pd-faust: An integrated environment for running Faust objects in Pd

Albert Gräf

pd-faust: An integrated environment for running Faust objects in Pd

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University
55099 Mainz, Germany
Dr.Graef@t-online.de

Abstract

This paper introduces pd-faust, a library for running signal processing modules written in Grame's functional dsp programming language Faust in Miller Puckette's graphical computer music environment Pure Data a.k.a. Pd. pd-faust is based on the author's faust2pd script which generates Pd GUIs from Faust programs and also provides the necessary infrastructure for running Faust dsps in Pd. pd-faust combines this functionality with its own Faust plugin loader which makes it possible to reload Faust dsps while a patch is running. It also adds automatic configuration of MIDI and OSC controller assignments, as well as OSC-based automation features.

Keywords

Faust, Pd, Pure, functional signal processing.

1 Introduction

We assume that the reader is familiar (or has at least heard of) Grame's popular Faust programming language [5], which greatly facilitates the programming of custom audio processing plugins. Faust programs can be compiled to native code for an abundance of different signal processing environments and plugin standards. In particular, Faust has had support for Miller Puckette's Pd [6] for some time now through its puredata.cpp architecture. Pd users in the Faust community have also been using the author's faust2pd script to create Pd GUIs (graphical user interfaces) for Faust externals which seem to work quite well for operating Faust dsps inside Pd [1].

One of faust2pd's shortcomings is that it generates the Pd GUIs outside of Pd. Thus the generated GUI is static, and changing the Faust source of a dsp generally requires regeneration of the GUI and reloading of the hosting Pd patch to make the changes take effect. Another obstacle is that the necessary infrastructure for processing MIDI note input and control changes is implemented entirely in Pd, which may involve a lot of Pd objects and become a cpu hog for complex Faust programs.

pd-faust was designed to overcome these limitations. Most notably, it loads Faust dsps dynamically and allows them to be reloaded at any time, in which case the Pd GUI is regenerated automatically and instantly. Thus you can now just edit and recompile the Faust source and have pd-faust pick up the changes on the fly, while the Pd patch keeps running.

pd-faust is implemented as a library of Pd objects written in the author's Pure programming language [2], which is compiled to native code, so that Faust dsps involving a lot of different controls are handled in an efficient manner. Another advantage of using Pure is that at present it is the *only* programming language with a built-in Faust interface based on the LLVM toolkit (which Pure itself uses as its code generation backend). This enables pd-faust to directly load compiled Faust programs in LLVM bitcode format [3] if you're running a suitable version of the Faust compiler [4].

pd-faust offers a number of other enhancements facilitating the use of Faust dsps in Pd. While it is based on the GUI generation code of the faust2pd script and thus supports all of faust2pd's global GUI layout options, it also provides various options to adjust the layout of individual control items. In addition, pd-faust recognizes the `midi` and `osc` controller attributes in the Faust source and automatically provides corresponding MIDI and OSC controller mappings. OSC-based controller automation is also avail-

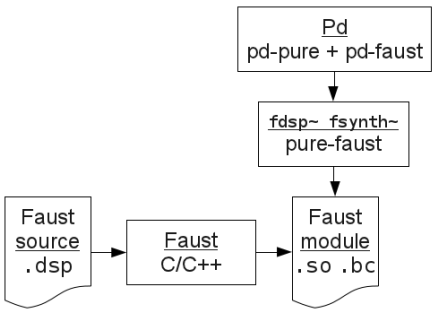

Figure 1: An overview of the pd-faust system.

able. These additional features are all described in some detail in this paper.

Section 2 starts out with a brief overview of pd-faust. In Sections 3 and 4 we then take a more in-depth look at the pd-faust objects and describe how pd-faust constructs the Pd GUI at runtime. Section 5 briefly discusses how to operate the resulting Pd patches. Sections 6 and 7 show how pd-faust uses metadata in Faust programs to define controller mappings and adjust the GUI layout. Section 8 briefly touches on the auxiliary facilities provided to ease livecoding with Faust dsps. Section 9 concludes with pointers to additional information and examples, and discusses possible directions for further work.

The paper assumes some working knowledge of Faust and Pd; please consult [5] and [6] if necessary.

2 Overview

Before we go into the technical details, let us first give a brief overview of pd-faust and the various software components which are involved in the system (cf. Fig. 1).

pd-faust is implemented as an object library for Pd. Since the pd-faust objects are written in Pure, you'll also need the pd-pure plugin loader [2] which provides the necessary infrastructure to run Pure objects in Pd. Both libraries are available in the form of shared modules which are loaded by Pd on startup, using either Pd's `-lib` option or a corresponding entry in Pd's startup preferences.

The main ingredients of pd-faust are the `fdsp~` and `fsynth~` objects which are used to load and run different kinds of Faust dsps inside Pd (the differences between these will be explained in the following section). The basic functionality to do this is actually provided by another Pure module, pure-faust, which can load Faust modules in one of two formats:

- *Native* modules are shared modules in the format supported by the host operating system (`.so` on ELF systems, `.dll` on Windows, etc.). These are created from Faust source programs by invoking the Faust compiler with the pure.cpp architecture file (`faust -a pure.cpp`) and compiling the resulting C++ source to a shared module.
- *Bitcode* modules are modules in LLVM bitcode format which can be created directly by Faust (`faust -lang llvm`). You'll need Faust2, the development version of Faust, to make this work [4]. The Pure interpreter has a built-in LLVM bitcode linker which enables it to load these modules [3]; the executable code of the module is then generated on the fly when the module is loaded.

The internal workings of pd-faust are illustrated in Fig. 2. The `fdsp~` and `fsynth~` objects use the operations provided by the Faust module to instantiate a Faust dsp and extract the needed information from the dsp in order to construct its Pd GUI. The GUI is then inserted into the hosting Pd patch by means of Pd's "FUDI" protocol.[1] All this happens automatically whenever the Pd patch is loaded or a Faust module gets reloaded by sending it a corresponding control message.

While the patch is running, an `fdsp~` or `fsynth~` object receives incoming control messages and audio data through its inlets and invokes the operations of the dsp to change the control values and compute blocks of audio data as they are requested by Pd's audio loop. The generated data is then output through the object's audio and control outlets.

3 The fdsp and fsynth objects

Working with pd-faust basically involves adding a bunch of `fsynth~` and `fdsp~` objects to a Pd patch along with the corresponding GUI subpatches, and wiring up the Faust units in some

[1] See `http://wiki.puredata.info/en/FUDI`. This protocol is also used internally by Pd to represent the contents of patches and communicate with its GUI process.

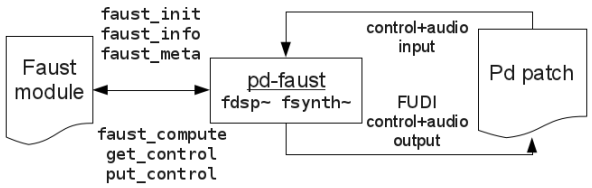

Figure 2: Internals of the pd-faust system.

variation of a synth-effects chain which typically takes input from Pd's MIDI interface (`notein`, `ctlin`, etc.) and outputs the signals produced by the Faust units to Pd's audio interface (`dac~`).

For convenience, pd-faust also includes the `midiseq` and `oscseq` objects as well as a corresponding `midiosc` abstraction which can be used to handle MIDI input and playback as well as OSC controller automation. These objects are described in more detail in Section 5.

The `fdsp~` object is invoked as follows:

```
fdsp~ dspname instname channel
```

- `dspname` denotes the name of the Faust dsp (usually this is just the name of the .dsp file with the extension stripped off). Please note that, as already mentioned, the Faust dsp must be provided in a form which can be loaded in *Pure* (not Pd!), so the pure.cpp architecture included in recent Faust versions must be used to compile the dsp to a shared library. (If you're already running Faust2, you can also compile to an LLVM bitcode file instead; Pure has built-in support for loading these.) The Makefiles included in the pd-faust distribution show how to do this.
- `instname` denotes the name of the instance of the Faust unit. Multiple instances of the same Faust dsp can be used in a Pd patch, which must all have different instance names. In addition, the instance name is also used to identify the GUI subpatch of the unit (see below) and to generate unique OSC addresses for the unit's control elements.
- `channel` is the number of the MIDI channel the unit responds to. This can be 1..16, or 0 to specify "omni" operation (listen to MIDI messages on all channels).

The `fdsp~` object requires a Faust dsp which can work as an effect unit, processing audio input and producing audio output.

The `fsynth~` object works in a similar fashion, but has an additional creation argument specifying the desired number of voices:

```
fsynth~ dspname instname channel nvoices
```

The `fsynth~` object requires a Faust dsp which can work as a monophonic synthesizer (having zero audio inputs and a nonzero number of audio outputs). To these ends, pd-faust assumes that the Faust unit provides three so-called "voice controls" which indicate which note to play:

- `freq` is the fundamental frequency of the note in Hz.
- `gain` is the velocity of the note, as a normalized value between 0 and 1. This usually controls the volume of the output signal.
- `gate` indicates whether a note is currently playing. This value is either 0 (no note to play) or 1 (play a note), and usually triggers the envelop function (ADSR or similar).

pd-faust doesn't care at which path inside the Faust dsp these controls are located, but for the synthesizer to function properly they must all be there, and the basenames of the controls must be unique throughout the entire dsp.

Like faust2pd, pd-faust implements the necessary logic to drive the given number of voices of an `fsynth~` object. That is, it will actually create a separate instance of the Faust dsp for each voice and handle polyphony by allocating voices from this pool in a round-robin fashion, performing the usual voice stealing if the number of simultaneous notes to play exceeds the number of voices.

The `fdsp~` and `fsynth~` objects respond to the following messages:

- `bang` outputs the current control settings on the control outlet in (symbolic) OSC format.[2]

[2] pd-faust represents OSC messages as ordinary Pd messages with the OSC address in the selector symbol of the message. Input and output of binary OSC messages is assumed to be handled by a separate OSC library which is not

- `write` outputs the current control settings to external MIDI and/or OSC devices. This message can also be invoked with a numeric argument to toggle the "write mode" of the unit; please see Section 6 for details.
- `reload` reloads the Faust unit. This also reloads the shared library or bitcode file if the unit was recompiled since the object was last loaded.
- `addr value` changes the control indicated by the OSC address `addr`. This is also used internally for communication with the Pd GUI and for controller automation.

In addition, the `fdsp~` and `fsynth~` objects respond to MIDI controller messages of the form `ctl val num chan`, and the `fsynth~` object also understands note-related messages of the form `note num vel chan` (note on/off) and `bend val chan` (pitch bend). In either case, pd-faust provides the necessary logic to map controller and note-related messages to the corresponding control changes in the Faust unit.

4 GUI subpatches

For each `fdsp~` and `fsynth~` object, the Pd patch should also contain an (initially empty) "one-off" graph-on-parent subpatch with the same name as the instance name of the Faust unit:

```
pd instname
```

You shouldn't insert anything into this subpatch, its contents (a bunch of Pd GUI elements corresponding to the control elements of the Faust unit) will be generated automatically by pd-faust when the corresponding `fdsp~` or `fsynth~` object is created, and whenever the unit gets reloaded at runtime. See Fig. 3 for an example.

As with faust2pd, the GUI layout follows the hierarchical structure of the controls in the Faust program which places controls in different *control groups*, please check the Faust documentation for details. The default appearance of the GUI can also be adjusted in various ways; see Section 7 for details.

part of pd-faust. E.g., one might use Martin Peach's collection of OSC objects for that purpose, see `http://puredata.info/Members/martinrp/OSCobjects`.

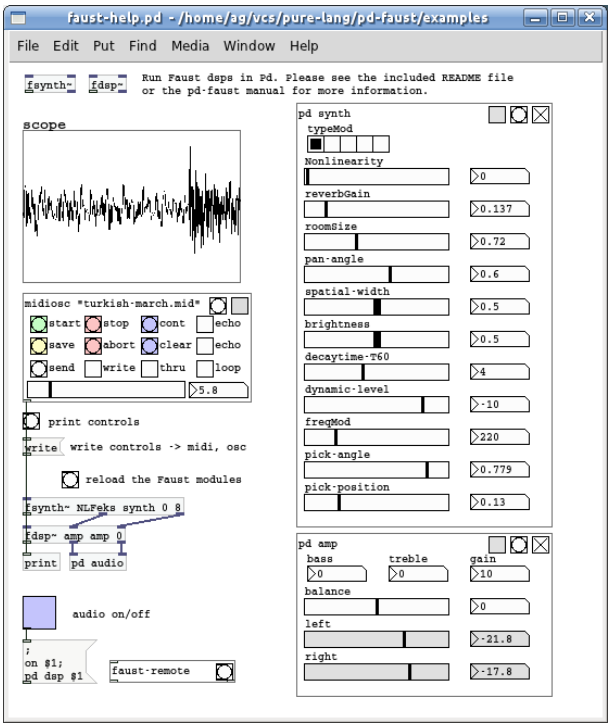

Figure 3: Sample pd-faust patch.

The relative order in which you insert an `fdsp~` or `fsynth~` object and its GUI subpatch into the main patch matters. Normally, the GUI subpatch should be inserted *first*, so that it will be updated automatically when its associated Faust unit is first created, and also when the main patch is saved and then reloaded later.

However, in some situations it may be preferable to insert the GUI subpatch *after* its associated Faust unit. If you do this, the GUI will *not* be updated automatically when the main patch is loaded, so you'll have to reload the dsp manually (sending it a `reload` message) to force an update of the GUI subpatch. This is useful, in particular, if you'd like to edit the GUI patch manually after it has been generated.

In some cases it may even be desirable to completely "lock down" the GUI subpatch. This can be done by simply *renaming* the GUI subpatch after it has been generated. When Pd saves the main patch, it saves the current status of the GUI subpatches along with it, so that the renamed subpatch will remain static and will *never* be updated, even if its associated Faust unit gets reloaded. This generally makes sense only if the

Figure 4: Close-up view of the `midiosc` abstraction.

control interface of the Faust unit isn't changed after locking down its GUI patch. To "unlock" a GUI subpatch, you just rename it back to its original name.

5 Operating the patches

The generated Pd GUI elements for the Faust dsps are pretty much the same as with faust2pd. See Fig. 3, which shows the `midiosc` abstraction and the actual Faust objects on the left, and the corresponding GUI subpatches on the right. The only obvious change is the addition of a "record" button (the gray toggle on the left of the button row located in the upper right corner of each GUI) which enables recording of OSC automation data (described below).

The `midiosc` abstraction (cf. Fig. 4) shown in the examples serves as a little sequencer applet that enables you to control MIDI playback and OSC recording. The creation arguments of `midiosc` are the names of the MIDI and OSC files. If the second argument is omitted, it defaults to the name of the MIDI file with new extension .osc. You can also omit both arguments if neither MIDI file playback nor saving recorded OSC data is required.

The abstraction has a single control outlet through which it feeds the generated MIDI data and other messages to the connected `fsynth~` and `fdsp~` objects. Live MIDI input is also accepted and forwarded to the control outlet, after being translated to the format understood by `fsynth~` and `fdsp~` objects. Moreover, you can also control `midiosc` with an external sequencer program, employing MIDI Machine Control (MMC) for synchronization.

At the bottom of the abstraction there is a little progress bar along with a time display which indicates the current song position. If playback is stopped, you can also use these to change the start position for playback and recording.

Here is a brief rundown of the available controls:

- The gray "record" toggle in the upper right corner of the abstraction enables recording of OSC controller automation data. Note that this toggle merely *arms* the OSC recorder; you still have to actually start the recording with the `start` button. However, you can also first start playback with `start` and then switch recording on and off as needed at any point in the sequence. Pushing the `stop` button then stores the recorded sequence for later playback. Also note that before you can start recording any OSC data, you first have to arm the Faust units that you want to record. This is done with the "record" toggle in the Pd GUI of each unit.
- The "bang" control next to the "record" toggle lets you record a snapshot of the current controller settings. This is also done automatically when you start recording a new sequence.
- The `start`, `stop` and `cont` controls in the first row of control elements start, stop and continue MIDI and OSC playback, respectively. The `echo` toggle in this row causes played MIDI events to be printed in the Pd main window.
- There are some additional controls related to OSC recording in the second row: `save` saves the currently recorded data in an OSC file for later use; `abort` is like `stop` in that it stops recording and playback, but also throws away the data recorded in this take (rather than keeping it for later playback); `clear` purges the entire recorded OSC sequence so that you can start a new one; and the `echo` toggle, if enabled, prints the OSC messages as they are played back.
- The controls in the third row provide some additional ways to configure the playback process. The `loop` button can be used to enable looping, which repeats the playback of the MIDI (and OSC) sequence ad infinitum. The `thru` button, when switched on, routes the MIDI data during playback through Pd's

MIDI output so that it can be used to drive an external MIDI device in addition to the Faust instruments. The `write` button does the same with MIDI and OSC controller data generated either through automation data or by manually operating the control elements in the Pd GUI, see Section 6 for details. The `send` button recalls the recorded OSC parameter settings at a given point in the sequence, and updates the GUI controls accordingly.

Once some automation data has been recorded, it will be played back along with the MIDI file. You can then just listen to it, or go on to record more automation data as needed. If you save the automation data with the `save` button, it will be reloaded from its OSC file next time the patch is opened.

OSC sequences are saved in a simple ASCII format which should be self-explanatory and can be edited with any text editor. For instance:

```
# written by oscseq Sun Dec 11 19:39:45 2011

# delta  /oscaddr value
0        /synth:Nonlinear-Filter/typeMod 0
0        /synth:Nonlinearity 0
0        /synth:Reverb/reverbGain 0.137
0        /synth:Reverb/roomSize 0.72
```

The OSC addresses are generated automatically from the hierarchical structure of the controls in the Faust program, using the instance name as well as the labels of control groups and elements to assign a unique pathname to each control of each Faust unit in the patch.

Note that `midiosc` is merely an example which should cover most common uses and can be customized for the target application as needed. You may even do without it and have your patches feed control messages directly into `fdsp~` and `fsynth~` objects instead. Internally, `midiosc` uses two utility objects `midiseq` and `oscseq` written in Pure and contained in the pd-faust object library. Together these implement most of the MIDI playback and OSC recording functionality; `midiosc` basically just adds the necessary wiring with Pd's MIDI I/O and a few control elements. The `midiseq` and `oscseq` objects can also be used directly in your patch if you prefer to forego `midiosc`, or you may replace them altogether with your own sequencer objects.

6 External MIDI and OSC controllers

The `fsynth~` object has built-in (and hard-wired) support for MIDI notes, pitch bend and MIDI controller 123 (all notes off).

Other controller data received from external MIDI and OSC devices is interpreted according to the controller mappings defined in the Faust source (this is explained below), by updating the corresponding GUI elements and the control variables of the Faust dsp. For obvious reasons, this only works with *active* Faust controls.[3]

An `fdsp~` or `fsynth~` object can also be put in *write mode* by feeding a message of the form `write 1` into its control inlet (the `write 0` message disables write mode again). For convenience, the `write` toggle in the `midiosc` abstraction allows you to do this simultaneously for all Faust units connected to `midiosc`'s control outlet.

When an object is in write mode, it also *outputs* MIDI and OSC controller data in response to both automation data and the manual operation of the Pd GUI elements according to the controller mappings defined in the Faust source, so that it can drive an external device such as a MIDI fader box or a multitouch OSC controller. Note that this works with both *active* and *passive* Faust controls.

To configure MIDI controller assignments, the labels of the Faust control elements have to be marked up with the special `midi` attribute in the Faust source. For instance, a pan control (MIDI controller 10) may be implemented in the Faust source as follows:

```
pan = hslider(
  "pan[midi:ctrl␣10]",0,-1,1,0.01);
```

pd-faust will then provide the necessary logic to handle MIDI input from controller 10 by changing the pan control in the Faust unit accordingly, mapping the controller values 0..127 to the range and step size given in the Faust source. Moreover, in write mode corresponding MIDI controller data will be generated and sent

[3] Faust distinguishes between *active* controls which take input from the user and *passive* controls displaying control data computed in the Faust program, such as RMS envelopes.

to Pd's MIDI output, on the MIDI channel specified in the creation arguments of the Faust unit (0 meaning "omni", i.e., output on all MIDI channels).

The same functionality is also available for external OSC devices, employing explicit OSC controller assignments in the Faust source by means of the `osc` attribute. E.g., the following enables input and output of OSC messages for the OSC `/pan` address:

```
pan = hslider("pan[osc:/pan]",0,-1,1,0.01);
```

Note that in contrast to some architectures included in the Faust distribution, pd-faust only allows literal OSC addresses here. That is, glob-style OSC patterns are not supported as values for the `osc` attribute. Also note that pd-faust currently does not include any facilities for actual OSC input/output, but it's easy to add this to `midiosc` if needed.[4]

7 Tweaking the GUI layout

As already mentioned, pd-faust provides the same global GUI layout options as faust2pd. There are a few minor changes in the meaning of some of the options, though. Here is a brief rundown of the available options, as they are implemented in pd-faust:

- `width=wd`, `height=ht`: Specify the maximum horizontal and/or vertical dimensions of the layout area. If one or both of these values are nonzero, pd-faust will try to make the GUI fit within this area.
- `font-size=sz`: Specify the font size (default is 10).
- `fake-buttons`: Render `button` controls as Pd toggles rather than bangs.
- `radio-sliders=max`: Render sliders with up to `max` different values as Pd radio controls rather than Pd sliders. Note that in pd-faust this option not only applies to sliders, but also to numeric entries, i.e., `nentry` in the Faust source. However, as with faust2pd's `radio-sliders` option, the option is only applicable if the control is zero-based and has a stepsize of 1.
- `slider-nums`: Add a number box to each slider control. Note that in pd-faust this is actually the default, which can be disabled with the `no-slider-nums` option.
- `exclude=pat,...`: Exclude the controls whose labels match the given glob patterns from the Pd GUI.

In pd-faust there is no way to specify the above options on the command line, so you'll have to put them as `pd` attributes on the *main group* of your Faust program, as described in the faust2pd documentation. For instance:

```
process = vgroup(
  "[pd:no-slider-nums][pd:font-size=12]",
  ...);
```

In addition, the following options can be used to change the appearance of individual control items. If present, these options override the corresponding defaults. (Each option can also be prefixed with "`no-`" to negate the option value. Thus, e.g., `no-hidden` makes items visible which would otherwise, by means of the global `exclude` option, be removed from the GUI.)

- `hidden`: Hides the corresponding control in the Pd GUI. This is the only option which can also be used for control groups, in which case *all* controls in the group will become invisible in the Pd GUI.
- `fake-button`, `radio-slider`, `slider-num`: These have the same meaning as the corresponding global options, but apply to individual control items.

These options are specified with the `pd` attribute in the label of the corresponding Faust control or control group. For instance, the following Faust code hides the controls in the `aux` group, removes the number entry from the `pan` control, and renders the `preset` item as a Pd radio control:

```
aux = vgroup("aux[pd:hidden]", aux_part);
pan = hslider(
  "pan[pd:no-slider-num]",0,-1,1,0.01);
preset = nentry(
  "preset[pd:radio-slider]",0,0,7,1);
```

[4] "Vanilla" Pd doesn't provide any objects for OSC I/O, but various suitable externals are readily available, such as Martin Peach's OSC objects already mentioned above. The pd-faust distribution includes a variation of the `midiosc` abstraction which implements this.

8 Remote control

Also included in the sources is a helper abstraction faust-remote.pd and an accompanying elisp program faust-remote.el. These work pretty much like pure-remote.pd and pure-remote.el in the pd-pure distribution, but are tailored for the remote control of Faust dsps in a Pd patch. In particular, they enable you to quickly reload the Faust dsps in Pd using a simple keyboard command (`C-C C-X` by default) from Emacs. The faust-remote.el program was designed to be used with Juan Romero's Emacs Faust mode.[5]

9 Conclusion

We hopefully convinced the reader that pd-faust is an interesting solution for developing, testing, deploying and running Faust dsps in the graphical Pd environment. It provides Pd and Faust users with a fairly complete integrated development environment for Faust, and supports a freewheeling interactive and experimental development style which should appeal to both developers and artists.

The software described in this paper is available freely under the LGPL[6] from the Pure website.[7] As already mentioned, pd-faust is written in Pure; thus, to build and run the software you'll also need an installation of the Pure interpreter and a couple of Pure addon packages including the latest release of pd-pure [2] which is needed to run Pure externals in Pd; please check the pd-faust documentation for details.

The package also contains a few example patches and accompanying files which illustrate how this all works. Here are some of the examples that you might want to take a look at:

- test.pd: very basic example running a single Faust instrument
- synth.pd: slightly more elaborate patch featuring a synth-effects chain
- bouree.pd: full-featured example running various instruments

pd-faust development continues, so you might want to check out the latest development sources in the Pure source code repository. There are some technical issues and limitations in the current pd-faust version which will hopefully be ironed out in the future. Most notably:

- In the present implementation, pure-faust supports Faust modules either in native or in LLVM bitcode form, but not both at the same time. Therefore there are two corresponding versions of the pd-faust object library, and you'll have to decide in advance whether you'd like to work with native or bitcode modules.
- The names of the `fsynth~` voice controls (`freq`, `gain`, `gate`) are currently hardcoded, and there must be exactly one instance of each of these controls in the dsp, otherwise `fsynth~` may not function as advertized.
- Passive Faust controls are only supported in `fdsp~` objects right now.
- The OSC recording capabilities are somewhat limited in the current version. In particular, it should be possible to erase and edit already recorded controller data while recording. At present you can only change existing automation data by purging the entire sequence and starting over, or by editing the OSC file. This is sufficient for testing purposes but not adequate for real musical work.

Further research

Faust's abstract GUI model seems perfectly appropriate for the type of control processing applications that are typically written using pd-pure. pd-faust is just one of the many possible pd-pure applications which might benefit from this approach. Therefore an interesting question for further research is how to integrate the corresponding pd-faust functionality into pd-pure and make it available to *all* pd-pure applications.

Also, it might be useful to port pd-faust to other realtime environments, such as SuperCollider, and suitable plugin environments, such as LV2 and VST. In principle it's also conceivable to run a suitably modified version of pd-faust directly as a Jack client, without the extra Pd layer. Of course, the details of integrating Faust dsps with these host environments differ considerably

[5] `https://github.com/rukano/emacs-faust-mode`

[6] `http://www.gnu.org/copyleft/lgpl.html`

[7] `http://pure-lang.googlecode.com`

from the current implementation running inside Pd, so porting the interface will be a substantial amount of work.

Acknowledgements

Special thanks are due to Julius O. Smith from CCRMA for following this project from its inception, taking the time to test early alpha versions and suggesting improvements. I'd also like to thank Johannes Zmölnig from the IEM Graz for pointing me to Pd's internal FUDI protocol which made this project feasible in the first place.

References

[1] A. Gräf. Interfacing Pure Data with Faust. In *Proceedings of the 5th International Linux Audio Conference*, pages 24–32, Berlin, 2007. TU Berlin.

[2] A. Gräf. Signal processing in the Pure programming language. In *Proceedings of the 7th International Linux Audio Conference*, Parma, 2009. Casa della Musica.

[3] A. Gräf. An LLVM bitcode interface between Pure and Faust. In *Proceedings of the 9th International Linux Audio Conference*, Maynooth, 2011. NUI Maynooth, Dept. of Music.

[4] S. Letz. LLVM backend for Faust. `http://www.grame.fr/~letz/faust_llvm.html`, 2012.

[5] Y. Orlarey, D. Fober, and S. Letz. FAUST : an efficient functional approach to DSP programming. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*. Editions Delatour France, 2009.

[6] M. Puckette. *The Theory and Techniques of Electronic Music*. World Scientific Publishing Co., 2007.