Linux Audio Conference 2013

Proceedings

May 9.–12. 2013 / Graz

Institute of Electronic Music and Acoustics (IEM)
University of Music and Performing Arts Graz

lac.linuxaudio.org/2013

Published by
Institute of Electronic Music and Acoustics,
University for Music and Performing Arts Graz, Austria
May 2013
Editors: IOhannes m zmölnig and Peter Plessas

`http://lac.linuxaudio.org/2013`
ISBN 978-3-902949-00-4

Credits
Logo Design LAC 2013: Renatn Oblak
Cover Design: Fränk Zimmer
Layout: Frank Neumann, Katharina Vogt
Typesetting: LaTeX and pdfLaTeX

Thanks to:
Martin Monperrus for his webpage "Creating proceedings from PDF files"

Printed in Austria

Welcome to the LAC2013 in Graz!

For eleven wonderful years the Linux Audio Conference has been one of the main events on the schedule of many developers and users of free/libre/open source software (FLOSS) in the field of music and multimedia.

It started with the "LAD booth" at the Linuxtag in Karlsruhe, and gradually became LAC (actually the Linux Audio Developers Meeting), a real-world meeting of audio developers who would otherwise only meet via Internet. Finally, it has turned into a full-blown conference that attracts people from a variety of locations and disciplines. With its all-encompassing commitment to open systems, LAC is known as one of the most diverse annual events in human history (at the very least).

The Institute of Electronic Music and Acoustics (IEM) at the University of Music and Performing Arts in Graz has a long history of using and teaching Linux and FLOSS audio applications for scientific research and works of art. After having benefitted for so many years, we felt like having to give something back to the community by bringing the conference to Graz.

When attending LAC 2009 in Parma, one of us was approached by LAC veteran Frank Neumann, who suggested us to host the conference at some point in the future. We were very thrilled, but it took another three years (and LAC crossing the Atlantic ocean for the first time in its history in 2012) until we finally felt confident to host such a great event.

The eleventh edition of the Linux Audio Conference features twenty-four paper presentations, twelve workshops, twenty-nine concerts and six installations. Like in recent years, all submitted papers have undergone a peer-review process, with each paper being reviewed by at least two independent experts, giving valuable feedback to the authors. The amount of papers submitted this year has been bitter sweet: due to the sheer amount of submissions we could only accept a fraction of them.

We hope that the variety in scientific, technical and artistic presentations, together with the efforts of the Graz community members, creates a wonderful event for everyone.

Have a wonderful time in Graz!

Peter Plessas & IOhannes m zmölnig

Conference Organization

Florian Hollerweger
IOhannes m zmölnig
Peter Plessas
Robin Gareus

Conference Team

Christian Pointner
Clara Hollomey
David Pirrò
Dominik Schmidt-Philipp
Frank Neumann
Fränk Zimmer
Georg Holzmann
Katharina Vogt
Josef Schauer
Ludwig Mohr
Margarethe Maierhofer-Lischka
Marian Weger
Martin Schitter
maru
Martin Rumori
Matthias Kronlachner
Peter Venus
Renatn Oblak
Reni Hofmüller
Stefan Warum
Thomas Musil
Visda Goudarzi
Winfried Ritsch
Ypatios Grigoriadis

Thanks to...

Alois Sontacchi
Brigitte Bergner
Djamil Vardag
Franz Zotter
Flo Stöffelmayr
Garfield
Georgios Marentakis
Jörn Nettingsmeier
Markus Guldenschuh
Robert Höldrich
Sieglinde Roth
Thomas Mayr
Tom – Café Erde
Uli Gladisch

...and to everyone else who helped in numerous places after the editorial deadline of this publication.

Paper Review Committee

Frank Neumann	Intel GmbH, Germany (Chair)
Albert Gräf	Johannes Gutenberg University Mainz, Germany
Dave Phillips	linux-sound.org, United States
Fernando Lopez-Lezcano	CCRMA, United States
Florian Hollerweger	Austria
Fons Adriaensen	Casa della Musica, Parma, Italy
Franz Zotter	Institute of Electronic Music and Acoustics Graz, Austria
Georgios Marentakis	Institute of Electronic Music and Acoustics Graz, Austria
Götz Dipper	ZKM | Institut für Musik und Akustik, Germany
IOhannes m zmölnig	Institute of Electronic Music and Acoustics Graz, Austria
John ffitch	United Kingdom
Jörn Nettingsmeier	freelancer, Germany
Marc Groenewegen	HKU, Netherlands
Marije Baalman	nescivi / STEIM, Netherlands
Martin Rumori	Institute of Electronic Music and Acoustics Graz, Austria
Peter Plessas	Institute of Electronic Music and Acoustics Graz, Austria
Robin Gareus	linuxaudio.org, University Paris 8, France
Victor Lazzarini	NUIM, Ireland
Yann Orlarey	Grame, France

Music Jury

Florian Hollerweger (Chair)
David Pirrò
IOhannes m zmölnig
Margarethe Maierhofer-Lischka
Martin Rumori
Peter Venus
Ypatios Grigoriadis

Table of Contents

netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data

Roman HAEFELI
Media Artist
Zürich,
Switzerland
roman.haefeli@gmail.com
http://www.netpd.org

Abstract

This paper presents *netpd*, a framework intended for making music collaboratively and in real-time written in Pure Data (Pd)[1]. Users join by connecting to a central server in order to have a session together (not much unlike a jam session in Jazz music) and load self-written or pre-existing instruments (Pd patches) to play with. The framework maintains state synchronicity between clients at any given time by exchanging control messages over the server. The protocol in use is fully based on OSC[2].

netpd does not address the transmission of audio data, thus it is primarily used for synthesized / generated sound, but might be useful in other areas where state synchronicity is a goal (networked games, graphics, etc.).

Keywords

Pure Data, Network, Music, Real-time, OSC

1 Introduction

Early experiments with transmitting control messages over a network for making music started in 2004 when an early version of *netpd* was developped. While first drafts of instruments were an interleaved blend of DSP parts, message control and state synchronization parts, it became clear soon that a design which clearly separates those parts would allow the creation of a framework that gives the designer of an instrument a high degree of freedom while keeping the complexity of state synchronization under hood. Crucial to the *netpd* experience are two distinct layers:

1. The *netpd* core framework (in this paper simply called framework), which consists of a server, a client application, and a set of abstractions[1] (*netpd-abstractions*) which are used to create *netpd-instruments*.
2. The *netpd-instruments*: Pd-patches created by *netpd* users which are loaded in the client application and played during a session.

This paper primarly addresses the design of the framework, which aims to provide the tools to enable skilled[2] users to design their own instruments to be used and shared with *netpd*. It is important to understand that the presented framework has no notion of music and is only the basis for user-designed instruments and that those instruments make up the *netpd* experience. It becomes apparent that collaboration happens on the level of playing together, but also on the level of designing and sharing patches.

2 Basic design

Users load netpd's client application chat.pd in Pd (or Pd-extended, for that matter) which establishes a connection to a central server. The server acts as a message relay between clients: it forwards incoming messages from clients to any or all other clients.

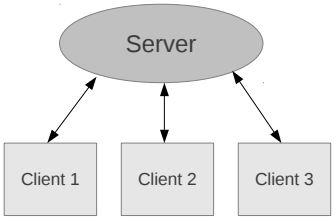


Figure 1: Client-server model

A session starts as soon as a user loads an instrument - which itself is a Pd patch with certain *netpd* specific properties - into the client. The clients keep the list of of loaded instruments synchronized among each other at any time. As necessary, clients even transfer the instruments (the .pd-files) to their peers in order

[1] abstractions are modules written in Pure Data that can be instantiated in a patch.

[2] knowing how to create Pd patches is sufficient to be able to create *netpd-instruments*

to ensure synchronicity . Any user may load more instruments into their client and these appear immediately (or after the time of transfer) in all clients.

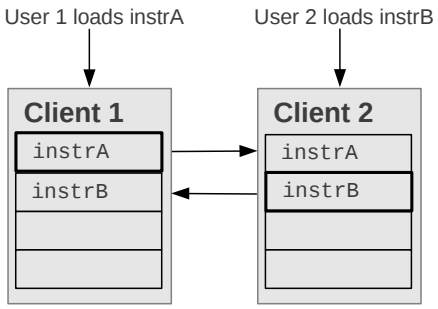


Figure 2: Instrument synchronization

Users play the instruments by manipulating their GUIs. Also the instruments keep their state synchronized among clients. Any change is immediately reflected on all clients. All users can play on all loaded instruments. Also, every user immediately experiences the manipulations of its peers. Although the generated sound is rendered on every client separately, the result is the same everywhere.

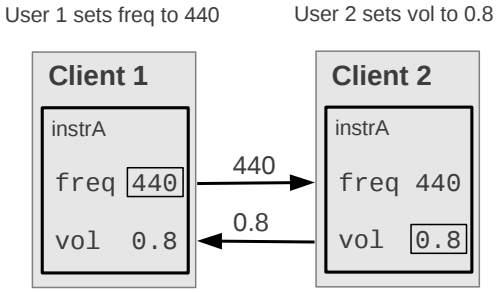


Figure 3: State synchronization

3 The framework

Let me divide the requirements of above scenario into three main tasks which will be covered separately in this document.

Obviously, the clients need a way to communicate with each other. A message protocol is defined, which the whole communication between clients (and between client and server) is based on.

Another task of the framework is to share instruments between clients and to make sure that at any given time the set of loaded instrument is synchronized among connected clients. In that respect, *netpd* acts as a peer-to-peer file sharing tool for Pd patches.

State synchronization among instances of *netpd-instruments* is a further goal of the framework. Similar to the sharing of instruments described above, state synchronicity must be maintained at any given time. Unfortunately, state synchronization does not automatically work for arbitrary patches. The use of *netpd-abstractions* facilitates the creation of state-synchronized instruments.

3.1 The message protocol

It was decided to make the communication of the framework fully based on the OSC protocol, mainly because of its flexibility and its wide acceptance in music and related fields. OSC is agnostic of the underlying transport layer. *netpd*'s requirements for reliability left TCP as the preferred transport protocol. *netpd* adheres to the version 1.1 of the OSC specification[3] which specifies SLIP[4] as a framing mechanism for stream-oriented protocols (such as TCP). Figure 4 shows how the protocols are stacked.

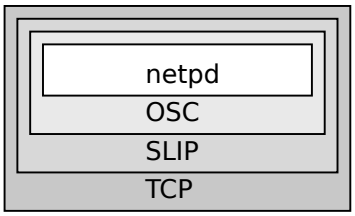


Figure 4: protocol stack

3.1.1 Receiver ID

The sole purpose of the server is to relay messages between clients. Clients may send messages either to all or to specific clients. This is achieved by defining the first field of the OSC address as the receiver ID. Table 1 shows the complete list of valid receiver IDs:

ID	Receiver
/b	broadcast (all clients)
/s	server (not forwarded)
/l	local (not sent to server)
/[ID]	client with given ID

Table 1: List of valid receivers

The /l address is used in a similar way to *localhost* in networking: /l-messages are looped back by the client. All other messages are sent to the server. The server checks the first field of

an incoming messages and forwards it accordingly. It disregards any message with an invalid receiver ID.

/b-messages are forwarded to all connected clients, even to the one the message originated from.

/s-messages are read by the server and allow the client to request certain data such as its client ID.

Any message whose first field is an integer number between 0 and 999'999 is forwarded to client with the given ID if it exists. Otherwise the message is disregarded. The server does not strip the receiver ID field when forwarding a message, instead it replaces it by the ID of the sender. This allows the receiving client to know the origin of the message.

For instance, the server receives the following message from the client 3:

```
/6/megasynth/voice1/freq 8000
```

The server replaces the receiver ID by the sender ID and forwards the message to the client 6:

```
/3/megasynth/voice1/freq 8000
```

3.1.2 Server methods

Only the first field of the incoming message is relevant to the server, except if the first field is /s. The second field of a /s-message is the namespace for optional server modules. At the time of writing none exist, so only /server is currently used. The third field specifies the type of data the client requests. Only a minimal set of client requests is supported:

/s/server	Data
/socket	socket number (client ID)
/ip	IP address of the client
/num_of_clients	number of clients
/protocol_version	protocol version

Table 2: List of supported server methods

The server responds to such messages only to the requesting client. Client requests may consist only of the OSC address, whereas the server appends the requested data to the message. A typical client request and the according server response looks like:

```
/s/server/protocol_version
/s/server/protocol_version 2 0
```

3.1.3 Interoperability

Since OSC is a standardized protocol, other applications (Pd based or not) may be used to participate in a *netpd* session as long as the messsages adhere to above definitions. Similarly, it is thinkable to intercept *netpd* traffic in order to control hardware, for instance. Altough the protocol as well as the server were designed with the presented framework in mind, those might be used solely for the purpose of sending messages between an arbitrary number of nodes in applications totally different from *netpd*. Modularity was one of the key aspects when rewriting the framwork, which should make it easy to take out and use only those parts of interest.

NOTE: so called OSC-Bundles are not supported by *netpd* and yet it is not clear if and how support for them can be added.

3.2 The client

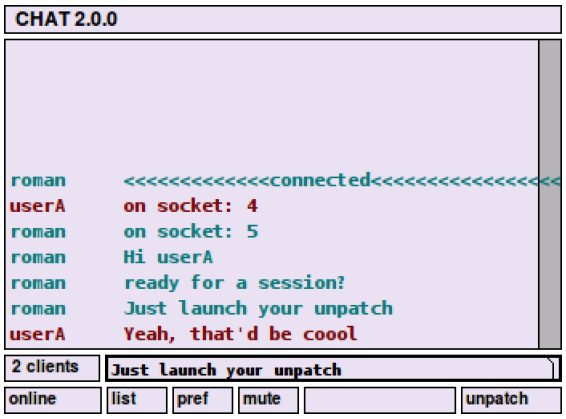


Figure 5: chat user interface

A user participates by opening the *netpd*-client's main patch called chat.pd in Pure Data. This patch immediately establishes a connection to the server and presents a user interface for chatting with other users. Being able to communicate (with words) is crucial for establishing a session. Before sending anything else though, chat checks the protocol version of the server and requests its own client ID. It also shows who is participating in the current session.

At this point the user does not yet take part in the ongoing session, but they might do so by launching unpatch (by clicking the unpatch button in chat).

Unpatch is the management interface for loading and closing *netpd-instruments*. As soon as it is launched, it automatically synchronizes with its peers and loads all *netpd-instruments*

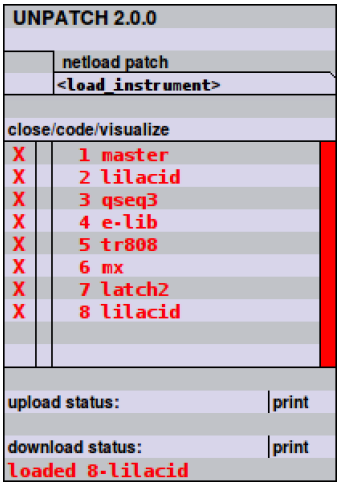


Figure 6: unpatch user interface

that are part of the running session. *netpd-instruments* that are not yet present locally are requested and transferred from the peers before loading. Once all instruments are loaded and synced, the user is able to control all instruments used in the session.

3.2.1 netpd meta tags

How does unpatch know which patches (instruments) need to be transferred? What happens when two users have different versions of the same instrument? What if a patch has dependencies, because it uses abstractions? *netpd* uses meta tags to define some properties of an instrument. A valid meta tag section in an instrument is mandatory, otherwise unpatch refuses to load the instrument. If present, those tags are read and parsed before the instrument is actually loaded. The *netpd* meta tags are organized hierarchically in subpatches and message boxes. The subpatches act as namespaces, whereas message boxes contain properties and optionally one or more values for those properties. A subpatch [pd abslist] that contains a messagebox [synthvoice(is equivalent to a messagebox containing [abslist synthvoice(. In terms of implementation, subpatches may use any depth of nesting. However, *netpd* uses one at most.

`pd NETPD 2 0` declares the section of the *netpd* meta tags. This subpatch may be placed anywhere in the instrument patch, however it is advised to put it into the main (topmost) canvas for readability. 'NETPD' is a reserved name and must not be used for any other subpatches in an instrument. The '2 0' part is optional and specifies the version of the meta tag definition. unpatch assumes the most current version if not specified.

`version 0 3 1(` is mandatory and specifies the version of an instrument. The version must consist of three integer numbers. *netpd* does not define the way they are used. When comparing two differing versions, it is only relevant for unpatch which is higher, whereas the first number is the most significant.

`pd abslist` is optional and defines the dependencies. It contains references to abstractions used by the instrument.

`synthvoice(` is the name of the abstraction and refers to a file netpd/abs/synthvoice.pd.

`singleton` is optional and defines a singleton instrument. Such an instrument may be only loaded once. Loading further instances of such an instrument is denied by unpatch.

Both instruments (patches) and their depencencies (abstractions) must contain a meta tags section. The meta tag 'singleton' only applies to instruments, since abstractions are never loaded directly by unpatch (they are instantiated within instruments). Dependencies are resolved recursively which allows to group several abstraction into a meta abstraction that holds nothing more but a meta tag section that references many abstractions (like meta packages in Debian).

Before unpatch loads an instrument, all its dependencies and child dependencies must be resolved. Instruments can only depend on abstractions, but not on other instruments, since instruments are only loaded by user interaction and never automatically. Because instruments and their abstractions are treated in distinct ways, they are saved in separate directories: netpd/patches for instruments and netpd/abs for abstractions. Only instruments that reside in netpd/patches can be loaded with unpatch.

3.2.2 Instrument synchronization

When a client loads an instrument, it notifies all other clients about the name of the instrument, its version, its dependencies and their version. Its peers check the list and issue a request to the initiating client for any item they do not have at all or whose version number is smaller.

If they find their local version of an item to be higher, they notify the initiating client about it and the user who loaded the instrument will be presented an according message ("found version 1.3.5 of abstraction 'synthvoice' on client 7"). Such a version mismatch is not resolved automatically in order to protect the user from loading a version different from the one they had in mind. The user may still decide to resolve the situation by reloading unpatch. In this case his client requests the intrument and its dependencies from a peer and the more up-to-date remote version will overwrite the local one.

When a client joins an already running session, it tries to find a peer in a 'synced' state. If there is any, it requests the list of currently loaded instruments and dependencies (with corresponding versions) from it. As soon as all instruments are loaded successfully, it marks itself as 'synced'. From this moment it will also advertise itself as 'synced' to new clients and answer instrument list requests. In case a client does not get a response to the initial request - because it is the first in the session - it sets itself to 'synced' after a timeout.

3.3 State synchronization

netpd thinks of instruments as containers of a variety of different data sets and data types that define the state of the instrument and may be modified at any time. Those data sets may be a number (changed by slider movements, for instance), a table of numbers, a list of strings, a string, a multi-dimensional number array, etc.

A user plays an instrument by modifying those data sets which in turn control the parameters of the instrument. *netpd* provides a handful of abstractions (a.k.a *netpd-abstractions*) that each cover a specific data set. Such an abstraction automatically keeps the content of a data set of a certain instrument synchronized among all clients, no matter which user is manipulating it. Depending on the network latency and the amount of data the synchronization might happen in near real-time.

All *netpd-abstractions* provide an input for manipulating data and an output for passing those modifications to the instrument. In the simplest case the modification and the data is equivalent. For instance, a user interaction changes a number that is sent to the abstraction, the abstraction stores the number and broadcasts it to all clients and finally outputs it to the instrument. Another example of a data set is a table of numbers that may be used for a table-lookup oscillator. A modification of such table can be the change of a value at a certain position, but also a change of a whole section of the table. Changing the size of the table may represent an other valid form of manipulation. The *netpd-abstraction* responsible for synchronizing tables broadcasts those modifications to all clients. When received, the modifications are applied to the table and output to the instrument. In the case of table-lookup oscillator the instrument may not need the output as it is reading the table constantly. In other cases it might be crucial to know what exactly has been changed.

If all variable parameters of an instrument are synchronized with above techniques, the generated sound (or whatever the instrument outputs) is identical for every client.

3.3.1 Namespaces

In order to allow many instances of an instrument simultaneously, each instrument is assigned a unique ID (an integer number) at loading time. Unpatch does not load instruments as stand-alone patches, but instantiates them as abstractions and gives the ID as argument. This allows an instrument to operate in its own namespace when exchanging messages with other clients. Those namespaces are used in OSC message by putting the instrument ID into the second field of the OSC address and the instrument name into the third field. Technically the instrument name is not necessary, but it makes OSC messages more human-readable. *netpd-abstractions* used within instruments operate in a child namespace of the instrument namespace.

```
  receiver ID
  | instrument ID
  | |     instrument name
  | |     |    netpd abstraction name
  | |     |        |
 /b/7/megasynth/lookup 12 0.7 0.8
```

Figure 7: OSC namespaces in *netpd*

Namespaces with more depth might be used if appropriate. A typical use case is to group many *netpd-abstractions* into another abstraction. This way a *netpd*-ified (state-synchronized) module is created that can be instantiated many times in an instrument, with different arguments for different namespaces.

3.3.2 *netpd-abstractions*

As explained in the previous section, several kinds of *netpd-abstractions* manage different kind of data sets. In order to ensure state synchronicity of instruments among clients - even if new clients join a session - each instrument must contain a special *netpd-abstraction* [netpd_head] that manages state initialization and state transfers between clients. It is kind of the master of all other *netpd-abstractions* in the instrument. Those do not send their data to the network directly, but to [netpd_head] which prepends the appropriate namespace of the instrument before forwarding it to the network. [netpd_head] also may request all *netpd-abstractions* to dump their current state when necessary.

At instrument loading time it initializes the instrument by requesting a dump which it forwards to /l (the local client [itself]). It does so in order to transfer the internal default values of all *netpd-abstractions* to the instrument. Then it tries to find a remote peer in 'synced' state. If such a peer is around, it prompts it to send back the current state of the instrument. This sets all *netpd-abstractions* of the local instrument to the current state. The same mechanism as the instrument list synchronization of unpatch is used here.

To sum it up, an instrument needs one [netpd_head] and any number of other *netpd-abstractions*:

`netpd_head $1 megasynth` is responsible for the state management of the instrument. The variable $1 is replaced by the instrument ID given by unpatch. The second argument represent the instrument name.

`netpd_f $1 volume 0.7` synchronizes a single number. Additionally, it reads the value from a GUI object (slider, number box, radio button, etc.) whose receive and send names are set to '$1-volume' and automatically updates the GUI object on state changes. The third argument '0.7' is optional and sets the init value.

`netpd_t $1 waveform 256` synchronizes a table named '$1-waveform'. The third argument sets the table size. Although it could have been designed to hold the table internally, an external table allows the use of a graphical array in the GUI of the instrument.

`netpd_r $1 something` is a simple receiver for content in the given namespace. This comes in handy when a certain data set is needed in different locations of the instrument.

`netpd_s $1 something` is the counterpart of [netpd_r]: It sends any kind of data under given namespace. It is only used in special cases, as it doesn't have any state and thus can't be used for state synchronization.

`netpd_a $1 anything` a is container for an anything message (a message with an arbitrary number of elements).

Currently there aren't more *netpd-abstractions*, though one could easily think of more data types to be synchronized. Pure Data provides only very few data types natively, so covering those in *netpd* would require additional external libraries. More *netpd-abstractions* might be added in future versions of *netpd* (for instance, for the 'matrix' type as defined by the 'iemmatrix' library).

4 Conclusions

Although all aspects *netpd* addresses seem to work flawlessly, the overall experience is not free of issues. A major culprit are audio drop outs caused by certain tasks like loading new instruments. Some causes for audio drop outs cannot be addressed by *netpd* as they are intrinsic to Pd's design. Others have been addressed by employing threaded externals. Also there are ways to work around certain causes, for instance by loading all instruments beforehand.

In past years, *netpd* enabled the creation of a community of ever varying members from around the globe. Some sessions were spanning three continents. Quite a few users wrote instruments and some more used to play with it. A huge pile of music[3] from recorded sessions grew over the years. While technically not matured, there used to be a lot of activity. After it got more quiet around *netpd*, I decided to rewrite the framework from scratch, since some design flaws became more apparent. In the meantime, Pure Data and many libraries evolved to a higer degree of maturity, which made the rewrite presented in this paper possible at all. Many instruments have been ported from the old framework and some new ones have been written. *netpd* has been "tinkered in the

[3] http://www.netpd.org/Listen

quiet" since and no community has been grown again. A few sessions with media art students revealed that there aren't any show stoppers with the framework and some people may be intrigued by playing with it. It's now time to spread the word again, which is one reason for the desire to present it at LAC 2013.

5 Acknowledgements

The companions from the early days of *netpd* - Enrique Erne, Moritz Wettstein, Syntax the Nerd - deserve credit for contributing their thoughts about design in uncountable discussions and for writing many instruments. Also, I would like to thank the authors of the external libraries that are the most crucial for *netpd* and without them the realization wouldn't have been possible: the 'osc' and 'slip' libraries written by Martin Peach and the 'iemnet' networking library written by Martin Peach and IOhannes Zmölnig. Both authors showed a great willingness to add features useful for *netpd* and to fix issues in their libraries. Collectively, I thank all people who helped realizing sessions on radio broadcasts, at concerts or other special occasions.

References

[1] Miller Puckette. Pure Data. `http://puredata.info`.

[2] Open Sound Control. `http://opensoundcontrol.org/`.

[3] Adrian Freed and Andy Schmeder. Features and future of open sound control version 1.1 for nime. In *NIME*, 2009.

[4] J. Romkey. A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP. Technical Report RFC 1055, IETF, Network Working Group, 1988.

Byzantium in Bing: Live Virtual Acoustics Employing Free Software

Fernando Lopez-Lezcano, Travis Skare, Michael J. Wilson, Jonathan S. Abel
CCRMA (Center for Computer Research in Music and Acoustics),
Stanford University
{nando|travissk|mwilson|abel}@ccrma.stanford.edu

Abstract

A Linux-based system for live auralization is described, and its use in recreating the reverberant acoustics of Hagia Sophia, Istanbul, for a Byzantine chant concert in the recently inaugurated Bing Concert Hall is detailed. The system employed 24 QSC full range loudspeakers and six subwoofers strategically placed about the hall, and used Countryman B2D hypercardioid microphones affixed to the singers' heads to provide dry, individual vocal signals. The vocals were processed by a custom-built Linux-based computer running Ardour2, jconvolver, jack, jack-mamba, SuperCollider and Ambisonics plugins and decoders among other free software to generate loudspeaker signals that, when imprinted with the acoustics of Bing, provided the wet portion of the Hagia Sophia simulation.

Keywords

impulse response, virtual acoustics, Ambisonics, auralization

1 Introduction

Acoustics are important to the experience of music: Singing in a large, stone cathedral evokes a much different response than the same singing in a small, wood frame recital hall. Acoustics are also important to the performance of music: Reverberation time can affect tempo, and room modes can influence pitch. Quite often, music written for a particular space works best when performed and experienced in that space.

As part of the "Icons of Sound" project, we are exploring the acoustics of Hagia Sophia, Istanbul, a nearly 1500-year-old World Heritage Site with marble floors and walls, 56-meter high dome and a reverberation time of over 10 seconds [1]. Hagia Sophia is presently a museum, and singing in the museum is not permitted. To better understand the aural experience of Hagia Sophia, we have attempted to synthesize the sound of Byzantine chant performed in Hagia Sophia.

Previous Icons of Sound acoustics and auralization work includes processing balloon pops recorded in Hagia Sophia into impulse responses of the space [2,3], and producing auralizations of Byzantine chant in a virtual Hagia Sophia [4]. The auralizations were accomplished by recording chant performed using headset microphones, so as to have separate dry tracks for each of the performer's vocals. While chanting, the microphone signals were processed using the estimated Hagia Sophia impulse responses, and played for the chanters over headphones to provide in real time a virtual sense of the performance space, while allowing dry vocal signals to be recorded. In post production, the recorded dry tracks were processed according to the estimated Hagia Sophia impulse responses to produce performance recordings in a simulated Hagia Sophia.

In this work, we describe a Linux-based system for live performance of Byzantine chant [10] in Stanford University's new Bing Concert Hall, modified to take on the acoustics of Hagia Sophia. Unlike virtual acoustic systems such as installed at McGill's CIRMMT [5], or LARES [6], any virtual acoustic system used at Bing must be able to be installed or removed within a few hours. For our Icons of Sound project, the system must also be able to handle the long reverberation times of Hagia Sophia.

The approach we take is to place two dozen loudspeakers in the hall, hanging many of them from the rigging points in the ceiling. Directional headset microphones are used to capture dry vocal signals, which are processed in a Linux machine to produce the needed acoustic enhancement.

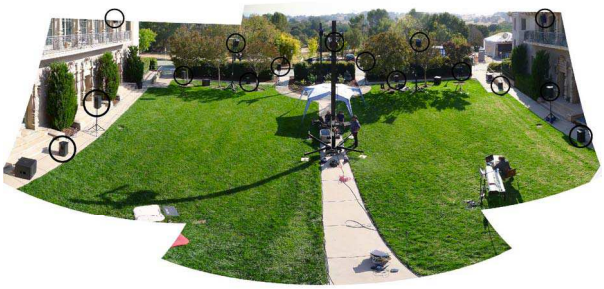
This paper is organized as follows: Section 2 describes the Listening Room and outdoor systems developed for synthesizing Hagia Sophia acoustics and listening at CCRMA. Sections 3 and 4 present the hardware and software components of the live virtual acoustics system used in Bing Concert Hall.

2 CCRMA-based systems

The auralization system used in the Bing Concert Hall performance was the result of a design and testing process lasting more than three years. The process included selecting and procuring the loudspeakers themselves, developing methods for diffusion and writing, testing and running the associated hardware and software. The real-time system was recently completed, and has been successfully used in concert several times.

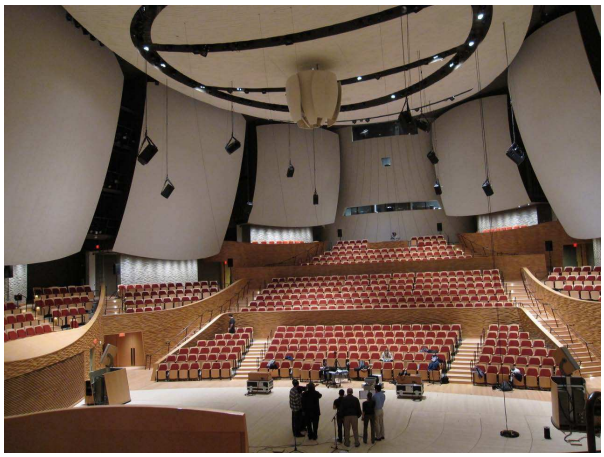
The first public performance of one of the Icons of Sound auralizations was the Prokeimenon mix which was performed in the second night of the 2011 Transitions concerts [4]. This short piece was recorded in our small recital hall called the Stage by members of the Cappella Romana early music vocal group using the process briefly described above: headset microphones captured dry vocals which were convolved with Hagia Sophia impulse responses and fed back to the performers via headphones. In a post-production process, auralizations were created using an off-line convolution process to produce reverberated tracks for an Ardour session. The original mixing was done in our 3D Listening Room using its built in Ambisonics decoder and 22.4 loudspeaker configuration. The final mix was diffused outdoors using our (at the time) 16.4 system with height and a combination of 2D and 3D Ambisonics decoders.

During the second night of the 2012 edition of Transitions [7] we did the first test of a live performance in the virtual Hagia Sophia simulation for two chanters, John Kocolas and Konstantine Salmas. Two issues were being addressed, first, with critical help from Aaron Heller and Eric Benjamin [8], an expanded 24.4 diffusion system was implemented using the 3D outdoor diffusion system with proper Ambisonics decoding. Second, we found that Countryman B6 headset microphones could provide sufficient voice signal-to-reverberation ratios that the very wet acoustics of Hagia Sophia could be simulated without feedback. The concert was successful and proved the feasibility of using the system in a concert situation.



Transitions 2012 outdoor concert setup

In December 2012 we had the opportunity to do a more complete test of the whole diffusion system in the framework of a two-day complete technical rehearsal in the Bing Concert Hall. It was the first time we had access to the Hall, and we installed a complete rigging of our 24.4 system in a full 3D dome.



Rehearsal, chanters from the Holy Cross Church

We had not tested the system in an enclosed environment, and needed to verify that we could achieve the wet signal energy of Hagia Sophia without feedback. Using a group of seven Byzantine chanters from the choir of Holy Cross Church in Belmont, we were able to produce the needed levels of reverberation without feedback, provided head-mounted microphones were used.

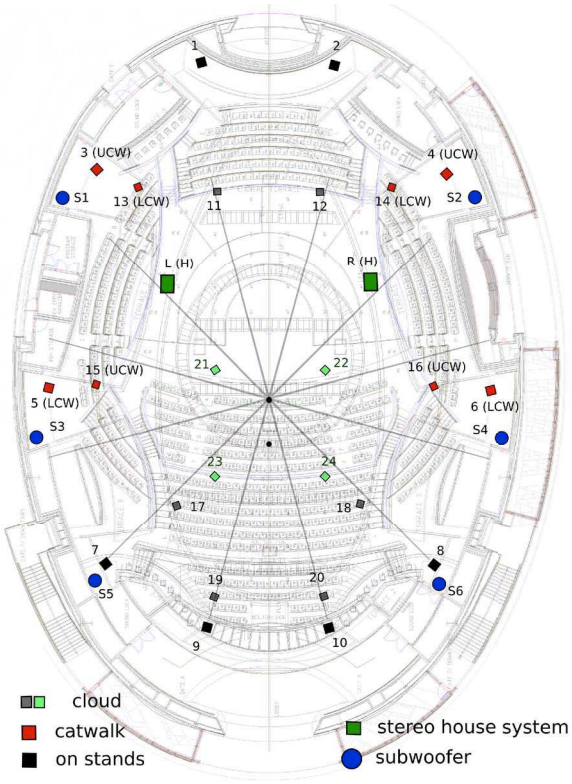
3 Speakers and rigging

The final speaker array for the concert was comprised of 24 QSC HPR122i main speakers and 6 QSC HPR181 subwoofers. 10 main speakers were arranged around the upper terrace. The hall has a terraced design with the audience seated at different levels and surrounding the stage [18].

Because of the terraced design, regretfully some of the audience would be necessarily close to a

speaker and that is a real problem. To try to minimize that problem we kept an earlier arrangement of speakers mounted in equipment catwalks that are located between some of the "sails" of the concert hall. So we had, at “ear” level, two front speakers on stands, four speakers on each side of the Hall that were elevated (in the lower catwalks) and four more speakers in the sides and back on stands. This was a compromise between ideal location of speakers and trying to not get them to be too close to audience members.

The speakers rigged from the ceiling were roughly arranged in two “rings”, a medium height one comprised of 10 speakers and a high ring of four speakers. The placement of the speakers was restricted by the spacing and availability of rigging points so it is not an exactly regular arrangement.



Aaron Heller and Eric Benjamin used their decoder design software to create an Ambisonics decoder tuned to the speaker arrangement we had in the Hall.

4 Signal routing

This section describes in some detail the signal routing of the whole diffusion setup.

4.1 Microphones

For the singers we used 15 Countryman B2D hypercardioid 2 mm diameter microphones with Sure wireless transmitters. Each singer had a microphone taped to his forehead. The wireless signals were picked up by antennas in the Hall which were connected to two racks of 8 wireless receivers each, located in the Amp/Patch Room to the side of the hall. From there they were patched into the hall's main Yamaha SL5 mixer which was used for level control and signal equalization. The 16 signals were then routed out of a dual ADAT link card into our sound diffusion workstation.

4.2 The workstation

Our workstation is a custom built no-fan workstation [11] currently equipped with a 6 core / 12 thread i7-3930K processor, 64G of RAM, an SSD system disk and a two disk mirrored RAID array for big audio and video files.

For this performance the computer had an RME RayDAT PCI Express audio interface which was slaved to the Yamaha digital mixer through Word Clock and received the microphone signals through two ADAT links coming from the Yamaha mixer. Jack 1.9.9 was used as the main audio connection and distribution system, and it was running during the performance at 256x2 and 48KHz. The preferred setting of 128x2 was actually not rock solid during tests so it was discarded, although it would have probably worked. The nature of the system being simulated did not require ultra low latency from the system (see below for the complex setup used). The computer was running Fedora 17 plus Planet CCRMA [21] and the Planet CCRMA RT patched kernel for best performace.

4.3 Signal processing

All signal processing in the computer was done with Open Source and Free Software tools. An Ardour2 [13] session plus 4 instances of jconvolver [14] created the routing and main spatialization component of the Hagia Sophia virtual acoustics recreation running a total of 48 16-second-long convolutions.

Each of the 16 microphone signals has a pre-fader send to three jconvolver inputs. Inputs 1-4 are handled by the first jconvolver instance, 5-8 by the second, etc. 48 Ardour2 bus tracks each receive one mono jconvolver output. Each has a post-fader Ambisonics 3,3 panner insert to position the sound and output to the master bus. Virtual reverb sources were positioned manually via the Ambisonics 3,3 panner plugin GUI inside Ardour [19]. Additionally, a small amount of direct signal is mixed in via post-fader

Ambisonics 3,3 panner plugins on the microphone input tracks. The project contains mix and edit groups for the dry (16) and wet (48) signals for ease of experimentation and mixing.

The resulting 16 channels of the 3rd order Ambisonics mix are then routed through Jack to the sound synthesis server of SuperCollider [16] (we were using the Supernova server running with 6 threads of parallelism, that corresponded to the 6 real cores of the machine).

SuperCollider received the Ambisonics signals and splits them into main speaker and subwoofer feeds using LR4 software crossover networks [9]. Those are sent out to two instances of Ambdec[15]. The first one decodes the main speaker feeds for our 24 QSC speaker dome, and uses coefficients calculated thanks to the help of Aaron Heller and Eric Benjamin decoder and optimization software [8]. The second decoder receives the subwoofer Ambisonics signals and decodes them with a standard hexagon decoder.

The output signals from the Ambdec decoders go back into SuperCollider, where a different set of SuperCollider instruments further process them to equalize all speaker and subwoofer feeds for delay and loudness (distances to all speakers and loudness are measured when the system is calibrated). At this point the signals for the 24 main speakers and 6 subwoofers are ready to be sent to the physical speakers.

4.4 Outputs

The workstation, which is located in the main mixing position of the Hall, is connected through a single ethernet cable to a Mamba AudioStreamer box that is located in the Amp Room. 30 of its 32 outputs are patched directly into the lines that are connected to each speaker or subwoofer.

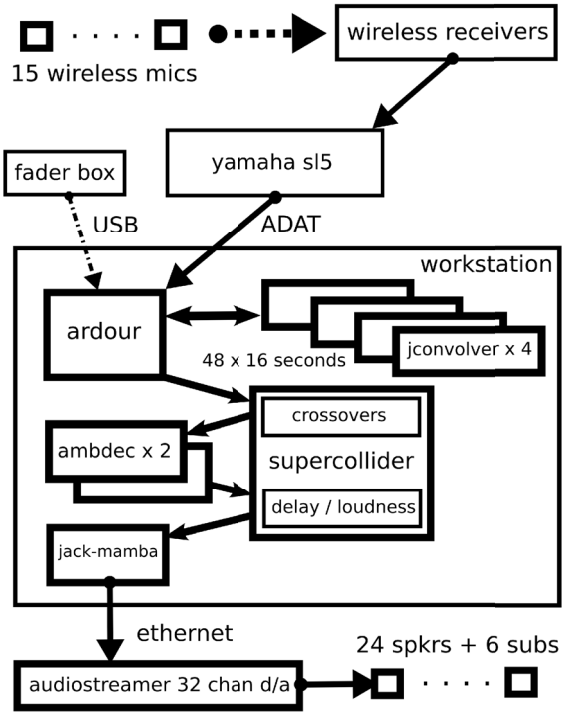
The outputs of SuperCollider are connected to the inputs of jack-mamba[12], a small jack client that translates jack frames into UDP packets that the AudioStreamer box can understand.

All of this running in realtime and providing a virtual Hagia Sophia auralization environment for the 15 singers of the Cappella Romana [17] vocal ensemble.

4.5 Control

A simple BCF2000 USB fader controller is connected to the workstation and three faders were programmed to control the 16 channel master Ardour bus (for overall level control), the DryGroup in Ardour which controls the level of the dry spatialized signal being fed to the main mix and the WetGroup which directly controls the outputs of all 48 convolvers. Those three faders were used live during the performance to control the gain of dry and wet signals and overall loudness of the virtual hall. Fernando was controlling the mix during the performance.

The microphone signals were separately controlled using the Yamaha mixer by Doyuen Ko, a member of Wieslaw Woszczyk's team that was doing the recording of the performance.



Signal and control routing

5 Conclusions

Cappella Romana's February 1, 2013, performance of Byzantine chant in a virtual Hagia Sophia demonstrated that the Linux-based virtual acoustics system described here can be used for real-time auralization. All of the hardware components used were off-the-shelf and relatively inexpensive, and all of the software was based on Open Source Free Software projects.

The concert performance garnered positive reviews for the immersive, transporting soundscape [20]. The performers also had a good experience, reporting that, unlike other

Cappella Romana concert, February 1st, 2013

performances involving artificial reverberation, the virtual Hagia Sophia created in the Bing Concert Hall "responded in a very natural way," and felt like a "real space," "holding its pitch" and allowing control over the harmonic content of the ison (drone).

6 Acknowledgements

This work would not have been possible without the help and encouragement of many individuals: Chris Chafe (CCRMA's Director), Bissera Pentcheva (who, with Jonathan Abel, leads Icons of Sound, http://iconsofsound.stanford.edu), Jenny Billfield (former director of Stanford Live), Aaron Heller for the timely delivery of optimized Ambisonics decoder coefficients, Sasha Leitman, Carr Wilkerson and Jay Kadis at CCRMA and especially to Alexander Lingas, Mark Powell and Cappella Romana, and Fr. Salmas and the Byzantine chanters from the choir of Holy Cross Church in Belmont, CA. This research was supported in part by the Stanford Presidential Fund for Innovation in the Humanities, granted for “Icons of Sound: Architectural Psychoacoustics in Byzantium”, and the Stanford Arts Initiave (formerly Stanford Institute for Creativity and the Arts, SiCa). We also want to thank Countryman Associates for donating the B2D Directional Lavalier microphones used for the concert and recording sessions. Finally, we greatfully acknowledge Christine and Reece Duca, whose generous support made possible the concert and recording sessions described here.

References

[1] Bissera V. Pentcheva, "Icons of Sound: Hagia Sophia and the Byzantine Choros," Chapter 2 in "The Sensual Icon: Space, Ritual, and the Senses in Byzantium," Penn State Press, 2010.

[2] Jonathan S. Abel, Nicholas J. Bryan, Patty P. Huang, Miriam Kolar, Bissera V. Pentcheva, "Estimating Room Impulse Responses from Recorded Balloon Pops," Convention Paper 8171, presented at the 129th Convention of the Audio Engineering Society, San Francisco, November 2010.

[3] Jonathan S. Abel, Nicholas J. Bryan, "Methods for Extending Room Impulse Responses Beyond Their Noise Floor," Convention Paper 8167, presented at the 129th Convention of the Audio Engineering Society, San Francisco, November 2010.

[4] Jonathan S. Abel, Bissera V. Pentcheva, Miriam R. Kolar, Mike J. Wilson, Nicholas J. Bryan, Patty P. Huang, Fernando Lopez-Lezcano and Cappella Romana, "Prokeimenon for the Feast of St. Basil (12th century)," from the concert "Transitions 2011, Night 1: Acousmatic Soundscapes under the stars," Center for Computer Research in Music and Acoustics, Stanford University, September 28, 2011.

[5] CIRMMT, Centre for Interdisciplinary Research in Music Media and Technology, http://www.cirmmt.mcgill.ca/

[6] David Griesinger, "Improving Room Acoustics Through Time-Variant Synthetic Reverberation," in Proc. AES 90th Convention, February 19–22, 1991.

[7] Jonathan S. Abel, Bissera V. Pentcheva, Miriam R. Kolar, Mike J. Wilson, Nicholas J. Bryan, Patty P. Huang, Fernando Lopez-Lezcano, John Kocolas and Konstantine Salmas, "Trisagion, Setting IX," from the concert "Transitions 2012, Night 2: Soundscapes under the stars," Center for Computer Research in Music and Acoustics, Stanford University, September, 2012.

[8] Aaron Heller, Eric Benjamin, Richard Lee, “A Toolkit for the Design of Ambisonic Decoders”, Proceedings of LAC2012

[9] Linkwitz–Riley crossovers:
“http://en.wikipedia.org/wiki/Linkwitz%E2%80%93Riley_filter”

[10] Cappella Romana Concert:

"From Constantinople to California", Stanford Live, Bing Concert Hall.
http://live.stanford.edu/event.php?code=CAP1

[11] Fernando Lopez-Lezcano, "The Quest for Noiseless Computers", Proceedings of LAC2009

[12] Fernando Lopez-Lezcano, "From Jack to UDP packets to sound and back", Proceedings of LAC2012

[13] Ardour2
http://ardour.org

[14] Jconvolver
http://kokkinizita.linuxaudio.org/linuxaudio/

[15] Ambdec
http://kokkinizita.linuxaudio.org/linuxaudio/

[16] SuperCollider
http://supercollider.sourceforge.net/

[17] Cappella Romana,
"http://www.cappellaromana.org/"

[18] Bing Concert Hall at Stanford
"http://binghall.stanford.edu/about/"

[19] Amb-plugins,
http://kokkinizita.linuxaudio.org/linuxaudio/

[20] Jason Victor Serinus, "Cappella Romana: Time Travel to Constantinople," in San Francisco Cassical Voice, February 2, 2013, http://www.sfcv.org/reviews/stanford-live/cappella-romana-time-travel-to-constantinople

[21] Planet CCRMA software package collection,
https://ccrma.stanford.edu/planetccrma/software/

Combining granular synthesis with frequency modulation.

Kim ERVIK
Department of music
University of Science and Technology
Norway
kimer@stud.ntnu.no

Øyvind BRANDSEGG
Department of music
University of Science and Technology
Norway
oyvind.brandsegg@ntnu.no

Abstract

Both granular synthesis and frequency modulation are well-established synthesis techniques that are very flexible. This paper will investigate different ways of combining the two techniques. It will describe the rules of spectra that emerge when combining, compare it to similar synthesis techniques and suggest some aesthetic perspectives on the matter.

Keywords

Granular synthesis, frequency modulation, partikkel, csound, sound synthesis

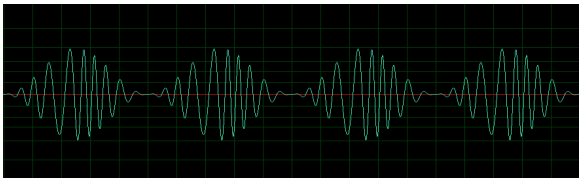


Fig 1:Grain Pitch modulation

1 Introduction

1.1 Method

Working on this subject, We´ve been using csound for implementing and generating sound, and reviewing graphical analysis of the sounds generated. We have been working systematic trying to find similarities and differences between regular FM, granular synthesis (GS), and FM in GS. For the purpose of this project, we have been using sinusoidal grains.

1.2 Granular synthesis

The idea of granular synthesis can be traced back to Gabor's theory of acoustical quantum in 1947 [7]. Thinking of sound as particles as a philosophical and a musical point of view can be very interesting, and has led to the development of granular synthesis, or particle synthesis. Granular synthesis means synthesizing sound based on adding thousands of sonically grains into larger acoustical events. Sound as particles has been used in applications like independent time and pitch scaling, formant modification, analog synth modeling, clouds of sound, granular delays and reverbs, etc [3]. Examples of granular synthesis parameters are density, grain pitch, grain duration, grain envelope, the global arrangement of the grains, and of course the content of the grains (which can be a synthetic waveform or sampled sound). Granular synthesis gives the musician vast expressive possibilities [10].

1.3 FM synthesis

Frequency modulation has been a known method of coding audio into radio signal since the beginning of commercial radio. In 1964 that John Chowning discovered the implication of frequency modulation of audio working on synthesis of brass instrument at Stanford [2]. He discovered that modulating the frequency resulted in sideband emerging from the carrier frequency. Yamaha later implemented the technique in the hugely successful DX7.

If we consider FM synthesis using sinusoidal carrier and modulation oscillator, the spectral components present in a FM sound can be mathematically stated as in figure 2.

$$f_n = f_c \pm n f_m$$

Fig 2:The spectral components present in a FM sound where n is an integer, fc is the carrier frequency and fm is the modulation frequency.

The intensity of the sidebands can be calculated with Bessel functions. When a sideband passes 0 Hz or the nyquist frequency, it is mirrored and 180 degrees phase shifted. The modulation frequency is usually a product of a modulation ratio and the carrier frequency, while the modulation amplitude is the product of a modulation index and the carrier frequency, which sets the sidebands in a constant relation to the carrier frequency [9].

FM in GS is earlier mentioned in Jones and Parks (1988) as a method to increase the range of

possibilities available to the composer or researcher using granular synthesis [8].

2 Combining the two techniques

2.1 FM of GS parameters

In granular synthesis the pitch interpretation of the sound is dependent on the grain rate. If the grain rate is within the audible range, the pitch is given by the grain rate[1]. If the grain rate is sub audio, the pitch of the source waveform within the grain defines the pitch of the generated sound. We can do both FM of the grain rate, and FM of the pitch within the grain. We´ve looked at different variants and investigated the differences in spectrum.

The grain pitch affects the spectrum of a GS tone with high grain rate. When the grain rate is low the spectrum might be affected by the grain rate. So there are two parameters that oppose each other. But seen as one, the two parameters controlling pitch and timbre are grain rate and grain pitch, and that's why we have tried to modulate the two.

If we look at regular FM synthesis, the perceived pitch is dependent on the modulation ratio. If the FM ratio is 0.5 the pitch is interpret as one octave below the carrier. The distance between the frequency components of the spectrum is an important part of determining the perceived pitch. If a tone is generated with FM ratio at 7/8, the timbre will be inharmonic resulting in a diffuse fundamental pitch. As shown above in both FM and GS we have many different factors contributing to the perceived pitch.

To sum up we have two different ways of combining FM with GS:

- Frequency modulation of the pitch within the grain
- Frequency modulation of the grain rate

3 Frequency modulation of the pitch within the grain

Grain pitch modulation is easily done in csound using the partikkel opcode [1]. The particle opcode has an a-rate input that alters the pitch of the grain, implemented as phase modulation, directly modifying the reading position of the source waveform. This input can be fed with any audio signal, for our experiments, we have used a sine wave signal. A modulation oscillator can be set up as in regular FM synthesis using the parameters FM ratio and FM index, although the calculation of modulation index is done somewhat differently than in regular FM [4].

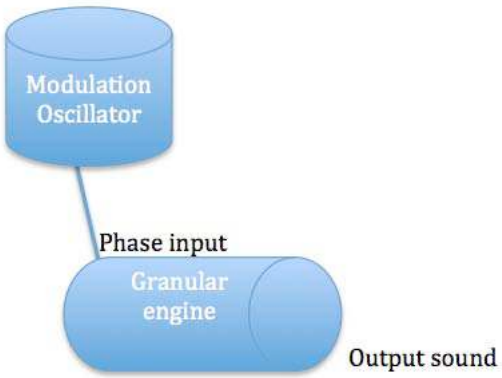


Fig 3: Grain pitch FM

3.1 Grain pitch modulation with modulation frequencies below 30 Hz

If we modulate the frequency at a rate below 30 Hz in a regular FM synthesis setup, the result is vibrato. The same thing happens in grain pitch modulation with grain rates below 30 Hz. With high grain rates the situation is somewhat different, as the grain pitch will affect the timbre. This means that a modulation frequency below 30 Hz applied to grain pitch modulation using high grain rates will lead to periodic spectral sweeps similar to filter sweeps.

There is an interesting transition area (approx. 20 – 50 Hz) both for grain rate and modulation frequency. In this area we move from a pitch perception defined by the grain pitch to the grain rate, similarly we move from vibrato to FM. We also note significant interaction between these parameters (grain rate and modulation frequency), so the situation is somewhat complex. Further exploration of this area might prove fruitful.

3.2 Grain pitch modulation with modulation frequencies above 30 hz compared to regular FM synthesis.

Since the grain rate constitutes the perceived pitch at high grain rates, it might be convenient to calculate the modulation frequency as a relation between the grain rate and the modulation ratio. Calculating modulation frequency based on a fixed pitch within the grain gives rather arbitrary spectrum. But with modulation frequency

[1]With some exceptions. For instance if the phase or the reading position of the grain source is modulated, the pitch may be defined by the pitch of the content in the grains. Another exception is if the grain stream is asynchronous. Then the result will be noisy dependent on the random range of the grain distribution.

calculated based on the grain rate, a constant relation between the fundamental pitch and the timbre is obtained.

The spectral behavior in grain pitch modulation is similar to regular FM. It might be relevant to compare grain pitch modulation with regular FM using a non-sinusoidal carrier wave. In that example all the harmonics get sidebands, resulting in very dense spectra's. The same thing happens in grain pitch modulation. There are three major differences in the spectral behavior. First of all the FM index is behaving differently. In grain pitch modulation the strength of the sidebands is weaker than those in regular FM.

Secondly simple FM ratios behave differently. The spectra of a regular FM tone will only contain even harmonics if the modulation ratio is two. But in grain pitch modulation with integer FM ratios, for instance 2, the sidebands line up with every other harmonics of the grain rate resulting in both odd and even harmonics (se figure 4). This is not the case though with complex FM ratios. FM ratio at for instance 0,5 will cause the sidebands from the frequency modulation to line up between the harmonics of the grain rate resulting in a drop in the perceived pitch (se figure 4). In some cases complex FM ratios might give inharmonic spectra, although the spectral behavior is different than in regular FM because of the dominant harmonics of the grain rate, which are still present.

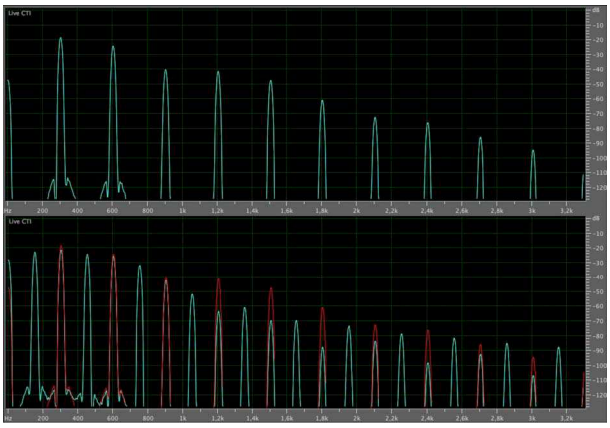


Fig 4: Grain rate at 300 Hz, grain pitch at 300 Hz. Shows the difference between grain rates at 3 and at 0,5.

The third difference is that in grain pitch modulation sidebands will also emerge from a sub harmonic tone at 0 hz, which is a sub harmonic from the grain rate in the GS. It is not audible, but the sideband might be. This results in a much fatter tone than in regular FM.

To understand the effect of the grain pitch that are modulated, we must look at FOF synthesis, a variant of particle synthesis. FOF synthesis is a formant synthesis where the grain rate gives the fundamental pitch and the grain controls formant frequency. The shape of the formant is given by the grain shape and duration. The same thing happens in grain pitch modulation, only with a much wider peak in the spectrum. It's so wide that it's perhaps not correct to call it a formant. This enables us to shift the spectral energy of the FM synthesis upwards.

If one uses Gaussian grain envelope and the grains don't overlap, the result is similar to amplitude modulation. One very important difference is that the phase of the wave inside the grains are resetting for each grain. That's means that if the grain frequency is not in an integer ration to the grain size, the wave cycles inside the grains will be cut short, resulting in a spectrum with a lot more components than in the case of integer ratios. So for cleaner sound one need the grain frequency to be an integer of the grain size, which limits the number of available grain frequencies. The way to solve this is to use two grain streams, each containing grain pitches at integer ratio to the grain rate, and then crossfade between them.

3.3 Similarities and differences between grain pitch modulation and regular granular synthesis using FM synthesized sound as wave source.

It might be useful to compare grain pitch modulation with regular granular synthesis using an FM synthesized sound as source for the grains. The major difference is the continuity of the phase in the FM synthesized source wave, whereas the phase will be reset for each new grain when doing FM on the pitch within the grain. In other respects, grain pitch modulation is comparable to combining FM with AM.

4 Frequency modulation of the grain rate

Since the perceived pitch is given by the grain rate with grain rates above 30 hz, its interesting to experiment with modulation of the grain rate, using a regular sine modulator. The modulation frequency can be calculated the same way as in grain pitch modulation. When the grain rate is modulated, it should be updated at audio rate. Ideally the grain rate and the modulation wave should be an a-rate variable. This is possible with the partikkel opcode [5].

4.1 Grain rate modulation with modulation frequencies below 30 hz

If we use modulation frequency below 30 hz we will get vibrato, just like in grain pitch modulation with low grain rates. But also here there are some interesting transition areas to explore in both grain rate and modulation frequency.

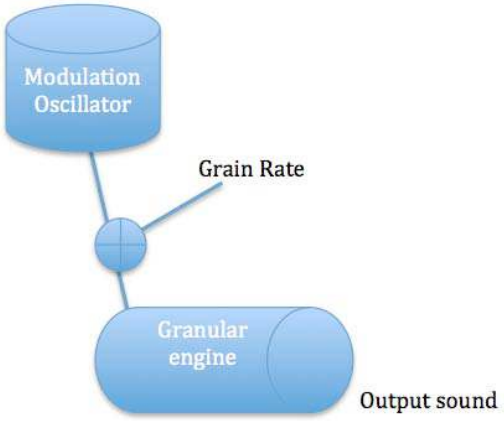


Fig 5: Grain rate modulation

4.2 Grain rate modulation with modulation frequencies above 30 hz compared to grain pitch modulation

This implementation gives similar spectral behavior as grain pitch modulation, except when the modulation index is above 1. The way we have used modulation index here, we have calculated it with a reference to the grain rate. In this respect, the modulation index is a measure of grain displacement within 1/grain rate seconds. If the modulation index exceeds one, the modulation wave leads to a negative grain rate. This allows extra grains to be generated, as the grain rate is mirrored around 0 Hz. This leads to an alteration of the perceived pitch.

Just as in grain pitch modulation, we already have harmonics from the granular synthesis before we start modulating the grain rate. This results in a perceived pitch that never can be higher than the fundamental of the grain rate. But in grain rate modulation the timbre don't get brighter with integer FM ratios above 1. This can be explained by looking at the grain rate as a sampling frequency. The modulation can only occur within the sampling frequency. According to the nyquist sampling theorem the highest frequency representable is ½ the sample rate. So the timbre of a tone with FM ratio at 0.4 is the same as the timbre of a tone with FM ratio at 0.6.

Of course even though the relation of the harmonics are similar in the two variants, the differences in the strength of the content result in sounds that have different tonal quality. That goes for grain pitch modulation versus regular FM synthesis as well.

5 Grain rate and grain pitch modulation

The two techniques described above can also be combined. This can be seen as two modulators in parallel. If one use grain pitch modulation ratio at ½ and grain rate modulation ratio at 1/3 one get a perceived pitch equivalent to modulation ratio at 1/6 [6]. Of course there are some differences in the strength of the sidebands leading to slightly different sounding timbres.

6 Other considerations

Because of incomplete wave cycles inside the grains, the spectra stretch further up in the frequency range than in regular FM. This results in a lot of aliasing. A suggested solution to this distortion is to use higher sample rates, or to implement the suggested solution containing two grain generators described earlier in this paper.

6.1 What Granular synthesis brings to FM

FM synthesis is a very flexible synthesis technique with a wide range of different sounding spectra. It has defined an entire decade of contemporary music via its widespread commercial use in Yamaha synthesizers. When combined with granular synthesis there are new unheard sounds and possibilities. Now you can create a Chowning brass sound, with overlapping grains, or with steeper grain envelopes. It is possible to make bells with grains panned in all directions and much more. Thinking of the sound as particles instead of as waves results in new ideas and alters the way we are creativ.

6.2 What FM brings to granular synthesis

To get a synth inside your granular engine enables us even more control compared to using a prerecorded sample as source inside the grains. One very interesting thing is the shift of the spectra when the carrier pitch within the grain is altered. For instance, for making new aggressive bass sounds one could control the center pitch inside the grain with an ADSR envelope. One could also make edgy sweep pads with a LFO on the center frequency.

7 Conclusion

We have investigated two different ways to combine FM synthesis with granular synthesis, compared them, and presented our results. The

two are grain rate modulation and grain pitch modulation. They behave similar to regular FM, but there are some differences in the strength of the sidebands, and in the spectra due to harmonics from the grain rate. We have also suggested some aesthetic perspectives on the combination of the two techniques.

8 Acknowledgements

Thanks to Sigurd Saue and Victor Lazzarini for help resolving technical issues.

9 Bibliography

[1] Brandtsegg, Ø. Saue, S. and Johansen, T. (2011) Particle synthesis, a unified model for granular synthesis. Linux Audio Conference 2011.
http://lac.linuxaudio.org/2011/papers/39.pdf

[2] Chowning, John M. (1973) The synthesis of complex audio spectra by means of frequency modulation. Journal of the audio engineer Society 21.

[3] Ervik, K. Brandsegg, Ø. (2011) Creating reverb effects using granular synthesis. Csound Conference 2011.

[4] Ervik, Kim csound exsample nr1: FM_Grain_Pitch.csd
http://folk.ntnu.no/kimer/LAC2013/FM_Grain_Pitch.csd
http://folk.ntnu.no/kimer/LAC2013/PartikkelArgs.inc

[5] Ervik, Kim csound exsample nr2: FM_Grain_Rate.csd
http://folk.ntnu.no/kimer/LAC2013/FM_Grain_Rate.csd
http://folk.ntnu.no/kimer/LAC2013/PartikkelArgs.inc

[6] Ervik, Kim csound exsample nr3: FM_Grain_Rate_and_Pitch.csd
http://folk.ntnu.no/kimer/LAC2013/FM_Grain_Rate_and_Pitch.csd
http://folk.ntnu.no/kimer/LAC2013/PartikkelArgs.inc

[7] Gabor, Dennis (1947) Acoustical quanta and the theory of hearing. Nature 159

[8] Jones, D. Parks, T (1988) Generation and Combination of grains for music synthesis. Computer Music Journal, Vol 12, no 2.

[9] Manning, Peter (2004) Electronic and computer music. Oxford University Press, New York.

[10] Roads, Curtis (2001) Microsound. MIT Press, Cambridge, Massachusetts.

SuperCollider IDE: A Dedicated Integrated Development Environment for SuperCollider

Jakob Leben
Koper, Slovenia
jakob.leben@gmail.com

Tim Blechmann
Vienna, Austria
tim@klingt.org

Abstract

SuperCollider IDE is a new cross-platform integrated development environment for SuperCollider. It unifies user experience across platforms and brings improvements and new features in comparison with previous coding environments, making SuperCollider easier to begin with for new users, easier to teach for teachers, and more efficient to work with for experienced users. We present an overview and evaluation of its features, and explain motivations from the point of view of user experience.

Keywords

SuperCollider, cross-platform, edit, code, GUI

1 Introduction

SuperCollider [McCartney, 2002] is a computer music system that was originally developed by James McCartney in the 1990s for Mac OS and has been ported to Linux and eventually Windows after it was open sourced in the early 2000s. It is a modular system based on an object oriented programming language (sclang) and a separate audio synthesis server (scsynth)[1].

1.1 History of SuperCollider and its Coding Environments

SuperCollider is heavily influenced by Smalltalk and was originally using a similar programming model: it strongly coupled the interpreter with the development environment. This integrated programming environment, commonly referred to as **SC.app** was developed specifically for Mac OS and therefore was not portable to other platforms. Nevertheless, it has been preserved and evolved throughout the development of SuperCollider to date, and is still in very wide use.

When porting SuperCollider to Linux, Stefan Kersten implemented **scel**, a SuperCollider editor mode for Emacs [Kersten and Baalman, 2011], which had been the most feature-rich solution for a long time, as it not only supported syntax highlighting, but also some introspection, a limited form of method call assistance and support for the old HTML-based help system.

At the moment, two other editor extensions are part of the official SuperCollider distribution: **scvim** (for vim) and **sced** (for gedit). Before developing the SuperCollider IDE, one of the authors of this paper also developed an extension for Kate (**scate**).

Apart from that, there have been other coding environments, either incomplete or not maintained anymore: **scfront** (a Tcl/Tk based editor), **qcollider** (a Qt-based editor) and extensions for the squeak Smalltalk environment, the TextMate editor, Eclipse and probably others [Kersten and Baalman, 2011]. A python-based editor called **PsyCollider** [Fraunberger, 2011] had first been distributed with the Windows port of SuperCollider, but later removed from distribution, as the code was unmaintained, unstable and made obsolete when gedit and sced were ported to Windows.

1.2 Motivation for the New IDE

The negative aspects of the situation prior to SuperCollider IDE may be summarized as follows:

- The user experience vastly differs among the different programming environments.
- No existing environment is working out of the box on every supported operating system.
- Some environments (e.g. scvim or scel) are based on editors that are not very accessible for beginners.

The lack of a single cross-platform coding environment is a disadvantage (particularly for ed-

[1] A multiprocessor-aware alternative to scsynth is supernova [Blechmann, 2011]

ucation of new users), because it renders impossible the exchange of experience among people who are forced to use different environments according to what is available for their operating system. Moreover, each programming environment has to be maintained separately, and long-term maintenance turned out to be a problem. The scarce development resources are spread among different projects instead of focused on a single system.

In late 2011 the authors therefore decided to start the development of a new IDE dedicated to SuperCollider (not merely an extension of a general-purpose code editor). The goal was to address all of the above issues by ensuring a unified user experience across all supported platforms and making the IDE both easy to use for beginners and powerful enough so that experienced users would not feel the need to switch to an advanced editor like Emacs.

The choice of Qt as the underlying GUI framework for the IDE came naturally, as one of the authors had previously reimplemented the GUI programming classes of the SuperCollider language itself using Qt, which turned out to be quite a success.

2 Overview of the new IDE

2.1 System Architecture

Since an IDE demands a tight integration with the target programming language, the question was raised immediately whether the new IDE should be coupled with the language interpreter into one process, as is the case in SC.app, or rather a separate process, as in existing editor extensions.

Consideration of benefits and drawbacks of the two options brought decision in favor of separating the IDE from the interpreter: the most important benefit of this strategy is that the decoupling allows the IDE to survive potential crashes of the interpreter, and maintain responsiveness and control in case running some SuperCollider code locks up in an infinite loop.

The major drawback of decoupling is increased effort for inter-process communication (IPC) with the interpreter. However, scel has proved that a powerful set of features may be built on top of IPC, and hence this did not outweight the benefits of decoupling.

2.2 Graphical Interface

Thanks to the Qt GUI framework, the appearance and behavior of the GUI is largely equal across supported platforms. Figure 1 shows the default appearance on Ubuntu.

The IDE has a single-window design - it features a single code editing widget at the center of the main window. Tabs are used to switch between multiple open documents. The editor widget can also be split horizontally and vertically to show more than one document at a time.

Below the code editor, there is an area where various tool panels are displayed on request via keyboard shortcuts:

- Find/Replace: a standard tool for finding and replacing text in the current document, supporting regular expressions and backreferences in replacement
- Go-To-Line: a standard tool to quickly jump to a line in the current document, by line number
- Command Line: a tool for one-line SuperCollider expressions to be evaluated, featuring history

Along the edges of the main window, there are *dock areas*, where other *dockable widgets* may be placed:

- Integrated help browser
- Document browser
- Language interpreter output panel

These widgets can be easily drag-and-dropped to different locations in the dock areas, either side-by-side, or stacked on top of each other (with tabs appearing to switch among the stacked widgets). They can also be undocked and moved out of the main window (e.g. to place them on a second screen etc.), or simply hidden.

The status bar on the bottom of the main window is used to show the state of the language interpreter and the default synthesis server. The server status box is a compact alternative to the SuperCollider server window, showing status information like CPU utilization, number of running synths, groups, synthdefs etc.

3 Interaction

Our guidelines in interaction design were to minimize the amount of constantly visible controls, so as not to clutter the GUI, but to make the most used functionality quickly accessible via keyboard shortcuts, and advanced features

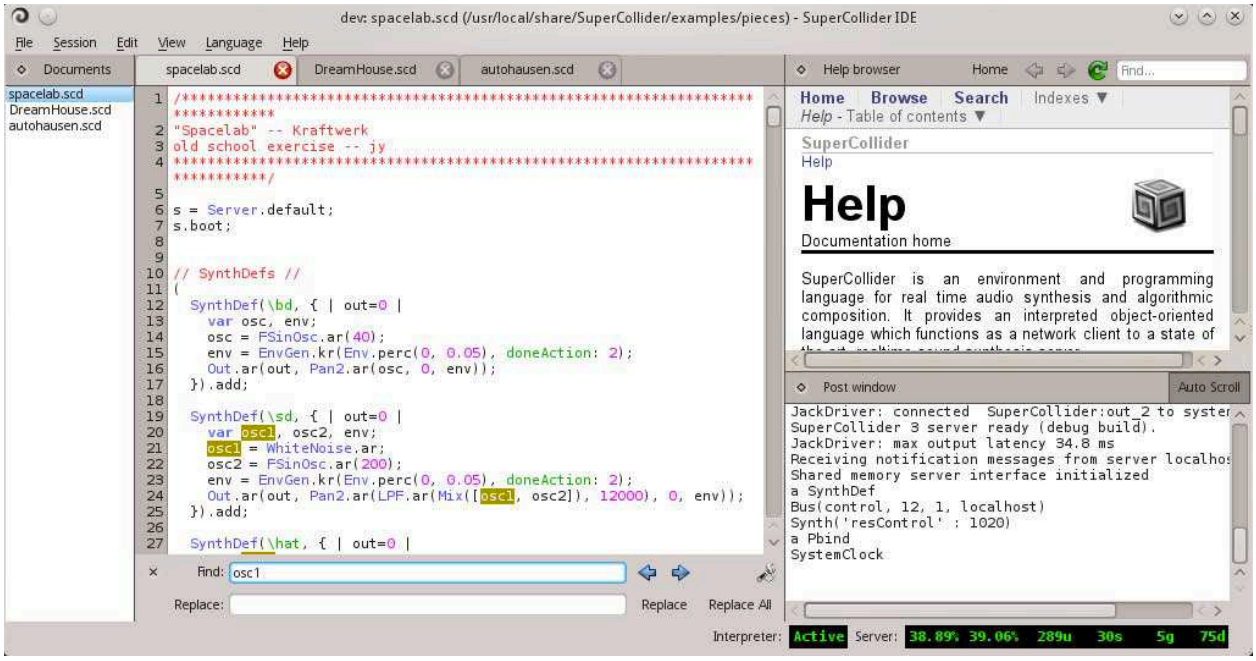


Figure 1: SuperCollider IDE on Ubuntu

easily discoverable via the main menu and *context menus* - i.e. menus that pop up when right-clicking (or Ctrl-clicking) on a GUI element and offer a choice of actions relevant for that element. To combine accessibility and discoverability the following rule is applied: as much functionality as possible is in the main menu, and each item in any menu may be assigned a shortcut.

We distribute the IDE with a large set of default shortcuts that cover most frequently used functionality by both SuperCollider beginners and experts, and try to adhere to shortcuts in other coding environments.

3.1 System Control

The language interpreter is started automatically with the IDE. Nonetheless, it can be stopped and restarted at will via the main menu or shortcuts, which is useful if code gets stuck in an infinite loop, or the interpreter simply crashes and stops by itself.

The audio server, on the other hand, is not started automatically, but can be quickly started using a shortcut or the main menu. The menu includes other audio-related actions: to dump the node tree, show sound level meters and the like. All these actions may also be accessed via the context menu associated with the audio server status box (see section 2.2 about the status bar).

3.2 Code Evaluation

Code evaluation is, naturally, the most valuable functionality of a SuperCollider coding environment, and making it as practical as imaginable is of highest importance.

All existing coding environments support evaluating a line of code using a keyboard shortcut without the need to select the line. Moreover, since SuperCollider code is often evaluated in groups of lines, there is typically support for enclosing such groups in parenthesis, then double-clicking one of the parenthesis to select the contents in order to evaluate them. Such groups of lines are commonly called *regions*.

Like scel has done previously, SuperCollider IDE goes a step further by automatically detecting the region enclosing the text cursor, so it can be evaluated with a shortcut without the need to select it. The evaluation behavior is intelligent: it will evaluate either the selection (if any), or the current region (if any), or the current line - where *current* means 'at the position of the cursor'.

Due to automatic region detection, large portions of code may be evaluated without selection. However, without any visual indication, this could easily create confusion and uncertainty as to what code has been evaluated. Hence, another very useful feature has been implemented: evaluated code is highlighted, and then the highlighting gradually fades away. An

additional benefit of highlighting is in demonstration scenarios: not only the demonstrator, but the audience as well knows exactly what code is evaluated, and when.

4 Code Editing

It is our goal for SuperCollider IDE to implement code-editing assistance on the level of support that general-purpose IDEs offer for most widely used programming languages. Namely, we consider the crucial features: syntax highlighting, automatic indentation, automatic code completion and method call assistance.

4.1 Syntax Highlighting

Existing SuperCollider editor extensions typically reuse generic support of their host editors for on-the-fly syntax highlighting. SC.app, albeit the oldest and most widely used environment, only updates highlighting on explicit request via the user interface.

Syntax highlighting in SuperCollider IDE has been implemented to update on-the-fly, and in a very efficient manner to never interfere with code typing. Attention was paid to strictly match the lexical rules obeyed by the SuperCollider language compiler. As a result, we have most efficient and correct syntax highlighting for SuperCollider language to-date.

4.2 Automatic Indentation

The IDE automatically indents code while typing, trying to mimic the most common ways people would indent code by hand. Automatic indentation may also be invoked explicitly for a selection of lines.

Automatic indentation is done on the basis of opening and closing brackets. When a line break is entered, the new line is indented by one level if the previous line contains any opening brackets that are not matched with a closing bracket on the same line. Whenever a closing bracket is typed on a subsequent line, a previous line containing the matching opening bracket is searched for, and if the matching brackets are the first and the last ones on their lines, respectively, the current line is made to match indentation of the line above. For example:

```
(
p = Pseq([
    Pbind(
        \degree, Pwhite(0,5,5),
        \dur, 0.1
    ),
    Pbind(\degree, Pseq([6,7]))
], inf)
)
```

As shown above, *regions* (see section 3.2) do not contribute to indentation, as is common in SuperCollider code.

One current issue with automatic indentation remains to be addressed: indentation of line continuations. It is common to have one expression extend over several lines; in this case, it is typically desired to increase indentation on all but the first line. For example:

```
In.kr(4, 2)
    .lag(0.3)
    .linexp(0, 1, 10, 1000)
```

This is currently not implemented yet; a solution will require enhanced grammatical analysis.

4.3 Automatic Completion

Automatic code completion (autocompletion) consists of offering the user a selection of possible continuations of text being typed, based on context.

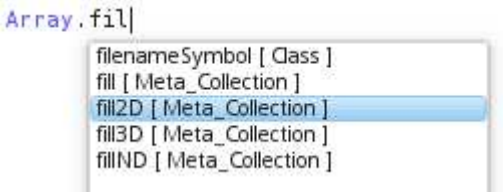


Figure 2: Autocompletion in SuperCollider IDE

As a weakly-typed programming language, SuperCollider poses limitations on the possibilities of autocompletion, compared to strongly-typed languages (e.g. C, C++). Namely, it is not always possible to infer the type of a variable identifier, and hence the set of its methods. We have worked in SuperCollider IDE towards offering completion as far as possible within these limitations.

Autocompletion is offered in the following cases:

- Class names:

  `Sin<...>`

  Since class names exclusively begin with an uppercase letter, it is straightfoward to complete them from the set of all classes.

- Method names following class names:
  `Array.<...>`
  They are completed from the set of class methods of the readily-available class.
- Method names following literals and built-ins
  `123.<...>`
  `topEnvironment.<...>`
  They are completed from the set of instance methods of the class inferred from the literal or the built-in.
- Method names following a variable name:
  `func.<...>`
  The class is not inferred, so the method is completed from the set of all methods of all classes.

Completion of methods of known classes starts immediately when the dot '.' is typed. One exception to this is the case of methods of Integer literals: it only begins after 1 character has been typed, or else redundant completion would be triggered on a dot in a Float literal, which proved to be a rather annoying experience.

In other cases the list of candidates may be quite large (the set of all classes, or all methods of all classes), hence completion only starts after 3 characters have been typed.

Although the current code base would easily support completion of built-ins (e.g. `topEnvironment`) and method names in functional notation (e.g. `min(1,2)`) we have decided to avoid that. The reason is that, formally, those cases would compete with other cases for which we currently do not offer completion: e.g. variable names in scope. It has been argued by one of the authors that autocompletion options may be understood (especially by novices) as the set of all and the only allowed options in a specific context, and hence misleading when incomplete.

The completion menu is hidden if the currently typed text matches one of the options exactly. In that case, the user's intention has likely been met, so the menu would only present an obstacle to changing activity: evaluating code, moving to another position in code, etc. However, this has been a point of debate, as it would be possible to automatically detect the change of activity and close the menu.

Although different aspects of usability often demand trade-offs, we will continue to refine the behavior so as to maximize usefulness and intuitivity of autocompletion.

As already noted, there is potential to improve the domain of autocompletion to include:

- Variables in scope:
  `var abcdef; abc<...>`
- Inferring class of Array and Event literals:
  `[1,2,3].<...>`
  `(freq: 321).<...>`
- Inferring class of variables by assignment
  `x = [1,2,3]; x.<...>`

4.4 Method Call Assistance

Method call assistance involves displaying a list of argument names and their default values, to aid entering expressions for arguments in a method call.

Figure 3: Method call assistance in SuperCollider IDE

It is implemented both for receiver notation as well as functional notation. In functional notation, an argument is prepended to denote the receiver of the message.

The assistance is invoked when a relevant opening bracket '(' is typed, or a comma ',' is typed to separate arguments, and additionally with a keyboard shortcut when the text cursor is anywhere within the brackets surrounding the arguments.

This assistance is subject to the same limitations as autocompletion, due to a weakly-typed language: to disambiguate the method, its owner class must be known. However, we have found a pragmatic solution: where the class can not be inferred, we let the user pick a class via a pop-up menu.

Hence, the following examples will offer assistance directly:

```
SinOsc.ar(
123.forBy(
```

...while the following will first display a list of classes that implement the method, then offer method call assistance once a class is selected:

```
min(
x.play(
[1,2,3].inject(
```

There is one special case in SuperCollider language where the method name is not explicit, namely an opening bracket immediately following a class name:

```
Synth(
```

In this case, the class method 'new' is implied, and SuperCollider IDE takes this into account and offers appropriate assistance.

Once the assistance is invoked, the name of the current argument being typed is highlighted, which is of great help when the number of arguments is large, or the expression for an argument is very long. Moreover, one can quickly insert and cycle through available argument names with a press of the Tab key, in order to realize argument addressing by name, as in:

```
SinOsc.ar(456, add: 1, mul:
```

Once assistance has been activated for a particular method call, it remains active in the background while assistance for a nested method call is being performed: when the user finishes typing the inner call, assistance is automatically displayed for the outer call again. This is especially useful in case assistance is based on explicit class selection (as explained above) - the selection is remembered during nested assistances so that method disambiguation does not need to be repeated.

As can be seen from examples above, this assistance would also benefit from increased ability to automatically infer classes from text. Nevertheless, the described solution via explicit class selection will remain to be useful where the intended method is absolutely ambiguous.

4.5 Editing Shortcuts

Akin to powerful general-purpose development environments, SuperCollider IDE provides a set of actions that help navigate and edit code and can be assigned arbitrary keyboard shortcuts.

Cursor movement actions include:

- Jump to next or previous empty line
- Jump to next or previous bracket
- Jumping to next or previous region

Editing actions include:

- Move current line up or down
- Copy current line up or down
- Comment or uncomment current line or selection

The comment/uncomment action intelligently uses either the single-line or the multiline comment syntax, whichever is more appropriate for the current selection.

5 Class Library Navigation

Within the SuperCollider community, the border between system developers and users has always been quite fuzzy. Furthermore, writing musical code often involves development of classes for purposes of a specific musical task and for personal class libraries. Jumping from code that uses a class to code that implements it is hence a frequent need.

The SuperCollider language interpreter has since the beginning featured introspection into where each class and method is implemented, and referenced within the class library. Existing development environments have already harnessed these capabilities to offer navigation between usage and definition via GUI.

SuperCollider IDE attempts to exploit these capabilities in most practical ways. Handy shortcuts will pop up a dialog that lists all methods whose name matches the text under cursor, or all methods of the class under cursor. Pressing Return on an entry will open the file at position where the selected method or class is implemented. The same dialog contains a search field which can be used to search for any class or method. An equivalent dialog is implemented also for class and method references: the listing contains all methods that contain references to another class or method.

The shortcuts and menu actions that bring up these dialogs work just as well in the code editor, as in any other GUI element that may contain code: the command line, the post window, and the help browser. Moreover, invoking help-related shortcuts within these dialogs will navigate the help browser to the help page related to the class or method selected in the dialog. Help and class library navigation are thus very efficiently linked.

6 Help

Recently, the traditional HTML-based help system has been superseded by **SCDoc**, authored by Jonatan Liljedahl, where help documents are written in a markup language developed specifically for this purpose and rendered to HTML on demand. SCDoc also monitors the filesystem for changes and updates its internal index of available documents at runtime. The benefits

are:

- Content is separate from style; consistent style can easily be applied to all documentation.
- Content may potentially be rendered to other formats than HTML, by implementing different rendering components.
- Due to on-demand rendering and filesystem monitoring, documentation served through the system is always up-to-date with respect to installed documents.

Interaction with SCDoc's on-demand rendering has previously only been implemented within the SuperCollider language, using its internal GUI capabilities. The SuperCollider IDE is the first code editing environment to integrate the new help system into its own GUI. There are two major benefits:

- Tighter integration with all the GUI components.
- The last displayed document and the entire browsing history is preserved across class library recompilations and interpreter restarts.

The help browser comes in form of a dockable widget (see section 2.2). When the user requests help using a related shortcut or menu action, on-demand rendering is performed via the SCDoc system, and the resulting HTML document is displayed in the help browser.

The help system is tightly connected to many GUI components: the help shortcut will recall a relevant help document for the text under cursor, when it is invoked within the code editor, the command line, the post window, the help browser itself, or - as noted above - for the selected entry in the class and method implementation and reference dialogs. Example code in help documents may also be evaluated. Another benefit of integration with the IDE is that the shortcut for evaluation is identical to the one in the code editor, even when customized by the user. Moreover, the same shortcuts as in other GUI components may be used for class library lookup (see section 5).

7 Sessions

A *session* is a snapshot of currently open documents and arrangement of GUI components that may be restored after the IDE is restarted. The IDE allows saving a number of different sessions and quickly switching between them, making it easy to store and recall the environment for different tasks.

8 Configuration

Many aspects of SuperCollider IDE can be customized, including:

- behavior of automatic indentation and code evaluation
- colors of the editor component and syntax highlighting
- keyboard shortcuts

The IDE also makes easy configuration of the SuperCollider language interpreter. Class library directories to include and exclude from compilation can be configured via the GUI, removing the need to hand-edit the interpreter's configuration file. There is also a handy menu action to open the SuperCollider startup file.

9 Conclusions and Ideas for Future Development

SuperCollider IDE has successfully reached the fundamental goal of providing a cross-platform SuperCollider coding environment. Not only has it integrated the individual strengths of previous coding environments, but it has brought important improvements on its own. Immediate benefits arise from a unified experience across Linux, Mac OS X and Windows. Furthermore, sophisticated user interface design and advanced coding assistance make it both easy to use by novices and a powerful tool for experienced users and developers. In consequence, it makes SuperCollider as a whole more accessible, eases its education and exchange of knowledge, as well as focuses future development work.

As described above, possibilities for improvements have been detected especially at automatic code indentation (4.2) and completion (4.3), and method call assistance (4.4), and are simply a matter of further work. Aside from that, there are many ideas for future development:

SCDoc Editing Support

Among the highest priority goals is support for syntax highlighting and editing assistance for the SCDoc markup language. This would be a very welcome aid in writing SuperCollider documentation, and might entice conversion of remaining old HTML-based documentation to

the SCDoc format (there is a lot of unconverted documents in various Quarks).

Scripting IDE Behavior

The standard SuperCollider class library includes the Document class which is used as a generic programming interface to various coding environments. It allows for controlling the open documents and manipulating with their contents. SuperCollider IDE does not support this interface yet, but the support for it is of high priority, including its potential extension.

Code Snippets

An alternative code editing mode could introduce code snippets as individual interactive components. This would be an alternative for the current concept of *regions* (3.2). The snippets would be separated at the level of graphical interface, instead of code syntax, which could allow for instance to move them freely around a "desk"-like area, hide and show them individually, and to evaluate their contents with a single click.

Visual SynthDef and Pattern Composition

For some tasks it would be welcome to be able to compose SynthDefs and Patterns in a visual way, akin to visual programming languages like PureData, Max, etc. Various diffuse efforts in this direction exist, mostly using the SuperCollider language itself. Most elaborate effort is probably by Jonatan Liljedahl in his ongoing development of algoSCore - a SuperCollider-based successor to AlgoScore [Liljedahl, 2011], which includes graphical composition of SuperCollider Patterns and SynthDefs. We consider potential integration of work in this field into SuperCollider IDE as a great benefit.

Integration of User-Created GUI

GUI creation by users would also benefit from a visual composition approach, as opposed to writing SuperCollider code. Moreover, it would be very practical if user-created GUIs could be integrated into the IDE's own GUI, as *docklets* (2.2) or similar.

10 Acknowledgements

The authors would like to thank the vibrant community of SuperCollider developers and users for critical evaluation of SuperCollider IDE and many useful insights. With such a productive feedback and intensive involvement in shaping ideas, even two lone developers never feel lonely in their efforts. SuperCollider IDE is much better because of you!

References

Tim Blechmann. 2011. Supernova - A scalable parallel audio synthesis server for SuperCollider. In *Proceedings of the International Computer Music Conference*.

Christopher Fraunberger. 2011. SuperCollider on Windows. In Scott Wilson, David Cottle, and Nick Collins, editors, *The SuperCollider Book*. MIT Press.

Stefan Kersten and Marije A.J. Baalman. 2011. "Collision with the Penguin": SuperCollider on Linux. In Scott Wilson, David Cottle, and Nick Collins, editors, *The SuperCollider Book*. MIT Press.

Jonatan Liljedahl. 2011. Algoscore. `http://kymatica.com/Software/AlgoScore`.

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

An Approach to Live Algorithmic Composition using Conductive

Renick BELL
Tama Art University
2-1723 Yarimizu
Hachioji, Tokyo 192-0394
Japan
renick@gmail.com

Abstract

Algorithmic composition can be done as a live performance using live coding tools. An example approach to such performances is described. Using the Conductive library for the Haskell programming language in conjunction with some external tools, samples are triggered according to interonset interval patterns generated at a variety of densities. Automatic movement through those density levels is accomplished through a specialized data structure, which is also used to time-vary other parameter values. The performer manages the state of the above items, and finally audio is output through effects.

Keywords

algorithmic composition, live coding, Haskell

1 introduction

This paper describes an approach to performing extended sets of live algorithmic composition. It was the author's goal to perform generative music live with the computer was an active partner of the user. Rather than prepare data and algorithms completely in advance, it was desired that those algorithms or at least their parameters could be adjusted as the music is being performed.

To carry out such performances, a live coding interface was chosen for its flexibility, its light-weight character, its compatibility with a tiling window manager, and its ability to employ a user interface that was already very familiar: the vim text editor and the command line. Some further discussion on the reasons for choosing this approach have been described in a previous paper, which led to the development of a Haskell library called Conductive to provide some basic components for doing such performances in conjunction with some external tools (Bell 2011). Those tools alone had been determined insufficient for extended performances, however. In order to fully realize such performances, additional modules were developed.

Before explaining this system of tools, the paper first briefly explains live coding. Tools used in conjunction with this system are listed. A brief review of Conductive core concepts is followed by a description of additional modules developed for handling event density, time-varying values, a sampling synthesizer, and mutable data. The paper concludes with a discussion of the results of this approach and some proposed future research directions.

2 live coding

According to TOPLAP, an organization for the promotion of live coding, live coding practice begins in the 80s. The 90s appear dry, while the 21st century starts with the band Slub in what TOPLAP calls the "projection era" and continues to the present in which an increasing variety of live coding systems are available and used in performances (McLean and Others 2010). Now live coding conferences take place (unknown 2013) and it is scheduled to be the theme of an upcoming issue of Computer Music Journal (McLean 2012).

Live coding enables a more abstract manipulation of a representation of music than physical gestures used for playing instruments. It is also thought to be more convenient in many regards than windows-icon-mouse-pointer (WIMP) software (Bell 2011).

From another perspective, it takes the potential of algorithmic composition and turns it into a live performance rather than a write/compile/run loop from traditional software development or electronic music composition. Seen this way, it can be thought of as an extension of algorithmic composition practices that could extend back as far as Ptolemy's music theory (Maurer 1999), and certainly as far back as music dice games such as Mozart's Dice Music (Hedges 1978). More modern examples of algorithmic composition practice include the twelve-tone music of Schoenberg (Schoenberg

1999), work by Caplin and Prinz followed by Hiller and Issacson (Ariza 2011), the aleatoric music of Cage and Stockhausen (Kostelanetz 2002)(Paul 1997), Xenakis's stochastic music (Xenakis 2001), and the generative sequences made in Max on Autechre's Confield (Tingen 2004).

One of the drawbacks of live coding is the hard mental operations that it requires. For a more complete discussion of the usability issues involved in live coding, see Blackwell and Collins (Blackwell and Collins 2005). Another factor is the potentially slow text manipulation that live coding requires (Sorensen and Brown 2007).

The system described below is intended to address some of these difficulties.

3 system and related tools

This section first explains what tools developed by other authors are used when performing. It then reviews some core concepts of Conductive, and finally it describes the newly-developed aspects of Conductive.

3.1 tools developed by other authors

In order to use this system, there are some prerequisites.

The first of those is a Haskell programming environment. The Glasgow Haskell Compiler, which contains an interpreter (GHCi) that allows the interactive evaluation of source code (SL Peyton Jones et al. 1993), is used by the performer to call functions from the Conductive library. The process of writing source code and sending it to GHCi is made more usable with vim (a text editor) (Moolenaar 2008), tmux (a terminal multiplexer)(Marriott and others 2013), and a vim plugin called tslime that allows text to be sent from the editor to the interpreter through tmux (Coutinho 2010).

As Conductive does not directly handle sound synthesis, a method for synthesizing sound is necessary. This paper describes the use of the scsynth component of the SuperCollider package (McCartney 2010). At present, synthesis events are programmed in Haskell and employ Rohan Drape's hsc3 Haskell library for communicating with scsynth (Drape 2009). A sampler (described below) uses samples that have been generally recorded and edited using Ardour (Davis 2006), and they have largely originated from hardware synths. All of the samples are individual sounds, from single-shot percussion sounds to bass samples. Most are wav files under 300 K.

Finally, in order to achieve a solid sound closer to that of commercial releases or broadcasts, the output of scsynth is processed through Calf plugins hosted by the Calf stand-alone host (Foltman et al. 2007). An EQ is followed by a multiband compressor and then a limiter, whose output is directed to the soundcard. Output is also directed to JAAA for monitoring (Adriaensen 2004). Patchage is used for ease of routing (Robillard 2011). Recording of performances is done with either Ardour (in the case of audio) or gtk-recordmydesktop (in the case of video) (Varouhakis and Nordholts 2008).

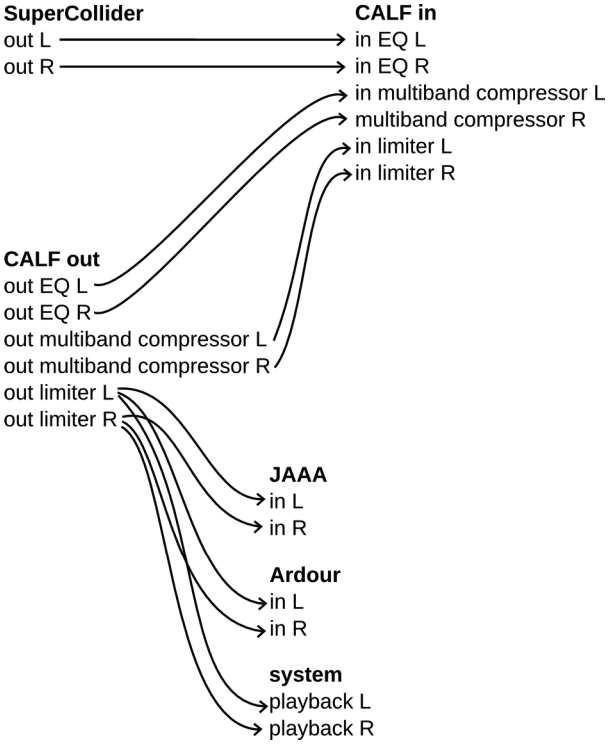


Figure 1: signal flow

3.2 summary of Conductive concepts

A system called Conductive is used, which is a set of modules for the Haskell programming language handling concurrent processes with a music-oriented interface.

Some basic concepts for using Conductive include the notion of Players, action functions, interonset interval (IOI) functions, and TempoClocks. These concepts are explained in more detail in a paper from 2011 (Bell 2011), but a short summary is included here.

Players are representations of concurrent processes that perform actions separated by peri-

ods of time called interonset intervals (IOIs). A Player runs its specified actions and then waits for an IOI determined by its specified IOI function. This loop is instantiated by employing the "play" function with a Player as an argument.

Actions functions define what is done by a Player. These actions could include triggering a synthesis event or modifying the general system state. The only limitation is their type signature, since Haskell is a statically-typed language. This means that the types of arguments to an action function are fixed, and they must return the unit type in the IO monad, or "IO ()". Currently, a sampler action is used predominantly.

IOI functions define how long to wait between actions. Any methods available to the programmer could be used to generate those times, from simply returning a value, such as one second, every time, to table lookup of values, to the calculation of values based on complex mathematical formulae.

One minor change from the 0.2 system of 2011 is that IOI functions now take additional arguments and return the beat of the next event rather than the IOI value directly. The play function uses that value in conjunction with a TempoClock to actually determine how long to wait before running the next action.

4 new modules for Conductive

This section explains the new modules for Conductive: density, TimespanMaps, and MutableMaps.

4.1 IOI values and density

Previously, IOI functions used hand-written IOI patterns or patterns which were determined mostly at random. A more sophisticated approach was sought that would require less manual intervention during a performance.

The sequence of IOI values determines the rhythm of a sequence of events. Rather than enter sequences by hand, they are generated algorithmically. The IOI patterns are looping ordered lists of IOI values in terms of beats, whole or fractional.

Pattern generation is based on a performer-selected core unit used to generate potential IOI values. Selection of a core unit, in conjunction with the length of the pattern, largely determines the metrical feel of the pattern. A list of scalars is determined by the performer, from which a function randomly selects a user-specified number of scalars.

The user specifies a number of subphrases to generate and the length of those phrases in terms of number of scalars to use, from which the final phrase will be constructed. Those subphrases are generated to the specified length by randomly choosing the specified number of scalars from the subset selected above and multiplying them by the core unit.

Finally, a user-specified number of subphrases are chosen at random from the resulting list by the algorithmic composition function. The user determines the length of the final phrase in terms of beats. If the length of the concatenated subphrases does not equal the specified length, the final IOI value is padded. If the length exceeds the specified length, the final IOI value is truncated.

An example of those steps follows. Items are determined by the user are followed with a "u":

- core unit (u): 0.25
- potential scalars (u): 1, 2, 3, 4, 5, 6, 7, 8
- number of scalars to be selected (u): 5
- selected scalars: 1, 2, 3, 4, 6
- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- number of subphrases (u): 2
- subphrase length (u): 3
- selected subphrase scalars: 1, 3, 2; 4, 1, 6
- initial subphrases: 0.25, 0.75, 0.5; 2, 0.25, 2.5
- phrase length in terms of subphrases (u): 3
- initial randomly determined phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.5
- total phrase length: 7.75
- phrase length in terms of beats (u): 8
- final phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75

Given a particular IOI pattern, a series of related patterns (both denser and less dense) is generated. It is built out to maximum and minimum density. This means making a list of IOI patterns ordered in terms of density. When reducing density, an item from the pattern is chosen at random and combined with a neighboring value to yield a similar pattern of reduced density. This process is repeated until the IOI pattern contains only a single item. When increasing density, an item is chosen at random and replaced with two items: an item of lesser value from the list of potential IOIs and the difference between the original IOI value and the lesser value. This is repeated until all of the items in the pattern are the smallest of the po-

tential IOIs. By sandwiching the original IOI pattern between the less-dense and denser patterns, a table is generated.

Here is a continuation of the previous example:

- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- input phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- one level decrease in density: 0.25, 0.75, 0.75, 0.75, 0.5, 2, 0.25, 2.75
- second decrease in density: 0.25, 0.75, 1.5, 0.5, 2, 0.25, 2.75
- minimum density phrase: 8
- one level increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- second increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 0.75, 2
- maximum density phrase: 0.25, 0.25

Based on a user-specified density value, a particular IOI pattern is chosen from the table. The user queries the table with a value between 0 and 1, and a linear conversion to a list index is done. The value returned is the IOI pattern at that index.

Density values can vary with time. One method for doing so is employing a TimespanMap, which is described below.

4.2 TimespanMaps

TimespanMaps are maps or dictionaries with intervals as keys to any kind of value and a specified total length for the whole TimespanMap. In the case of IOI pattern tables described above, the values are either density values (for determining which IOI pattern to use) or an IOI value to be selected from a particular IOI pattern.

A time in beats is passed to the dictionary. The interval that time falls in is determined to be the key, and the corresponding value for that interval is returned.

TimespanMaps can be used for any parameter, not just the ones described above. For example, in this system it also used for determining the amplitude and pitch of a particular triggering of a sample, as well as which sample to trigger.

The rate of change in a TimespanMap is up to the user and can change within the map. The number of items in a TimespanMap is limited only by memory or performance constraints. The range of time covered by a particular key can be as large or small as the user determines to be appropriate and is limited only by the Double data type in the Haskell language (double-precision floating point number).

A sample TimespanMap with a time length of four might look like this:

- length: 4
- 0: "a"
- 2: "b"
- 2.5: "c"

When passed a time of 0, the TimespanMap returns "a". With 0.5, "a" is also returned. With a time of 2.25, "b" would be returned, and with a time of 3.5, "c". If passed a time of 4, the list loops and "a" is returned.

Figure 2 shows the relationship between IOI patterns and density tables. It includes one TimespanMap mapping intervals to density values. Missing from the illustration (in order to keep it less cluttered) is the fact that the IOI patterns themselves are TimespanMaps from intervals to IOI values. The figure shows the calculation of the next IOI value of a running Player from beat 16 to beat 24. The example does not show the complete contents of the density map in order to save space, but actually generating a density map provides the full range of IOI patterns.

Convenience functions for TimespanMaps with random key values and interpolated TimespanMaps (in which fixed-length steps are linearly interpolated from a set of points in time) are provided in the library.

In addition to using TimespanMaps with IOI values and in density maps, they have been used for samples. Previously, one Player was assigned to each sample. In order to use 70 samples, it was necessary to instantiate 70 players. Managing 70 Players was challenging, so the sampler was rewritten to employ a TimespanMap. Subsets of sample sets are chosen at random and one is assigned for each interval in the map. When the sampler is triggered, the sample is chosen according to the current beat. By doing so, the number of Players needed was reduced by roughly a factor of 10.

4.3 mutable state

Several stateful parameters have now been described, and that state is stored in mutable data

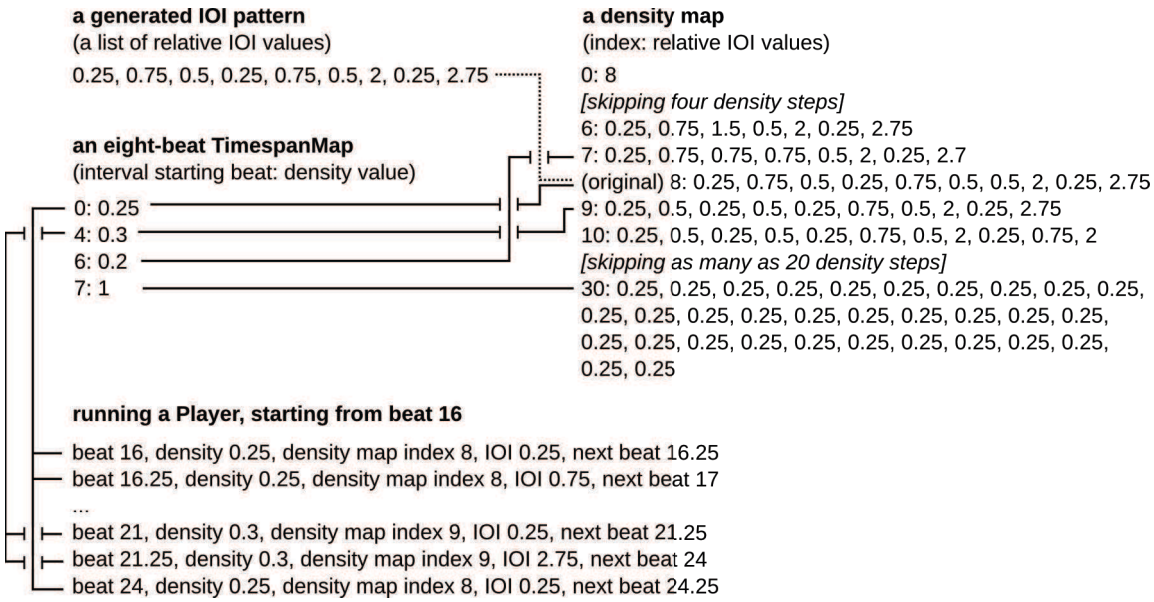


Figure 2: an example showing the relationship between IOI patterns, density tables, and TimespanMaps, based on the example from section 4.1

structures appropriately called MutableMaps. They are maps containing keys and their corresponding values, often a string key and a TimespanMap as the returned value or a mapping of string to string. These maps are stored in a container. Previously, some stateful containers had been used (MVar), but it was suspected that the concurrent operations were not functioning properly in all cases. This was replaced with a TVar, which employs software transactional memory (STM) in order to safely carry out concurrent processes. A full discussion of STM can be found in Jones (2007).

Much of a performance consists of adjusting this state by changing values or adding new key-value pairs. The MutableMap data structure is intended to make doing so easier, with convenience functions for adding, deleting, or swapping values, as well as changing keys.

Because this is live coding, it is easy to write functions to execute multiple simultaneous state changes. Once such functions have been defined, they can be used with a single keyword: the function name.

5 performance method

This section explains how the system above is used to perform. It is divided into two parts: preparation necessary before a performance, and what occurs during a performance.

5.1 pre-performance preparation

An initiation module is prepared before performing. This module imports the various Conductive modules as well as other modules that are useful during performance, such as those in the Data.Map module or list utilities in Data.List. This module also provides functions for the initialization of the mutable state to be used, such as synthesis parameters and references to which sets of synthesis parameters each Player is to use. It also contains the definitions of actions to be used by those Players.

Immediately before the performance, necessary applications must be launched: JACK, Patchage, scsynth, calfjackhost, JAAA, ghci, and vim. The necessary routing must be set up between these applications, including manually routing audio in Patchage and setting vim to send to ghci.

The Conductive modules are loaded in GHCi as well as the previously prepared initialization routines. Preparation of scsynth is done, and samples are loaded into scsynth. For the sampler to be effective for an extended performance, it is necessary to prepare enough samples.

IOI patterns are generated along with their corresponding density tables. TimespanMaps are created to vary the density values used for selecting IOI patterns. Players are specified, along with their initial action and IOI functions. The density patterns and IOI patterns are assigned to Players, which use the current beat to

determine a density and thus an IOI value to be used between instances of triggering a sample. The sample that is triggered is also determined by a TimespanMap.

5.2 during a performance

When the data has been prepared, Players are started according to user's intentions for the performance. As the Players are running, the state is manipulated by the user to vary the performance. This includes changing synthesis parameters or other system parameters. New patterns can be generated by the performer running the algorithmic composition routine described above. New density maps can be generated and assigned to Players. New samples can also be loaded. During the performance, Players can be stopped, restarted, or additional Players can be added and similarly manipulated. It is also possible to define new action functions or IOI functions during a performance. All Players are stopped when the performance has reached an end.

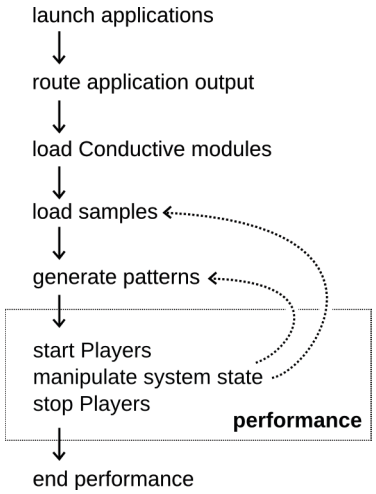


Figure 3: performance flowchart

6 conclusion

This section evaluates the results of using the system described above. It then describes some directions for future research.

6.1 evaluation of results

The approach above makes it possible to perform for extended periods of time, mostly limited by the amount of samples that have been prepared in advance. It is also necessary to use different sets of parameters when generating IOI patterns, such as differing numbers of scalars, differing core unit sizes, and so on.

It is challenging to keep a mental model of the parts described above during a performance, even though what has been described is mainly concerned with the timing of events and not timbre. This suggests that adding the complexities of generating different timbres through synthesis during performance will be burdensome. Part of this burden can be overcome through more practice with the system, but it seems that there is still a higher-level of abstraction to be achieved for optimal usage.

By using TimespanMaps with the sampler, it was possible to reduce the number of players for 70+ samples from 70 Players to between four and eight Players triggering hundreds of samples. This arrangement was found to be much more manageable and sonically-attractive than the previous one. It was very difficult previously to look at the list of Players and see which ones were playing and which ones were not. It also made changing the arrangement very hard, as Player-related functions often required long lists of Players. The current arrangement still uses lists of Players at times, but the lists are much shorter, rarely containing more than three or four Players.

The lack of continuous timbral modification through effects is a sore spot. A moving timbre can make the sound much more lively, but this is possible to a limited degree in the system above because of the design of the current sampler. Varying the sample of a particular Player does change the timbre, but sometimes a change which unfolds in a discernible direction over time can be a more effective compositional device. This has only been achieved in the current version for the sample pitch and amplitude.

In the system described above, parameters for synthesis are initiation-rate values. That means that the timbre of a particular event does not change over time other than what is contained in the original sample.

Changing from MVars to TVars with STM is thought to have solved some mysterious runtime misbehavior.

Current methods for organizing the text data or source code used during a performance are poor. As a result, the text in the text editor quickly becomes messy in the course of a performance. That makes it harder to stay in control of the performance or to run previously-defined functions at the most ideal times. Maximum effectiveness of use of the editor environment probably has not yet been achieved. Editor us-

age skills or tools to aid in this are probably needed.

6.2 future directions

Many things for this system can be developed. Those possibilities include the following ideas.

Further refinements of the abstractions described above can be done. That includes using value-lists rather than single values as core units in the IOI pattern generation process. More intelligent ways of generating the various densities of those patterns can be imagined.

Methods for generating IOI patterns with a greater sense of relationship is desired. While the patterns generated above are related in terms of density and rerunning a pattern generation function with the same parameters can yield similar patterns, there must be more sophisticated ways to generate sets of related patterns. More investigation into music theory and the algorithmic composition techniques of others is needed. Such research should be included in future versions of the pattern-generation functions.

Several chance operations are involved in this approach. It would be desirable to try weighted probabilities or other deterministic means as substitutes for those chance operations.

An increased use of pitched synths can be included. This will make it easier to achieve the timbral variation desired as well as expand the focus from its time-based focus at the moment to more involvement with the frequency domain. An efficient, easy-to-use method of synthesis that can also provide a wide range of timbres is being sought. Samples are musically effective but take a lot of time to prepare and remove a level of spontaneity that is desired.

Algorithmic control of effects at various stages would be nice. This means writing those effects and the corresponding action functions.

Player processes which alter other running Player processes should be experimented with, such as Players that stop and start other players. Another possibility to try in the near future is Players which change between sample sets.

Visualization methods for system state should be undertaken.

A convenience function for concatenating TimespanMaps is also desired.

Better methods for managing the code used in a performance should be sought.

7 acknowledgements

Thanks to Henning Thielemann and the reviewers for useful suggestions on the contents of this paper. Thanks also goes to Akihiro Kubota and Yoshiharu Hamada for research support.

8 bibliography

Adriaensen, Fons. 2004. “Kokkini Zita - Linux Audio.” `http://kokkinizita.linuxaudio.org/linuxaudio/`.

Ariza, Christopher. 2011. “Two pioneering projects from the early history of computer-aided algorithmic composition.” *Computer Music Journal* 35 (3): 40–56.

Bell, Renick. 2011. “An Interface for Realtime Music Using Interpreted Haskell.” In *Proceedings of LAC 2011*.

Blackwell, Alan, and Nick Collins. 2005. “The Programming Language as a Musical Instrument.” In *Proceedings of PPIG05*. University of Sussex.

Coutinho, C. 2010. “tslime.” `http://www.vim.org/scripts/script.php?script_id=3023`.

Davis, Paul. 2006. “Ardour.” `http://ardour.org`.

Drape, Rohan. 2009. *Haskell supercollider, a tutorial*.

Foltman, Krzysztof, Markus Schmidt, Christian Holschuh, and Thor Johansen. 2007. “Home @ Calf Studio Gear - Audio Plugins.” `http://calf.sourceforge.net/`.

Hedges, Stephen A. 1978. “Dice music in the eighteenth century.” *Music & Letters* 59 (2): 180–187.

Jones, SL Peyton, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. “The Glasgow Haskell compiler: a technical overview.” In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*. Vol. 93.

Jones, Simon Peyton. 2007. “Beautiful concurrency.” *Beautiful Code: Leading Programmers Explain How They Think*: 385–406.

Kostelanetz, Ric. 2002. *Conversing with Cage*. Routledge.

Marriott, Nicholas, and others. 2013. “tmux.” `http://tmux.sourceforge.net/`.

Maurer, John. 1999. “A Brief History of Algorithmic Composition.” `https://ccrma.stanford.edu/~blackrse/algorithm.html`.

McCartney, J. 2010. “SuperCollider Documentation.” *http://www.audiosynth.com*.

McLean, Alex. 2012. “Computer Music Journal special issue on Live Coding \textbar TOPLAP.” http://toplap.org/cmj/.

McLean, Alex, and Others. 2010. *TOPLAP website*. http://www.toplap.org/index.php/Main_Page.

Moolenaar, Bram. 2008. “The Vim Editor.” http://www.vim.org.

Paul, David. 1997. “Karlheinz Stockhausen.” *interview, Seconds Magazine* 44.

Robillard, David. 2011. “Patchage.” http://drobilla.net/software/patchage/.

Schoenberg, Arnold. 1999. *Fundamentals of Musical Composition*. Ed. Gerald Strang and Leonard Stein. Faber & Faber.

Sorensen, A., and A. R. Brown. 2007. “aa-cell in Practice: An approach to musical live coding.” In *Proceedings of the International Computer Music Conference*.

Tingen, Paul. 2004. “Autechre, recording electronica.” *Sound on Sound* 19 (6): 96–102.

Varouhakis, John, and Martin Nordholts. 2008. *recordMyDesktop Version 0.3. 7.3*.

Xenakis, Iannis. 2001. *Formalized Music: Thought and Mathematics in Composition*. 2nd ed.. Pendragon Pr.

unknown. 2013. “live.code.festival 2013 – Call for Participation.” http://imwi.hfm.eu/livecode/call/.

MorphOSC- A Toolkit for Building Sound Control GUIs with Preset Interpolation in the Processing Development Environment

Liam O'SULLIVAN
Electronic & Electrical Engineering, Trinity College Dublin
Dublin 2, Ireland
lmosulli@tcd.ie
https://github.com/LiamOSullivan/MorphOSC

Abstract

MorphOSC is a new toolkit for building graphical user interfaces for the control of sound using morphing between parameter presets. It uses the multidimensional interpolation space paradigm seen in some other systems, but hitherto unavailable as open-source software in the form presented here. The software is delivered as a class library for the Processing Development Environment and is cross-platform for desktop computers and Android mobile devices.

This paper positions the new library within the context of similar software, introduces the main features of the initial code release and details future work on the project.

Keywords

Toolkit, Processing Development Environment, Open Sound Control, User Interface, Preset Interpolation.

1 Introduction

The control of complex, dynamic sound typically involves manipulation of a large number of parameters. Complex mappings that link one-or-more input controls to one-or-more outputs have been seen to be more effective for the provision of engaging, expressive play than simple one-to-one mappings [6]. One approach to the control of multiple parameters in real time is the use of a multidimensional space superimposed on a two-dimensional graphical controller [10]. Particular settings for an ordered collection of parameters can be associated with anchor points on the controller surface and the movement of a cursor provides an interpolated output value for each parameter. The usefulness of such an approach for the provision of musical control has been noted previously in the above examples; although independent control over each output is compromised, an intuitive and 'playable' space is provided. This two-input to many-output (two-to-many) mapping can be well-suited for live performance or the exploration of timbre spaces generated when the interpolated output is sent to a synthesiser.

Although several systems provide a graphical user interface (GUI) to some implementation of such a scheme, they are usually tied to a particular application, are commercial products or are not portable to multiple platforms. To address this, a new code library is presented that facilitates rapid prototyping of interfaces utilising preset morphing- MorphOSC.

Section 2 of this paper briefly describes similar work in the form of existing GUIs that allow complex mappings through interpolated parameter spaces. The design goals for the new tool are then identified as a reaction to what is currently available. Some background to one method of parameter morphing is provided in section 3. Section 4 outlines the current library implementation, identifying key features of the software in its current state. Future areas of development are discussed in section 5 and final conclusions are made.

2 Similar work

Several software systems exist that facilitate the exploration of multidimensional parameter spaces. While some of these are sophisticated systems offering extensive functionality, it will be shown that a gap exists for the approach being outlined here due to the limitations described in each case. Previous work by the author describes a number of more general mapping interfaces [12] and will not be repeated in this paper, but salient examples of more general software controllers and specific interpolating interfaces are now presented.

2.1 Interpolating interfaces

The provision of effective control of computer music systems via preset interpolation has historically been of interest. As far back as the late 1970s, researchers at the Inaís Groupe de Recherches Musicales provided such functionality in the GUI component of the SYTER system [5]. Today, the real-time processing capability of desktop computers and even mobile devices means that interfaces can be implemented as components of a larger software system. The Max/MSP programming environment [9] is a popular example and provides many data-manipulation tools; a recent implementation of an interpolating controller for this environment is the *nodes* object. This allows many inputs to be weighted and combined to a single output based on the positions of overlapping circular graphical nodes. Similarly, the IRCAM MnM mapping toolbox, (part of the FTM external object library [3]) allows the user to build patches with existing Max/MSP GUI elements. For instance, an example patch allows the specification of two-to-many mappings using a two-dimensional controller and a set of linear sliders. The system can associate points on the controller with particular slider arrangements and value settings; moving between points provides a smooth morph between the sliders' states.

The MetaSurface is an interface for interpolating between parameter 'snapshots' for two-to-many mappings [1] and an example implementation is included with the AudioMulch software [2]. Still more recently, one project [8] provides a preset-interpolation interface for the SuperCollider environment [18], designed for use with a bespoke physical controller.

The above examples are part of more fully-featured programs rather than standalone controllers and/or are commercial products. They cannot be used with Android mobile devices. The ability to include subsets of output parameters in the interpolation space is provided in some cases, typically via check-boxes or a set-up dialogue. However, an interface which uses the drag-and-drop metaphor to manipulate parameter sets would provide a more interactive experience. The use of a multi-layered GUI approach would also allow more complex mapping relationships to be built and refined.

2.2 Standalone interfaces for Open Sound Control

Software controllers already exist for mobile and touch-screen devices that can output messages over a network formatted using the Open Sound Control (OSC) standard [11]. From simple applications like *andOSC* [12] to more sophisticated tools such as the popular *TouchOSC* [15], these offer real-time control of musical applications and exploit the multi-touch capability of contemporary phones and tablets (as well as additional sensor input from accelerometers etc.). Although functionality-limited free versions are available, they are not open source. Neither do they provide an interpolation surface, meaning this must be implemented on a networked computer if required. This separates the mapping configuration from the interface, inhibiting engagement and obstructing work-flow A more unified interface would facilitate greater exploration of the parameter space and dynamic mapping during performance.

2.3 Processing Development Environment

The Processing Development Environment (PDE) is an open-source initiative that attempts to make it easier for artists, designers and novice programmers to implement computer-based projects. It uses a streamlined form of the Java programming language and has evolved to become a very popular tool for creatives. Code libraries provide additional functionality such as enhanced interactivity and sound generation.

One such contributed library is the recent JunctionBox toolkit [4], which can provide interaction capability beyond the use of traditional controller widgets. Code 'sketches' written in Processing can include this library's functionality to produce OSC messages triggered by common mouse-based or multi-touch interactions (e.g. scaling, rotation etc.). As the PDE now supports rapid Android application prototyping[1], this allows easier implementation of novel OSC controllers for mobile devices. However, as the focus is on the generation of messages based on common spatial manipulations of graphical objects, it does not particularly address the production of more complex GUIs including preset-interpolation surfaces. Nevertheless, the library serves as a useful template for the provision of such functionality through a code library for the PDE.

2.4 Project goals

The design goals which emerge from the initial motivations for the project and subsequent consideration of similar work are as follows:

[1] As of version 2.0 beta 7, March 2013.

- Freely available, open source, cross-platform compatible toolkit for rapid prototyping of preset-interpolation interfaces.
- Interaction design exploiting familiar metaphors for intuitive configuration of the parameter interpolation space (e.g. drag-and-drop, layering).
- OSC-formatted output.

3 Interpolation methods

A full discussion of the various methods available for interpolation between a set of scattered data points is beyond the scope of this paper and the reader is directed to an overview from the field of cartography [7]. However, the techniques used in some examples of similar interfaces are summarised in table 1.

Software	Method
SuperCollider *PresetInterpolator*	Intersecting N-Spheres
Max/MSP *nodes*	Inverse Distance Weighting
AudioMulch/ MetaSurface	Natural Neighbour

Table 1: Interpolation methods used in some existing GUIs for musical control.

The need for real-time performance and the suitability of the software for mobile platforms prioritises the use of computationally inexpensive interpolation techniques. For the initial toolkit release, the method of Inverse Distance Weighting (IDW) was preferred.

3.1 Inverse-Distance Weighting

IDW is commonly called Shepard's Method following an early documentation of the technique [17]. In essence, it can assign values to unknown points by calculating a weighted average of the values at scattered sample points. The normalised distances from the interpolation point to the known values, $\mathbf{d_n}$, are used to scale the values of each parameter at these points, $\mathbf{p_{ni}}$, in an inverse relationship. The results are then averaged, meaning points further away have less effect on the interpolated value of a particular parameter. A general expression for the operation is therefore:

$$p_i = \frac{\sum_{n=0}^{k} p_{ni} d_n^{-1}}{\sum_{n=0}^{k} d_n^{-1}}$$

modification to the technique uses the square of the distance involved and may be more suited to the control of musical parameters, due to the non-linear nature of certain aspects of human perception and experience of the real world (e.g. inverse-square law attenuation of sound with distance). IDW considers all points on the surface, but may also be modified to only consider the nearest points and reduce the computation required for interpolating the output. Figure 1 shows an interactive Processing sketch that outputs a set of interpolated values for a three-dimensional parameter space mapped to a two-dimensional controller surface[2]. This illustrates how parameters at the interpolation point (i.e. the output) are calculated from their ordered counterparts at the 'known' sample points (i.e. the anchor points).

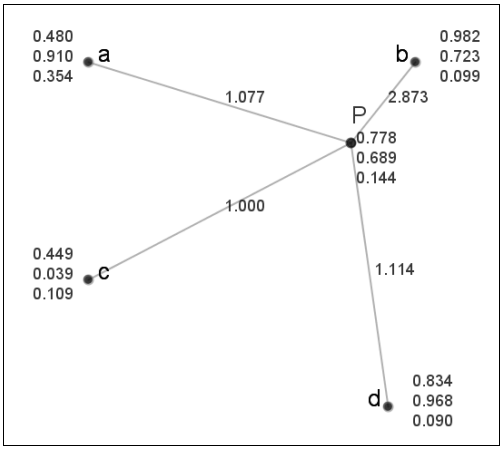


Figure 1: Inverse Distance Weighting used to interpolate values for the parameters at a point **P** from a set of scattered sample points **a, b, c** and **d**. The normalised values for the weights (inverse distances) are shown along the vector lines and three parameter values are placed at each point.

4 Implementation

The current toolkit was programmed in Java and is available as a library for the PDE. This includes example interfaces which can be loaded into the environment and modified, or exported to be run as standalone applications across multiple platforms (OSX, Windows, Linux, Android).

The toolkit builds on the functionality of other contributed libraries for Processing to allow easy

[2] The Processing code for this example is avaliable at: https://github.com/LiamOSullivan

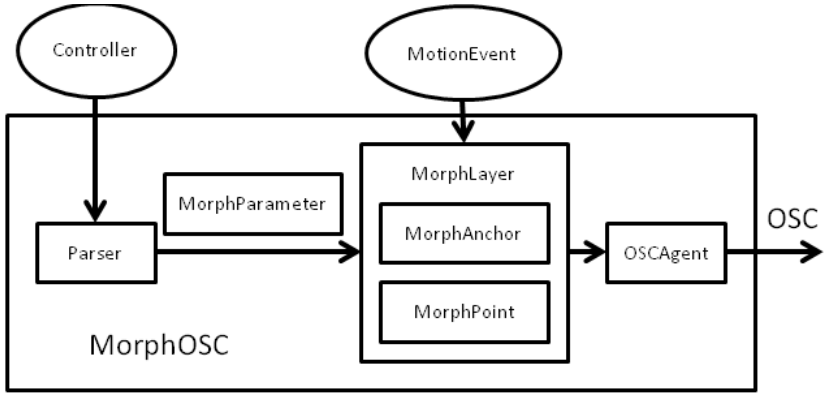


Figure 2: Overview of the core library classes (rectangular boxes) of the *MorphOSC* toolkit.

integration into the work-flow of developers and to keep the code base to a minimum. The library is design to make use of the popular ControlP5 [15] library for the provision of on-screen control widgets, while the OSC subsystem uses the oscP5 library [16] to format output appropriately.

Settings for the constructed GUIs can be stored and recalled using a preset file, formatted with extensible mark-up language (XML) for ease of portability.

Class	Description
MorphOSC	Base class, manages interaction space.
Parser	Parses subset of widget fields.
MorphLayer	Interactive GUI element. Container for (i), (ii), (iii).
(i) MorphAnchor	Holds a set of parameter values.
(ii) MorphPoint	An interpolation point.
(iii) MorphParameter	Parameter value parsed from widget.
OSCAgent	Formats outgoing messages.

Table 2: Core classes of the MorphOSC library.

The core classes that implement the MorphOSC library are listed in table 2. Figure 2 outlines their inter-relationships.

4.1 Usage

The library employs the conventions common to contributed libraries for the PDE, as shown in the example code of table 3. The base class for the library is instantiated in the usual way, by passing a reference to the parent PApplet (the encapsulating class for a Processing program). This effectively creates an interaction area at runtime with the same dimensions as the parent. Widgets are defined using the ControlP5 library to implement the interface design in the usual way. Any widgets which are to be included for morphing are then added to the MorphOSC instance using the **add()** method. This sends the element to the Parser class; a subset of the controller properties are extracted and a MorphParameter instance for each added controller is returned.

```
MorphOSC morph = new MorphOSC(this);
ControlP5 cp5 = new ControlP5(this);
Slider s = cp5.addSlider();
morph.add(s);
```

Table 3: Example Processing code. MorphOSC and ControlP5 base classes are instantiated. A slider is created and added to the MorphOSC object.

All other public classes are modified at runtime through interaction with the GUI.

4.2 Interaction Design

Manipulation of MorphOSC elements through the interaction area depends on the current mode of the interface, which can be in *Edit Mode* or *Performance Mode*.

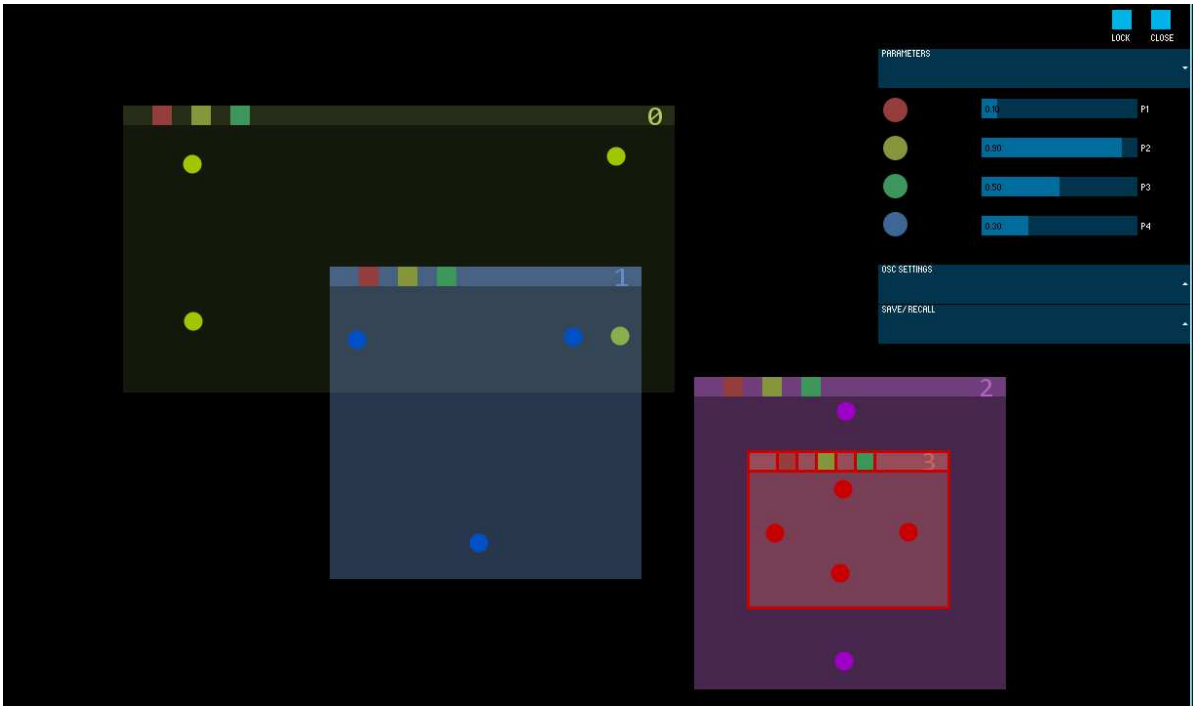


Figure 3: An example GUI created in Processing with MorphOSC and ControlP5. In the unlocked *Edit Mode*, GUI elements may be modified but interpolation output may still be auditioned in real time. MorphLayer number 3 is in focus and the various widgets for system settings are visible on the right hand side of the screen.

4.2.1 Edit mode

When unlocked in *Edit Mode*, MorphOSC elements may be created, modified and destroyed. For example, a MorphLayer can be instantiated with an event (e.g. a mouse click or a double finger-tap) in free space on the interaction area. Layers may subsequently be moved/ resized and overlapping is possible. Any widgets which have an associated MorphParameter are indicated in the GUI with a unique colour. A MorphAnchor may be added to a layer by using a drag-and-drop action from the numerical value attached to the corresponding widget. This adds the associated MorphParameter to the layer and initialises the MorphAnchor value for that parameter to the current widget value. Additional parameters may be added to existing anchors or anchors may have their values overwritten by subsequent drag-and-drop actions. Anchors may be moved about their layer to reconfigure the underlying interpolation space; finer control may be attained by moving anchors further apart, for example.

Edit mode produces interpolated values for parameters by dragging (with mouse or touch) in free space. This allows the user to audition parameter interpolation in real-time as they are manipulating them, but is not meant to provide a full performance mode.

4.2.2 Performance mode

When in the locked performance mode, MorphLayers, MorphAnchors and other instantiated classes cannot be modified other than through the specification of MorphPoints to generate interpolated parameter values. Interaction with the MorphLayers produces interpolated values for their associated MorphParameters based on the arrangement of MorphAnchors within them. MorphPoints interpolate values from all layers behind them. This means that subsets of parameters can be associated with different layers and spatial positions, providing a lot of flexibility in design of the control space.

Performance mode contains the option to hide all ControlP5 GUI elements so that the whole space is available for gestural input.

5 Conclusion and future work

This paper introduced a new toolkit to aid in the rapid development of GUIs utilising preset interpolation for the control of sound over OSC. A short review of similar work identified the need

for a code library for the popular Processing environment, in order to allow cross-platform interface development. Following a brief discussion of a suitable interpolation method, the new toolkit- MorphOSC- was then introduced and key features were outlined.

The software is currently in beta version and there is much work to be done to produce a release candidate. A full evaluation of the system is required to assess stability and performance. Use of the system in a workshop setting is proposed to evaluate usability and performance is to be tested 'in the wild'.

The current system implements a simple averaging interpolation scheme through IDW, but as this can have some limitations (e.g. computation time proportional to the number of anchor points) other methods will be examined. It is envisaged that the toolkit will serve as a test bed for evaluating the effectiveness of various interpolation methods for the provision of real-time control of musical output.

This work forms part of a larger project which attempts to leverage the benefits of two-dimensional interfaces for musical control. The multi-layer paradigm is seen as a strong metaphor for the provision of intuitive interactions not currently supported in existing software.

Acknowledgements

Thanks to Andreas Schlegel for both the oscP5 and controlP5 libraries. The author appreciates the comments of the reviewers and the beta-testers for their invaluable feedback.

References

[1] Bencina, R. The Metasurface – Applying Natural Neighbour Interpolation to Two-to-Many Mapping. *Proceedings of the 2005 Conference on New Interfaces for Musical Expression (NIME'05)* (Vancouver, BC, Canada, May 26-28, 2005), 101-104.

[2] Bencina, R., AudioMulch interactive music studio. http://www.audiomulch.com/

[3] FTM & Co., IRCAM. http://ftm.ircam.fr/

[4] Fyfe, L., Tindale, A. and Carpendale, S. JunctionBox for Android: An Interaction Toolkit for Android-based Mobile Devices. *Proceedings of the Linux Audio Conference (LAC2012)*, (CCRMA, Stanford University, CA, USA. April 12-15, 2012).

[5] Geslin, Y., Digital Sound and Music Transformation Environments: A Twenty-year Experiment at the Groupe de Recherches Musicales. *Journal of New Music Research* 31(2): 99–107, 2002.

[6] Hunt, A. and Kirk, R., Mapping Strategies for Musical Performance. *Trends in Gestural Control of Music*, M. Wanderley and M. Battier, Editors, 2000.

[7] Lam, N., Spatial Interpolation Methods: A Review. *The American Cartographer*. 10(2): 129-149, 1983.

[8] Marier, M., Designing Mappings for Musical Interfaces Using Preset Interpolation. *Proceedings of the Conference on New Interfaces for Musical Expression (NIME '12)*, (May 21 – 23, 2012, University of Michigan, Ann Arbor).

[9] Max/MSP environment from Cycling 74. http://cycling74.com/products/max/

[10] Momeni, A., Wessel, D., Characterizing and controlling musical material intuitively with geometric models. *Proceedings of the 2003 conference on New Interfaces for Musical Expression* (NIME '03) (2003, National University of Singapore, Singapore), 54-62.

[11] Open Sound Control. http://www.opensoundcontrol.org

[12] O'Sullivan, L., Furlong, D., and Boland, F. Introducing CrossMapper: Another Tool for Mapping Musical Control Parameters. *Proceedings of the Conference on New Interfaces for Musical Expression (NIME '12)*, (May 21 – 23, 2012, University of Michigan, Ann Arbor).

[13] Primevision andOSC Android application. https://play.google.com/store/apps/details?id=cc.primevision.andosc&hl=en

[14] Processing Development Environment. http://www.processing.org

[15] Schlegel, A., Sojamo ControlP5 Library for Processing. http://www.sojamo.de/libraries/controlP5/

[16] Schlegel, A., Sojamo OscP5 Library for Processing. http://www.sojamo.de/libraries/oscP5/

[17] Shepard, D., A two-dimensional interpolation function for irregularly-spaced data. *Proceedings of the 1968 23rd ACM national conference*, 517–524.

[18] SuperCollider. supercollider.sourceforge.net

[19] TouchOSC. http://hexler.net/software/touchosc

Design of an audio oscilloscope application

Fons ADRIAENSEN,

Casa della Musica,
Pzle. San Francesco 1,
43000 Parma (PR),
Italy,
fons@linuxaudio.org

Abstract

This paper documents some aspects of the design of zita-scope, an Audio Oscilloscope application for the GNU/Linux system. It is designed to permit accurate display and measurements on audio waveforms captured from any source via the Jack audio server. Topics covered include performance requirements, an analysis of some problems that need to be considered, and an overview of the implemention structure. The software will be available at the time this paper is presented at the 2013 Linux Audio Conference in Graz.

Keywords

linux, oscilloscope, audio measurement, time-domain, jack

1 Introduction

The oscilloscope has for a long time been a standard instrument for any engineer developing audio equipment, and in fact for almost everyone 'doing electronics'. In the all-digital era its importance in an audio related context may have declined a bit, except for debugging digital audio hardware. Fact is that many measurements on audio systems are better performed using spectral analysis or dedicated tools, but in some cases the ability to view the time-domain waveform and perform measurements on it remains essential.

Very few Linux applications for this use seem to exist. There are various 'scrolling scopes' which will display a waveform in real time, but don't permit any form of measurement. Some graphical synthesis environments include a 'scope' module or object, but these scopes are little more than a toy. They allow the user to see that a waveform is indeed a sine or a square wave, or to get an idea of the waveform envelope, but there it ends.

The only more ambitious application found by the author at the time of writing was something called *xoscope* [1]. After some patching it compiled, but it takes its inputs from */dev/dsp*, *EsounD* or some esoteric hardware only, doesn't know about ALSA or Jack, and the user interface really looks very dated. Probably its development has stopped years ago.

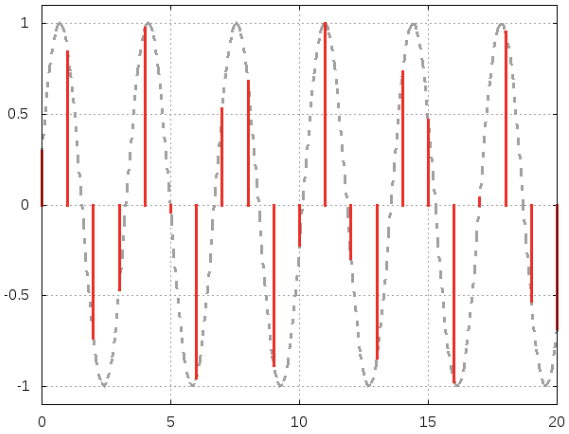


Figure 1: A sampled sine wave

Reasons for this state of affairs are clear enough: 'technical' applications (as opposed to those meant for creating music) are a minority interest, and actually creating a usable software scope isn't as simple as it seems — there is a lot more involved than just 'plotting the samples'.

2 Requirements

Displaying samples is what any serious oscilloscope application must *not* do. If a signal contains any significant energy above say 1/10 of the sample rate, the sample values provide a very bad or at least a quite unintuitive visual representation of the actual waveform. See for example Fig.1. After some training one may be able to recognise this as a 14 kHz sine wave sampled at 48 kHz, but in general it's near impossible to obtain any meaningful information from such a display.

Assuming a scope will be used to perform measurements and not just as a visual gadget,

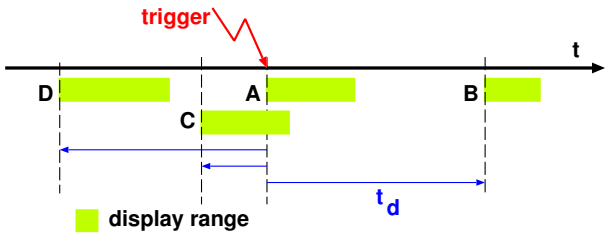


Figure 2: Trigger and display range

the following should be considered essential:

- An *accurate* and *stable* display of the analog waveform corresponding to a stream of samples.
- A wide range of *calibrated* display ranges and resolutions in both the time and amplitude domains.
- At least two and preferably more simultanuous channels.
- A *flexible* and *accurate* system allowing the user to capture particular events in an audio signal.
- The ability to store a signal and examine it at all available gain and time resolution settings.
- Calibrated markers to aid accurate measurement.
- Responsive user controls, e.g. changing display parameters should produce an almost immediate result.

And less essential but nice to have:

- Facilities to perform more complex measurements, e.g. the RMS value of a range, spectrum, etc.
- Remote control, allowing the applicatin to be configured by and report to automated test systems or scripts.
- 'Reasonable' CPU and other resource usage.

3 Problem analysis

3.1 Triggering

While a scope can be used in *free running* mode, in most cases a *triggered* display is essential. The principle is illustrated in Fig.2. The user will select a trigger condition, for example a positive going zero crossing. The start of the displayed range will then be at a fixed offset t_d from that point, selected by the user. In many cases the trigger point will be the start of the displayed range (case **A** in the figure), but even most analog scopes offer a *delayed trigger* option (case **B**), with a delay that can be much longer than the displayed range. A digital scope can easily store the signal, and allow to display part of the signal *before* the trigger (cases **C** and **D**). This is very useful when the trigger condition is the consequence of something that happened before and which the user wants to investigate.

Triggering can be *continuous* or *single shot*. In the first case, if a trigger has been found, and as soon as enough signal has been captured to fill the display and all of it is processed, the system can start looking for the next trigger and the cycle repeats. This could result in a very high update frequency (if the display range is short and close to the trigger) which would just lead to an excessively high CPU load without improving the visual result. In such cases looking for the next trigger should be delayed by 50 milliseconds or so.

In the *single shot* mode, signal capturing will stop at some point after the displayed range, allowing the user to examine all of the stored signal. In that case, the position of the trigger point in the stored buffer becomes a parameter that should be controllable by the user — this determines how much he/she will be able to scroll forward or back from the initial display range.

The usual trigger condition is the signal crossing a given value in a specified direction, up or down. This point needs to be determined with high accuracy. Consider the following conditions: we are looking in continuous trigger mode at some high frequency waveform, with a display range of 50 microseconds (one period at 20 kHz). Assume the display is 1000 pixels wide. Then each pixel corresponds to 0.05 microsecond, and if we want a stable display the jitter on the trigger position must be at least ten times smaller than that value, say 5 nanoseconds or around 1/4000 of a sample at a sample rate of 48 kHz. Simple linear or even cubic interpolation on the original samples won't be sufficient to achieve this .

The solution used in *zita-scope* is to first upsample the signal selected as the trigger source by a factor of 5. This means that even in the

worst case — a sine wave near half the sample frequency — in each half cycle there will always be samples covering the range of -0.95 to 0.95 times the amplitude, and triggering within that range will be reliable. Assume the trigger level is V with the signal going up. We scan the interpolated waveform for two consecutive samples v_0 and v_1 such that $v_0 \leq V \leq v_1$. When these are found, the signal is locally upsampled by a factor of 25, and we search for v_0, v_1, v_2 and v_3 such that $v_1 \leq V \leq v_2$. Given these we can find the best fitting parabola $f(x) = ax^2 + bx + c$ with $f(0) = v_1$ and $f(1) = v_2$. Solving the quadratic equation then provides the exact location of the trigger point, with a worst case error of around 1/100000 of a sample at the original sample rate. The calculations are quite simple but require some attention to cover special cases, e.g. the quadratic coefficient could be near zero. Four points rather than three are used to provide an estimate of the quadratic term at the center of the interval $[v_1, v_2]$.

Another option, usually not available on analog scopes, is to trigger on the first positive or negative peak exceeding a given value. This can be done using a similar method, in this case searching e.g. for three samples v_0, v_1, v_2 with $v_0 < v_1 > v_2$, and then solving the derivative of the quadratic equation.

The first release of zita-scope can have up to four displayed channels, and each of those can be the trigger source. Also a separate trigger input is provided. This can be used in the way described above, or it can be put in 'digital' mode, meaning that the trigger position will be the first sample crossing a given value, e.g. an impulse provided by some external software.

Another option is the *manual* trigger mode. Clicking a button in the GUI generates a single sample pulse on a trigger output, and the trigger point is exactly one period later (looping the pulse back to the digital trigger input would give the same result). This can be used to measure e,g, the impulse response of a filter.

Some other modes could be useful, for example triggering on a MIDI note-on event delivered via Jack-midi, for example to test the latency of a soft synth, or on Jack transport reaching a preset value. These could be built-in, or provided by a separate app connected to the external trigger input.

3.2 Waveform display

As already illustrated by Fig.1, displaying the waveform corresponding to a sampled signal involves more than just plotting the sample values. A digital audio scope could have a horizontal scale ranging from a second per grid division down to a microsecond, a range of one to a million. In all cases the user wants to see a more or less accurate representation of the waveform. For an analog scope this is no problem as both the signal and the display device have 'infinite' resolution. For a digital scope we need to consider that the waveform is sampled and the display consists of discrete pixels.

The first question is which graphics library will be used. On Linux, the choice is between the basic X11 drawing routines and Cairo [2]. GUI toolsets offering a 'canvas' object will also use one of these. X11 graphics are defined entirely in terms of pixels. Cairo offers subpixel coordinates and anti-aliased line drawing. This provides a much better visual quality, but not a higher resolution.

On recent multi-core hardware there is really no reason for not using Cairo or something similar. The situation is different if somewhat older systems are considered, e.g. a single core 2 GHz Pentium 4. On such hardware, when drawing four waveforms 20 times per second on a full screen window, using Cairo can easily take the CPU power to its limits.

The solution adopted in zita-scope is to provide both. By default Cairo will be used in all cases, but there is an option to use X11 when the display is updated at a high frequency, automatically switching to Cairo in all other cases.

Assume the display is showing one or a few cycles of a sine wave, so each cycle has a nontrivial width on the screen. An accurate display of say 1000 by 1000 pixels requires something like 70 points per cycle in that case. This ensures that the extreme values shown are no less than 0.999 times the real peaks (i.e. less than half a pixel error), and the waveform doesn't look like a series of connected straight lines. Since the frequency could be near half the sample rate, this would require upsampling by a factor of at least 35.

A brute-force technique would be to always upsample by a factor of at least 35 and plot all the points. But this would be very inefficient in almost all cases. Consider a display that is 100 ms wide — this would mean 168000 points after resampling, and most of the effort spent com-

puting and displaying them would be wasted as the display doesn't have the resolution required to show all that detail. Clearly some better idea is needed.

To get a grip on the issues involved we will use the following parameters:

- F_{sig} : the original signal sample frequency, e.g. 48 kHz.
- F_{pix} : the pixel frequency. For example if we have 1 millisecond per division and a division is 100 pixels, then F_{pix} is 100 kHz.
- F_{res} : the sample frequency after upsampling.

Zita-scope uses two different algorithms and display routines, depending on some of those parameters.

If $F_{pix}/F_{sig} \geq 35$, we compute one sample per k pixels on the x-axis, with k integer. These points are then plotted as a sequence of straight lines. This provides the best that can be done when using X11 (unless we would implement some ad-hoc anti-aliasing scheme), and Cairo will show a smooth anti-aliased line. In this case we have:

$$\begin{aligned} k &= \lfloor F_{pix}/(35 \times F_{sig}) \rfloor \\ F_{res} &= F_{pix}/k \end{aligned}$$

In practice the value of k is limited to some small value (currently 5, so there will be at least one point every 5 pixels) to avoid having too long straight lines.

In the other case, if $F_{pix}/F_{sig} < 35$, each x-axis pixel is assumed to represent a *range* of time, and we compute the minimum and maximum values the signal will take within that interval. The resulting data are then plotted as a series of vertical lines, one for each x-axis pixel. For X11 this is again more or less the best we can do. But this scheme doesn't work well when using Cairo if the signal doesn't contain significant high (relative to F_{pix}) frequency energy, and the resulting plot is reduced to a line instead of being a broader band of pixels. The result isn't much better than for X11 as we have in effect disabled Cairo's anti-aliasing capabilities. This situation arises if the waveform is monotonic within each time interval represented by a single pixel. Fortunately there is an simple solution, which is illustrated in Fig. 3.

In the right half of (a) we have a waveform that can be *assumed* to be representable by a

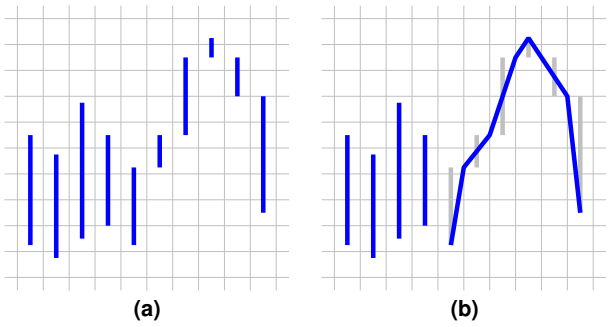


Figure 3: Connecting segments

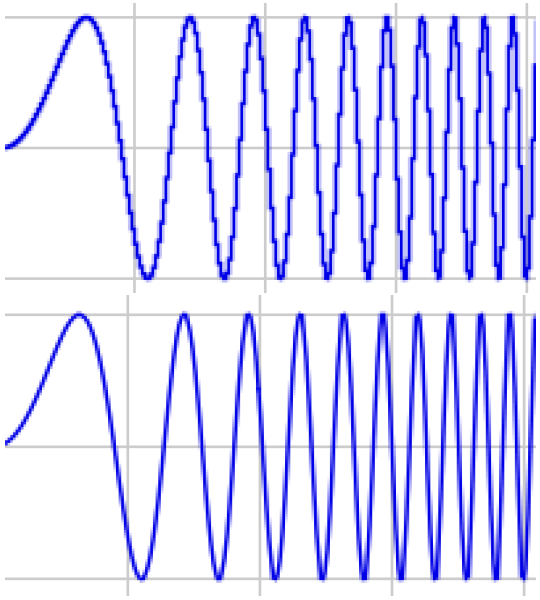


Figure 4: Visual effect of connecting segments

smooth line. In this case we can replace the vertical segments by connected lines just by moving the x-coordinates by half a pixel, and splitting the vertical segment at an extreme into two lines, as shown in (b). This only requires the original x, min, max data, and results in a dramatic improvement in display quality, as illustrated by Fig.4.

To compute the min, max pairs the display algorithm upsamples the original data by a factor of at least 6, and such that we have a sample on every border between two adjacent pixels — this ensures that there will be no gaps between segments. The extreme values can then be found using inverse quadratic interpolation. This is essentially the same algorithm used to trigger on a peak, except that the function value is computed instead of the argument, and considerably less precision is required.

In this case we have

$$\begin{aligned} k &= \lceil 6 \times F_{sig}/F_{pix} \rceil \\ F_{res} &= k \times F_{pix} \end{aligned}$$

The table below shows the resulting display parameters as a function of the horizontal resolution, for $F_{sig} = 48$ kHz, and 100 pixels per division. The SPP value is the number of samples (after upsampling) per horizontal pixel.

T/Div	F_{res}/F_{sig}	SPP
1 s	6.000000	2880/1
0.5 s	6.000000	1440/1
0.2 s	6.000000	576/1
0.1 s	6.000000	288/1
50 ms	6.000000	144/1
20 ms	6.041667	58/1
10 ms	6.041667	29/1
5 ms	6.250000	15/1
2 ms	6.250000	6/1
1 ms	6.250000	3/1
500 us	8.333333	2/1
200 us	10.416667	1/1
100 us	20.833333	1/1
50 us	41.666667	1/1
20 us	52.083333	1/1
10 us	52.083333	1/2
5 us	41.666667	1/5
2 us	104.166667	1/5
1 us	208.333333	1/5

In this example the switch between the two algorithms discussed above occurs between 100 and 50 usecs per division.

Note that in both these cases one sample per pixel is computed, but in a different way. For the first algorithm the single sample corresponds to the center of an horizontal pixel. For the second it is positioned on the border between pixels.

To obtain this exact alignment of the upsampled signal to the pixel grid we must initialise the phase of the polyphase filter used by the resampling algorithm to the required value. The current release of *zita-resampler* includes support for this.

4 Software structure

4.1 Data flow

Figure 5 shows the main elements of the implementation. Almost no work is done in the Jack callback, it just copies the input signals to a lock-free buffer. Apart from that it contains some code to support the manual trigger mode. All the rest is done in a non real-time context, so zita-scope will impose only a very light load on the Jack processing graph.

The lock-free buffer is around 1.5 seconds long. In single-trigger mode input is discarded until the user enables the next trigger, but the lock-free buffer it is used to store the last second of input. This ensures that this data is always available at the next trigger (which may be a manual one).

The trigger logic determines which part of the input is copied to the capture buffer. In continuous mode this will be little more than the displayed range — if the user changes the trigger position w.r.t. to the display range this is taken into account on the next trigger. In single-trigger mode the capture buffer can store up to a few seconds of data, allowing the user to examine any part of it. To allow triggering on a wide range of signal levels the input gains set in the GUI are taken into account by the trigger algorithms, but the signals written to the capture buffer are always the original ones without any gain applied.

The following step implements one of the two algorithms presented in the previous section, depending on the selected display range. These computations are performed when the contents of the capture buffer are updated by the trigger logic, or 'on demand' when the user changes the time axis parameters.

The plotting routines finally display the data on the screen. Any gain and vertical offset selected by the user are only taken into account at this point, so changing the these parameters does not require recomputing the display buffer data.

Some logic and state machinery is required to coordinate all of this. For example, in single trigger mode the display must be redrawn immediately if the user changes any parameters, while in continuous mode it could be better to wait until the capture buffer is updated.

4.2 Display markers

To perform accurate measurements zita-scope offers various types of on-screen markers, shown as vertical or horizontal dotted lines on the display. Their absolute and relative positions are also shown in numerical form. These numerical values are always computed from the original signal stored in the capture buffer, not from the

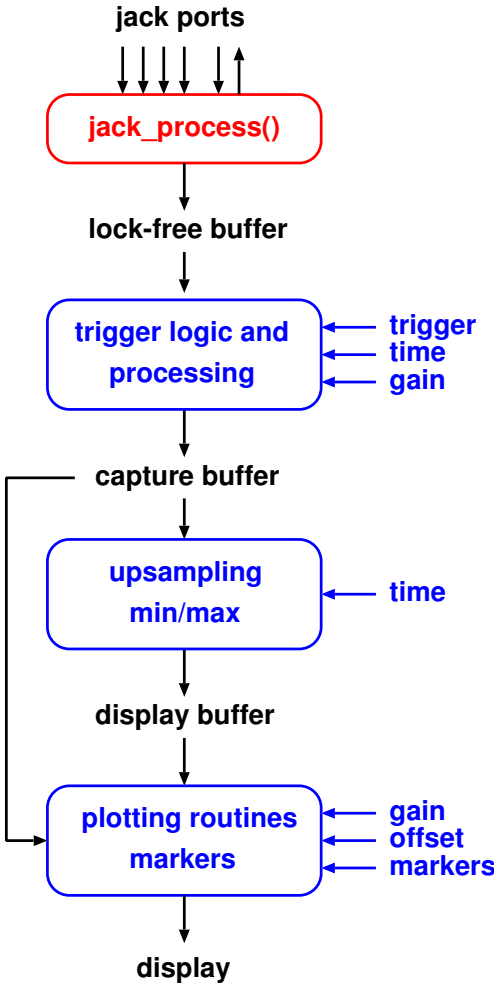


Figure 5: Processing flow

display data, and are not modified by any gain or offset settings.

Time axis markers can be positioned manually, or snap to a zero crossing or a peak, using the same algorithms as for triggering. Amplitude axis markers can be set manually, or they can follow the time axis ones on a selected channel, or snap to exact peak values. More complex measurements (RMS levels, spectrum,...) may be implemented in future releases of the application.

4.3 Additional facilities

Zita-scope offers some additional convenience functions:

- Storing and recalling the complete state of the instrument, including the capture buffer. The data is stored as a regular CAF audio file with the instrument settings in a dedicated GUID chunk.
- Creating a PNG file of the current display. For images to be included in printed documents the display background can be changed to white.

5 Acknowledgements

The author has contemplated writing an oscilloscope app for years, but kept postponing it until some Linux audio users got impatient and 'increased the pressure'. Without them zita-scope probably wouldn't exist.

Writing this application in the relatively short time it finally took was possible only because of the existence of some excellent and well-documented software taking care of some aspects, in particular Jack and Cairo.

A sincere thanks also to the (near future) beta-testers who will without doubt provide invaluable feedback and suggestions for improvements.

References

[1] T. Witham and B. Baccala, "Xoscope for Linux." `http://Xoscope.sourceforge.net/`, 2009. [Accessed 27/1/2013].

[2] K. Packard *et al.*, "Cairo." `http://www.cairographics.org/`. [Accessed 27/1/2013].

Ambisonics plug-in suite for production and performance usage

Matthias KRONLACHNER
Institute of Electronic Music and Acoustics, Graz (student)
M.K. Čiurlionio g. 5-15
LT - 03104 Vilnius
m.kronlachner@gmail.com

Abstract

Ambisonics is a technique for the spatialization of sound sources within a circular or spherical loudspeaker arrangement. This paper presents a suite of Ambisonics processors, compatible with most standard DAW[1] plug-in formats on Linux, Mac OS X and Windows. Some considerations about usability did result in features of the user interface and automation possibilities, not available in other surround panning plug-ins. The encoder plug-in may be connected to a central program for visualisation and remote control purposes, displaying the current position and audio level of every single track in the DAW. To enable monitoring of Ambisonics content without an extensive loudspeaker setup, binaural decoders for headphone playback have been implemented.

Keywords

Ambisonics, Plug-ins, Digital Audio Workstations, Binaural, Ardour

1 Introduction

It turns out to be difficult finding platform independent audio plug-ins for encoding and decoding Ambisonics. Following section should give a brief overview of still maintained plug-ins.

Fons Adriaensen's AMB-plugins[2] offer encoders and rotators (yaw axis only) until 3^{rd} order. LADSPA can be used with the DAW Ardour under Linux and Mac OS X. There is no Windows host supporting LADSPA. For decoding Ambisonics signals into loudspeaker feeds, Adriaensen's Jack client application AmbDec [Adriaensen, 2005] is often used.

Bruce Wiggins offers his WigWare[3] plug-ins in VST format for Windows and MacOS X. These processors include 2D and 3D encoders until 3^{rd} order as well as 1^{st} order decoders for several fixed loudspeaker arrangements.

Daniel Courville's[4] Audio Unit (Mac OS X only) plug-in suite offers 3D encoders for 1^{st} and 2^{nd} order as well as 2D encoders for 5^{th} order.

For the plug-ins created in this work, the C++ cross-platform programming library JUCE[5] [Storer, 2012] has been used to develop audio plug-ins compatible to most DAWs on Linux, Mac OS X and Windows. JUCE is being developed and maintained by Julian Storer, who used it as the base of the DAW Tracktion. JUCE is released under the GNU Public Licence. A commercial license may be acquired for closed source projects. It is possible to build JUCE audio processors as LADSPA, VST, AU, RTAS and AAX plug-ins or as Jack standalone applications. LV2 (LADSPA version 2) support currently has to be added manually from the separate project DISTRHO[6] [Coelho and Rodrigues, 2012].

Ambisonics suffers from different existing standards concerning channel order and normalization. To overcome this problem, a conversion tool is included in the plug-in suite. Encoders, rotators and decoders from the authors suite are designed for the ACN channel order and SN3D normalization, proposed by [Nachbar et al., 2011]. Conversion between standards of the input and/or output format can be done by the conversion plug-in.

Apart from platform compatibility, some considerations about usability did result in features of the user interface, not available in other surround panning plug-ins. Continuously rotating a sound source results in a discontinuity of the angular representation between $-180°$ and $+180°$. This jump is also reflected when drawing automation curves resulting in a mismatch between the visual representation and auditory perceived movement of a sound source. A solution allowing to define absolute starting points and angular velocities for relative movements is

[1] Digital Audio Workstation
[2] `http://kokkinizita.linuxaudio.org`
[3] `http://www.brucewiggins.co.uk`
[4] `http://www.radio.uqam.ca/ambisonic/`
[5] `http://www.rawmaterialsoftware.com`
[6] `http://distrho.sourceforge.net`

proposed.

For headphone monitoring several binaural decoders have been implemented simulating the Ambisonics half sphere of the medium sized IEM Cube with 24 speakers and the concert hall Mumuth[7] with 29 speakers in a elliptical stretched half sphere.

For visualization and external control purposes, a bidirectional Open Sound Control (OSC) communication layer has been implemented in the encoder plug-in. An external program is able to display the current position and audio level of every track in the DAW. The visualization program may also take control over the sources. This can be very useful in performance situations while having a multitrack playback coming from the DAW and a central display to control the position of the individual tracks.

Currently no audio plug-in format can handle dynamic input/output channel counts. Therefore all plug-ins may be compiled for fixed Ambisonics orders.

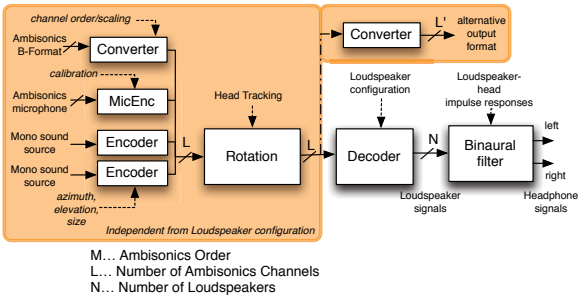


Figure 1: Ambisonics production and playback chain

2 Encoder

The encoder plug-in distributes a mono source signal into Ambisonics channels (derived from spherical harmonic functions), according to *azimuth* and *elevation* settings, representing the coordinates on a spherical surface. An additional parameter called *size* $[0, 1]$ may be used to adjust the spatial directivity. Adjusting the *size* parameter from *zero* towards *one* results in a scaling of the higher order components. For a *size* setting of 1, all Ambisonics channels will be zero except the 0^{th} order (also known as W channel), resulting in an equally distributed signal over all loudspeakers.

[7] `http://www.kug.ac.at/en/studies-further-education/studies/infrastructure/the-mumuth.html`

2.1 Automation parameters

Most current DAWs are limited to represent automation parameters between 0.0 and 1.0 along a time line. Panning plug-ins usually map this range for azimuth and elevation between $-180°$ and $+180°$. A full circle rotation results in a mismatch between the visual representation (Fig. 2) of the automation curve and the perceived continuous rotation of the source. Additionally the plug-in host may interpolate between a jump from 1.0 to 0.0, resulting in a very fast audible jump.

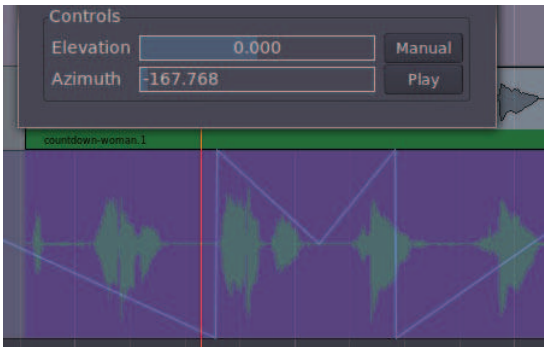


Figure 2: Automation curve for full circle rotation using Ardour and AMB plug-ins

To overcome this problem, automation parameters (Fig. 3) have been added for setting start points *(SetAzimuth, SetAzimuthRel)* and angular velocities *(tgl-rot-azimuth)*. The maximum speed of the angular velocity may be adjusted between 0 and $360deg/sec$ by an additional parameter (*max-speed*). This guarantees a wide range of adjustment and at the same time accuracy for the rotation speed.

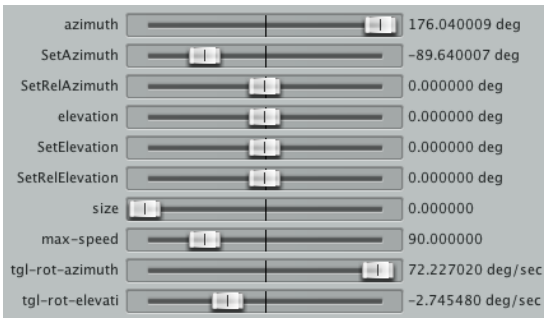


Figure 3: Encoder automation parameters

2.2 Remote control and visualization

Keeping track of all sound source positions within a DAW may be a difficult task for the mixing engineer. To allow a better overview

and control of the spatial scenery, a cooperative visualization and control unit has been implemented. All encoder plug-ins are equipped with a bidirectional OSC layer (Fig. 4), sending and receiving control and status parameters. Currently a functional prototype has been implemented in Pd/GEM (Fig. 5) displaying all tracks (encoders) on a sphere, including visualization of their audio levels. Currently the audio level is represented by the variable length of the cylinder representing a source signal. This concept may be extended to a more sophisticated implementation.

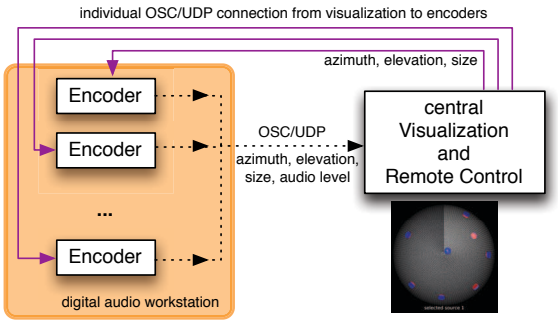


Figure 4: Encoder OSC communication with remote visualization and control

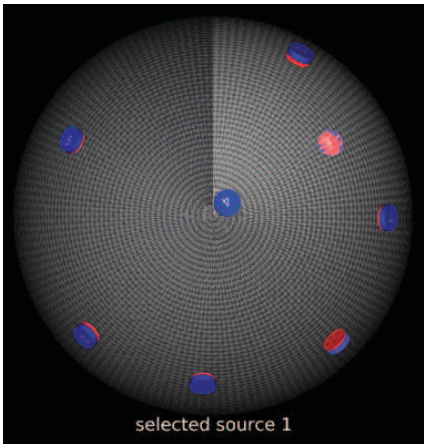


Figure 5: Visualization and remote control of several encoders with Pd/GEM

3 Rotator

The rotation plug-in (Fig. 6) may be used to manipulate the orientation in the Ambisonics domain, as described in [Musil et al., 2003]. An optimized way to calculate rotation matrices can be found in [Rumori, 2009]. Therefore yaw, pitch and roll can be adjusted by ±180°. This is very useful for the incorporation of head movements during binaural playback. The rotation plug-in listens to an adjustable UDP port for incoming OSC messages. This allows to bypass host automation and controlling the rotation directly from the head tracking software (Sec. 4.1).

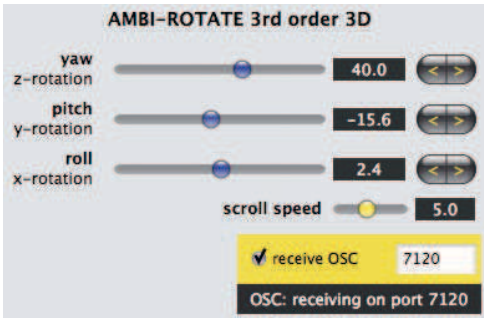


Figure 6: Ambisonics Rotator

4 Binaural decoder

The binaural decoder computes virtual loudspeaker feeds as a linear combination of the Ambisonics signals according to a given decoding matrix. These virtual loudspeaker signals are convolved with their individual stereo impulse responses, modeling the transfer path from the loudspeaker position to the left and right ear of the listener (Fig. 1).

4.1 Head tracking

Head tracking is a significant feature for virtual reality scenes and headphone playback. Small head movements change the relative position of a sound source in aspect to the listeners ears, making localization more easy and removing ambiguity.

The *Kinect*^{TM} sensor as add-on for the gaming console *XBox 360*^{TM} by Microsoft offers a low budget depth sensor. [Fanelli et al., 2011] developed a software to gather head orientation angles and the head position relative to the Kinect sensor (Fig. 7). The author extended this head pose estimation software about sending OSC data to the Ambisonics rotator plug-in[8]. The Ambisonics sound field has to be rotated in opposite direction to keep the sound source positions fixed and suppress the rotation with the listeners head.

[8] `http://github.com/kronihias/head-pose-estimation`

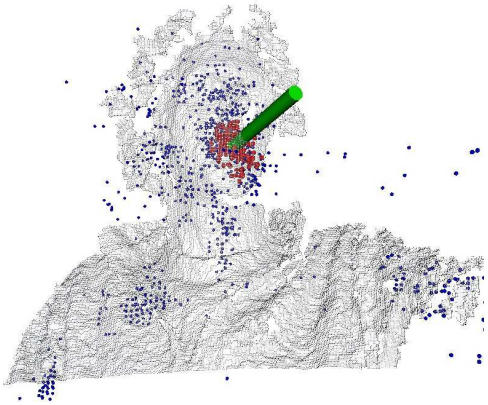


Figure 7: Head pose estimation

4.2 Matrices and virtual venues

To allow monitoring in various virtual scenes, different decoding matrices and sets of loudspeaker-to-head impulse responses were used.

4.2.1 Reduced decoder (3rd order 3D)

This binaural decoder uses specific symmetry relations between the HRTFs and decoder matrices to achieve a reduced set of head releated impulse responses, directly applicable on the Ambisonics Signals without computing virtual loudspeaker signals[Musil et al., 2007]. This approach reduces the numbers of convolutions and therefore the CPU load. The algorithm is implemented in the `iem_bin_ambi` library for Pure data.

4.2.2 IEM Cube and Mumuth decoder (4th and 5th order 3D)

The $120m^2$ IEM Cube (Fig. 8) serves as the main lab of the Institute of Electronic Music and Acoustics Graz. 24 Tannoy coaxial speakers are mounted in a hemispherical arrangement consisting of three rings (12 - 8 - 4). During the years of Ambisonics research at the IEM, several different approaches for finding an optimal decoder matrix have been taken [Zotter et al., 2012; Sontacchi, 2003]. The IEM Cube binaural decoder implemented in the authors Ambisonics plug-in allows to switch between the different available decoder matrices.

The Mumuth (Fig. 9) was opened in 2009 as multi purpose venue for the University of Music and Performing Arts Graz. It houses the $600m^2$ *György Ligeti* concert hall. The 33 Kling&Freitag CA 1001 SP loudspeakers (29 are used for the half sphere, 4 more speakers are located in the corners) may be arranged by a special motor controlled mounting, resulting in a versatile loudspeaker setup that can be changed within a minute. The Mumuth binaural decoder uses the Ambisonics decoder matrix and loudspeaker setting for the IEM Demosuite [Plessas and Zmölnig, 2012] by Thomas Musil.

The impulse responses of the IEM Cube and the Mumuth were recorded by Martin Rumori and David Pirrò as part of the artistic research project *The Choreography of Sound (CoS)* [Eckel et al., 2012].

Figure 8: IEM Cube

Figure 9: Mumuth

5 Ambisonics converter

Since the beginning of Ambisonics in the 1970s and the extension to Higher Order Ambisonics, several different approaches for arranging and normalizing the spherical harmonic components have been taken. A good summary may be found in [Nachbar et al., 2011] and [Chapman et al., 2009]. All plug-ins in this suite are operating with the ACN channel order and SN3D normalization, proposed in [Nachbar et al., 2011]. The Ambisonics converter plug-in allows to interchange the normalization schemes SN3D, N3D, Furse-Malham and the channel order schemes ACN, SID and Furse-Malham. Thus, it is possible to incorporate various standards into the production chain.

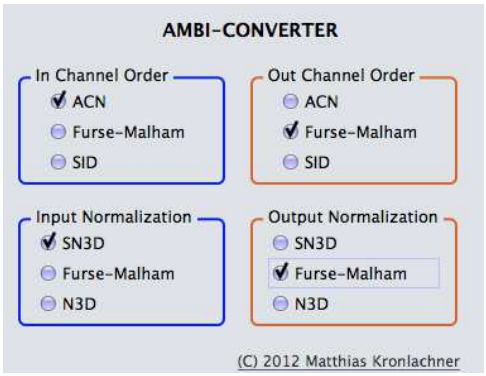


Figure 10: Converter plug-in

6 Summary

This paper presented an Ambisonics processor suite, usable in production or live performance. The JUCE framework proved to be a stable solution for developing cross platform audio plug-ins and standalone applications. By the time of writing this paper, no Mixed-Order systems have been implemented. Mixed-Order Ambisonics uses different orders for the horizontal and the vertical part of the sound field. It is planned to incorporate this into the converter plug-in.

7 Acknowledgements

The author would like to thank Winfried Ritsch, who advised the roots of this work as bachelor thesis. Many people from IEM supported this work by contributing with their valuable experience in talks, software and decoder matrices. Namely Peter Plessas, IOhannes Zmölnig, Franz Zotter and Thomas Musil. Special thanks also to Martin Rumori for providing the author with the impulse responses of the Cube and Mumuth. Last but not least thanks to Fons Adriaensen for his great work on Ambisonics software.

References

Fons Adriaensen. 2005. AmbDec - Jack Ambisonics Decoder for Linux und Mac OS X.

Michael Chapman, Winfried Ritsch, Thomas Musil, IOhannes Zmölnig, Hannes Pomberger, Franz Zotter, and Alois Sontacchi. 2009. A Standard for Interchange of Ambisonic Signal Sets. *Ambisonics Symposium 2009, Graz.*

Filipe Coelho and Antonio Rodrigues. 2012. DISTRHO, Cross-Platform Audio Plugins.

Gerhard Eckel, Martin Rumori, David Pirrò, and Ramón González-Arroyo. 2012. A Framework for the Choreography of Sound. *Proceedings of the International Computermusic Conference 2012, Ljubljana.*

G. Fanelli, T. Weise, J. Gall, and L. Van Gool. 2011. Real Time Head Pose Estimation from Consumer Depth Cameras. In *33rd Annual Symposium of the German Association for Pattern Recognition (DAGM'11).*

Thomas Musil, Johannes Zmölnig, Markus Noisternig, Alois Sontacchi, and Robert Höldrich. 2003. AMBISONIC 3D-Beschallungssystem 5.Ordnung für PD. *IEM Report 15/03.*

Thomas Musil, Alois Sontacchi, Markus Noisternig, and Robert Höldrich. 2007. Binaural-Ambisonic 4.Ordnung 3D-Raumsimulationsmodell mit ortsvarianten Quellen und Hörerin bzw. Hörer für PD. *IEM Report 38/07.*

Christian Nachbar, Franz Zotter, Etienne Deleflie, and Alois Sontacchi. 2011. AMBIX - A suggested Ambisonics Format. *Ambisonics Symposium 2011, Lexington.*

Peter Plessas and IOhannes Zmölnig. 2012. The IEM Demosuite, a large-scale jukebox for the MUMUTH concert venue. *Proceedings of the Linux Audio Conference 2011.*

Martin Rumori. 2009. Girafe a versatile ambisonics and binaural system. *Ambisonics Symposium 2009, Graz.*

Alois Sontacchi. 2003. *Dreidimensionale Schallfeldreproduktion für Lautsprecher- und Kopfhöreranwendungen.* Ph.D. thesis, Graz, Austria.

Julian Storer. 2012. JUCE (Jules' Utility Class Extensions).

F. Zotter, H. Pomberger, and M. Noisternig. 2012. Energy-Preserving Ambisonic Decoding. *Acta Acustica united with Acustica*, 98(1):37–47.

The Rationale behind Rationale: Designing a Sequencer for Unlimited Just Intonation

Chuckk HUBBARD
badmuthahubbard.com
Bucharest, Romania
chuckk.hubbard@gmail.com

Abstract

This article presents some of the considerations that went into determining how the Rationale Just Intonation sequencer should work. Various special problems that came about because of the number and nature of usable tones are discussed, as well as reasons for eschewing other existing notation systems. Programming specifics are ignored in favor of questions of interface design.

Keywords

Rationale, Just Intonation, xenharmonic, microtonal, composition

1 Introduction

Among the many things made vastly simpler by computers is the generation of tones of very precise arbitrary frequency. That should mean that composing in xenharmonic[1] tunings, i.e., unusual tunings, is now a piece of cake. It is, if the composer knows exactly what tones should be sounded and when, which is to say, if the piece has already been composed, away from the computer. Otherwise, there are text-based scores to edit manually and then process (e.g. traditional Csound score format, Scala's sequence file format[2]), or there are sets of predetermined possible tones that can be chosen from (e.g. most retunable soft or hard synths triggered by MIDI). Most existing options use one of these two methods to bridge the gap between the composer's creativity and the computer's power, i.e., to input notes.

There's nothing wrong with composing using text files; it can and does produce brilliant results. Still, it can only be a good thing for there to be more options, and for many people, seeing and dragging notes is more intuitive. As for sets of predetermined possible tones, there is theoretically no limit to the number of tones that could occur in Just Intonation, but the relationships between them are very important, so having too many of them available at once is almost as bewildering as having too few. You could set up a software synthesizer with thousands of tones per octave, but this would make it harder, not easier, to intuitively choose the right tone.

Just Intonation is a method of musical tuning whereby all frequencies have whole-number ratios[3] with all of the others. The simplest such ratio, which is the only one unchanged in the majority of tuning systems, is a ratio of 2 to 1, commonly called an octave. A tone with a frequency of 300 Hertz will sound an octave higher than a tone with 150 Hertz. A ratio of 3:2 is what we call a perfect fifth, e.g. a tone of 300 Hertz compared to a tone of 200. A ratio of 5:4 is a major third, but in 12-tone equal temperament, this interval is already noticeably detuned. Three tones in the ratio 4:5:6 will sound as a major triad, e.g., C E G.

Simple music using these tunings will probably not sound especially groundbreaking to most Western listeners. We have a strong innate tendency to sing simple harmonies and melodies in Just Intonation, even when struggling to sing in tempered tunings. But, a few hundred years ago, some composers began to crave more harmonic freedom to change keys quickly, without going out of tune, and using a small number of total notes-around the time of the advent of the keyboard. This last requirement excluded Just Intonation. Three successive major thirds in Just Intonation would go from 5:4 (C-E) to (5x5):(4x4) (C-G#) to

1 While the term 'microtonal' would be more immediately clear to many people, it has become irksome to most enthusiasts, since much of the music composed in xenharmonic tunings doesn't use microtones (smaller than semitones) at all. They sometimes occur in Just Intonation, but the music can hardly be defined by this element; the *large* uncommon intervals are just as important.

2 Scala's sequence file format: http://www.huygens-fokker.org/scala/seq_format.html

3 I.e., the ratio of any positive integer to any other positive integer.

(5x5x5):(4x4x4) (C-??[4]), that is, 125:64. An octave is 2:1, or 128:64, and no one wants to build accordions with separate keys for 128:64 and 125:64 and all of the possibilities in between. It was simpler to alter that 5:4 ratio so that applying it three times would give exactly an octave. By adjusting some of the intervals, the number of possibilities was deliberately limited for practicality.

Rationale[5] is a flexible sequencer for composing in extended Just Intonation, first released in 2008. It is free, licensed under GPL v.3, and depends on the Python interpreter, the Python Tkinter toolkit (included by default with the Python interpreter), and the Csound API for Python. It is theoretically cross-platform, but has only been tested extensively on Linux. The idea with Rationale was to have a graphical score on which notes can be placed that have certain frequency ratios to other notes, but that, at any time, the composer may wish to modulate, permanently or temporarily, or simply to see the relations between any arbitrary set of notes, not always including the starting tonic. The ratios of 3:2, 5:4, 6:5, and 7:4 are all options in most JI systems, but if one wants to make that 7:4 into the tonic of a new harmony, it may be necessary to include ratios like 49:32 (7:4 x 7:8), 63:32 (7:4 x 9:8), and so on. With experience, the meanings of those ratios become more obvious too, but there are no limits; some composers may be fine with using a ratio like (7^5):(2^5*3*5^3), i.e., 16807:12000, but again, more options is a good thing, and some composers will prefer something more intuitive.

Ben Johnston's system of accidentals, among others, is one way of making this more intuitive. No written ratios are needed. He starts with a diatonic C major scale and uses various symbols to notate specific adjustments. He uses the standard ♭ and ♯ familiar to most musicians, as well as + and -, ↑ and ↓, and small superscript numbers 7, 13, 17, and further prime numbers with their upside-down counterparts, to specify these adjustments next to standard notes on a staff.[6] He has composed extraordinary music this way, as have many others, but to understand the scores takes serious study; rows of accidentals may be placed in front of any one note to show how it is related to other notes with almost as many accidentals. It is perfectly usable, but I was looking for something else.

Harry Partch wrote that, “It is quite conceivable that an instrument could be built that would be capable of an automatic change of pitch throughout its entire range, up or down by any reasonable interval, and if Just Intonation can surmount the many hazards and problems ... the problem of transposition may be considered minor, one for which a solution will inevitably be found.”[1] He didn't seem to devote much energy to inventing such a retunable instrument, preferring tuned reeds, fixed-length strings and stuck percussion. If I had lived at a time when computers didn't exist or were very expensive, I probably wouldn't have tried to create such an instrument either. But that comment of his seems very compelling at a time when GUI programming is as advanced as it is today.

My decision to create a Just Intonation sequencer with automatic tonality changes, without accidentals and with an undetermined number of tones led me to a series of unusual decisions during the creation of Rationale.

2 Automatic tonality changes

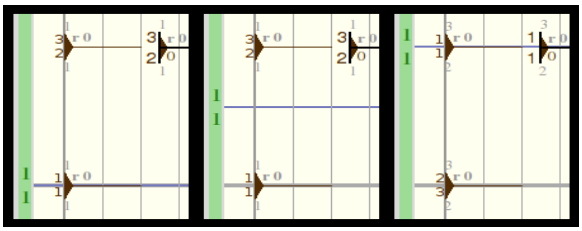


Figure 1: Before, during and after a tonality change. The two forms on the left in each frame are entered notes and the one on the right is the hover.

By “automatic tonality changes”, I actually mean redrawing existing notes with different reference points, but the same frequencies. The notes on the screen do not move, but the ratios change and the horizontal reference lines slide to new positions. In traditional theory, this could be compared to transposing by, perhaps, a diminished second: same sound, different note names. This is useful for several reasons. A composer may want to modulate (multiply) repeatedly during a piece, and, as mentioned, if you modulate e.g. by a major third three times, you arrive at 125:128, not 2:1. With Rationale, I wanted to be able to always be looking at small-number intervals. This “modulation” could also be very temporary, just for some short phrase to be repeated at different frequencies before returning to a broader theme-something that is a piece of cake in traditional notation.

There is a kind of secondary mouse cursor used in Rationale for placing notes. As the mouse moves, it cycles through a set of ratios, and a click places a note at the appropriate height. In the

4 This would be 41 cents below C, or 41/100ths of a semitone.

5 Rationale : rationale.sourceforge.net

6 http://en.wikipedia.org/wiki/Just_intonation#Staff_notation

documentation, this cursor is referred to as the *hover*. The hover has another important purpose, which is to select the ratio that will become 1:1 (the tonic, representing a 1:1 frequency ratio with the reference frequency) after a tonality change. The tonality change is accomplished by hitting 'T' on the keyboard when the hover is at the desired ratio. All existing notes' ratios change and the hover's ratio becomes 1:1, wherever it is. Afterwards, the hover can be moved to a new ratio and another tonality change performed, again and again. This idea was evidently subconsciously inspired by exposure to the concept of *movable do*.[7] You can always return to the original 1:1, which is by default middle C.

2.1 Tonality regions

The problem that arises when jumping arbitrarily from *do* to *do* is that the earlier notes entered with simple ratios often show more and more complex ratios as *do* changes. The computer doesn't care, but the composer should be able to see the simple relationships in different "keys" easily; not all notes need to change. The solution was tonality regions. Holding down 'R' and typing a number will switch the hover to that region; future tonality changes will only affect notes in that region. Every note shows its region next to a small letter 'r'. This region symbol can be assigned different colors and shades to be more visibly distinct.

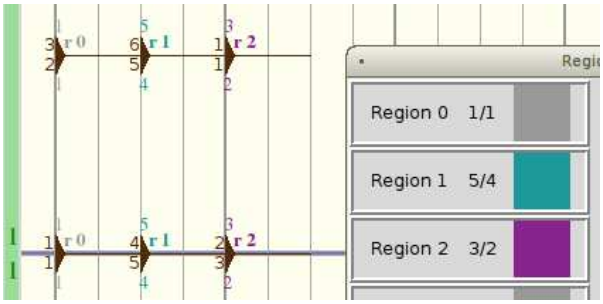


Figure 2: Three different regions with the region dialog. The entered notes sound exactly the same.

3 Decisions influenced by the number of tones

Many decisions were influenced by the need to differentiate between a large number of tones within a small space. As mentioned, others have added more variations of accidentals to traditional notation. In turning away from accidentals, Rationale cannot fit an octave into one inch, with five lines and four spaces. Even with the ratio for each note displayed by the note, the resolution of the screen would force some ratios to be skipped while moving the hover.

3.1 Universal staff

It is extremely unwieldy to show each instrument on a separate staff when each instrument requires a large amount of screen space. The concept of a staff is thus almost completely absent from Rationale. Horizontal guide lines are present but are all one octave apart. There is one extra horizontal guideline that always remains at middle C while the rest move according to the current position of 1:1. All of the instruments are represented in the same area and frequency is vertically absolute, i.e., a vertically higher note always has a higher frequency (although what that frequency actually means depends on the design of the instrument). In very simple songs, this causes no problems, but in more involved compositions, notes from different instruments are hopelessly mixed around each other, even superimposed over each other. The benefits of a universal staff are that one can see everything that is happening and it is really quite intuitive; the instruments are not isolated from each other. And, of course, there is room to visually differentiate between hundreds of possible frequencies. I didn't invent the idea; some of Stockhausen's famous scores, e.g., represent sound the same way, albeit without printed frequency ratios. Rationale has several features intended to resolve various problems presented by this universal score.

3.1.1 Shortcut for switching instruments

In typical sequencers, the instrument being entered depends on which staff the mouse approaches. That's impossible with only one staff, so the next most efficient way is to switch instruments with a keyboard shortcut. That shortcut is to hold Shift while typing a number. Rationale can easily provide over 1,000 instruments.

3.1.2 Different colors for instruments

The most obvious need with a universal staff is simply to see which instrument a note belongs to. Each instrument starts medium gray, and colors can be assigned to them arbitrarily. Typically, various instruments are all assigned sharply contrasting colors, but it is up to the composer. The structure of a piece can be easily visible with such a setup. So instead of different instruments'

7 Where, upon modulating to a new key, the tonic of the new key is treated as *do* in solfeggio, rather than saying, for instance, that you have modulated to *re*.

notes looking the same but being in different places, they are all in the same place and look different.

3.1.3 Independent horizontal and vertical zoom

Rationale also uses proportional notation, meaning the duration of a note is represented by its physical length, but this was more a matter of simplicity than a major decision. Still, with notes arbitrarily close to each other, it was necessary to be able to manually change the vertical scale without affecting the horizontal, and vice versa. Horizontal zoom is changed with -/+, zooming in and out respectively, and with Backspace, which resets the zoom to default. Vertical zoom is controlled the same way but while holding the Shift key. Such separate zoom control is surprisingly useful. It amounts to more control over how you see your music.

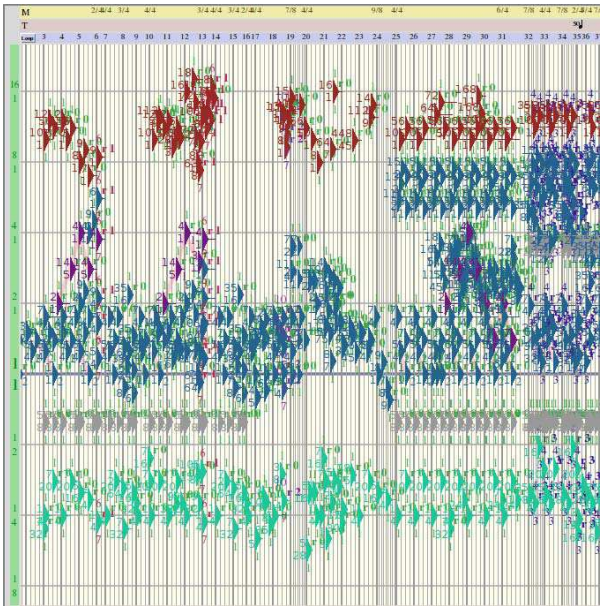


Figure 3: The universal staff, zoomed out both horizontally and vertically to show structure

3.1.4 Mutually exclusive horizontal and vertical note dragging

Rationale snaps notes to a customizable rhythmic grid, but the vertical grid, representing frequency, is normally much finer: at times, a movement of one pixel changes a note's ratio.[8] It would be easy to accidentally change frequency when trying to change time. For this reason, clicking on a note in EDIT mode only allows it to be dragged vertically. Holding the Shift key and dragging the note only moves it horizontally. This allows broad mouse gestures for moving notes drastically without changing their ratios, or without changing their times. Large groups of notes can be moved this way.

3.1.5 Hiding / showing specific instruments

Sometimes zooming in isn't enough to make all the notes clearly visible. It wasn't until I had worked with a fully functional Rationale for a while that I realized that two or more instruments might have to play the same frequency at the same time, and that, with the then-current incarnation of the program, which instrument's note ended up visible (and editable) was arbitrary. Being able to instantly hide all notes from any instrument seemed the best solution available. This is achieved by holding the Alt key and typing the number of the instrument. If an instrument is hidden when selecting a broad area, its notes are not selected. However, if previously selected notes are hidden, they remain selected and are still affected by any operations performed while they are hidden. The Alt key + the instrument number will show that instrument again, and Alt + 'S' will show all instruments. The only other way I could think of to manage superimposed notes of different instruments would have been with a three-dimensional score, which is beyond my programming abilities.

3.2 Notebanks

It was mentioned in the introduction that Rationale is an alternative to using accidentals, but also an alternative to finite, predetermined sets of tones. Part of the escape from predetermined sets is in the arbitrary tonality changes, and part of it is in notebanks. The default set of frequency ratios through which the hover will cycle has 57 possibilities in an octave, but this can be customized easily, and separate banks of ratios can also be remembered and switched between. This also allows, for instance, having higher prime-limit ratios set aside for when needed, or simply dividing a set of ratios into multiple smaller sets. For example, One's default notebank could include ratios like 1:1, 9:8, 5:4, 3:2 and 7:4, and a separate notebank could have higher harmonics like 17:16, 19:16 and so on. This is one way to have less notes available at any moment, allowing one to zoom out more vertically. Regardless, the idea is that there is no predetermined set of possible tones. The only limit is that the ratios must use only positive integers.

Holding the Control key and typing 'B' will open the notebank dialog. From there, notes can be

8 The horizontal rhythmic grid is, as mentioned, customizable, and quantization could be set to 1/9999 of a quarter note, for example, so it could be made finer than the vertical frequency grid if desired.

traded in and out of the chosen notebank, and more notebanks can be created. It is possible to specify here arbitrary whole-number ratios. When finished, it is possible to switch between active notebanks by holding 'B' and typing a number. The hover will then cycle through the ratios listed in the active notebank until it changes again.

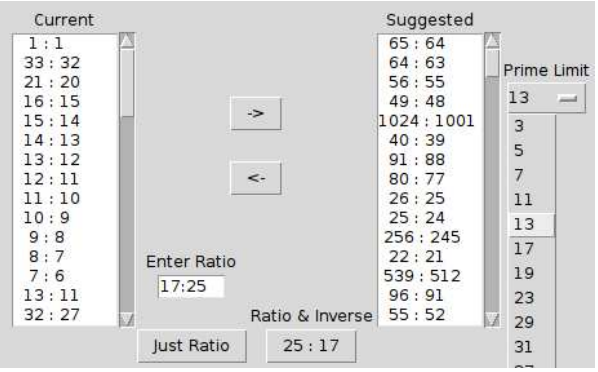


Figure 4: The notebank dialog. If an arbitrary ratio is typed, there is an option to add just that ratio and an option to add that ratio and its inverse. All ratios will be adjusted to be between 1:1 and 2:1

Rationale maintains a list of mouse positions for the ratios in the notebank. If several ratios are very close to each other, Rationale will try to make them each fit somehow, in the sense that the mouse should not skip any of them as long as there are as many pixels within an octave as there are ratios in the active notebank. If you have 40 pixels in an octave and 20 ratios, but all very close to each other, the hover shows them farther apart than they really are so they can all be accessed, but entered notes will appear at the correct height.

4 Output options

Rationale has no ambitions of becoming an audio processing tool. The Csound engine is used for all audio and all timing. What to do with the note data once it is composed is basically left to the user, although it can be sent out in several forms.

The output dialog appears upon pressing 'O'. The first tab displayed is for loading or typing a Csound csd file. There is an option to automatically reload this file every time playback starts, which creates the possibility of looping a segment and editing the csd file in a text editor in the meantime.[9] A new tab is created in this dialog every time a new instrument is created. Each instrument's tab has the possibility to add numerous outputs, which can be Csound instruments, Soundfont programs, or OSC commands.[10] Any time an instrument is called, it will send messages to all outputs listed on its tab, unless they are muted. It is possible to create outputs for different combinations of Rationale instruments to Csound instruments, and to selectively mute them, in order to quickly switch parts between instruments.

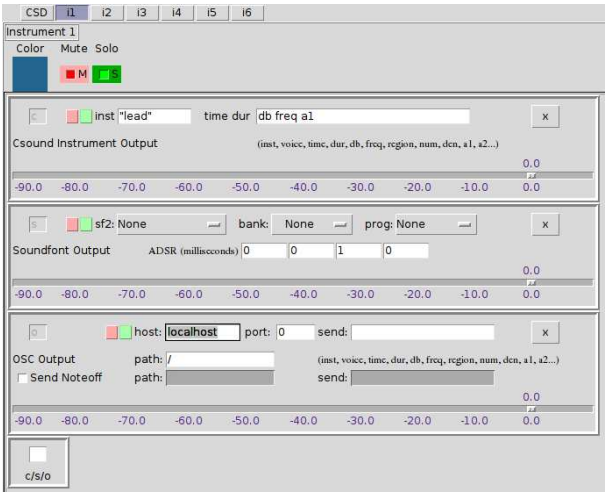


Figure 5: A Csound output, a Soundfont output and an OSC output.

Most of the elements of these outputs are standard for sequencers, with one exception: Csound and OSC outputs have message boxes to construct the messages to be sent with each note. Csound outputs by default send the time and duration of the note first, and after that any aspect of the note may be added in any order. Things like the instrument number, numerator, denominator, or region can be sent with each note. There may also be arbitrary fields, entered as a1, a2, a3, etc. These arbitrary fields can be assigned per note by right-clicking on the score. This allows fine control of Csound's many complex forms of synthesis. Granular synthesis, for example, needs much more information than the time, length, frequency and loudness of each note. The ability to draw automations for these values is a goal, but a distant one. OSC outputs have similar custom message-building, while Soundfont output has the usual Soundfont options.

5 Conclusion

One driving goal behind Rationale was to make the composition process in extended Just

9 The mechanism of looping, like many others, is fairly standard among sequencers, and so not addressed in this paper.

10 Unfortunately, MIDI output has proven too complicated so far. Many composers have asked about it, and it would open up many possibilities for sound production, but so far it has not materialized.

Intonation as intuitive as possible, in the hopes that more composers, who may not have initially been curious, would be drawn to experiment with the tuning system. The kinds of gestures they use in their other works or on their own instruments should be relatively simple to achieve in Rationale. This paper should have made the decisions involved in creating that environment a little clearer.

Using Rationale involves many other features, but most of them are more or less standard. This paper addresses mainly features that resulted specifically from the demands of composing in unlimited Just Intonation, namely the large number of tones and the inclusion of all instruments on one staff. Selecting notes, copy/paste, saving, undo/redo, transport, meter and tempo changes, note loudness, looping, audio options and the four basic modes (ADD, EDIT, DELETE and SCRUB) are all present but are not specific to this scope. They are documented in the Rationale help file.

6 Acknowledgements

Many thanks to all of the Csound developers, and in particular Victor Lazzarini, John ffitch, Oeyvind Brandtsegg, Rory Walsh, Michael Gogins and Steven Yi. All have explained things to me and/or fixed things for me over the course of several years, asking nothing in return. Thanks also go out to the xenharmonic community, including Dr. John Chalmers, Aaron Krister Johnson, Carl Lumma, Toby Twining and many others.

I must also thank my wife Irina for her love and support and for not throwing my computer in front of a subway train while I was programming Rationale.

References

[1] H. Partch. 1974. Genesis of a Music. Da Capo Press, New York, New York.

Chino – a framework for scripted meta-applications

David ADLER
david.jo.adler@gmail.com

Abstract

Chino is presented, a framework for creating meta-applications from Linux audio and Midi tools. It provides command line options to create or open sessions, a runtime user interface for adding, restarting or removing applications and a hand-editable file format to which sessions are saved. Graphviz is used to optionally display the layout of a session.

Chino itself is a Bash script that just provides generic functionality, users can create presets to implement what is desired for their use cases. Presets are prototypes for sessions, multiple sessions can be derived from a preset.

A preset is made up of a number of applications, each defined as a program together with its usage. For every application, the preset contains required application files and a library file that, via variables and functions, defines how the program is to be started and interconnected. Defining applications together with their connections results in dependencies, which are tied via user-defined port-groups.

In this paper, we will explain the architecture of Chino and take a look at some implications and limitations of this session management model.

Keywords

Linux audio, Bash, session management

1 Introduction

Chino[1] is yet another approach to session management for Linux audio. It is geared towards applications using the Jack Audio Connection Kit (Jack) for audio and either Jack or the Alsa Sequencer for Midi.

The modularity of UNIX/Linux software is proverbial and usually well received. In the realm of Linux audio, however, this appreciation hat its limits, as manually restoring modular sessions quickly becomes prohibitively complex. Consequently, users often complain about the lack of a comprehensive session management system.[2]

Regardless of complaints, some progress is taking place. Jack Session, the LADCCA/LASH/LADISH lineage and the Non Session Manager are currently coexisting[3] and the number of applications supporting one or more of them is steadily increasing.

Those session managers are capable of storing and restoring an arbitrary setup, as long as the applications involved are supported in some way. A Chino session, in contrast, is restricted to a limited number of setups prepared by the user via presets. No support for a protocol by applications is required, any application capable of restoring a previous state by command line options and/or file loading can be used.

Once a preset is prepared, usage is dead simple. No manual connection making is involved and all files belonging to a session are automatically placed in an ordered manner below one base directory.

In section 2 we will go through some underlying concepts, followed by a description in section 3 of how they are implemented. Those two sections together explain the architecture of Chino; providing useful knowledge for creating custom presets. While we can create sessions based on an existing preset without such a level of understanding, restricting oneself in that way neglects one of the main features of Chino, which is customisability. In section 4, eventually, we will take a look at some of the implications and limitations of the presented design.

[1] The application, online documentation and an example preset are available from `http://www.chino.tuxfamily.org`.

[2] In February 2013, Dave Phillips started a thread on the Linux Audio Developer (LAD) mailing list with the subject line *"So what do you think sucks about Linux audio?"* that pretty much confirms existence of those complaints. `https://lists.linuxaudio.org/mailarchive/lad/2013/2/5/196481`

[3] Dave Phillips' article *"A brief survey of Linux audio session managers"* from January 2013 on LWN.net gives a good overview. `https://lwn.net/Articles/533594/`

2 Concepts

This section covers a number of concepts underlying the design of Chino, without going too much into detail. Having these concepts in mind will aid us in understanding the subsequent section covering implementation.

2.1 Sessions

While Chino does manage *sessions*, the term session management is somewhat misleading. As stated in the introduction, Chino lacks the ability to just save and restore any setup involving supported applications. Sessions in Chino might be better described as "instances of a meta-application" or as "patch files" to which the meta-application saves its state.

2.2 Presets

A *Preset* defines the meta-application of which the sessions are instances. Chino, the core script, only provides generic functionality. Whenever running a session, it needs to be pointed to a preset. Figure 1 shows the relations between Chino, presets and sessions.

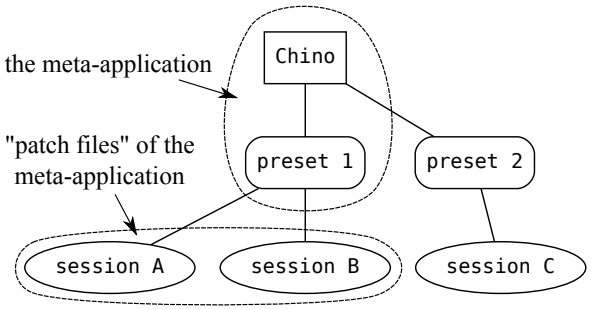


Figure 1: A preset—together with Chino—can be viewed as a meta-application. Sessions derived from that preset then are the "patch files" saved by the meta-application.

It is up to the user to create one or more presets in order to cover desired use cases, a default preset is provided as an example or starting point.[4]

2.3 Applications

A preset defines a set of *applications*. An application—in Chino—consists of three things:

1. the actual *program* used (like amSynth or Pure Data);
2. *application files* belonging to the program (patch files, configuration files and the like, if any);
3. an *application library*, a text file in Bash syntax defining how the program is to be started and interconnected.

While the programs—of course—are installed to the operating system, application files and application libraries are part of the preset.

One program can serve as several applications. We could define two applications `amssynth` and `amsfilter`, both using the program AlsaModularSynth—in one case as a synthesiser and in the other case as a filter.

Henceforth, we will use that distinction between the terms *program* and *application* throughout this document.

2.4 Methods

Applications are grouped into *methods*, categories for applications that can be handled in similar ways. Methods are defined in *method libraries* that are also part of the preset.

Two *method types* are hard-coded into Chino: *unique methods* and *channel methods* (see figure 2).

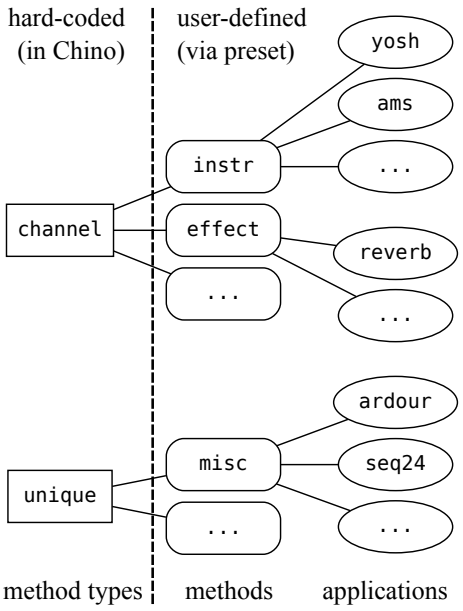


Figure 2: Method types, methods and applications.

- For **unique methods**, application names are assigned to a consecutively filled array of variable length. Entries must be unique, thus the name.

[4] The default preset is documented on `http://chino.tuxfamily.org/preset.html`.

- For **channel methods**, application names are assigned to indices of a fixed-sized array, where indices may be left empty. Applications need not occur uniquely within a channel method. An obvious use case (though not the only one possible) is using the index to connect an application to a certain audio or Midi channel, thus the name.

Introducing that extra layer of methods has two main advantages. First, methods simplify adding support for applications to a preset, as it is often the method library that does most of the work. Secondly, having channel methods allows for some desirable flexibility in arrangement of the connection graph.

2.5 Templates

In addition to pointing to a preset, we may optionally point to a *template session*. Any session derived from that same preset can serve as a template. To illustrate this, figure 3 shows a version of figure 1, modified to include a session that points to a template.

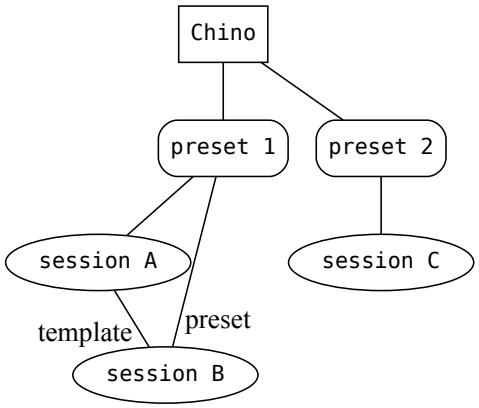


Figure 3: `session B` uses `preset 1` as a preset and `session A` as a template.

2.6 Inheritance...

2.6.1 ...of application files

On running a session, missing application files are copied from the template or preset, in that order of precedence.

2.6.2 ...of libraries

Libraries are sourced from the local session, the template or the preset, in that order of precedence.

Per default, application libraries will remain with the preset. Unlike application files, they will not get copied to the local session.

For custom application behavior on a per-session base, an option exists to "localise" a library. For making a session self-contained, an option exists to localise all used libraries. Self-contained sessions point to themselves as a preset.

Libraries local to the template, however, will get localised automatically; to not disrupt matching pairs of application files and application libraries for future child sessions.

2.7 Session hierarchy

Since applications are defined together with their connections, they may *depend* on other applications *providing* for certain ports to connect to. This leads to a hierarchical session layout, a dependency tree.

Ports provided by the sound card and Midi hardware (those devices are made applications as well) will usually form the root of the hierarchy. More layers can then be added to form a virtual studio to the user's liking. A mixer might for instance form the layer on top of the hardware ports, instruments and effects can then be connected to the mixer.

Applications do not depend on other applications, they rather depend on or provide for user defined *port-groups*. A port-group is just a name for a set of *port-variables* to which the real Jack/Alsa port names are assigned.

That way, applications providing for the same port-group can be exchanged without breaking the session. We could define two applications `seq24` and `nonseq` both providing for a port-group `SEQ`; then either can be used as a sequencer without making any further changes to the session.

Just as applications, methods may also depend on and and provide for port-groups. In the context of dependencies, we will sometimes use the term *nodes* when referring to anything that can depend or provide, i.e. either methods or applications.

Dependencies are handled separately for audio and Midi, so the place of a node in the dependency tree is defined by four lists of port-groups: *audio depends*, *Midi depends*, *audio provides* and *Midi provides*. Collectively, we will refer to them as *anchors*. We can make depends optional by prefixing them with a colon, this just suppresses the warnings otherwise displayed for unsatisfied depends.

Figure 4 shows the way a node is represented

in the session graph (or dependency graph) that Chino displays using the Graphviz software, figure 5 shows the graph of a small session.

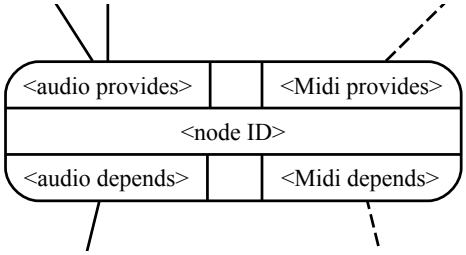


Figure 4: Representation of a node and its anchors in the dependency graph. Solid lines represent audio dependencies, dashed lines represent Midi dependencies.

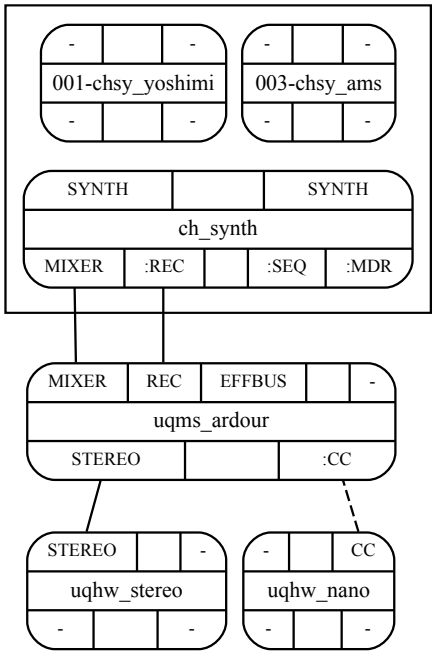


Figure 5: A session graph. Three unique method applications are started: `uqhw_stereo` (a stereo sound card), `uqhw_nano` (a Midi controller) and `uqms_ardour`. The channel method `ch_synth` has anchors, so it gets its own node in the graph and its applications `chsy_yoshimi` and `chsy_ams` drawn inside a cluster.

3 Implementation

Chino is written in the Bash scripting language. This section goes into some detail on how it is implemented.

3.1 Names

Rather strict rules exist regarding names of sessions, methods, applications and port-groups. The rules are ruthlessly enforced, as those names are part of file paths, variables or functions. Allowing special characters would just not work and would be potentially harmful.

- *Session names* may only consist of letters, numbers and underscores.
- *port-group names* may only consist of letters and numbers. They must be unique within each connection type (audio and Midi).
- *Method names* may only consist of letters and numbers. The first two characters of a method name must be unique within each method type, those characters make up the *method acronym* (consequently, method names must be at least two characters long). *Method IDs* are the method names prefixed with `uq_` for unique methods and `ch_` for channel methods.
- *Application names* may only contain letters and numbers, names must be unique within their method. *Application IDs* are the application names prefixed with `uq<method_acronym>_` for unique methods and `ch<method_acronym>_` for channel methods.

Given that nomenclature, all method IDs and application IDs will be unique strings within the set of all methods and applications.

3.2 The session directory tree

To facilitate automated copying of application files on creating or expanding a session, we must make sure the directories and files of a preset are named according to some rules.[5]

When running a session, its directory tree will automatically get filled and supplemented with files from the preset or template. At that point, thus, we don't need to meticulously follow naming rules anymore—Chino does it for us.

The base directory of a preset or session holds one subdirectory `<application_ID>` for every application. For presets, that is one for every implemented application; for sessions, that is one for every application that is or ever was part

[5]Naming rules are comprehensively covered in the online documentation. `http://chino.tuxfamily.org/documentation.html#file`

of the session (obsolete files do not get deleted by Chino, as they might not be obsolete from a user's perspective). Those application subdirectories hold the application files; applications not needing files do not get such a directory.

The base directory of a session also holds the *session definition file* `<session_name>.sdef` to which Chino saves the session.

3.3 Libraries

A `libs` directory, below the base directory of a preset, contains all libraries.

A file `libs/<session_name>-listlib` is the "root-library" of a preset. It holds arrays of allowed methods and applications and sets the initial array sizes for channel methods.

Methods and applications listed in `listlib` are each defined in their own library file, method libraries or application libraries respectively.

3.3.1 Variables in libraries

Via variables, all libraries define anchors of the node they represent.

Method libraries additionally have a variable that allows to give them a custom option in the rutime user interface. In the default preset, the `ch_dssi` method uses that feature to let users split the configuration of ghostess into the the parts belonging to the single DSSI plugins.

As shown in the example below, application libraries additionally have variables defining a number of properties: whether the application comes with application files; whether a function is required for adapting application files to their new names after copying; what Midi system to use and whether to disconnect autoconnected ports.

```
FILE_uqms_seq24='true'
MOVE_uqms_seq24=''

APRO_uqms_seq24=''
ADEP_uqms_seq24=''
MPRO_uqms_seq24='SEQ'
MDEP_uqms_seq24=':KBD :CC'

MIDI_uqms_seq24='alsa'
AUTO_uqms_seq24=''
```

3.3.2 Functions in libraries

Via functions, methods and applications accomplish the rest of the work; like a solid-state Bash session script ripped into pieces, those pieces then being called on demand by Chino.

Method libraries must provide a number of mandatory functions; requirements for application libraries are largely determined by their method.

If an application or a method has any anchors defined, this triggers the requirement for additional functions to be present. Details on those functions will follow in sections 3.6.3 and 3.6.4.

3.4 Steps

To illustrate what *steps* are about, it is useful to picture the "lifecycle" of an application (not in terms of code development but in terms of running the application).

When manually running an application, the single steps to be accomplished will be something along the line of:

- starting the program;
- establishing audio connections;
- establishing Midi connections;
- making music (the "tweak-and-save loop");
- quitting the program.

The attempt to adapt those steps to match the requirements of Chino led to the following list of steps:

- `assign` for assigning an application to an array and sourcing application libraries;
- `check` for checking whether an application file is present, if applicable;
- `list` for displaying a summary to the user;
- `copy` for copying and renaming the application file if the above check was negative;
- `start` for starting the program, includes assignment of port-variables;
- `acn` for establishing audio connections using the assigned port-variables;
- `mcn` for establishing midi connections using the assigned port-variables;
- the "tweak-and-save loop"(not a step);
- `unassign` for unassigning and killing an application when removed using the runtime user interface, or `kill` for killing an application on quitting the session.

For each of those steps, a method library must provide a so-called *step function* `s_<method_ID>_<step_name>()`. Chino simply calls the appropriate step function from the method library whenever a step needs to be done for one of the applications belonging to that method.

3.5 Tasks

A *task* is just a series of steps that accomplishes something useful.

Tasks may be vertical, calling a number of steps for one application, e.g. for adding an application to a session or for restarting an application (see table 1).

step	a1	a2	a3	a4
assign			s2	
check			s3	
list			s4	
copy			s5	
start			s6	
acn			s7	
mcn			s8	
unassign			s1	
kill				

Table 1: Illustration of a vertical task. An application a3 is restarted in a session consisting of applications a1 to a4. The task runs steps `s1` to `s8`.

Tasks may be horizontal, calling one step for all applications, e.g. for re-establishing all audio connections (see table 2).

step	a1	a2	a3	a4
assign				
check				
list				
copy				
start				
acn	s1	s2	s3	s4
mcn				
unassign				
kill				

Table 2: Illustration of a horizontal task. Audio connections are re-established for all applications a1 to a4. The task runs steps `s1` to `s4`.

Tasks may be both horizontal and vertical, e.g. for opening a session (see table 3).

Tasks are part of Chino, so users won't be bothered with them. The appropriate tasks get called whenever a session is opened or closed or when applications are added, restarted or removed.

3.6 Helper functions

Helper functions, prefixed with `h_`, exist for every step except `unassign`. Helper functions are never called by Chino itself, they are just tools

step	a1	a2	a3	a4
assign	s1	s2	s3	s4
check	s5	s6	s7	s8
list	s9	s10	s11	s12
copy	s13	s14	s15	s16
start	s17	s18	s19	s20
acn	s21	s22	s23	s24
mcn	s25	s26	s27	s28
unassign				
kill				

Table 3: Illustration of a task that is both horizontal and vertical. A session consisting of four applications a1 - a4 is opened. The task runs steps `s1` to `s28`.

available to accomplish steps in standardised ways; to be called from the step functions inside the method libraries.

The design of Chino attempts to find a reasonable balance between being as generic as possible and making implementation of presets as simple as possible. The combination of methods and helper functions serves that goal.

In most cases, using helper functions is desirable, resulting in a method doing barely more than calling the appropriate helper functions with appropriate arguments inside its step functions, as seen in the following example.

```
s_ch_synth_copy()
{
    declare -ri chan=$1
    h_copy ch_synth $chan
}
```

Nevertheless, it is also desirable to have the freedom of not using helper functions or using them in non-standard ways, to adapt to non-standard use cases. Three of the methods implemented in the default preset are non-standard in that sense:

- the unique method `hw`, for hardware devices, neither using `h_copy` nor `h_start()` as those are not applicable to hardware;
- the channel method `dssi`, which uses one instance of ghostess to harbour all DSSI plugins assigned as applications to its channels;
- the channel method `senv` (from **s**ynthesis **env**ironments) for instruments capable of multi-channel audio and Midi, in which the program is started just once for all instances of the same application.

A few helper functions deserve a closer look, since they implement essential functionality within Chino.

3.6.1 h_assign

`h_assign()` sources libraries, thereby implementing the inheritance rules for application libraries described in section 2.6.2.

3.6.2 h_copy

`h_copy()` copies application files, thereby implementing the inheritance rules for application files described in section 2.6.1. Application files may be single files or directories containing files.

3.6.3 h_start

`h_start()` starts the program and calls the appropriate *assignment functions* from the method libraries and application libraries.

In the assignment functions, we assign real Jack/Alsa port names to the appropriate port-variables, a prerequisite for later having the connection graph established.

For both anchor types—audio and Midi—a respective assignment function is required whenever the node has any anchors of that type.

To illustrate how `h_start()` works, let us look at a case where it gets called for an application that has audio anchors, thereby triggering the necessity for Jack audio port assignment.

1. a snapshot of audio ports is taken via `jack_lsp`;
2. an `<application_ID>_start()` function gets called from the application library, starting the program;
3. After the new ports have appeared, another snapshot of audio ports is taken.
4. Two newline separated lists—of new Jack audio input and output port names—are retrieved from a diff of the two snapshots;
5. The appropriate *assignment function* from the application library is called with the two lists as arguments.

Accordingly, new Jack-Midi and Alsa-Midi ports are retrieved and passed to the functions for assignment.

In the case of programs with many ports—like a mixer with inputs, outputs, sends and returns—it is helpful to make up a suitable naming scheme for ports within the application, then using tools like `grep` or `sed` for variable assignment.

3.6.4 h_acn and h_mcn

`h_acn()` and `h_mcn()` call one *connect-function* for each depend of the method and one for each depend of the application.

In those connect-functions, we must establish connections using the port-variables assigned during the `start` step.

Two functions are available to aid in establishing connections:

- `msaudioconnect()` for mono/stereo-agnostic audio connections.
- `ajmidiconnect()` for Alsa/Jack-agnostic Midi connections. Whenever required, Chino will launch `a2jmidid` to facilitate translation.

Both functions require certain suffixes being part of the port-variable names, documented in detail in the comments inside the libraries of the dummy preset that comes with Chino for documentation purposes.

If a depend is unsatisfied—i.e. either not provided or ambiguously provided—no attempt will be made to establish connections.

Connect-functions are not exclusively called during the `acn` and `mcn` steps. The tasks for adding or restarting applications will, after having completed all steps, call connect-functions for all nodes depending on newly provided port-groups. That way the connection graph is kept sane regardless of application launch order.

3.7 User interface

The user interface consists of the configuration file, command line options and arguments, the hand-editable session definition file and a runtime user interface.

To give an examplary session definition file, here is what a file defining the session shown in figure 5 would look like.

```
NAME=graph
PRESET=/path/to/preset:preset_name
UQMETHS=hw msc
uq_hw=stereo nano
uq_msc=ardour
CHMETHS=synth
ch_synth-CH-001=yoshimi
ch_synth-CH-003=ams
```

Via command line, new sessions can be created and existing ones can be opened. For convenience, some more options exist: for writing a prototype session definition file and for creating new libraries by using existing methods or applications as a template.

Whenever running a session, its base directory must be the present working directory. For creating a new session, at least a session name must be given.

```
$ chino -n name_of_session
```

To open an existing session, we point Chino to the session definition file.

```
$ chino -o name_of_session.sdef
```

Once a session is running, the runtime user interface offers keybindings to add, remove or restart applications, to re-establish connections, to localise libraries, to check dependencies, to display the dependency graph and to save the session state. The latter will only save the current setup, state of the involved applications needs to be manually saved to the appropriate application files.

4 Conclusions

4.1 Field of application

Due to said differences, Chino does not so much compete with the other session managers. Being a command line tool that requires some editing of Bash scripts for customisation, it will certainly not fulfil the desire of many users for a comprehensive session manager with a graphical user interface.

Chino just attempts to fill a niche in the ecosystem of session managers by embracing modularity and customisability. One of its strong points is the use of presets and templates, although other session managers also offer features in that direction.

Not being able to store any setup clearly is a disadvantage, though one that is somewhat mitigated once the following assumptions are made:

- Any one user will only use a small subset of all possible setups;
- The user will use that subset repeatedly.

Admittedly, those assumptions do not apply to someone in the phase of exploring the variety of Linux audio applications. For someone who is comfortable with Bash and knows which set of applications to use for what purpose, however, Chino might be a convenient tool.

4.2 Session portability

Sessions turn out to be somewhat portable. Limitations that come to mind are:

1. the session must either be self-contained or its preset must be present on the host system;
2. we might run into incompatibility-issues when versions of programs are mismatching;
3. hardware requirements of the sessions, like audio channel counts, must be met;
4. depending on the programs used, matching sample rates might be required;
5. hardware applications might need to be adapted to Jack/Alsa port names of the host system;
6. program behaviour might differ due to local configuration files.

While points (1) to (4) are mere facts, point (5) can be resolved by agreeing on a naming scheme for port-groups the hardware applications provide for, host systems can then use their own hardware applications.

Point (6) can be mitigated if applications make use of as many command line flags as possible, to override local settings. If the program allows to specify a configuration file on the command line, we can include one as part of the application.

4.3 Known issues

Due to the fairly small user base (consisting of just the author himself), this list is most likely incomplete.

- Establishing connections takes rather long for large sessions.
- It takes time and care to build a preset (though once that is accomplished, Chino doesn't get in the way anymore).
- It's a crude hack still in development.

Given the last point, Chino actually runs surprisingly well.

5 Acknowledgements

Sincere thanks go to all Linux (audio) developers, collectively constituting the giant's shoulders upon which this little script resides.

Thanks go also to the entire Linux audio community. Especially the mailing lists have provided some highly educational reading matter over the years.

Csound6: old code renewed

John FITCH and **Victor LAZZARINI** and **Steven YI**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{jpff@codemist.co.uk, victor.lazzarini@nuim.ie, stevenyi@gmail.com}

Abstract

This paper describes the current status of the development of a new major version of Csound. We begin by introducing the software and its historical significance. We then detail the important aspects of Csound 5 and the motivation for version 6. Following, this we discuss the changes to the software that have already been implemented. The final section explores the expected developments prior to the first release and the planned additions that will be coming on stream in later updates of the system.

Keywords

Music Programming Languages, Sound Synthesis, Audio Signal Processing

1 Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6. This meant that most of the major changes and improvements to the software would cease to be made in Csound 5, and while new versions would be released, these will consist mainly of bug fixes and minor changes (possibly including new opcodes). Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations. The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

This paper describes the motivation for the changes, current state of development and prospective plans for the system.

1.1 Short History of Csound

1.1.1 Early History

Csound has had a long history of development, which can be traced back to Barry Vercoe's MUSIC 360[Vercoe, 1973] package for computer music, which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV[Mathews and Miller, 1964]. Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11[Vercoe, 1981]. Later, with the availability of C (and UNIX), this program was re-written in that language as Csound[Boulanger, 2000], allowing a simpler cycle of development and portability, in comparison to its predecessor.

The system, in its first released version, embodied a largely successful attempt at providing a cross-platform program for sound synthesis and signal processing. Csound was then adopted by a large development community in the mid 90s, after being translated into the ANSI C standard by John ffitch in the early half of the decade. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

1.1.2 Csound 5

The need for the further development of the Csound API, as well as other innovations, prompted a code freeze and a complete overhaul of the system into version 5[ffitch, 2005]. Much of this development included updating 1970s programming practices by applying more modern standards. One of the major aims was to make the code reentrant, so that its use as a library could be made more robust. In 2006, version 5.00 was released. The developments embodied by this and subsequent releases allowed a varied use of the software, with a number of third-party projects benefitting from them.

1.2 Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score. In Csound 5, compilation can only be done once per performance run, so new instruments cannot be added to an already running engine (for this performance

needs to be interrupted so the compilation can take place).

The steps involved in the compiler can be divided into two: parsing, and compilation proper. The first creates an abstract syntax tree (AST) representing the instruments. The compilation then creates data structures in memory that correspond to the AST. When an instrument is instantiated, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data. The list orders instruments by ascending number, so higher-order ones will always be executed last. All of the key aspects of Csound operation are exposed by the API.

2 Motivation

In the six years since its release, Csound 5 continued to develop in many ways, mostly in response to user needs, as well as providing further processing capabilities in the form of new opcodes. After a long gestation, early in 2012, the new flex-bison parser was completed and added as a standard option. This was the final major step of development for Csound, where the last big chunk of 1970s code, the old ad-hoc parser, was replaced by a modern, maintainable, and extendable parser. Following the 2011 Csound Conference in Hannover, it was clear that there were a number of user requests that would be more easily achievable with a rethink of the system. Such suggestions included:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- facilities for live coding

The time was ripe for major changes to be made. User suggestions prompted developers to begin an internal cleanup of code, the removal of older components (such as the old parser), and a reorganisation of the API. It was also an opportunity to code-walk, and with that find inconsistencies and bugs that would normally be hidden. In particular, changes related to repeated loading and compilation of new instruments would require (and indeed force) a welcome separation of language and synthesis engine, which is well underway at present.

3 Developments to date

3.1 Build System and Tests

In Csound 5, the official build system is SCons[1]. Over time, a CMake-based[2] build was introduced and used for local developer use, as well as later for Debian packaging and iOS builds. In Csound 6, the official build system is now the CMake-based build. Moving to CMake introduced some hurdles and changes in workflow, but it also brought with it generation of build system files, such as Makefiles, XCode projects, and Eclipse projects. This solved a problem of IDE-based projects for building Csound becoming out of sync with changes in the SConstruct file for SCons, as well as brought more ways for developers to approach building and working with Csound code, particularly through IDE's.

Using the CTest feature in CMake, unit and functional tests have been added to Csound 6's codebase. CTest is the test running utility used to execute the individual C-code tests. In addition, CUnit[3] is employed to create the individual tests and test-suites within the test code files. In addition to C-code testing, the suite of CSD's used for application/integration testing continues to grow, and a new set of Python tests has also been added for testing API usage from a host language.

3.2 Code reorganisation

The Csound code base is passing through a significant reorganisation. Firstly, parts of it that are now obsolete, such as the old parser, have been removed. Some opcodes with special licensing conditions that have been deemed not to be conducive to further development have been completely rewritten (also with some efficiency and generality improvements). The CSOUND struct has been rationalised and reorganised, with many modifications due to the various changes outlined in the next sections.

[1] http://www.scons.org
[2] http://www.cmake.org
[3] http://cunit.sourceforge.net

Finally, the public API is going through a redesign process (details of which are discussed below).

3.3 Type system

The Csound Orchestra language uses strongly typed variables and enforces these at compile-time. This type information is used to determine the size of memory to allocate for a variable as well as for specifying the in- and out-arg types for opcodes. The system of types used prior to Csound 6 was hard-coded into the parser and compiler. Adding new types would require adding code in many places.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language. The system also helps clarify how types are used during compilation. Another feature is that variable definitions and types were previously discarded after compile-time; in Csound 6, this information is kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

3.4 Generic Arrays

In Csound 5, a 't' type was added that provided a user-definable length, single-dimension array of floating-point numbers. In Csound 6, with the introduction of the generic type system, the code for t-types was extended to allow creation of homogenous, multi-dimensional arrays of any type. Additionally, the argument list specification for opcodes was extended to allow denoting arrays as arguments.

3.5 On-the-fly Compilation

The steps necessary for the replacement or addition of new instruments or UDOs to a running Csound engine, or, more concisely, on-the-fly compilation, started to be taken in the latter versions of Csound 5. It was, of course, sine-qua-non to have a properly structured parser, which we did in 5.17. Also, as a side-effect from the Csound for Android project, compilation from text files was replaced by a new core (memory) file subsystem, so now strings containing Csound code could be presented directly to the parser.

The first step in Csound 6 was made by breaking down the monolithic API call to compile Csound (`csoundCompile()`) into `csoundParseOrc()` and `csoundCompileTree()`, as well as by the addition of a general `csoundStart()` function to get the engine going. The parsing function creates an abstract syntax tree (AST) from a string containing Csound code. The compilation function then creates the internal data structures that the AST represents, ready for engine instantiation(see figure 1).

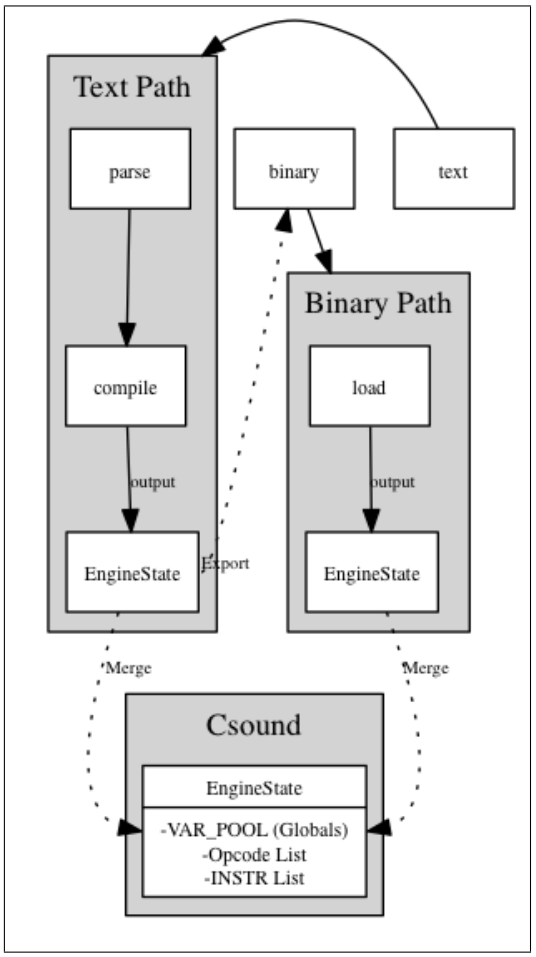


Figure 1: Csound compilation and engineState.

These modifications provided the infrastructure for changes in the code to allow repeated compilation. For this, we have abstracted the data objects relating to instrument definition into an engineState structure. On first compilation, Csound creates its global instrument 0, which is made up of the header statements, global variables declared outside instruments and their initialisation. It then proceeds to compile any other instruments defined in the orchestra (including UDOs, which are a special kind of instrument). On any subsequent compilations, instruments other than 0 are added to a newly-created engineState. After compilation, the new engineState is merged into the current one be-

longing to the running Csound object.

Instrument definitions with the same name or number will replace previously existing ones, but any instances of the old definitions that are active are not touched. New instances will use the new definition, and replaced instruments get added to a deadpool for future memory recovery (which will happen once all old instances are deallocated). A similar process applies to UDOs.

Currently, no built-in thread-safety mechanisms have been placed in the API, so hosts are left to make sure compilation calls are not made concurrently to audio processing calls. However, it is envisaged that the final API will provide functions with built-in thread safe as well as ordinary calls.

3.6 Sample-level accuracy

Csound has always allowed sample-level accuracy, a feature present since its MUSIC 11 incarnation. However, a performance penalty was incurred, since the requirement for this was to set the size of the processing block (`ksmps`) to 1 sample. Code can become very inefficient, since there is a single call of an opcode performance function for each sample of output and this is in conflict with caching.

In Csound 6, an alternative sample accuracy method has been introduced. This involves setting an offset into the processing block, which will round the start time of an event to a single sample. Similarly, event durations are also made to be sample accurate, as the last iteration of each processing loop is limited to the correct number of samples (see figure 2). This option is provided with the non-default `--sample-accurate` flag, to preserve backward compatibility.

Tied events[4] are not subject to sample accurate processing as they involve state reuse and are, in its current form, incompatible with the mechanism. Real-time events are also not affect by the process, as event sensing works on a ksmps-to-ksmps basis. Events scheduled to at least one control-cycle ahead can be made to be sample accurate through this mechanism.

The changes needed for this mechanism to work were significant. Each opcode had to be modified to take account of the offset and end position. The scheduler had to be altered so the start of all events was truncated, instead of rounded, to ksmps boundaries, and the calculation of event duration had to be modified. The offset and end position had to be properly defined for each event, as well as set and reset at specific times for each instrument instance.

[4]In Csound, it is possible to have instrument instances that take up a previously-used memory space, which allows the 'tieing' of events, in analogy to slurs in instrumental music

3.7 Realtime priority mode

Csound has been a realtime audio synthesis engine since 1990. However, it was never provided with strict realtime-safe behaviour, even though in practice, it has been used successfully in many realtime applications. Given the multiple applications of Csound, it makes sense to provide separate operation modes for its engine. In Csound 6, we introduce the realtime priority mode, set by the `--realtime` option, which aims to provide better support for realtime safety, with complete asynchronous file access and a separate thread for unit generator initialisation.

3.7.1 Asynchronous file access

For Csound 6, a new lock-free mechanism has been introduced and some key opcodes have been modified to use it when operating in realtime. It uses a circular buffer, employing an interface which had been already present in Csound (used previously only for lock-free realtime audio). It shares the common file IO structure adopted throughout Csound, with a similar, but dedicated interface. For specific file reading/writing requirements, though, as required for instance by diskin, diskin2 or pvsfwrite, the general interface is not suitable. For this case, special opcode-level asynchronous code has been designed.

3.7.2 Unit generator initialisation

Another important modification of the engine in realtime priority mode is the spawning of a separate thread that is responsible for running all of the unit generator initialisation code. This is more commonly known as the 'init-pass', which is separate from synthesis performance ('perf-pass'). In this mode, when an instrument is instantiated, the init-pass code is immediately run in a separate thread. Once this is done, an instrument is allowed to perform. What this does is to prevent any interruption in the synthesis performance due to non-realtime-safe operations in the initialisation code (memory allocation, file opening, etc.). A side-effect of this is that in some situations, an instrument may be prevented to start performing straight away, as

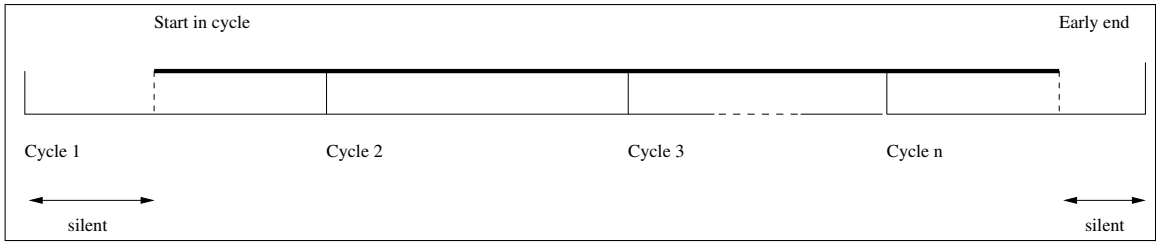


Figure 2: Sample accurate scheme.

the initialisation has not been done. However, this is balanced with the gains in uninterrupted performance.

3.8 Multicore operation

In 2009 an experimental system for using multiple cores for parallel rendering of instruments was written [Wilson, 2009], and this was later incorporated in the standard Csound [ffitch, 2009]. While the design was generally semantically correct it only delivered a performance gains in the case of low control rate and computationally heavy unit generators. Profiling the code showed that the overheads in creating and consuming the directed acyclic graph (DAG) of dependencies, and especially in memory allocation activity.

For Csound 6 we are developing a different approach, that while maintaining the semantic analysis only needs to rebuild the DAG when a score event starts or stops, and in use does not call for changes in the structure. The clue is in the use of watch-lists as found in SAT-solvers [Brown and Purdom Jr, 1982; **?**]. For each task we only need to watch for the completion of one of the dependencies; when a task finishes it can release any task that is waiting for it, and for which all other precursors have already finished. This strategy is also possible with no locking of critical sections, and can use atomic swap primitives instead.

At the same time some simplification of the semantics-gathering has been achieved. This scheme preserves the order-semantics that Csound has always had, but offers efficient utilisation of multiple cores with threads without user intervention beyond saying how many threads to use for the performance stage. Initial measurements (see table 3.8) are very encouraging, in most cases providing significant speed-up. We are continuing to work on possible optimisations.

4 Further work

4.1 Pre-release prospective development (i.e. the "todo list")

The final feature set of Csound 6 is still not finalised. There are a number of possible enhancements that we are considering; some grow from the changes we have described above, and some are long-standing desires.

The introduction of separate compilation and replaceable instruments naturally suggests that we could add a fast loadable format for instruments, building on for example LISP FASL formats, and API and opcode access to loading. It remains to be seen if the source version is sufficiently fast, and whether we can solve the semantic issues that arise, such as f-table independence. What is needed is to document the abstract syntax tree that the parser produces, and thus allow advocates of alternative orchestra languages to provide them.

A restriction in Csound than has long been an irritation is the limit of one string in a score statement. Previous work in this area has attempted to allow up to four strings, but this is both limiting and still buggy. The radical solution would be to introduce a flex/bison parser for the score language and take the opportunity for rethinking the score area. A small start has been made, but the need to support users and the amount of effort needed here has relegated this work to a later release. Until then a simpler scheme will have to be tried for the interim.

The Csound suite of software include a number of analysis programs, most dating from an early time, and written without regard of floating point formats or byte order. From time to time this has caused problems. The task here is to redefine these formats to indicate at least their formats, or even to make the readers capable of format transformations. This needs to be done at some stage and this break seems like a good moment.

With the introduction of on-the-fly compilation one can consider that a user might main-

-j	CloudStrata	Xanadu		Trapped...		
	ksmps=500 (sr=96000)	ksmps=10	ksmps=100	ksmps=10	ksmps=100	ksmps=1000
1	1	1	1	1	1	1
2	0.54	0.57	0.55	0.75	0.79	0.78
3	0.39	0.40	0.40	0.66	0.76	0.73
4	0.32	0.39	0.33	0.61	0.72	0.70

Table 1: *Relative performance with multiple threads in three existing Csound code examples, -j indicates the number of threads used.*

tain a long-running Csound binary and use it for different tasks at different times. This suggests that the current command-line options or API equivalents may need to change at some time after the initialisation. Some changes may be easy, but some may require re-engineering of parts of the engine. We have not yet realised to use-changes that the compilation change will engender.

The new API still needs to be refined. In response to what has been discussed above, we plan, for instance, to expose the configuration parameters in some form (currently held in the OPARMS data structure). At the moment, there is a simple provision for setting separately specific configuration items in the API (as flags). This is to be substituted by a more flexible form, via the exposing of the OPARMS or an OPARMS-like struct to API users.

A number of other changes are planned, some of which are already present in an early form. For instance, the various stages of parsing, compilation, and engine start are now exposed in the provisional API (as detailed for instance in 3.4). There is a plan to provide built-in thread-safety, so some functions can be used directly in a multi-threading environment without further synchronisation or resource protection. The software bus, which now exists in three forms, will be unified to a single mechanism.

4.2 Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation)
- further language features (e.g. namespaces, functions with more than one argument, tuples)
- a system for streaming linear predictive coding processing (in similar fashion to PVOC)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

5 Conclusions

In this paper, we have sought to examine the current development status of Csound 6, as well as the motivations for the fundamental re-engineering of the code that has been underway. We hope to have demonstrated how the technology embodied in this software package has been renovated continuously in response to developments in Computer Science and Music. Our aim is to continue to support a variety of styles of computer music composition and performance, as well as the various ways in which Csound can be used for application development. It is also important to note, for readers, that the re-engineering of Csound is taking place quite publicly in the Csound 6 git repository on Sourceforge (`git://git.code.sf.net/p/csound/csound6-git`). Anyone is welcome to check out and examine our struggles with computer technology and the solutions we are putting forward in this paper.

6 Acknowledgements

Our thanks go to the Csound community for their indulgence, suggestions and support. In addition Martin Brain introduced the idea of watch-lists and co-developed the detailed performance algorithm. We also acknowledge the implicit support from Sourceforge hosting

References

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design.* MIT Press, February.

Cynthia A. Brown and Paul Walton Purdom Jr. 1982. An Empirical Comparison of Backtracking Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(3).

John ffitch. 2005. The Design of Csound5. In *LAC2005*, pages 37–41, Karlsruhe, Germany, April. Zentrum für Kunst und Medientechnologie.

John ffitch. 2009. Parallel Execution of Csound. In *Proceedings of ICMC 2009*, Montreal. ICMA.

M. Mathews and J. E. Miller. 1964. *MUSIC IV Programmer's Manual*. Bell Telephone Labs.

B. Vercoe. 1973. *Reference manual for the MUSIC 360 language for digital sound synthesis*. Studio for Experimental Music, MIT.

B. Vercoe. 1981. *MUSIC 11 Reference Manual*. Studio for Experimental Music, MIT.

Christopher Wilson. 2009. Csound Parallelism. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath.

Linux AVB Stack Workshop for LAC2013, IEM Graz

Christoph Kuhr
Freelance Audio Engineer and Digital Audio Developer
Bonnstr 430
50321 Bruehl,
Germany,
christoph.kuhr@web.de

Abstract

This workshop and discussion panel is intended to get a working concept for an IEEE 802.1AVB (Audio Video Bridging) Linux stack. AVB is an audio and video (AV) network streaming infrastructure, designed to provide realtime transmission and admission control of AV streams on the DLL/MAC layer (OSI layer 2). To enable Linux computers to stream AV using an AVB infrastructure, an Linux AVB stack is required.

Keywords

AV networking, ALSA driver, realtime communication

1 Introduction

Audio Video Bridging (IEEE 802.1AVB) is a promising and evolving industry standard using Ethernet networks.
It would be desireable to record or playback AVB Streams with a DAW, as well as streaming Ambisonic signals to many AVB capable loudspeakers. These are not the only possible scenarios for connecting a Linux computer to an AVB network, but those are of broad concern in the Linux Audio community.
Until now, there is no open source attempt for an Linux AVB stack. This document lines out a concept for the development of such a stack.

2 AVB - The Standards

2.1 General Precision Timing Protocol (gPTP)

The standard IEEE 802.1AS [1] defines a more restricted profile of the IEEE 1588-2008 standard and describes the mechanisms used to maintain a domain with consistent timing and synchronization.
To achieve consistent timing and synchronization, the clock is organized in a hierarchical tree structure. This tree sturucture has a single root element, the GrandMasterClock. The root element is connected to the port of an AVB capable switch. This port is a clock slave port and triggers the clocks of the other switch ports. Any other switch port, triggerd by a clock slave port, is then called clock master port. The clock master ports again triggers either the connected end station or the next AVB switch.
Any AVB participant has to implement the Best Masterclock Algorithm (BMCA) and announce its' own clock. The BMCA is used to determine, whether the announced clock has better properties then the participant's own clock. If so, the participant switches its' clock role to slave and listens to the better remote clock.

2.2 Multiple Stream Registration Protocol (MSRP [2])

In order to guarantee a glitch-free transmission first the network resources for the Stream have to be reserved throughout the entire path between AVB talker and listener. The responsibility for the resource reservation is up to both sides, the talker and the listener. The talker tries to announce the stream's requirements and the listener acknowledges them. This process is done in any AVB switch in the transmission path.
The actual process of announcing and acknowledging is described in chapter 10 of IEEE 802.1ak [3]

2.3 Forwarding and Queueing of Time-sensitive Streams (FQTSS)

The second factor in guaranteeing a glitch-free transmission is forwarding and queueing in any AVB participant.
FQTSS [4] is defined as a superset of the Leaky-Bucket algorithm, extended by a credit

based queueing upon the StreamId of any incoming stream. Without a StreamId, traffic will be handeled as non-AVB and therefore has a lower priority, hence less credits. However, any frame is forwarded before it is timed out.

2.4 Transport Protocol for Time-Sensitve Streams (AVTP)

The standard IEEE 1722 describes a pair of layer two protocols, first the Audio Video Transport Protocol (AVTP [5, ch 5]) is a transport format for transmitting IEC 61883-6 (AMDTP [6]) AV data and secondly the MAC Address Acquisition Protocol (MAAP [5, Annex B]), which reserves the multicast addresses for the stream.
IEEE 1722 frames contain amongst other information the unique 64-Bit StreamID as well as the 32-Bit presentation time. The StreamID is used by MSRP and FQTSS, while the presentation time is only used by FQTSS, to determine the latest point in time to transmit the frame.

2.5 Audio/Video Device Discovery, Enumeration, Connection Management & Control Protocol for AVTP devices (AVDECC)

The standard IEEE 1722.1 [7] describes a set of layer two protocols. Any AVB end station has to implement an AVDECC Entity Model (AEM [7, ch 7]), which contains information about the configuration.
This information is used by the three protocols AVDECC Discovery Protocol (ADP [7, ch 6]), AVDECC Connection Management Protocol (ACMP [7, ch 8]) and AVDECC Enumeration and Control Protocol (AECP [7, ch 9]). The protocol ADP implements routines to announce the availability of an end station to an AVB domain. ACMP manages the connection of certain streams by using MSRP. AVB end points implementing a AVDECC controller can be controlled with the AECP protocol.

3 Requirements

3.1 Hardware Requirements

To enable AVB on a Linux computer, it has to provide firstly an AVB capable NIC, including a gPTP timestamping and a hardware 802.1Qav queueing.
Secondly, it has to provide a soundcard with the possibility of fine tuning the sampling rate, like the RME HDSP 9652 or the M-Audio Delta1010 / Audiophile96 soundcards.

3.2 Software Requirements

The first thing required is a NIC kernel driver, which implements the SO_TIMESTAMPING API and is supported by the new PHC interface.

The interface to the AVB stack simply is an IEEE 1722.1 AVDECC Controller, which provides the possibility to control any aspect of the AVB stack.
The connection management is done by the MRP kernel module.
To maintain the newly introduced MAC multicast address spaces, a 1722 MAAP implementation is required.

To provide the presentation time to the AVB talkers, it has to be derived from a gPTP server. The gPTP server also participates in the BMCA, synchronizes with the gPTP Domain and fine tunes the sampling rate of the soundcard.
AVB listeners need to play audio samples at the correct time. Since all AVB talkers on the network have their own sampling clock and only one sampling clock can be provided on the local computer without multiple PLL hardware instances, sample rate conversion has to be used to align the AVB listeners audio samples to the local sampling clock.

The audio samples need to be de- / encapsulated in AMDTP packets, into AVTP packets, into 1722 Frames and vice versa.

4 Brainstorming

4.1 Linux AVB Stack

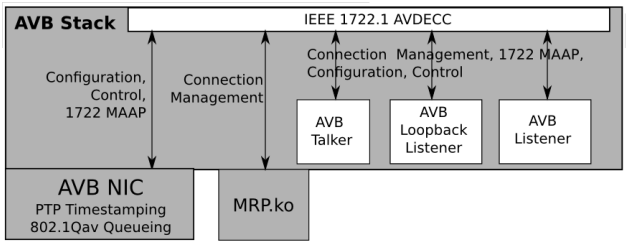


Figure 1: Connection Management, Configuration and Control

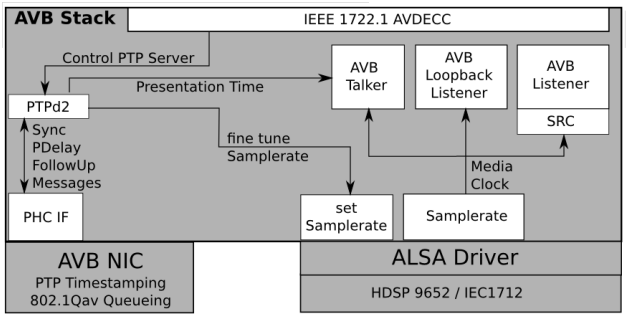


Figure 2: Timing and Synchronization

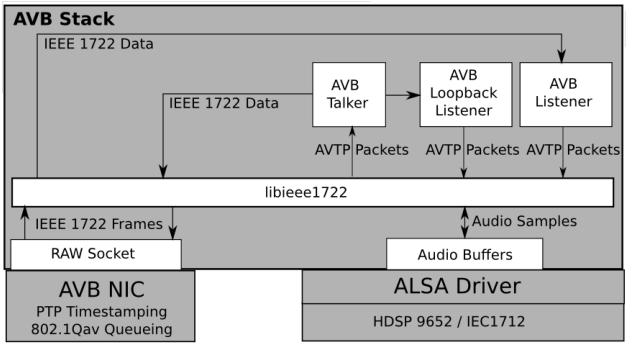


Figure 3: Audio and Network Transport

And. . . ?

4.2 Work already Done

4.2.1 PTP Hardware Clock (PHC)

With the kernel 2.6.30, a new timestamping API was introduced. The PHC requires the underlying network driver to use the SO_TIMESTAMPING API and triggers the POSIX Clock API. Thus the whole operating system is synced to the same clock.
For a list of yet supported hardware, have a look at section four of the following link:
http://lwn.net/Articles/406978/

Further reading at:
http://lwn.net/Articles/315941/
http://sourceforge.net/p/ptpd/patches/30/
http://kerneltrap.org/mailarchive/linux-netdev/2010/4/27/6275689/thread

4.2.2 PTPd2 - PTP daemon V2 by Alan Bartky

PTPd2 is a software implementation of the end station procedures of gPTP. It uses the POSIX API to run the synchronization routines and the BMCA.

http://code.google.com/p/ptpv2d/

4.2.3 MSRP - Multiple Stream Registration Protocol by Philip Foulkes

A running Linux kernel module MRP.ko is available.

http://code.google.com/p/kmrp/
http://code.google.com/p/kmsrp/
http://code.google.com/p/kmmrp/
http://code.google.com/p/kmvrp/

4.2.4 AVDECC-pdu - AVDECC by Jeff Koftinoff (MeyerSound)

AVDECC-pdu is a C library maintained by Jeff Koftinoff and is an implementation of the AVDECC standard for end stations.

https://github.com/jdkoftinoff/avdecc-pdu

4.2.5 HDSP9652, IEC1712/1724 Kernel Modules

Since the patch alsa-driver-1.0.14rc3, the HDSP driver has the possibility to fine tune the sample rate.

http://www.spinics.net/lists/alsa-devel/msg06237.html

4.2.6 libraw1394 and libiec61883

Modifications to libraw1394/libiec61883 (Firewire AMDTP library) or similar IEC 61883-6 libraries

git://git.user.in-berlin.de/s5r6/libraw1394.git
git://git.user.in-berlin.de/s5r6/libiec61883.git

4.3 Work to be Done

4.3.1 AVB Stack

The AVB stack has to be incorporated, so that it can be used by any ALSA application. All components mentioned above have to be modifed and interconnected.
This concept lacks the possibility for deriving multiple AVB listener instances' media clocks from a PLL, due to major difficulties in generating a software PLL with minimum jitter. A compromize is to do a sample rate conversion.

This feature would be very useful for using Jack with AVB.

And. . . ?

4.3.2 Work Group

Are there people, who are interested in forming a working group for the development of an AVB stack?

I have opened a Git Repository under the address:
git://github.com/voodoosound/linux_avb_stack.git

5 Conclusions

The system components that are required to implement an AVB stack are mostly available. A library to provide any IEEE 1722 functionality has to be developed.

At the moment not many AVB capable NICs and PHC supporting driver modules are available. Also a little problematic, is the use of the MRP kernel module, since it is not a mainline module and therefor has to be installed in addition. Any other component runs in user space.

However these conclusions, as well as the Requirements and Brainstorm section are open for discussion.
The results of the discussion, along with this paper will be available in the Git repository.

6 Acknowledgements

Many thanks go to the carneval season for writing time on the road. Further, my tomcat for decorating this paper with typos and my wife for correcting them.

References

[1] IEEE, *IEEE Std P802.1AS/D7.6 - Draft Standard for Local and Metropolitan Area Networks Timing and Synchronization for Time- Sensitive Applications in Bridged Local Area Networks*, 11/11/2010.

[2] IEEE, *IEEE Std P802.1Qat/D6.1 - Draft Standard for Local and Metropolitan Area Networks Virtual Bridged Local Area Networks - Amendment 34: Stream Reservation Protocol (SRP)*, 09/04/2010.

[3] IEEE, *IEEE Std 802.1ak-2007 IEEE Standard for Local and metropolitan area networks Virtual Bridged Local Area Networks Amendment 7: Multiple Registration Protocol*, 06/22/2007.

[4] IEEE, *IEEE Std P802.1Qav-2009 - IEEE Standard for Local and metropolitan area networks Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, 01/05/2010 .

[5] IEEE, *IEEE P1722/D2.5 - Draft Standard for Layer 2 Transport Protocol for Time Sensitive Applications in a Bridged Local Area Network*, 09/2010.

[6] 1394 Trade Association, *Audio and Music Data Transmission Protocol 2.0*, 08/21/2001.

[7] IEEE, *IEEE P1722.1/D20 - Draft Standard for Device Discovery, Connection Management and Control Protocol for IEEE 1722 Based Devices*, 06/2012.

Live music programming in Haskell

Henning Thielemann
Pfännerhöhe 42,
06110 Halle,
Germany,
lac@henning-thielemann.de

Abstract

We aim for composing algorithmic music in an interactive way with multiple participants. To this end we have developed an interpreter for a sub-language of the non-strict functional programming language Haskell that allows the modification of a program during its execution. Our system can be used both for musical live-coding and for demonstration and education of functional programming.

Keywords

Live coding, MIDI, Functional programming, Haskell

1 Introduction

It is our goal to compose music by algorithms. We do not want to represent music as a sequence of somehow unrelated notes as it is done on a note sheet. Instead we want to describe musical structure. For example, we do not want to explicitly list the notes of an accompaniment but instead we want to express the accompaniment by a general pattern and a sequence of harmonies. A composer who wants to draw a sequence of arbitrary notes might serve as another example. The composer does not want to generate the random melody note by note but instead he wants to express the idea of randomness. Following such a general phrase like "randomness" the interpreter would be free to play a different but still random sequence of notes.

The programmer shall be free to choose the degree of structuring. For instance, it should be possible to compose a melody manually, accompany it using a note pattern following a sequence of user defined harmonies and complete it with a fully automatic rhythm.

With a lot of abstraction from the actual music it becomes more difficult to predict the effect of the programming on the musical result. If you are composing music that is not strictly structured by bars and voices then it becomes more difficult to listen to a certain time interval or a selection of voices for testing purposes. Also, the classical edit-compile-run loop hinders creative experiments. Even if the music program can be compiled and restarted quickly, you must terminate the running program and thus the playing music and you must start the music from the beginning. Especially if you play together with other musicians this is unacceptable.

In our approach to music programming we use a purely functional *non-strict*[1] programming language [Hughes, 1989], that is almost a subset of Haskell 98 [Peyton Jones and others, 1998]. Our contributions to live music coding are concepts and a running system offering the following:

- algorithmic music composition where the program can be altered while the music is playing (Section 2.1),
- simultaneous contributions of multiple programmers to one song led by a conductor (Section 2.2).

2 Functional live programming

2.1 Live coding

We want to generate music as a list of MIDI events [MMA, 1996], that is events like "key pressed", "key released", "switched instrument", "knob turned" and wait instructions. A tone with pitch C-5, a duration of 100 milliseconds and an average force shall be written as:

```
main =
   [ Event (On c5 normalVelocity)
   , Wait 100
   , Event (Off c5 normalVelocity)
   ] ;

c5 = 60 ;
normalVelocity = 64 ;
.
```

[1] All terms set in italics are explained in the glossary on page 7. In the PDF they are also hyperlinks.

Using the list concatenation "++" we can already express a simple melody.

```
main =
   note qn c ++ note qn d ++
   note qn e ++ note qn f ++
   note hn g ++ note hn g ;

note duration pitch =
   [ Event (On pitch normalVelocity)
   , Wait duration
   , Event (Off pitch normalVelocity)
   ] ;

qn = 200 ;  -- quarter note
hn = 2*qn ; -- half note

c = 60 ;
d = 62 ;
e = 64 ;
f = 65 ;
g = 67 ;
normalVelocity = 64 ;
```

We can repeat this melody infinitely by starting it again when we reach the end of the melody.

```
main =
   note qn c ++ note qn d ++
   note qn e ++ note qn f ++
   note hn g ++ note hn g ++ main ;
```

Please note, that this is not a plain recursion, but a so called *co-recursion*. If we define the list `main` this way it is infinitely long but if we expand function applications only when necessary then we can evaluate it element by element. Thanks to this evaluation strategy (in a sense *lazy evaluation* without *sharing*) we can describe music as pure list of events. The music program does not need, and currently cannot, call any statements for interaction with the real world. Only the interpreter sends MIDI messages to other devices.

In a traditional interactive interpreter like the `GHCi`[2] we would certainly play the music this way:

```
Prelude> playMidi main       .
```

If we would like to modify the melody we would have to terminate it and restart the modified melody. In contrast to this we want to alter the melody while the original melody remains playing and we want to smoothly lead over from the old melody to the new one. In other words: The current state of the interpreter consists of the program and the state of the interpretation. We want to switch the program, but we want to keep the state of interpretation. This means that the interpreter state must be stored in a way such that it stays sensible even after a program switch.

We solve this problem as follows: The interpreter treats the program as a set of term rewriting rules, and executing a program means to apply rewrite rules repeatedly until the start term `main` is expanded far enough that the root of the operator tree is a terminal symbol (here a *constructor*). For the musical application the interpreter additionally tests whether the root operator is a list constructor, and if it is the constructor for the non-empty list then it completely expands the leading element and checks whether it is a MIDI event. The partially expanded term forms the state of the interpreter. For instance, while the next to last note of the loop from above is playing, that is, after the interpreter has sent its `NoteOn` event, the current interpreter state would look like:

```
Wait 200 :
   (Event (Off g normalVelocity) :
      (note hn g ++ main))
.
```

The interpreter will rewrite the current expression as little as possible, such that the next MIDI event can be determined. On the one hand this allows us to process a formally infinite list like `main`, and on the other hand you can still observe the structure of the remaining song. E.g. the final call to `main` is still part of the current term. If we now change the definition of `main` then the modified definition will be used when `main` is expanded next time. This way we can alter the melody within the loop, for instance to:

```
main =
   note qn c ++ note qn d ++
   note qn e ++ note qn f ++
   note qn g ++ note qn e ++
   note hn g ++ main ;
.
```

But we can also modify it to

```
main =
   note qn c ++ note qn d ++
   note qn e ++ note qn f ++
   note hn g ++ note hn g ++ loopA ;
```

[2] Glasgow Haskell Compiler in interactive mode

in order to continue the melody with another one called `loopA` after another repetition of the `main` loop.

We want to summarise that the meaning of an expression can change during the execution of a program. That is, we give up a fundamental feature of functional programming, namely *referential transparency*.

We could implement the original loop using the standard list function `cycle`

```
main =
  cycle
    ( note qn c ++ note qn d ++
      note qn e ++ note qn f ++
      note hn g ++ note hn g ) ;
```

and if `cycle` is defined by

```
cycle xs = xs ++ cycle xs ;
```

then this would be eventually expanded to

```
( note qn c ++ note qn d ++
  note qn e ++ note qn f ++
  note hn g ++ note hn g )
++
cycle
   ( note qn c ++ note qn d ++
     note qn e ++ note qn f ++
     note hn g ++ note hn g ) ;
.
```

Using this definition we could leave the loop only by changing the definition of `cycle`. But such a change would affect *all* calls of `cycle` in the current term. Further, in a rigorous module system without import cycles it would be impossible to access functions of the main module from within the standard module `List` that defines the `cycle` function. But this would be necessary in order to not only leave the `cycle` loop but to continue the program in the main module.

From this example we learn that a manually programmed loop in the form of `main = ... ++ main` has advantages over a loop function from the standard library, because the manual loop provides a position where we can insert new code later.

Additionally to the serial composition of musical events we need the parallel composition for the simultaneous playback of melodies, rhythms and so on. At the level of MIDI commands this means that the commands of two lists must be interleaved in the proper way. For details we refer the reader to the implementation of "`=:=`".

2.1.1 User interface

The graphical user interface is displayed in Figure 1. In the upper left part the user enters the program code. Using a keyboard short-cut he can check the program code and transfer it to the buffer of the interpreter. The executed program is shown in the upper right part. In this part the interpreter highlights the function calls that had to be expanded in order to rewrite the previous interpreter term into the current one. This allows the user to trace the melody visually. The current term of the interpreter is presented in the lower part of the window. The texts in the figure are essentially the ones from our introductory example.

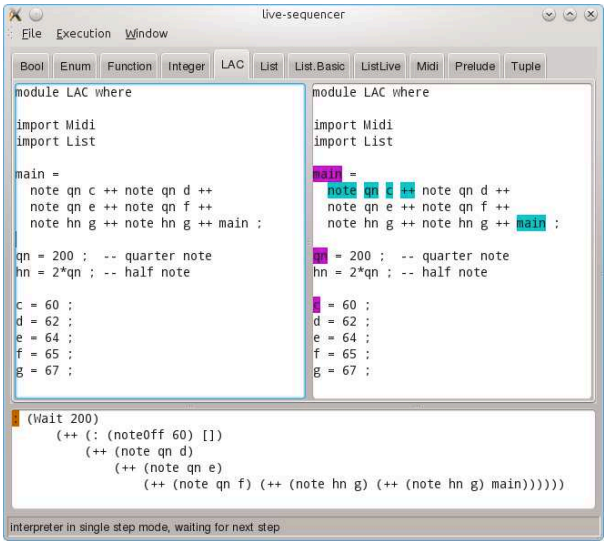


Figure 1: *The running interpreter*

Our system can be run in three modes: "real time", "slow motion" and "single step". The real-time mode plays the music as required by the note durations. In contrast to that the other two modes ignore the wait instructions and insert a pause after every element of the MIDI event list. These two modes are intended for studies and debugging. You may also use them in education if you want to explain how an interpreter of a non-strict functional language works in principle.

We implemented the interpreter in Haskell using the Glasgow Haskell Compiler GHC [Peyton Jones and others, 2012], and we employ WxWidgets [Smart et al., 2011] for the graphical user interface. Our interpreted language supports pattern matching, a set of predefined infix operators, higher order functions, and partial function application. For the sake of a sim-

ple implementation we deviate from Haskell 98 in various respects: Our language is dynamically and weakly typed: It knows "integer", "text" and "*constructor*". The parser does not pay attention to layout thus you have to terminate every declaration with a semicolon. Several other syntactic features of Haskell 98 are neglected, including list comprehensions, operator sections, do notation, "let" and "case" notation, and custom infix operators. I/O operations are not supported as well.

2.2 Distributed coding

Our system should allow the audience to contribute to a performance or the students to contribute to a lecture by editing program code. The typical setup is that the speaker projects the graphical interface of the sequencer at the wall, the audience can listen to music through loud speakers, and the participants can access the computer of the performer via their browsers and wireless network.

Our functional language provides a simple module system. This helps the performer to divide a song into sections or tracks and to put every part in a dedicated module. Then he can assign a module to each participant. This is still not a function of the program, but must be negotiated through other means. For instance the conductor might point to people in the audience. Additionally the performer can insert a marker comment that starts the range of text that participants can edit. The leading non-editable region will usually contain the module name, the list of exported identifiers, the list of import statements, and basic definitions. This way the performer can enforce an interface for every module.

A participant can load a module into his web browser. The participant sees an HTML page showing the non-editable header part as plain text and the editable region as an editable text field. (cf. Figure 2) After editing the lower part of module he can submit the modified content to the server. The server replaces the text below the marker comment with the submitted text. Subsequently the new module content is checked syntactically and on success it is loaded into the interpreter buffer. In case of syntax errors in the new code the submitted code remains in the editor field. The performer can inspect it there and can make suggestions for fixes.

Generally it will not be possible to start composition with many people from scratch. However, the performer can prepare a session by defining a set of modules and filling them with basic definitions. For instance he can provide a function that converts a list of zeros and ones into a rhythm, or a list of integers into a chord pattern or a bass line. By providing a meter and a sequence of harmonies he can assert that the parts contributed by the participants fit together loosely. In this application the performer no longer plays the role of the composer but the role of a conductor.

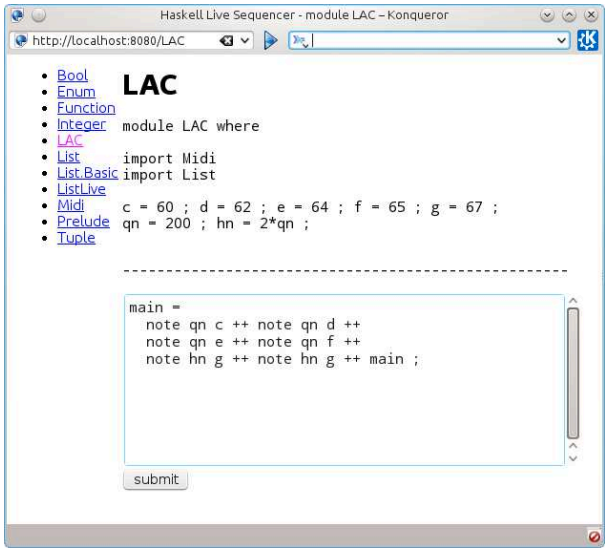


Figure 2: *Accessing a module via HTTP*

2.3 Timing

For a good listening experience we need precise timing for sending MIDI messages. A naive approach would be to send the messages as we compute them. I.e. in every step we would determine the next element in the list of MIDI events. If it is a wait instruction then we would wait for the desired duration and if it is a MIDI event then we would send it immediately. However this leads to audible inaccuracies due to processor load caused by term rewriting, garbage collection and GUI updates.

We are using the ALSA sequencer interface for sending MIDI messages. It allows us to send a MIDI message with a precise but future time stamp. However we still want that the music immediately starts if we start the interpreter and that the music immediately stops if we stop it and that we can also continue a paused song immediately. We achieve all these constraints the following way: We define a latency, say d milliseconds. The interpreter will always com-

pute as many events in advance until the computed time stamps are d milliseconds ahead of current time. This means that the interpreter will compute a lot without waiting when it is started. It will not immediately send a MIDI message because it needs to compute it first. This introduces a delay, sometimes audible, but we cannot do it faster. When the user pauses the interpreter, we halt the timer of our outgoing ALSA queue. This means that the delivery of messages is immediately stopped, but there are still messages for the next d milliseconds in the queue. If the interpreter is continued these messages will be send at their scheduled time stamps. If the interpreter is stopped we simply increase the time of the ALSA queue by d milliseconds in order to force ALSA to send all remaining messages.

3 Related work

Algorithmic composition has a long tradition. The musical dice games by Mozart and the Illiac Suite [Hiller and Isaacson, 1959] might serve as two popular examples here. Further on, the Haskore project (now Euterpea) [Hudak et al., 1996] provides a method for music programming in Haskell. It also lets you control synthesisers via MIDI and it supports the generation of audio files via CSound, SuperCollider or pure Haskell audio signal synthesis. Like our approach, Haskore relies upon lazy evaluation which allows for an elegant definition of formally big or even infinite songs while its interpretation actually consumes only a small amount of memory. However, the creative composition process is made more difficult by the fact that you can listen to a change to a song only after terminating the old song and starting the new one. In a sense, our system is an interactive variation of Haskore.

So-called functional reactive programming is a very popular approach for programming of animations, robot controls, graphical user interfaces and MIDI processing in Haskell [Elliott and Hudak, 1997]. Functional reactive programming mimics working with a time ordered infinite list of events. But working with actual lazy lists leads to fundamental problems in real-time processing, e.g. if a stream of events is divided first but merged again later. This is a problem that is solved by functional reactive programming libraries. The advantage of functional reactive MIDI processing compared to our approach is that it allows the processing of event input in realtime. The disadvantage is that usually you cannot alter a functional reactive program during execution.

Erlang is another functional (but not purely functional) programming language that accepts changes to a program while the program is running [Armstrong, 1997]. Erlang applies eager evaluation. That is, in Erlang you could not describe a sequence of MIDI commands by a lazy list of constructors. Instead you would need iterators or similar tools. You can insert new program code into a running Erlang programming in two ways: Either the running program runs functions (e.g. lambda expressions) that it receives via messages or you replace an Erlang module by a new one. If you upload a new Erlang module then the old version is kept in the interpreter in order to continue the running program. Only calls from outside the module jump into the code of the new module, but by qualification you can also simulate an external call from within the replaced module. That is, like in our approach, you need dedicated points (external calls or calls of functions received via messages) where you can later insert new code.

Summarised, our approach for changing running programs is very similar to "Hot Code loading" in Erlang. However, the non-strict evaluation of our interpreter implies that considerable parts of the program are contained in the current term. These are not affected immediately by a change to the program. This way we do not need to hold two versions of a module in memory for a smooth transition from old to new program code. In a sense, Erlang's external calls play the role of our top-level functions.

Musical live coding, i.e. the programming of a music generating program, while the music is playing, was in the beginning restricted to special purpose languages like SuperCollider/SCLang [McCartney, 1996] and ChucK [Wang and Cook, 2004] and their implementations. With respect to program control these languages adhere to the imperative programming paradigm and with respect to the type system they are object oriented languages. The main idea in these languages for creating musical patterns is constructing and altering objects at runtime, where the objects are responsible for sending commands to a server for music generation.

Also in our approach the sound generation runs parallelly to the interpreter and it is controlled by (MIDI) commands. However, in our

approach we do not program how to change some runtime objects but instead we modify the program directly.

In the meantime also Haskell libraries for live coding are available, like Tidal ([McLean and Wiggins, 2010]) and Conductive ([Bell, 2011]). They achieve interactivity by running commands from the interactive Haskell interpreter GHCi. They are similar to SCLang and ChucK in the sense that they maintain and manipulate (Haskell) objects at runtime, that in turn control SuperCollider or other software processors.

4 Conclusions and future work

Our presented technique demonstrates a new method for musical live coding. Maybe it can also be transferred to the maintenance of other long-running functional programs. However, we have shown that the user of the live-sequencer must prepare certain points for later code insertion. Additionally our system must be reluctant with automatic optimisations of programs since an optimisation could remove such an insertion point. If you modify a running program then functions are no longer *referentially transparent*; that is, we give up a fundamental feature of functional programming.

Type system A static type checker would considerably reduce the danger that a running program must be aborted due to an ill-typed or inconsistent change to the program. The type checker would not only have to test whether the complete program is type correct after a module update. Additionally it has to test whether the current interpreter term is still type correct with respect to the modified program.

A type checker is even more important for distributed composition. The conductor of a multi-user programming session could declare type signatures in the non-editable part of a module and let the participants implement the corresponding functions. The type checker would assert that participants could only send modifications that fit the rest of the song.

Evaluation strategy Currently our interpreter is very simple. The state of the interpreter is a term that is a pure tree. This representation does not allow for *sharing*. E.g. if `f` is defined by `f x = x:x:[]` then the call `f (2+3)` will be expanded to `(2+3) : (2+3) : []`. However, when the first list element is evaluated to `5`, the second element will not be evaluated. I.e. we obtain `5 : (2+3) : []` and not `5 : 5 : []`. Since the term is a tree and not a general graph we do not need a custom garbage collector. Instead we can rely upon the garbage collector of the GHC runtime system that runs our interpreter. If a sub-term is no longer needed it will be removed from the operator tree and sooner or later it will be detected and de-allocated by the GHC garbage collector.

Even a simple co-recursive definition like that of the sequence of Fibonacci numbers

```
main = fix fibs
fibs x = 0 : 1 : zipWith (+) x (tail x)
fix f = f (fix f)
```

leads to an unbounded growth of term size with our evaluation strategy. In the future we want to add more strategies like the graph reduction using the STG machine [Peyton Jones, 1992]. This would solve the above and other problems. The operator tree of the current term would be replaced by an operator graph. The application of function definitions and thus the possibility of live modifications of a definition would remain. However, in our application there is the danger that program modification may have different effects depending on the evaluation strategy. On the one hand, the sharing of variable values at different places in the current term would limit the memory consumption in the Fibonacci sequence defined above, on the other hand it could make it impossible to respect a modification of the called function.

Our single step mode would allow the demonstration and comparison of evaluation strategies in education.

Currently we do not know, whether and how we could embed our system, including live program modifications, into an existing language like Haskell. This would simplify the study of the interdependence between program modifications, optimisations and evaluation strategies and would provide many syntactic and typing features for free.

For this purpose we cannot use an interactive Haskell interpreter like GHCi directly:

- GHCi does not let us access or even modify a running program and its internal representation is optimized for execution and it is not prepared for changes to the running program.
- GHCi does not allow to observe execution of the program, and thus we could not highlight active parts in our program view.

- GHCi does not store the current interpreter state in a human readable way that we can show in our display of the current term.

Nonetheless, we can imagine that it is possible to write an embedded domain specific language. That is, we would provide functions that allow to program Haskell expressions that only generate an intermediate representation that can then be interpreted by a custom interpreter.

Highlighting We have another interesting open problem: How can we highlight program parts according to the music? Of course, we would like to highlight the currently played note. Currently we achieve this by highlighting all symbols that were reduced since the previous pause. However if a slow melody is played parallelly to a fast sequence of controller changes this means that the notes of the melody are highlighted only for a short time, namely the time period between controller changes. Instead we would expect that the highlighting of one part of music does not interfere with the highlighting of another part of the music.

We can express this property formally: Let the serial composition operator `++` and the parallel composition operator `=:=` be defined both for terms and for highlighting graphics. Consider the mapping `highl`, that assigns a term to its visualisation. Then for every two musical objects `a` and `b` it should hold:

```
highl (a ++  b)  =  highl a ++  highl b
highl (a =:= b)  =  highl a =:= highl b
```

If you highlight all symbols whose expansion was necessary for generating a `NoteOn` or `NoteOff` MIDI command, then we obtain a function `highl` with these properties. However this causes accumulation of highlighted parts. In

```
note qn c ++ note qn d ++
note qn e ++ note qn f
```

the terms `note qn c` and `note qn d` would still be highlighted if `note qn e` is played. The reason is that `note qn c` and `note qn d` generate finite lists and this is the reason that `note qn e` can be reached. That is the expansion of `note qn c` and `note qn d` is necessary to evaluate `note qn e`.

JACK support In the future our system should support JACK in addition to ALSA. It promises portability and synchronous control of multiple synthesisers.

Beyond MIDI MIDI has several limitations. For example, it is restricted to 16 channels. In the current version of our sequencer the user can add more ALSA sequencer ports where each port adds 16 virtual MIDI channels. E.g. the virtual channel 40 addresses the eigth channel of the second port (zero-based counting). MIDI through wires is limited to sequential data, that is, there cannot be simultaneous events. In contrast to that the ALSA sequencer supports simultaneous events and our Live sequencer supports that, too.

Thus the use of MIDI is twofold: On the one hand it is standard in hardware synthesisers and it is the only music control protocoll supported by JACK. On the other hand it has limitations. The Open Sound Control protocol lifts many of these limitations. It should also be relatively simple to add OSC support, but currently it has low priority.

5 Acknowledgments

This project is based on an idea by Johannes Waldmann, that we developed into a prototype implementation. I like to thank him, Renick Bell, Alex McLean, and the anonymous reviewers for their careful reading and several suggestions for improving this article.

You can get more information on this project including its development, demonstration videos, and papers at

`http://www.haskell.org/haskellwiki/Live-Sequencer` .

A Glossary

A constructor is, mathematically speaking, an injective function and, operationally speaking, a way to bundle and wrap other values. E.g. a list may be either empty, then it is represented by the empty list constructor `[]`, or it has a leading element, then it is represented by the constructor `:` for the non-empty list. For example, we represent a list containing the numbers 1, 2, 3 by `1 : (2 : (3 : []))`, or more concisely by `1 : 2 : 3 : []`, since the infix `:` is right-associative.

Co-recursion is a kind of inverted recursion. Recursion decomposes a big problem into small ones. E.g. the factorial "!" of a number can be defined in terms of the factorial of a smaller number:

$$n! = \begin{cases} 1 & : n = 0 \\ n \cdot (n-1)! & : n > 0 \end{cases}$$

A recursion always needs a base case, that is, a smallest or atomic problem that can be solved without further decomposition.

In contrast to this, co-recursion solves a problem assuming that it has already solved the problem. It does not need decomposition and it does not need a base case. E.g. a co-recursive definition of an infinite list consisting entirely of zeros is: `zeros = 0 : zeros`

Lazy evaluation is an evaluation strategy for *non-strict semantics*. An alternative name is "call-by-need". It means that the evaluation of a value is delayed until it is needed. Additionally it provides *sharing* of common results.

Non-strict semantics means that a function may have a defined result even if it is applied to an undefined value. It is a purely mathematical property that is independent from a particular evaluation strategy.

E.g. the logical "and" operator `&&` in the C programming language is non-strict. In a strict semantics the value of `p && *p` would be undefined if `p` is `NULL`, because then `*p` would be undefined. However, `&&` allows the second operand to be undefined if the first one is `false`.

Referential transparency means that function values depend entirely on their explicit inputs. You may express it formally by:

$$\forall x, y: \quad x = y \Rightarrow f(x) = f(y) \quad .$$

For mathematical functions this is always true, e.g. whenever $x = y$ it holds $\sin x = \sin y$. However for sub-routines in imperative languages this is not true, e.g. for a function `readByte` that reads the next byte from a file, `readByte(fileA)` may differ from `readByte(fileB)` although `fileA` = `fileB`.

Sharing means that if you read a variable multiple times it is still computed only once and then stored for later accesses.

References

Joe Armstrong. 1997. The development of erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP 1997, pages 196–203, New York, NY, USA. ACM.

Renick Bell. 2011. An interface for realtime music using interpreted haskell. In Frank Neumann and Victor Lazzarini, editors, *Proceedings LAC2011: Linux Audio Conference*, pages 115–121, Maynooth, May.

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, volume 32, pages 263–273, August.

Lejaren A. Hiller and Leonard M. Isaacson. 1959. *Experimental Music: Composition With an Electronic Computer*. McGraw-Hill, New York.

Paul Hudak, T. Makucevich, S. Gadde, and B. Whong. 1996. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June.

John Hughes. 1989. Why functional programming matters. *The Computer Journal*, 32(2):98–107.

James McCartney. 1996. Super Collider. `http://www.audiosynth.com/`, March.

Alex McLean and Geraint Wiggins. 2010. Tidal - pattern language for the live coding of music. In *Proceedings of the 7th Sound and Music Computing conference 2010*.

MMA. 1996. Midi 1.0 detailed specification: Document version 4.1.1. `http://www.midi.org/about-midi/specinfo.shtml`, February.

Simon Peyton Jones et al. 1998. Haskell 98 language and libraries, the revised report. `http://www.haskell.org/definition/`.

Simon Peyton Jones et al. 2012. GHC: The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

Simon Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2(2):127–202, April.

Julian Smart, Robert Roebling, Vadim Zeitlin, Robin Dunn, et al. 2011. wxwidgets 2.8.12. `http://docs.wxwidgets.org/stable/`, March.

Ge Wang and Perry Cook. 2004. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 812–815, New York, NY, USA. ACM.

ipyclam, empowering CLAM with Python

David GARCÍA-GARZÓN
Departament de Tecnologia,
Universitat Pompeu Fabra
Tànger, 122-140
08018 Barcelona
Spain
david.garcia@upf.edu

Xavier SERRA-ROMÁN
Imm Sound, a Dolby Company
Diagonal 177
08018 Barcelona
Spain
Xavi.Serra@dolby.com

Abstract

This paper introduces ipyclam, a new way of manipulating networks in CLAM (C++ Library for Audio and Music) by using the Python language. This extends the power of the framework in many ways. Some of them are exploring and manipulating live processing networks via interactive Python shells, or extending the power of visual prototyping in CLAM by adding complex application logic and user interfaces with PyQt/PySide. The described Python API, ipyclam, by redefining the engine layer, can be reused to control other patching based systems such as JACK, gAlan...

Keywords

Python, CLAM, Qt, patching

1 Introduction

CLAM (C++ Library for Audio and Music)[1] is a free software framework to develop advanced signal processing systems [Amatriain et al., 2007]. Some successful use cases include instruments [Haas, 2001; Mann et al., 2007], voice processing [Sommavilla et al., 2007], audio and music information retrieval [Gómez, 2006; Gouyon, 2005; Amatriain et al., 2005; Ong, 2007], and 3D audio [Arumi et al., 2009; Giulio Cengarle, 2012].

As its name states, CLAM is a C++ framework. General purpose dynamic languages, such as Python, do not mix well with real-time audio programming. Those languages hide aspects that are important to control in real-time programming, for example, memory management and operations that imply system calls that could stall the real-time thread. But real-time restrictions only apply to the processing code. Properly designed audio software separates the real-time code from the rest where those restrictions does not apply: setup, user interface, application logic... CLAM fosters a programming style which clearly localizes real-time code. For the remaining code without real-time restrictions, Python may still have an interesting role to play.

This paper introduces ipyclam, a new way of manipulating CLAM data flow definitions (*networks*) by using the Python language. This extends the power of the framework in many ways. For example, it can be used to build complex networks, like the one shown in Figure 1, that are hard to build by graphical means. Those manipulations could be done interactively, by integrating interactive Python shells like IPython [Pérez and Granger, 2007], into the CLAM patching tool, the NetworkEditor. And last but not least, it extends CLAM graphical prototyping architecture, currently based on graphical design tools that generate fixed data flow and single dialog interfaces. With Python we can add rich application logic and interfaces based on PySide [Bert, 2012] or PyQt [Summerfield, 2007] without raising the difficulty to the point of requiring C++ development.

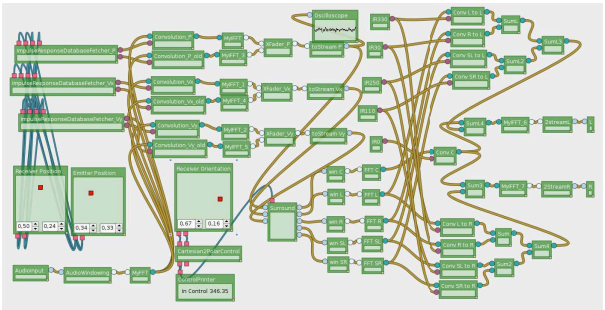


Figure 1: Complex networks are hard to design by pointing and clicking.

The rest of this paper has the following structure: Key concepts of the CLAM framework are introduced in section 2. Section 3 describes the new Python API at user level. Section 4 explains the internal design and how it enables

[1] http://clam-project.org

the reuse of the user API for other patching systems. Section 5 explains how to build PyQt/PySide interfaces that can be related to CLAM networks and how all that leads to a more powerful prototyping architecture. Finally, section 6 evaluates the already reached milestones and the ones that are at reach from now on.

2 CLAM elements

This section will shortly introduce the basic components of the CLAM framework needed to understand this paper. A more insightful description can be found in the referred literature about CLAM.

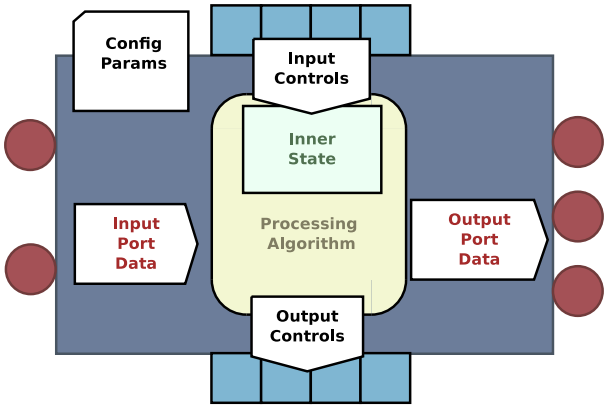


Figure 2: A processing unit

Audio processing is modularized into objects called *processing units* according the CLAM meta-model [Amatriain, 2005]. A processing unit consumes and produces data tokens by its *input and output connectors*. Connectors are called *ports* when data flow is continuous and they are called *controls* when data is sent or received unevenly. Token data can be any C++ type but each connector is bound to a single type. When connecting connectors of different processing units, they must be complementary (input and output), same *kind* (port or control), and same *type* (data token C++ class).

Each processing unit has a set of structured *configuration parameters*. Configuration and connection is done before run-time so that any operation that requires resource allocation can be done outside the real-time thread.

A *network* is a set of interconnected processing units. The network schedules the execution of the units under a given *audio back-end* (PortAudio, JACK, LADSPA, LV2, VST...). Back-end data is fed from and to special units inside the network called *sources* and *sinks*. Then the network topology mandates the data-flow scheduling [Arumí, 2009].

UI *binders* are used to relate a CLAM network to a user interface, currently Qt, but not restricted to it. The programmer can establish such relation by defining custom properties on the elements of the user interface (*widgets*). UI binders detect such properties and add any required stuff to bind them to the network. Common examples of UI binders are the ones used to bind user interface for playback control and monitoring, processing unit configuration, data token visualization, and user control sending.

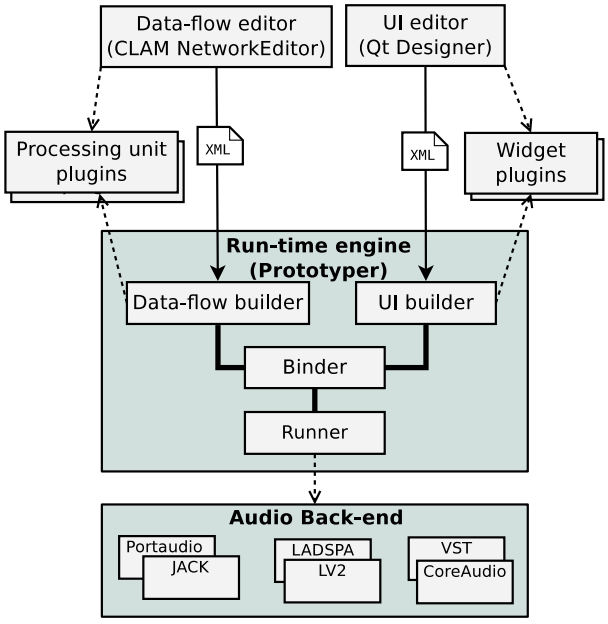


Figure 3: Visual prototyping architecture

All those elements enable the CLAM visual prototyping architecture [Garcia, 2007] illustrated in Figure 3. Both the processing network and the user interface can be designed with graphical tools, CLAM NetworkEditor and Qt Designer respectively. Both can be stored as XML, loaded later in run-time, and related by applying the binders. A tool called Prototyper does that by taking the XML files by command line.

Most elements in this architecture (processing units, token data type handlers, back-ends, UI binders, widgets...) can be extended via plugins. Most of those extendible objects are available through abstract interfaces and factories.

3 ipyclam user API

3.1 Goals

ipyclam's main goal is providing the API to be able to build and explore a CLAM network. Defining the processing code inside processing units is reserved to C++ code to fit real-time constraints.

An explicit choice has been taken on not designing Python API as a direct map of the C++ CLAM API, but to make it conveniently Pythonic. Direct C++ library mapping often leads to a badly designed Python interface. Design decisions taken in C++ API just because of C++ idiosyncrasy, may get pointlessly replicated in Python, and opportunities of using Python features such as attributes, iterators, generators or dynamic interface creation, may get lost.

Another choice is to provide a powerful tab completion for interactive Python shell. It is not just about discovering the static API but taking a step further and discovering the runtime structure of the objects via tab completion. Because of that, such structure should be available as completable attributes.

Convenient ways of expressing things are favored but whenever those convenient ways are not expressive enough to express everything, instead of discarding the convenient one, we add the less convenient one as alternative.

For example, processing units are accessible as network attributes with their names. Also processing ports, controls and configuration parameters are accessible as processing attributes as well. That interface is compact and convenient when doing tab completion.

```
net.Sink.Audio
```

But this is not a general solution. Many names are not valid identifiers. Subscript accessors are provided to solve those cases:

```
net["A processing"]["1"]
```

Still, you may find two subelements of different kind with the same name, or with a name that matches an actual attribute or method of the object. For those cases, it is useful to provide scoping attributes. So the syntax that will always work would be:

```
net.processings["Sink"].inports["Audio"]
```

But this is more verbose than the first proposal. It is a design choice when this kind of situation appears, to provide both the convenient and the complete options so we get the best of them.

3.2 An example

This is a minimal example that creates a 3 channel cable, that just copies the input to the output, and plays it under JACK:

```
from ipyclam import Network, time
net = Network()
# creating units
net.source = "AudioSource"
net.sink = net.types.AudioSink
# configuring
net.source.NSources = 3
net.sink.NSinks = 3
# connecting
net.source > net.sink
# Playing as JACK client for 1 minute
net.backend = "JACK"
net.play()
time.sleep(60)
net.stop()
```

3.3 Creating processing units

Notice that the first processing in the example, *source*, is created by assigning a string, the processing type name, to a new attribute with the name of the processing. The second one, *sink*, is created instead by using the 'net.types' object. Such object is convenient for interactive use to discover the available types by tab completion.

```
>> net.types.Audio [tab]
AudioSink, AudioMixer, AudioSource
...
```

If the unit name is not a proper Python identifier, the subscript syntax can be used as well:

```
net["My Sink"] = net.types.AudioSink
```

3.4 Configuring

Configuration parameters can be accessed directly as direct attribute or subscript of the processing unit. They can be accessed as well inside the scoping attribute *config* to avoid conflicts with other processing subelements or common attributes and methods.

```
net.source.config.NSources = 3
```

Every time a parameter is set, the object is reconfigured, but reconfiguration may be an expensive process. To address that issue reconfiguration may be held while setting a set of parameters for a given unit using the *with* statement:

```
with net.mymodule.config as c :
  c.AParameter = "A Value"
  c.AnotherParameter = 23.2
```

Configuration parameters are typed, type checking is done on assignment rising *TypeError* if the type is not the proper one. In CLAM, configuration parameters can be instantiated or not. In Python uninstantiated state is represented by the *None* value.

Some configurations are structured using parameters that are configurations themselves. Such sub-configurations can be accessed as natural by accessing successive attributes.

```
net.mymodule.SubConfig.Param1 = 4
```

3.5 Connecting

The example uses the greater-than operator to establish the connection. Both sides of the operator refer to the processing units, but indeed what gets connected are the connectors. So this is a short-cut for connecting each port pair-wise:

```
net.source["1"] > net.sink["1"]
net.source["2"] > net.sink["2"]
net.source["3"] > net.sink["3"]
```

Or, generally, by using the iterators of *inports* and *outports* attributes:

```
for inport, outport in zip(
    net.source.outports,
    net.sink.inports,
  ) :
  inport > outport
```

Similar iteration can be done with *incontrols* and *outcontrols* processing unit attributes. They can be used as well with Python slices. For example, if we want to reverse the channels:

```
net.source > net.sink.inports[::-1]
```

Or first and third to the first two:

```
net.source.outports[::2] > \
  net.sink.inports[:2]
```

3.6 Playback control

The audio back-end can be set by assigning the *backend* special network attribute. For example, if we wanted to use the PortAudio audio backend we could use:

```
net.backend = "PortAudio"
```

The network has several methods, *pause()*, *play()* and *stop()*, to control the playback, and several methods, *isPlaying()*, *isPaused()* and *isStopped()*, to query the playback status.

3.7 Serialization

Networks can be loaded from XML files generated by NetworkEditor.

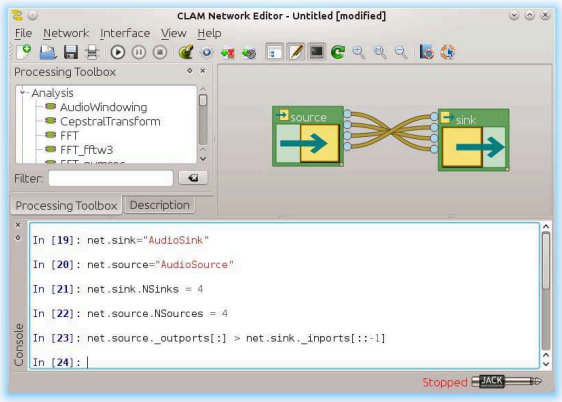


Figure 4: IPython console integrated in the NetworkEditor interface

```
net = Network()
net.load("mynetwork.clamnetwork")
net.save("mynetwork-copy.clamnetwork")
```

Indeed you can get the XML string for the current network using *xml()*.

```
print net.xml()
```

Although XML is somehow readable, in fact, we found that Python code is even more readable than XML. An ipyclam network is able to generate code to reconstruct itself.

```
>> print net.code("mynet")
mynet = Network()
mynet.sink = "AudioSink"
nynet.sink.NSinks = 3
...
```

This feature is quite powerful. Given a static network stored as XML, it can be converted to Python code and as Python code it can be parametrized or turned into a more smart program.

This also opens the door to the use of Python code as serialization format instead of XML. Indeed Python code using ipyclam API is more compact and readable than XML. Despite that, deprecating XML is not yet an option as it is not save to use Python interpreter as parser. A Python interpreter will allow to execute more than just network definitions.

4 Implementation

This section gives a slight overview on how ipyclam API has been internally implemented and how this design allows extending the use of the API to control other patching systems.

ipyclam is designed in two layers as shown in Figure 5. The user layer is the one that provides

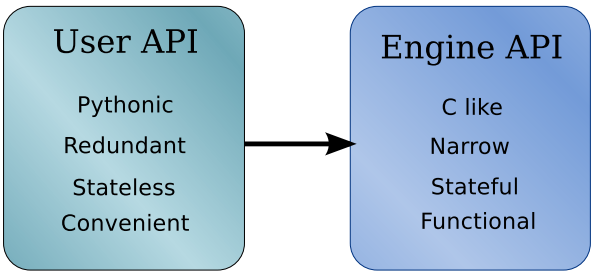


Figure 5: Two layers architecture

the API explained on previous sections, with all the sugar for the many redundant and pythonic ways of expressing the same operation.

But that layer is stateless. In order to perform the actual operations it relies on a engine layer which holds the actual state, in this case, the C++ CLAM Network object. Those many ways of performing a given operation at the user API converge in a single entry point at the engine layer resulting in a narrower API at that level.

This design in two layers strengthens the reliability of the implementation. The user API can be developed ignoring all the complexities of the adapters to the C++ CLAM code by providing a mock-up engine in pure Python. A narrow engine API reduces the number of operations to test for the engine and centralizes the state checks for the front-end testing. A stateless front-end avoids errors on the bookkeeping of duplicated information.

Another positive side effect of this design is that this narrow engine API can be reimplemented to address any other patch like systems, such as JACK, Patchage, gAlan... As result all the rich ipyclam API interface can be reused for those systems. Other patching programs can integrate the Qt console like the one that now NetworkEditor has and is shown in Figure 4.

5 Prototyping user interfaces

CLAM visual prototyping architecture, explained in section 2, provided a way to build a simple audio application by joining two parts designed visually: a CLAM network and a Qt Designer interface. Although that architecture generated decent applications, it has a clear ceiling of what you can build. Applications are limited to simple application logic, a single dialog and a fixed processing data-flow. If anyone wants to go beyond that, C++ programming skills are required, so the learning threshold goes up and the development work-flow gets harder and slower [Garcia, 2007].

An intermediate solution is to introduce Python as programming language for the user interface and application logic. Python is easier to learn and has a faster development work-flow. This section explains some features of ipyclam that facilitate building such applications and shows some examples that illustrate the scope of what you can do.

5.1 PyQt4 and PySide

Two Python bindings are available for Qt: PyQt4[2] and PySide[3]. Each one uses a different binding generator technology: PyQt4 uses SIP while PySide uses Shiboken. The resulting Python APIs are mostly identical, so writing Python code that works for either is not hard. ipyclam supports both. In the following examples, PyQt4 is used but using PySide is just a matter of changing the `import` lines.

5.2 A Python based Prototyper

The following Python code provides a simplified version of Prototyper.

```
import ipyclam, sys
from PyQt4 import QtGui
import ipyclam.ui.PyQt4 as ui
# network setup
net = Network()
net.backend = "JACK"
net.load(sys.argv[1])
# ui setup
app = QtGui.QApplication([])
w = ui.loadUi(sys.argv[2])
net.bindUi(w)
# run
w.show()
net.play()
app.exec_()
net.stop()
```

The interesting bits are the `loadUi` function from the `ipyclam.ui.PyQt4` module and the `bindUi` method of the network. The `loadUi` function is a helper that instantiates a Qt Designer file. The `bindUi` method applies all the available binders to the user interface. Possible bindings are searched recursively so you can use it with a full interface as well as a single widget.

This snippet has the same restrictions as Prototyper: It is general but it is limited to a single processing data flow and a single interface with no application logic.

[2] http://www.riverbankcomputing.com/software/pyqt
[3] http://www.pyside.org

The good news is that now we can change that code to modify the network with the ipyclam API exposed on previous versions and modify the interface with regular PyQt4/PySide API.

5.3 Building interfaces from scratch

A counterexample would be building the processing network and the interface without XML files, that is, using ipyclam and PyQt4/PySide APIs. A problem with this approach is that some useful audio widgets provided by CLAM as Qt plugins have no specific Python wrappers. Providing such wrappers would imply to generate them for SIP and Shiboken for each specific widget class in the plugin. Instead, ipyclam provides a helper method to access the Qt widget factory, which creates the widgets from the class name string. Factory created widgets are handled by the generic QWidget interface, which includes composing them and accessing their properties.

The following example implements an oscilloscope, by binding a CLAM Oscilloscope widget with an AudioSource.

```
import ipyclam, sys
from PyQt4 import QtGui
import ipyclam.ui.PyQt4 as ui
# network setup
net = Network()
net.backend = "JACK"
net.source = net.types.AudioSource
# ui setup
app = QtGui.QApplication([])
w = ui.createWidget("Oscilloscope")
w.setProperty("lineColor", "red")
w.setProperty(
  "clamOutPort", "source.1")
net.bindUi(w)
# run
w.show()
net.play()
app.exec_()
net.stop()
```

This example accesses specific behaviour of the Oscilloscope, the *lineColor*, by using the generic property interface. The same method is used to set the binding property *clamOutPort* that in a visually designed prototype should have been defined with Qt Designer.

5.4 Hybrid approaches

Any combined approach is feasible. Figure 6 shows an example that comes with ipyclam that combines a Qt Designer file with a coded interface. Indeed, this example has some application logic not available with simple visual prototyping. Notice that the combo box is filled with

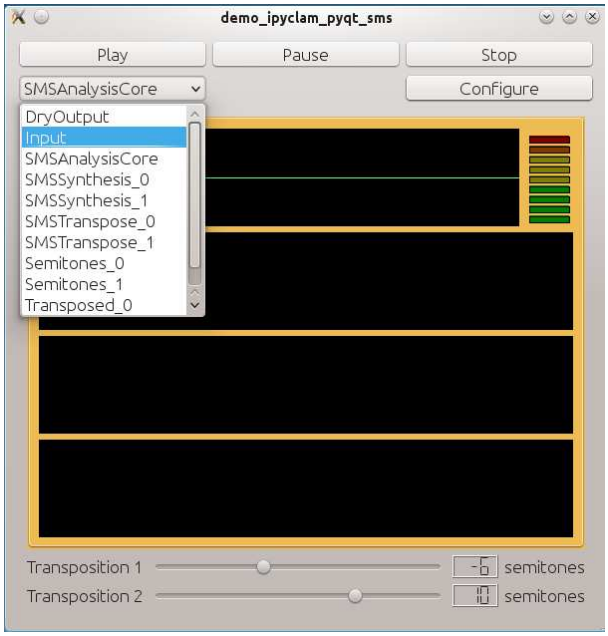


Figure 6: Extending with Python an existing visual prototype that uses Spectral Modeling Synthesis for a two voices transposition. The extension provides detailed configuration of every unit.

information, the names of the processing units, taken from the network with the ipyclam API. The configure button, instead of being a bound widget, activates a function that takes the currently selected processing unit, and launches a configuration dialog bound to the given processing configuration.

6 Conclusions

The API presented in this paper offers a new way of developing real-time audio applications by combining the power and flexibility of CLAM, Qt and Python. The API has been designed with a strong stress on convenience and expressiveness which results in very readable and compact code.

An interactive Python console has been integrated with the graphical patching tool. This enables the user to build complex networks by interactive programming, and having visual feedback of the results. This work can be easily extended to other patching systems just by implementing a narrow API.

Indeed, a promising engine to implement in the future is one relying on JACK because CLAM users are likely to be interested in controlling JACK application interconnections

from the console, just as they control inner units.

Another work to be done is providing some useful examples built with ipyclam that give potential users a clear idea of the horizons of the platform. They also will help to mature the API highlighting any unpolished edges left.

Right now, the platform excludes Python for processing tasks. But Python has a nice collection of numerical libraries based on the numpy package [Ascher et al., 1999]. They could be used for processing algorithms for off-line processing or situations where lesser real-time conditions are required. Two approaches are being considered. One is being able to implement processing units in Python. The other is a Python audio back-end where Python code feeds the network with numpy arrays as audio input and output.

7 Acknowledgements

We would like to thank Pau Arumí, Natanael Olaiz and Eduard Aylon for helping us to define and test the API. Part of this work has been done using technical resources gently offered by Fundació Barcelona Media and ImmSound.

References

Xavier Amatriain, Jordi Massaguer, David Garcia, and Ivan Mosquera. 2005. The clam annotator a cross-platform audio descriptors editing tool. 1

Xavier Amatriain, Pau Arumi, and David Garcia. 2007. A framework for efficient and rapid development of cross-platform audio applications. *ACM Multimedia Systems Journal*. 1

Xavier Amatriain. 2005. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. Ph.D. thesis, Universitat Pompeu Fabra. 2

Pau Arumi, Natanael Olaiz, and Toni Mateos. 2009. Remastering of movie soundtracks into immersive 3D audio. In *Proceedings of Blender Conference 2009*. 1

Pau Arumí. 2009. *Real-time Multimedia Computing on Off-the-Shelf Operating Systems: From Timeliness Dataflow Models to Pattern Languages*. Ph.D. thesis, Universitat Pompeu Fabra. Master Thesis. 2

David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant, 1999. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition. 7

A.C. Bert. 2012. *Pyside*. Chromo Publishing. 1

David Garcia. 2007. Visual Prototyping of Audio Applications. Master's thesis, Universitat Pompeu Fabra. 2, 5

Giulio Cengarle. 2012. *3D audio technologies: applications to sound capture, post-production and listener perception*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Fabien Gouyon. 2005. *A computational approach to rhythm description — Audio features for the computation of rhythm periodicity functions and their use in tempo induction and music content processing*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Emilia Gómez. 2006. *Tonal Description of Music Audio Signals*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Joachim Haas. 2001. Salto - a spectral domain saxophone synthesizer. In *Proceedings of Mosart Conference 2001*. 1

Steve Mann, Ryan Janzen, and James Meier. 2007. The electric hydraulophone: A hyperacoustic instrument with acoustic feedbacks. In *Proceedings of the 2007 International Computer Music Conference (ICMC2007)*, pages 27–31. 1

Bee Suan Ong. 2007. *Structural Analysis and Segmentation of Music Signals*. Ph.D. thesis, University Pompeu Fabra, Barcelona, Spain, February. 1

Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May. 1

Giacomo Sommavilla, Carlo Drioli, Piero Cosi, and Giulio Paci. 2007. SMS-FESTIVAL: a New TTS Framework. In *Models and analysis of vocal emissions for biomedical applications: 5th International workshop*, pages 89–92. Firenze University Press, December 13-15. 1

Mark Summerfield. 2007. *Rapid gui programming with python and qt: the definitive guide to pyqt programming*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition. 1

Music for Programmers (MFP): A Dataflow Patching Language

Bill GRIBBLE
grib@billgribble.com

Abstract

MFP is a graphical dataflow patching language in the tradition of Max/MSP and Pure Data. It expands on its predecessors by integration of higher-level language constructs from Python, including a variety of data types and operations and the widespread use of the Python evaluator. A new lexical scoping system, a *layers* approach to building logical code blocks, and a UI optimized for keyboard control are also featured.

Keywords

Patching languages, Python, JACK, OSC, Pure Data

1 Introduction

Graphical patching languages have a number of basic principles in common. A "patch" is a computer program specified by a diagram. The patching system acts as the development environment, compiler, and interpreter for this program. The diagram consists of processing elements and connections between them in the form of *patch cords* or virtual wires. Typically patches exist and operate in 3 domains: the graphical domain, which includes the visual elements displayed in the patch and any interactive controls such as buttons and sliders; the control or symbol domain, where patch elements communicate by sending discrete messages; and the signal domain, where communication is in blocks of audio data.

Possibly because the "patching" metaphor is so familiar to electronic musicians, there exist several patching languages for audio and music, both commercial and FLOSS. Notable examples include Miller Puckette's languages (Max/MSP [Puckette, 1989], jMax, and Pure Data [Puckette, 1997]), Blechmann's Nova/SuperNova [Blechmann, 2008], and Ross Bencina's AudioMulch [Bencina, 1998]. The notion of a visual graph of processing nodes is also popular in other domains, notably scientific data collection where National Instruments' LabView [National Instruments, 1986] has used this metaphor since 1986.

In the Linux audio world, Pure Data is probably the leading patching system. Its large library of built-in and third-party modules make it a versatile toolbox for audio synthesis, performance control, interfacing with experimental input and output devices, video creation, and video interpretation. The user and developer communities are full of enthusiastic, helpful, and talented people. In short, Pure Data is awesome.

However, in my experience with Pure Data I have been frustrated at times with the difficulty of simple operations with basic data types like strings and lists. I am more proficient as a programmer than as a musician, so this kind of thing annoys me perhaps more than most PD users. Often third-party packages are required to perform what seem to be elementary operations on data. Interpreting literal data entered in message boxes, or building non-numeric values, often requires trial-and-error and results in solutions that are nonintuitive for the non-guru. Resolving issues of names and namespaces often involves what appear to be kludgy solutions (I'm looking at you, `$0`).

When I was faced with tackling a significant project in PD (a system to analyze the dynamic behavior of a piece of external audio equipment), I simply could not bring myself to do it. I wanted a different tool, with more support for general-purpose programming and a more familiar approach to data. This was the genesis of MFP.

I began to explore starting from a few basic goals:

> Use Pure Data's graphical metaphor and idiom as a baseline, without attempting to preserve compatibility
>
> Expose Python wherever possible, and use plain Python data natively

Rethink name resolution and scoping

Implement a clean and simple GUI that assists in the construction of patches

Expand the range of system, file, and string operations, to make general-purpose programming easier

Integrate readily into a variety of audio production workflows as an instrument, a forensic tool, or an audio swiss army knife

The work-in-progress result of this exploration is MFP. MFP includes elements familiar to high-level language programmers, with a standard library and graphical presentation layer that will be familiar to users of Max/MSP and Pure Data, though there are many differences large and small.

My hope is that it will appeal to patching musicians while providing a stronger foundation for analytical, scientific, and general-purpose programming. The popularity of tools like LabView in domains other than music shows that dataflow patching systems can be useful in a variety of control and analysis tasks, given an appropriate infrastructure. MFP should be of interest to musicians, particularly those focused on the symbolic domain (MIDI, OSC, and generative music applications) where Python will provide significant leverage, but also to audio software developers, plugin authors, and recording engineers who need to build custom tools to interact with audio data.

2 Architecture

MFP is implemented in Python 2 [Van Rossum, 2010a], with C extensions for real-time DSP. In order to mitigate Python's Global Interpreter Lock (GIL) bottleneck [Van Rossum, 2010b], processing for each of the three domains (graphical, control, and signal) is performed in a separate process. The three processes ("nodes") are coupled via the `multiprocessing` facility present in Python 2.6 and later.

A patch, as represented in the user interface for editing or control, appears as a multi-layered diagram of visual elements such as boxes, controls (sliders, buttons) and displays (indicators, meters, signal graphs) connected by lines representing communication pathways. Some connections terminate in *vias*, which represent invisible communications between endpoints. Each element in the display domain has a corresponding unit or connection in the control domain, and may or may not have elements in the signal domain. The use of layers, "wired" connections within a layer, and vias between layers evoke a printed-circuit board metaphor, but this is not rigorously followed.

Figure 1: "Hello, world" program in MFP (`doc/hello_world.mfp`)

The "hello, world" example in Figure 1 demonstrates the basic properties of an MFP patch in the simplest way. The literal string "hello, world!" is contained in a *message box*, which is an interactive element that emits its contents when clicked. The `print` object is a *processor* which prints its argument to the MFP log window.

In this example we see the first differences from Pure Data: in PD, it is not possible without some difficulty to print messages containing commas, since strings are not one of the basic "atom" types that can be represented in message boxes. In MFP, the literal contents of the message box are interpreted at creation time by the Python evaluator; a message box can contain any Python expression, including literal data or code that evaluates to a Python object. In this case, if the message box was filled with the text `", ".join(["hello", "world!"])`, which is an idiomatic Python expression for joining a list of strings into a single comma-separated string, it would have produced the same message to the log when clicked: `"hello, world!"`

2.1 Layers

Layers break a patch into "pages", providing visual grouping and separation of elements, and are are somewhat equivalent to code blocks in traditional languages or subpatches in Pure Data. Layering is a key mechanism for program decomposition in MFP.

Figure 2 shows views of a more complex multi-layered patch and the application context when using it. This patch implements a basic looping sampler inspired by the Akai Headrush

looping pedal. Four layers group the elements of the patch into blocks. The Front Panel layer of the patch contains the user interface: signal level meters for input and output, indicators of current state, and two buttons to control the sampler. The Buffer Control layer contains the state machine controlled by the front-panel buttons, with transition actions that change the configuration of the sampling buffer at the core of the patch. The third layer, Audio Processing, contains the signal input/output and the sampling buffer object. Finally, the Indicators layer updates the front-panel indicator toggles based on the state machine state.

This patch demonstrates how layers enable program decomposition in a way similar to Pure Data's subpatches. Each layer contains a block of functionality, with send/receive vias showing names for inputs and outputs of the block. The organization is not as structured as Pure Data's subpatches, but has the advantage of being clearly a different mechanism from patch reuse (Pure Data "abstraction").

The Python-familiar will note that the contents of many of the message boxes in the Buffer Control layer (those which are a series of comma-separated `key=value` assignments) are not exactly valid Python code. This is MFP-specific syntactic sugar for the Python expression `dict(key1=value1, key2=value2)` to create a dictionary object.

2.2 Control domain

The control domain is the backbone of MFP's processing, and the control node is the master and controller of the MFP application.

The control node hosts zero or more patches, where each patch is an instance of the `Patch` class. Each `Patch` consists of a connected graph of processors (instances of the `Processor` class). A `Processor` instance has one or more inlets and zero or more outlets. Each inlet may be connected to zero or more outlets of other processors, and each outlet to zero or more inlets of other processors. Communication between processors consists of sending a message from an outlet of one processor to an inlet of another. The distinguishing feature of control domain communication is that it happens in discrete chunks called *messages*.

Messages sent between processors in the control domain are ordinary Python objects of any type: numbers, strings, lists, dicts, functions, or other class instances. This is significantly

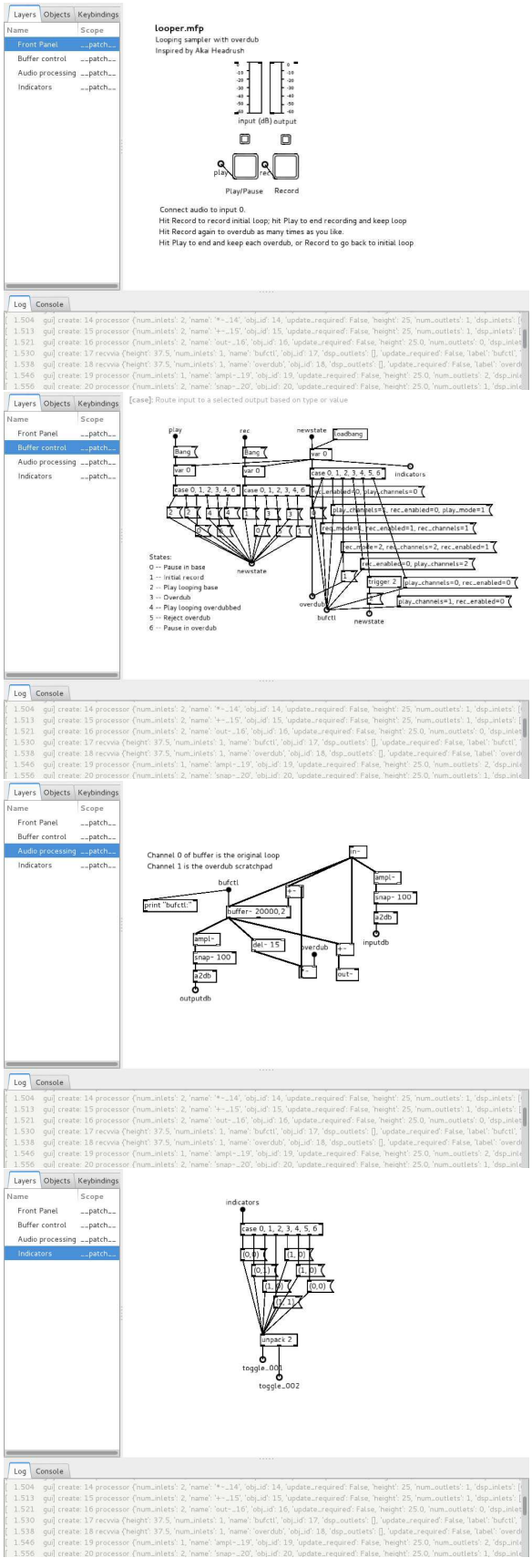


Figure 2: Looping sampler (`doc/looper.mfp`)

different from Pure Data and other patching languages, which define a limited set of "atom" types that can be used within the patch.

In many cases it is useful to think of a message processor as a function or method, where the arity is determined by the number of inlets. This model is supported by MFP's default marshaling policy, which buffers inputs to all inlets until a message is received on a "hot" inlet. Pure Data also takes this approach. By default, only the leftmost inlet (inlet 0) is "hot", but that behavior may be changed by a particular `Processor` subclass. The processor's trigger method is then called to perform message processing. Functions of null arity are triggered by any input on their inlet (by convention, the special value `Bang`).

2.2.1 Method calls and dispatching

In other cases it is useful to think of a processor as an object with methods of its own. For example, a number box might have an API to control the number of decimal digits to display. In interactive usage, configuration of this property might be accomplished by a dialog or key sequence, but the underlying mechanism is going to call a `configure` method somewhere down the line. In the spirit of exposing Python where possible, we allow patches to directly call methods on the objects that make up the patch.

The control domain structure of MFP matches fairly neatly with a *message passing* metaphor for method calls. A method is called on a control-domain object by sending it a message representing the method call. In MFP, the message is an instance of `MethodCall`. This object captures the name and any arguments of the method call (other than the object to call the method on, which is always the recipient of the `MethodCall` message).

MFP provides classes and syntax to support this style of usage by allowing for concise and flexible creation of `MethodCall` objects. The example in Figure 3 shows a patch fragment that uses `@conf(digits=3)` as syntactic sugar for `MethodCall("conf", digits=3)`. When this message is received by the number box, the method `conf` is called, with the keyword argument `digits` having the value of 3. The `conf` method is supported by all `Processor` instances as a way to directly set GUI display parameters.

Note that this activity is represented by objects in the graphical domain (mostly `PatchElement` subclasses) but the Python evaluation and message passing all takes place in the control domain.

Figure 3: Sending a method-call object to change displayed digits (`doc/enum_control.mfp`)

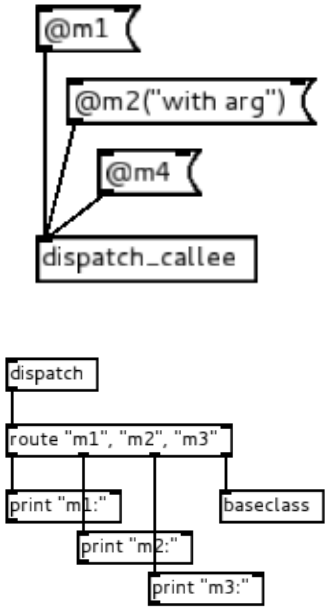


Figure 4: Custom method dispatch (`doc/dispatch_caller.mfp`, `doc/dispatch_callee.mfp`)

User patches can dispatch their own method calls using the `[dispatch]` builtin. This processor outputs any method call objects sent to the patch as a `(name, MethodCall)` tuple suitable for input into the `[route]` processor. The companion `[baseclass]` processor handles method resolution for methods implemented by the base class. Figure 4 shows a patch fragment handling dispatch of methods `m1`, `m2`, and `m3`, while passing all others back to the `Patch` class for resolution as methods of the Python `Patch` class or its base class `Processor`.

2.2.2 Names and scoping

At first glance, names may not seem to be that important in a patching language. Patch connections directly designate the caller and callee objects without need for names. In reality, larger patches need to hide some connections for readability and structure. Pure Data provides the `[s name]` and `[r name]` pair (send/receive), which create a "virtual patch cable", as well as some special message-box syntax to send messages directly to an `[r]`. MFP uses send/receive via pairs for the same purpose. In both cases, names are required to connect sender to receiver.

MFP gives each `Patch` a lexical scope, and allows each layer of the patch to either use the patch scope or to specify a different one. Separate scopes can make it possible to hygienically copy a layer or a group of layers without name collisions. For example, a synthesizer patch could use hygienic layer duplication to create a dynamic number of polyphonic voices, if a single voice was built in a layer or set of layers sharing a scope distinct from the patch scope.

As a consequence, there is no need to "mangle" names to make them unique to a patch instance in MFP. Names are automatically scoped within the patch instance where they are created. This contrasts with Pure Data, where names are global by default; names intended to be local use the magic variable `$0`, which expands to a unique-per-patch symbol, as part of the variable name. For instance, the name `foo` would be global, and a message sent to `foo` from any open patch will go to all recipients of `foo` messages. `$0-foo` would be local to the patch containing it.

2.3 Signal domain

The signal domain component of MFP is primarily implemented in a C library containing the Python extension `mfpdsp`. MFP uses the JACK Audio Connection Kit (JACK) [JACK Team, 2002] to interface with the system audio hardware and other audio applications.

As in the control domain, processing is implemented in a connected graph of processing nodes. However, communication between nodes is a block-based stream of sample data rather than a sequence of messages. For simplicity, the processing block size used is always the JACK block size, but this will likely change in future releases.

A patch element which performs signal processing activity will have both a control domain representation (an instance of a `Processor` class) and a signal domain representation (a C-allocated instance of `struct mfp_proc`). The identity between the two is maintained using an integer `obj_id` which is shared.

JACK does the hard work in the signal layer, and there are only a few built-in DSP operations (about 20 in all) which, frankly, are not that interesting: arithmetic, comparisons, simple oscillators/noise, delay/buffering, simple filters, envelope follower. It is expected that LADSPA (and, later: LV2, etc) plugins ([LADSPA Team, 2000], [LV2 Team, 2008]) will provide the more sophisticated DSP processing tools.

The graph topology of the signal processing network imposes some ordering on execution of the node algorithms. A particular node is only ready to process when all its inputs have been computed. To manage this, a simple scheduling step is performed whenever nodes are added or removed. Units that are marked as *generators* (their output in a particular processing cycle is not dependent on their input during that cycle) are always ready to process, nodes directly connected to them may be processed next, and so on. It is possible for cycles in the connectivity graph to make a network that cannot be scheduled. In this case, a delay of at least one processing block must be added to break the cycle.

Communication within the signal layer is driven by the JACK callback thread and is decoupled from the message domain. Interaction between message and signal occurs at block boundaries and consists of parameter get/set and simple messages from the signal back to the message layer. This allows the real-time component of the system to operate without a dependency on the timeliness of processing in the message domain.

2.4 Graphical domain

The graphical UI borrows its appearance heavily from Pure Data. Each processor is represented by a visual element, with connections represented by lines. Most processors have simple flow-chart style representations, with distinctive shapes providing cues as to their function.

Control of the UI is largely keyboard-driven. The modal input system is patterned after text editor controls, stacking modes based on the current context. Authoring and editing of a

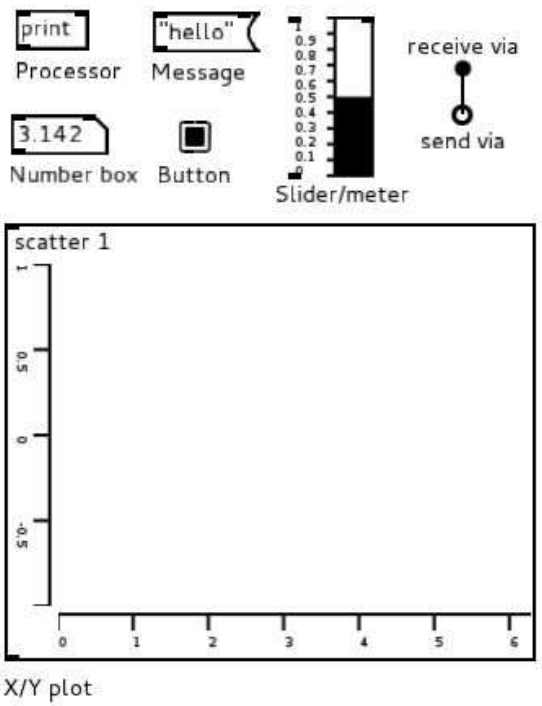


Figure 5: Graphical patch element types (`doc/gui_elements.mfp`)

patch can be accomplished without using the mouse or touchpad at all, though a combination of pointing and typing is more efficient.

The UI is implemented using the Clutter ([Clutter Team, 2006]) and Gtk+ toolkits. Figure 5 shows samples of each visual type of element, though one representation (such as the button) can have several distinct identities depending on parameters (clickable or display only, momentary or latching, etc). The functional element types are:

Processor box: The most common element, a plain box containing the name and initialization arguments of the processor. Arguments are interpreted by the Python evaluator at creation time.

Message box: Interactive element emitting a message when clicked. The message is displayed in the element and is interpreted by the Python evaluator when entered through the GUI.

Text comment: Free text display. Uses Pango markup[1] to enable a variety of text styles, sizes, fonts, and colors.

Slider control/Bar meter display: Vertical or horizontal slider/meter with optional scale display. Displays a solid bar indicating a value, draggable for slider control

Number box: A simple box for interactively entering or editing a number. Responds to mouse and keyboard actions to increment/decrement the value, and emits the value as a message when it is changed.

X/Y chart: Multi-curve scatter/line chart with plot, roll, and signal-view (oscilloscope) modes. Can work in the control domain as a scatter plot or strip chart, or in combination with a shared memory signal buffer as an oscilloscope-type display.

Toggle button/indicator: Two-state button with visual indicator of on/off state. When created as an indicator, it shows the underlying state but does not respond to clicks.

Momentary ("bang") button: Emits a `Bang` object (or other object as configured) when clicked. This is similar to a message box, but does not display the value to be sent.

Send and receive vias: Circular pads representing the end points of an invisible virtual patch cord (`[s name]` and `[r name]` in Pure Data). The name and appearance are inspired by printed circuit board vias, which are conductive pathways connecting one layer of a circuit board to another.

3 Extensibility

User-created processing modules in the signal and message domains can be loaded at runtime via several mechanisms:

Patch file discovery. When a reference to an unknown processor type is made, the search path is crawled to find a patch file (*.mfp) with a matching name. User patches are equivalent to builtins, except for the additional overhead of network iteration.

Processor subclassing. User code loaded at startup or patch load time can create new Processor subclasses which can be referenced in patches.

User-specified function wrapping. A utility API is provided to wrap arbitrary Python functions or methods as MFP processors. Code in user rcfile or other startup file can create simple or complex processor types using these tools.

Automatic Python callable wrapping. An attempt to create a Processor will succeed if any Python function with the specified name is known to the MFP evaluator at runtime. The matching function will be automatically

[1] An SGML-like syntax, see `http://www.gtk.org/api/2.6/pango/PangoMarkupFormat.html`

wrapped in a simple Processor subclass that uses introspection to discover the arity of the provided function.

Compiled DSP processor definitions. Dynamically linked libraries containing DSP type definitions can be loaded at runtime via `dlopen`. A simple C-language processor type definition API makes creation of new unit types straightforward.

Plugin hosting. LADSPA plugins can be hosted and controlled via the `[plugin~]` builtin.

4 Interoperability

MFP implements interfaces to other software via open standards:

JACK: MFP is a standalone JACK application. The number of input and output ports is specified at app startup time. Support for JACK MIDI, transport, and timecode is planned.

Open Sound Control (OSC): Every MFP object in the control domain has an OSC address and can receive messages in numeric or Python expression form. Every `Processor` supports OSC controller learning via the `@osc_learn` method. OSC message send and additional routing for incoming messages are provided through builtin processors `[osc_in]` and `[osc_out]`.

MIDI: The ALSA sequencer API is used to provide MIDI I/O. MIDI data is processed in the message domain, and is routed in and out via the `[midi_in]` and `[midi_out]` builtins. Chasing and generating MIDI timecode is planned.

LADSPA: A builtin DSP type hosts LADSPA plugins. LADSPA plugin metainformation is used to add input and output ports for all plugin parameters at run time. An LV2 host is planned.

5 Implementation status

As of the submission of this paper (Feb 2013) MFP is under heavy development leading up to an initial public release. The functionality described in this document is implemented and exercised by demonstration patches provided in the source code repository.

Much of the development process has been exploratory in nature, so the feature set as a whole is a bit spotty. Significant features are missing or incomplete, including the ability to have more than one patch open for editing, exported patch UIs ("graph-on-parent"), hosting LV2 and other types of plugins, support for session APIs, online help and other documentation, undo/redo, menu control of most functions, and saved configurations (presets) in patches.

6 Getting MFP

Source code and issue tracking for MFP are on GitHub:

`https://www.github.com/bgribble/mfp`

The project is licensed under the GNU General Public License (GPL) version 2. Your interest and participation is invited and welcomed.

References

R. Bencina. 1998. (Software) AudioMulch. `http://www.audiomulch.com`.

T. Blechmann. 2008. nova - A New Computer Music System with a Dataflow Syntax. Bachelor's thesis, Vienna University of Technology, Vienna, Austria.

Clutter Team. 2006. (Software) Clutter Project. `http://www.clutter-project.org`.

JACK Team. 2002. (Software) JACK Audio Connection Kit. `http://www.jackaudio.org`.

LADSPA Team. 2000. (Software) LADSPA Project. `http://www.ladspa.org`.

LV2 Team. 2008. (Software) LV2 Project. `http://lv2plug.in`.

National Instruments. 1986. (Software) NI LabVIEW. `http://www.ni.com/labview/`.

M. Puckette. 1989. (Software) Max/MSP, currently distributed by Cycling '74. `http://www.cycling74.com/max`.

M. Puckette. 1997. (Software) Pure Data. `http://www.puredata.info`.

G. Van Rossum. 2010a. Python 2.7 Documentation. `http://docs.python.org/2.7/`.

G. Van Rossum. 2010b. Python 2.7 Python/C API Reference Manual: Thread State and the Global Interpreter Lock. `http://docs.python.org/2/c-api/init.html#threads`.

A *Pure Data* toolkit for real-time synthesis of *ATS* spectral data

Oscar Pablo DI LISCIA
Universidad Nacional de Quilmes
Roque Saenz Peña 180
Quilmes, Argentina, 1876
odiliscia@unq.edu.ar

Abstract

This paper presents software development and research on the field of digital audio synthesis of spectral data using the *Pure Data environment* (Miller Puckette et al) and the *ATS* spectral analysis technique (by Juan Pampin [6]). The *ATS* technique produces spectral data using a deterministic-plus-stochastic representation. The focus is on the methods by which such data may be real-time read and synthesized using several *Pure Data* externals developed by the author and others, as well as on the involved audio synthesis strategies. All the software involved in this development are GNU Licensed and run under Linux.

Keywords

Digital Signal Processing, Sound Analysis, Computer Music.

1 The *ATS* analysis technique

1.1 General

The *ATS* technique (*Analysis-Transformation-Synthesis*) was developed by Juan Pampin. Its comprehensive study exceeds the goal of this paper[1] but, essentially, it may be said that it represents two aspects of the analyzed signal: the deterministic part and the stochastic or residual part. This model was initially conceived by Julius Orion Smith and Xavier Serra [11], but *ATS* refines certain aspects of it, such as the weighting of the spectral components on the basis of their *SMR*[2].

The deterministic part consists in sinusoidal trajectories with varying amplitude, frequency and phase. It is achieved by means of the depuration of the spectral data obtained using *STFT* (*Short-Time Fourier Transform*) analysis.

The stochastic part is also termed *residual*, because it is achieved by subtracting the deterministic signal from the original signal. For such purposes, the deterministic part is synthesized preserving the phase alignment of its components in the second step of the analysis. The residual part is represented with noise variable energy values along the 25 critical bands [12].

The ATS technique has the following advantages:

a-The splitting between deterministic and stochastic parts allows an independent treatment of two different qualitative aspects of an audio signal.

b-The representation of the deterministic part by means of sinusoidal trajectories improves the information and presents it on a way that is much closer to the way that musicians think of sound. Therefore, it allows many 'classical' spectral transformations (such as the suppression of partials or their frequency data transforming) in a more flexible and conceptually clearer way.

c-The representation of the residual part by means of noise values among the 25 critical bands simplifies the information and its further reconstruction. Namely, the common artifacts that arise in synthesis using oscillator banks or *IDFT*, when the time of a noisy signal analyzed using a FFT is warped may be completely suppressed[3].

1.2 Performing and storing ATS analysis

ATS was initially developed for the *CLM* environment (*Common Lisp Music* [5]), but at present there exist several *GNU* applications that can perform the *ATS* analysis, among them the *Csound* Package command-line utility *ATSANAL* [8][4], and the *ATSH* software (Di Liscia, Pampin,

[1] For a detailed reference of this technique, see [6].

[2] *Sound-to-Masking ratio*, see [12].

[3] This is possible because the residual part representation allows its synthesis using noise generators. More on this will be further explained in this paper.

[4] ATSANAL is based on the program ATSA (by Pampin, Di Liscia and Moss) and was ported to Csound

Moss [3]). The analysis parameters are somewhat numerous, and must be carefully tuned in order to obtain good results [2].

As documented in [3], the ATS files store a representation of a digital sound signal in terms of sinusoidal trajectories (called *partials*) with instantaneous frequency, amplitude, and phase changing along temporal frames. Each frame has a set of partials, each having (at least) amplitude and frequency values (phase information might be discarded from the analysis). Each frame might also contain noise information, modeled as time-varying energy in the 25 critical bands of the analysis residual.

The ATS files start with a header at which their description is stored (such as frame rate, duration, number of sinusoidal trajectories, etc.).

After the header data, the time, amplitude, frequency, phase and residual (these two may or may not be present) data of each partial in each frame are stored as 64 bits double values.

The format of the ATS files can be found in [3] but it is important to keep in mind that, at present, the ATS files come in four types:

Type 1: only sinusoidal trajectories with amplitude and frequency data on file.

Type 2: only sinusoidal trajectories with amplitude, frequency and phase data on file.

Type 3: sinusoidal trajectories with amplitude, and frequency data as well as residual data on file.

Type 4: sinusoidal trajectories with amplitude, frequency and phase data as well as residual data on file.

In Figures 2 and 3, plots of the Deterministic and of the Residual parts of a steady, 440 Hz sound of a Flute can be seen[5].

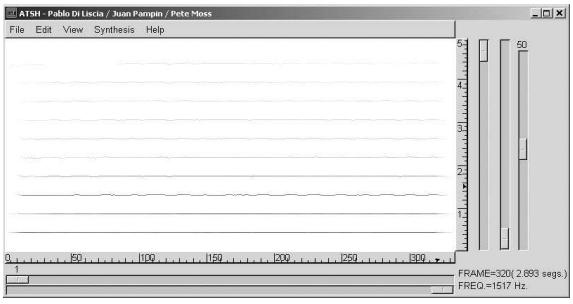


Figure 1: Plot of the Deterministic part of an *ATS* analysis.

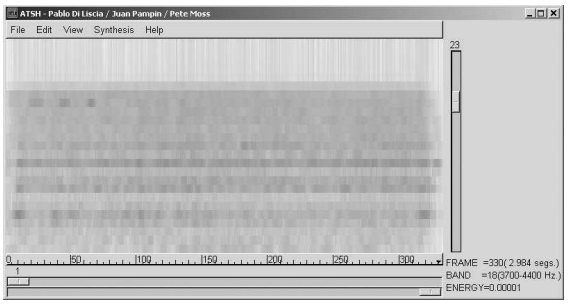


Figure 2: Plot of the Residual part of an *ATS* analysis.

2 *ATS* spectral data synthesis

At present, besides the original version in *CLM*, several *GNU* applications for *ATS* synthesis already exist. Several UGens for the *Csound* program were developed by Alex Norman [10]. Also, the *ATSH* program [3], allows the synthesis, editing and transformation of *ATS* data by means of a graphic interface. *SuperCollider* interfaces for *ATS* (including *classes* to read *ATS* files as well as UGens to do transformation and synthesis) are included in Josh Parmenter's UGen library, which is now part of the *SuperCollider* standard distribution.

2.1 *ATS* spectral data synthesis using *PD*

The synthesis procedure of an *ATS* analysis was designed in the following stages:

1-The *ATS* analysis data must be read from a file, parsed, and loaded in memory. The data of the file header (mainly the *ATS* file type, its duration, its frame rate, and the amount of partials per frame) must be also decoded and stored.

2-According to the *ATS* file type, the frequency, amplitude and phase (if any) and (if existing and required) the energy information of the residual part for each frame, must be sent at the right time (or with modifications that will warp the time of the original signal, if desired) to the synthesis units.

3-If the *ATS* file does not have residual data (types 1 or 2), using a bank with as many sinusoidal oscillators as partials, with their variable amplitude and frequency values for each frame properly interpolated would suffice to achieve the deterministic part synthesis.

4-If the *ATS* file has residual data (types 3 or 4), two special cases must be considered. The first case is when only the residual part is to be synthesized. The second case is when both, the deterministic and the residual part are to be synthesized.

In both cases, software units that produce random values in the range of -1 to 1 with periodicity *n*

by Itzvan Varga.

[5] These plots were obtained using the ATSH program (Di Liscia, Pampin and Moss) [3].

(frequency $f = 1/n$) interpolating them linearly, are used as sources to reconstruct the original noise. As stated in [9][6], such units (which will be termed from here on *randi*[7]) produce a signal whose spectrum has a main lobe at 0 *Hz* and (*Nyquist_frequency*/*f* - *1*) lobes starting at (*f* + *f* / *2*), and spaced by frequency intervals of *f Hz*. The amplitude peak of the second lobe is approximately 24 *dB* below the amplitude peak of the main lobe, and the remaining lobes decrease even more in amplitude as their frequency rises. If such signal is multiplied by a sinusoidal signal of frequency *fc*, its main lobe will be centered in this frequency due to the convolution of the spectra of both signals, and the result is very similar to band-pass filtered noise with center frequency at *cf.* A plot of one of such signals is shown in Figure 3.

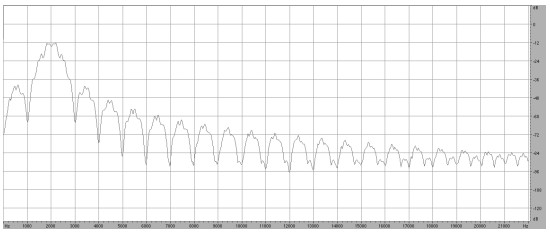


Figure 3: a plot of the spectrum of the output of a 1000 *Hz randi* unit convolved by a sine wave at 2000 *Hz* and sampled at 44.1 *KHz* A *FFT* of 1024 samples with a *Blackman-Harris* smoothing window was used in the analysis.

In the first case (only residual synthesis), the output signal is the sum of the output signals of 25 *randi* units with their frequencies set as the bandwidth of each one of the critical bands, each one modulated by a sinusoidal signal with its frequency set as the center frequency of the corresponding critical band. The amplitudes of each one of the resulting 25 signals are scaled by the *RMS* power values obtained in the residual analysis for each critical band and frame . The latter *RMS* values for each frame are linearly interpolated.

The second case is somewhat more complex. The residual energy values for each critical band must be ´re-injected´ into each deterministic sinusoidal trajectory according to the critical band where each one of these trajectories is located. Since this information is not stored in the *ATS* files, and to save computation time, this must be achieved before the real-time synthesis process. A function that evaluates the locations in the 25 critical bands of each frequency value for all the sinusoidal trajectories is used to compute and store in memory variable energy values of noise for each partial at each frame. The resulting signal for the synthesis of both, the deterministic part and the residual part of each partial, *k*, with frequency f_k, amplitude $a_k$and noise energy r_k, is made of the sum of the deterministic part (synthesized according to the procedure described in stage 3) plus the deterministic part (with changing frequency f_k, but scaled by the *RMS* values of noise for each partial, r_k, instead of the deterministic amplitude a_k), modulated by noise (produced by *randi* units with variable frequency proportional to f_k). The following formula shows the computing of a sample, *n*, of the output signal[8]:

$$output_n = \sum_{k=0}^{par-1} \left(a_k \sin(2\pi f_k n/R) + r_k \sin(2\pi f_k n/R)\, randi(f_k s)\right)$$

Where *par* is the number of partials (sinusoidal trajectories), *s* is a scaling factor and *R* is the Sampling Rate. The scaling factor *s* should be less than 1, because the resulting frequency values ($f_k s$) will determine the rate at which the *randi* units get a new random value and, essentially, the bandwidth of the resonant peaks of the output signal spectra[9].

The following sections will explain how several *Pure Data* externals were created and/or modified to properly connect them in order to perform the synthesis according to the stages and techniques above mentioned. It is assumed that the reader is familiar with the fundamentals of *Pure Data* programming, at the very least. The source code of all the externals, as well as some *ATS* files, and several *Pure Data* example patches can be found at [1].

2.2 Reading the data: The *atsread* external

The *atsread* external (by Alex Norman [7]), for *Pure Data*, was programmed to read *ATS* files and

[6] See pages 205-207.

[7] Just because the *randi* UGen of the *Csound* program generates this kind of signals.

[8] The deterministic part frequency and amplitude values (f_k, a_k) as well as the *RMS* power of the residual part values (r_k) must be obtained by interpolating the values of the actual frame and the next one according to the desired time.

[9] The bandwidth of the main lobe of the spectra will be of $2f_k s$ *Hz* A scaling factor of 0.1 (10% of f_k) seems to work fairly well for this application, but even a non- uniform scaling could be applied in order to get 'optimal' bandwidth values as a function of frequencies f_k.

to send its data to the required synthesis units. Essentially, this object takes an *ATS* file, parses its data and stores them in memory and allows sending them to the required synthesis units. The object expects a succession of *float*s indicating the time of the analysis that is required to be synthesized. According to the time value received, the frame of the analysis where it is located is fetched, and its corresponding data values are obtained by linear interpolation of the data of the current and the next frames, and sent through the respective *outlets*.

The author of this paper made several improvements to the *atsread* external in order to adhere to the needs of the synthesis units that are to be further used. The modifications are listed below and briefly explained, but their relevance will become more understandable in the next sections, where the synthesis units are explained.

1-Inclusion of an extra *outlet* for the output of the header data of the *ATS* file. As previously mentioned, knowledge of the type (whether residual data is present or not) and of the duration and number of partials of the opened file, at least, is needed in order to properly set the corresponding synthesis units and to send to them the time data. The header data is sent in the form of a *list* of *floats* once a file is opened.

2-Inclusion of a function to compute the residual noise values corresponding to each partial, for the case were residual information is present and both, deterministic and residual synthesis, are required[10]. If the file has residual data, then this function is called once it is opened, and the resulting values for each partial at each frame are stored in memory.

3-Inclusion of an extra *outlet* for the output of the noise values for each partial computed in 2.

4-Inclusion of an extra *outlet* for the output of the index (i.e., the partial number). The partial number if sent as a *float* value before its amplitude, frequency and noise (if any) data are sent.

5-Modification of the output format of the amplitude, frequency, phase and noise values. These are now sent as independent float values (with their partial index preceding them), instead of as *lists* of *float* values.

The modifications 4 and 5 were meant to simplify the connections with the synthesis units.

[10] Such *C* language function (*band_energy_to_res*) was taken from the analysis engine code of the program *ATSA* (by Pampin, Di Liscia and Moss).

2.3 Sinthesizing the deterministic part: The *oscbank~* external

If only the synthesis of the deterministic part is required, the *atsread* external may be connected with the *oscbank~* external (by Richie Eakin, [4]). This external is designed to perform additive synthesis and produces its output audio signal through the use of a *lookup oscillator bank*. For better performance, the oscillator units are not interpolating, but as the default (sine) table length used is large enough (65536 values), the artifacts of truncation are minimized[11]. The *oscbank~* external takes *control rate* successions of three *floats* (each one being respectively the oscillator number, its frequency and its amplitude). The received values are used to control the output of the corresponding oscillator and are interpolated linearly at a rate that can be set by the user. The default table and its size are appropriate for the purposes of the ATS deterministic part synthesis, but the user may experiment with other wave shapes and table sizes[12].

2.4 Sinthesizing the residual part: The *ats-noisy~* external

If only the synthesis of the residual part is required, the *atsread* external may be connected with the *ats-noisy~* external (by Pablo Di Liscia). The output audio signal of this *external* is obtained adding the outputs of 25 *randi* units, each one with its frequency adjusted to the bandwidth of each critical band and modulated by a sine wave whose frequency is adjusted by the central frequency of each critical band. The output of each *randi* unit is scaled by the *RMS* power of the residual part of each critical band. These 25 *RMS* values for each analysis frame are received from the *atsread* external in the form of a *float list* and linearly interpolated. The interpolation rate may be set by the user.

2.5 Sinthesizing the deterministic and the residual parts: The *ats-sinnoi~* external

If the synthesis of both, the residual and the deterministic parts is required, the *atsread* external may be connected with the *ats-sinnoi~* external (by Richie Eakin and Pablo Di Liscia).

[11] As stated by Moore ([9], pp. 166), a truncating oscillator reading a table of 65536 values, produces a *signal-to-error-noise ratio* of approximately 85 *dB*.

[12] Possibly obtaining strange, but musically interesting, results!

This external was programmed modifying the *oscbank~* external (by Richie Eakin).
Basically, the above mentioned modifications include the addition of a bank with as many *randi* units as deterministic partials are to be synthesized, plus an *inlet* to retrieve the residual RMS power data for each partial, which must be sent by *atsread* as a succession of *float* values. As explained in section 2.1, in this case the corresponding noise of the residual part must be re-injected in each deterministic trajectory. This is computed previously by the *atsread* external once a file having both residual and deterministic part is loaded. The residual part, in this case, is synthesized using *randi* units as sources, but the output of each one is multiplied by the output of each deterministic trajectory and scaled by the RMS of the noise computed for each partial. The deterministic part is synthesized as explained in section 2.3. An extra *audio outlet* was added to the external as well, in order to have individual outputs of both, the residual and the deterministic parts. This allows the user to further mix them with individual amplitude scaling, if desired. The user may synthesize only the deterministic part with this external as well, but doing so with the *oscbank~* external (as explained in section 2.3) will be much less CPU consuming.

3 Conclusions

The synthesized signals of all the presented units were found of similar quality of the ones produced by the synthesis units listed in the beginning of Section 2. Other strategies for synthesizing the residual part such as, for instance, a bank of *band-pass* filters processing *white noise* sources, could be used as well. However, the method described in this paper seems to provide a better 'blending' of the residual and deterministic signals with the additional benefit of being less CPU consuming.
The example Patches that were developed are simple, and only deal with the synthesis of all the partials with modifications in the duration of the output signal and its frequency. However, the examples suggest that, using the *Pure Data* toolkit that was presented in this paper, the user skilled in *Pure Data* programming may achieve very interesting transformations in a relatively straight forward way.
Future research will be focused in the development of an analysis external to perform high level analysis of the *ATS* data, with the purpose of allowing real time transformations of timbre and spectral morphing between different sounds.

4 Acknowledgements

The author thanks the *Universidad Nacional de Quilmes* (*UNQ*, Buenos Aires, Argentina) for supporting and hosting this research, to Juan Pampin (the developer of the *ATS* technique) for his advice, and to Alex Norman and Richie Eakin, the authors of the *Pure Data* externals that were taken as the basis for this development.

References

[1] Oscar Pablo Di Liscia. 2013. *PD-ATS Toolkit*. https://puredata.info/Members/pdiliscia/ats-pd

[2] Oscar Pablo Di Liscia and Juan Pampin. 2002: *ATSH Manual*, http://musica.unq.edu.ar/personales/odiliscia/software/ATSH-doc.htm

[3] Oscar Pablo Di Liscia and Juan Pampin. 2003. Spectral analysis based synthesis and transformation of digital sound: the ATSH program. *Proceedings of the IX Brazilian Symposium of Computer Music*, NUCOM, Minas Gerais, Brasil.

[4] Richie Eakin. 2007. *oscbank~*. https://github.com/pdl2ork/pd/tree/master/externals/oscbank~

[5] Juan Pampin. 1999. ATS: a Lisp environment for Spectral Modeling. *Proceedings of the International Computer Music Conference*, Beijin.

[6] Juan Pampin. 2011. *ATS_theory*, http://wiki.dxarts.washington.edu/groups/general/wiki/39f07/attachments/55bd6/ATS_theory.pdf

[7] Juan Pampin, Oscar Pablo Di Liscia, Pete Moss and Alex Norman. 2004. ATS user Interfaces. *Proceedings of the International Computer Music Conference*, Miami University, USA.

[8] Itzvan Varga, *ATSANAL documentation*, http://www.csounds.com/manual/html/UtilityAtsa.html

[9] F. Richard Moore.1990. *Elements of Computer Music.* Prentice-Hall., New Jersey, USA.

[10] Alex Norman. 2004. *Csound ATS spectral processing UGens*,

http://www.csounds.com/manual/html/SpectralATS.html

[11] Xavier Serra and Julius O. Smith III. 1990. A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition, *Computer Music Journal*, Vol.14 #4, MIT Press, USA.

[12] Ernst Zwiker and Hugo Fastl. 1990. *Psychoacoustics Facts and Models*. Springer, Berlin, Heidelberg.

Multi-Channel Noise/Echo Reduction in PulseAudio on Embedded Linux

Karl FREIBERGER and **Stefan HUBER** and **Peter MEERWALD**
bct electronic GesmbH
Saalachstraße 88
A-5020 Salzburg, Austria
{k.freiberger, s.huber, p.meerwald}@bct-electronic.com

Abstract

Ambient noise and acoustic echo reduction are indispensable signal processing steps in a hands-free audio communication system. Taking the signals from multiple microphones into account can help to more effectively reduce disturbing noise and echo. This paper outlines the design and implementation of a multi-channel noise reduction and echo cancellation module integrated in the PulseAudio sound system. We discuss requirements, trade-offs and results obtained from an embedded Linux platform.

Keywords

Acoustic echo cancellation, noise reduction, hands-free telephony, beamforming, performance

1 Introduction

At bct electronic, we develop a speakerphone for hands-free Voice over Internet Protocol (VoIP) telephony and intercom. On our communication device, we run a custom embedded Linux system created with OpenBricks[1]. The device is designed for desktop or wall-mount use, has a 7" touch-screen and is powered by a TI OMAP3 processor (DM3730). Two independent hardware audio codecs enable hands-free communication as well as hand-set or headset use at the same time in order to support flexible intercom and VoIP scenarios.

Speech quality is a very important criterion for us. Therefore, our device is equipped with a 4-channel array of digital, omnidirectional MEMS microphones[2]. This allows to reduce noise without distorting the desired speech signal [Souden et al., 2010]. However, elaborate digital signal processing (DSP) is required to achieve good speech quality in challenging acoustic environments with high levels of ambient noise.

Several open-source software components are available in our application area: SIP stacks (Linphone[3], Sophia SIP[4]), audio compression codecs (G722, Opus[5]), sound servers (JACK [Davis, 2003], PulseAudio[6]), DSP primitives for resampling and preprocessing (Speex[7]), to give a few examples. Open-source SIP software has gained support for single-channel acoustic echo and noise reduction (AENR) recently. However, we are not aware of an open-source framework for multi-channel audio communication and AENR.

In section 2 we describe the acoustic setting and the related challenges in AENR. The basic principles behind common methods are explained. In section 3 we motivate the use of PulseAudio as a sound server and integrating component of our software architecture and outline the design and implementation of a multi-channel AENR plug-in module. While we cannot release the DSP code at this point, several improvements to PulseAudio have been made available to enable multi-channel audio processing on embedded Linux platforms. Section 4 outlines algorithms for multichannel AENR. We have prototyped the algorithms in MATLAB and Octave on the PC, transcribed the code to C/C++, and successively adapted and optimized the code to target the ARMv7 platform. Runtime performance analysis and optimization techniques are discussed in section 5. The test setup and experimental results are detailed in section 6. Finally, Section 7 summarizes results and outlines further work.

2 Acoustic Echo and Noise Reduction

The acoustic front-end of a basic speakerphone comprises a microphone for picking up the near-end speaker (NES) and a loudspeaker for play-

[1] http://www.openbricks.org
[2] http://mobiledevdesign.com/tutorials/mems-microphones
[3] http://www.linphone.org
[4] http://sofia-sip.sourceforge.net
[5] http://www.opus-codec.org
[6] http://www.pulseaudio.org
[7] http://www.speex.org

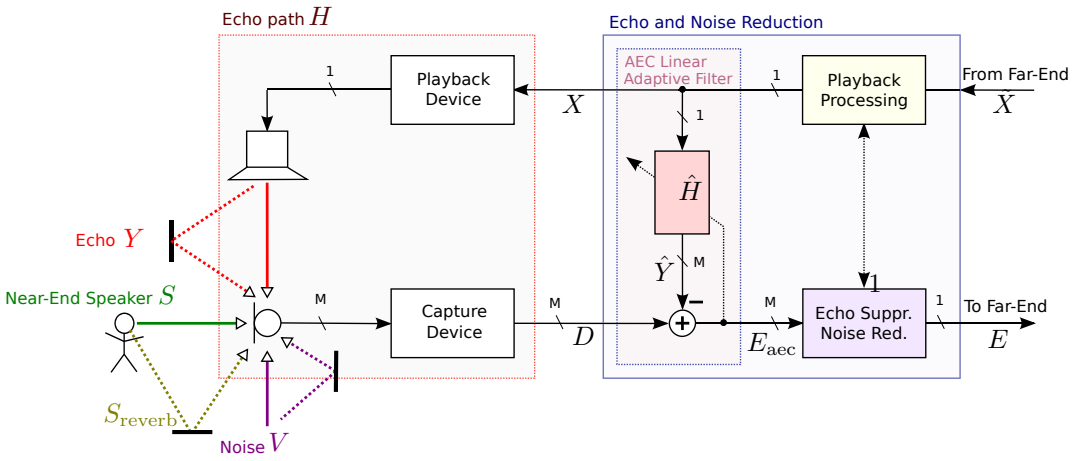


Figure 1: Near-end acoustic setting and general AENR system for one loudspeaker and M microphone channels. For $M > 1$ the echo suppression & noise reduction module may include beamforming. The dashed, colored lines indicate room reflections.

ing back the far-end speaker (FES), see Fig. 1. In practice, the captured microphone signal D does not only contain the desired NES signal S but also undesired components that degrade speech intelligibility, namely room reverberation S_{reverb}, the so-called *echo signal* Y and an additive noise signal V:

$$D = S + S_{\text{reverb}} + Y + V \qquad (1)$$

Here, S_{reverb}, Y and V are mutually uncorrelated, S_{reverb} is correlated (only) with S and Y is correlated only with the playback signal X, containing the FES. V denotes all other unwanted parts neither correlated with S nor X. The challenge is to remove or at least reduce the undesired components without (too much) distortion of S.

2.1 Acoustic Echo

The echo signal can be written as $Y = H\{X\}$, where $H\{\cdot\}$ denotes the echo path system consisting of playback device, loudspeaker, room, microphone and capture device. The term "echo signal" stems from the fact that Y is contained in D and, thereby, a delayed and filtered version of the FES signal X is sent back to the far-end. It follows that if the near-end device has an insufficient echo-reduction system, an echo becomes obvious on the far-end. The larger the delay of the echo, the more irritating is the echo of a given level, cf. [Hänsler and Schmidt, 2004]. The overall delay of the echo signal consists of delays due to capture and playback, the acoustic path, the speech codec and VoIP transmission. Because of the limited physical size of a speakerphone, the loudspeaker is located close to the microphone. The level of the echo might hence be several times higher than that of the NES. This makes high quality echo cancellation and/or suppression indispensable.

The terms cancellation and suppression — they are subsumed under the term *reduction* in this paper — shall not be confused: The idea behind echo cancellation is to find an estimate $\hat{Y}$ of the echo and subtract it from the microphone signal, i.e., $E_{\text{aec}} = D - \hat{Y}$, with $\hat{Y} = \hat{H}\{X\}$. By inserting D from Eq. (1), one can see that Y can be fully removed without distorting S if Y equals $\hat{Y}$. Most practical systems use a linear adaptive filter with finite impulse response (FIR) to identify and model the echo path H. Nonlinear models exist, but are in less widespread use due to their higher complexity and slower convergence.

In practice, there are several reasons why the adaptive filter does not fully cancel the echo and a residual echo (RE) Y_{res} remains in E_{aec}: The adaptive FIR filter (i) does not model the nonlinearity of the loudspeaker or a potential clipping of the echo signal, (ii) is too short to model the echo path impulse response $h(t)$, (iii) is too slow to follow changes of the echo path, and (iv) does not fully converge or even diverge due to double talk. As a consequence, E_{aec} is usually further processed by a RE suppression postfilter. The principle of suppression is to apply a real gain factor $G(l, f)$ to the input of the suppression filter. Because echo suppression is typically performed in the frequency domain or subbands of a filterbank, the indices l and f are introduced to indicate the time-

and frequency-dependence, respectively. If D is directly plugged into a suppression filter, we have $E_{\text{suppr}}(l, f) = D(l, f) \cdot G(l, f)$. Looking at Eq. (1), we see that suppression of echo or noise goes along with suppression of the NES S. Because Y and S do typically not fully overlap in the time-frequency plane, duplex communication is possible at least to some extend.

2.2 Ambient Noise

In our application, the NES shall be able to move freely around the device and still be picked up flawlessly, even when being several meters away from the microphone and having a low level. Therefore, the microphone must be very sensitive and/or highly amplified. As a consequence, we face high levels of ambient noise, e.g., fan noise in an office, traffic noise, as well as the acoustic echo described above. Reverberation and the self-noise of the microphone must also be taken into account.

In the single microphone case, noise reduction (NR) is based on the suppression principle. To compute the suppression filter $G_{\text{noise}}(l, f)$, the power spectral density (PSD) of the noise must be estimated. This can be done in speaking pauses, i.e., when $S = 0$ is detected by voice activity detection. Today, more advanced statistical methods are typically used [Hänsler and Schmidt, 2004]. These allow for updating the noise estimator even in times when both V and S are active. Still, single channel NR delivers best results if the noise is stationary, i.e., the noise PSD does not change much over time. Otherwise, the PSD estimation is likely to be inaccurate, which may cause unnatural artifacts in the residual noise and speech. Typically, strong single channel noise reduction comes at the cost of speech distortion. However, it is theoretically possible to perform single channel NR without speech distortion [Huang and Benesty, 2012].

By using more than one microphone, we can not only exploit time-frequency information but also spatial information. This allows for improved NR, which is discussed in section 4. At this point we note that the cancellation principle can also be applied to NR if a reference of the noise signal is available. In section 4 we explain how a so called blocking matrix can provide a noise reference in adaptive beamforming.

3 Echo Cancelling in PulseAudio

Over the last years, several widely-used desktop Linux distributions adopted PulseAudio [Poettering, 2010] as the default sound system. More recently, PulseAudio became an option to enable software audio routing and mixing in embedded Linux handheld devices [Sarha, 2009], competing with AudioFlinger on Android. An alternative sound server, JACK [Davis, 2003; Phillips, 2006], is predominantly used for professional, low-latency audio production.

PulseAudio is the software layer that controls the audio hardware exposed via the ALSA interface by the Linux kernel. Towards the application layer, PulseAudio offers to connect multiple audio streams to the actual hardware, providing services such as mixing, per-application volume controls, sample format conversion, resampling, et cetera. This allows concurrent use of the audio resources and matches the requirements of the application layer. An important service for hands-free telecommunication systems is acoustic echo and noise reduction (AENR). Since version 1.0, PulseAudio furnishes an echo cancellation framework as a pluggable module. In PA's terms, the echo cancellation (EC) module subsumes AENR. The actual AENR implementations (AENRI) are provided by the Speex library and Andre Adrian's code[8]. With version 2.0, the WebRTC[9] AENRI was introduced and became PulseAudio's default.

The decisive advantage of the sound server architecture is that the responsibility for AENR can be separated from the VoIP application, permitting reuse of the AENR resources by multiple software components and saving duplicate development effort. Furthermore, hardware constraints are hidden from the application: While the audio hardware may only handle interleaved stereo samples in 16-bit signed integers with 48 KHz, the application is actually interested in a mono audio stream represented by single-precision floating-point data sampled at 16 KHz.

So far, the PulseAudio echo-cancellation framework was limited to a symmetric number of channels entering and leaving the AENRI, typically a mono audio stream. However, in an audio setup with an array of microphones, a multi-channel audio stream is processed by the AENRI and generally reduced to mono output, see Fig. 2. The AENRI signal processing pipeline may choose to incorporate sample rate adaption as well, leading to an additional asymmetry of sample data entering and exiting the

[8] `http://www.andreadrian.de/intercom/`

[9] `http://www.webrtc.org`

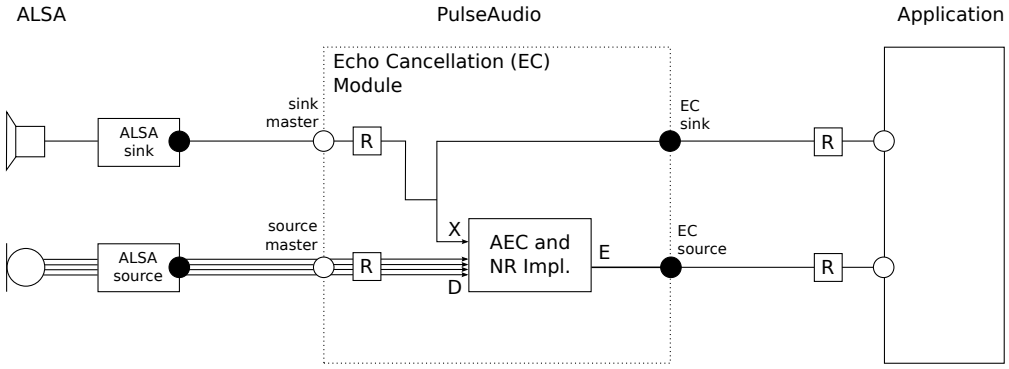


Figure 2: Overview of the PulseAudio sound system providing acoustic echo and noise reduction (AENR) service to an application (with 4 microphone channels).

EC module. A number of patches addressing this issue and related limitations have been submitted during the PulseAudio version 4.0 development cycle.

Fig. 2 shows the PulseAudio sound server in between the ALSA sink/source and the application. Instead of directly connecting to the ALSA sink/source, the application binds to the EC sink/source. Note that the EC module specifies its internal audio sample format and rate, hence resampling stages (denoted by R) may become necessary. Resampling, in PulseAudio's terms, includes sample format conversion, channel remapping, and sample rate conversion as necessary. The modular sound server design brings great flexibility, but efficient implementation of the resampling stages becomes paramount, especially if microphones, AENRI and application layer depend on different sample specifications.

4 Multi-Channel Audio Processing

A multi-channel noise reduction system optimal in the minimum mean square error sense can be factorized in a linearly constrained minimum variance (LCMV) beamformer followed by a single channel postfilter [Wolff and Buck, 2010]. The postfilter is essentially a noise suppressor as explained in chapter 2. Echo suppression can be efficiently combined with noise suppression [Gustafsson et al., 2002].

A beamformer is a spatial filter, i.e., a beam is steered towards a target direction, whereas other directions are suppressed. The basic operation behind linear beamforming is to filter-and-sum the M input signals, i.e., the output F of a filter-and-sum beamformer (FSB) $\boldsymbol{W}$ is

$$F(l,f) = \sum_{m=0}^{M-1} W_m(l,f) D_m(l,f) \qquad (2)$$

where m is the microphone index and $W_m(l,f)$ is the filter weight for the m-th microphone.

A fixed beamformer (FBF) uses fixed weights $\boldsymbol{W}$, that can be precomputed, whereas an adaptive beamformer adapts the weights $W_m(l,f)$ in dependence of the current noise field. The most basic FBF is the delay-sum beamformer (DSB), where $\boldsymbol{W}$ implements pure, frequency independent time delays. The idea is to time-align signals from the target direction. Signals from other directions are to some extent out of phase and cancel partially because of the summation. The DSB exhibits a broad mainlobe of the beampattern at low frequencies and a very narrow mainlobe at high frequencies, i.e., at low frequencies it cannot reduce much noise, whereas at high frequencies little deviation from the target direction causes strong attenuation, leading to a low-pass filtered sound in practical conditions with steering errors. Using filter optimization strategies, better low-end suppression and a wider mainlobe at high frequencies can be achieved [Tashev, 2009]. A FBF can however only be optimal for a certain, given noise-field.

Adaptive beamformers can adapt to changing noise fields and can hence achieve more noise reduction. Still, it is possible to set linear constraints, like distortion-less operation towards the target direction. It can be shown that an adaptive LCMV beamformer can be implemented in the Generalized Sidelobe Canceller (GSC) form that transforms the constrained optimization in an unconstrained one [Souden et al., 2010]. Though formally the same, the GSC has advantages in the implementation and provides an intuitive access to the adaptive beamforming problem, cf. Fig. 3.

The noisy M-channel input is processed by

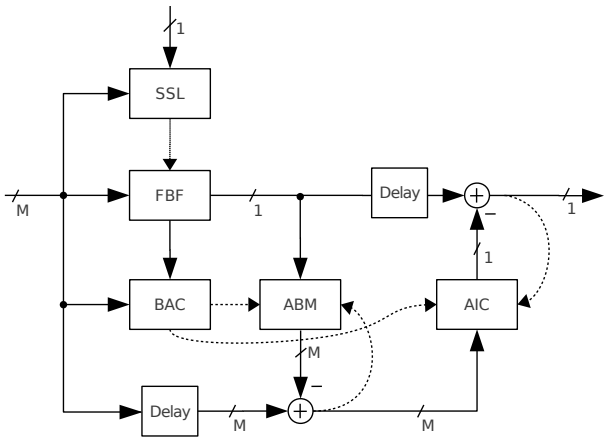


Figure 3: Structure of a Generalized Sidelobe Canceler (GSC) beamformer.

an FBF that keeps the distortion-less constraint towards the target direction. The output of the FBF is further enhanced by subtracting the output of an adaptive interference canceller (AIC). The AIC should be fed with noise-only signals. To this end, the adaptive blocking matrix (ABM) subtracts the target from the noisy microphone signals. The purpose of the beamformer adaption control (BAC) is to guarantee that the AIC is adapted in times of noise-only, whereas the ABM should only be adapted in times of high SNR. The delays are necessary to ensure causality. The FBF needs the target direction as a control input. If the target direction cannot be set to a fixed value, a sound source localization (SSL) algorithm can be used to track the source of interest. SSL is typically based on estimating the direction-dependent time delay of arrival between the individual microphones. In [Souden et al., 2010] a formulation of the GSC is stated, that does not require knowledge of the target direction or the microphone locations, but only the source and noise statistics. This shows the strong link between adaptive beamforming and linear blind source separation.

Our current multichannel AENR system contains a self-steered adaptive beamformer and a postfilter. The latter performs combined echo and noise suppression. A dedicated AEC module has also been developed, but is not yet implemented in C. Combining an AEC with adaptive beamforming promises synergy effects [Herbordt and Kellermann, 2002], i.e., the beamformer can assist the AEC during adaption. Once the AEC is adapted, the beamformer can focus on reducing interfering noise. All processing steps can be done in the frequency domain (FD). To transform a time domain signal block to FD and back, we use the forward and inverse Fast Fourier Transform (FFT), respectively.

```
float v = *(src++) * (1 << 15);
  // load 4 floats from src, increment pointer
  vld1.32      {q0}, [%[src]]!
  // scale by q1 (= 32767)
  vmul.f32     q0, q0, q1

*dst++ = CLAMP(lrintf(v), -0x8000 , 0x7FFF );
  // convert float to 16:16 fixed-point
  vcvt.s32.f32 q0, q0, #16
  // shift right, round, narrow to 16 bit
  // with saturation
  vqrshrn.s32  d0, q0, #16
  // store 4 int16 values, increment pointer
  vst1.16      {d0}, [%[dst]]!
```

Listing 1: Using ARM NEON to convert float to 16-bit integer samples with saturation.

5 Targeting an embedded ARMv7 Cortex-A8 platform

To realize the actual VoIP/intercom application, we build upon the Linux kernel and ALSA for hardware handling and the PulseAudio sound server. Here, we focus on the intermediate component. In order to integrate hardware, AENRI and application, PulseAudio must mediate sample format, sample rate and number of channels at substantial runtime costs. Besides the AENRI, sample rate adaptation is expensive.

The ARMv7 processor architecture is quite power-efficient, yet offers significantly less computational resources than current desktop computers. Especially the Cortex-A8 has weak single-precision floating-point (FP) performance (i.e., most FP instructions take multiple cycles) and requires SIMD-type instructions named NEON [Anderson, 2011] for best performance. Later CPU designs (e.g., Cortex-A9/A15) have improved FP units and performance is less dependent on NEON optimizations. Algorithms in fixed-point arithmetic are more tedious to develop and often have less numeric precision. Hence, we decided to implement all audio signal processing in floating-point arithmetic. On the OMAP3 processor, single-precision FP NEON operations are often executed in a single cycle and are not necessarily slower than equivalent fixed-point/integer instructions.

Resampling is provided by the Speex library

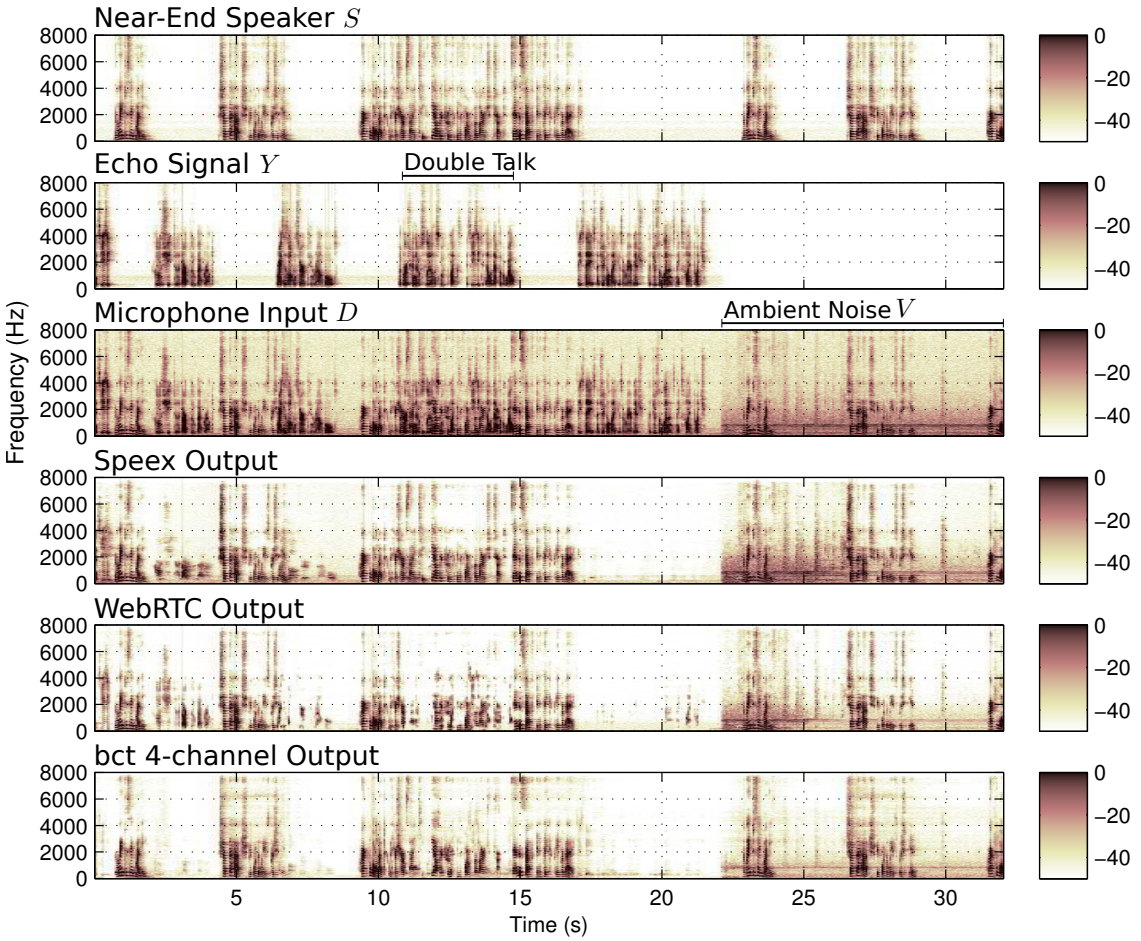


Figure 4: Spectrograms of the test audio signal (top three plots) at 16 KHz and the corresponding output signals (bottom three plots).

for which an ARM NEON patch is available[10]. On the target CPU, the FP implementation is more efficient than fixed-point. Typically, the AENR will be implemented in the frequency domain (FD). To this end, the libav project[11] provides a fast ARM NEON FFT-implementation[12] with a public interface. Listing 1 illustrates how ARM NEON instructions can be used to exploit data parallelism. For the float to 16-bit integer sample conversion operation shown, a speedup of 11× is achieved primarily due to the implicit saturation.

The overall runtime requirements of PulseAudio on the target platform depend on the signal-processing implementation, but to a large part also on the audio latency requirements (set to 50 ms). We observe approximately 25 % CPU load due to PulseAudio providing 4-channel AENR at 16 KHz. Profiling has been performed using the Linux `perf` tool.

6 Test Setup and Experimental Results

Assessing AENR systems is a broad and controversial topic. In our experience, metrics that access speech quality [Loizou, 2011] are often not well suited to describe the behavior and artifacts that occur in complex, real world scenarios. In this work, we rely on spectrogram plots to make an exemplary comparison of different algorithms in a complex scenario with double-talk and noise. We do however believe that listening tests are crucial and need to complement any numerical results.

In order to benchmark the different pluggable AENRIs, PulseAudio's `echo-cancel-test` program is used: it reads raw audio data from a play (denoted signal X) and record (signal D) file and outputs the processed audio data (signal E). All experiments have been performed

[10] `http://blog.gmane.org/gmane.comp.audio.compression.speex.devel/month=20110901`

[11] `http://libav.org`

[12] See `http://pmeerw.net/blog/programming/arm_fft.html` for an informal comparison.

at a sample rate of 16 KHz with PulseAudio 3.0 on a Linux operating system. The GNU compiler in version 4.6 has been invoked with the options `-O2 -ffast-math`. The flags `-march=core2` and `-march=armv7 -mfpu=neon -mfloat-abi=softfp` were used for the x86 64-bit and ARM 32-bit target, respectively.

6.1 Audio Quality

The spectrogram plots in Fig. 4 depict the audio energy in different frequency bands over time (32 seconds; horizontal axis). The audio signals[13] shown are near-end speaker (S), echo signal (Y), microphone input (D) and the output of three AENRIs (Speex, WebRTC, BCT4CH). The Adrian AEC, turned out to not be competitive and completely diverged during double-talk. Therefore, we chose to not devote space to it in our plots.

S and Y are obtained by convolution of speech signals with measured impulse responses H_S and H_X of our device/microphone array in a medium-sized office room. In Fig. 4, only the first channel $m = 0$ (farthest from the loudspeaker) is shown. This channel is also used as an input for the single channel AENRIs Speex and WebRTC. The Cartesian coordinates of the location of microphone m are $\vec{p}_m = [0, p_{m,y}, 0]$, with $p_0 = -0.12$, $p_1 = -0.03$, $p_2 = 0.03$, $p_3 = 0.12$. For measuring H_S, a loudspeaker was placed at $\vec{p}_S \approx [0.5, 0, 0.25]$. We used the exponential sweep method to compute the impulse responses [Holters et al., 2009]. H_X is obtained with the integrated loudspeaker having its acoustic center at $\vec{p}_X \approx [0, 0.1, 0.1]$. In Fig. 4 clearly discernible, alternating speech segments including a period of double talk starting after about 11 seconds can be seen. Before second 22 a recording of the "quiet" office room has been added. After second 22, a broadband ambient noise signal – a recording of a ventilator, placed at $\vec{p}_v \approx [-1.5, 0.3, 0.5]$ – is added to S to compare the noise reduction capabilities of the tested AENRIs. The added noise recordings include the self-noise of the microphones.

Observing the outputs, the echo signal is only partially attenuated in the Speex and WebRTC results during the adaptation (learning) period in the beginning. BCT4CH however delivers echo reduction right from the start and provides good double talk performance. Once adapted, Speex delivers very good double talk performance. This can probably be attributed to its advanced AEC learning rate adjustment[Valin, 2007]. WebRTC, on the other hand, suppresses large portions of the high frequency content of S. Furthermore, WebRTC retains audible echo, see e.g. second 20–22. In other, practical situations WebRTC might however still be preferred to the Speex AENRI, because it employs a more rigorous echo suppression and loss/gain control, which works as a safety guard if nonlinearities or sudden changes of the echo path occur and AEC fails. As outlined in Section 4 BCT4CH does currently not contain an actual AEC module. Knowing this, our good echo reduction performance is even more remarkable. It stems from the superb interference suppression capability of our adaptive beamformer and our high quality postfilter.

Taking a look at the ambient noise scenario at second 22–32 in Fig. 4, all methods are able to reduce noise, however Speex and WebRTC require some time to initially adapt to the new noise characteristics. This clearly show the benefit of the microphone array processing that is less dependent on a stationary noise PSD estimate.

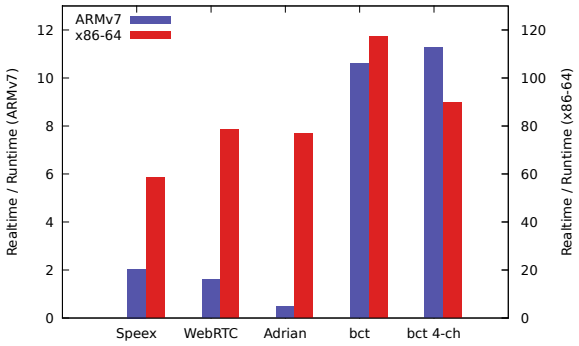


Figure 5: Comparing realtime vs. runtime of several AEC plugins on x86-64 and ARMv7 (higher results are better).

6.2 Runtime

In Fig. 5 we compare the runtime of different AENRIs on an ARMv7 Cortex-A8 platform (TI OMAP3 processor, DM3730, clocked at 800 MHz) and a x86-64 platform (Intel i7-870 clocked at 3 GHz) relative to realtime. Not surprisingly, the embedded platform turns out to be more than 10 times slower than the PC platform. BCT and BCT4CH refer to a single-channel and multi-channel implementation developed by bct electronic. The BCT and BCT4CH code has been optimized and implemented using

[13] Available at `http://bct-electronic.com/lac13/`.

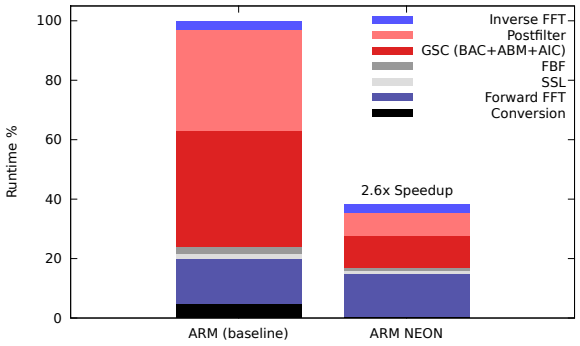


Figure 6: Runtime breakdown and ARM NEON optimization result of the BCT4CH implementation.

the ARM NEON instruction set; they consume approximately 10 % CPU. The other ARMv7 AENRIs lacking optimization compare less favorable with the Intel platform.

Fig. 6 breaks down the runtime of the BCT4CH AENRI according to the processing structure outlined in Section 4. Straightforward optimization of the C/C++ code yields an overall speedup of 2.6×. The runtime contribution in % of the total ARM execution time can be observed: postfilter and GSC are the most expensive execution blocks. The performance of the FFT is not improved as baseline and optimized code both depend on the external libav FFT implementation.

7 Conclusions

We have presented first results of a multi-channel noise/echo reduction solution built on top of PulseAudio and motivated the design decisions. The work has resulted in a number of improvements in the PulseAudio echo cancellation and signal-processing framework, which have been contributed during the version 3.0/4.0 development cycle and should facilitate future embedded Linux audio solutions. Further work includes optimizing code for audio stream mixing, more efficient resampling methods, and the implementation of an efficient AEC in the multi-channel processing pipeline.

References

M. Anderson. 2011. ARM NEON instruction set and why you should care. In *Embedded Linux Conf.'11*, San Francisco, CA, USA.

P. Davis. 2003. The JACK audio connection kit. In *Proc. Linux Audio Conference, LAC'03*, Karlsruhe, Germany.

S. Gustafsson, R. Martin, P. Jax, and P. Vary. 2002. A psychoacoustic approach to combined acoustic echo cancellation and noise reduction. *IEEE Trans. on Speech and Audio Processing*, 10(5):245–256.

E. Hänsler and G. Schmidt. 2004. *Acoustic Echo and Noise Control: A Practical Approach*. Wiley, New York.

W. Herbordt and W. Kellermann. 2002. Frequency-domain integration of acoustic echo cancellation and a generalized sidelobe canceller with improved robustness. *Europ. Trans. on Telecomm.*, 13(2):123–132.

M. Holters, T. Corbach, and U. Zölzer. 2009. Impulse response measurement techniques and their applicability in the real world. In *Proc. 12th Int. Conference on Digital Audio Effects, DAFx'09*, Como, Italy.

Y.A. Huang and J. Benesty. 2012. A multi-frame approach to the frequency-domain single-channel noise reduction problem. *IEEE Trans. on Audio, Speech & Language Processing*, 20(4):1256–1269.

P. Loizou, 2011. *Speech quality assessment*, volume 346, pages 623–654. Springer Verlagq.

D. Phillips. 2006. Knowing Jack. *Linux Magazine*, (67).

Lennart Poettering. 2010. Pro audio is easy, consumer audio is hard. In *Proc. Linux Audio Conference, LAC'10*, Utrecht, Netherlands.

J. Sarha. 2009. Practical experiences from using PulseAudio in embedded handheld device. In *Linux Plumbers Conf.: Audio Miniconf.*, Portland, OR, USA.

M. Souden, J. Benesty, and S. Affes. 2010. On optimal frequency-domain multichannel linear filtering for noise reduction. *IEEE Trans. on Audio, Speech & Language Processing*, 18(2):260–276.

Ivan Tashev. 2009. *Sound Capture and Processing*. Wiley.

J.-M. Valin. 2007. On adjusting the learning rate in frequency domain echo cancellation with double-talk. *IEEE Trans. on Audio, Speech & Language Processing*, 15(3):1030–1034.

T. Wolff and M. Buck. 2010. A generalized view on microphone array postfilters. In *Proc. Int. Workshop on Acoustic Echo and Noise Control, IWAENC'10*, Tel Aviv, Israel.

Lyapunov Space of Coupled FM Oscillators

Claude Heiland-Allen
claude@mathr.co.uk

Abstract

Consider two coupled oscillators, each modulating the other's frequency. This system is governed by four parameters: the base frequency and modulation index for each oscillator. For some parameter values the system becomes unstable. The Lyapunov exponent is used to measure the instability. Images of the parameter space are generated, with the number crunching implemented on graphics hardware using OpenGL. The mouse position over the displayed image is linked to realtime audio output, creating an audio-visual browser for the 4D parameter space.

Keywords

chaos, DSP, GPU

1 Introduction

Soft Rock EP [ClaudiusMaximus, 2005] and Soft Rock DVD [ClaudiusMaximus, 2006] explored the transitions between order and chaos in coupled FM oscillators. A more recent continuation of this project is to make a map of the parameter space of coupled FM oscillators on a perceptually relevant level and use it in live performance, choosing parameters on the basis of desired sound character.

A bifurcation diagram produced by an analogue Moog synthesizer [Slater, 1998] and images of Lyapunov fractals [Dewdney, 1991] were inspiration to apply the latter technique to the parameter space of coupled FM oscillators in the digital realm.

Figure 1: Example output.

2 Formulation

2.1 Coupled FM Oscillators

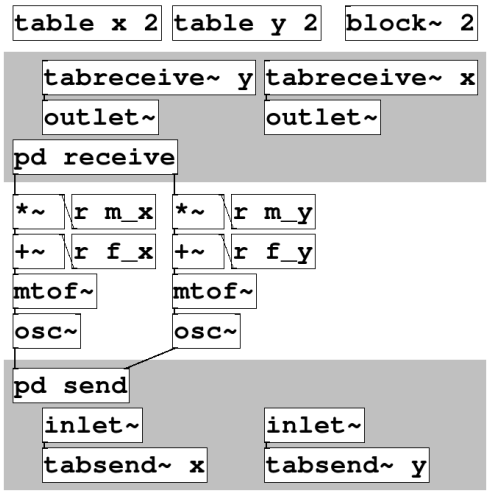


Figure 2: Coupled FM oscillators in Pure-data.

Consider the two coupled oscillators in Figure 2. Pure-data's model of interconnected components each with their own internal state maps poorly to GPU architecture. Considering the whole system as one and flattening the internal state into a single phase space vector leads to the following formulation as a mutual recurrence relation:

$$\begin{aligned} x_{n+1} &= \%(x_n + \mathrm{I}(f_x + m_x \cos(2\pi y_{n-d}))) \\ y_{n+1} &= \%(y_n + \mathrm{I}(f_y + m_y \cos(2\pi x_{n-d}))) \end{aligned} \quad (1)$$

where

$$\%(t) = t - \lfloor t \rfloor, \quad \mathrm{I}(t) = \frac{440}{\mathrm{SR}} 2^{\frac{t-69}{12}}$$

Here x_n, y_n is the phase of each oscillator at time step n, d is a delay measured in samples, f_x, f_y is the base frequency of each oscillator as a MIDI note number, and m_x, m_y is the modulation index of each oscillator as a MIDI note number. $\%(t)$ performs wrapping into $[0, 1)$, with $\lfloor t \rfloor$ being the flooring operation (the largest integer not greater than t).

The four-dimensional parameter space vector will be written

$$a = (f_x, f_y, m_x, m_y)$$

and the $(2d+2)$-dimensional phase space vector

$$z = (x_n, y_n, x_{n-1}, y_{n-1}, \ldots, x_{n-d}, y_{n-d})$$

with sample rate $\mathrm{SR} = 48000$. For reasons explained in Section 5.2, $d = 1$ will be fixed.

2.2 Lyapunov Exponents

Lyapunov exponents can be used to measure the stability (or otherwise) of a dynamical system. A good introduction is found in Chapter 4.3 *Lyapunov Exponent* [Elert, 2007]. The definition is covered in Chapter 13.7 *Liapounov exponents and entropies* [Falconer, 2003] which also relates it to measures of fractal dimension.

The Lyapunov exponent λ measures divergence in phase space:

$$|z_1(t) - z_0(t)| \approx e^{\lambda t} |z_1(0) - z_0(0)|$$

$$\lambda = \lim_{\substack{t \to \infty \\ z_1(0) \to z_0(0)}} \frac{1}{t} \log \frac{|z_1(t) - z_0(t)|}{|z_1(0) - z_0(0)|} \quad (2)$$

An attracting orbit has $\lambda < 0$ and a divergent (chaotic) orbit has $\lambda > 0$.

A modified norm is required to take into account the wrapping of phase into $[0, 1)$:

$$|z|_{\%} = \sqrt{\sum_i \left(\min(\%(z_i), 1 - \%(z_i))\right)^2}$$

For example the distance between 0.1 and 0.9 is properly 0.2 (not 0.8) because 0.1 can be phase-unwrapped to 1.1.

2.3 Viewing Planes

An image is 2D, which requires choosing a subset of the 4D parameter space to visualize. Two particular planes were chosen:

$$\begin{aligned} A_+(a_0, r_0) &= a_0 + r_0 \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \\ A_-(a_0, r_0) &= a_0 + r_0 \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \end{aligned} \quad (3)$$

where (u, v) is the coordinates of the pixel, a_0 is the centre of the view, and r_0 is the radius of the view.

These planes were chosen because they are simple, while still being flexible enough to explore the whole 4D space. The A_+ plane varies both oscillators in the same direction, while the A_- plane varies each oscillator in opposite directions. To center on a particular target point (f_x, f_y, m_x, m_y) one might use the A_+ plane to

center on the midpoint

$$\left(\frac{f_x + f_y}{2}, \frac{f_x + f_y}{2}, \frac{m_x + m_y}{2}, \frac{m_x + m_y}{2}\right)$$

and then switch to the A_- plane to break the (x, y) symmetry.

3 Implementation

The implementation uses OpenGL [Segal, 2013] and OpenGL Shading Language [Kessenich, 2013] for computation and graphical rendering, GLUT [Kilgard, 1996] for windowing and input event handling, and JACK [Davis, 2013] for audio output.

3.1 Introduction to Modern OpenGL

Modern OpenGL has a programmable shader pipeline. Vertex attributes are read from vertex buffers and processed by vertex shaders. The outputs of the vertex shader (called varyings) are further manipulated by an optional geometry shader stage. Geometry shaders can output a different vertex count to their input count, whereas vertex shaders are one-in one-out. The result of the geometry shader can be captured into another vertex buffer using transform feedback. Following the geometry shader the primitives (points or triangles) are rasterized, and varyings interpolated across each primitive. Finally a fragment shader takes these values and computes the colour at that pixel. The output of a fragment shader can be captured by attaching a texture to a framebuffer.

3.2 Computation Overview

To render an image a texture is first filled with (u, v) coordinates using a framebuffer object and a fragment shader. This texture is copied to a vertex buffer object, interleaved with an initial phase space vector $z = (0, 0, 0, 0)$ and Lyapunov exponent statistics vector $l = (0, 0, 0, 0)$ for each point.

Using a vertex shader, a is calculated from (u, v) using Equation 3, and then a rough estimate of the Lyapunov exponent is computed using Equation 2 by perturbing $z_1(0) = z_0(0)+\delta$ with δ small and performing $t = 256$ iterations of Equation 1. The first few repetitions are discarded, along with those resulting in $-\infty$, and the rough λ estimates are accumulated in l.

Between each repetition the working set is compacted using a geometry shader. Points whose mean Lyapunov exponent estimate changed very little during the previous step are plotted and removed from the working set. The other points are kept to be refined further, directing the computational effort on the points that need it most: those slow to converge.

To ensure user interface responsiveness, the computation is amortized over several frames. The target frame period is divided by the measured time for one repetition to compute how many repetitions to perform that frame. The repetitions-per-frame increases as the working set becomes smaller.

3.3 Noise Increases Stability

At the end of each repetition z_1 is kept instead of z_0. This effectively adds a small amount of noise, counter-intuitively increasing stability. Noise allows more of the phase space to be explored, and makes it more likely for the perturbed orbit to reach an attracting part of the phase space.

3.4 Dither Increases Quality

To reduce grid sampling artifacts, (u, v) is perturbed within the bounds of its corresponding pixel before calculating the a parameter vector for each repetition.

4 Results

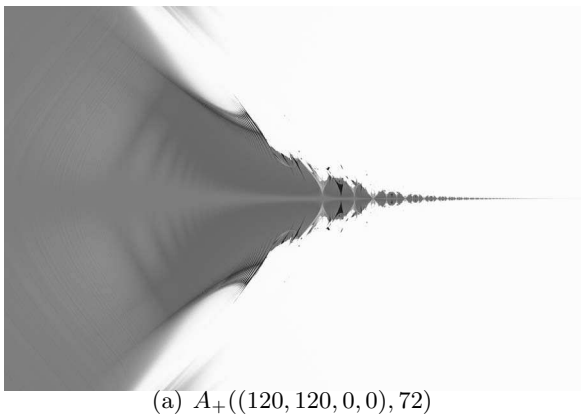
4.1 Examples

Figure 3(a) shows the initial view on starting the interactive browser. Low frequencies to the left are stable even at high modulation index away from the central axis. High frequencies to the right become chaotic at progressively lower modulation index. (b) shows the A_- plane at the same location. (c) shows bands alternating between stability and chaos. The bands become distorted and collapse as the modulation index and frequency increase. (d) shows its A_- plane, bands become rings. When the frequency is greatly increased, the shapes become more intricate. (e) exhibits spirals of stability, with similar spirals in the A_- plane in (f).

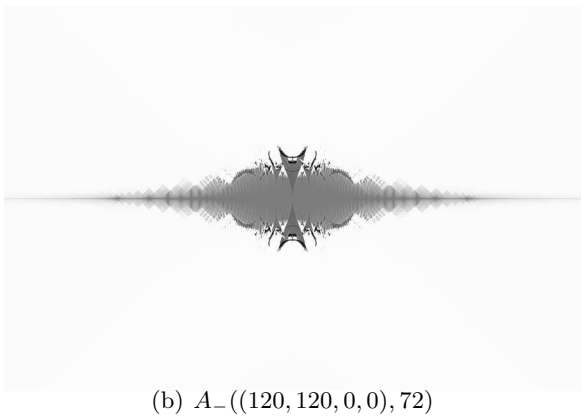
When $f_x = f_y$ and $m_x = m_y$ the A_+ plane has mirror symmetry about its horizontal axis, and the A_- plane has two-fold rotational symmetry about its centre. Breaking the symmetry and setting $f_x \neq f_y$ or $m_x \neq m_y$ leads to diverse forms. In particular Figure 3(h) has shapes that resemble those of Lyapunov space images of the logistic map.

4.2 Interactive Explorer

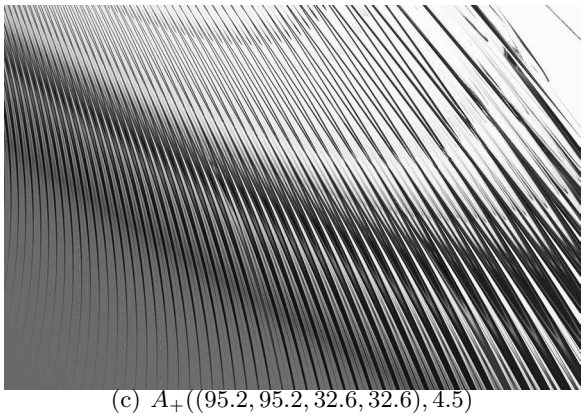
The implementation is an interactive audio-visual explorer for the parameter space of cou-



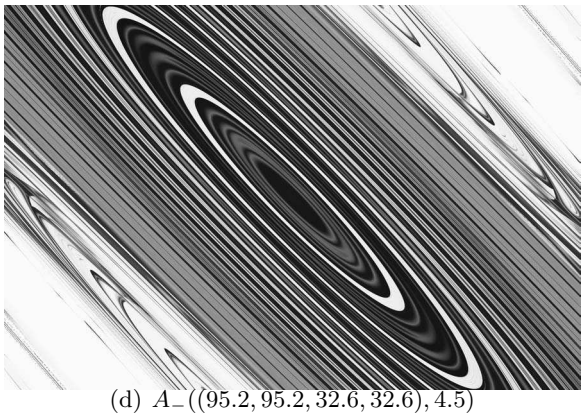
(a) $A_+((120, 120, 0, 0), 72)$



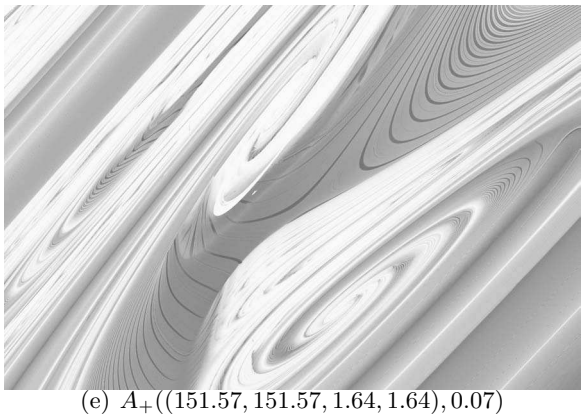
(b) $A_-((120, 120, 0, 0), 72)$



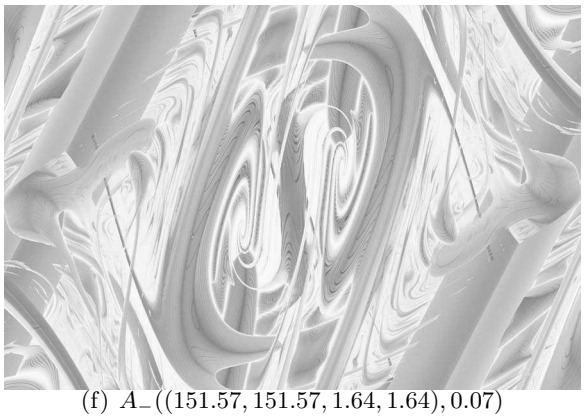
(c) $A_+((95.2, 95.2, 32.6, 32.6), 4.5)$



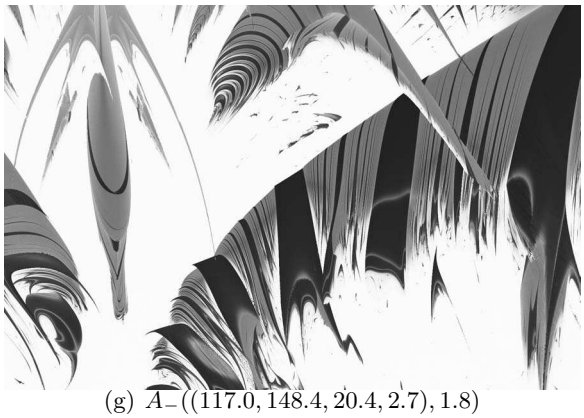
(d) $A_-((95.2, 95.2, 32.6, 32.6), 4.5)$



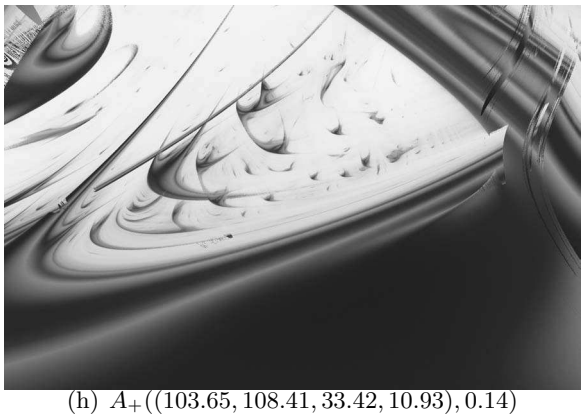
(e) $A_+((151.57, 151.57, 1.64, 1.64), 0.07)$



(f) $A_-((151.57, 151.57, 1.64, 1.64), 0.07)$



(g) $A_-((117.0, 148.4, 20.4, 2.7), 1.8)$



(h) $A_+((103.65, 108.41, 33.42, 10.93), 0.14)$

Figure 3: Example images. Darker shades are stable, lighter shades chaotic.

pled FM oscillators. Clicking with the mouse zooms the view about the clicked point. The left button (or scroll up) zooms in, the right button (or scroll down) zooms out, the middle button centers the view on the target point. Pressing the TAB key toggles between the A_+ and A_- planes in Equation 3, and F11 toggles full screen operation.

While the GPU simulates and analyses one oscillator pair per pixel, the CPU simulates one oscillator pair with a determined from the pixel under the mouse pointer. The image acts as a map, a reference frame for chosing parameters to audition by moving the mouse.

5 Conclusions

5.1 Original Intent

Earlier experiments used one Pure-data batch mode instance per CPU core each sending analysis data to a realtime Pure-data instance. The analysis used various methods (including FFT for spectral statistics and the sigmund external for pitch tracking) to classify points into pitched (ordered, stable) or unpitched (chaotic, unstable) with measures of distortion or noisiness. Sadly this approach proved impractical as it achieved only tens of pixels per second, even with a fast multi-core CPU, and porting these signal analysis algorithms to massively-parallel programmable graphics hardware seemed to be too difficult.

5.2 OpenGL Issues

The current implementation is hardcoded with delay $d = 1$ and would be very awkward to generalize. OpenGL architecture limits each vertex attribute to four components with the maximum number of attributes typically limited to sixteen. This totals 64 floats per vertex, 6 of which are needed for the pixel coordinates and Lyapunov exponent statistics accumulation. Therefore using OpenGL imposes a limit $d < 28$. For comparison the original experiments in *Soft Rock EP* used Pure-data's default block size of 64, with $d = 32$. Moreover, increasing d increases video memory consumption. With the maximum $d = 27$, browsing at 1920×1080 resolution would require over 1GB.

Future work on this project will look into using OpenCL, which provides a heterogenous CPU and GPU computation framework, in the hope that it will avoid the inherent awkwardness of abusing OpenGL shaders to perform calculations.

5.3 Audio Issues

While the implementation works as intended, with $d = 1$ the sound is nowhere near as rich and varied as with $d = 32$. With small d there is much more very high frequency content in interesting-looking regions. There seem to be few if any regions of the parameter space with both interesting appearance and palatable audio frequencies at $d = 1$, while with high d there are parameters that generate sounds that fluctuate intermittently between smooth tones and noise. Visualization with high d has not been possible so far, so whether their neighbourhoods look as interesting as they sound remains an openquestion .

Unfortunately, heavy use of the GPU in the interactive browser can block the operating system for too long and cause audible glitches (JACK xruns). This situation may change as free drivers continue to improve, allowing use of the browser in a live situation.

5.4 Pretty Pictures

Despite these shortcomings, I think the images look good. I plan to render a selection at high resolution and print postcards and posters. For huge images it is possible to divide the image plane into tiles and compute each tile in succession, finally combining the pieces into one large picture.

There is also scope for video work, moving and rotating the viewing plane through the 4D parameter space, with different shapes forming and collapsing over time. Rough benchmarks take 5-10 seconds per frame at 1920×1080, so it seems sensible to wait until faster cheaper graphics cards become available.

6 Obtaining the Implementation

The implementation was written on GNU/Linux Debian Wheezy running on a quad-core AMD64 processor with NVIDIA GTX 550Ti graphics card using proprietary drivers. The source code is available at: `https://gitorious.org/maximus/lyapunov-fm`

7 Acknowledgements

Thanks to the anonymous reviewers for their constructive criticism on a number of issues, and to Rob Canning, Adnan Hadzi, and Joanne Seale for their helpful feedback on earlier versions of this paper.

References

ClaudiusMaximus. 2005. *Soft Rock EP*. `http://archive.org/details/ClaudiusMaximus_-_Soft_Rock_EP`.

ClaudiusMaximus. 2006. *Soft Rock DVD*. `http://archive.org/details/ClaudiusMaximus_-_Soft_Rock_DVD`.

Paul Davis. 2013. *The JACK Audio Connection Kit*. `http://jackaudio.org`.

A. K. Dewdney. 1991. Mathematical Recreations: Leaping into Lyapunov Space. *Scientific American*, 265:178–180.

Glenn Elert. 2007. *The Chaos Hypertextbook*. `http://hypertextbook.com/chaos/`.

Kenneth Falconer. 2003. *Fractal Geometry: Mathematical Foundations and Applications, Second Edition*. Wiley.

John Kessenich. 2013. *The OpenGL Shading Language*. `http://www.opengl.org/registry/doc/GLSLangSpec.4.30.8.pdf`.

Mark J. Kilgard. 1996. *The OpenGL Utility Toolkit (GLUT) Programming Interface*. `http://www.opengl.org/documentation/specs/glut/glut-3.spec.pdf`.

Mark Segal. 2013. *The OpenGL Graphics System: A Specification*. `http://www.opengl.org/registry/doc/glspec43.core.20130214.pdf`.

Dan Slater. 1998. Chaotic Sound Synthesis. *Computer Music Journal*, 22(2):12–19.

Production and Application of Room Impulse Responses for Multichannel Setups using FLOSS Tools

Florian HOLLERWEGER
A – 4020 Linz
Austria
flo@mur.at

Martin RUMORI
University of Music and Performing Arts Graz
Institute of Electronic Music and Acoustics
Inffeldgasse 10 · A – 8010 Graz · Austria
rumori@iem.at

Abstract

We present the outcomes of a series of room impulse response (IR) measurements. We have recorded binaural, Ambisonics-encoded and regular stereo/mono IRs of multichannel loudspeaker arrays in various concert halls in Austria, Northern Ireland, Germany and New Zealand. The resulting IRs and accompanying documentation have been made publicly available on a website for composers to use in the production and documentation of their multichannel pieces. The paper also discusses several custom-written shell scripts and extensions to the *Aliki* and *Jconvolver* software packages, which we have developed for the production of the presented IRs.

Keywords

Impulse response, binaural auralisation, virtual acoustics, convolution reverb

1 Introduction

In an earlier paper [Rumori et al., 2010], we have discussed *binaural room impulse responses* (BRIRs), i.e., impulse responses (IRs) recorded using a dummy head in a 'real' room, as opposed to an anechoic chamber. We have described the usefulness of BRIRs for the pre-production and documentation of multichannel electroacoustic compositions. By recording the BRIRs of all loudspeakers in a multichannel electroacoustic concert hall, one can generate binaural mixdowns of pieces composed for and/or performed on that loudspeaker array. When listened to on stereo headphones, such a mixdown preserves not only the hall's reverberation characteristics, but also the perceived 3D direction of all loudspeakers and phantom sources between them. BRIRs thus combine the advantages of single-source stereo IRs used in standard convolution reverbs with those of head-related IRs usually recorded in anechoic chambers.

Binaural mixdowns created from BRIRs assist the composer with the pre-production of multichannel pieces written for concert halls where access is often restricted and rehearsal time limited. Moreover, they provide an efficient method for documenting multichannel pieces in a format that can be played back on a simple pair of headphones, but nevertheless conveys the acoustic spatiality intended by the composer. Realtime convolution engines such as *BruteFIR* or *Jconvolver* extend the usefulness of BRIRs towards real-time audio input.

Since the publication of our last paper, we have conducted additional room impulse response recordings, using both binaural and other recording techniques, in a variety of concert halls. One purpose of this paper is to provide an overview of these recordings (cf., section 2) and the resulting IR repositories, which we have made freely available (cf., section 5). We also present some examples of how different artists have applied these impulse responses (cf., section 4).

Free/Libre Open Source Software (FLOSS) packages have played a central role in the production of the IRs that we present, and are also essential for their actual application. The creative use of IRs is an area of audio processing where the limited scope of commercial software packages can easily become apparent. The ability to customise many of the FLOSS tools that we have used was both welcome and necessary to achieve our goals. Another purpose of our paper is to provide an overview of our personal extensions to FLOSS tools such as *Aliki* and *Jconvolver* (cf., section 3).

2 Recording Sessions and Venues

2.1 IEM *CUBE* and SARC *Sonic Lab*

The *CUBE* concert hall at the Institute of Electronic Music and Acoustics (IEM) in Graz, Austria is a small concert space equipped with 24 Tannoy System 1200 loudspeakers, which are arranged in a hemisphere above the listener and optimised for Ambisonics playback.

The *Sonic Lab* at the Sonic Arts Research Centre, Queen's University Belfast, is a large

Figure 1: The *CUBE* at the Institute of Electronic Music and Acoustics in Graz (photograph by courtesy of the Graz University of Music and Performing Arts)

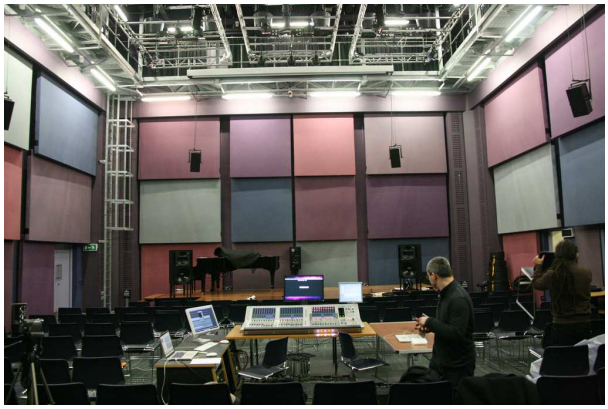


Figure 2: The *Sonic Lab* at the Sonic Arts Research Centre in Belfast (photograph by courtesy of Gawain Morrison)

electroacoustic concert hall whose acoustics can be adapted through movable wall panels. At the time of our recording, the Sonic Lab featured 40 full-range speakers (Meyer Sound and Genelec) and 6 subwoofers.

Details on the IRs that we have recorded in the CUBE and the Sonic Lab can be found in our previous paper [Rumori et al., 2010], so we will not reiterate them here.

2.2 New Zealand School of Music

The Adam Concert Room (ACR) is located at the Kelburn Campus of Te Kōkī New Zealand School of Music (NZSM), a joint venture of Massey University and Victoria University of Wellington. The ACR is a medium-size concert hall which is predominantly constructed from wood and seats up to 200 people.

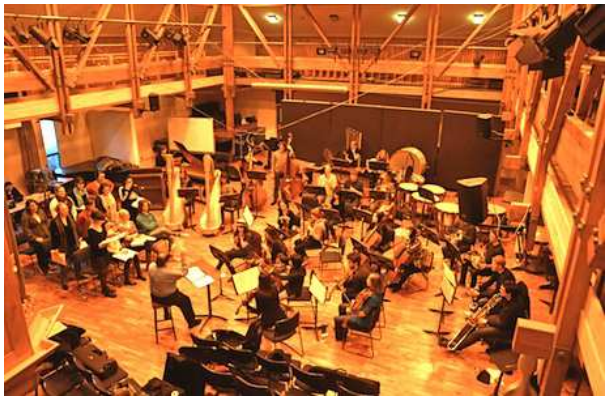


Figure 3: The Adam Concert Room (ACR) at Te Kōkī New Zealand School of Music in Wellington (photograph by courtesy of Stephen Gibbs)

In October 2011, an IR recording session was conducted in the ACR to conclude a one-trimester seminar on *Spatial Audio*. The final repository includes binaural IRs (recorded by a *Kemar 45BA*) and experimental Ambisonics-encoded IRs (*CoreSound TetraMic*) of a circular eight-channel speaker array (*Genelec 1037B/C*), plus standard stereo IRs (X/Y pair of *Neumann KM184s*) of the same array and for a single centred speaker. The repository is accompanied by extensive documentation of the production process and includes several demo scripts to illustrate the usage of the final IRs.

2.3 Academy of Media Arts Cologne

The auditorium of the Academy of Media Arts Cologne (KHM) is a medium-size, multipurpose hall used for lectures, presentations, film screenings and concerts, the latter mostly of an experimental kind, featuring electroacoustic and improvised music. The reverb time of the hall is rather short as it is mainly conceived for good speech intelligibility.

In November 2011, a hands-on workshop called *Raumfaltung* on measuring and using impulse responses has been carried out at KHM. During the workshop, a "classic" speaker ring of eight *K+H O 108* has been set up in the auditorium and room impulse responses for each of the speakers were measured using an array of different microphones: two kinds of custom-built dummy heads,[1] an ORTF configuration of two *Schoeps MK 4*, a small AB configuration of two *Schoeps MK 2 s*, a *Soundfield ST 250* first-

[1] Binaural microphones *Sellmeier Grey* and *Sellmeier Brown*, both constructed by Wilfried Sellmeier

Figure 4: The microphone setup at Academy of Media Arts Cologne (photograph by courtesy of Dirk Specht)

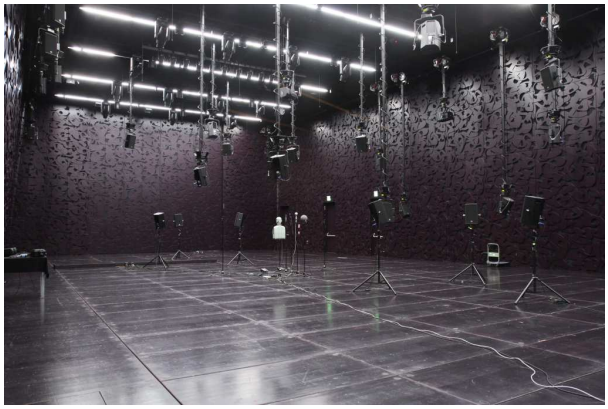


Figure 5: Speaker rig and measurement setup at *Ligeti Hall* in MUMUTH Graz

order Ambisonics microphone and a *Behringer ECM 8000* measurement microphone as a reference channel above one of the dummy heads.

The impulse responses are suitable for mono or stereo convolution reverb applications as well as for auralisation of up to eight channels or binaurally rendered horizontal Ambisonics.

2.4 MUMUTH Graz

MUMUTH Graz is the House of Music and Music Theatre at the University of Music and Performing Arts Graz (KUG). Its main hall, the *Ligeti Hall*, is a multi-purpose space for theatre and opera production, concerts, dance and electroacoustic music. It features a rig of 33 speakers with a motorised position control, which allows for adjusting the heights, pan and tilt of every speaker individually. This way, speaker configurations can be changed and evaluated rather quickly. Additionally, *Ligeti Hall* is equipped with 64 fixed smaller speakers and eight subwoofers, which belong to a virtual room acoustics system, but which can be also directly fed with arbitrary signals. Further speakers and subwoofers may be hung or installed on stands if needed.

Since fall 2010, the artistic research project *The Choreography of Sound*[2] is carried out at the Institute of Electronic Music and Acoustics (IEM) at KUG. For the case studies of this project, the *Ligeti Hall* is considered the main instrument. As access times to the hall are quite limited, a binaural auralisation of its acoustic properties became desirable, which turned out to be also quite useful for taking up a different perspective on the instrument. As part of the *VirtualMUMUTH* subproject (see section 4.4), many room impulse response measurement sessions were undertaken between February 2011 and August 2012, for several combinations of loudspeaker configurations and microphone (i. e. listening) positions.

In order to achieve a high versatility, the measurements were done using an arrangement of multiple microphones: a *Brüel & Kjær* dummy head with additional *DPA 4060* capsules mounted on its temples, a *Schoeps* spherical stereo microphone, an Ambisonics *Soundfield* mic, a large AB configuration of two *Schoeps MK 2s* and a mono reference. Due to the high number of channels and the resulting large amount of data, some of the measurement tools needed to be extended as described in section 3.

3 Tools and Customisations

The abovementioned measurements and their post-production have been carried out entirely using FLOSS tools. The central tool being used for all the venues described here is *Aliki* by Fons Adriaensen [Adriaensen, 2006a]. In larger multichannel settings with many speakers, several microphone channels and lots of measurement iterations like at MUMUTH (cf. section 2.4), a few additions and adaptations to Aliki became necessary.

For post-processing and using the measured impulse responses, some further tools have been developed, such as scripts based on `sox` for trimming and fading and a script for generating configurations for *Jconvolver*, a high-performance convolution engine [Adriaensen, 2006b].

[2] `http://cos.kug.ac.at`, last retrieved 2013–02–17, see also [Eckel et al., 2012]

3.1 Aliki and around

Aliki is a powerful and free impulse response measurement framework, which implements the swept-sine measurement method [Farina, 2000]. It provides means for generating the sweep signal, playing it back and recording the room's response, for deconvolving the recorded sweep in order to get the impulse response, some functionality for editing the resulting responses and for exporting them.

3.1.1 Automated speaker channel switching

In a larger setup such as in MUMUTH (cf. section 2.4), it is desirable to carry out multiple iterations, e. g. for several speaker channels in a row, in an automated way. Unfortunately, standard Aliki only offers eight output channels and no means of directly switching them automatically. There is a functionality though for specifying the number of iterations and a trigger command, which is executed once every cycle.

For the automated measurements described here, this trigger has been used to send an OSC message to a small SuperCollider patch, which receives the sweep signal on a single input channel (fixed output on channel 1 of Aliki) via the Jack audio server. Upon an incoming OSC trigger message, the sweep signal is routed sequentially to the corresponding loudspeaker output channels. This way, also more complex channel sequences or groups with non-contiguous channel numbers could be easily realised.

For convenience, the custom version of Aliki (as used in MUMUTH) also contained a *reset command* field, which could be triggered by a button. It was used to send another OSC message to reset the channel routing to the starting point.

3.1.2 More input and output channels

Standard Aliki provides eight recording channels for the microphone signals. In MUMUTH, due to the high number of speaker channels (up to 128) and long sweep times (> 10 sec), the actual net measurement time was already so long that it seemed appropriate to add a few more microphones, as those additional impulse responses come "for free" (time wise) and might be suitable for future applications, even if the focus was on binaural room impulse responses. Thus, a configuration of 13 microphone channels arose, including a dummy head, several different stereo pairs, a Soundfield mic (Ambisonics A-format) and a reference channel.

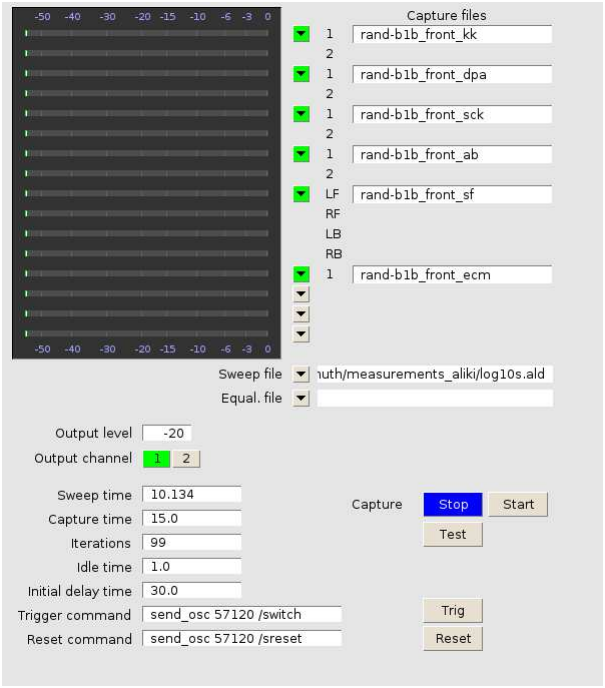


Figure 6: Capture dialog of the extended *Aliki* using 16 input channels, trigger and reset commands and an initial delay

The custom version of Aliki was modified such that the number of audio input channels could be specified as a command line option at startup. In the same manner, also the number of output channels has been parametrised, although this was not needed for the scope of these measurements.

3.1.3 Initial delay

While carrying out a large, automated sequence of iterations, it might be convenient to leave the space being measured in order to minimise the extra noise induced (and to minimise the sweep level in the poor operator's ears). In the custom version of Aliki, an *initial delay time* may be specified which allows for leaving the space well before the first sweep starts.

Before Aliki was customised for that, sometimes the first measurement (first sweep playback) was regarded a "Nop" and discarded later. This creates problems such as reduced flexibility (fixed sweep length depending on the measurement) and messed-up channel numbers when exporting the impulse responses to single files automatically.

3.1.4 CLI deconvolution and export

Aliki includes the deconvolution stage of the recorded sweeps in order to generate the actual impulse responses, and an export function for

converting from Aliki's sound file format (`.ald`) to standard formats such as `.wav` or `.aiff`. Aliki's file format supports multiple sections in a single file, which correspond to the iterations of a sequential measurement. Both the deconvolution and the export may be performed for a single section of a file or for all sections of that file, using the same parameters. The latter option comes in handy when post-processing measurements with a high number of iterations.

However, the measurements reported here, especially those in MUMUTH (cf. section 2.4), not only included many iterations per file but also a large number of those files, due to the high amount of combinations of loudspeaker configurations and microphone positions. Therefore, the tedious and error-prone task of manually loading each file for deconvolution and export was replaced by two custom command-line applications, `aliki-convol` and `aliki-export`, which are basically factored-out versions of the respective built-in functions.

Corresponding to the GUI options in Aliki, `aliki-convol` allows for specifying the start and end times of the deconvolution and whether one or all sections of the file shall be processed. Likewise, `aliki-export` accepts options for the export file format and bit depth, whether a single section, all sections in one file or all sections in separate files shall be exported, the file name base and an offset for file numbering.

3.2 Post-processing impulse responses

All impulse responses taken in the above-mentioned measurement series were not post-processed in Aliki, but through separate tools or scripts. This is mostly due to the high number of responses and the different applications they are targeted at.

Post-processing here basically means trimming the impulse responses, compensating the measurement system latency at the beginning of the file, in some cases also the air travelling time of the direct sound, adding a short fade-in to the beginning, trimming the length of the impulse response and applying a fade-out at the end. Other common tasks are gain adjustments, either by a fixed factor or an individual normalisation, and possibly resampling to a different target sample rate.

3.2.1 `imptrim.sh`

`imptrim.sh` is a shell script used for post-processing the MUMUTH measurements. It is a simple wrapper script around `sox`, which facilitates the handling of large sets of files with a contiguous number part of a fixed width (e. g. zero-padded) in their filenames. A certain range of numbered files may be processed or all contiguously numbered files found, starting from a given index. For the output files, a different filename pattern and a different starting index for filename numbering may be specified, which enables simple renaming operations for chunks of files at the same time as processing.

`imptrim.sh` allows for separately specifying two portions of the audiofiles to be cut from the beginning, e. g. a compensation portion in samples and another trim in seconds. Further options include the overall length of the resulting impulse response, the length and shape of the fade-in and fade-out and a gain factor. Optionally, a stamp file may be generated in the output directory as a log for the parameters being used for processing.

3.2.2 `remapch.sh`

This is a helper script used for the MUMUTH measurements which actually does not process any audio but only facilitates renaming chunks of numbered impulse response files. It is included here because it might be of general interest for adapting it to other settings.

Measuring and exporting a sequence of speakers whose channel numbers do not start at 1 will result in filenames which do not match the original channel numbers anymore. For example, measuring the MUMUTH sky (channels 65 – 128) will result in numbered filenames ranging from 1 to 64. This may be avoided by using `aliki-export` and specifying an offset for the output filename (see section 3.1.4).

But when carrying out a large sequence of measurements over a non-contiguous, i. e. "sparse" array of channel numbers, the filename number offset will be different for each loudspeaker block. In MUMUTH for example, when measuring the hemisphere, corner and sky speakers plus two subwoofers in a row, the hemisphere and corner speaker numbers will remain (channels 1 – 33), the subwoofer channels 42 and 43 will be named 34 and 35, respectively, and the sky channels 65 – 128 will result in files numbered from 36 to 99.

`remapch.sh` allows for entering such translations for certain regions of channel numbers into a simple database within the script. It may then be used for renaming operations on chunks of response files in both directions, based on the specified configuration (translation).

3.2.3 `post_export.sh`

The `post_export.sh` script was written specifically for the production of the IRs of the Adam Concert Room at the New Zealand School of Music (cf., section 2.2). It provides an alternative implementation of some of the functions provided by the `imptrim.sh` and `remap.sh` scripts, as well as some additional features.

`post_export.sh` aims at automating the entire IR post-processing after the recorded sweeps have been deconvolved and exported to `.wav`. Similar to the tools presented so far, the idea is to automate the process as much as possible. Thus, all edits are performed with command line tools such as `sox` rather than graphically through the Aliki GUI. Being very much tailored to the specific requirements of the production it was used for, the script is not entirely generic, but hopefully still useful for other IR measurements after according adaptations. The script performs the following processing steps:

Renaming of the IRs exported from Aliki with `mv`, such that they are named according to the respective loudspeaker's direction. For example, the suffix `_nnw` denotes a loudspeaker in north-north-west direction, deliberately (but falsely) assuming that the listener always faces north.

A-to-B format conversion of the experimental Ambisonics-encoded IRs. This step is performed using the `tetrafile` tool by Fons Adriaensen, who also kindly provided the required calibration file for the specific TetraMic (serial № 2099) that was used in the recording.

Latency compensation countering the effects of the recording system's latency, similar to the `imptrim.sh` script.

Trimming the IRs to a pre-defined length with `sox`, similar to `imptrim.sh`.

Fade-in and fade-out with `sox`, similar to `imptrim.sh`.

Normalisation of the IRs to an arbitrary peak value in dB.[3] With the help of some bash logic, this is performed in groups, such that for every recorded take, the IRs of all loudspeakers in the multichannel setup are normalised to the same target value. The normalisation is deliberately performed prior to converting the IRs to different sample rates, despite the theoretical possibility of clipping due to intersample peaks (which can in practice be avoided by choosing the target value low enough). If an IR was to be normalised *after* resampling, intersample peaks might translate to slightly varying peak levels for different sample rates. The same IR would then be normalised by a different amount for each sample rate.

Resampling from 96 to 44.1 and 48 kHz, using the `sndfile-resample` tool from the `samplerate-programs` Debian package.

[3] This step is not strictly necessary, assuming that a floating point number representation is used for the underlying audio files, and considering that in a binaural mixdown of a multichannel piece, many signals will eventually be superimposed anyway.

3.3 Jconvolver helpers

The tools described in this section do not actually belong to the measurement or post-processing of impulse responses but to their application. In many use cases of the measurements described here, the free command-line convolution engine *Jconvolver* [Adriaensen, 2006b] has been used. Its highly efficient implementation of the so-called *Gardner convolution* using increasing partition sizes makes it especially suitable for long impulse responses and many channels, while at the same time allowing for low-latency operation.

3.3.1 `genjconv.sh` and `genjconvn.sh`

Jconvolver uses a configuration file in OSC-message-style syntax for describing the convolution matrix. The settings include the number of input and output channels, the minimum and maximum partition size, the locations, names and channels of the soundfiles containing the impulse response and a couple of options for each impulse response, such as gain or delay.

For the variety of impulse response sets described in this paper, especially those of different combinable loudspeaker groups in MUMUTH (cf. section 2.4), one should be able to quickly and flexibly generate *Jconvolver* configurations. For this task, the `genjconv.sh` and `genjconvn.sh` scripts have been conceived.

Both scripts write a *Jconvolver* configuration file to stdout based on a list of impulse response files read from stdin. The general usage scheme therefore is:

```
ls ir_*.wav | sort -n | genjconv.sh \
    > irjconv.conf
```

The scripts try to be smart with respect to the input and output channels: By default, the number of input channels of the convolution matrix is assumed to equal the number of input files, while the number of output channels is determined from the number of channels of the first input file. Both parameters may also be specified manually, as well as the number of input file channels used and a channel offset.

`genjconv.sh` maps the input channels of the convolution matrix sequentially to the list of input impulse response files. It is possible to specify exclude regions for both input and output channel numbers of the matrix in order to leave "holes", i. e. produce a "sparse" convolution matrix.

`genjconvn.sh` parses for numbers in the input impulse response filenames and creates a matrix whose input channel numbers are mapped to the corresponding filenames. Currently, this works for zero-padded three-digit numbers directly in front of the last dot in the filename. This facilitates the generation of "sparse" convolution matrices and ensures that impulse response files for certain speaker groups are always associated with the right input channels, independent of the total number of input files given (assuming that they are named accordingly).

For completeness, both scripts allow for specifying a fixed gain factor for all impulse responses, the base path to be used, the minimum partition size, and the maximum impulse response length.

3.3.2 `jjconvolver.sh`

When using manually written or automatically generated *Jconvolver* configuration files across different systems, a common problem is that the impulse responses reside in a different location than specified in the configuration file as an absolute path. Often, the difference is negligible and is caused only by the different name of the home directory or a few initial path elements, as the deeper structure of a commonly used impulse response database is likely to be the same.

`jjconvolver.sh` tries to solve this problem by simply replacing the `/cd` directive in the *Jconvolver* configuration, generating a temporary file and starting *Jconvolver* using the adapted configuration. It acts as a frontend script to *Jconvolver* and passes through any of its options. Additionally, it accepts a few extra options which either specify a full path to completely replace the existing one in the configuration, or the initial elements of the new path from which the complete path is guessed. This is done by concatenating the given initial path and the existing one, subsequently stripping one hierarchy from the front of the existing path, until a directory with the resulting name is found in the filesystem. Although this is a very simple mechanism, it turned out to be quite effective in many practical use cases.

4 Application examples

4.1 Electroacoustic compositions

In our previous paper [Rumori et al., 2010], we have already discussed some artistic applications of our impulse responses by different composers. Justin Yang, Gary Kendall, and John Moeller have used the IRs recorded in the SARC Sonic Lab for the production and documentation of several multichannel electroacoustic compositions. Dirk Specht and Gerriet K. Sharma have published binaural mixdowns of their multichannel pieces on two CDs, using the IRs recorded in the IEM CUBE.

4.2 Pop Production

Nic McBride has used the binaural room impulse responses from the Adam Concert Room at the New Zealand School of Music to create a headphone mix of eight distinct stems in a pop production entitled *L'addio Scontato*. This example represents an original use of BRIRs in the sense that the final production was never primarily targeted at multichannel playback – one cannot really speak of a stereo 'reduction' in this context. According to Nic, the BRIRs allowed her to achieve a natural-sounding spatial separation in the final binaural mix, which has been released online [McBride, 2012].

4.3 Audio Augmented Environments

Binaural impulse responses recorded in several of the described measurement sessions have been used by Martin Rumori in sound installations which form different kinds of audio-augmented environments. They include navigable virtual environments incorporating a headphone tracking system, such as *Parisflâneur*,[4] or mixed-reality installations, where the virtual layer blends with the real auditive surrounding of the listener, using open-loop headphones, such as in *ruhrprotokolle*.[5]

[4] `http://www.rumori.at/do/parisflaneur`

[5] `http://www.ruhrprotokolle.de`

4.4 VirtualMUMUTH

Within the research project *The Choreography of Sound* (see section 2.4), the impulse responses measured in MUMUTH's *Ligeti Hall* are used for an auralisation of the acoustic properties of the space. The *Jconvolver* helpers described in section 3.3 have been integrated with the *CoS* software framework conceived in SuperCollider, which will be published along with the project's documentation in the second half of 2013. The auralisation is completed by a visual three-dimensional navigable model of *Ligeti Hall*, realised in Blender and also integrated with the software library.

Participants of the *CoS* workshop at *Impuls Academy 2013*[6] in February 2013 and artists invited to the *Mind the Gap!* symposium in MUMUTH in March 2013[7] prepared their contributions using the *VirtualMUMUTH* standalone application.

5 Availability

The impulse responses from all recording sessions described in section 2, including the patches, scripts and helpers introduced in section 3 as well as the *VirtualMUMUTH* auralisation tool mentioned in section 4.4, have been made freely available at `http://irdb.kug.ac.at`.

6 Conclusion

Impulse responses of multichannel loudspeaker arrays in real concert halls provide many useful applications for music production and the documentation of electroacoustic compositions. FLOSS tools continue to play an essential role in the recording and usage of such IRs, due to the customisability that these tools offer, and the current limitations of commercial software packages.

We are looking forward to feedback from music producers and composers who would like to use the presented impulse responses for their own work, hopefully in unusual and original ways that we have not envisioned here.

7 Acknowledgements

The measurements at MUMUTH Graz were carried out within the *Choreography of Sound* artistic research project initiated by Gerhard Eckel and Ramón González-Arroyo, funded by the Austrian Science Fund (FWF) under the project code PEEK AR 41.

Fons Adriaensen has provided much-appreciated support for his excellent *Aliki*, *Jconvolver* and *Tetrafile* packages.

We would also like to acknowledge the contributions of Bridget Johnson, Jason Post, Stuart Macann, Roy Carr, Dugal McKinnon and Mark McGann at the New Zealand School of Music, and of Mark Poletti at Industrial Research Ltd. in Lower Hutt.

At the Sonic Arts Research Centre in Belfast, we would like to thank Andrés Cabrera, Gary Kendall, John Moeller, Justin Yang and Chris Corrigan.

Thanks to Anthony Moore and Dirk Specht for organising the measurement workshop at the Academy of Media Arts Cologne.

[6] `http://www.impuls.cc`

[7] `http://www.researchcatalogue.net/view/33841/37723/1022/567`

References

Fons Adriaensen. 2006a. Acoustical impulse response measurement with ALIKI. In *Proceedings of the International Linux Audio Conference 2006*, Karlsruhe.

Fons Adriaensen. 2006b. Design of a convolution engine optimised for reverb. In *Proceedings of the International Linux Audio Conference 2006*, Karlsruhe.

Gerhard Eckel, Martin Rumori, David Pirrò, and Ramón González-Arroyo. 2012. A framework for the Choreography of Sound. In *Proceedings of the International Computer Music Conference 2012*, Ljubljana.

Angelo Farina. 2000. Simultanuous measurement of impulse response and distortion with a swept-sine technique. In *Audio Engineering Society Preprint 5093*.

Nic McBride. 2012. L'addio scontato. `http://soundcloud.com/amynicmcbride/l-addio-scontato-elyssa-vulpes`. Produced by Nic McBride, written and performed by Elyssa Vulpes. Last retrieved on 12 February 2013.

Martin Rumori, Florian Hollerweger, and Andrés Cabrera. 2010. Binaural room impulse responses for composition, documentation, virtual acoustics and audio augmented environments. In *Proceedings of the 26th VDT International Convention*, pages 670–679, Leipzig.

Pitch-class Set design in SuperCollider

Lucas SAMARUGA
Universidad Nacional de Quilmes
Roque Saenz Peña 180
Quilmes, Argentina, 1876
lsamaruga@becarios.unq.edu.ar

Oscar Pablo DI LISCIA
Universidad Nacional de Quilmes
Roque Saenz Peña 180
Quilmes, Argentina, 1876
odiliscia@unq.edu.ar

Abstract

The *Pitch-class set* theory [6] and its extensions [8] constitute an important basis for mastering multi- layered atonal composition. The *SuperCollider* [10] environment offers significant possibilities of applying this technique in the creation of abstract *Pitch-class* designs that may be used as a part of more complex algorithmic composition developments. This paper presents *pcslib-sc* a *quark* (library) for *Pitch-class set* design in *SuperCollider* and a use case in order to demonstrate its musical relevance.

Keywords

Pitch-class Set Composition, SuperCollider, Musical Composition.

1 Introduction

The *Pitch-class Set* theory uses both the combinatorial and set theory to organize the twelve *Pitch-classes* of the tempered system in Sets (*Pitch-class Set* will be from here abbreviated as PCS and *Pitch-class* as PC) in order to exploit their structural properties on atonal music composition and analysis. Although it is evident that this system was inspired on the European pre and post serial atonal music[1], it was initially developed by American composers and theorists like Milton Babbitt [1], and Allen Forte [6].

Generally speaking, the PCS theory covers three aspects. The first aspect deals with the concept of the PCS as a subset of the Universal Superset formed by the twelve Pitch-classes (also called "aggregate") and the concepts of equivalence by inversion-transposition that generate the 224 different set classes. The second defines, encodes, analyses and classifies the structural features of each set class (such as, for example, their Interval Class Vector). The third deals with the possible relations between PCSs and set classes and their significance in the musical context[2].

A latter projection of this system explores the possibilities of disposition of PCSs in the musical space producing *Combinatorial Matrices* (*Combinatorial Matrix* will be from here on abbreviated as *CM*) and creating abstract compositional designs[3].

The complexity of atonal theory makes its practical application almost impossible without the aid of computer applications. Therefore, one of the computer applications developed by the team of this project was the *pcslib* library (by Pablo Di Liscia and Pablo Cetta [2]) to be used in the *PD* environment (*Pure Data*, by Miller Puckette et al). *pcslib* is a set of "external objects" that allow the work with PCSs and CMs in the *PD* environment [4].

pcslib-sc for *SuperCollider*[4] is based on *pcslib* for *PD* with two small differences: 1) the adaption of the original library interface to a more general-purpose object-oriented language and 2) the generalisation of some functionality in relation to set theory.

Although *SuperCollider* is generally more oriented to real-time sound synthesis and algorithmic composition, its language is very useful in manipulating and analysing musical data mainly because of its dedicated library. Therefore this *quark* is intended to work with the structure generation approach of the PCS theory rather than the pattern generation, task that can be accomplished using the standard library.

This paper outlines the usage of three main resources of PCS theory, its related data structures and their combined use: *Pitch-class sets*, *Pitch-class chains* and *Pitch-class matrices*. The following discussion assumes that the reader is

[1] Mainly, the music of Arnold Schönberg, Anton Webern and Alban Berg, the three leading composers of the so-called Viennese School.

[2] For an extensive review on this specific subject, see Di Liscia [5].

[3] See Morris [7], [8].

[4] *pcslib-sc* was written by Lucas Samaruga under the supervision of Pablo Di Liscia and can be obtained at: https://github.com/smrg-lm/pcslib-sc.

aware of both the fundamentals of the PCS theory and of *SuperCollider* programming.

2 Data Structures

There are three main classes that are intended to work in combination for the elaboration of pitch structures. The *PCS* class, which defines a particular PCS (together with its properties and operations), the *PCSChain* class, which defines a *chain* of *PCSs* and its elaboration methods and the *PCSMatrix* class, which defines a *combinatorial matri*x of *PCS* (together with its particular properties, generation methods and operations).

The *PCS* class inherits from the library class *OrderedIdentitySet* not only because they share common set operations, but also because of the importance of the *Pitch-classes* order for some data calculation. *PCSChain* and *PCSMatrix* classes are more likely utility classes to work with their respective PCS theory counterpart. The former is a list and the latter represents a matrix of *PCS*. In their combined usage, chains are built from *PCS* and matrices can be built from *PCS* or chains. There is also a *SCTable* class, which holds the *Set-class* (SC)[5] table used for information retrieval from the *PCS* class.

These classes do not try to cover exhaustively all structural processes, but rather the common qualities used for compositional design. Therefore, the *PCS* class is the most complex in terms of information retrieval, operations and transformations, as they are the basic resources for further developments.

2.1 Basic Properties and Information

As stated before, the *PCS* class is the core class of the quark, it holds all the operations and properties related to the PCS theory.

A *PCS* can be built from its symbolic table name, consisting of its cardinal and ordinal numbers separated by a hyphen, like '4-16' or '5-12', but without the '*Z*' pair identifier which can be queried with the *z* method, e.g. '5-12' is written '5-Z12' in Forte's nomenclature [6]; or can be created from an array of numbers, like *PCS[0, 1, 3, 5, 6]* with the array syntax shortcut. Internally, the *PCs* are stored as an ordered set in modulo 12, so the conversion of an array of MIDI notes such as *[77, 72, 54, 49, 51, 60, 51, 48]* will result in *PCS[5, 0, 6, 1, 3]*.

Once the note numbers are stored in a *PCS* its *prime form*, *normal order*, Forte's *name*, *Z pair*, *interval-class vector*, *invariance vector* and *cyclic adjacent interval array*, can be obtained from the *SCTable* through the *PCS* instance methods. Also the *twelve-tone operators*, *relations*, *similarity* and *status* between different *PCSs* and its *prime form* are supported as basic operations aside of the inherited set operations.

2.2 Combined Use and Design Methods

PCSChain and *PCSMatrix* classes are related to the pitch design in one and two dimensions respectively (see below). They holds the structured data as higher abstractions and provides different creation, information and manipulation methods.

PCSChain is used to build chains with the procedures explained in Section 3.2. The built lists can be used as streams by the pattern library.

PCSMatrix can be built with different methods: from arrays (a free matrix), from chains (special cases were the chain is specially set, as explained in Section 3.1), from SC to build different Morris's CM types, and from twelve-tone operators [8]. Basic operations like swapping, transposition, multiplications, inversion, rotation, and information about the properties of a CM, such as *sparseness* and *fragmentation* factors and *histogram* of PC density are also provided [8]. The data of the rows and columns of the matrix can be converted back to *PCS* or used as streams as well.

3 Use case: constructing Combinatorial Matrices from chains

In this section a use case, out of the many possible using *pcslib-sc*, will be presented with the objective of showing its musical relevance. The particular subject of PCS composition addressed here will be CMs. In the next section, the basic underliying theory of CMs is briefly explained.

3.1 Introduction to the theory of Combinatorial Matrices and chains

CMs are two-dimensional arrays that hold in their vertical and horizontal dimensions PCSs of one or more SCs. The classes of those PCSs are referred to as the *norm* of a CM and are meant to produce sonic coherence with respect to some particular pitch organization. As shown in [7] and [8], there are several methods to deal with the construction of CMs, and several CMs types. The method addressed here is the construction of *chains* of PCSs.

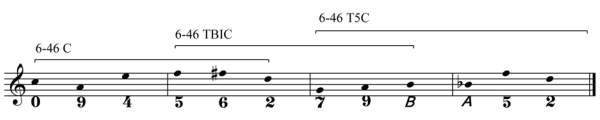


Figure 1

[5] In this paper SC stands for *Set-class*, and it should not be confused with *SuperCollider*, which is written always with no acronym.

A chain is a succession of PCSs that, being considered in adjacent pairs, form a PCS of a particular SC referred to as norm. An example of such a chain is presented in Figure 1. The ordered succession of the (unordered) PCSs: < {094} {562} {79B} {A52} >[6] constitutes a chain having the class 6-46 as its norm. The horizontal brackets show how the norm of the chain overlaps between the adjacent pairs of PCSs.

A chain with a unique norm may be taken as a basis for constructing a CM with the same norm. A chain constructed with the set class 5-15 together with its corresponding CM is shown below:

PCS chain: < {01} {268} {07} {15B} {67} {028} {16} {57B} >

Resulting CM (Table 1):

01	268		
	07	15B	
		67	028
57B			16

Table 1

As can be seen, the union of the PCS of each one of the columns and each one of the rows of the CM form a PCS of the class 5-15 (the norm of the CM). The distribution of the PCS in the resulting CM may be further improved through swapping operations[7]. One possible result would be (Table 2):

1	02	6	8
B	7	15	0
0	8	7	26
57	6	B	1

Table 2

3.2 Constructing chains

Essencially, it can be said that the method[8] for chain construction consists in connecting different transposed and/or inverted, partially-ordered versions of a PCS. Such partially ordered versions are the binary partitions of a PCS which will be termed *partitions*. For example, the Table 3 below shows the ten different partitions of a PCS of the class 5-15, and has all the information needed for constructing a chain with this set-class as norm:

If, for instance, the partition F (01|268) is selected for starting the chain, the different 2/3 transposed and/or inverted partitions having a PCS of cardinality 3 that match the PCS in the 'right part' of the starting partition are candidates for continuing the chain. When one of these candidates is selected and added to the chain, there will be new candidates to continue the chain according to the new PCS added, and the procedure is continued as explained until it is decided that the chain must be finished or when a partition that closes the chain is found[9].

5-15 {0,1,2,6,8}			
Partitions 1/4			
A	0|1268	1-1	4-16
B	1|0268	1-1	4-25
C	2|0168	1-1	4-16
D	6|0128	1-1	4-5
E	8|0126	1-1	4-5
Partitions 2/3			
F	01|268	2-1	3-8
G	02|168	2-2	3-9
H	06|128	2-6	3-5
I	08|126	2-4	3-4
J	12|068	2-1	3-8
K	16|028	2-5	3-8
L	18|026	2-5	3-8
M	26|018	2-4	3-4
N	28|016	2-6	3-5
O	68|012	2-2	3-1

Table 3

3.3 Choosing partition candidates

The explained method suggest that more than one candidate for continuing a chain may be found, depending on both the chain itself and the properties of the SC of its norm. If the norm does not change along the chain, then at least one candidate partition to continue it will exist[10]. When more than one candidate exists, one or several selection criteria must be applied. A criterion for measuring the 'qualification' of a list of candidates is to score them according their contribution of new PCs in the chain or, if the aggregate set is complete, according the distance of the PCs added

[6] The convention of representing PC 10 with an *A* and PC 11 with a *B* will be followed from here for practical reasons.

[7] Such swapping operations are documented in [7] and [8] and will not be explained here.

[8] The method is fully explained in [7] and [8]. See also [3].

[9] The possibility of finding a *partition* that may close the *chain* is explained in [7].

[10] This partition may not be musically interesting, as it is the same *partition* in reverse order,.

to their previous presentation[11]. Such criterion is formalized as:

$$score(cand_i) = \frac{\sum_{n=0}^{C_{i-1}} dist(pc_n, cand_i)}{(S)C_i}$$

EQ. 1

Where *S* is the chain size (number of positions); C_i is the cardinality of the *ith* PCS to be added; $dist(pc_n, cand_i) = S - (pos(pc_n, cand_i) + 2)$ and $pos(pc_n, cand_i)$ is the last position in which the pc_n of $cand_i$ was found (i=0 to S-1, and n=0 to C_i). If the pc_n is not found in the chain, then $pos(pc_n, cand_i)=0$.

For example, considering a chain having 3 positions (S=3), whose norm is of the class 5-3:

2 4 | 0 1 5 | 4 3 |

And the following candidates to be added with their scores (the repeated PCs are marked in Italics Bold):

candidate$_0$ = {***5 1 0***}
score = [(3-3) + (3-3) + (3-3)] / (3*3) = 0
candidate$_1$ = {B ***0 1*** }
score = [(3-0) + (3-3) + (3-3)] / (3*3) = 0.33…
candidate$_2$ = {***2 0*** B}
score = [(3-2) + (3-3) + (3-0)] / (3*3) = 0.44…
candidate$_3$ = {***5*** 7 8}
score = [(3-3) + (3-0) + (3-0)] / (3*3) = 0.66…
candidate$_4$ = {***2*** 6 7}
score = [(3-2) + (3-0) + (3-0)] / (3*3) = 0.77…
candidate$_5$ = {8 7 6}
score=[(3-0) + (3-0) + (3-0)] / (3*3) = 1

It can be easily seen that –according to this criterion- the 'best' candidate is scored by 1 and is also the one that adds three new PCs to the chain whilst the 'worst' candidate is scored by 0 and it merely repeats the PCs of the norm of the chain. Finally, it is worth noting that is not mandatory at all to select the candidate with the highest score, because there may be many other criteria by which a PCS may not be considered a 'good' candidate (just to mention one of them, the PCS candidate may belong to a SC that was decided to be excluded because of aural or stylistic reasons).

3.4 Constructing *chains* with more than one norm

It is possible to extend the already explained method for constructing chains to obtain a CM with different norms. A case having special relevance in music is presented here. If it is desired to achieve a CM whose vertical norm is always of the same SC, *x*, whilst all the horizontal norms are of different classes, *a*, *b*, *c*, *d* and *e* and supposing the cardinality of the norms is always 5, the scheme of the chain to be generated is shown in Table 4[12]:

a		***b***		***c***		***d***		***e***	
**	***	**	***	**	***	**	***	**	***
	x		***x***		***x***		***x***		

Table 4

That will be the base for the CM shown in Table 5 below:

(sc)	***x***	***x***	***x***	***x***	***x***
a	**	***			
b		**	***		
c			**	***	
d				**	***
e	***				**

Table 5

Achieving such structures is a key for mastering atonal counterpoint, since they may be very effectively used for controlling both the simultaneity and the succession of PCs and their SCs on a polyphonic musical thread.

3.5 Using *pcslib* in the *SuperCollider* environment to construct *chains* and CMs

In this section, a use case in which the construction of the chains and CMs above mentioned will be presented.

Being the following PCSs:

```
a = PCS('5-1');
b = PCS('5-21');
c = PCS('5-35');
d = PCS('5-7');
e = PCS('5-33');
x = PCS('5-12'); // 5-Z12
```

A chain may be constructed using the methods explained in Section 3.2. First a *PCSChain* is created and its initial norm is set. Then the candidates for continuing it are computed and evaluated, and a selected partition out of the candidates list is added:

```
~chain = PCSChain.new.norm_(a);
~chain.candidates(false);
~chain.addCand(7);
```

[11] This criterion was formalized by Pablo Di Liscia [3].

[12] Where each '*' represents a Pitch-class.

Next, the criteria described above to create a chain is applied. Note that it is known beforehand that the chain can be constructed, so just to execute the following *ad hoc* algorithm is needed (the resulting chain is show in Table 6):

```
[x, b, x, c, x, d, x, e].do({ arg pcs;
  ~chain.norm = pcs;
  ~chain.candidates(false);
  ~chain.candList.notEmpty.if({
    ~chain.addCand(
      ~chain.scores.indexOf(
        ~chain.scores.maxItem
      );
    );
  }, {
    "candidates for %"
    .format(pcs.name).throw;
  });
});
```

A (5-1)		B (5-21)		C (5-35)		D (5-7)		E (5-33)	
03	*124*	*67*	*3AB*	*14*	*68B*	*5A*	*349*	*68*	*02A*
	X (5-Z12)		X (5-Z12)		X (5-Z12)		X (5-Z12)		

Table 6

Now a *PCSMatrix* from the generated chain is created (shown in Table 7):

```
~matrix = PCSMatrix.fromChain(~chain);
```

set-class	4-11	X (5-Z12)	X (5-Z12)	X (5-Z12)	X (5-Z12)
A (5-1)	03	124			
B (5-21)		67	3AB		
C (5-35)			14	68B	
D (5-7)				5A	349
E (5-33)	02A				68

Table 7

and the default swapping algorithm is performed to improve the distribution of the CM:

```
~matrix.swapping;
```

Finally, the PC 9 of the fourth row is duplicated in the first column to keep all the vertical norms within the SC 5-Z12:

```
~matrix.addAt(3, 0, PCS[9]);
```

which will result in Table 8.

set-class	X (5-Z12)	X (5-Z12)	X (5-Z12)	X (5-Z12)	X (5-Z12)
A (5-1)	02	1	4		3
B (5-21)	A	7	3	B	6
C (5-35)		6	1B	8	4
D (5-7)	3**9**	4		5A	9
E (5-33)	0	2	A	6	8

Table 8

The process described so far results in a particular and coherent PCS distribution in two dimensions but it only defines the sonic potential of the pitch organization. There are many possible 'realizations' of this structural organization which will turn in different musical results. No rhythmic constrains are given except for the vertical alignment that provides a relative temporal 'window' within which the harmony can remain in norm. Other parameters of the pitch organization like register, range and timbre are not given either. All of these basic variables remain free for further development.

4 Conclusion

The *pcslib-sc* library presented in this paper is a flexible and robust tool for effectively handling the main features of atonal pitch organization. Although the structures that can be created are highly abstract, they may constitute the basis for pitched music organization. The realization of such abstract structures (i.e., the conversion of them in music) requires the setting of numerous sound features (such as register, duration, intensity and timbre among others) which are suppose to be congruent with the underlying pitch organization. *SuperCollider* is a very powerful environment for the latter accomplishment, and the objective of the *pcslib-sc* library was to add to it yet a new extension of its capacities.

5 Acknowledgments

The authors thank to *Universidad Nacional de Quilmes* (Buenos Aires, Argentina) for supporting and hosting this research.

References

[1] Milton Babbit. 1961. *Set Structure as Compositional Determinant*. Journal of Music Theory 5, no.1, USA.

[2] Oscar Pablo Di Liscia. 2012. *PCSLIB* site. https://puredata.info/Members/pdiliscia/pcslib/

[3] Oscar Pablo Di Liscia. 2012: *PCSLIB* reference. https://puredata.info/Members/pdiliscia/pcslib/Help-English.doc/view

[4] O. P. Di Liscia and P. Cetta. 2009. *Pitch-class composition in the pd environment*. XII Simposio Internacional de Computación y Música, Recife, Brasil.

[5] O. P. Di Liscia. 2011. *Medidas de similitud entre conjuntos ordenados de grados cromáticos*. Revista de Investigación

Multimedia, Vol III, IUNA, Buenos Aires. Argentina.

[6] Allen Forte. 1974. *The Structure of Atonal Music*. Yale University Press. England.

[7] Robert Morris. 1984. *Combinatorialty without the aggregate*. Perspectives of new Music. USA.

[8] Robert Morris. 1987. *Composition with Pitch-classes: A Theory of Compositional Design*. Yale University Press. USA.

[9] Puckette, Miller. 2007. *The theory and technique of electronic music*, world scientific publishing co. Pte. Ltd.

[10] Mccartney, James. 2002. *Rethinking the computer music language: supercollider*. Computer music journal, 26:4:61-68, mit press, massachussets.

Experiments with dynamic convolution techniques in live performance

Øyvind BRANDTSEGG
Music technology
NTNU - Department of Music
NO-7491 Trondheim, Norway
oyvind.brandtsegg@ntnu.no

Sigurd SAUE
Music technology
NTNU – Department of Music
NO-7491 Trondheim, Norway
sigurd.saue@ntnu.no

Abstract
This article discusses dynamic convolution techniques motivated by the musical exploration of interprocessing between performers in improvised electroacoustic music. After covering some basic challenges with convolution as live performance tool we present experimental work that enables dynamic updates of impulse responses and parametric control of the convolution process. An audio plugin implemented in the open source software Csound and Cabbage integrates the experimental work in a single convolver.

Keywords
Convolution, live processing, Csound, cross synthesis

1 Introduction

The Music Technology section at NTNU Department of Music has established the ensemble T-EMP (Trondheim ensemble for Electroacoustic Music Performance). The ensemble focuses on new modes of improvisation and music making, utilizing the possibilities inherent in contemporary electroacoustic instrumentation (Figure 1). A key objective is to blur the separation between the individual contributions from each musician and collectively develop tight-woven timbral gestures.

Through live sampling and processing the very generation of sonic material may grow out of a collaborative effort where acoustic sounds (voice, drums, etc.) are processed in real-time by digital instruments. Ultimately any sound produced could serve as source material for processing by another member of the ensemble. This concept of live interprocessing has a huge potential for timbral experimentation, but there are some very challenging issues regarding performance complexity.

Figure 1: T-EMP playing live with guests from Maynooth, Ireland.

We are experimenting with processing tools built upon the open source, platform-independent computer music software Csound [2], either written in the Csound language itself or implemented as opcodes[1] extending the language [3].

So far we have concentrated our attention on two different processing techniques: Granular synthesis and convolution. We have already presented our work on particle synthesis through the Hadron synthesizer, a digital instrument that unifies all known variants of time based granular synthesis [4, 5] . In the present paper we will focus on the musical uses of convolution, as an extension of previous work by our colleague,Trond Engum [6].

2 Convolution

Convolution is a well known signal processing technique, but the theory behind it remains unknown to most musicians [7]. The convolution of two finite sequences *x(n)* and *h(n)* of length N is defined as:

[1]An opcode is a basic Csound module that either generates or modifies signals.

$$x(n) * h(n) = \sum_{k=0}^{N-1} h(k)\, x(n-k)$$

A time-domain, direct-form implementation (similar to a FIR[2] filter) will require on the order of N^2 multiplications. We typically use segments of 2 seconds length, equivalent to 88200 sample points at 44,1 kHz sampling rate, which makes the computational complexity prohibiting. A far more efficient solution is *fast convolution* using FFT and simple multiplication in the frequency domain [8]. It does however introduce latency equal to the segment length, which is undesirable for real-time applications. Partitioned convolution reduces latency by breaking up the input signal into smaller partitions. Techniques combining partitioned and direct-form convolution can eliminate processing latency entirely [9].

There are many well-known applications of convolution, such as filtering, spatialization and reverberation. Common to them is that one of the inputs is a static impulse response (characterizing a filter, an acoustic space or similar), allocated and preprocessed prior to the convolution operation. Impulse responses are typically short and/or with a pronounced amplitude decay throughout its duration. The convolution process does not normally allow parametric real-time control.

We wanted to explore convolution as a creative sound morphing tool, using the spectral and temporal qualities of one sound to filter another. This is closely related to cross-filtering [7] or cross-synthesis, although in the latter case one usually extracts the spectral envelope of one of the signals prior to their multiplication in the frequency domain [10].

Trond Engum employed similar techniques in his artistic research project where he did real-time convolution of drums and guitars with industrial sounds such as trains and angle grinders. There are a few earlier references of related uses of convolution, starting with Barry Truax [10-13].

An important aspect of our approach is that both impulse response and input should be dynamically updated during performance. This adds significant amounts of complexity, both with respect to technical implementation and practical use. Without any real-time control of the convolution process, it can be very hard to master in live performance. Depending on the amount of overlap of spectral content between the two signals, the output amplitude may vary by several orders of magnitude. Also, when both input sounds are long, significant blurring may appear in the audio output as the spectrotemporal profiles are layered.

A possible workaround is to convolve only short fragments of the input sounds at a time, multiplying them frame by frame in the frequency domain. The drawback is that any musically significant temporal structure of the input signals will be lost in the convolution output. To capture the sound's evolution over time requires longer segments, with the possible artifact of time smearing as a byproduct. This seems to be a distinguishing factor in our approach to convolution and cross-synthesis.

This paper presents some experiments that try to overcome some of the issues above. Our aim has been to:

- create dynamic parametric control over the convolution process in order to increase playability
- investigate methods to avoid or control dense and smeared output
- provide the ability to update/change the impulse responses in real-time without glitches
- provide the ability to use two live input sounds to a continuous, real-time convolution process

The motivation behind the experiments is the artistic research within the ensemble T-EMP and specific musical questions posed within that context.

3 Experiments

The experimental work has produced various digital convolution instruments, for simplicity called *convolvers,* using Csound and Cabbage[3].

The experiments can be grouped under two main headings: Dynamic updates of the impulse response, and parametric control of the convolution process.

3.1 Real-time convolution with dynamic impulse response

From our point of view processing with a static impulse response does not fully exploit the potential of convolution in live performance. We therefore wanted to investigate strategies for dynamically updating the impulse response.

[2]FIR: Finite Impulse Response

[3] Cabbage is a toolkit for making platform-independent Csound-based audio plugins [1]. See also http://www.thecabbagefoundation.org/

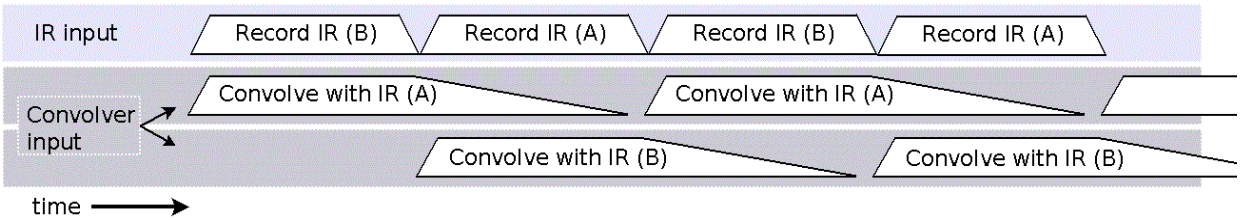


Figure 2: The stepwise updated IR buffer

A note on terminology: The term *impulse response* relates explicitly to the mathematical theory behind FIR filters or to the acoustic measurements of reverberation. Still we find it convenient to use the term (or its abbreviation IR) to signify the static input of a convolver, even when the signal no longer is the response to any impulse, strictly speaking.

3.1.1 The live sampling convolver

As a first attempt, we implemented a convolver effect where the impulse response could be recorded and replaced in real-time. This was intended for use in improvised music performance, similar to traditional live sampling, but using the live recorded audio segment as an impulse response for convolution instead. No care was taken to avoid glitches when replacing the IR in this case, but the instrument can be used as an experimental tool to explore some possibilities. Input level controls were used to manually shape the overall amplitude envelope of the sampled IR, by fading in and out of the continuous external signal. This proved to be a simple, but valuable method for controlling the timbral output.

In our use of this instrument we felt that the result was a bit too static to provide a promising basis for an improvised instrumental practice. Still, with some enhancements in the user interface, such as allowing the user to store, select and re-enable recorded impulse responses, it could be a valuable musical tool in its own right.

3.1.2 The stepwise updated IR buffer

The next step was to dynamically update the impulse response during convolution. A possible application could be to tune a reverberation IR during real-time performance. A straightforward method to accomplish this without audible artifacts is to use two concurrent convolution processes and crossfade between them when the IR needs to be modified.

When combined with live sampling convolution, the crossfade technique renders it possible to do real-time, stepwise updates of the IR, using a live signal as input. In this manner the IR is updated and replaced without glitches, always representing a recent image of the input sound.

Figure 2 illustrates the concept: Impulse responses are recorded in alternating buffers A and B from one of the inputs. Typical buffer length is between 0.5 and 4 seconds with 2 seconds as the most common. An envelope function is applied to the recorded segments for smoother convolution. The second input is routed to two parallel processing threads where it is convolved with buffer A and B respectively. The convolution with buffer A fades in as soon as that buffer is done recording. Simultaneously the tail of convolution with buffer B is faded out and that buffer starts to record.

This allows us to use two live input signals to the convolution process. There is however an inherent delay given by the buffer length. Future research will explore partitioned IR buffer updates to reduce the delay to the length of a single FFT frame.

3.2 Parametric control of the convolution process

Convolution can be very hard to control even for a knowledgeable and experienced user [7]. A fundamental goal for our experiments has been to open up convolution by providing enhanced parametric control for real-time exploration of the technique.

3.2.1 Signal preprocessing

As we have noted, the convolution process relates all samples of the IR to all samples of the input sound. This can easily result in a densely layered, muddy and spectrally unbalanced output. Various forms for preprocessing of the input signals has been proposed, such as high-pass filtering [14], compression/expansion and square-root scaling [10].

We furnished our convolver with user-controlled filtering (high-pass and low-pass) on both convolution inputs, as this can reduce the problem of dense, muddy output considerably.

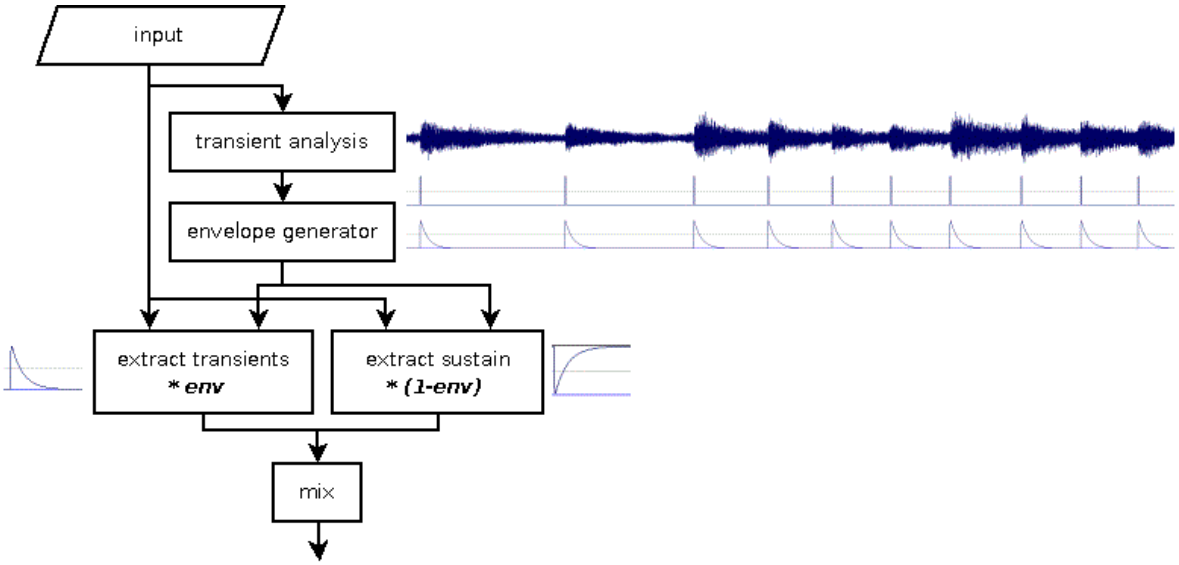


Figure 3: Splitting transient and sustained parts of an input signal

The point is to provide dynamic control over the "degree of spectral intersection" [10].

3.2.2 The transient convolver

As another strategy for controlling the spectral density of the output, while still keeping with the basic premise that we want to preserve the temporal characteristics of the IR sound, we split both the IR and the input sound into transient and sustained parts. Convolving with an IR that contains only transients will produce a considerably less dense result, while still preserving the large-scale spectral and temporal evolution of the IR.

The splitting of the transient and sustained parts was done in the time domain by detecting transients and generating a transient-triggered envelope (see Figure 3). The transient analysis can be tuned by a number of user-controlled parameters. The sustained part was extracted by using the inverted transient envelope. Hence the sum of the transient and sustained parts are equal to the original input.

Transient splitting enables parametric control over the density of the convolution output, allowing the user to mix in as much sustained material as needed. It is also possible to convolve with an IR that has all transients removed, providing a very lush and broad variant.

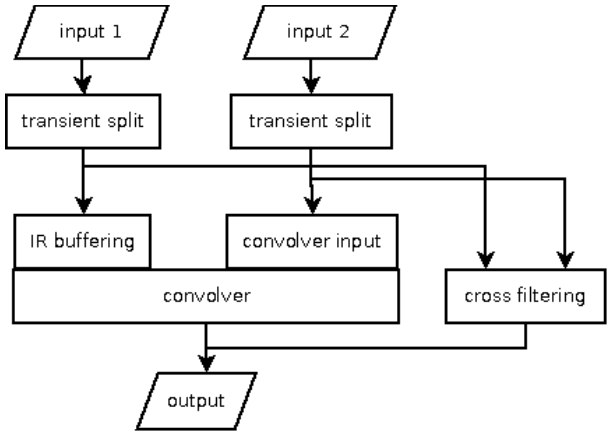


Figure 4: Convolver with transient split and stepwise update IR buffer. Straightforward cross-filtering is added as reference.

3.3 Combining the results

Finally, the various convolution experiments outlined above were combined into a single convolver[4]. It works with two live audio inputs: one is buffered as a stepwise updated IR and the other used as convolution input (see Figure 4).

The IR can be automatically updated at regular intervals. The sampling and replacement of the IR can also be relegated to manual control, as a way of "holding on" to material that the performer finds musically interesting or particularly effective.

Each of the two input signals can be split into transient and sustained parts, and simple low-pass and high-pass filters are provided as rudimentary methods for controlling spectral spread.

Straightforward cross-filtering, continuously multiplying the spectral profile of the two inputs frame by frame, was also added to enable direct comparison and further experimentation. As should be evident from Figure 5 the user interface

[4] The Csound code for this convolver is available at http://folk.ntnu.no/oyvinbra/LAC2013/, ready to be compiled into a VST using Cabbage.

Figure 5: The convolver user-interface

provides a great deal of parametric control of this convolver.

4 Conclusion and further work

We have implemented a number of variations of convolution as an attempt to overcome limitations inherent in convolution as a music processing technique. The context for our experimental work is musical objectives growing out of improvisational practice in electroacoustic music. Preliminary tests show that some of the limitations have been lifted by giving real-time parametric control over the convolution process and the density of its output, and by allowing real-time updates of the IR.

In practical use, the effect is still hard to control. This relates to the fact that, with real-time stepwise updates of the IR, the performer does not have detailed control over the IR buffer content. The IR may contain rhythmic material that are offset in time, creating offbeat effects or other irregular rhythmic behavior. With automatic IR updates the performer does not have direct and precise control over the timing of IR changes. Instead the sound of the instrument will change at regular intervals, not necessarily at musically relevant instants.

A possible way of controlling rhythmic consistency would be to update the IR in synchrony with the tempo of the input material, for instance so that the IR always consists of a whole measure or beat and that it is replaced only on beat boundaries. Another proposal would be to strip off non-transient material at the start of the IR, so that the IR would always start with a transient. This is ongoing work.

We hope to present our convolver live at the conference.

5 Acknowledgements

Our thanks goes to the performers of T-EMP who in addition to author Øyvind Brandtsegg are: Tone Åse, Ingrid Lode, Carl Haakon Waadeland, Bernt Isak Wærstad and in particular Trond Engum, who has advocated creative uses of convolution for many years now. We would also like to thank the Norwegian Artistic Research Programme for supporting T-EMP as a performing laboratory.

References

[1] R. Walsh. 2011. Audio Plugin development with Cabbage. Proceedings of Linux Audio Conference, pages 47-53.

[2] Csound. See http://www.csounds.com/.

[3] V. Lazzarini. 2005. Extensions to the Csound Language: from User-Defined to Plugin Opcodes and Beyond. LAC2005 Proceedings, pages 13.

[4] Ø. Brandtsegg, S. Saue, T. Johansen. 2011. Particle synthesis–a unified model for granular synthesis. Proceedings of the 2011 Linux Audio Conference(LAC'11).

[5] Ø. Brandtsegg, S. Saue. 2011. Performing and composing with the Hadron Particle Synthesizer. Forum Acusticum, Aalborg, Denmark.

[6] T. Engum. Real-time control and creative convolution. Proceedings of the International Conference on New Interfaces for Musical Expression}, pages 519-22.

[7] C. Roads. 1997. Sound transformation by convolution. In C. Roads, A. Piccialli, G. D. Poli, S. T. Pope, editors, *Musical signal processing*, pages 411-38. Swets & Zeitlinger.

[8] R. Boulanger, V. Lazzarini. 2010. *The Audio Programming Book*, The MIT Press.

[9] W. G. Gardner. 1995. Efficient Convolution without Input-Output Delay. *J Audio Eng Soc*, 43(3), pages 127.

[10] Z. Settel, C. Lippe. 1995. Real-time musical applications using frequency domain signal processing. Applications of Signal Processing to Audio and Acoustics, 1995, IEEE ASSP Workshop on, pages 230-3 IEEE.
[11] B. Truax. 2005. Music and science meet at the micro level: Time-frequency methods and granular synthesis. *Acoustical Society of America Journal*, 117pages 2415-6.
[12] R. Aimi. 2007. Percussion instruments using realtime convolution: Physical controllers. Proceedings of the 7th international conference on New interfaces for musical expression, pages 154-9 ACM.
[13] D. Merrill, H. Raffle, R. Aimi. 2008. The sound of touch: physical manipulation of digital sound. Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems, pages 739-42 ACM.
[14] T. Erbe. 1997. SoundHack: A Brief Overview. *Computer Music Journal*, 21(1), pages 35-8.

Creating LV2 Plugins with Faust

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

Abstract

The faust-lv2 project aims to provide a complete set of LV2 plugin architectures for the Faust programming language. It currently implements generic audio and MIDI plugins with some interesting features such as Faust MIDI controller mapping, polyphonic instruments with automatic voice allocation and support for the MIDI tuning standard. You can use these architectures to quickly turn Faust programs into working LV2 audio effects and instrument plugins, ready to be run with LV2-capable DAWs such as Ardour and Qtractor. The plugin architectures and some helper scripts are now also available in the Faust distribution, and the Faust online compiler supports these as well.

Keywords

Faust, LV2, plugins, audio, MIDI.

1 Introduction

Most Linux audio users will be familiar with David Robillard's LV2 [4], the successor of the venerable LADSPA plugin standard. LV2 has been supported by major Linux DAWs such as Ardour and Qtractor for quite some time, and version 1.0 of the standard has been released in 2012, so that LV2 host and plugin authors now have a stable specification to base their work on. LV2 is much more complex than LADSPA, but it is also much more capable. In particular, it supports both audio and MIDI plugins and can thus be used to develop audio effects as well as software instruments. One of LV2's strong points is that it is extensible, so that new extensions for various special needs can be developed and deployed in LV2 hosts with (relative) ease. This makes LV2 very flexible. A number of both open source and proprietary suites of LV2 plugins have been developed or ported over to LV2, such as Calf, CAPS, TAL, drowAudio, Loomer and linuxDSP, so that Linux audio users now have a variety of high-quality plugins available to them. Nevertheless, compared to other plugin standards such as Steinberg's VST, the number of available plugins is still quite small.

The goal of the faust-lv2 project is to bring LV2 to Faust, Grame's funtional DSP programming language [3], so that LV2 plugins can be developed more easily. The interface is implemented in terms of corresponding LV2 architectures for Faust. At present two architecture (C++) files are provided, one for ordinary audio (effect type) plugins and one for polyphonic MIDI (instrument type) plugins.

We should note here that this is not the first time that LV2 has been targeted by Faust developers; projects such as Sampo Savolainen's Foo YC-20 organ emulation [5] or the Guitarix tube amplifier simulation by Hermann Meyer and others [1] utilize Faust as well. However, the goal of faust-lv2 is different. The architectures provided by faust-lv2 are completely generic and thus allow you to compile *any* Faust source and get a working LV2 plugin from it. There is a growing collection of Faust programs available, ranging from simple routing and panning plugins to sophisticated sound effects and instruments such as the Faust Synthesis Toolkit [2]. faust-lv2 enables you to use all of these in your favourite LV2 host without any further ado. Many sources will work out of the box, while others may require a few edits to make the Faust program behave nicely as an LV2 plugin. And of course faust-lv2 now also provides a convenient way to develop new sound modules and instruments in Faust and deploy them as LV2 plugins (in fact, recent posts to the Linux audio mailing lists seem to indicate that faust-lv2 is already being used that way in some projects).

This paper gives a brief overview of faust-lv2 and how you can use it to compile your own plugins. We also discuss major features and current limitations of the software and give an outlook on future work. We don't go into all the gory

details here, however, so the interested reader should refer to the extended version of this paper at the faust-lv2 website for more information:

`http://faust-lv2.googlecode.com`

2 Installation and basic usage

Chances are that if you are running a recent Faust version (Faust 0.9.58 will do) then faust-lv2 is already included, so you don't have to install anything extra. faust-lv2 is also fully supported by the Faust online compiler, so you can just drop your dsp sources there and, after a few clicks, grab your ready-made LV2 plugin bundles, even without installing Faust on your computer.

Another option is to install faust-lv2 from the source distribution tarball available at the faust-lv2 project website. As a bonus this also gives you a few plugin examples you can start playing with. The package also demonstrates how you can put together your own LV2 plugin collections ready to be compiled from source. faust-lv2 is distributed as free and open-source software, licensed under the LGPL. More detailed information about the source package can be found at the project website. Briefly, if you go this route then you can compile and install faust-lv2 as follows:

```
./waf configure && ./waf && sudo ./waf
install
```

This will install both the Faust architecture files and the sample plugins under `/usr/local`, so that you can compile your own plugins and try the sample plugins in your favourite DAW. You can also just drop your Faust dsp files into the `effects` and `synths` subfolders and have them compiled and installed when running `waf`.

Alternatively, if you already have Faust installed, you can also employ two convenience scripts `faust2lv2` and `faust2lv2synth` distributed with recent Faust versions, which make the creation of LV2 bundles very easy; this is also the approach shown in the remainder of this paper.

If you want to learn exactly how this works, you should note that compiling LV2 plugins using Faust is a bit more involved than usual. This is because LV2 plugins aren't mere shared library (`.so`) files, but collections of libraries and RDF description files in Turtle syntax (`.ttl`) in their own directory. This is also known as an *LV2 bundle*. The precise steps needed to create plugin bundles with Faust are described in the extended version of this paper and in the faust-lv2 online documentation, both available at `http://faust-lv2.googlecode.com`. Developers may want to study these if they want to come up with their own build systems for compiling Faust LV2 plugins.

3 Supported plugin types

At present, faust-lv2 supports two types of plugins: the usual audio processing plugins as well as MIDI-driven software synthesizer plugins. Together these should cover most common uses in Linux audio software.

3.1 Audio plugins

Audio plugins can be added to the signal pathway in a DAW in order to realize audio effects such as amplification, panning, filtering, distortion, chorus, reverbation, etc. They are implemented by the `lv2` architecture. Please check the `lv2.cpp` file in the faust-lv2 distribution or the Faust library directory (`/usr/local/lib/faust` or `/usr/lib/faust` in most installations) if you are interested in how exactly these plugins are implemented. Plugins created with the `lv2` architecture provide the following basic features:

- Audio inputs and outputs of the Faust dsp are made available as LV2 audio input and output ports.
- Faust controls are made available as LV2 control ports with the proper label, initial value, range and (if supported by the host) step size. Both "active" (input) and "passive" (output) Faust controls are supported and mapped to the corresponding LV2 input and output ports, but note that most LV2 hosts don't provide access to LV2 control output ports (a.k.a. Faust passive controls) at this time.
- If the dsp defines any controls with corresponding MIDI mappings (`midi:ctrl` attributes in the Faust source), the plugin also provides an LV2 MIDI input port and interprets incoming MIDI controller messages accordingly.
- Plugin name, description, author and license information provided as metadata in the Faust source are translated to the corresponding fields in the LV2 manifest of the plugin.

The architectures also recognize the following Faust control metadata and set up the LV2 control port properties accordingly. Note that some of these properties rely on extensions which may not be supported by all LV2 hosts. Please refer to the LV2 documentation for a closer description of these options.

- The `unit` attribute (e.g., `[unit:Hz]`) in the Faust source is translated to a corresponding LV2 `unit` attribute. The host may then display this information in its GUI rendering of the plugin controls.
- LV2 scale points can be set with the `lv2:scalePoint` (or `lv2:scalepoint`) attribute on the Faust side. The value of this attribute in the Faust source takes the form of a list of pairs of descriptive labels and corresponding values, for instance:

  ```
  toggle = button(
  "trigger [lv2:scalepoint on 1 off 0]");
  ```

  The host may then display the given scale points with a descriptive label in its GUI.
- The `lv2:integer` attribute in the Faust source is translated to the `lv2:integer` LV2 port property, so that the control may be shown as an integer-only field in the host's GUI.
- The `lv2:hidden` or `lv2:notOnGUI` attribute maps to the LV2 `notOnGUI` port property, so that hosts honoring this property may suppress the display of this control in their GUI.

It is worth noting here that the special treatment of MIDI controllers and metadata in the Faust source can also be turned off, either with corresponding waf configure options (when using the faust-lv2 source package) or by disabling corresponding conditional compilation symbols in the `lv2.cpp` file.

For instance, consider the `chorus.dsp` example in the faust-lv2 source (cf. Fig. 1).

Compiling this program to an LV2 bundle can be done conveniently with the `faust2lv2` helper script included in recent Faust versions:

```
faust2lv2 chorus.dsp
```

This leaves a subfolder named `chorus.lv2` with the LV2 plugin (`.so` file) itself and the requisite `.ttl` files in the current directory. You can just copy this folder to `/usr/lib/lv2`, `/usr/local/lib/lv2` or any other directory on your `LV2_PATH` to have the plugin recognized by your DAW or other LV2 host program.

Besides the usual options supported by Faust compilation scripts, `faust2lv2` also understands the following target-specific options:

- `-nometa`: Normally, metadata in the Faust program (plugin description, author information, etc., as shown in the chorus example) will be translated to corresponding LV2 properties so that this data becomes available in the LV2 plugin host. When using the `-nometa` option, the metadata from the Faust source is ignored, which may be useful if you prefer to specify the corresponding information by manually editing the `manifest.ttl` file in the plugin bundle.
- `-nomidicc`: If you specify this, the plugin will not process any MIDI control data. This might be useful if the built-in MIDI control processing of the plugin gets in the way of the plugin host's own MIDI controller and automation features.
- `-uri-prefix` *URI*: This option specifies the URI prefix of the plugin. The argument must be a valid URI designation which, together with the name of the plugin uniquely identifies the plugin; please check the LV2 documentation for details. By, default, the URI prefix `http://faust-lv2.googlecode.com` will be used. You may want to replace this with the URL of the website where your plugins can be downloaded, or any other (possibly abstract) URI prefix which uniquely identifies your plugins so that they don't clash with other LV2 plugins installed on your system.
- `-dyn-manifest`: This enables dynamic manifests in the plugin, see Section 4.2 below for details. Note that to make this work, your LV2 host must support dynamic manifests. (For hosts like Ardour and Qtractor which are based on David Robillard's `lilv` library, you'll have to make sure that `lilv` was built with the `--dyn-manifest` waf configure option.)

Note that the faust-lv2 source package supports similar (as well as a bunch of other) options when configuring the package; run `./waf`

```
declare name "chorus";
declare description "stereo chorus effect";
declare author "Albert Graef";
declare version "1.0";

import("music.lib");

level   = hslider("level", 0.5, 0, 1, 0.01);
freq    = hslider("freq", 3, 0, 10, 0.01);
dtime   = hslider("delay", 0.025, 0, 0.2, 0.001);
depth   = hslider("depth", 0.02, 0, 1, 0.001);

tblosc(n,f,freq,mod)    = (1-d)*rdtable(n,waveform,i&(n-1)) +
                          d*rdtable(n,waveform,(i+1)&(n-1))
with {
        waveform        = time*(2.0*PI)/n : f;
        phase           = freq/SR : (+ : decimal) ~ _;
        modphase        = decimal(phase+mod/(2*PI))*n;
        i               = int(floor(modphase));
        d               = decimal(modphase);
};

chorus(dtime,freq,depth,phase,x)
                        = x+level*fdelay(1<<16, t, x)
with {
        t               = SR*dtime/2*(1+depth*tblosc(1<<16, sin, freq, phase));
};

process                 = vgroup("chorus", (left, right))
with {
        left            = chorus(dtime,freq,depth,0);
        right           = chorus(dtime,freq,depth,PI/2);
};
```

Figure 1: Faust program `chorus.dsp`.

`configure --help` in the faust-lv2 source directory to get a list of these.

3.2 MIDI plugins

faust-lv2 also fully supports instrument plugins a.k.a. software synthesizers, which can be employed as the head of the synth-effects chain in a MIDI track of your DAW. These are implemented by a separate `lv2synth` architecture.

Besides all of the features of the audio plugins described above, plugins created with the `lv2synth` architecture also provide the necessary logic to drive a polyphonic synth with automatic voice allocation. To make this work, the Faust dsp must be able to function as a monophonic synth which provides controls named `freq`, `gain` and `gate` to set the pitch (as a frequency in Hz), velocity (as a normalized value in the range 0...1) and gate (as a binary 0 or 1 value) of a note, respectively; the example below illustrates how this is done. The desired maximum number of voices can be configured with the `--nvoices` option (when using the faust-lv2 source package) or by setting the `NVOICES` macro in the `lv2synth.cpp` file accordingly. The plugin will manage at most that many instances of the Faust dsp. The actual number of voices can be changed dynamically from 1 to `NVOICES` with a special `Polyphony` control provided by the plugin.

This kind of plugin always provides a MIDI input port and interprets incoming MIDI note and pitch bend messages, as well as a number of General MIDI standard controller and system exclusive (sysex) messages, as detailed below. By default, the synth units have a pitch bend range of ± 2 semitones (General MIDI default) and are tuned in equal temperament with A4 at 440 Hz. These defaults can be adjusted as needed using some of the controller and sysex messages described below.

- The "all notes off" (123) and "all sounds off" (120) MIDI controllers stop sounding notes on the corresponding MIDI channel.
- The "all controllers off" (121) MIDI controller resets the current RPN ("registered parameter number") and data entry controllers on the corresponding MIDI channel (see below).
- The registered parameters (RPNs) 0 (pitch bend range), 1 (channel fine tuning) and 2 (channel coarse tuning) can be used to set the pitch bend range and fine/coarse master tuning on the corresponding MIDI channel in the usual way, employing a combination of the RPN (101, 100) and data entry controller pairs (6 and 38, as well as 96 and 97). Please check the MIDI specification for details.
- Universal realtime and non-realtime scale/octave tuning messages following the MIDI Tuning Standard (MTS), Section MIDI Tuning Scale/Octave Extensions, can be used to set the synth to a given octave-based tuning specified as cent offsets relative to equal temperament, which is repeated in every octave of the MIDI note range 0...127. Please check Section 4.1 below for further details.

For instance, consider the `organ.dsp` example from the faust-lv2 distribution (cf. Fig. 2).

Note the `freq`, `gain` and `gate` controls which turn this Faust dsp into a monophonic synthesizer. Polyphony with automatic allocation of up to `NVOICES` voices is implemented in the plugin architecture. Also note the `midi:ctrl 10` attribute in the label of the `pan` control. This is Faust control metadata which denotes that MIDI controller 10 (the MIDI pan position controller) should be associated with this control value. The plugin architecture will add a MIDI input port and the required MIDI controller processing to the plugin in order to implement this. (Whether your LV2 host actually passes such MIDI controller messages to the plugin depends on the host, though.)

Compiling the plugin works as with audio plugins, using `faust2lv2synth` in lieu of `faust2lv2`:

```
faust2lv2synth organ.dsp
```

You'll get an `organ.lv2` folder which you simply copy to your LV2 library directory to have the plugin recognized. In addition to the target-specific options recognized by `faust2lv2`, `faust2lv2synth` also lets you specify the desired maximum number of voices with the `-nvoices` option which takes the desired number of voices as its argument (the default is 16). In principle, any positive integer can be specified here, but the feasible range will of course depend on how much cpu power you have to spare.

Figure 3 shows the `organ.lv2` instrument along with some other Faust-generated LV2 plugins running in Qtractor.

4 Special features and limitations

In this section we discuss some notable features and limitations of the Faust LV2 implementation. The generated plugins should work with any LV2 1.0 compatible host which supports either the `urid` or the older `uri-map` extension (most if not all LV2 hosts will have this). MIDI input requires a host capable of delivering MIDI events through LV2's `event` extension. faust-lv2 also supports the `dynmanifest` extension (see Section 4.2 below), but this is an optional feature which is by no means required for proper operation of the plugins.

4.1 MIDI tunings

The MTS support of instrument plugins mentioned in the previous section calls for a more detailed explanation. The general format of the supported MTS messages is as follows (using hexadecimal notation):

```
f0 7f/7e id 08 08/09 bb bb bb tt ... tt f7
```

Note that the `f0 7f` and `f0 7e` headers are used to denote a universal realtime and non-realtime sysex message, respectively, and the final `f7` byte terminates the message. Both types of messages will take effect immediately, but the realtime form will also change the frequencies of already sounding notes. The device id can be any 7-bit value from `00` to `7f` and will be ignored, so that the unit will always respond to these messages, no matter which device id is specified. The following `08` id denotes an MTS message, followed either by the `08` subid to denote 1-byte, or the `09` subid to denote 2-byte encoding (see below).

The `lv2synth` architecture keeps track of separate tunings for different MIDI channels. The three `bb` bytes together specify the bitmask of MIDI channels the message applies to, most significant byte first; the bitmask `03 7f 7f` thus sets the tuning for all MIDI channels, while the

```
declare name "organ";
declare description "a simple additive synth";
declare author "Albert Graef";
declare version "1.0";

import("music.lib");

// control variables

vol     = hslider("vol", 0.3, 0, 10, 0.01);     // %
pan     = hslider("pan [midi:ctrl 10]", 0.5, 0, 1, 0.01); // %
attack  = hslider("attack", 0.01, 0, 1, 0.001); // sec
decay   = hslider("decay", 0.3, 0, 1, 0.001);   // sec
sustain = hslider("sustain", 0.5, 0, 1, 0.01);  // %
release = hslider("release", 0.2, 0, 1, 0.001); // sec
freq    = nentry("freq", 440, 20, 20000, 1);    // Hz
gain    = nentry("gain", 0.3, 0, 10, 0.01);     // %
gate    = button("gate");                       // 0/1

// relative amplitudes of the different partials

amp(1)  = hslider("amp1", 1.0, 0, 3, 0.01);
amp(2)  = hslider("amp2", 0.5, 0, 3, 0.01);
amp(3)  = hslider("amp3", 0.25, 0, 3, 0.01);

// additive synth: 3 sine oscillators with adsr envelop

partial(i) = amp(i+1)*osc((i+1)*freq);

process = sum(i, 3, partial(i))
  * (gate : vgroup("1-adsr", adsr(attack, decay, sustain, release)))
  * gain : vgroup("2-master", *(vol) : panner(pan));
```

Figure 2: Faust program `organ.dsp`.

bitmask `00 00 01` only affects the tuning of the first MIDI channel.

The `tt` bytes specify the tuning itself, as a sequence of 12 tuning offsets for the notes C, C♯, D, etc., thru B. In the one-byte encoding (subid `08`), each tuning offset is a 7 bit value in the range `00`...`7f`, with `00`, `40` and `7f` denoting -64, 0 and +63 cents, respectively. Thus equal temperament is specified using twelve `40` bytes, and a quarter comma meantone tuning could be denoted, e.g., as `4a 32 43 55 3d 4e 36 47 2f 40 51 39`. The two-byte encoding (subid `09`) works in a similar fashion, but provides both an extended range and better resolution. Here each tuning offset is specified as a 14 bit value encoded as two data bytes (most significant byte first), mapping the range 0...16384 to -100..+100 cents with the center value 8192 (`40 00`) denoting 0 cents. Please check the MMA's MIDI Tuning Standard document for details.

Using these messages you can tune a Faust synth in any octave-based temperament you like, provided that your DAW supports sending sysex messages to LV2 instrument plugins. (Qtractor allows you to enter the sysex messages in its "Buses" dialog. Ardour 3 doesn't support editing sysex messages yet, but it is still under development, so there is hope that this will be fixed before the final release.) A large repository of historical and contemporary microtonal tunings is available on the website of the Scala program; writing a little script to convert the Scala tuning files to binary sysex files in one of the formats described above should be a fun exercise for Linux audio developers.

4.2 Dynamic manifests

Plugins created with faust-lv2 support the LV2 dynamic manifest extension, so that all requisite information about the plugin's name, author, ports, etc. can also be included in the plugin module (`.so` file) itself. This also cuts down the compilation time since the manifest doesn't have to be generated from the plugin executable

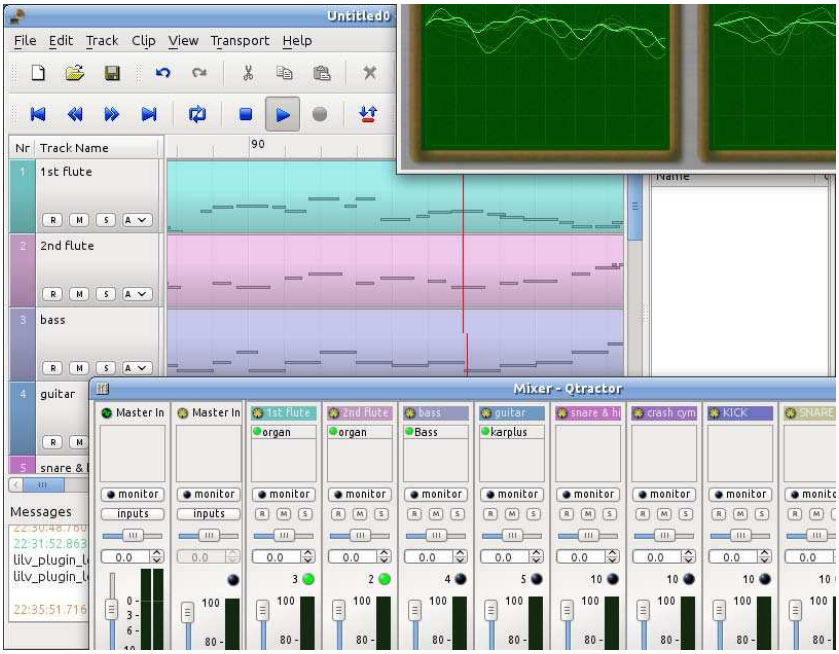


Figure 3: faust-lv2 plugins running in Qtractor.

beforehand.

Note that in order to provide better compatibility with current LV2 hosts, which usually don't have this extension enabled, this feature isn't used by default in the provided build scripts. But you can select it by configuring faust-lv2 with the `--dyn-manifest` option, when using the faust-lv2 source package, or with the `-dyn-manifest` option of the `faust2lv2` and `faust2lv2synth` scripts included in recent Faust versions.

4.3 GUIs

One major limitation of faust-lv2 is that it does not support custom plugin GUIs in the current version. This might be added in the future, but for the time being you'll have to rely on the LV2 host to display a GUI for the control elements. Both Ardour and Qtractor do a reasonably good job at this. (However, the hierarchical layout of GUI controls prescribed by the Faust source is lost in the generic plugin GUIs provided by LV2 hosts.)

5 Future work

While the LV2 plugin implementation of the Faust LV2 architectures is fully functional and reasonably complete already, there are ways in which they could be further improved. Some items which are worth further consideration are listed below.

- Add improvements for smoother playback. In particular, the polyphony control provided by `lv2synth.cpp` is fairly disruptive right now, as it simply resets all voices each time the control changes.
- Add custom plugin GUIs which honor the hierarchical GUI layout defined in the Faust source. Corresponding code is readily available in other Faust architectures such as jack-gtk and jack-qt, but would need to be integrated with the LV2 architectures and the LV2 GUI extension.
- Update the architectures so that they employ the new atom-based interface for MIDI input instead of the older (and now deprecated) LV2 Event extension.
- Add support for the new LV2 Time extension, which provides transport information such as the current position, tempo and time signature to a plugin.
- Implement MIDI output for passive Faust controls. It's unclear if and how existing LV2 hosts would process such data, however, so there's still some research to be done there.

Besides these, LV2's extensible nature might call for completely new plugin types in the future. While the audio and instrument plugin types implemented by faust-lv2 seem to cover the requirements of the current generation of DAWs,

it is good to know that Faust's and LV2's modular nature will make it easy to support new types of audio applications when they emerge.

References

H. Meyer, A. Degert, and P. Shorthose. Guitarix tube amplifier simulation for Jack/ Linux. `http://guitarix.sourceforge.net`, 2013.

R. Michon and J. O. Smith. Faust-STK: a set of linear and nonlinear physical models for the Faust programming language. In G. Peeters, editor, *Proceedings of the 11th International Conference on Digital Audio Effects (DAFx-11)*, pages 199–204, Paris, 2011. IRCAM.

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.

D. Robillard. LV2 1.2.0 Specifications. `http://lv2plug.in/ns/`, 2013.

S. Savolainen. Emulating a combo organ using Faust. In *Proceedings of the 9th International Linux Audio Conference*, pages 21–29, Utrecht, 2010. Hogeschool voor de Kunsten.

Towards a live-electronic setup with a sensor-reed saxophone and Csound

Alex Hofmann, Alexander Mayer, and **Werner Goebl**
Institute of Music Acoustics, University of Music and Performing Arts Vienna, Austria

hofmann-alex@mdw.ac.at

Abstract

This paper presents a setup to pick up saxophone reed vibrations directly, in an attempt to monitor the saxophone signal without risky feedbackloops despite drastic dynamic manipulations. We prepared synthetic saxophone reeds with strain gauge sensors and proposed a circuit to connect the sensor reed to a line-level soundcard input. Furthermore, we discussed possible open-source software to emulate classic stompbox effects. Finally, we presented a Csound instrument design, that allows on-the-fly signal routing between multiple effects in an ongoing live performance.

1 Introduction

On the electric guitar, the coil pickup converts the vibrations of the steel strings directly into an alternating current through electromagnetic induction [Campbell et al., 2004]. This makes it possible to amplify the instrument without risking feedback loops between a microphone and the speaker system [Lemme, 1994]. To enrich the sound possibilities of the electric guitar, musicians and instrument makers developed several circuits to modify the clean pickup signal. These circuits were build into little boxes, which the performer turns on and off by foot (stompboxes) during live performance [Bacon, 1984; Collins, 2009].

With todays processing speed of portable computers, digital signal processing can replace analog signal modifiers on stage [Noble, 2009; Boulanger and Lazzarini, 2011]. Besides stand alone guitar effect software (e.g., on Linux: Rakarrack, Guitarix), computer music languages like Csound or Supercollider provide ready-made signal processing modules (Csound opcodes, SuperCollider Ugens) to rebuild classic stompbox effects [Mikelson, 2000]. New effects can easily be designed and tested live with such toolboxes [Ervik and Brandtsegg, in press; Waerstad, 2010].

When playing wind instruments, like the saxophone, the sound source is usually picked up with a microphone. Drastic, dynamic modifying sound effects like distortion or resonating filters are difficult to apply to a microphone signal in a live situation. The high volume levels on stage lead to an increased risk of microphone-speaker feedback loops, which might disturb the performance.

Sensor saxophone reeds were developed in acoustic research to investigate single reed behavior under real playing conditions [Hofmann et al., 2012a]. In this paper, we will discuss applications for such sensor-reeds within an electronic live performance setup with the open source audio software Csound.

2 Method

2.1 Saxophone reed pickup

We attached a strain gauge sensor to a synthetic alto saxophone reed, to directly capture the vibrations of the reed during performance (Figure 1). A detailed description of how to prepare synthetic reeds with strain gauge sensors is given in [Hofmann et al., 2013].

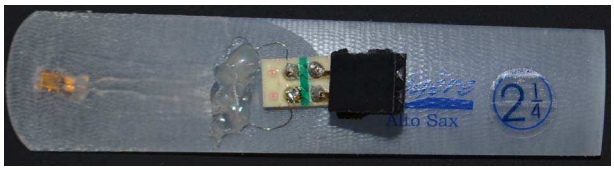


Figure 1: Synthetic alto-saxophone reed with 2 mm strain gauge attached

The standard measurement setup foresees the strain gauge as one resistor in a quarter Wheatstone bridge [Scott and Owens, 1989]. The range of the resulting signal depends on the supply voltage of the bridge (+3V, -3V). An instrumentation amplifier (INA 126) is used to adjust the signal amplitude of the differential bridge voltage. A first order RC filter removes the DC offset. Figure 2 (C) depicts a circuit with two operational amplifiers (LM358N) to

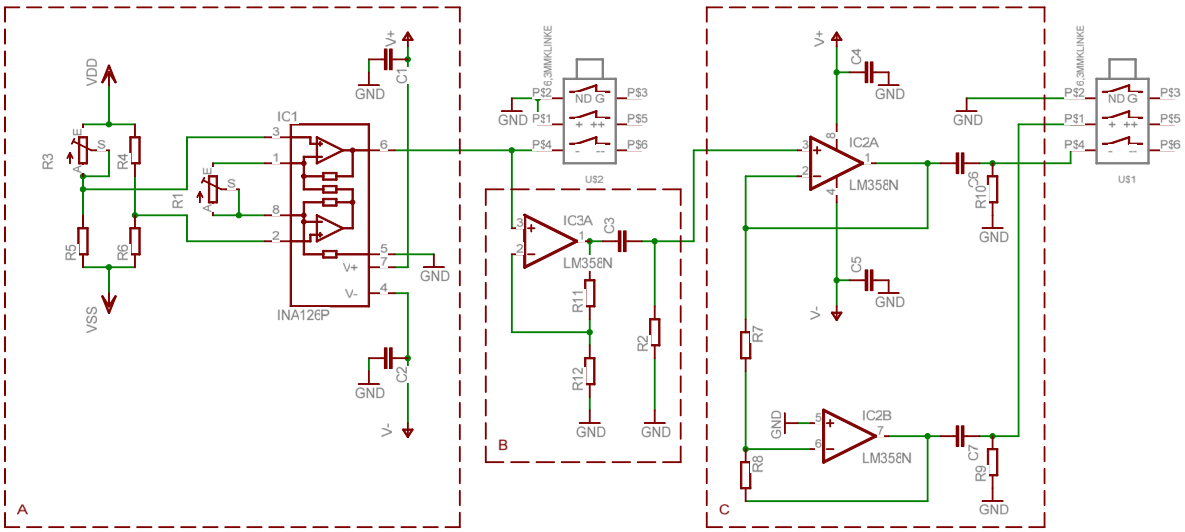


Figure 2: Circuit with a quarter Wheatstone bridge and an instrumentation amplifier (A), a RC filter (B), and two operation amplifiers for symmetric signal output (C)

gain symmetric output. This allows direct plugging into the symmetric *Line-Input* of a professional soundcard for A/D conversion.

2.2 Effect Setup

We decided to build signal modifying sound effects with Csound, a well documented, flexible, open source, platform independent audio programming language [Heintz et al., 2011].

We will encapsulate the signal modifiers (effects) as independent *Csound Instruments* and use the *subinstr* opcode to enable signal rerouting in real-time, during performance.

2.3 Signal Flow

The current system is intended for mono sound input and stereo sound output. Two global audio rate variables (gaOutL, gaOutR) are keeping the main output of the system. A simple audio input to output patch looks like shown in Figure 3.

The main controller Csound Instrument (instr 1) is used to receive MIDI note messages during performance. Received MIDI notes determine which sub-instrument's effect is processed. A sub-instrument reads audio samples from the global audio variables (gaOutL, gaOutR), modifies them and then overwrites them.

Usually Csound instruments are calculated by the order of their instrument number. During one audio processing cycle, instrument 10 is always calculated before instrument 30, independent of their sequence in the Csound score. The use of sub-instruments (*subinstr*) makes it possible to start multiple versions of instrument 1 by MIDI note messages. Csound treats each instance like a voice of instrument 1. According to the received MIDI note number, instrument 1 is holding a different sub-instrument each time. This method allows to shift calculations of sub-instrument 30 before sub-instrument 10 within the current audio cycle.

```
<CsInstruments>
gaOutL init 0.0
gaOutR init 0.0

instr 1 ;Main Controller Instrument
  inote  notnum
  ichn midichn
   if inote == 10 then
     gaOutL, gaOutR subinstr 10, 0, -1
   elseif inote == 30
     gaOutL, gaOutR subinstr 30, 0, -1
   elseif inote == 31
     gaOutL, gaOutR subinstr 31, 0, -1
   elseif ..
   endif
endin
```

2.3.1 Input and Output

In our setup, the soundcard input is also implemented as a sub-instrument (instr 10). For example, this can be useful to loop an audio sequence by routing the soundcard input into a delay (Figure 4, top). When the soundcard input instrument is later released and triggered

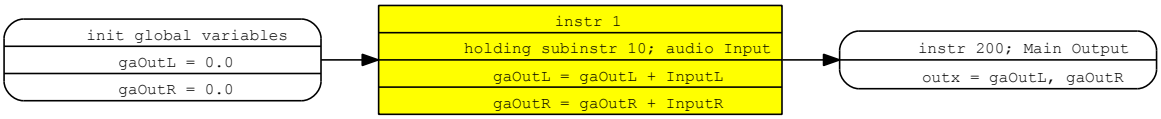


Figure 3: Signal flow of a simple input-to-output routing

again, it is automatically placed after the delay and you can play over the recorded loop without changing it (Figure 4, bottom).

```
instr 10 ;Soundcard Input Instrument
  aIn inch 1
  outs gaOutL + aIn, gaOutR + aIn
endin


instr 30  ;Loop-Delay
  aDelayL delayr 3 ;init 3 sec. delayline
   aWetL deltapi 3 ;read from delayline
   delayw gaOutL+(aWetL) ;write to dlyln.
  aDelayR delayr 3
   aWetR deltapi 3
   delayw gaOutR+(aWetR)
  outs gaOutL + aWetL, gaOutR + aWetR
endin


instr 31  ;Amplitude Modulation Effect
  aMod oscils 0.5, 300, 0 ;Sinus 300 Hz
  aL = gaOutL * (aMod+0.5)
  aR = gaOutR * (aMod+0.5)
  outs aL, aR
endin
```

To make sure the overall audio output to the soundcard is never overwritten, an output instrument with the highest instrument number (here instr 200) is started directly from the Csound score.

```
instr 200 ;Main Output
outs gaOutL, gaOutR
gaOutL = 0.0
gaOutR = 0.0
endin

</CsInstruments>
<CsScore>
i 200 0 3600 ;Output runs for one hour
e
</CsScore>
</CsoundSynthesizer>
```

This technique allows on-the-fly re-routing of sub-instruments and supports the creation of various effect chains also with any number of effects.

Additionally we recommend to add a cross-fade function to the sub-instruments to avoid clicks when changing the order of effects.

3 Discussion

We developed a live-electronic performance setup, that monitored and processed the saxophone's reed vibrations directly. With this method we can work at a wide range of amplitude levels on stage and apply drastic sound effects to the signal without the risk of feedback loops. Also when playing in an ensemble, the reed signal is free of surrounding noise from other instruments. This makes the reed signal also more suitable for frequency-domain feature detection (e.g., pitch-tracking, onset detection). In addition, we explained a setup within Csound, which allows re-routing of sound effects on-the-fly during live performance.

One disadvantage of this pickup method is that the characteristic air noise is missing in the sound. For compensation a synthetic noise signal could be added to the output.

An other difficulty is the overall high amount of background noise in the signal, which is a common problem with strain gauge measurements [Scott and Owens, 1989]. Further research on the characteristics of each single component is intended to improve the signal to noise ratio.

In future work, we also plan to optimize articulation detection algorithms, based on findings from performance research [Hofmann et al., 2012b], in terms of real-time onset detection applications.

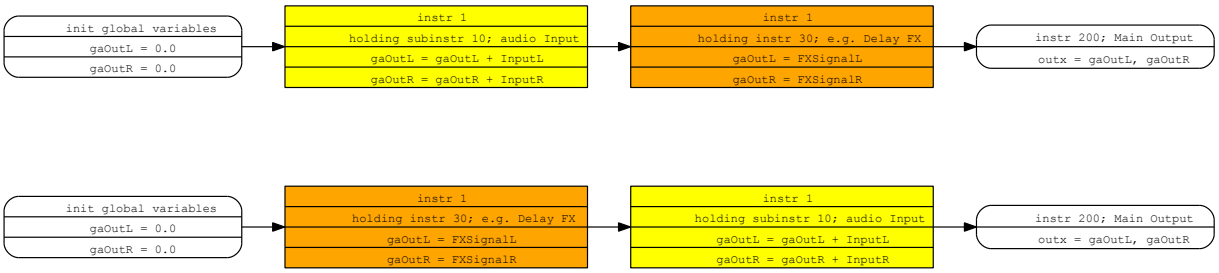


Figure 4: Top: Routing of soundcard input into an (delay) effect instrument. Bottom: Changed processing order. The input is placed after the (delay) effect.

4 Acknowledgements

The authors would like to thank the Csound mailing list for their support, particularly Victor Lazzarini, Oeyvind Brandtsegg and Joachim Heintz for discussing possiblities of instrument routing in Csound, and Roman Rofalski for live testing the effect setup. This research was supported by the Austrian Science Fund (FWF): P23248-N24.

References

T. Bacon, 1984. *The New Grove dictionary of musical instruments*, chapter 'Electric guitar' S. Sadie (Ed.). London: Macmillan Press Ltd.; Grove's Dictionaries of Music.

R.C. Boulanger and V. Lazzarini. 2011. *The Audio Programming Book*. Cambridge: MIT Press.

M. Campbell, C.A. Greated, and A. Myers. 2004. *Musical Instruments: History, Technology, and Performance of Instruments of Western Music*. Oxford: Oxford University Press.

N. Collins. 2009. *Handmade Electronic Music: The Art of Hardware Hacking*. New York: Routledge, second edition.

K. Ervik and O. Brandtsegg. (in press). Creating reverb effects using granular synthesis. *Proceedings of the First Csound Conference*. Retrieved from www.incontri.hmtm-hannover.de/fileadmin/www.incontri/Csound-_Conference.

J. Heintz, A. Hofmann, and I. McCurdy, 2011. *Csound - Floss Manual*, first edition. www.flossmanuals.net/csound.

A. Hofmann, V. Chatziioannou, W. Kausel, W. Goebl, M. Weilguni, and W. Smetana. 2012a. The influence of tonguing on tone production with single-reed instruments. In *5th Congress of the Alps Adria Acoustics Association*, Zadar, Croatia.

A. Hofmann, W. Goebl, M. Weilguni, A. Mayer, and W. Smetana. 2012b. Measuring tongue and finger coordination in saxophone performance. In *12th International Conference on Music Perception and Cognition (ICMPC) and 8th Triennial Conference of the European Society for the Cognitive Sciences of Music (ESCOM)*, pages 442–445, Thessaloniki, Greece.

A. Hofmann, M. Chatziioannou, V.and Weilguni, W. Goebl, and W. Kausel. 2013. Measurement setup for articulatory transient differences in woodwind performance. In *Proceedings of the 21st International Congress on Acoustics*, page to appear.

H. Lemme. 1994. *Elektro Gitarren Sound*. Richard Pflaum Verlag GmbH & Co. KG, München.

H. Mikelson. 2000. Modeling a multieffects processor in csound. In *The Csound Book*, pages 575–594. Cambridge: MIT Press.

J. Noble. 2009. *Programming Interactivity*. Oreilly Series. O'Reilly Media.

K. Scott and A. Owens, 1989. *Strain Gauge Technology*, chapter 'Instrumentation' A.L. Window (Ed.), pages 151–216. An Objective publication. Essex: Elsevier Science Publishers, second edition.

B.I. Waerstad. 2010. Granulated guitar - en digital utvidelse av gitaren som instrument. Master's thesis, Institutt for Musikk, Norwegian University of Science and Technology.

MOD – An LV2 host and processor at your feet

Gianfranco Ceccolini
The MOD team
Rua Júlio Rebollo Perez, 488 – cj 3
São Paulo, Brasil
contato@portalmod.com

Leonardo Germani
The MOD team
Rua Júlio Rebollo Perez, 488 – cj 3
São Paulo, Brasil
leogermani@hacklab.com.br

Abstract

MOD is a linux-based LV2 [1] plugin processor and controller. It is accessed by musicians via bluetooth to setup their pedalboards by making internal digital connections between plugins and audio inputs/outputs.

After a pedalboard set is saved, it can be shared with other users at the MOD Social network.

The software components are Open Source, which means you can also run it on any computer using GNU/Linux operating system, not only on MOD hardware.

The presentation aims to introduce the device to the community and discuss how its development may interact with plugins and the LV2 standard's development.

Keywords
LV2, host, hardware, jack, plugins

1 Introduction

We have been developing MOD for more than three years. From the first draft to a usable prototype, now our first end-user ready units are being produced and we would love to show them at LAC, since MOD is only possible thanks to the work of many people we expect to meet there.

MOD is a LV2 host device at your feet, that can be used on a live performance disconnected from a computer. It has a powerful and intuitive user interface to create setups via tablet or computer and it supports any LV2 plugin.

1.1 Motivation

As old-time users of Jack, LADSPA and all the GNU/Linux pro-audio stack, we always missed an easy, plug-and-play way to use the effects and tools available in our GNU/Linux boxes. MOD was born with this in mind, to provide an accesible and user-friendly device that supports the tools GNU/Linux users are used to and also accesible to more traditional musicians.

2 Overview

2.1 How it works

In a pinch, you plug your instrument and amp to the MOD and play. It processes the pedalboards you have saved in it.

To create or change a pedalboard you connect to the MOD using a tablet or a computer via bluetooth and access it using a browser. You will be presented with a dashboard where you can drag and drop plugins and make connections. At the left side on the screen you will see two fixed sound sources that represents the physical audio inputs, where you will plug in your instruments, and two audio outputs on the right, representing the physical audio outputs.

You can make as many connections as you like from each sound source, being it the physical audio input or a plugin output.

Each plugin is represented by a pedal. Click on it and you can tweak all its parameters. You can also address a plugin parameter to a physical knob, foot switch or expression pedal.

MOD comes with many plugins installed, but you can search and install more plugins on the fly, browsing the MOD repository (see 3.1). Plugins are organized into categories and rated by the users.

Once you are done with your setup, we call it a pedalboard, you can name it and save it. When you have many pedalboards you can organize it in banks so you can easily access it from the MOD menu when you are not using your tablet/computer.

MOD can be used in a standalone mode, which mean you don't need to carry your computer to the gig. You can easily browse the banks and pedalboards you organized from the menu, or you can even address one or two foot-switches to go to the next/previous pedalboard.

Back to your computer, you can now share your pedalboards with other MOD users and exchange ideas and tones.

2.2 Hardware

The MOD is enclosed in a robust folded sheet-metal steel enclosure with spill-proof top face.

Inside it there's an embedded Intel Atom dual core running at 1.8GHz with an SSD drive for storage.

It sports a 2 x 2 24bit / 48kHz audio interface with very low noise (>100dB). The input section consists of two pre-amped XLR + P10 combo jacks with selectable line/instrument level and adjustable gain. The output section consists of two balanced P10 connectors with adjustable gain plus a Headphone jack.

The controller is composed of four incremental encoders with LCD displays for parameter changing and visualization plus four heavy duty foot-switches. It is equipped with an ARM processor for parameters treatment and displaying.

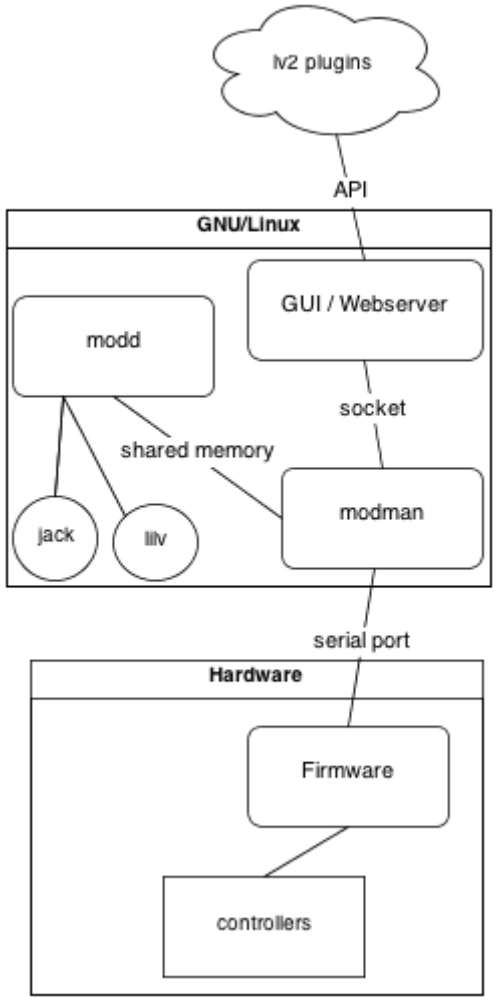
There's also an internal full range power supply - 100V-240V - to deliver adequate power to all internal components.

2.3 Software

MOD software is divided into an LV2 host, the firmware, a web-based GUI and the manager - the software responsible for the communication between all the other pieces.

All software, with the exception of the firmware, is Free Software and licensed under the GPLv3 license [2].

3 Software Components



Picture 1: Software components

3.1 Operating System

MOD runs a customized ArchLinux [3] based distro, currently running the kernel 3.6.11 tuned for real-time and some other performance patches.

It uses systemd and all daemons rely on it for the start/stop/dependency/keep alive actions. The software is packaged as pacman [4] packages and are published on MOD arch repository which is used to manage the dependencies between packages and to keep the OS stable/frozen.

The communication between the hardware and the OS/software is done through a real serial port.

3.2 LV2 host / Jack Client - modd

The host is called modd and is really simple, so far it supports all the basic / core lv2 functionalities and also the following lv2 features/extensions:

- uri-map
- uridmap
- MIDI events (event and atom)

Support to the time extension is being developed, but at first it will not have transport support, being frame and tempo aware only, so the user can specify a BPM (via knobs or using the tap-tempo foot-switch) and syncing time/bpm/hz parameters.

Modd is mostly written in C using David Robillard's lilv [5] library and is our main jack [6] client. It runs with real-time priority.

3.3 MOD Manager - modman

The manager is responsible for handling the communication between the GUI, the Controller Firmware, the Host and the OS. It also have some core responsibilities for example when saving user data, turning on/off the bluetooth and other system related tasks.

It's also written in C and it communicates with modd using a shared struct (shared memory). It uses a serial port to send messages to the firmware and a socket to the Python [7] based web server and JavaScript [8] GUI. All the communication uses a simple protocol specified by MOD's team. The protocol describes all the functionalities from adding/removing plugins to assigning control ports to hardware encoders or setting/reading control port values.

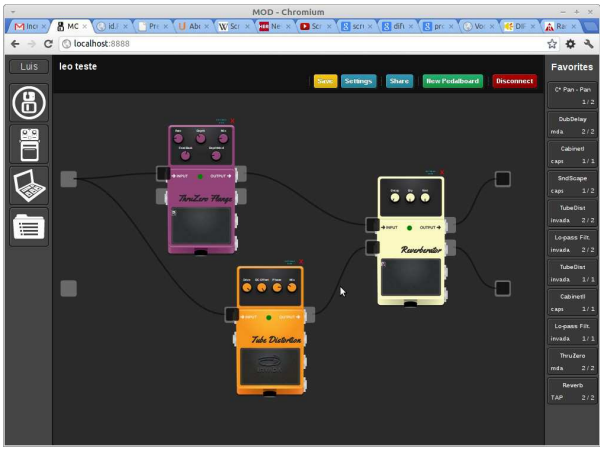
3.4 Bluetooth

The bluetooth host is written in Python using Dbus [9] and Bluez [10]. It simply pairs, authorizes and waits for new bluetooth network connections, whenever a connection is created (using the bnep0 interface) an udev [11] rule is matched and executes a script that will set the IP address for the interface and run dnsmasq [12] on that interface. dnsmasq will then provide a dhcp and dns server for clients connected through bluetooth, that way if the user's computer or tablet tries to resolve the name "mod" it will resolve to the device's IP address.

3.5 Web GUI

The GUI is mostly written in JavaScript and Python using the Tornado framework. It uses the commands described by the MOD protocol to communicate with the manager (modman) through a socket, modman will then route the messages to the other components.

The MOD device is never connected directly to the Internet, the GUI JavaScript code will detect if the user's computer/tablet is connected and, if it is, it will be able to access the cloud and download new plugins, upgrade older plugins or upgrade the whole system, including the firmware.



Picture 2: The Beta GUI

3.6 Firmware

The firmware that drives the controller manages all it's hardware : displays, encoders and foot-switches.

It is also responsible for:

- managing the exchange of data between the actuators and modman;
- dealing with controlwise parameters properties, such as using logarithmic scales or setting a tap-tempo chronometer when a time-related parameter is addressed to a footswitch;

4 MOD and LV2

4.1 Plugins repository

To make MOD's users life easier, we are packaging and hosting LV2 plugins in a repository, just as linux distributions do.

We have ported some LADSPA [13] plugins to LV2, such as the CAPS [14] and TAP [15] packages and we will have around sixty plugins packaged on the ongoing release date.

From the user interface users can only install plugins from this repository, but we are already working to setup a "contrib" repository and also the possibility to install any LV2 plugin you have downloaded from anywhere.

4.2 MOD LV2 extension

Any LV2 plugin will work well on MOD. But since we are focusing on user experience, we want each plugin to have its own look.

Part of it is supported by the LV2, but another part of it is specific to MOD. For instance, on the MOD dashboard, each plugin has its icon, usually a pedal icon. It would be nice that the plugin developer could choose the icon he/she wanted. When a parameter is addressed to a physical control, it has a label that appears on the LCD screen, this could be customized too. Also, the configuration GUI, that LV2 allows developer to build, could only be used in MOD if it was written in HTML/javascript.

For those reasons we are extending the LV2 standard, adding some specific features that MOD supports. Nevertheless, we are not touching the standard, and we are supporting any plugin that does not have this extensions with the use of default options.

4.2.1 SDK

In the future we want to release a SDK to help plugin developers adapt their plugin to the MOD extensions. It will give them layout templates and a set of icons to choose from.

5 Conclusions

Being based on open standards we believe MOD can become a much better and versatile multi-effect processor than any other found on the market. We want to improve the social functionalities and support the LV2 standard so new developers get interested and new plugins are developed.

6 Acknowledgments

MOD started because of the LAD and LAU community, because of the softwares and concepts that originated there, so we would like to thank this community of users and developers. We also want to thank the estudiolivre.org community (a Brazilian community focused on the use of free and open-source software for multimedia production).

7 References

[1] LV2, a plugin standard for audio systems http://lv2plug.in/trac/

[2] GPL, the GNU General Public License http://www.gnu.org/licenses/gpl.html

[3] ArchLinux, a simple and lightweight GNU/Linux distribution http://www.archlinux.org

[4] PACMAN, the ArchLinux package manager https://wiki.archlinux.org/index.php/Pacman

[5] LILV, a library to make the use of LV2 plugins as simple as possible http://drobilla.net/software/lilv

[6] JACK Audio Connection Kit http://www.jackaudio.org

[7] Python Programming Language http://www.python.org/

[8] JavaScript, the scripting language of the Web http://www.w3schools.com/js/default.asp

[9] Dbus, a message bus system for applications to talk to one another http://www.freedesktop.org/wiki/Software/dbus

[10] BlueZ, the official Linux bluetooth protocol stack http://www.bluez.org/

[11] UDEV, Linux dynamic device management https://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html

[12] DNSMASQ, a lightweight DNS forwarder and DHCP server http://www.thekelleys.org.uk/dnsmasq/

[13] LADSPA, Linux Audio Developer's Simple Plugin API http://www.ladspa.org

[14] CAPS, the C* Audio Plugin Suite http://quitte.de/dsp/caps.html

[15] TAP, Tom's Audio Processing plugins http://tap-plugins.sourceforge.net/

Supporters


virtual vehicle


TU
Graz


mur at
initiative netzkultur

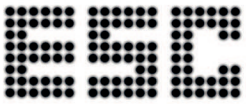

linuxaudio.org


discovering
music
archives


postgarage


FORUM STADTPARK


Pd~graz




RADIO
HELSINKI
Freies Radio Graz
92,6
MHz


auphonic


kunst
uni
graz


iem