) % wait for the buffer to contain valid data
  try
    hdr=buffer('get_hdr',[],buffhost,buffport);
  catch
    hdr=[];
    fprintf('Invalid␣header␣info ...␣waiting.\n');
  end;
  pause(1);
end;
[clsfra,path]=uigetfile([pwd filesep filesep '..' filesep 'classifiers' filesep '*.mat']);
clsfr=load([path,clsfra])
trlen_samp=250;%samples
state=hdr;

while true
[data,devents,state]=buffer_waitData(buffhost,buffport,state,'startSet',{'stim.target'},'trlen_samp',trlen_samp
      ,'exitSet',{'data'});
if devents(end).value==0
    break
end
[f,fraw,p]=buffer_apply_ersp_clsfr(data.buf,clsfr);
sendEvent('classifier.prediction',(f>0)*2-1);
end
exit
```

B.8 Classifier for multiple datasets (private/train_classifier.m)

```
%      ____                _       ______ _          ____   _____ _____
%     |  _ \           (_)      |  ____| |        |  _ \ / ____|_   _|
%     | |_) |_ __ __ _ _ _ __ | |__  | |_   _  | |_) | |      | |
%     |  _ <| '__/ _` | | '_ \|  __| | | | | | |  _ <| |      | |
%     | |_) | | | (_| | | | | | |    | | |_| | | |_) | |____ _| |_
%     |____/|_|  \__,_|_|_| |_|_|    |_|\__, | |____/ \_____|_____|
%                                         __/ |
%                                        |___/
%
% This functions is used to train a classifier on multiple merged
% datasets. The functions prompts a dialog to select a EEG cap file that
% should correspond to the input data provided later. All data located
% within the /output folder the follows the naming data* will be merged in
% to one dataset and a classifier will be trained. The classifier result
% will be saved in \classifiers
%
% default settings:
%
% trlen_ms=2000;
% fs = 250; default sampling rate of the Mobita system
% spatialfilter = 'CAR'
% freqband = [0.1 1 30 31]
% badchrm = 0
% badtrrm = 0

clc
clear
close all

try;run([pwd filesep filesep '..' filesep 'external' filesep 'matlab' filesep 'utilities' filesep 'initPaths'])
    ;end

CLASSIFIER_PATH = ['..' filesep 'classifiers'];
DATA_PATH = [pwd filesep '..' filesep 'output' filesep 'Robert'];

cap_=uigetfile([pwd filesep filesep '..' filesep 'external' filesep 'resources' filesep 'caps' filesep '*.txt'
    ]);
cap_=cap_(1:end-4);

trlen_ms=2000;

%[data,devents,hdr,allevents] = sliceraw(DATA_PATH,'startSet',{'stim.target'},'trlen_ms',trlen_ms);
num = numel(dir([DATA_PATH filesep 'data*.mat']));
data = [];
devents = [];
for i=1:num
        all_data = load([DATA_PATH filesep 'data' num2str(i) '.mat']);
        data = [data; all_data.data];
        devents = [devents; all_data.devents];
end

[clsfr,res,X,Y]= buffer_train_ersp_clsfr(data, devents,[],'fs', 250, 'spatialfilter','CAR','freqband',[0.1 1 30
    31],'badtrrm',0,'badchrm',0,'capFile',cap_,'overridechnms',1);

num = numel(dir([CLASSIFIER_PATH filesep 'clsfr_offline*.mat'])) + 1;
save([CLASSIFIER_PATH filesep 'clsfr_offline' num2str(num)],'-struct','clsfr');
```