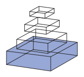
frontiers in
NEUROSCIENCE

METHODS ARTICLE
published: 02 December 2010
doi: 10.3389/fnins.2010.00179



Pyff – a Pythonic framework for feedback applications and stimulus presentation in neuroscience

Bastian Venthur[1]*, Simon Scholler[1,2], John Williamson[3], Sven Dähne[1,2], Matthias S. Treder[1], Maria T. Kramarek[1,2], Klaus-Robert Müller[1,2,4] and Benjamin Blankertz[1,3,5]

[1] Machine Learning Laboratory, Berlin Institute of Technology, Berlin, Germany
[2] Bernstein Center for Computational Neuroscience, Berlin, Germany
[3] Department of Computing Science, University of Glasgow, Glasgow, Scotland
[4] Bernstein Focus: Neurotechnology, Berlin, Germany
[5] Fraunhofer FIRST (IDA), Berlin, Germany

Edited by:
David N. Kennedy, University of Massachusetts Medical School, USA

Reviewed by:
Jonathan Peirce, Notthingham University, UK
K. Jarrod Millman, University of California at Berkeley, USA
Satrajit S. Ghosh, Massachusetts Institute of Technology, USA

***Correspondence:**
Bastian Venthur, Machine Learning Laboratory, Berlin Institute of Technology, Franklinstraße 28/29 10587 Berlin, Germany.
e-mail: bastian.venthur@tu-berlin.de

This paper introduces Pyff, the Pythonic feedback framework for feedback applications and stimulus presentation. Pyff provides a platform-independent framework that allows users to develop and run neuroscientific experiments in the programming language Python. Existing solutions have mostly been implemented in C++, which makes for a rather tedious programming task for non-computer-scientists, or in Matlab, which is not well suited for more advanced visual or auditory applications. Pyff was designed to make experimental paradigms (i.e., feedback and stimulus applications) easily programmable. It includes base classes for various types of common feedbacks and stimuli as well as useful libraries for external hardware such as eyetrackers. Pyff is also equipped with a steadily growing set of ready-to-use feedbacks and stimuli. It can be used as a standalone application, for instance providing stimulus presentation in psychophysics experiments, or within a closed loop such as in biofeedback or brain–computer interfacing experiments. Pyff communicates with other systems via a standardized communication protocol and is therefore suitable to be used with any system that may be adapted to send its data in the specified format. Having such a general, open-source framework will help foster a fruitful exchange of experimental paradigms between research groups. In particular, it will decrease the need of reprogramming standard paradigms, ease the reproducibility of published results, and naturally entail some standardization of stimulus presentation.

Keywords: neuroscience, BCI, Python, framework, feedback, stimulus presentation

1 INTRODUCTION

During the past years, the neuroscience community has been moving toward increasingly complex stimulation paradigms (Pfurtscheller et al., 2006; Brouwer and van Erp, 2010; Schreuder et al., 2010) that aim to investigate human function in a more natural setting. A technical bottleneck in this process is programming these complex stimulations. In particular, the rapidly growing field of brain–computer interfacing (BCI, Dornhege et al., 2007) requires stimulus presentation programs that can be used within a closed loop, i.e., feedback applications that are driven by a control signal that is derived from ongoing brain activity.

The present paper suggests a Python-based framework for experimental paradigms that combines the ease in programming and the inclusion of all necessary functionality for flawless stimulus presentation. This is well in line with the growing interest toward using Python in the neuroscience community (Spacek et al., 2008; Brüderle et al., 2009; Drewes et al., 2009; Jurica and VanLeeuwen, 2009; Ince et al., 2009; Pecevski et al., 2009; Strangman et al., 2009). Pyff provides a powerful yet simple and highly accessible framework for the development of complex experimental paradigms containing multi-media. To this end, it accommodates a standardized interface for implementing experimental paradigms, support for special hardware such as eye trackers and EEG equipment as well as large library of ready-to-go experiments. Class experience shows that non-expert programmers typically learn the use of our framework within 2 days. Note that a C++ implementation can easily take one order of magnitude more time to learn than the corresponding Python implementation and even for an experienced programmer a factor of two still remains (Prechelt, 2000). The primary aim of Pyff was to provide a convenient basis for programming paradigms in the context of brain–computer interfacing. To that end, Pyff can also easily be linked to BCI systems like BCI2000 (Schalk et al., 2004), and the Berlin BCI via a standard communication protocol, see Section 3. Furthermore, Pyff can be used for general stimulus presentation. This allows a seamless transition from experiments in the fields of cognitive psychology, neuroscience, or psychophysics to BCI studies.

Since Pyff is open source, it makes an ideal basis for a vivid exchange of experimental paradigms between research groups and it releases the user from needing to reprogram standard paradigms. Furthermore, providing paradigm implementations as supplementary material within the Pyff framework will ease the reproducibility of published results. The following sections of this paper introduce the software concept, detail a number of typical paradigms and conclude. Several appendices expand on the software engineering side and the code of a sample paradigm is discussed in detail.

Throughout this paper, we use the term *stimulus and feedback applications* synonymous with *experimental paradigm*. The former term is common in the BCI field, whereas the latter is better known in neuroscience. The difference between a feedback application and a stimulus application is defined by the setup of the experiment. If the experimental setup forms a closed loop, such as in a neurofeedback paradigm, we call the application a *feedback application*. If the loop is not closed, we call it *stimulus presentation*. When referring to the actual software implementation of a paradigm, we use the term *Feedback* (with a capital f), synonymous for stimulus and feedback applications.

2 RELATED WORK

Pyff is a general, high-level framework for the development of experimental paradigms within the programming language Python. In particular, Pyff can receive control signals of a BCI system to drive a feedback application within a closed-loop mode. Other software related to Pyff can be grouped in the following three categories:

(1) General Python module for visual or auditory presentation
(2) Packages for stimulus presentation and experimental control
(3) Feedback applications for the use with BCI systems

Software of categories (1) and (2) can be used within Pyff, while (3) is an alternative to Pyff. In the following, we will shortly discuss prominent examples of the three groups.

(1) These modules are usually used to write games and other software applications and can be used within Pyff to control stimulus presentation. Pygame[1] is a generic platform for gaming applications. It can be readily used to implement visual and auditory stimulus presentation. Similar to Pygame, pyglet[2] is a framework for developing games and visually rich applications and therefore suited for visual stimulus applications. PyOpenGL[3] provides bindings to OpenGL and related APIs, but requires the programmer to be familiar with OpenGL.
In Pyff, a Pygame base class (see Section 13) exists, that facilitates the development of experimental paradigms based on this module.
(2) There are some comprehensive Python libraries that provide means for creating und running experimental paradigms. Their advanced functionality for stimulus presentation can be used within Pyff. Vision Egg (Straw, 2008) is a high-level interface to OpenGL. It was specifically designed to produce stimuli for vision research experiments. PsychoPy (Peirce, 2007) is a platform-independent experimental control system written in Python. It provides means for stimulus presentation and response collection as well as simple data analysis. PyEPL (Geller et al., 2007) is a another Python library for object-oriented coding of psychology experiments which supports the presentation of visual and auditory stimuli as well as manual and sound input as responses.
Pyff provides a `VisionEggFeedback` base class which allows for easily writing paradigms using Vision Egg for stimulus presentation. The other two modules have up-to-date not been used within Pyff.
The Psychophysics Toolbox (Brainard, 1997) is a free set of Matlab and GNU/Octave functions for vision research. Being available since the 1990s, it is now a mature research tool and particularly popular among psychologists and neuroscientists. Currently, there is no principle framework to couple the Psychophysics Toolbox to a BCI system.
In addition to this, there are also commercial solutions such as E-Prime (Psychology Software Tools, Inc) and Presentation (Neurobehavioral Systems) which are software for experiment design, data collection, and analysis.
(3) BCI2000 (Schalk et al., 2004) is a general-purpose system for BCI research that is free for academic and educational purposes. It is written in C++ and runs under Microsoft Windows. BCPy2000 (Schreiner, 2008) is an extension that allows developers to implement BCI2000 modules in Python, which is less complex and less error prone than C++. BCPy200 is firmly coupled to BCI2000 resulting in a more constraint usage compared to Pyff.

3 OVERVIEW OF PYFF

This section gives an overview of our framework. A more complete and technical description is available in the Appendix.

The Pythonic feedback framework (Pyff) is a framework to develop experimental paradigms. The foremost design goal was to make the development of such applications fast and easy, even for users with little programming experience. For this reason we decided to the Python programming language as it is easier to learn than low level languages like C++. The code is shorter and clearer and thus leads to faster and less error prone results. Python is slower than a low level language like C++, but usually fast enough for multi media applications. In rare cases where Python is too slow for complex calculations, it is easy to port the computationally intensive parts to C and then call them within Python.

The framework consists of four parts: the Feedback Controller, a graphical user interface (GUI), a set of Feedbacks and a collection of Feedback base classes (see **Figure 1**).

The *Feedback Controller* controls the execution of the stimulus and feedback applications and forward incoming signals from an arbitrary data source such as a BCI system to the applications. To enable as many existing systems as possible to communicate with Pyff, we developed a simple communications protocol based on the user datagram protocol (UDP). The protocol allows for transportation of data over a network using extensible markup language (XML) to encode the signal. This protocol enables virtually any software that is able to output it's data in some form to send it to Pyff with only minor modifications.

The *GUI* controls the Feedback Controller remotely. Within the GUI the experimenter can select and start Feedbacks as well as inspect and manipulate their variables (e.g., number of trials, position, and color of visual objects). The ability to inspect the Feedback application's internal variables in real time while the application is running makes the GUI an invaluable debugging tool. Being able to modify these variables on the fly also provides a great way to explore different settings in a pilot experiment. The GUI also uses the aforementioned

[1]Pygame homepage. http://pygame.org/
[2]pyglet homepage. URL http://pyglet.org/
[3]Pyopengl homepage. URL http://pyopengl.sourceforge.net/

communication protocol and thus does not need to run on the same machine as the Feedback Controller. This can be convenient for experiments where the subject is in a different room than the experimenter. Note, that the use of the GUI is optional as everything can also be controlled remotely via UDP/XML, see Section 11.

Pyff not only provides a platform to develop *Feedback applications* and *stimuli* easily, but it is also equipped with a variety of paradigms (see Section 4 for examples). The BBCI group will continue to publish feedback and stimulus applications under a Free Software License, making them available to other research groups and we hope that others will also join our effort.

The collection of *Feedback base classes* provides a convenient set of standard methods for paradigms which can be used in derived Feedback classes to speed up the development of new Feedback applications. This standard functionality reduces the overheads of developing a new Feedback as well as minimizing code duplication. To give an example, Pygame is frequently used to provide the graphical output of the Feedback. Since some things need to be done in every Feedback using Pygame (i.e., initializing the graphics or regularly polling Pygame's event queue), we created the PygameFeedback base class. It contains methods required by all Pygame-based Feedbacks and some convenient helper methods we find useful. Using this base class for in a Pygame-based Feedback can drastically reduce the amount of new code required. It helps to concentrate on the code needed for the actual paradigm instead of dealing with the quirks of the library used. Pyff already provides some useful base classes like `VisionEggFeedback`, `EventDrivenFeedback`, and `VisualP300`. Our long term goal is to provide a rich set of base classes for standard experimental paradigms to ease the effort of programming new Feedbacks even more.

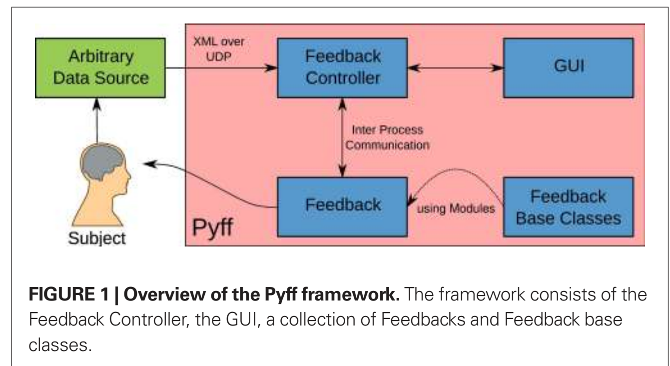


FIGURE 1 | Overview of the Pyff framework. The framework consists of the Feedback Controller, the GUI, a collection of Feedbacks and Feedback base classes.

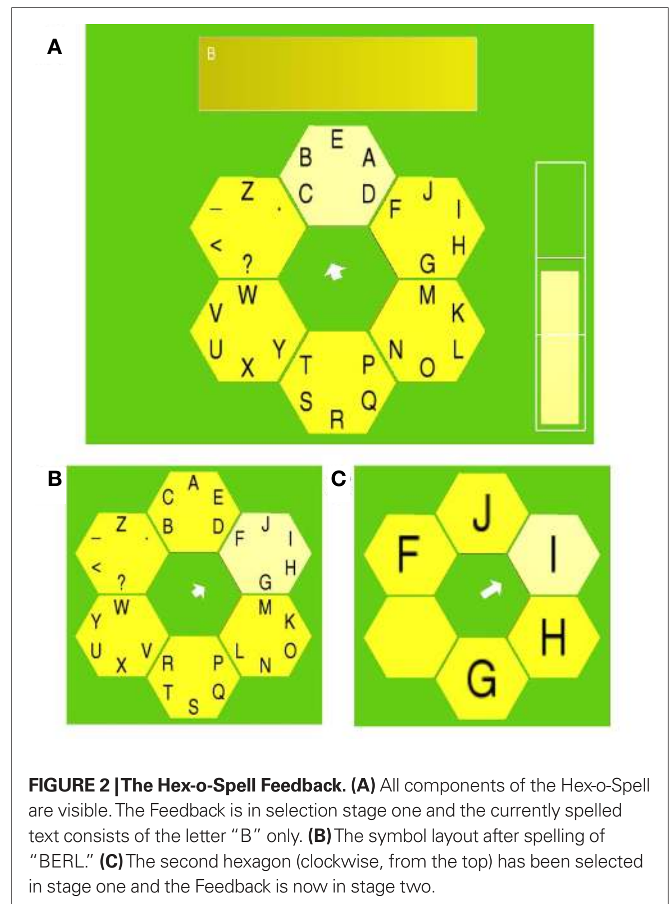


FIGURE 2 | The Hex-o-Spell Feedback. (A) All components of the Hex-o-Spell are visible. The Feedback is in selection stage one and the currently spelled text consists of the letter "B" only. **(B)** The symbol layout after spelling of "BERL." **(C)** The second hexagon (clockwise, from the top) has been selected in stage one and the Feedback is now in stage two.

4 SELECTED FEEDBACKS

Pyff allows for the rapid implementation of one's own paradigms, but it also comes equipped with a variety of ready-to-use paradigms. In the following sections, we will present a few examples.

4.1 HEX-o-SPELL FOR CONTINUOUS INPUT SIGNALS

The Hex-o-Spell is a text-entry device that is operated via timing-based changes of a continuous control signal (Müller and Blankertz, 2006; Williamson et al., 2009). These properties render it a suitable paradigm for BCI experiments in which brain-state discriminating strategies are employed that also have a fine temporal resolution.

The structure of the Hex-o-Spell Feedback with all its visual components is shown in **Figure 2A**. The main visual elements are an arrow that is surrounded by an array of six hexagons in the center of the screen, a large text board that displays the spelled text, and a bar of varying height that indicates the current value of the control signal. The hexagons surrounding the central arrow contain the symbols that are used for spelling.

The actual selection of a symbol is a two stage process and involves the subject controlling the orientation and length of the arrow. How exactly the parameters of the arrow are manipulated by the subject is explained in the next paragraph, while the remainder of this paragraph outlines the symbol selection process. In the first stage each hexagon contains five symbols. The subject has to select the hexagon that contains the desired symbol by making the arrow point to it and then confirming the choice by making the arrow grow until it reaches the hexagon. After this confirmation, the symbol content of the selected hexagon is distributed over the entire hexagon array, such that each hexagon now contains maximally one symbol only. There is always one hexagon that contains no symbol, which, in conjunction with the delete symbol "<," represents a practically unlimited *undo* option. The location of individual symbols in stage two reflects the positions they had in the single hexagon in stage one (compare **Figures 2B,C**). Now the subject has to position the arrow such that it points to the hexagon that contains the desired symbol and, again, confirm the selection by making the arrow grow until the respective hexagon is reached. After the symbol has been selected, the spelled text in the text board is updated accordingly. The Hex-o-Spell Feedback now returns to stage one,

i.e., the hexagons again show their original symbol content and the symbol selection process can begin anew. All major events, including for example on- and offsets of transition animations between stages, symbol selection, and GUI interaction (play, pause, stop) are accompanied by sending an event-specific integer code to the parallel port of the machine that runs the Hex-o-Spell Feedback. These codes can be incorporated in the marker structure of the EEG recording software and used for later analysis.

In order to operate the Hex-o-Spell symbol selection mechanism, the subject has to control the behavior of the arrow. The arrow is always in one of three distinct states: (1) clockwise rotation, (2) no rotation, and (3) no rotation and growth, i.e., increase in length until a certain maximum length. Upon returning from state (3) to state (2), the arrow shrinks back to default length. The states of the arrow are directly linked to the control signal from the Feedback Controller, which is required to be in the range between −1 and 1. Two thresholds, t_1 and t_2 with $-1 < t_1 < t_2 < 1$, partition the control signal range in three disjunct regions and thereby allow switching of arrow states by altering the control signal strength. The two thresholds are visualized as part of the control signal bar and therefore provide the subject with feedback as to how much more they have to in-/decrease signal strength in order to achieve a certain arrow state. The thresholds, time constants that determine rotation and scaling speed as well as other parameters of the Feedback can be adjusted during the experiment to allow for further accommodation to the subject.

So far all ingredients that are essential for operating the Hex-o-Spell Feedback have been explained. Additionally, our implementation includes mechanisms that speed up the spelling of words considerably by exploiting certain statistical properties of natural language. We achieve this by making those symbols easier accessible that are more likely to be selected next. In stage one the arrow starts always pointing to the hexagon containing the most probable next symbol. Additionally the positions of letters within each hexagon in stage one are arranged so that the arrow always starts pointing to the most probable symbol in stage two, followed by the second most probable, etc. The selection probability distribution model is updated after each new symbol. With these text-entry aids, the Hex-o-Spell Feedback allows for faster spelling rates of up to 7.6 symbols/min (Blankertz et al., 2007).

This Feedback requires the following packages to be installed: NumPy[4] and Panda3D[5]. Numpy is essential for the underlying geometrical computations (angles, position, etc.) and the handling of the language model data. Panda3D provides the necessary subroutines for rendering the visual feedback. Both packages are freely available for all major platforms from their respective websites.

4.2 ERP-BASED HEX-o-SPELL

The ERP-based Hex-o-Spell (Treder and Blankertz, 2010) is an adaptation of the standard Hex-o-Spell (see Section 4.1) utilizing event related potentials (ERPs) to select the symbols. The ERP is the brain response following an external or internal event. It involves early positive and negative components usually associated with sensory processing and later components reflecting cognitive processes. The main course of selecting symbols works in a two stage process as described in Section 4.1, but in the ERP-based variant the symbols are not selected by a rotating arrow, but by the ERPs caused by intensification of elements on the screen: Discs containing the symbols are intensified in random order and if the attended disc is intensified (**Figure 3**), a characteristic ERP is elicited. After 10 rounds of intensifications the BCI system has enough confidence to decide which of the presented discs the subject wanted to select.

Intensification is realized by up-sizing the disc including the symbol(s) by 62.5%. Each intensification is accompanied by a trigger which is sent to the EEG using the `send_parallel()` method of the Feedback base class. The triggers mark the exact points in time of the intensifications and are essential for the BCI system to do ERP-based BCI.

The ERP Hex-o-Spell Feedback class is derived from the `VisualP300` base class, provided by Pyff. This base class provides many useful methods to quickly write Visual ERP-based feedbacks or stimuli. For the actual drawing on the screen, Pygame[1] is used.

4.3 SSVEP-BASED HEX-o-SPELL

The brain responds to flickering visual stimuli by generating *steady state visual evoked potentials* (SSVEP, Herrmann, 2001) of corresponding frequency and its harmonics. Various studies have used this phenomena to characterize the spatial attention of a subject within a set of stimuli with differing frequencies. Cheng et al. (2002) showed a multi-class SSVEP-based BCI, where the subject had to select 1 of 10 numbers and two control buttons with mean information transfer rate of 27.15 bits/min. Müller-Putz and Pfurtscheller (2008) demonstrated that subjects where able to control a hand prosthesis using SSVEP.

In our SSVEP-based variant of the Hex-o-Spell, the selection of symbols is again a two stage process as described in Section 4.1. In the SSVEP-based variant there is no rotating arrow but the hexagons are all blinking. The blinking frequencies are pairwise different and fixed for each hexagon. Two different approaches can be used to select the hexagon containing the given letter: *overt*- and *covert attention*. In the overt attention case the subject is required to look at the hexagon with the letter, whereas in the case of covert attention, the subject must look at the dot in the middle and only concentrate on the desired hexagon. In both cases it is tested if the subject looks at the required spot by means of an eye tracker. If the subject aims the gaze somewhere else, the trial is stopped and, after showing an error message accompanied by a sound, it is restarted.

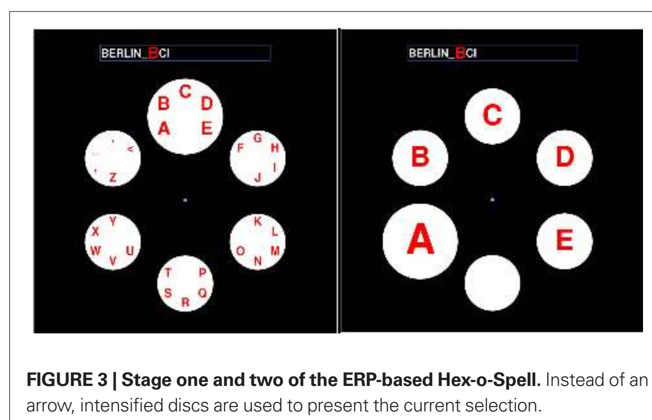


FIGURE 3 | Stage one and two of the ERP-based Hex-o-Spell. Instead of an arrow, intensified discs are used to present the current selection.

[4]Numpy homepage. URL http://numpy.scipy.org/

[5]Panda3d homepage. URL http://panda3d.org/

Before the experiment one might be interested in the optimal blinking frequencies for the subject. The SSVEP-based Hex-o-Spell, supports a training mode where one centered hexagon is blinking in different frequencies. After testing different frequencies in random order, the experimenter can chose the six frequencies with the highest signal to noise ration in the power spectrum of the recorded SSVEP at the flicker frequency of the stimulus or one of its harmonics. Since harmonics of different flicker frequencies can overlap, care must be taken so that the chosen frequencies can be well discriminated (Krusienski and Allison, 2008).

Since in SSVEP experiments appropriate frequencies of the flashing stimuli are very important, special attention was given to this matter. Thus, the Feedback was programmed using Vision Egg (Straw, 2008) which allows very accurate presentation of time-critical stimuli. We verified that the specified frequencies where exactly produced on the monitor by measuring them with an oscilloscope (Handyscope HS3 by Bitzer Digitaltechnik).

4.4 ERP-BASED PHOTO BROWSER

The photobrowser Feedback enables users to select images from a collection of images, e.g., for selecting images from an album for future presentation, or simply for enjoyment. Images are selected by detecting a specific ERP response of the EEG upon attended target images compared to non-attended non-target images, similar to the ERP-based Hex-o-Spell, see Section 4.2. The photobrowser uses the images themselves as stimuli, tilting and flashing some subgroup of the displayed images at regular intervals (see **Figure 4**). Users simply focus attention on a particular image, and the system randomly stimulates the whole collection. After several stimulation cycles, there is sufficient evidence from the ERP signals to detect the image that the user is responding to.

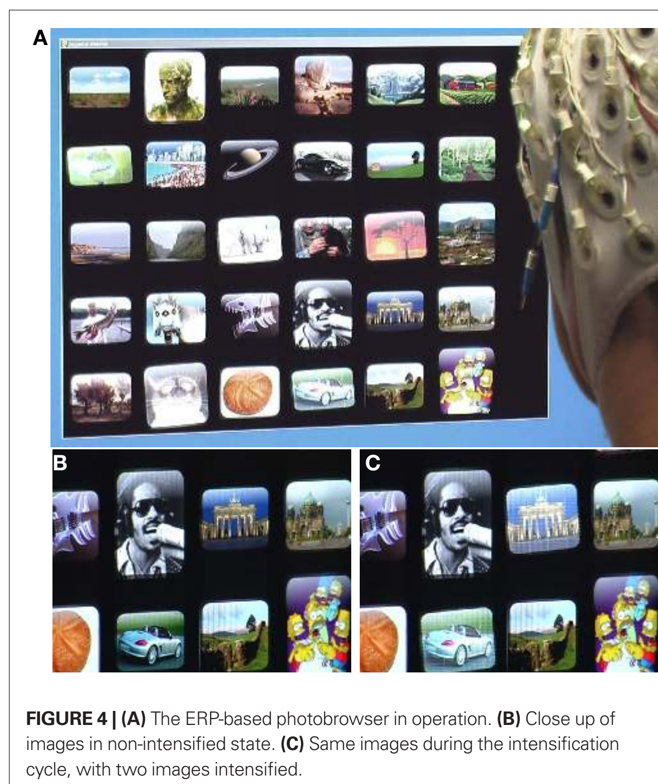


FIGURE 4 | (A) The ERP-based photobrowser in operation. **(B)** Close up of images in non-intensified state. **(C)** Same images during the intensification cycle, with two images intensified.

Using the Pyff framework for communication with a BCI system, the photobrowser implementation displays a 5×6 grid of photographs. The subject watches the whole grid, but attends to an image to be selected for further use. This can be done by direct gaze, but alternatively also covert attention mechanisms without direct gaze can be utilized by the subject. [For an investigation of the effect of target fixation see (Treder and Blankertz, 2010)]. During this time, the implementation stimulates subsets of images in a series of discrete steps, simultaneously sending a synchronization trigger into the EEG system, so that the timing of the stimulation can be matched to the onset of the visual stimulation. The photobrowser goes through a series of stimulation cycles, where subsets of the photographs are highlighted. After a sufficient number of these cycles, enough evidence is built up to determine the photograph the user is interested in selecting.

Stimulated images flash white, with a subtle white grid superimposed, and are rotated and scaled at the same time. This gives the visual impression of the images suddenly glowing and popping. The specific set of the visual parameters used (e.g., scale factor, tilt factor, flash duration, flash color) can all be configured to maximize the ERP response. The timing of these stimulations is also completely adjustable via the Pyff interface. The browser uses an immediate onset followed by an exponential decay for all of the stimulation parameters (for example, at the instant of stimulation the image turns completely white and then returns to its normal luminosity according to an exponential schedule).

The feedback uses PyOpenGL[3] and Pygame[1] for rapid display updates and high-quality image transformation. OpenGL provides the hardware transformation and alpha blending required for acceptable performance. NumPy[4] is used in the optimization routines to optimize the group of images stimulated on each cycle. Pyff provides access to the parallel ports for hard synchronization between the visual stimuli and the EEG recordings. Even in this time-critical application, Python with these libraries has good enough timing to allow the use of the ERP paradigm.

4.5 GOALKEEPER

The Goalkeeper Feedback is intended to be used for rapid-response BCIs (e.g., Ramsey et al., 2009). In a rapid-response BCI, subjects are forced to alter their brain states in response to a cue as fast as possible in order to achieve a given task. The major aim of such experiments is to increase the bit-rates of BCI systems.

The main components of this Feedback are a ball and a keeper bar (see **Figure 5**). The ball starts at the top of the screen and then descends automatically with a predefined velocity while the keeper is controlled by the subject. The task of the subject is to alter the keeper position via the BCI such that the keeper catches the ball. The powerbar at the bottom of the screen visualizes the classifier output and thus gives immediate feedback to the subject, thereby helping them to perform the task more successfully.

Each trial starts with an animation (two hemispheres approaching each other) that is intended to make the beginning of the actual trial (i.e., the descent of the ball) predictable for the subject. After the animation, the ball will choose one of two directions at random (i.e., left or right) and start to descend. Since subjects normally need some time to adapt their brain states according to the direction of ball movement, the keeper is not controllable (i.e., the classifier output is not used) in the first period of the descent (e.g., the first 300 ms).

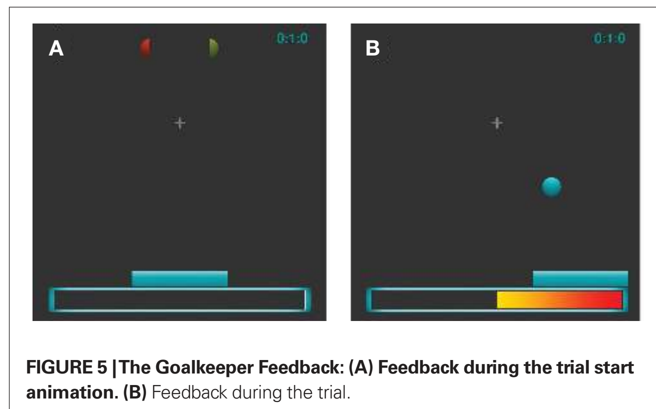


FIGURE 5 | The Goalkeeper Feedback: (A) Feedback during the trial start animation. (B) Feedback during the trial.

There are three main positions for the keeper: middle (initial position), left, and right. If the classifier threshold on either side is reached, the keeper will change its position to the respective side. This is typically realized as a non revertible jump according to the goalkeeper metaphor, but optionally a different behavior can be chosen. The velocity of the ball can be gradually increased during the experiment in order to force the subjects to speed up their response.

In addition to a number of general variables which can be used to define the time course (e.g., the time of a trial or the start animation) or the layout (e.g., the size of the visual components of the Feedback), there are also several main settings governing the work flow of the Feedback: (1) The powerbar can either show the direct classifier output or integrate the classifier output over time in order to smooth rapid fluctuations. (2) The keeper can either be set to perform exactly one move, or the Feedback can allow for a return of the keeper back to other positions. (3) The position change of the keeper can either be realized as a rapid jump or a smooth movement with a fixed duration. (4) Two additional control signals can optionally be visualized in the start animation by color changes between green and red of the two hemispheres. This option is inspired by the observation that a high pre-stimulus amplitude of sensorimotor rhythms (SMR) promotes better feedback performance in the subsequent trial (Maeder et al., 2010). (5) The trial can be prolonged if the keeper is still in the initial position (i.e., the subject did not yet reach the threshold for either side).

The implementation of the Feedback is done in a subclass of the `MainLoop_Feedback` base class provided by Pyff and using Pygame[1] and the Python image library (PIL)[6] for presentation.

4.6 OTHER FEEDBACKS AND STIMULI

While the previous selection of paradigms had a focus on brain–computer interface research, Pyff also ships with a growing number that are widely used in neuroscience, neuroergonomics, and psychophysics. The following list gives a few examples.

d2 test A computerized version of the d2 test (Brickenkamp, 1972), a Psychological pen and paper test to assess concentration- and performance ability of a subject. The complete listing of the application is in Section 13.

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

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

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

4.7 SUPPORT FOR SPECIAL HARDWARE

Since Python can utilize existing libraries (e.g., C-libraries, dlls). It is easy to use special hardware within Pyff. Pyff already provides a Python module for the IntelliGaze eye tracker by Alea Technologies and the g.STIMbox by g.tec. Other modules will follow.

5 USING Pyff

So far, we stressed the point that implementing an experiment using Pyff is fast and easy, but what exactly needs to be done? Facing the task of implementing a paradigm one has three options: First, check if a similar solution already exists in Pyff that can be used for the given experiment with minor modifications. If it is a standard experiment in BCI or psychology, chances are high that it is already included in Pyff. Second, if there is no application available matching the given requirements, a new Feedback has to be written. In this case one has to check if one of the given base classes match the task or functionality of the paradigm (e.g., is it a P300 task?, is it written using Pygame?, etc.). If so, the base class can be used to develop the feedback or stimulus application which will drastically reduce the code and thus the time required to create the Feedback. Third, if everything has to be written from scratch and no base class seems to fit, one should write the code in a way that is well structured and reusable. A base class may be distilled from the application to reduce the amount of code to be written the next time a similar task appears. In the second and third case, the authors would ideally send their code to us in order to include it in the Pyff framework. Thus the collection of stimuli and base classes would grow, making the first case more probable over time.

5.1 EASE OF USE

We have successfully used Pyff for teaching and experiments in our group since 2008. Our experience shows that researchers and students from various backgrounds quickly learn how to utilize Pyff to get their experiments done. We discuss three cases exemplary: In the early stages of Pyff, we asked a student to re-implement a given Matlab-Feedback in Pyff, to test if Pyff is feasible for our needs. We used Matlab for our experiments back then and we wanted to test if our new framework is capable of substituting the Matlab solutions. The paradigm was a Cursor-Arrow task, a standard BCI experiment. Given only the framework and documentation, the student completed the task within a few days without any questions. The resulting Feedback looked identical to the Matlab version but ran much smoother. Shortly later we wanted to compare the effort needed to implement a Feedback in Matlab and with our framework. We asked a student who was proficient in Matlab and

[6]Pil homepage. URL http://www.pythonware.com/products/pil/

Python to implement a Feedback in Matlab and our framework. The paradigm was quite simple: a ball is falling from the top of the screen, and the users tasks was to "catch" the ball with a bar on the bottom of the screen, which can only be moved to the left or right. The student implemented both Feedbacks within days without any questions regarding Pyff or Matlab. After the programming he told us that he had no problems with any of the two implementations. Since the given task was relatively simple he could use Matlab's plotting primitives to draw the Feedback on screen which was easier than with Pygame, which he used to draw the primitives on the screen, where he had to read the documentation first. He reported however that a more complex paradigm would also have required much more effort for the Matlab solution and only a little more for the Pyff version. In a third test we wanted a rather complex paradigm and see how well our framework copes with the requirements. The idea was to simulate a liquid floating on a plane which can tilt in any direction. The plane has three or more corners and the user's task is to tilt the plane in a way that the liquid floats to a designated corner. The simulation included a realistic physical model of liquid motion. A Ph.D. student implemented the Feedback in Pyff, the physical model was developed in C and then SWIG (Beazley, 1996) was used to wrap the C-code in Python. He implemented the Feedback without any questions regarding the framework. The result is an impressive simulation, which runs very smoothly within our framework.

These three examples indicate that students and researchers without experience with Pyff have no difficulties implementing paradigms in a short time. The tasks varied from simple Matlab to Python comparisons to significantly more complex applications. This is consistent with our more than 2 years of experience using Pyff in our lab, an environment where co-workers and students use it regularly for teaching and experiments.

6 CONCLUSION

The Pythonic Feedback Framework provides a platform for writing high-quality stimulus and feedback applications with minimal effort, even for non-computer scientists.

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

By providing an interface utilizing well known standard protocols and formats, this framework should be adaptable to most existing neuro systems. Such a unified framework creates the unique opportunity of exchanging neuro feedback applications and stimuli between different groups, even if individual systems are used for signal acquisition, processing, and classification.

At the time of writing Pyff has been used in four labs and several publications (Ramsey et al., 2009; Acqualagna et al., 2010; Höhne et al., 2010; Maeder et al., 2010; Schmidt et al., 2010; Treder and Blankertz, 2010; Venthur et al., 2010).

We consider Pyff as stable software which is actively maintained. To date, new versions are released every few months that include bug fixes and new features. We plan to continue development of Pyff as our group uses it to conduct many experiments and other groups are starting to adopt it. As such, we realize that backward compatibility of the API is very important and we work hard to avoid breakage of existing experiments when making changes.

Pyff has a homepage[7], where users can download current as well as older versions of Pyff. The homepage also provides online documentation for each Pyff version and a link to Pyff's mailing list for developers and users and to Pyff's repository.

Pyff is free software and available under the terms of the GNU general public license (GPL)[8]. Pyff currently requires Python 2.6[9] and PyQT version 4[10] or later. Some Feedbacks may require additional Python modules. Pyff runs under all major operating systems including Linux, Mac, and Windows.

ACKNOWLEDGMENTS

This work was partly supported by grants of the Bundesministerium für Bildung und Forschung (BMBF) (FKZ 01IB001A, 01GQ0850) and by the FP7-ICT Programme of the European Community, under the PASCAL2 Network of Excellence, ICT-216886. This publication only reflects the authors' views. Funding agencies are not liable for any use that may be made of the information contained herein. We would also like to thank the reviewers, who helped to substantially improve the manuscript.

[7]Pyff homepage. URL http://bbci.de/pyff/

[8]GNU general public license. URL http://www.gnu.org/copyleft/gpl.html

[9]Python homepage. URL http://python.org/

[10]Pyqt homepage. URL http://www.riverbankcomputing.co.uk/software/pyqt/

REFERENCES

Acqualagna, L., Treder, M. S., Schreuder, M., and Blankertz, B. (2010). A novel brain–computer interface based on the rapid serial visual presentation paradigm, *Proc. 32nd Ann. Int. IEEE EMBS Conf. 2686–2689*.

Beazley, D. M. (1996). Swig: an easy to use tool for integrating scripting languages with c and c++, *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996, USENIX Association, Berkeley, CA, USA, 1996,* pp. 15–15. URL http://www.swig.org/

Blankertz, B., Krauledat, M., Dornhege, G., Williamson, J., Murray-Smith, R., and Müller, K.-R. (2007). A note on brain actuated spelling with the Berlin brain-computer interface. *Lect. Notes Comput. Sci.* 4555, 759.

Brainard, D. (1997). The psychophysics toolbox. *Spat. Vis.* 10, 433–436.

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

Brickenkamp, R., and Zillmer, E. (1998). *D2 Test of Attention*. Göttingen, Germany: Hogrefe and Huber.

Brouwer, A.-M., and van Erp J. B. F. (2010). A tactile p300 brain-computer interface. *Front. Neuroprosthetics* 4:19. doi: 10.3389/fnins.2010.00019.

Brüderle, D., Müller, E., Davison, A., Muller, E., Schemmel, J., and Meier, K. (2009). Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. *Front. Neuroinformatics* 3:17. doi: 10.3389/neuro.11.017.2009.

Cheng, M., Gao, X., Gao, S., and Xu, D. (2002). Design and implementation of a brain-computer interface with high transfer rates. *IEEE Trans. Biomed. Eng.* 49, 1181–1186.

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

Drewes, R., Zou, Q., and Goodman, P. (2009). Brainlab: a Python toolkit to aid in the design, simulation, and analysis of spiking neural networks with the NeoCortical simulator. *Front. Neuroinformatics* 3:16. doi: 10.3389/neuro.11.016.2009.

Geller, A., Schleifer, I., Sederberg, P., Jacobs, J., Kahana, M. (2007). PyEPL: a cross-platform experiment-programming library. *Behav. Res. Methods* 39, 950–958.

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

Herrmann, C. S. (2001). Human EEG responses to 1-100 hz flicker: resonance phenomena in visual cortex and their potential correlation to cognitive phenomena. *Exp. Brain Res.* 137, 346–353.

Höhne, J., Schreuder, M., Blankertz, B., and Tangermann, M. (2010). Two-dimensional auditory P300 speller with predictive text system, *Proc. 32nd Ann. Int. IEEE EMBS Conf. 4185–4188*.

Ince, R., Petersen, R., Swan, D., and Panzeri, S. (2009). Python for information theoretic analysis of neural data. *Front. Neuroinformatics* 3:4. doi: 10.3389/neuro.11.004.2009.

Jurica, P., and Van Leeuwen, C. (2009). OMPC: an open-source MATLAB®-to-Python compiler. *Front. Neuroinformatics* 3:5. doi: 10.3389/neuro.11.005.2009.

Krusienski, D. J., and Allison, B. Z. (2008). Harmonic coupling of steady-state visual evoked potentials. *Conf. Proc. IEEE Eng. Med. Biol. Soc.* 2008, 5037–5040.

Lutz, M. (2006). *Programming Python*. Sebastopol, CA, USA: O'Reilly Media, Inc.

Mackworth, N. (1948). The breakdown of vigilance during prolonged visual search. *Q. J. Exp. Psychol.* 1, 6–21.

Maeder, C., Sannelli, C., Haufe, S., Lemm, S., and Blankertz, B. (2010). Effect of prestimulus SMR amplitude on BCI performance. Poster at the TOBI Workshop 'Integrating Brain-Computer Interfaces with Conventional Assistive Technology' in Graz.

Müller-Putz, G., and Pfurtscheller, G. (2008). Control of an electrical prosthesis with an SSVEP-based BCI. *IEEE Trans. Biomed. Eng.* 55, 361–364.

Müller, K.-R., and Blankertz, B. (2006). Toward noninvasive brain-computer interfaces. *IEEE Signal Process Mag.* 23, 125–128.

Pecevski, D., Natschläger, T., and Schuch, K. (2009). PCSIM: a parallel simulation environment for neural circuits fully integrated with python. *Front. Neuroinformatics* 3:11. doi: 10.3389/neuro.11.011.2009.

Peirce, J. W. (2007). Psychopy–psychophysics software in python. *J. Neurosci. Methods* 162, 8–13.

Pfurtscheller, G., Leeb, R., Keinrath, C., Friedman, D., Neuper, C., Guger, C., and Slater, M. (2006). Walking from thought. *Brain Res.* 1071, 145–152.

Prechelt, L. (2000). An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Comput.* 33, 23–29.

Ramsey, L., Tangermann, M., Haufe, S., and Blankertz, B. (2009). Practicing fast-decision BCI using a "goalkeeper" paradigm. *BMC Neurosci.* 10 (Suppl. 1), P69.

Schalk, G., McFarland, D., Hinterberger, T., Birbaumer, N., and Wolpaw, J. (2004). Bci2000: a general-purpose brain-computer interface (BCI) system. *IEEE Trans. Biomed. Eng.* 51, 1034–1043.

Schmidt, N. M., Blankertz, B., and Treder, M. S. (2010). Alpha-modulation induced by covert attention shifts as a new input modality for EEG-based BCIs. *Proc. 2010 IEEE Conf. Syst. Man Cybernet.*, (in press).

Schreiner, T. (2008). *Development and Application of a Python Scripting Framework for bci2000.* Master's thesis. Universität Tübingen, Tübingen.

Schreuder, M., Blankertz, B., and Tangermann, M. (2010). A new auditory multi-class brain-computer interface paradigm: spatial hearing as an informative cue. *PLoS One* 5, e9813. doi: 10.1371/journal.pone.0009813.

Spacek, M., Blanche, T., and Swindale, N. (2008). Python for large-scale electrophysiology. *Front. Neuroinformatics* 2:9. doi: 10.3389/neuro.11.009.2008.

Strangman, G., Zhang, Q., and Zeffiro, T. (2009). Near-infrared neuroimaging with NinPy. *Front. Neuroinformatics* 3:12. doi: 10.3389/neuro.11.012.2009.

Straw, A. D. (2008). Vision Egg: an open-source library for realtime visual stimulus generation. *Front. Neuroinformatics* 2:4. doi: 10.3389/neuro.11/004.2008.

Tanenbaum, A. S. (2001). *Modern Operating Systems.* Upper Saddle River, NJ, USA: Prentice Hall PTR.

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

Venthur, B., Blankertz, B., Gugler, M. F., and Curio, G. (2010). Novel applications of BCI technology: psychophysiological optimization of working conditions in industry, in: *Proc. 2010 IEEE Conf. Syst. Man Cybernet.* in press.

Williamson, J., Murray-Smith, R., Blankertz, B., Krauledat, M., and Müller, K.-R. (2009). Designing for uncertain, asymmetric control: interaction design for brain-computer interfaces. *Int. J. Hum. Comput. Stud.* 67, 827–841.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Received: 14 June 2010; accepted: 02 October 2010; published online: 02 December 2010.

*Citation: Venthur B, Scholler S, Williamson J, Dähne S, Treder MS, Kramarek MT, Müller K and Blankertz B (2010) Pyff – a Pythonic framework for feedback applications and stimulus presentation in neuroscience. Front. Neurosci. **4**:179. doi: 10.3389/fnins.2010.00179*

This article was submitted to Frontiers in Neuroscience Methods, a specialty of Frontiers in Neuroscience.

Copyright © 2010 Venthur, Scholler, Williamson, Dähne, Treder, Kramarek, Müller and Blankertz. This is an open-access article subject to an exclusive license agreement between the authors and the Frontiers Research Foundation, which permits unrestricted use, distribution, and reproduction in any medium, provided the original authors and source are credited.

APPENDIX

Throughout this appendix we use the term *feedback* synonymous for feedback and stimulus application as both concepts make technically little difference in the context of the framework: feedbacks are usually stimuli with some sort of closed loop from the subject to the application.

7 COMPONENTS OF THE FRAMEWORK

Pyff consists of four major parts: the *Feedback Controller*, the *graphical user interface* (GUI), a set of *Feedback baseclasses* and a set of *Feedbacks*. **Figure 6** shows an overview of Pyff.

Pyff communicates with the rest of the world via a standardized communication protocol using UDP and XML (Section 11).

7.1 THE FEEDBACK CONTROLLER

The Feedback Controller manages the communication between the Feedback and the world outside Pyff. It is responsible for spawning new Feedback processes, starting, pausing and stopping them as well as inspecting and manipulating their internal variables.

Once started, the Feedback Controller acts like a server, waiting for incoming signals from the network. Incoming signals are encoded in XML (Section 11), the Feedback Controller converts them to message objects and – depending on the kind of signal (Section 8) – either processes them directly or passes them to the currently running Feedback.

The Feedback Controller starts new Feedbacks by spawning new Feedback processes. This has the advantage that a crashing or otherwise misbehaving Feedback application does not directly affect the Feedback Controller as it would do if we had used Threads. Since this framework also aims to be a workbench for easy Feedback development, misbehaving Feedbacks can be quite common, especially in the beginning of the Feedback development. The communication between two different processes however, is a bit more complicated than between two threads since two processes do not share the same address space. In our framework we solved this issue with an Inter-Process Communication mechanism using sockets to pass message objects back and forth between the Feedback Controller and a running Feedback (Section 9).

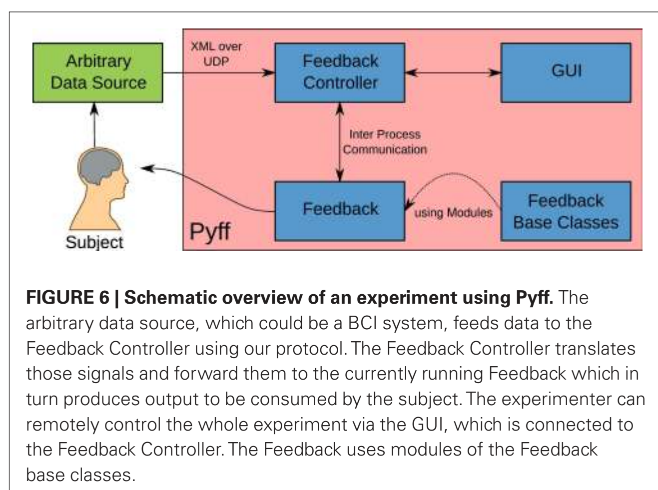


FIGURE 6 | Schematic overview of an experiment using Pyff. The arbitrary data source, which could be a BCI system, feeds data to the Feedback Controller using our protocol. The Feedback Controller translates those signals and forward them to the currently running Feedback which in turn produces output to be consumed by the subject. The experimenter can remotely control the whole experiment via the GUI, which is connected to the Feedback Controller. The Feedback uses modules of the Feedback base classes.

7.2 THE GRAPHICAL USER INTERFACE

The framework provides a graphical user interface (GUI) which enables easy access to the Feedback Controller's main functions and the Feedback's variables. The GUI communicates with the Feedback Controller like the system connected to Pyff does: via XML over UDP. Therefore, the GUI does not have to run on the same machine as the Feedback Controller does, which is particularly useful for experiments where the experimenter and subject are in different rooms.

Figure 7 shows a screen shot of the GUI. The drop down menu presents a list of available Feedbacks. On the right of this list are various buttons for Init, Play, Pause, etc. The main part of the GUI is occupied by a table which presents the name, value, and type of variables belonging to the currently running Feedback. The table is editable so that the user can modify any Feedback variable as desired and send it back to the Feedback where the change is directly applied. The possibility to inspect and manipulate the running Feedback's object variables gives a great deal of flexibility for experimenters to explore and try new settings.

7.3 THE FEEDBACK BASE CLASSES

The Feedback base class is the base class of all Feedbacks and the interface to the Feedback Controller's plugin system.

As mentioned in Section 7.1, the Feedback Controller is able to load, control, and unload Feedbacks dynamically. Feedbacks can be very different in complexity, functionality, and purpose. To work with different Feedbacks properly, the Feedback Controller relies on a small set of methods that every Feedback has to provide. The Feedback base class declares these methods and thus guarantees that from the Feedback Controller's point of view, a well defined set of operations is supported by every Feedback.

The methods are: `on_init`, `on_play`, `on_pause`, `on_stop`, `on_quit`, `on_interaction_event`, and `on_control_event`. The Feedback Controller calls them whenever it received a respective signal. For example: when the Feedback Controller receives a control signal, it calls the `on_control_event` method of the Feedback. For a complete overview which events cause which method calls in the Feedback, see Section 8.

Most of the above methods are parameterless, they are just called to let the Feedback know that a certain event just happened. Exceptions are `on_control_event` and `on_interaction_event`. These events carry an argument *data*, which is a dictionary containing all variables which where sent to the Feedback Controller and should be set in the Feedback. For convenience, a Feedback does not have to implement these two methods just to get the data, the Feedback Controller takes care that the data is already set in the Feedback before the respective method is called in the Feedback.

The Feedback base class also has a method `send_parallel(data)` which the can be used to send data to the parallel port of its host machine and this is a typical way to set markers into the acquired EEG.

For convenience the Feedback base class also provides a logger attribute. The logger attribute is a logger object of Python's standard logging facility. A logger is basically like the print statement with a severity (the loglevel) attached. A global severity threshold can be set which stops all log-messages below that level from appearing on the console or in the logfile. The global loglevel and the output format is configured by the Feedback Controller: The

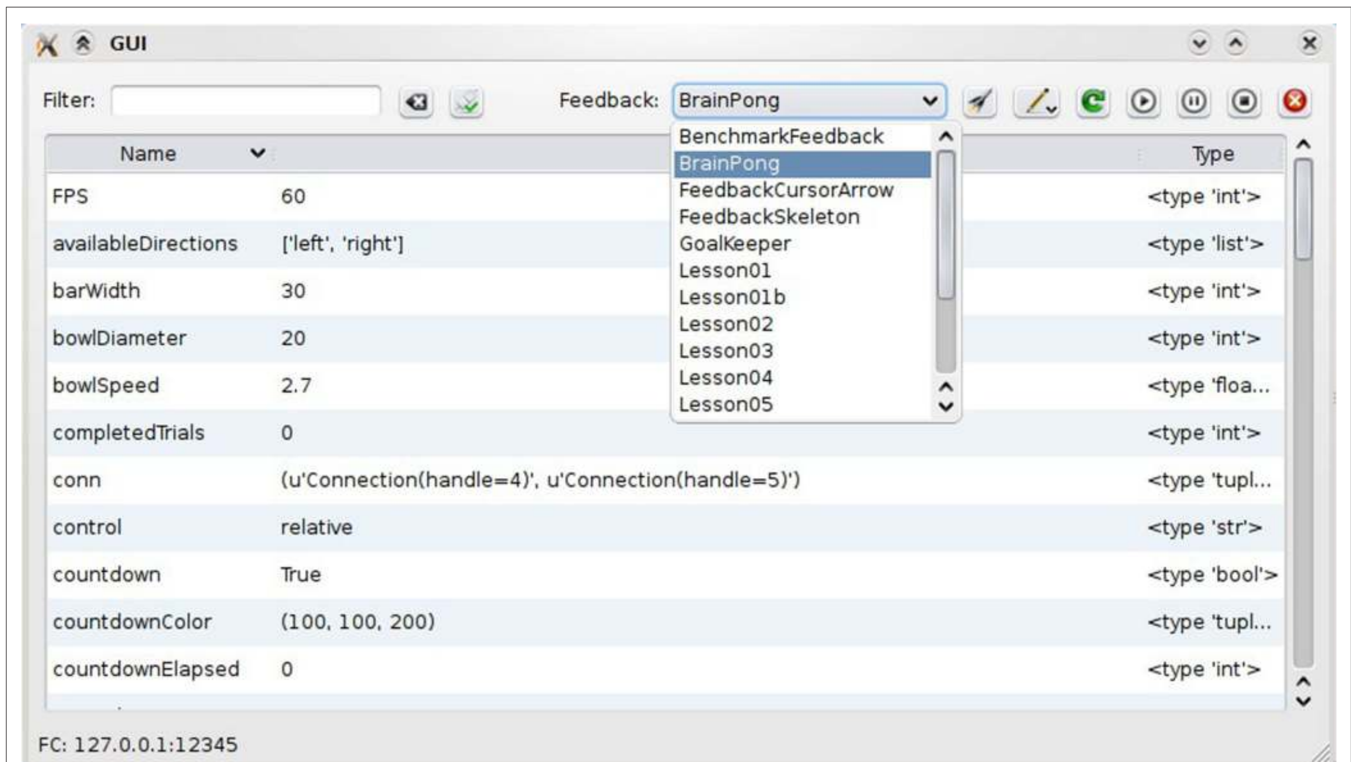


FIGURE 7 | The graphical user interface of the framework. Within the GUI the experimenter can select a Feedback, start, pause, and stop it and he can inspect and manipulate the Feedback's internal variables.

loglevel has a default value but it can be modified via the Feedback Controller's command line option. A Feedback programmer can use this logger directly without any extra initialization of the logger whatsoever.

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

This object-oriented approach drastically simplifies the development of Feedbacks, since common code of similar Feedbacks can be moved out of the actual Feedbacks into a common base class. **Figure 8** shows an example: on the left side, two similar Feedbacks share a fair amount of code. Both Feedbacks work with a main loop. They have a set of main loop related methods and also more Feedback specific related methods. Implementing a third Feedback with a main loop means the programmer will likely copy the main loop from one of the existing Feedbacks into his new Feedback. This a bad approach for several reasons, the most important one being that a bug in the main loop related logic will probably appear in all of the Feedbacks. Due to code duplication, it will then have to be fixed every feedback using the logic. A solution is to extract the main loop related logic into a base class and implement it there (**Figure 8**, right). The Feedbacks can derive from this class, inherit the logic, and only need to implement the Feedback specific part. The code of the Feedbacks is much shorter, less error prone, and main loop related bugs can be fixed in a single file.

7.4 FEEDBACKS

Besides providing a platform for easy development of feedback applications, Pyff also provides a set of useful and ready-to-use Feedbacks. The list of those Feedbacks will grow as we and hopefully other groups will develop more of them.

8 CONTROL- AND INTERACTION SIGNALS

Pyff receives two different kinds of signals: *Control Signals* and *Interaction Signals*. The difference between the two is that Interaction Signals can contain variables and commands and Control Signals only variables. The commands in the Interaction Signal are for the Feedback Controller to control the behavior of the Feedback, the variables contained in both the Control- and Interaction Signals are to be set in the currently running Feedback. Technically both variants are equivalent but practically they are used for different things: The variables in the Interaction Signal are used to modify the Feedback's object variables and therefore influence the Feedback's behavior. The variables in the Control Signal are the actual data coming from a data source like EEG. The distinction is useful for example when a Feedback needs to do something on every arriving block of data. It can then take advantage of the fact that incoming Control- and Interaction Signals trigger `on_control_event` respective `on_interaction_event` methods in the Feedback. The programmer of the Feedback can focus on the handling of the data by implementing `on_control_event` and does not have to worry if the variables sent by the signal are actual data or just object variables of the Feedback.

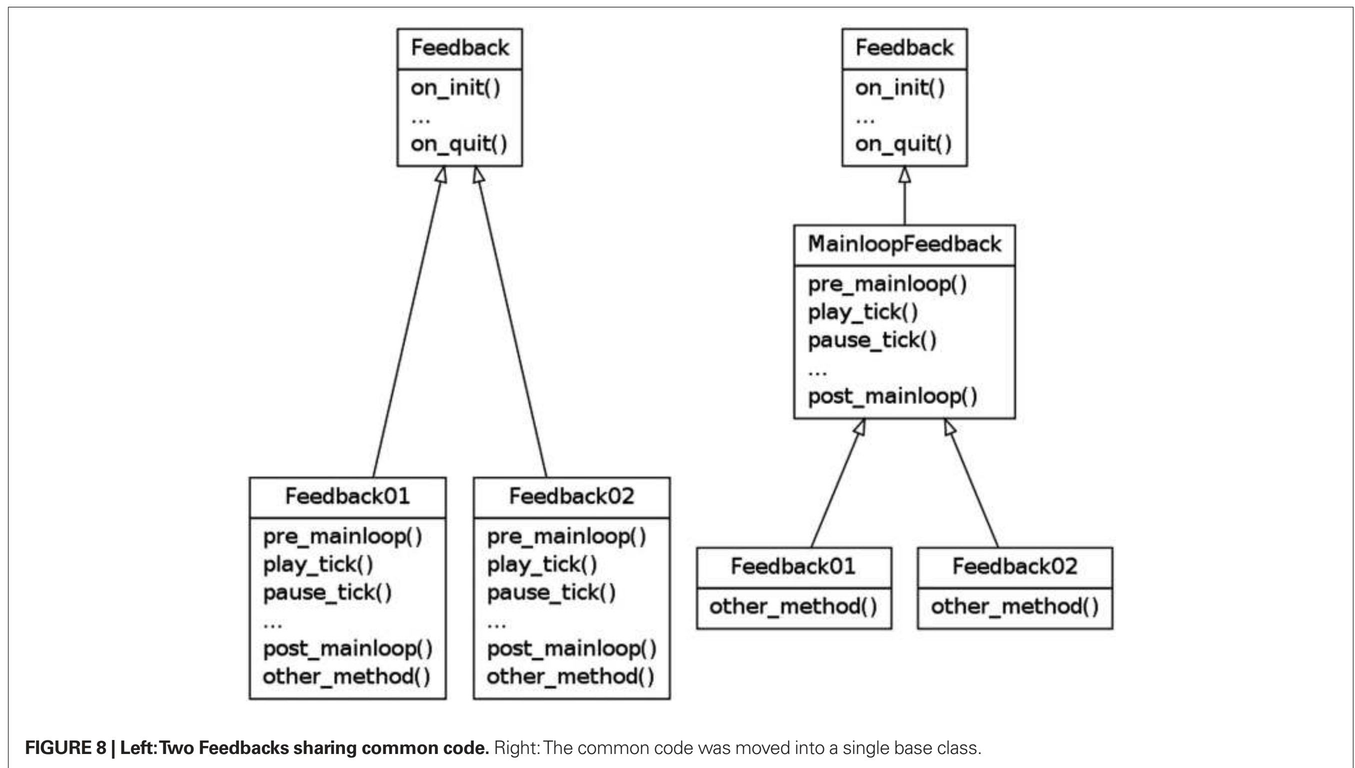


FIGURE 8 | Left: Two Feedbacks sharing common code. Right: The common code was moved into a single base class.

Table 1 shows the complete list of available commands in Control signals and **Figure 9** shows a sequence diagram illustrating the interaction between the GUI, the Feedback Controller, and the Feedback. All signals coming from the GUI are Interaction Signals, the signals coming from the BCI system are Control signals. Once the GUI is started, it tries to automatically connect to a running Feedback Controller on the machine where the GUI is running. If that fails (e.g., since the Feedback Controller is running on a different machine on the network) the experimenter can connect it manually by providing the host name or IP address of the target machine. Upon a successful connection with the Feedback Controller, the Feedback Controller replies with a list of available Feedbacks which is then shown in a drop down menu in the GUI. The experimenter can now select a Feedback and click the Init Button which sends the appropriate signal to the Feedback Controller telling it to load the desired Feedback. The Feedback Controller loads the Feedback and requests the Feedback's object variables which it sends back to the GUI. The GUI then shows them in a table where the experimenter can inspect and manipulate them. Within the GUI, the experimenter can also Start, Pause, Stop, and Quit the Feedback.

Table 1 | Available commands.

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

9 INTER-PROCESS COMMUNICATION

When writing applications, there are usually two possible options to achieve concurrency: The first one is to use threads, the second is to use processes. Processes are heavyweight compared to threads. Processes have their own address space and some form of inter-process communication (IPC) is needed to allow processes to interact with each other. Threads on the other side are lightweight, there can be one ore more thread in the same process and threads of the same process share their address space. The implication of the shared address space is, that a modification of a variable within one thread is immediately visible to all other threads of this process. A complete description of processes, threads, and inter-process communication can be found in (Tanenbaum, 2001). Since there is no need for a IPC when using threads, threads are often more desirable than processes and a sufficient solution for many common concurrent applications. In Python however things are a bit different. First, and most importantly: in Python two threads of a process do not run truly concurrently, but sequentially in a time-sliced manner. The reason is Python's global interpreter lock (GIL) which ensures that only one Python thread can execute in the interpreter at once (Lutz, 2006)[11]. The consequence is that Python programs cannot make use of multi processors using Python's threads. In order to use real concurrency one has to use processes, effectively sidestepping the GIL. The second important point is that many important graphical Python libraries like pygame or PyQT need to run in the main (first) thread of the Python process. In a threaded version

[11]Thread state and the global interpreter lock. URL http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock

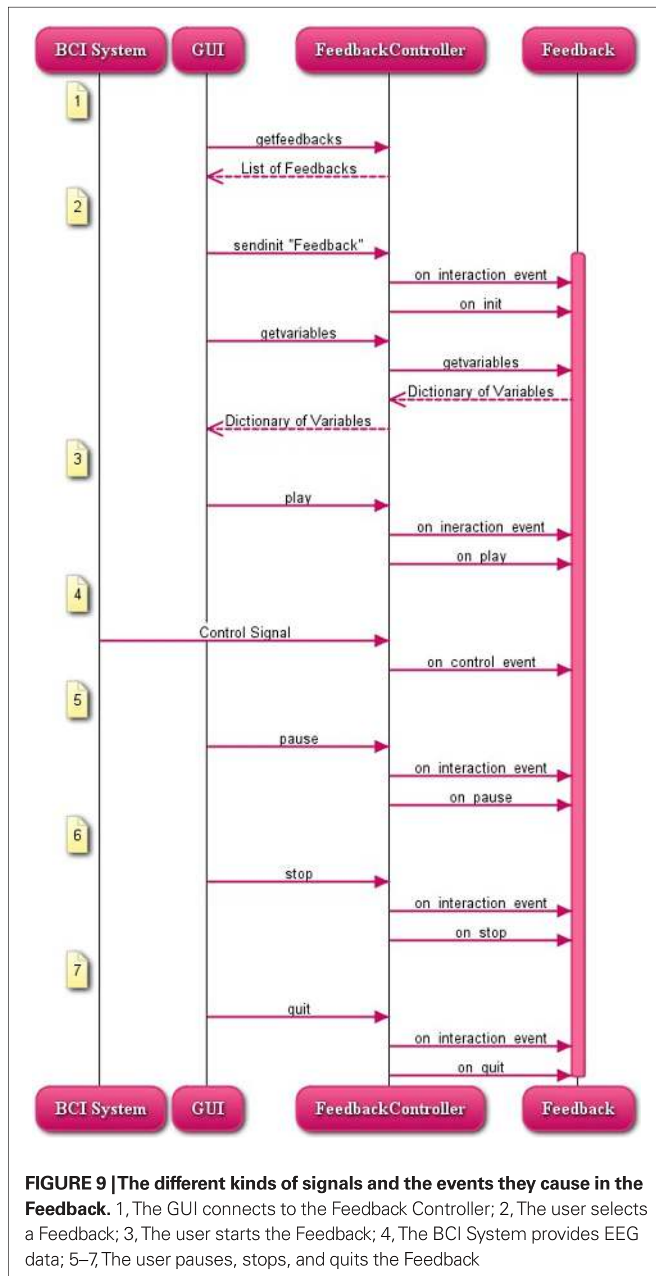


FIGURE 9 | The different kinds of signals and the events they cause in the Feedback. 1, The GUI connects to the Feedback Controller; 2, The user selects a Feedback; 3, The user starts the Feedback; 4, The BCI System provides EEG data; 5–7, The user pauses, stops, and quits the Feedback

of Pyff this would make things a lot more complicated, since the Feedback Controller as a server runs during the whole lifetime of the experiment, while Feedbacks (where those libraries are used) usually get loaded, unloaded, started and stopped many times during an experiment. The natural way to implement this in a threaded way would be to run the Feedback Controller in the main thread and let it spawn Feedback threads as needed. Doing it the other way round by reserving the main thread for the Feedback and letting the Feedback Controller insert Feedbacks into the first thread on demand would be a lot more complicated and error prone.

For that reasons we decided to use processes instead of threads. The feedback applications have much more resources running in their own process (even on their own processor) while keeping the programming of Pyff's logic simple and easy to maintain.

9.1 INTER-PROCESS COMMUNICATION IN Pyff

We decided to use a socket based IPC since it works on all major platforms. The basic idea is that two processes establish a TCP connection to communicate. If a peer wants to send a message to the other one, it uses his end of the connection-called socket-to send the message. The message is then sent to the other end of the connection, where the second peer can read it from his socket (Lutz, 2006).

The messages itself are message objects which are serialized before being sent and unserialized on the receiving end. Serialization and Unserialization is done using Python's pickle library.

10 REMOTELY CONTROLLING Pyff

To make the framework as portable as possible to other existing systems, it was critical to design an interface generic enough to support a wide range of programming languages and operating systems. This loose coupling is achieved through our choice to use the UDP for the transport of the data through the network and XML for the encoding of the signal.

User datagram protocol was chosen because it is a well known and established standard network protocol and because virtually every programming language supports it. UDP-clients and -servers are fairly easy to implement and it is possible to send arbitrary data over UDP. For similar reasons, XML was chosen for the encoding of the signals: XML is an established standard for exchanging data and libraries for parsing and writing XML are available for most common programming languages.

Once the FeedbackController starts, it listens on UDP port 12345 for incoming control- and interaction signals. Clients can remotely control the FeedbackController as they would using the GUI by sending signals to that address. The signals have to be wrapped in XML. Documentation for Pyff's XML scheme is given in Section 11.

11 Pyff's XML SCHEME

To wrap the content of the control- and interaction-signals, we chose XML. XML is a well known, accepted standard and was specifically designed for tasks like this, where arbitrary data needs to be exchanged between different systems[12].

The Feedback Controller accepts a defined set of *commands* and arbitrary *variables* which should be set in the Feedback. A command is a simple string and has no parameters, they are not arbitrary Python statements but a well defined set of commands that the Feedback Controller executes, like loading a Feedback or starting it. A variable is defined as a triple (*type*,*name*,*value*), where *type* is the data type, *name* the name of the variable and *value* the actual value of the variable.

For this purpose, we created an XML scheme capable of containing variables and commands. The following describes the version 1.0 of the scheme.

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

[12]Extensible markup language 1.0, design and goals. URL http://www.w3.org/TR/REC-xml/#sec-origin-goals

```
<?xml version=”1.0”?>
<bci-signal version=”1.0”>
  <interaction-signal>
    <command value=”start”/>
    <s name=”string” value=”foo”/>
    <f name=”float” value=”0.69”/>
    <list name=”list”>
      </i value=”1”/>
      </i value=”2”/>
      </i value=”3”/>
  </list>
 </interaction-signal>
</bci-signal>
```

Commands

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

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

where *commandname* is a member of the set of supported commands. Supported commands are: *getfeedbacks*, *play*, *pause*, *stop*, *quit*, *sendinit*, *getvariables*. Only one command is allowed per interaction signal. The meaning of the commands is explained in Section 8. An interaction signal can contain a command and several variables.

Variables

Variables are allowed in interaction- and control signals. A variable is represented by the triple (*type*,*name*,*value*) and has the following form in XML:

```
<type name=”varname” value=”varvalue”>
```

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

```
<integer name=”foo” value=”42”>
```

The XML scheme supports all variable types supported by Python, **Table 2** shows a complete listing. Sometimes there is more than one way to express the type of a variable. For example, a boolean can be expressed in this XML scheme via `<boolean… />, <bool… />` and `<b… />`. All alternatives are equivalent and exist merely for convenience.

Table 2 | Data types supported by our XML scheme.

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

Nested Variables

Some variables can contain other variables, like Lists (Vector, Array) or Dictionaries (Hashes, Hash Tables). The following example shows the XML representation of a List called *mylist* containing three Integers:

```
<list name=”mylist”>
  <i value=”1”/>
  <i value=”2”/>
  <i value=”3”/>
</list>
```

Nested variables can also contain other nested variables. The following example shows a list containing two integers and a list which also contains two integers:

```
<list name=”mylist2”>
  <i value=”1”/>
  <i value=”2”/>
  <list>
    <i value=”3”/>
    <i value=”4”/>
  </list>
</list>
```

No restrictions are put upon the depth of the nested variables.

Special care has to be taken using Dictionaries. Dictionaries are not simply collections of variables but collections of mappings from Strings to a Variables. The mappings are expressed through Tuples (String, Variable) and consequently, a Dictionary is expressed as a collection of Tuples in our XML scheme. The following example shows a dictionary *mydict* containing three mappings (foo, 1), (bar, 2) and (baz, 3):

```
<dict name=”mydict”>
  <tuple>
    <s value=”foo”/>
    <i value=”1”/>
  </tuple>
  <tuple>
    <s value=”bar”/>
    <i value=”2”/>
  </tuple>
  <tuple>
    <s value=”baz”/>
    <i value=”3”/>
  </tuple>
</dict>
```

This XML scheme provides a well defined and clean interface for other systems to communicate with Pyff: we provide a set of commands to control the Feedback Controller and the Feedback and a way to read and write the Feedback's variables. The whole protocol is operating system and programming language independent. It is simple enough to grasp it without much effort but generic enough to send any kind of data to the Feedback.

12 DOCUMENTATION AND EXAMPLES

Part of the framework is a complete documentation of the system and its interfaces. All modules, classes, and methods are also extensively documented in form of Python docstrings. Those docstrings

are used by integrated development environments (IDEs) and tools like Python's pydoc to generate help and documentation from the source files.

The framework also provides Tutorials explaining every major aspect of Feedback development with example Feedbacks.

13 EXAMPLE FEEDBACK

This section contains a complete listing of the TestD2 Feedback, a Feedback implementing a computer version of the classic d2 test of attention, developed by Brickenkamp and Zillmer (1998) as the paper-and-pencil test. The listing is complete and functionally identical with the TestD2 Feedback delivered with Pyff, however many blank lines, comments, and the copyright statement in the beginning of the file were removed make it shorter. The following subsections correspond to one or two methods of the TestD2 class. The complete listing of the TestD2 module is given at the end of this section.

In the paper-and-pencil version, the test consists of the letters *d* and *p*, which are printed on a sheet of paper in 14 lines with 47 letters per line. Each letter has between one and four vertical lines above or below. The subject's task is to cross out all occurrences of the letter *d* with two lines (target) on the current line as fast and correct as possible. A *d* with more or less than two lines or a *p* (non-targets) must not be crossed out. The subject usually has 20 s to process a line. After those 20 s the experimenter gives a signal and the subject has to process the next line. The number of mistakes (erroneously crossed out non d2s or not crossed out d2s) can be used afterward to quantify the attention of the subject.

Our computerized version of TestD2 differs from the paper-and-pencil variant in the following ways: The stimuli are not presented in rows which have to be processed in a given short amount of time, but are presented one by one on the screen. For each stimulus the subject has to decide whether it is a target or a non-target by pressing the according key on the keyboard. **Figure 10** shows screenshots of the running Feedback. In the paper-and-pencil version, the subject has to process 14 rows of 47 symbols and has 20 s per row. In the standard configuration of our Feedback we present $14 \cdot 20$ symbols and give the user a maximum of $14 \cdot (20/47)$ s. Of course this makes the results of our version not directly comparable with the paper-and-pencil variant, we think however that it still makes a good demonstration on how to implement a sophisticated feedback application with our framework.

13.1 INITIALIZATION OF THE FEEDBACK

The TestD2 (Listing 1) class is derived from the PygameFeedback Class. PygameFeedback takes care of proper initialization and shutdown of Pygame before and after the Feedback runs. It also provides helpful methods and members which make dealing with Pygame much easier.

The `init` method of TestD2 sets various variables controlling the behavior of the Feedback. In the first line of `init` the `init` method of the parent class is called. This is necessary since the parent's `init` declares variables needed to make PygameFeedback's methods work. `caption` sets the caption text of the Pygame window, `random_seed` sets the seed for the random number generator. This is important to make experiments reproducible when using random numbers. `number_of_symbols`, `seconds_per_symbol`, and `targets_persent` set how many symbols are presented at most, how much time the subject has process one symbol and the percentage of target symbols of all symbols presented. The number of symbols and seconds per symbol are used to calculate the duration of the experiment. The three numbers are taken from the standard Test D2 condition where a subject has 14 lines with 47 symbols per line and 20 s time per line. `color`, `backgroundColor`, and `fontheight` control the look of the Feedback, `color` is for the color of the symbols, `backgroundColor` for the color of the background and `fontheight` the height of the font in pixels. `key_target` and `key_nontarget` are the keys on the keyboard the user has to click if s/he wants to mark a target or a non-target.

All variables declared in this method are immediately visible in the GUI after the Feedback is loaded. The experimenter can modify them as he likes and set them in the Feedback.

13.2 PRE- AND POSTMAINLOOP

Pre- and postmainloop (Listings 2 and 3) are invoked respectively, immediately before after the mainloop of the Feedback. Since the mainloop of a Feedback can be invoked several times during it's lifetime, variables which need to be initialized before each run should be set in `pre_mainloop` and evaluations of the run should be done in `post_mainloop`.

In `pre_mainloop` we call the parent's `pre_mainloop` which initializes Pygame. Then we generate the sequence of stimuli (see Section 13.4), the graphics for the stimuli (see Section 13.5). The variables `current_index` represents the current position in the list of stimuli, `e1` and `e2` are the number of errors of omission and

FIGURE 10 | Screenshots of the stimuli presented by the Feedback. A stimulus is presented until the subject presses one of two available buttons to decide if he sees a d2 or not. After the key is pressed the next stimulus is presented immediately.

errors of commission. The `QUIT` event is then scheduled. This event will appear in Pygame's event queue after the given time and marks the regular end of the Feedback. The current time is measured and the first stimulus presented and the mainloop starts.

In `post_mainloop` we measure the time needed to run the experiment by calling `clock.tick()` a second time which returns the time in milliseconds since the last call. Then we call the parent's `post_mainloop` which cleanly shuts down Pygame. After that, various results of the experiment are calculated and printed out.

13.3 TICK

The `tick` method (Listing 4) is called repeatedly during the mainloop by the MainloopFeedback which is a parent Feedback of the PygameFeedback. Usually the `tick` method of PygameFeedback limits the Framerate by calling `self.clock.tick(self.FPS)` and processes Pygame's event queue. This is the desired behavior in most Pygame Feedbacks containing a mainloop. In this case, however the Feedback is not paced by small timesteps, but by the keypresses of the subject, so we overwrite the parent's `tick` to omit the call of `self.clock.tick` and call `self.wait_for_pygame_event` instead of `self.process_pygame_events`. This means tick waits at this call until the user pressed a key (or another Pygame event appears in the event queue). Then the key is checked if it is one of the two available keys and if the correct key was pressed. If not, the corresponding error accumulator is increased by one. Then the position in the list of stimuli is increased by one. If the index reached the end of the list, the Feedback is stopped, otherwise the next stimulus is presented (Listing 7).

13.4 GENERATING THE D2 LIST

The method `generate_d2list` (Listing 5) sets the seed for the random number generator to make the results reproducible and generates the list of targets and non-targets. It uses the `targets_percent` attribute to obey the correct ratio of targets and non-targets in the list. The method also ensures that a symbol never appears twice or more in a row.

13.5 GENERATING THE SYMBOLS

The method `generate_symbols` (Listing 6) generates the images, or surfaces in Pygame lingo, for the various symbols and stores them in an object attribute so the Feedback can later use them directly when painting them on the screen. The Symbols are generated as a combination of a letter and one of two possible lines above and below the letter. The method first generates the images for the letters then the images for the lines and glues them together in the last loop.

14 USING PYFF

After downloading and extracting Pyff from[7] there are two directories: src and tools. To start Pyff go into src and run FeedbackController.py. This will start the Feedback Controller and the GUI. The control elements of the GUI are explained in **Figure 11**.

When the GUI has started, it will automatically ask the Feedback Controller for available Feedbacks and will populate the Feedback Selector. The experimenter selects the desired Feedback from there and loads it by pushing the Init button. The Feedback Controller will now load the Feedback and send the Feedback's object variables back to the GUI where they are presented in the Table. The experimenter can now modify variables as needed and apply the changes in the Feedback by pushing the Send button.

Up to this point, the Feedback has just been loaded and initialized but not yet started. To start, pause and stop the Feedback the experimenter pushes the Play-, Pause- and Stop buttons. A Feedback can be started, stopped and paused unlimited times during the Feedback's lifetime.

To quit the Feedback the experimenter uses the quit button. Quitting the Feedback will stop and unload the Feedback from the Feedback Controller. Selecting and loading a Feedback while another one is already running, will stop and quit the running Feedback and load the new one afterwards.

14.1 FEEDBACK CONTROLLER'S OPTIONS

The Feedback Controller supports various options. To get a complete listing start the Feedback Controller with the `Δhelp` parameter.

Feedback Controller's Loglevel

The `--loglevel=LEVEL` parameter sets the loglevel of the Feedback Controller and it's components. Accepted levels in increasing order are: notset, debug, info, warning, error and critical. Setting the loglevel will cause the Feedback Controller to output certain log messages of the given level and higher.

Feedback's Loglevel

When developing Feedback applications it is important to have a logger dedicated for Feedbacks. Pyff provides a logger for Feedbacks which is configurable separately from the Feedback Controller's loglevel. The `--fb-loglevel=LEVEL` parameter controls the loglevel of the Feedback's logger. It accepts the same values as the `--loglevel` parameter.

Additional Feedback Directory

When developing Feedback applications which are not to be included in the Pyff framework, it might be desirable to locate them in a different directory than Pyff's default directory for Feedbacks.

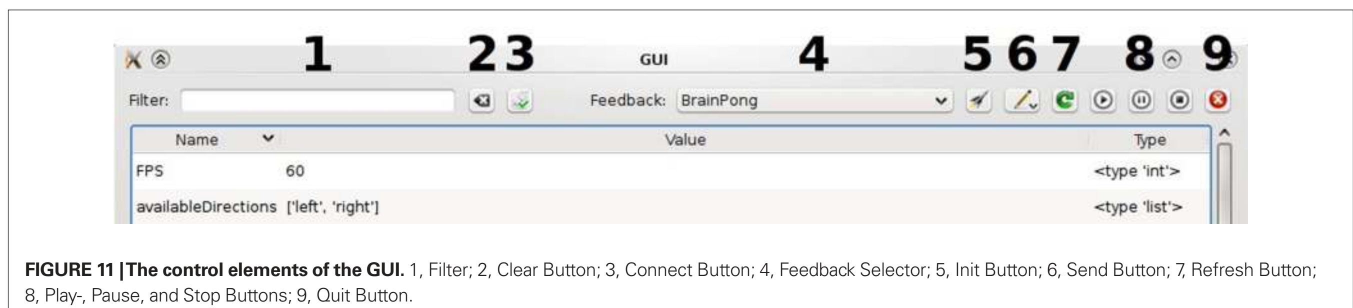


FIGURE 11 | The control elements of the GUI. 1, Filter; 2, Clear Button; 3, Connect Button; 4, Feedback Selector; 5, Init Button; 6, Send Button; 7, Refresh Button; 8, Play-, Pause, and Stop Buttons; 9, Quit Button.

The `--additional-feedback-path=PATH` parameter supports this, by causing the Feedback Controller to additionally search for Feedbacks under the given path.

Starting Without GUI

When running the Feedback Controller in an automated experiment setup where the interaction signals are emitted from a different source than the GUI, the experimenter can start the Feedback Controller without the GUI via the `--nogui` parameter.

Configuring The Parallel Port

On different computers the parallel port sometimes has a different address than the default. The `--port=PORTNUM` parameter configures the parallel port address (in hexadecimal) the Feedback Controller tries to use.

Listing 1: First par of the file

```
1  import random
2  import pygame
3  from FeedbackBase.PygameFeedback
   import  PygameFeedback
4
5  TARGETS=['d11', 'd20', 'd02']
6  NON_TARGETS=['d10', 'd01', 'd21', 'd12', 'd22',
7                'p10', 'p01', 'p11', 'p20', 'p02',
                  'p21', 'p12', 'p22']
8
9  class TestD2(PygameFeedback):
10
11     def init(self):
12         PygameFeedback.init(self)
13         self.caption="Test D2"
14         self.random_seed=1234
15         self.number_of_symbols=47 * 14
16         self.seconds_per_symbol=20 / 47.
17         self.targets_percent=45.45
18         self.color=[0, 0, 0]
19         self.backgroundColor=[127, 127, 127]
20         self.fontheight=200
21         self.key_target="f"
22         self.key_nontarget="j"
```

Listing 2: pre_mainloop

```
1  def pre_mainloop(self):
2      PygameFeedback.pre_mainloop(self)
3      self.generate_d2list()
4      self.generate_symbols()
5      self.current_index=0
6      self.e1=0
7      self.e2=0
8      pygame.time.set_timer(pygame.QUIT,
        self. number_of_symbols * self.
          seconds_per_symbol * 1000)
9      self.clock.tick()
10     self.present_stimulus()
```

Listing 3: post_mainloop

```
1  def post_mainloop(self):
2      elapsed_seconds=self.clock.tick() / 1000.
3      PygameFeedback.post_mainloop(self)
4      tn=self.current_index + 1
5      error=self.e1 + self.e2
6      error_rate=100. * error / tn
7      correctly_processed = tn - error
8      cp=correctly_processed - self.e2
9      rt_avg=elapsed_seconds / tn
10     print "Results:"
11     print " ========"
12     print
13     print "Processed symbols: %i of %i" % (tn,
         self.number_of_symbols)
14     print "Elapsed time: %f sec" %
         elapsed_seconds
15     print "Correctly processed symbols: %i" %
        (correctly_processed)
16     print "Percentage of Errors: %f" %
        (error_rate)
17     print "Errors: %i" % error
18     print "… errors of omission: %i" % self.e1
19     print "… errors of commission: %i" % self.e2
20     print "Concentration Performance: %i" % cp
21     print "Average reaction time: %f sec" %
        rt_avg
```

Listing 4: tick

```
1  def tick(self):
2      self.wait_for_pygame_event()
3      if self.keypressed:
4          key=self.lastkey_unicode
5          self.keypressed=False
6          if key not in (self.key_target,
               self. key_nontarget):
7              return
8          else:
9              if key == self.key_nontarget
10             and self.d2list[self. current_index]
                 in TARGETS:
11                 self.e1+= 1
12             elif key == self.key_target
13             and self.d2list[self. current_index]
                 in NON_TARGETS:
14                 self.e2 += 1
15             else:
16                 pass
17             self.current_index += 1
18             if self.current_index > self.
   number_of_symbols - 1:
19 self.on_stop()
20 else:
21 self.present_stimulus()
```

Listing 5: generate_d2list

```
1  def generate_d2list(self):
2      random.seed(self.random_seed)
3      targets=int(round(self.number_of_symbols *
         self.targets_percent / 100))
4      non_targets=int(self.number_of_symbols
         - targets)
5      l=[random.choice(TARGETS) for i in
          range(targets)] + \
6      [random.choice(NON_TARGETS) for i in
```

```
        range(non_targets)]
 7      random.shuffle(l)
 8      for i in range(len(l) - 1):
 9          if l[i]== l[i + 1]:
10              pool=TARGETS if l[i] in TARGETS
                  else NON_TARGETS
11              new=random.choice(pool)
12              while new == l[i + 1]:
13                  new=random.choice(pool)
14              l[i]=new
15      self.d2list=l
```

Listing: 6 generate_symbols

```
 1  def generate_symbols(self):
 2      linewidth=self.fontheight / 11
 3      font=pygame.font.Font(None, self.fontheight)
 4      surface_d=font.render("d", True, self.color)
 5      surface_p=font.render("p", True, self.color)
 6      width, height=surface_d.get_size()
 7      surface_l1=pygame.Surface((width, height),
          pygame.SRCALPHA)
 8      surface_l2=pygame.Surface((width, height),
          pygame.SRCALPHA)
 9      pygame.draw.line(surface_l1, self.color,
10      (width / 2, height / 10),
11      (width / 2, height - height / 10),
           linewidth)
12      pygame.draw.line(surface_l2, self.color,
13      (width / 3, height / 10),
14      (width / 3, height - height / 10), linewidth)
15      pygame.draw.line(surface_l2, self.color,
16      (2 * width / 3, height / 10),
17      (2 * width / 3, height - height / 10),
           linewidth)
18      self.symbol= {}
19      for symbol in TARGETS + NON_TARGETS:
20          surface = pygame.Surface((width, height
              * 3), pygame.SRCALPHA)
21          letter= surface_d if symbol[0]= = 'd'
              else surface_p
22          surface.blit(letter, (0, height))
23          if symbol[1]= = '1':
24          surface.blit(surface_l1, (0, 0))
25          elif symbol[1]= = '2':
26          surface.blit(surface_l2, (0, 0))
27          if symbol[2]= = '1':
28          surface.blit(surface_l1, (0, 2 * height))
29          elif symbol[2]= = '2':
30          surface.blit(surface_l2, (0, 2 * height))
31          self.symbol[symbol]=surface
```

Listing 7: present_stimulus

```
1  def present_stimulus(self):
2      self.screen.fill(self.backgroundColor)
3      symbol=self.d2list[self.current_index]
4      self.screen.blit(self.symbol[symbol],
5      self.symbol[symbol].get_rect(center=self.
         screen.get_rect().center))
6      pygame.display.flip()
```

Lisiting 8: End of file

```
1  if __name__ == "__main__":
2      fb=TestD2()
3      fb.on_init()
4      fb.on_play()
```