Pythonic Feedback Framework Introduction and Manual

Bastian Venthur
mail@venthur.de

2008-01-07

This document provides an introduction to the pythonic feedback framework as well as a manual and some examples how to write your own BCI feedbacks using this framework.

Contents

1 Introduction

This document provides an introduction to the pythonic feedback framework as well as a manual and some examples how to write your own BCI feedbacks using this framework. It is not meant to be an introduction into Python. If you're new to Python but already have programming experience with an other language, [8] gives a very good starting point to learn Python. The documentation (especially the Library Reference) available on [7] is essential when programming with Python.

If you don't have any programming experience, you should read [6].

1.1 BCI Setup

Figure 1 shows the setup of a BCI experiment. The subject is sitting in front of a monitor (if the feedback is of visual nature) and is wearing a EEG cap, collecting brain signals and submitting them to the data acquisition and signal processing units. The signal is processed and translated into Matlab code and sent to the feedback, where, depending on the feedback application, the feedback reacts on some way on the subject's EEG signals. The processed and translated EEG signal which is sent to the feedback, is called control signal.

The experimenter has the option to control certain variables of the feedback via the GUI which also sends singals to the feedback. Those signals are called interaction signals.

Currently everything is written in Matlab. Since many feedbacks are of a very graphic nature, may need to provide audible output and Matlab wasn't really designed for tasks like this, the BCI group wants to move away from Matlab towards Python when writing such feedbacks.

1.2 Pythonic Feedback Framework

The pythonic feedback framework tries to solve this task. It provides the Feedback Controller which acts like a server and collects the control- and interaction signals. It also features a plugin system which makes it fairly easy to write new feedbacks in Python. Figure 2 shows the Feedback Controller in the BCI setup.

The Feedback Controller transparently (no changes in the BCI setup are necessary) replaces the single feedback from the old setup and allows dynamic loading and un-loading of feedbacks through it's plugin system.

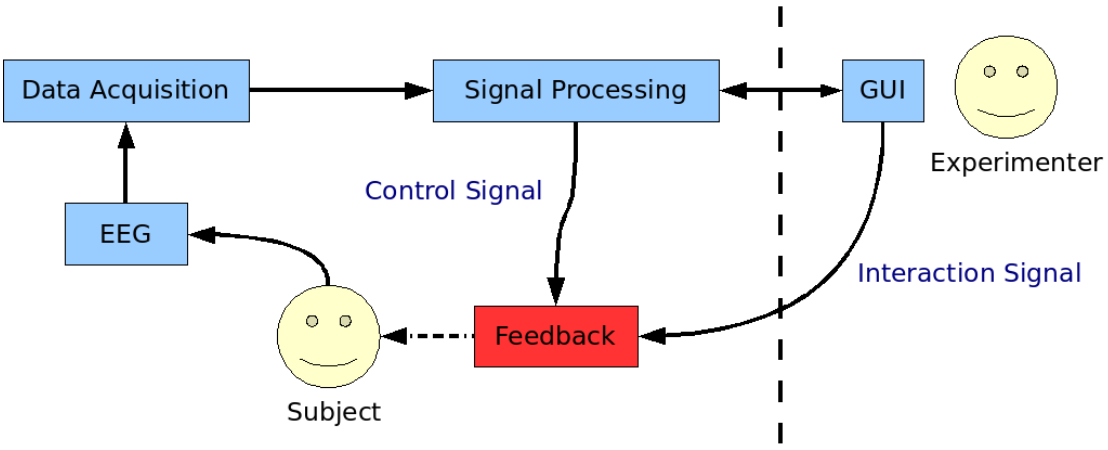

Figure 1: Setup of an BCI experiment

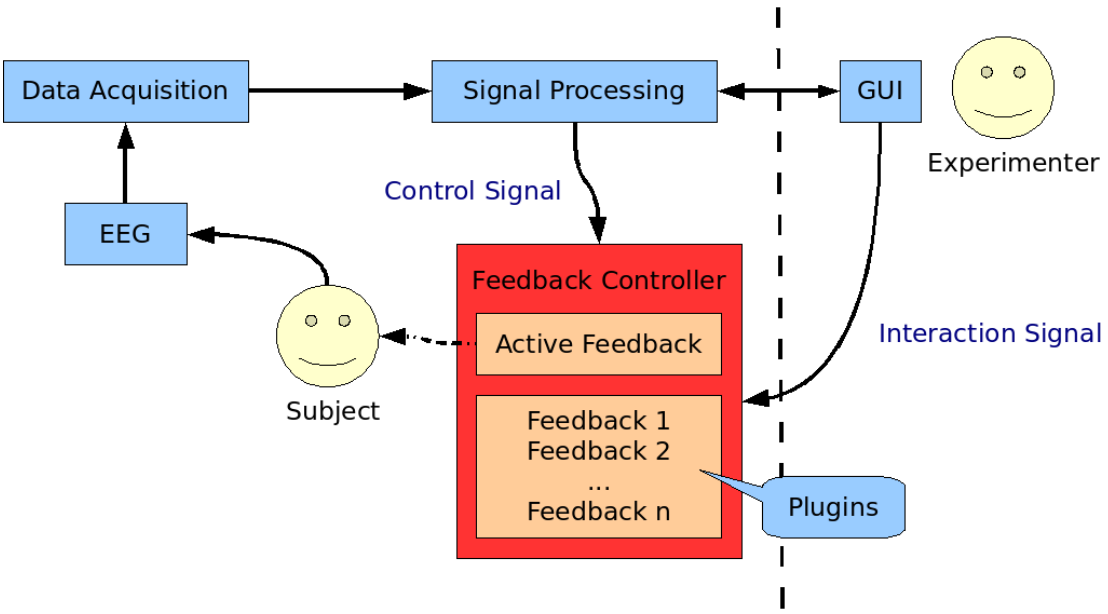

Figure 2: Setup of an BCI experiment with the Feedback Framework

2 Introduction to the Pythonic Feedback Framework

2.1 Structure of the Framework

The basic idea behind the Feedback framework is to have a Server sitting in the background collecting control- and interaction signals, translating them into Python datatypes and sending them to the currently active feedback. The server is called Feedback Controller. Once started, it is fully controllable via interaction signals. You can remotely load feedbacks, start, stop and pause them via the GUI.

When receiving control signals the Feedback Controller translates the data into Python and passes the data to the currently loaded feedback.

The Feedback Controller supports a plugin system which makes it very easy to create new feedbacks. Basicly all you have to do to create a new feedback is to subclass the Feedback base class and implement the functions as needed or create your own ones.

2.2 The Feedback Base Class

The Feedback base class is the interface to the Feedback Controller's plug-in system. By subclassing the Feedback class and putting it somewhere into the Feedbacks directory your feedback is already a valid and ready-to-use (although quite useless at this moment) feedback, available to the Feedback Controller.

The mode of operation of the feedbacks is event driven. Whenever the Feedback Controller receives a signal, it translates and analyzes it and calls the appropriate method of the feedback to notify it. When writing your own feedback, all you have to do is to implement some or all methods of the Feedback base class to react on those signals as needed.

The following methods are supported by the Feedback base class:

on_init(self) This method is called by the Feedback Controller after the feedback has been successfully loaded. Since the initialization of the feedback is triggered by an interaction signal, `on_interaction_event` is called right after `on_init` returned.

on_interaction_event(self, data) This method is called when the Feedback Controller received an interaction signal. data contains all the variables sent by the GUI. You don't have to use the data from here directly, see section 2.3 for details.

on_control_event(self, data) This method is called by the Feedback Controller after the Feedback Controller received a control signal. data contains the tuple containing the processed and translated EEG data. You don't have to use this variable, see section 2.5 for details.

on_play(self) If the Feedback Controller detects the Play command in an interaction signal, it calls this method just before it calls `on_interaction_event`.

on_pause(self) If the Feedback Controller detects the Pause command in an interaction signal, it calls this method just before it calls `on_interaction_event`.

on_quit(self) If the Feedback Controller detects the Stop command in an interaction signal, it calls this method just before it calls `on_interaction_event`.

send_parallel(self, data) This is the only method of the Feedback base class which actually provides functionality. You should not overwrite this method if you don't want to replace the underlying code. Use this method to send a byte to the parallel port of the machine running the feedback.

`on_play`, `on_pause` and `on_quit` are a convenient methods which are always called before `on_interaction_event`. You don't have to use them but then you will almost certainly have to inspect every interaction signal to check whether it contains the Start, Pause or Stop trigger if you want to react on them.

2.3 Interaction Signals

Everytime the Feedback Controller receives a valid interaction signal from the GUI, it translates it and calls the `on_interaction_event` method of the currently running feedback. We can distinguish between five different types of interaction signals:

Send With this signal the GUI sends various variable-value pairs to the feedback.

Send+Init Like the Send signal plus the command to initialize the feedback.

Start This signal tells the feedback to start its main routine (e.g. the trials).

Pause Tells the feedback to pause the run.

Stop Tells the feedback to quit.

If the Feedback Controller recognizes a generic interaction signal (Send) it just calls `on_interaction_event` of the feedback. If it additionally recognizes an Init, Start, Pause or Stop command in the signal, it also calls the appropriate `on_`-method of the feedback. In case of Init, it calls `on_init` before `on_interaction_event`, in every other case after. Table 1 shows the ordering of the method calls of the feedback triggered by the different interaction signals.

The Feedback controller also analyzes the data in the interaction signal, extracts the variables and it's values and puts them into the currently running feedback. See 3.4 for details.

2.4 Details About the Matlab-Python Translation

The unpacked control- and interaction signal is in fact code which is directly interpretable by Matlab. The control signal contains just a tuple of numbers representing the processed EEG signal, the interaction signal contains Matlab variable-assignments. Since the Matlab syntax and it's data types are not Python compatible, it is necessary to translate the variable names and Matlab's datatypes into Python.

GUI sends	Feedback receives
Send	`on_interaction_event(self, data)`

Send+Init	`on_quit(self)` (old feedback)
	`on_init(self)`
	`on_interaction_event(self, data)`

Start	`on_interaction_event(self, data)` `on_play(self)`
Pause	`on_interaction_event(self, data)` `on_pause(self)`
Stop	`on_interaction_event(self, data)` `on_quit(self)`

Table 1: Overview of the method calls triggered by the different types of interaction signals.

Translation of Variablenames The translation of Matlab's variable names is straightforward. The only important aspect is that if the Feedback Controller detects dots in the variable names it truncates the name after the last dot and takes the result as variable name.

If for example the Feedback Controller receives a variable `feedback_opt(1).type`, the resulting variable name in the feedback will be `_type` (note the underscore prefix, see section 3.4 for details). If the Feedback Controller receives a variable name without a dot, the whole name will be taken on if possible.

Translation of Datatypes Matlab supports many data types which have no equivalent in Python while it also shares a subset of types which have a equivalent in Python. Fortunately the Feedback Controller does not have to support all available types available in Matlab (although it would be possible). Only the most common ones like strings, integers, floats and lists are needed to communicate with the feedbacks. All of them are currently recognized. If the Feedback Controller receives an unknown type it will print out a warning and move on. If needed the Feedback Controller can be enhanced to recognize more types. The relevant method is `__parse_type` of the UdpDecoder class.

2.5 Control Signals

Whenever the Feedback controller receives a control signal, it translates it, extracts the data (a tuple containing a few numbers), puts it in the currently running feedback (see 3.4 for details) and calls the feedback's `on_control_event(self, data)` method.

3 Writing Your Own Feedbacks

The Feedbacks/Tutorial directory contains a few example feedbacks which will hopefully help you to understand how to develop your own ones. This section will explain them.

3.1 Naming Convention for the Module- and Class Name

In order to make your Feedback available to the Feedback Controller's plugin system, your feedback has to follow a certain naming convention. The rules are:

1. *The feedback has to be located somewhere in the Feedbacks directory.* It is possible to place it in an arbitrary deep subdirectory inside of the Feedbacks directory. There are no restrictions on the subdirectories names as long as they represent valid Python packages.

2. *The feedback's class name must match it's module name.* If the class name of your feedback is `FooBar` then it has to be written in the file `FooBar.py`.

When the experimenter wants to load a certain feedback, he sends a Send+Init signal which contains at least the assignment of the `type` variable. The variable holds a string which is interpreted in the following way:

- If the string contains no dots (the "." character), the string is interpreted as the module *and* class name of the feedback.

- If the string contains one or more dots, the string is split up and the last element represents the module/class name and the element(s) before the last element the package name.

Examples:

- If the Send+Init signal contains `type = Foo.baR.Baz`, the Feedback controller tries to load the Class `Baz` in the module `Baz` (the file `Baz.py`) in the Directory `Feedbacks/Foo/baR`.

- If the Send+Init signal just contains `type = MyFeedback`, the Feedback Controller searches for the class `MyFeedback` in the module `MyFeedback` located directly in the `Feedbacks` directory.

3.2 Subclassing the Feedback Class

Following the naming conventions of section 3.1 ensures that the Feedback Controller *finds* the plugin. In order to make it working properly as a valid plugin of the Feedback Controller, the feedback has to be a subclass of the `Feedback` base class.

Listing 1 shows a fully working Feedback without any functionality other than printing the feedback has been successfully loaded and quit. The lines 1-3 are the important ones in this lesson: You have to import the Feedback class from the Feedback package and subclass your Feedback (Lesson01 in this case) from it. You don't have to override the on_init and on_quit methods to make the feedback working if you would replace the lines 5-9 with a simple `pass` statement, the feedback would still be perfectly running without any functionality.

Listing 1: Trivial Feedback

```
1 from Feedback import Feedback
2
3 class Lesson01(Feedback):
4
5     def on_init(self):
6         print "Feedback successfully loaded."
7
8     def on_quit(self):
9         print "Feedback quit."
```

Overwriting `Feedback.__init__` Since the `__init__` method of the Feedback base class is already implemented (it sets up the feedbacks logger and opens the parallel port), you should not overwrite it blindly. If you want to overwrite it, make sure to call the base class' `__init__` method before anything else in this method. Listing 2 shows an example. The second parameter is a handle to the parallel port, the third parameter of the `__init__` method changes the prefix the feedback uses for variables which are automaticly set and updated by the feedback controller (see 3.4 for details). The parameter is optional and defaults to the underscore character if omitted.

Listing 2: Trivial Feedback with `__init__` overwritten

```
1 from Feedback import Feedback
2
3 class Lesson01b(Feedback):
4
5     def __init__(self, pp):
6         Feedback.__init__(self, pp, "foo-")
7         # Your own stuff goes here
8
9     def on_init(self):
10        print "Feedback successfully loaded."
11
12    def on_quit(self):
13        print "Feedback quit."
```

Since Feedback's `on_init` is guaranteed to be the next method called after `__init__` it should not be necessary to overwrite `__init__` in most cases. You should use `on_init` instead.

3.3 Reacting on Play, Pause and Stop/Writing a Threaded Main Loop

Listing 3 shows a very simple feedback which is already able to react on Play, Pause and Stop interaction signals. It starts it's main loop in a new thread which does nothing more than incrementing a number and sleeping for a half second. Although very simple, this example already contains many important aspects of real life feedbacks.

`on_play` runs in a different thread It is very important to know that `on_play` runs in a different thread than all the other Feedback methods. This is important since in most cases the `on_play` method will execute for a potentially long time (like several minutes). In order to allow the Feedback Controller to call other methods of the Feedbacks in the meantime, it is necessary to let this method run in a different thread than the other ones.

How to make sure `on_quit` only returns when the main loop is finished? Since the main loop of the feedback will run in a different thread than the rest of the Feedback's methods, you have to make sure that the `on_quit` method does not return until the other thread is killed. In order to achieve that you should set a variable (in this example `quitting`) which will cause the main loop to exit and wait until another variable (`quit`) has been set by the main loop when it exited. This can be done by a simple loop which does nothing until the `quit` variable has been set by the main loop (busy waiting).

Listing 3: A more realistic example.

```
# Lesson02
#  - Starting a main loop in a thread when the feedback gets the start signal
#  - Pausing and unpausing it
#  - Quitting the main loop

from Feedback import Feedback

import time

class Lesson02(Feedback):

    def on_init(self):
        print "Feedback successfully loaded."
        self.quitting, self.quit = False, False
        self.pause = False

    def on_quit(self):
        self.quitting = True
        # Make sure we don't return on_quit until the main_loop (which runs in
        # a different thread!) quit.
        print "Waiting for main loop to quit."
        while not self.quit:
            pass
        # Now the main loop quit and we can safely return

    def on_play(self):
        # Start the main loop. Note that on_play runs in a different thread than
        # all the other Feedback methods, and so does the main loop.
        self.quitting, self.quit = False, False
        self.main_loop()

    def on_pause(self):
        self.pause = not self.pause
```

```
34
35      def main_loop(self):
36          i = 0
37          while 1:
38              time.sleep(0.5)
39              if self.pause:
40                  print "Feedback Paused."
41                  continue
42              elif self.quitting:
43                  print "Leaving main loop."
44                  break
45              i = i+1
46              print "Iteration Number %i" % i
47
48          print "Left main loop."
49          self.quit = True
```

3.4 Working With Data Sent by Control- and Interaction Signals

The Feedback Controller provides a very convenient way to access the data sent by the control- and interaction signals. Listing 4 shows a snippet of Lesson03, which is a variant of Lesson02 as shown in Listing 3 with a modified main loop. The listing demonstrates how to access the data sent by the control- and interaction signal.

Control Signal The data sent by the control signal is stored automaticly in the `_data` variable of your feedback. Everytime the Feedback Controller receives a control signal it overwrites the `_data` variable with the new value.

Interaction Signal The variable assignments sent via the interaction signal are translated into Python and also stored as variables in your feedback. If the interaction signal contains `foo='some string',bar=123,baz=74.11` then your feedback automaticly provides the variables `_foo`, `_bar` and `_baz` with the corresponding values.

Listing 4: Accessing control- and interaction data.

```
1       def main_loop(self):
2           i = 0
3           while 1:
4               time.sleep(0.5)
5               if self.pause:
6                   print "Feedback Paused."
7                   continue
8               elif self.quitting:
9                   print "Leaving main loop."
10                  break
11              print self._data
12              print self._type
13
```

```
14          print "Left main loop."
15          self.quit = True
```

The escape character In order to avoid pollution of the namespace of your feedback, all the variables which are created and updated automaticly by the Feedback Controller are prefixed. By default the prefix is a single underscore character ("_"), you can change the prefix by calling the `Feedback.__init__(self, prefix="_")` method. As you can see the prefix parameter is optional an by default the undersore. Listing 2 shows a variant where the prefix is `foo-`. In that case the variables in the above examples would be `foo-data`, `foo-foo`, `foo-bar` and `foo-baz`.

3.5 Reacting on Control- and Interaction Events

Since the Feedback Controller already stores the data sent by the interaction- and control signals in the feedback, it should not be necessary to directly react on control- or interaction events in most cases (Please consider the hints about coding style in section 4.6). However, in some cases you might want to do something whenever the feedback receives such a signal. In this case you have to overwrite `on_interaction_event` respectively `on_control_event`. Both methods have the `data` variable which holds the processed and translated contents of the signal. At the time the methods are called those data is already present as attributes in the feedback.

Listing 5: Reacting on control- and interaction events.

```
1  from Feedback import Feedback
2
3  class Lesson04(Feedback):
4
5      def on_init(self):
6          self.myVariable = None
7          self.eegTuple = None
8
9      def on_interaction_event(self, data):
10         # this one is equivalent to:
11         # self.myVariable = self._someVariable
12         self.myVariable = data.get("someVariable")
13         print self.myVariable
14
15     def on_control_event(self, data):
16         # this one is equivalent to:
17         # self.eegTuple = self._data
18         self.eegTuple = data
19         print self.eegTuple
```

Listing 5 shows a trivial example how to react on interaction- and control events.

3.6 Sending Markers to the Parallel Port

Most feedbacks need to send so called markers to the parallel port on various events. For this task the Feedback base class provides the `send_parallel(self, data, reset=True)` method, which sends the given data to the parallel port if possible.

Unlike the other methods of the Feedback base class you don't need to overwrite this method to do something usefull. The method is already implemented in the Feedback base class.

The `send_parallel` method will send the given byte to the parallel port and reset the port again to zero after 10ms. Listing 6 shows the example feedback Lesson05.

Listing 6: Sending markers.

```
1  from Feedback import Feedback
2
3  class Lesson05(Feedback):
4
5      def on_init(self):
6          self.send_parallel(0x1)
7
8      def on_play(self):
9          self.send_parallel(0x2)
10
11     def on_pause(self):
12         self.send_parallel(0x4)
13
14     def on_quit(self):
15         self.send_parallel(0x8)
```

Read section 4.7 if you have problems acessing the parallel port under Linux.

3.7 Using the Framework's Logging Facility

The Feedback base class already has a logger build in, available through the `self.logger` variable. Listing 7 shows Lesson06, a modified version of Lesson01 as shown in 1. Instead of a simple `print`, this version uses the logger which can be silenced at a central point in the Feedback Controller so you don't have to comment in and -out all the print lines as you would do without using a logger.

Listing 7: Using the Feedback's logger.

```
1  from Feedback import Feedback
2
3  class Lesson06(Feedback):
4
5      def on_init(self):
6          self.logger.debug("Feedback successfully loaded.")
7
8      def on_quit(self):
9          self.logger.debug("Feedback quit.")
```

Configuring the Logger The framework's logging facility is configurable through command line parameters:

–loglevel=LOGLEVEL Controls which messages appear on the console and which are suppressed. Possible values are in ascending order: notset, debug, info, warning, error and critical, the default loglevel is warning. Setting the loglevel to error means every message with a level error and higher will be printed out to the console, while the messages with a lower level than error are suppressed.

This option controls the loglevel of the Feedbacks and the Feedback Controller. It's use is equivalent of setting both levels (see below) separately to the same level.

–fb-loglevel=LOGLEVEL Sets the loglevel for the Feedback, default is warning.

–fc-loglevel=LOGLEVEL Sets the loglevel for the Framework, default is warning.

Note that you can assign a different loglevels to the feedback and the framework. This is very useful if you want to use lower loglevels for your feedback but are not interested in the low level logmessages of the feedback controller.

Since the loglevel option sets both levels: for the Feedbacks and the Feedback Controller, it is possible to set conflicting loglevels. If you set conflicting loglevels, the lowest of the desired loglevels will be taken.

Example: Setting `--loglevel=info --fb-loglevel=debug --fc-loglevel=warning` will set the loglevel for the Feedback Controller to `info` since it is the smallest of the two conflicting values `info` and `warning`. The loglevel for the Feedbacks will be set to `debug` since `debug` is lower than `info`.

3.8 FeedbackCursorArrow – a Complete Example

The above sections only discussed single aspects of writing feedbacks. There also exists a complete example feedback called FeedbackCursorArrow. It is a complete rewrite of the Matlab feedback `Feedback_cursor_arrow`. The feedback is a game where an arrow is randomly shown pointing at one of two different directions and where the subject has to try to move a cursor to the correct direction into the target zone in a short period of time.

The feedback is located in the Feedbacks directory and gives you an example how a feedback written in Python, using the Framework and Pygame could look like.

4 Notes

4.1 Packages the Framework Depends on

In order to get the framework running you have to install the following packages:

Python The framework needs this package to execute. The version under which the framework was developed is 2.4.

pyParallel Used by the framework to utilize the machine's parallel port. You can download it from here [2] or on Debian based distributions get it by installing the package python-parallel.

pyGame Download it from here [1], the according Debian package is called: python-pygame

4.2 Using Threads in Your Feedback

If you for some reason have to use threads in your feedback, you should take special care to kill them before the feedback returns it's `on_quit` method. Please keep in mind, that by default the Feedbacks `on_play` method already runs in a different thread than all the other methods of the Feedback (and so do all the methods which are called by this method).

Python supports two ways of using threads in your application. The first one comes from the thread-module and provides very low-level primitives for working with multiple threads [4]. The `start_new_thread(function, args[, kwargs])` method provides a painless way to start a method in a new thread. Unfortunately it does not return a handle to the thread and you have no way to control it directly.

The second way is to use Python's threading module . It provides higher-level interfaces on top of the low-level thread module [5]. Among others, it provides the Thread class which provides important methods like `start`, `join` and `isAlive`. The `join` method is particularly important if you want to make sure that the `on_quit` method of your feedback does not return before all the threads of the feedback have quit. Listing 8 shows an example, where the `on_quit` method checks via `isAlive` if the thread is alive and waits until the thread has quit before calling `pygame.quit()` and returning to the Feedback Controller.

Listing 8: on_quit method of the CursorArrow-Feedback.

```
1     def on_quit(self):
2         self.quit = True
3         if self.someThread.isAlive():
4             self.someThread.join()
5         pygame.quit()
```

4.3 Pygame and Threads

When writing a graphical feedback in Python using Pygame, you will almost certainly want to use pygame.time.Clock.tick(FPS) to limit the framerate your feedback is running. Python currently does not utilize more than one CPU and just emulates threads by running each thread for a small amount of time sequentially. For a detailed explanation about Python's way of handling threads and a nice introduction into threads in general see [9].

Under some circumstances Pygame's clock.tick() implementation might use busy waiting and will therefore not put the thread to sleep to share the processor with other

threads. Since most reasonable non-trivial feedbacks will at least use one thread for their main loops, this could lead to a situation where the framework's network threads starve and aren't able to provide new control- and interaction signals.

A workaround for this problem is to forcefully put the thread into sleep for a small amount of time before limiting the framerate via Clock.tick(). So instead of the intuitive variant in listing 9

Listing 9: Limiting the framerate the intuitive (but wrong) way.

```
1 self.clock.tick(self.FPS)
```

you should wait for a small amount of time to put the thread into the sleeping state to share the CPU with the other threads. Listing 10 shows an example.

Listing 10: A working alternative.

```
1 pygame.time.wait(10)
2 self.clock.tick(self.FPS)
```

wait(10) will sleep the current thread for 10 milliseconds and share the processor with other threads. Since `tick()` remembers the time it was called the last time, calling `wait(n)` will have no negative effects as long as the sum of the time n and the time needed for one loop of the main loop is not bigger than the reciprocal of the desired framerate.

4.4 Pygame and Threads II

Another important aspect of threading with Pygame comes with a limitation of pygame: The `pygame.init()` and the polling of Pygame's event queue have to take place in Python's main thread! Otherwise on some operating systems the whole application will not respond correctly to the operating system an might appear to be hanging.

The Feedback's `on_play` method already runs in Python's main thread for this very reason. All you have to take care of is to initialize pygame and poll pygame's event queue in this method (or in one of the methods `on_play` calls).

4.5 Polling Pygame's Event Queue

This important point is somewhat hidden in pygame's online documentation [3]:

> "Your program must take steps to keep the event queue from overflowing. If the program is not clearing or getting all events off the queue at regular intervals, it can overflow. When the queue overflows an exception is thrown."

This means that you must poll pygame's event queue once per frame(!) even if you don't use the event queue.

The example `FeedbackCursorArrow` provides a method `process_pygame_events` which can serve as an example. It polls the event queue to react on changes of the window dimensions.

If you don't use pygame's event queue, you still have to poll the queue so pygame can process the events internally, you can achieve this by calling `pygame.event.pump()` once per frame

4.6 Good Coding Style Regarding the Interaction Signal

Allthough the Feedback Controller automaticly creates and updates variables in your feedback on interaction signals, you should consider having a central point where you connect the variable names from the interaction signal with the counterparts of your feedbacks. For example if the GUI sends the `foo` variable and your feedback uses the `_foo` provided by the Feedback Controller many times on different places in the code. Imagine the variable name is changed in the GUI from `foo` to `bar`, now you would have to search all over the code and change all occurrences of `_foo` to `_bar`. This can be done semi automaticly with the right tools but is a potentially dangerous action (imagine you already had a variable called `_foo`).

A clean solution would be to utilize the `on_interaction_event` method of your feedback to translate the external variable names to your local ones. This way you would have one central point where you would have to care about each variable from the interaction signal exactly once.

4.7 Using the Parallel Port Under Linux

On our Linux systems running Debian/Etch we were not able to use the parallel port by default. The problem was the lp kernel module which occupies the parallel port and insufficient user permissions to use the parallel port. The following commands in listing 11 will unload the lp kernel module and make the parallel port writable for everyone.

Listing 11: Making the parallel port available on Debian/Etch

```
1 user@box$ sudo modprobe -r lp
2 user@box$ sudo chmod 666 /dev/parport0
```

This has to be done everytime the machine on which the framework is running is rebooted.

References

[1] Homepage of pygame.
http://www.pygame.org.

[2] Homepage of pyparallel.
http://pyserial.sourceforge.net/pyparallel.html.

[3] Important note on polling pygame's event queue.
http://www.pygame.org/docs/ref/event.html.

[4] Python's documentation of the thread module.
http://www.python.org/doc/2.4.4/lib/module-thread.html.

[5] Python's documentation of the threading module.
http://www.python.org/doc/2.4.4/lib/module-threading.html.

[6] Python's wiki, providing tutorials for non-programmers.
http://wiki.python.org/moin/BeginnersGuide/NonProgrammers.

[7] Various documentation available on python's website.
http://www.python.org/doc/.

[8] Mark Lutz. *Python Pocket Reference, Third Edition.* O'Reilly Media, Inc., 2005.

[9] Norman Matloff and Francis Hsu. *Tutorial on Threads Programming with Python.* University of California, 2003-2007.
http://heather.cs.ucdavis.edu/~matloff/Python/PyThreads.pdf.