LAC2004 Proceedings

2nd International Linux Audio Conference

April 29 – May 02, 2004

ZKM | Zentrum für Kunst und Medientechnologie

Karlsruhe, Germany

Published by
ZKM | Zentrum für Kunst und Medientechnologie Karlsruhe, Germany
December, 2004
All copyright remains with the authors
www.zkm.de/lac/2004

Staff

Organization Team LAC2004

Götz Dipper	ZKM | Institute for Music and Acoustics
Matthias Nagorni	SuSE Linux GmbH
Frank Neumann	LAD

ZKM

Jürgen Betker	Graphic Artist
Hartmut Bruckner	Sound Engineer
Ludger Brümmer	Head of the Institute for Music and Acoustics
Uwe Faber	Head of the IT Department
Viola Gaiser	Head of the Event Department
Hans Gass	Technical Assistant
Joachim Goßmann	Tonmeister
Martin Herold	Technical Assistant
Markus Kritzokat	Event Management
Andreas Liefländer	Technical Assistant
Tanya Lübber	Event Management
Alexandra Mössner	Assistant of Management
Caro Mössner	Event Management
Marc Riedel	Event Management
Thomas Saur	Sound Engineer
Joachim Schütze	IT Department
Bernhard Sturm	Production Engineer
Manuel Weber	Technical Director of the Event Department
Monika Weimer	Event Management

LAD

Jörn Nettingsmeier	Coordination of the Internet Audio Stream

Relay Operators

François Déchelle	IRCAM
Fred Gleason	Salem Radio Labs
Marco d'Itri	Italian Linux Society
Patrice Tisserand	IRCAM

Chat Operator

Sebastian Raible

Icecast/Ices Support

Jan Gerber	
Karl Heyes	Xiph.org

(In alphabetical order)

Content

Preface

The Linux Audio “movement” has a long history. It started with individuals working on pet projects at home, only occasionally showing the results of their work in postings on forums such as Usenet News. In 1998 a mailing list was formed to serve as a meeting point for these individuals — developers and users alike. This was later branded the “Linux Audio Developer’s Mailing List” (LAD).

It became obvious that there was a lot of potential in these people and their software, so it seemed like a good idea to demonstrate this to the public in some form. In July 2001 at the German “LinuxTag” (Europe’s largest expo on all things Linux), we had an open-source booth (with all expenses paid for by the LinuxTag organizers) for the first time where we demonstrated certain selected programs to visitors for 4 days. It was well received, so we did the same thing in the following years.

However, it became apparent that while this was informative and useful for the visitors, it left little time for the booth staff to talk to each other and share knowledge about their ideas and projects. This fact gave birth to the idea of a “programmer’s meeting”. We contacted the ZKM during the search for a meeting location in late 2002. The ZKM, in turn, expressed their interest in taking part, and we began planning together. The ZKM proposed the inclusion of public talks and presentations, thus transforming the meeting into a conference which could reach a much larger audience. On March 14–16, 2003, the first conference with roughly 10 presentations took place (free of charge for all), and everyone agreed that it was a good thing that should be repeated the following year.

The 2004 conference occurred once again at ZKM, in cooperation with SuSE Linux, from April 29 to May 2. Due to the success of the first conference, the second was planned as a significantly larger event. There were more than 30 talks, several workshops, demos, a panel discussion, and lots of spontaneous meetings and discussions. The ZKM engagement in the second conference also increased. While LAC2003 focused on the development of audio software, in 2004 the ZKM helped bring musicians into the mix. The 2004 conference featured several concerts of different types of electro-acoustic music, including world premieres and a piece by Orm Finnendahl commissioned by and developed at ZKM. The increase in scope was reflected in the name change from “Linux Audio Developer’s Conference” to “Linux Audio Conference”. Attendance to the talks remained free of charge, thanks to the ZKM’s efforts.

As it was the case in 2003 already, interested parties who were unable to attend the conference in person could listen to the talks through a live audio stream on the Internet. The talks covered a wide spectrum of topics, including audio architecture, hard disk recording, audio mastering, software synthesis, sampling, virtual instruments, spatialization, music notation, computer music, and documentation.

The idea to publish proceedings did not come up until during the conference. Therefore the proceedings of LAC2004 comprise only a portion of the talks, and we are especially grateful for the papers that we received. Links to ogg files and abstracts of all talks can be found at www.zkm.de/lac/2004.

The conference was sponsored by Lionstracs and Hartmann. During the conference two Linux based synthesizers were presented, Mediastation by Lionstracs and Neuron by Hartmann.

We owe many thanks to everybody who contributed to the conference and helped in making it such a great success. We are looking forward to the LAC2005 which will take place on April 21–24, 2005.

Frank Neumann, Matthias Nagorni and Götz Dipper
Organization Team LAC2004
Karlsruhe, December 20, 2004

Application of Wave Field Synthesis in electronic music and sound installations

M.A.J. Baalman, M.Sc.

Electronic Studio, Communication Sciences, University of Technology, Berlin, Germany
email: marije@baalt.nl.
web: www.nescivi.de

Abstract

Wave Field Synthesis offers new possibilities for composers of electronic music and to sound artists to add the dimension of space to a composition. Unlike most other spatialisation techniques, Wave Field Synthesis is suitable for concert situations, where the listening area needs to be large. Using the software program "WONDER", developed at the TU Berlin, compositions can be made or setups can be created for realtime control from other programs, using the Open Sound Control protocol. Some pieces that were created using the software are described to illustrate the use of the program..

Introduction

Wave Field Synthesis is a novel technique for sound spatialisation, that overcomes the main shortcoming of other spatialisation techniques, as there is a large listening area and no "sweet spot".

This paper describes the software interface WONDER that was made as an interace for composers and sound artists in order to use the Wave Field Synthesis technique. A short, comprehensive explanation of the technique is given, a description of the system used in the project at the TU Berlin and the interface software, followed by a description of the possibilities that were used by composers.

Wave Field Synthesis

The concept of Wave Field Synthesis (WFS) is based on a principle that was thought of in the 17th century by the Dutch physicist Huygens (1690) about the propagation of waves. He stated that when you have a wavefront, you can synthesize the next wavefront by imagining on the wavefront an infinite number of small sources, whose waves will together form the next wavefront (figure 1).

Based on this principle, Berkhout (1988) introduced the wave field synthesis principle in acoustics.

By using a discrete, linear array of loudspeakers (figure 2), one can synthesize correct wavefronts in the horizontal plane (Berkhout, De Vries and Vogel 1993). For a complete mathematical treatment is

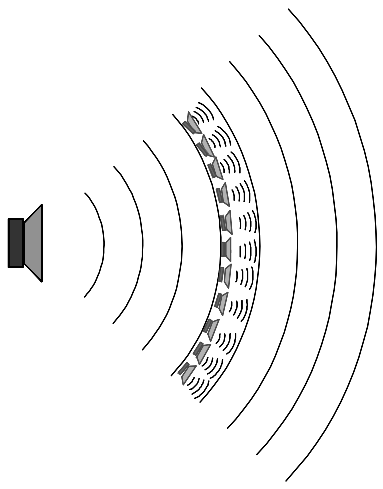


Figure 1. The Huygens' Principle

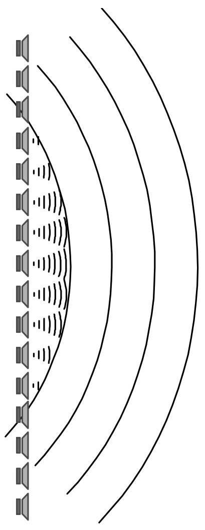


Figure 2. The Wave Field Synthesis principle

referred to Berkhout (1988, 1993) and various other papers and theses from the TU Delft[1].

An interesting feature is that it is also possible to synthesize a sound source in front of the speakers (Jansen 1997), something which is not possible with other techniques.

[1] Sound Control Group, TU Delft, http://www.soundcontrol.tudelft.nl

Jansen (1997) derived mathematical formulae for synthesising moving sound sources. He took into account the Doppler effect and showed that for its application one would need to have continuously time-varying delays. He also showed that for slow moving sources the Doppler effect is negligible and one can resort to updating locations and calculating filters for each location and changing those in time.

This approach was chosen in this project. Additionally, in order to avoid clicks in playback, an option was built in to crossfade between two locations to make the movement sound smoother.

Theoretical and practical limitations of Wave Field Synthesis

There are some limitations to the technique. The distance between the speakers needs to be as small as possible in order to avoid spatial aliasing. From (Verheijen 1998) we have the following formula for the frequency above which spatial aliasing occurs:

$$f_{Nyq} = \frac{c}{2\Delta x \sin\alpha}$$

where c is the speed of sound in air, Δx the distance between the speakers and α the angle of incidence on the speaker array. Thus the frequency goes down with increasing distance between the speakers, but it also depends on the angle of incidence, thus the location of the virtual source, whether or not aliasing occurs.

Spatial aliasing has a result that a wave field is not correctly synthesized anymore and artefacts occur. This results in a bad localisable sound source. This limitation is a physical limitation, which can not really be overcome. However it depends on the sound material whether or not this aliasing is a problem from a listener's point of view. In general, if the sound contains a broad spectrum with enough frequencies below the aliasing frequency, the source is still well localisable.

On the other end of the frequency spectrum there is the problem that very low frequencies are hard to play back on small speakers. For this can however, just as in other spatialisation systems, a subwoofer be added, as low frequencies are hard to localise by the human ear.

Another limitation is that a lot of loudspeakers are needed to implement the effect. Because of this, there is research done into loudspeaker panels, so that it is easier to build up a system.

Finally, a lot of computation power is needed, as for each loudspeaker involved a different signal needs to be calculated. With increasing compuation power of CPU's, this is not really a big problem. At the moment it is possible to drive a WFS-system with commercially available PC's.

System setup at the TU Berlin

The prototype system in Berlin was created with the specific aim to make a system for the use in electronic music (Weske 2001). The system consists of a LINUX PC driving 24 loudspeakers with an RME Hammerfall Soundcard.

For the calculation (in real time) of the loudspeaker signals the program BruteFIR by Torger[2] is used. This program is capable of making many convolutions with long filters in realtime. The filter coefficients can be calculated with the interface software described in this paper.

With the current prototype system it is possible to play a maximum of 9 sound sources with different locations in realtime, even when the sources are moving. This is the maximum amount of sources; the exact amount of sources that can be used in a piece depend on the maximum filter length used. A detailed overview of the capacity was given in a paper presented at the ICMC in 2003 (Baalman 2003).

Interface software

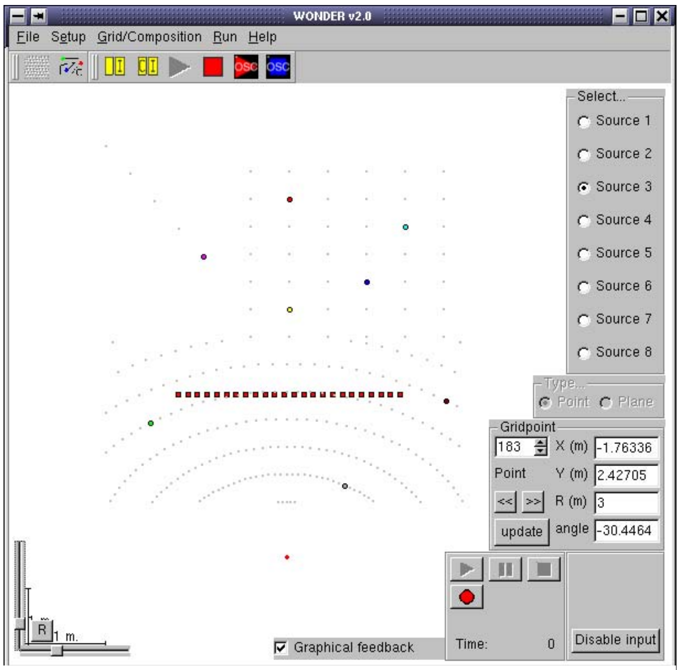
In order to work with the system, interface software was needed to calculate the necessary filter coefficients. The aim was to create an interface that allows composers to define the movements of their sounds, independent of the system on which it eventually will be played. That is, the composer should be bothered as less as possible with the actual calculations for each loudspeaker, but instead be able to focus on defining paths through space for his sounds.

The current version of the program WONDER (Wave field synthesis Of New Dimensions of Electronic music in Realtime) allows the composer to do so. It allows the composer to work in two ways with the program: either he creates a composition of all movements of all the sound sources with WONDER, using the composition tool, or he defines a grid of points that he wants to use in his piece and controls the movement from another program using the OpenSoundControl protocol (Wright e.a, 2003). The main part of the program is the play engine which can play the composition created or move the sources in realtime; a screenshot is given in figure 3.

The array configuration can be set in the program. It is possible to define the position of various array segments through a dialog.

WONDER includes a simple room model for calculation of reflections. The user can define the position of four walls of a rectangular room, an absorption factor and the order of calculation. The calculations are done with the mirror image source model (see also Berkhout 1988).

[2] Torger, A., *BruteFIR*, http://www.ludd.luth.se/~torger/brutefir.html



Select...
Source 1
Source 2
[✓ Source 3]
Source 4
Source 5
Source 6
Source 7
Source 8

Type...
[✓ Point] Plane

Gridpoint
183 X (m) -1.76336
Point Y (m) 2.42705
<< >> R (m) 3
update angle -30.4464

☑ Graphical feedback

Time: 0

Figure 4. The user interface of the play engine of WONDER

Experiences with composers

During the development of the program, various compositions were made by different composers, to test the program and to come up with new options for the program. These compositions were presented at different occasions, amongst which festivals like Club Transmediale in Berlin (February 2003) and Electrofringe in Newcastle, Australia (October 2003). I will elaborate about two compositions and one sound installation.

Marc Lingk, a composer residing in Berlin, wrote a piece called Ping-Pong Ballet. The sounds for this piece were all made from ping-pong ball sounds, which were processed by various algorithms, alienating the sound from its original. Using these sounds as a basis, the inspiration for the movements was relatively easy as the ping-pong ball game provides a good basis for the distribution in space of the sounds. In this way he created various loops of movement for the various sounds as depicted in figure 4. Paths 1 & 2 are the paths of the ball bouncing on the table, 3 & 4 of the ball being hit with the bat, 5 & 6 of multiple balls bouncing on the table, 7 & 8 of balls dropping to the floor. Choosing mostly prime numbers for the loop times, the positions were constantly changing in relative distance to each other. The movement was relatively fast (loop times were between 5 and 19 seconds). In the beginning, the piece gives the impression of a ping-pong ball game, but as it progresses the sounds become more and more dense, creating a clear and vivid spatial sound image.

In the composition "Beurskrach" created by Marije Baalman, four sources were defined, but regarded as being points on one virtual object, i.e. these points made a common movement; the sound material for these four points were also based on the same source material, but slightly different filterings of this, to simulate a real object where from different parts of the object different filterings of the sound are radiated. During the composition, the object comes closer from afar and even comes in front of the loudspeakers, there it implodes and scatters out again, making a rotating movement behind the speakers, before falling apart in the end. See figure 5 for a graphical overview of this movement.

The sound installation "Scratch", that was presented during the Linux Audio Conference, makes use of the OSC-control over the movements. The sound installation is created with SuperCollider, which makes the sound and which sends commands for the movement to WONDER. The concept of the sound installation is to create a

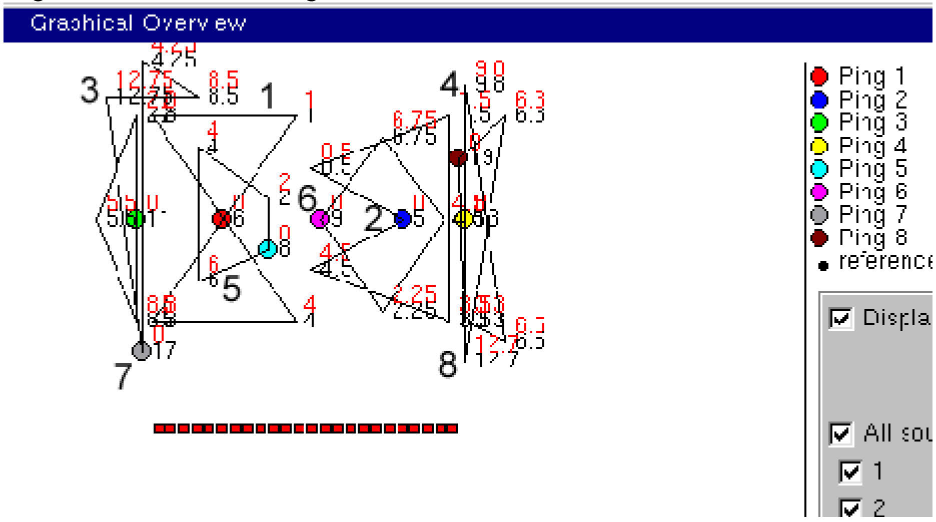
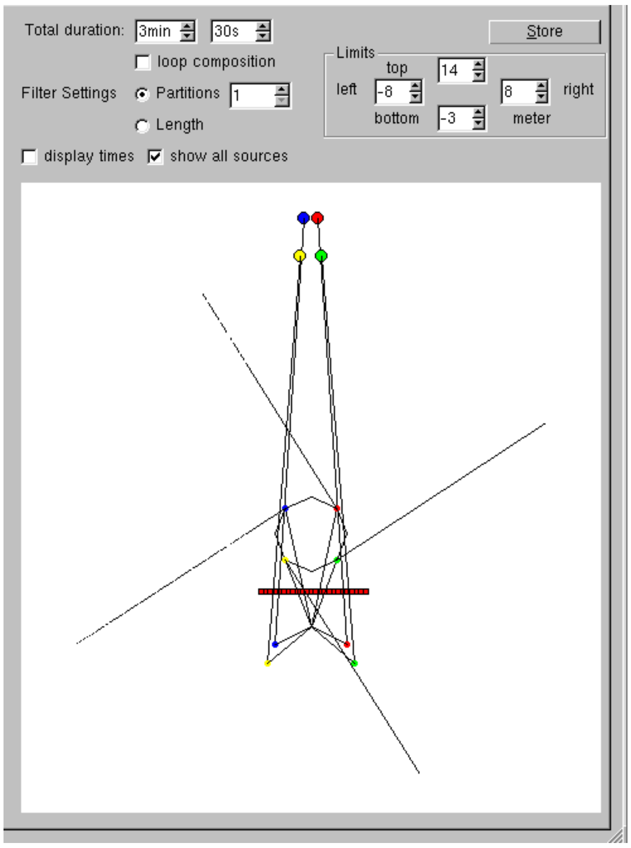


Figure 3. Overview of the movements of the composition "Ping Pong Ballet" (screenshot from a previous version of WONDER)


Total duration: 3min 30s

Store

☐ loop composition

Filter Settings [✓ Partitions] 1 Length

Limits: top 14, left -8, right 8, bottom -3, meter

☐ display times ☑ show all sources

Figure 5. Overview of the movements of the sound sources of the composition "Beurskrach"

kind of sonic creature, that moves around in the space. Depending on internal impulses and on external impulses from the visitor (measured with sensors), the creature develops itself, and makes different kinds of sounds, depending on its current state. The name "Scratch" was chosen because of two things: as the attempt to create such model for a virtual creature was the first one, it was still a kind of scratch for working on this concept. The other reason was the type of sound, which were kind of like scratching on some surface.

Conclusions and future work

The program WONDER provides a usable interface for working with Wave Field Synthesis, as shown by the various examples of compositions that have been made using the program.

Future work will be, apart from bug fixing, on integrating BruteFIR further into the program, in order to allow for more flexible use in realtime. Also an attempt will be made to incorporate parts of SuperCollider into the program, as this audio engine has a few advantages over BruteFIR that could be used. Also, there will be work done on more precise synchronisation possibilities for use with other programs.

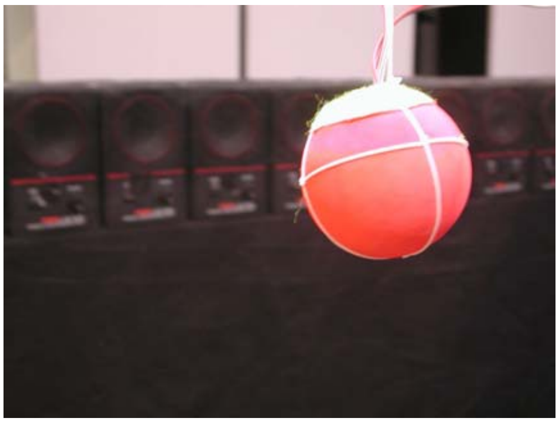


Figure 6. The sound installation "Scratch" during the Linux Audio Conference. In the ball are accelerometers to measure the movement of the ball, which influences the sound installation (photo by Frank Neumann).

Other work will be done on creating the possibility to define more complex sound sources (with a size and form) and implementing more complex room models.

Credits

WONDER is created by Marije Baalman. The OSC part is developed by Daniel Plewe.

References

Baalman, M.A.J., 2003, Application of Wave Field Synthesis in the composition of electronic music, *International Computer Music Conference 2003,* Singapore, October 1-4, 2003

Berkhout, A.J. 1988, A Holographic Approach to Acoustic Control, *Journal of the Audio Engineering Society*, 36(12):977-995

Berkhout, A.J., Vries, D. de & Vogel, P. 1993, Acoustic Control by Wave Field Synthesis, *Journal of the Acoustical Society of America*, 93(5):2764-2778

Jansen, G. 1997, *Focused wavefields and moving virtual sources by wavefield synthesis*, M.Sc. Thesis, TU Delft, The Netherlands

Huygens, C. 1690, *Traite de la lumiere; ou sont expliquees les causes de ce qui luy arrive dans la reflexion et dans la refraction et particulierement dans l'etrange refraction du cristal d'Islande; avec un discours de la cause de la pesanteur*, Van der Aa, P., Leiden, The Netherlands

Verheijen, E.N.G. 1998, *Sound Reproduction by Wave Field Synthesis*, Ph.D. Thesis, TU Delft, The Netherlands

Weske, J. 2001, *Aufbau eines 24-Kanal Basissystems zur Wellenfeldsynthese mit der Zielsetzung der Positionierung virtueller Schallquellen im Abhörraum*, M.Sc. Thesis, TU Chemnitz/TU Berlin, Germany

Wright, M., Freed, A. & Momeni, A. 2003, “OpenSoundControl: State of the Art 2003”, *2003 International Conference on New Interfaces for Musical Expression*, McGill University, Montreal, Canada 22-24 May 2003, Proceedings, pp. 153-160

RRADical Pd

Author: Frank Barknecht <fbar@footils.org>

Abstract

RRADical Pd is a project to create a collection of Pd patches, that make Pd easier and faster to use for people who are more comfortable with commercial software like Reason(tm) or Reaktor(tm). RRAD as an acronym stands for "Reusable and Rapid Audio Development" or "Reusable and Rapid Application Development", if it includes non-audio patches, with Pd. In the design of this system, a way to save state flexibly in Pd (persistence) had to be developed. For communication among each other the RRADical patches integrate the Open Sound Control protocol.

What it takes to be a RRADical

RRAD as an acronym stands for "Reusable and Rapid Audio Development" or "Reusable and Rapid Application Development", if it includes non-audio patches, with Pd. It is spelled RRAD, but pronounced "Rradical" with a long rolling "R".

The goal of RRADical Pd is to create a collection of patches, that make Pd easier and faster to use for people who are more used to software like Reason(tm) or Reaktor(tm). For that I would like to create patches, that solve real-world problems on a higher level of abstraction than the standard Pd objects do. Where suitable these high level abstractions should have a graphical user interface (GUI) built in. As I am focused on sound production the currently available RRADical patches mirror my preferences and mainly deal with audio, although the basic concepts would apply for graphics and video work using for example the Gem and PDP extensions as well.

Pre-fabricated high-level abstractions may not only make Pd easier to use for beginners, they also can spare lot of tedious, repeating patching work. For example building a filter using the `lop~` object of Pd usually involves some way of changing the cutoff frequency of the filter. So another object, maybe a slider, will have to be created and connected to the `lop~`. The typing and connecting work has to be done almost every time a filter is used. But the connections between the filter's cutoff control and the filter can also be done in advance inside of a so called abstraction, that is, in a saved Pd patch file. Thanks to the Graph-On-Parent feature of Pd the cutoff slider even can be made visible when using that abstraction in another patch. The new filter abstraction now carries its own GUI and is immediately ready to be used.

Of course the GUI-filter is a rather simple example (although already quite useful). But building a graphical note sequencer with 32 sliders and 32 number boxes or even more is something, one would rather have to do only once, and then reuse in a lot of patches.

Problems and Solutions

To build above, highly modularized system several problems have to be solved. Two key areas turned out to be very important:

Persistence How to save the current state of a patch? How to save more than one state (state sequencing)?

Communication The various modules are building blocks for a larger application. How should they talk to each other. (In Reason this is done by patching the back or modules with horrible looking cables. We must do better.)

It turned out, that both tasks are possible to solve in a consistent way using a unique abstraction. But first lets look a bit deeper at the problems at hand.

Persistence

Pd offers no direct way to store the current state of a patch. Here's what Pd author Miller S. Puckette writes about this in the Pd manual in section "2.6.2. persistence of data":

> Among the design principles of Pd is that patches should be printable, in the sense that the appearance of a patch should fully determine its functionality. For this reason, if messages received by an object change its action, since the changes aren't reflected in the object's appearance, they are not saved as part of the file which specifies the patch and will be forgotten when the patch is reloaded.

Still, in a musician's practice some kind of persistence turns out to be an important feature, that many Pd beginners do miss. And as soon as a patch starts to use lots of graphical control objects, users will - and should - play around with different settings until they find some combination they like. But unless a way to save this combination for later use is found, all this is temporary and gone, as soon as the patch is closed.

There are several approaches to add persistence. Max/MSP has the `preset`-object, Pd provides the similar `state`-object which saves the current state of (some) GUI objects inside a patch. Both objects also support changing between several different states.

But both also have at least two problems: They only save the state of GUI objects, which might not be everything that a user wants to save. And they don't handle abstractions very well, which are crucial when creating modularized patches.

Another approach is to (ab)use some of the Pd objects that can persist itself to a file, especially `textfile`, `qlist` and `table`, which works better, but isn't standardized.

A rather new candidate for state saving is Thomas Grill's `pool` external. Basically it offers something, that is standard in many programming languages: a data structure that stores key-value-pairs. This structure also is known as hash, dictionary or map. With `pool` those pairs also can be stored in hierarchies and they can be saved to or loaded from disk. The last but maybe most important feature for us is, that several pools can be shared by giving them the same name. A `pool MYPOOL` in one patch will contain the same data as a `pool MYPOOL` in another patch. Changes to one pool will change the data in the other as well. This allows us to use `pool MYPOOL`s inside of abstractions, and still access the pool from modules outside the abstractions, for example for saving the `pool` to disk.

A `pool` object is central to the persistence in RRADical patches, but it is hidden behind an abstracted "API", if one could name it that. I'll come back to how this is done below.

Communication

Besides persistence it also is important to create a common path through which the RRADical modules will talk to each other. Generally the modules will have to use, what Pd offers them, and that is either a direct connection through patch cords or the indirect use of the send/receive mechanism in Pd. Patch cords are fine, but tend to clutter the interface. Sends and receives on the other hand will have to make sure, that no name clashes occur. A name clash is, when one target receives messages not intended for it. A patch author has to remember all used send-names, which might be possible, if he did write the whole patch himself and kept track of the send-names used. But this gets harder to impossible, if he uses prefabricated modules, which might use their own senders, maybe hidden deep inside of the module.

So it is crucial, that senders in RRADical abstractions use local names only with as few exceptions as possible. This is achieved by prepending the RRADical senders with the string "$0-". So instead of a sender named `send volume`, instead one called `send $0-volume` is used. $0 makes those sends local inside their own patch borders by being replaced with a number unique to that patch. Using $0 that way is a pretty standard idiom in the Pd world.

Still we will want to control a lot of parameters and do so not only through the GUI elements Pd offers, but probably also through other ways, for example through hardware Midi controllers, through some kind of score on disk, through satellite navigation receivers or whatever.

This creates a fundamental conflict:

We want borders We want to separate our abstraction so they don't conflict with each other.

We want border crossings We want to have a way to reach their many internals and control them from the outside.

The RRADical approach solves both requirements in that it enforces a strict border around abstractions but drills a single hole in it: the **OSC inlet**. This idea is the result of a discussion on the Pd mailing list and goes back to suggestions by Eric Skogen and Ben Bogart. Every RRADical patch has (to have) a rightmost inlet that accepts messages formatted according to the OSC protocol. OSC stands for Open Sound Control and is a network transparent system to control (audio) applications remotely and is developed at CNMAT in Berkley by Matt Wright mainly.

The nice thing about OSC is that it can control many parameters over a single communication path (like a network conneciton using a definite port). For this OSC uses a URL-like scheme to address parameters organized in a tree. An example would be this message:

```
/synth/fm/volume 85
```

It sends the message “85” to the “volume” control of a “fm” module below a “synth” module. OSC allows many parameters constructs like:

```
/synth/fm/basenote              52
/synth/virtualanalog/basenote   40
/synth/*/playchords             m7b5 M6 7b9
```

This might set the base note of two synths, fm and virtualanalog and send a chord progression to be played by both – indicated by the wildcard * – afterwards.

The OSC-inlet of every RRADical patch is intended as the border crossing: Everything the author of a certain patch intends to be controlled from the outside can be controlled by OSC messages to the OSC-inlet. The OSC-inlet is strongly recommended to be the rightmost inlet of an abstraction. At least all of my RRADical patches do it this way.

Trying to remember it all: Memento

To realize the functionality requirements laid out so far I resorted to a so called Memento. “Memento” is a very cool movie by director Christopher Nolan where - quoting IMDB:

A man, suffering from short-term memory loss, uses notes and tattoos to hunt down his wife’s killer.

The movie’s main character Leonard has a similar problem as Pd: he cannot remember things. To deal with his persistence problem, his inability to save data to his internal harddisk (brain) he resorts to taking a lot of photos. These pictures act as what is called a Memento: a recording of the current state of things.

In software development Mementos are quite common as well. The computer science literature describes them in great detail, for example in the Gang-Of-Four book “Design Patterns” [Gamma95]. To make the best use of a Memento science recommends an approach where certain tasks are in the responsibility of certain independent players.

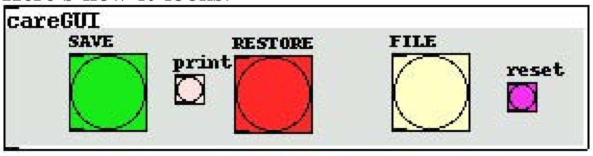
The Memento itself, as we have seen, is the photo, i.e. some kind of state record. A module called the “Originator” is responsible for creating this state and managing changes in it. In the movie, Leonard is the Originator, he is the one taking photos of the world he is soon to forget.

The actual persistence, that could be the saving of a state to harddisk, but could just as well be an upload to a webserver or a CVS check-in, is done by someone called the “Caretaker” in the literature. A Caretaker could be a safe, where Leonard puts his photos, or could be a person, to whom Leonard gives his photos. In the movie Leonard also makes “hard saves” by tattooing himself with notes he took. In that case, he is not only the Originator of the notes, but also the Caretaker in one single person. The Caretaker only has to take care, that those photos, the Mementos, are in a safe place and no one fiddles around with them. Btw: In the movie some interesting problems with Caretakers, who don’t always act responsible, occur.

Memento in Pd

I developed a set of abstractions, of patches for Pd, that follow this design pattern. Memento for Pd includes a `caretaker` and an `originator` abstraction, plus a third one called `commun` which is responsible for the **internal** communication. `commun` basically is just a thin extension of `originator` and should be considered part of it. There is another patch, the `careGUI` which I personally use instead of the `caretaker` directly, because it has a simple GUI included.

Here’s how it looks:



[Gamma95] E. Gamma and R. Helm and R. Johnson and J. Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley 1995

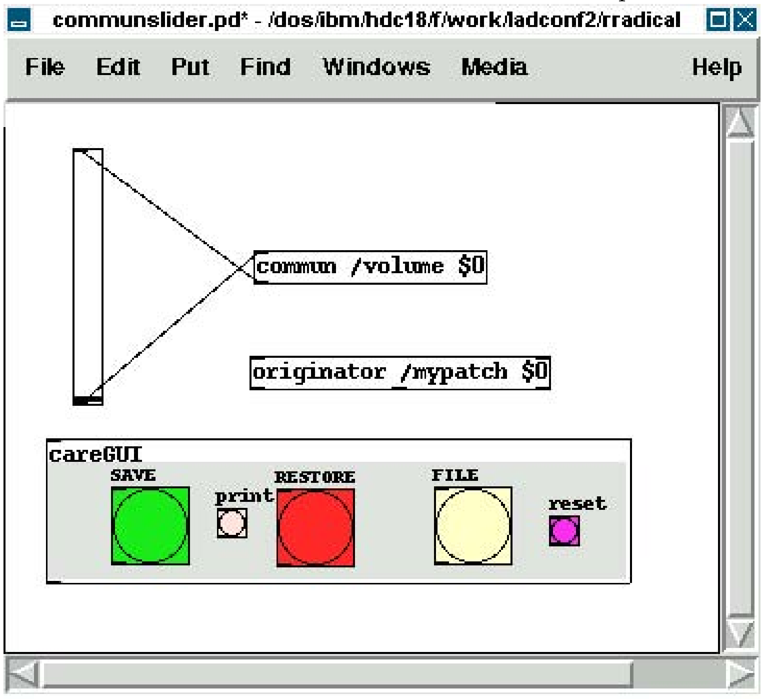
The careGUI is very simple: select a FILE-name to save to, then clicking SAVE you can save the current state, with RESTORE you can restore a state previously saved. After restore, the outlet of careGUI sends a bang message to be used as you like.

Internally caretaker has a named pool object using the global pool called "RRADICAL". The same pool RRADICAL also is used inside the originator object. This abstraction handles all access to this pool. A user should not read or write the contents of pool RRADICAL directly. The originator patch also handles the border crossing through OSC messages by its rightmost inlet. The patch accepts two mandatory arguments: The first on is the name under which this patch is to be stored inside the pool data. Each originator SomeName secondarg stores it's data in a virtual subdirectory inside the RRADICAL-pool called like its first argument - SomeName in the example. If the SomeName starts with a slash like "/patch" , you can also access it via OSC through the rightmost inlet of originator under the tree "/patch"

The second argument practically always will be $0. It is used to talk to those commun objects which share the same second argument. As $0 is a value local and unique to a patch (or to an abstraction to be correct) each originator then only can talk to communs inside the same patch and will not disturb other commun objects in other abstractions.

The commun objects finally are where the contents of a state are read and set. They, too, accept two arguments, the second of which was discussed before and will most of the time just be $0. The first argument will be the key under which some value will be saved. You should use a slash as first character here as well to allow OSC control. So an example for a usage would be commun /vol $0.

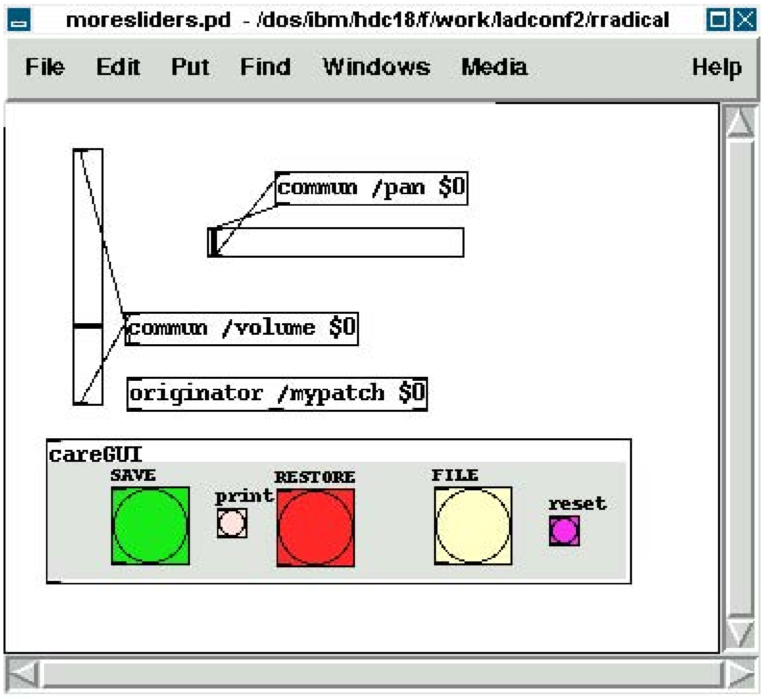
commun has one inlet and one outlet. What comes in through the inlet is send to originator who stores it inside its Memento under the key, that is specified by the commun's first arg. Actually originator. The outlet of a commun will spit out the current value stored under its key inside the Memento, when originator tells it to do so. So communs are intended to be cross-connected to some thing that can change. And example would be a slider which can be connected as seen in the next picture:


communslider.pd* - /dos/ibm/hdc18/f/work/ladconf2/rradical
File Edit Put Find Windows Media Help
commun /volume $0
originator /mypatch $0
careGUI
SAVE
print
RESTORE
FILE
reset

In this patch, every change to the slider will be reflected inside the Memento. The little print button in careGUI can be used to print the contents to the console from which Pd was started. Setting the slider will result in something like this:

```
/mypatch 0 , /volume , 38
```

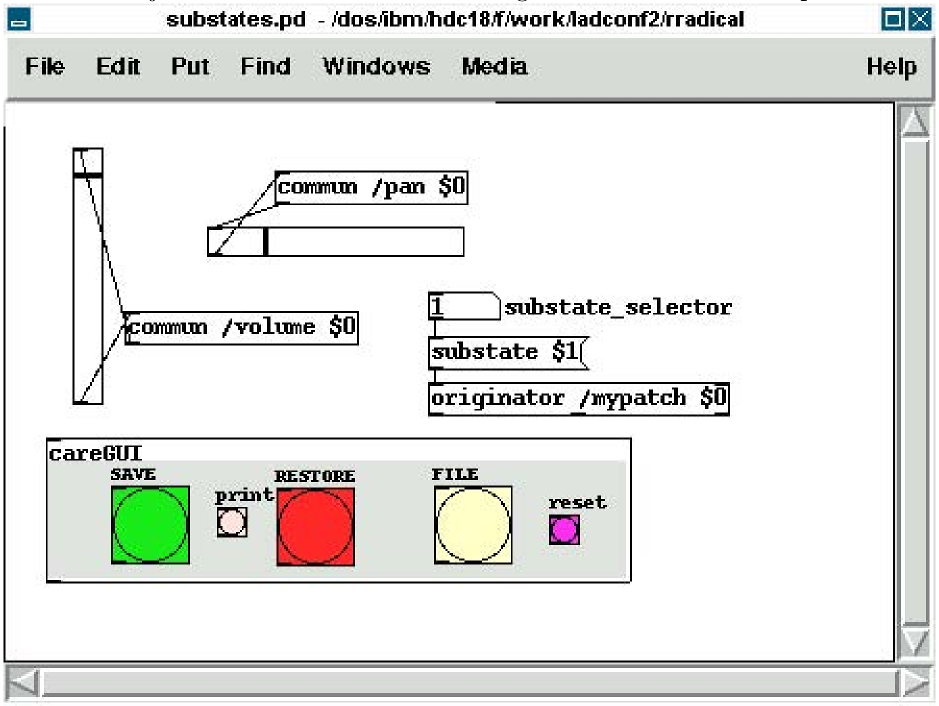
Here a comma separates key and value pairs. "mypatch" is the top-level directory. This contains a 0, which is the default subdirectory, after that comes the key "/volume", whose value is 38. Let's add another slider for pan-values:



Moving the /pan slider will let careGUI print out:

```
/mypatch 0 , /volume , 38
/mypatch 0 , /pan , 92
```

The `originator` can save several substates or presets by sending a `substate #number` message to its first inlet. Let's do just this and move the sliders again as seen in the next picture:



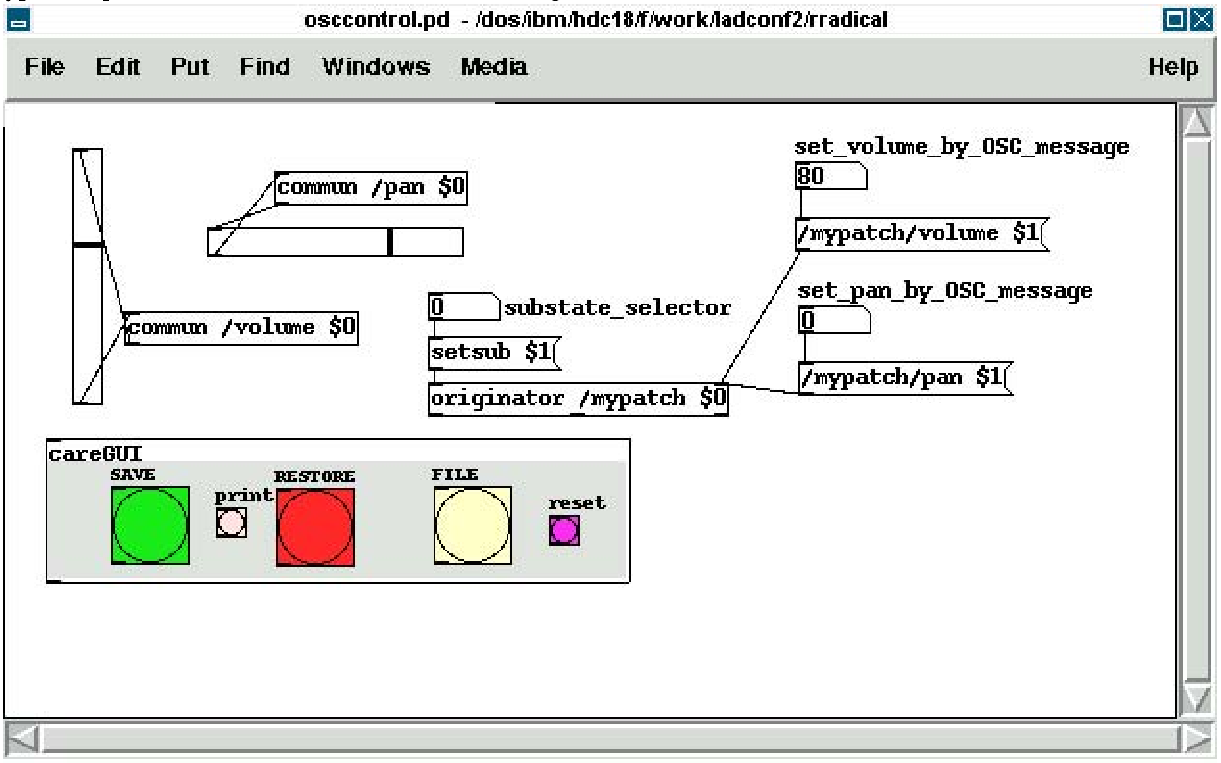
Now careGUI prints:

```
/mypatch 0 , /volume , 38
/mypatch 0 , /pan , 92
/mypatch 1 , /volume , 116
/mypatch 1 , /pan , 27
```

You see, the substate 0 is unaffected, the new state can have different values. Exchanging the `substate` message with a `setsub` message will autoload the selected state and "set" the sliders to the stored values immediately.

OSC in Memento

The whole system now already is prepared to be used over OSC. You probably already guess, how the message looks like. Any takers? Thank you, you're right, the messages are built as `/mypatch/volume #number` and `/mypatch/pan #number` as shown in the next stage:



Sometimes it is useful to also get OSC messages out of a patch, for example to control other OSC software through Pd. For this the **OSC-outlet** of `originator` can be used, which is the rightmost outlet of the abstraction. It will print out every change to the current state. Connecting a `print OSC` debug object to it, we get to see what's coming out of the OSC-outlet when we move a slider:

```
OSC: /mypatch/pan 92
OSC: /mypatch/pan 91
OSC: /mypatch/pan 90
OSC: /mypatch/pan 89
```

Putting it all to RRADical use

Now that the foundation for a general preset and communication system are set, it is possible to build real patches with it that have two main characteristics:

Rapidity Ready-to-use high-level abstraction can save a lot of time when building larger patches. Clear communication paths will let you think faster and more about the really important things.

Reusability Don't reinvent the wheel all the time. Reuse patches like instruments for more than one piece by just exchanging the Caretaker-file used.

I already developed a growing number of patches that follow the RRADical paradigm, among these are a complex pattern sequencer, some synths and effects and more. All those are available in the Pure Data CVS, which currently lives at pure-data.sourceforge.net in the directory "abstractions/rradical". The RRADical collection comes with a template file, called `rrad.tpl.pd` that makes deploying new RRADical patches easier and lets developers concentrate on the algorithm instead of bookkeeping. Some utilities help with creating the sometimes needed many `commun`-objects. Several usecases show example applications of the provided abstractions.

Much, but not all is well yet

Developing patches using the Memento system and the design guidelines presented has made quite an impact on how my patches are designed. Before Memento quite a bit of my patches' content dealed with saving state in various, crude and non-unified ways. I even tried to avoid saving states at all because it always seemed to be too complicated to bother with it. This limited my patches to being used in improvisational pieces without the possibility to prepare parts of a musical story in advance and to “design” those pieces. It was like being forced to write a book without having access to a sheet of paper (or a harddisk nowadays). This has changed: having “paper” in great supply now has made it possible to “write” pieces of art, to “remember” what was good and what rather should not be repeated, to really “work” on a certain project over a longer time.

RRADical patches also have proven to be useful tools in teaching Pure Data, which is important as usage of Pd in workshops and at universities is growing – also thanks to its availability as Free Software. RRADical patches directly can be used by novices as they are created just like any other patch, but they already provide sound creation and GUI elements that the students can use immediately to create more satisfactory sounds that the sine waves used as standard examples in basic Pd tutorials. With a grown proficiency the students later can dive into the internals of a RRADical patch to see what’s inside and how it was done. This allows a new top-down approach in teaching Pd which is a great complement (or even alternative) to the traditional, bottom-up way.

Still the patches suffer from a known technical problem of Pd. Several of the RRADical patches make heavy use of graphical modules like sliders or number boxes, and they create a rather high number of messages to be send inside of Pd. The message count is alleviated a bit by using OSC, but the graphical load is so high, that Pd’s audio computation can be disturbed, if too many GUI modules need updating at the same time. This can lead to dropouts and clicks in the audio stream, which is of course not acceptable.

The problem is due to the non-sufficient decoupling of audio and graphics rsp. message computations in Pd, a technical issue that is known, but a solution to my knowledge could require a lot of changes to Pd’s core system. Several developers already are working on this problem, though.

The consistent usage of OSC throughout the RRADical patches created another interesting possibility, that of collaboration. As every RRADcial patch not only can be controlled through OSC, but also can control another patch of its own kind, the same patch could be used on two or more machines, and every change on one machine would propagate to all other machines where that same patch is running. So jamming together and even the concept of a “Pd band” is naturally build into every RRADcial patch.

RECOMBINANT SPATIALIZATION FOR ECOACOUSTIC IMMERSIVE ENVIRONMENTS

Matthew Burtner and David Topper,

VCCM, McIntire Department of Music,
University of Virginia
Charlottesville, VA 22903 USA

mburtner@virginia.edu, topper@virginia.edu,

ABSTRACT

An approach to digital audio synthesis is implemented using recombinant spatialization for signal processing. This technique, which we call Spatio-Operational Spectral Synthesis (SOS), relies on recent theories of auditory perception, especially research by Kubovy and Bregman. In SOS, the perceptual spatial phenomenon of objecthood is explored as an expressive musical tool. In musical applications of these theories, we observe the emergence of a "persistence of audition" exposing interesting opportunities for compositional development.

In essence, SOS, breaks an audio signal into salient components then recombines and spatializes them in a multichannel environment. Following an introduction to the technique and several examples demonstrating potential applications, this paper concentrates on some applications of the technique in ecoacoustic compositions by Matthew Burtner, *Anugi Unipkaaq, Sikniq Unipkaaq and Siku Unipaaq.* These works draw on environmental systems as models for multichannel processing.

1. INTRODUCTION

Spatial techniques in music composition can be traced at least to the 16th century. In the Venetian polychoral antiphonal tradition in the late 16th and early 17th centuries, composers composed for multiple choruses set around the space, creating a *cori spezzati* or *split chorus.* From the two choir works of Willaert, ca. 1580 the tradition of Cori Spezzati evolved into an ellaborate practice in the music of Giovanni Gabrieli.

The electroacoustic multichannel tradition has roots back to Varese's *Poeme Electronique* (1958) in which over 400 loudspeakers routed multichannel sound throughout the Philips Pavilion in the Brussels World Fair. These techniques, including the more recent practices of electroacoustic music, have concentrated on the projection of coherent sound object or objects into a defined space.

Spatio-Operational Spectral Synthesis or SOS, is a signal processing technique based on recent psychoacoustic research. The literature on auditory perception offers many clues to the psychoperceptual interpretation of audio objecthood as a result of streaming theory. Streaming describes audio objects as sequences displaying internal consistency or continuity (McAdams and Bregman 1979). Bregman has further defined a stream as, "a computational stage on the way to the full description of an auditory event. The stream serves the purpose of clustering related qualities (Bregman, 1999)." Thus it becomes the primary defining factor of an acoustic object.

SOS breaks apart an existing algorithm (ie, Additive Synthesis, Physical Modeling Synthesis, etc.) into salient spectral components, with different components being routed to individual or groups of channels in a multichannel environment. Due to the inherent limitations of audition, the listener cannot readily decode the location of specific spectra, and at the same time can perceive the assembled signal. In this sense, the nature of the auditory object is altered by situating it on the threshold of streaming, between unity and multiplicity.

The "Theory of Indispensable Attributes" (TIA) proposed by Michael Kubovy (Kubovy and Valkenburg, 2001) puts forth a framework for evaluating the most critical data the mind uses to process and identify objects. In the case of audio objects, TIA holds that pitch is an indispensable attribute of sound while location is not, simply put, because the perception of audio objects can not exist without pitch. His experiments have demonstrated that pitch is a descriminating factor the brain seems to use in distinguishing sonic objecthood, whereas space is not as critical.

Bregman notes that conditions can be altered to make localization easier or more difficult, so that, "conflicting cues can vote on the grouping of acoustic components and that the assessed spatial location gets a vote with the other cues. (Bregman p305)": " Curious about how Kubovy's and Bregman's theories could be utilized for signal processing, we began applying spatial processing algorithms to spectral objects.

When spectral parameters are spatialized in a certain manner the components fuse and it is impossible to localize the sound, yet when they are spatialized differently the localization or movement is predominant over any type of spectral fusion. Creatively modulating between fusion and separation is where SOS comes into being. One of our main questions is this: if the mind does not treat location as indespensible, can SOS force the signal into an oscillation between unity and multiplicity by exploiting spatialization of the frequency domain?

The technique exploits what might be called a "Persistence of Audition" insofar as the listener is aware that auditory objects are moving, but not always completely aware of where or how. This level of spatial perception on the part of the listener can also be controlled by the composer with specific parameters.

SOS is essentially a two-step operation. Step one consists of taking an existing synthesis algorithm and breaking it apart into logical components. Step two re-assembles the individual components generated in the previous step by applying various spatialization algorithms. Figure 1 illustrates the basic notion of SOS as demonstrated in the following example of a square wave.

2. SOS ADDITIVE SYNTHESIS

In initial experiments testing SOS we used simple mathematical audio objects such as a square wave generated by summing together sinusoids having odd harmonics and inversely proportional amplitudes. Formula (1) describes the basic formula used in this initial example:

$$x_s(t) = \sin(w_0 t) + 1/3 \sin(3w_0 t) + 1/5 \sin(5w_0 t) \ldots \tag{1}$$

In this experiment the first eight sine components of the additive synthesis square wave model were separated out and assigned to a specific speaker in an eight-channel speaker array. Although the square wave is spatially separated, summation of the complex object is accomplished by the mind of the listener (Figure 1).

Separation need not be completely discrete however. Any number of sinusoids can be used and animated in the space, sharing speakers. In a simple extension of this example sinusoids were used to generate a sawtooth wave as shown in Formula (2).

$$x_s(t) = \sin(w_0 t) + 1/2 \sin(2w_0 t) + 1/3 \sin(3w_0 t) \ldots \tag{2}$$

When the sinusoids were played statically, in separate speakers, the ear can identify the weighting of the frequency spectrum between different speakers. For example, if the fundamental is placed directly in front of the listener and each subsequent partial is placed in the next speaker clockwise around the array, a slight weighting occurs in the right front of the array. The First Wavefront law would of course suggest this, but in actuality the blending of the sinusoids into a square wave is more perceptible than the sense of separation into components. In fact, the effect is so subtle that a less well-trained ear still hears a completely synthesized square wave when listening from the center of the space.

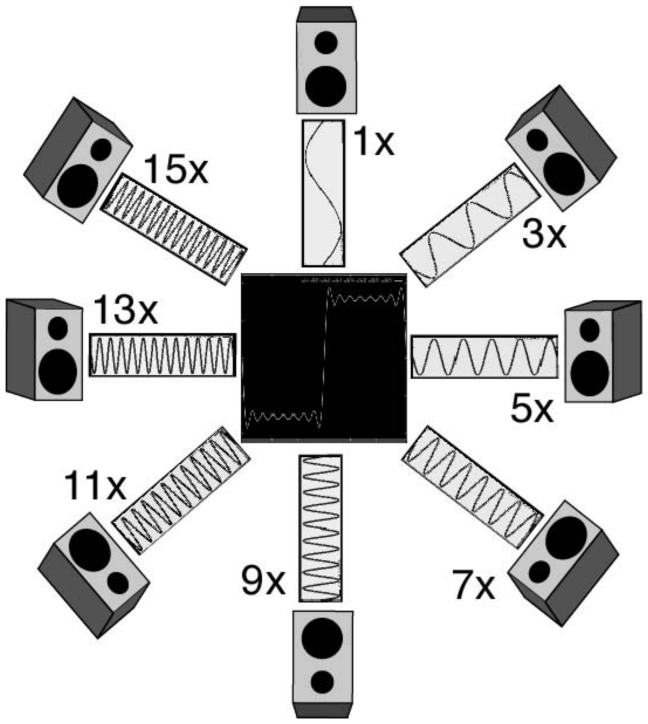


Figure 1. *SOS Recombinant Principle.*

Animating each of the sinusoids in a consistent manner exhibits a first example of the SOS effect. By assigning each harmonic a circular path, delayed by one speaker location in relation to each preceding harmonic, the unity of the square wave was maintained but each partial also began to exhibit a separate identity. This of course is the result, in part, of phase and shifting (eg., circularly moving) amplitude weights. The mind of the listener, tries to fuse the components while also attempting to follow individual movement.

This simple example illustrates how the Precedence Effect can be confused so that the mind simultaneosly can cast conflicting cognitive votes for oneness and multiplicity in the frequency domain. This state of ambiguity, as a result of spatial modulation, is what we call *the SOS effect*.

We experimented with different rates of circular modulation of each sine component. Interestingly, each relationship was different but not necessarily more pronounced than the similar, delayed motion. Using the same, non-time-varying signal, a time-varying frequency effect can be achieved due to spatial modulation using only circular paths in the same direction. Figure 2 illustrates this type of movement.

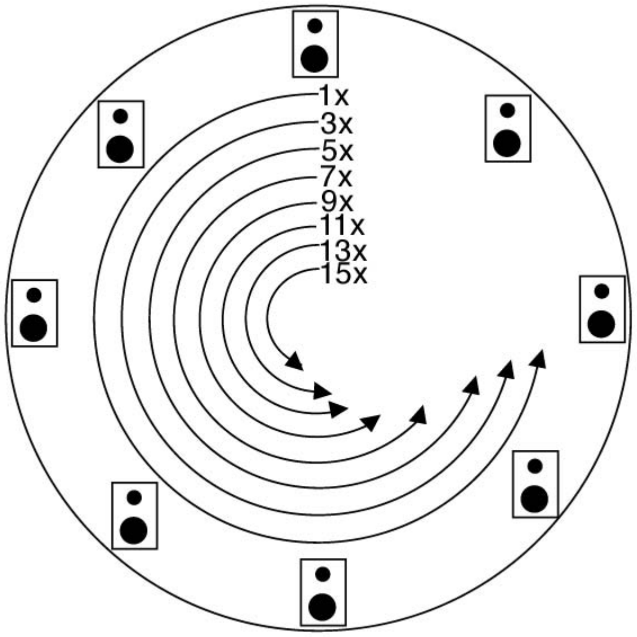


Figure 2. SOS with varying rate circular spatial path of the first eight partials of a square wave

An early example of spectral separation of this sort has been implemented in Roger Reynolds' composition, *Archepelago* (1983) for orchestra and electronics (Bregman p296). In tests done at the IRCAM, Reynolds and Thiery Lancino divided the spectrum of an oboe between two speakers and added slight frequency modulation to each channel. If the FM were the same in both channels the sound synthesized, but if different FM were added to each channel, the sounds divided into two independent auditory objects.

In our later tests, we noticed similar results to Reynolds and Lancino, even within the context of animated partials. By exaggerating the movement of one partial, either by increasing its rate of revolution, or assigning it a different path, the partial in question stood out and the SOS effect was somewhat reduced. By varying the amount of oscillation and specific paths of different partials, the SOS effect can be changed subtly.

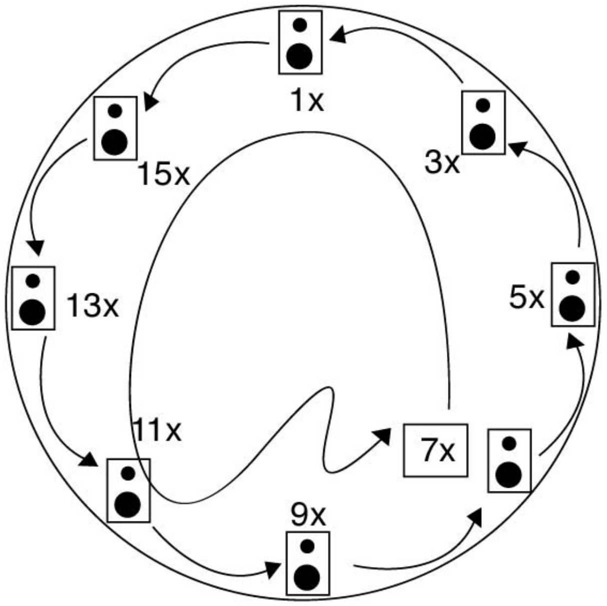


Figure 3. SOS with one partial moving against the others moving in a unified circular motion.

3. DEFINITIONS OF SOS SPATIAL ARCHETYPES

Any number of spatialization algorithms can be applied to the separated components' variables or audio stream. The types of spatialization employed by SOS can be thought of as having two attributes: motion and quality. A series of archetypal quality attributes were explored in a two dimensional environment.

Motion was divided into three categories:

1) static: no motion
2) smooth: a smooth transition between points
3) cut: a broken transition between points

Quality was divided into five archetypical forms:

1) circle: an object defines a circular pattern
2) jitter: an object wobbles around a point
3) across: an object moves between two speakers
4) spread: an object splits and spreads from one point to many points
5) random: an object jumps around the space between randomly varying points

These archetypes can be applied globally, to groups, or to individual channels. Each archetype has specific variables that can be used to emphasize or de-emphasize the SOS effect. Variables can also be mapped to trajectory or rate of change, defined by a time-varying function, or generated gesturally in real time.

4. SOS FILTER SUBBAND DECOMPOSITION

The balance between frequency separation and sonic object animation became much more complicated when we attempted to apply our initial technique to an audio signal. Our initial tests assigned eight simple two pole IIR filter outputs to discrete speaker locations. Selection of the ration between the filters became a critical component in being able to achieve any effect at all. With filters set to frequencies that were not very strong in the underlying signal, the filters tended to blend together and sound as if some type of combined filtering were taking place rather than SOS. Similarly, when spatialization algorithms were applied with an improper filter weight, the underlying movement was more apparent than the separation.

We tested the filter technique with both white noise and live instrument (eg., Tenor Saxophone). The former of course offered much more flexibility with respect to frequency range and filter setup. The saxophone signal used, having the majority of its spectrum located between 150Hz and 1500Hz (with significant spectral energy up to approximately 8000Hz) suggested a filter/bandwidth weighting of: 32/5Hz, 65/15Hz 130/30Hz, 260/60Hz, 520/120Hz, 1000/240Hz, 2000/500Hz, 4000/1000Hz.

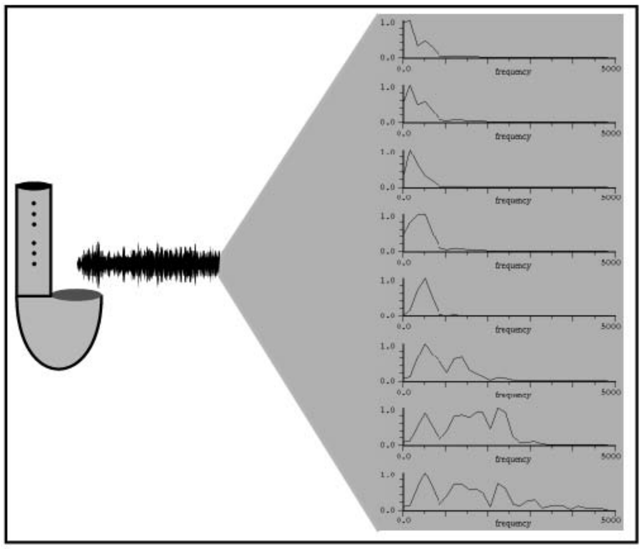


Figure 4: Saxophone signal subband filter decomposition for SOS.

5. SOS ECOACOUSTIC EMMERSIVE ENVIRONMENTS=

Multichannel composition has a basis in acoustic ecology through Soundscape composition (Truax, 1978/99, 1994). Multichannel soundscape compositions reconstruct sonic environments through the sampling and redistribution of distinct sounds to construct externally referential environments. A related area of research is ecoacoustics, an approach that derives musical procedures from abstract environmental systems, remapping data into structural musical material. it is a form of sonification for ecological models (Keller 1999, 2000).

In the most general sense, ecoacoustics is a type of environmentalism in sound, an attempt to develop a greater understanding of the natural world through close perception. In the field of composition, this takes the form of musical procedures and materials that either directly or indirectly draw on environmental systems to structure musical material.

In *Winter Raven* (Burtner 2001), a large scale work for instrumental ensemble, 8-channel computer-generated sound, three video projections, dance and theater, SOS techniques were implemented in a multimedia context. Each of the three acts of *Winter Raven* contains one *Unipkaaq* or “story” in Unupiaq Inuit language. Each of these pieces is scored for 8-channel computer-generated sound using SOS techniques, percussion, and a dancer wearing a specially constructed mask. The masked dancer represents a magical character playing a shamanic role in the evolution of the piece.

The Shaman character uses three different masks in *Ukiuq Tulugaq*, representing Sun, Ice and Wind. Each mask is distinguished by different choreography, music and video processing. An interface written with Isadora, processes the incoming live video and layers it with prerecorded video. The electronics from these three movements contain different SOS processing of the electronic sound. Each spatialization model corresponds to a dance mask with interactive video. The combination of video and multichannel audio evoke a personification of the environmental elements of sun, ice and wind. In Figure 8, the live video is shown above the corresponding staged scene.

In the first of these three pieces, *Siknik Unipkaaq (the story of sun),* a group of interlocking concentric planal paths were created (figure 5).

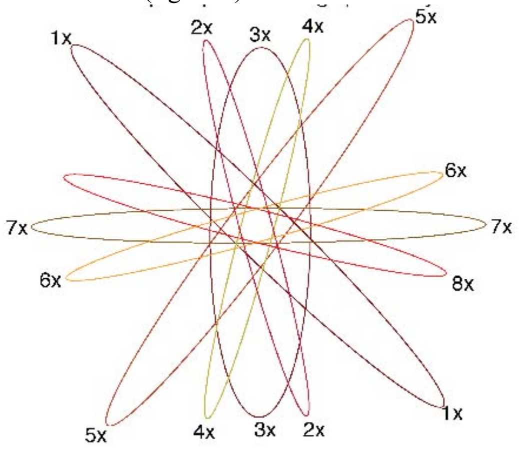


Figure 5: *Siknik Unipkaaq* SOS processing

Spatial modulation tempo ratios of 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 were employed for the eight independent paths of audio. The base tempo of the structure was modulated globally, accelerating from a time base of 1 = 120” to a time base of 1 = 20”. This yields a meta-tempo structure of 120” : 60” : 40” : 30” : 24” : 20” : 17” : 15” which is gradually collapsed into a mesa-tempo structure of 20” : 10” : 6.7” : 5” : 4” : 3.3” : 2.8” : 2.5”.

In addition to the electronics, a battery of percussion helps articulate the perpetual motion of this composition. Two percussionists playing timpani and cymbals create slow crescendo/decrescendo pulses. Two other percussionists play congas, bass drum and floor toms, following a repetitive pattern derived from the spatial motion. Both the repeated dynamic changes of the timpani/cymbals and the repeated rhythmic patterns of the drums, help underscore the cyclical motion of the computer-generated sound.

In *Siku Unipkaaq (the story of ice)* a “shaking” algorithm was employed to model the freezing of motion in the spatial domain. Each component of the ice sound pans between two randomly selected points very rapidly and gradually reduces movement, increasing frequency. The panning occurs on the order of 600 to 20 milliseconds, varying for each particle of sound. The result is a feeling of gravity pulling the sound towards a single point between the two spatial anchors. Thus the sound is “frozen” into multifaceted crystals, continually spawning new paths that are again frozen. At any given time there are four simultaneous paths of shaking. In addition, the ice sound is played out of each speaker quietly to create a background into which the shaking algorighm can blend smoothly. Figure 6 depicts this motion type.

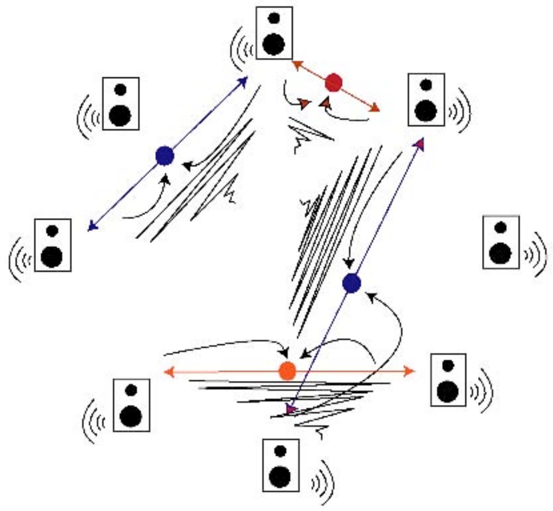


Figure 6: *Siku Unipkaaq* SOS “shaking” algorithm

A global freezing process is created by two glockenspiel played by four players. Over the course of the four minutes of the piece, the density and variety of pitches are reduced, focussing the frequency energy into reduced bands of sound. Finally, the voices slow and freeze into individual points in the frequency spectrum.

Anugi Unipkaaq (the story of wind) most effectively captures the principle of SOS in this group. The source material of the work is the sound of wind recorded in Alaska. The wind is band pass filtered to isolate individual frequency regiouns of the sound. In this sense it is treated as the saxophone signal in the experiment discussed previously. Four such independent wind bands are created from the original source.

Each excerpted wind channel is panned rapidly between groups of randomly selected speakers. The path accelerates lograithmically, speeding up as it approaches its target point. In figure 7, each straight line represents this accelerating curve. Amplitude is tied to spatial change such that the wind sounds crescendo into each new location. The bands of wind rush simultaneously around the space, creating a kind of SOS blizzerd of wind.

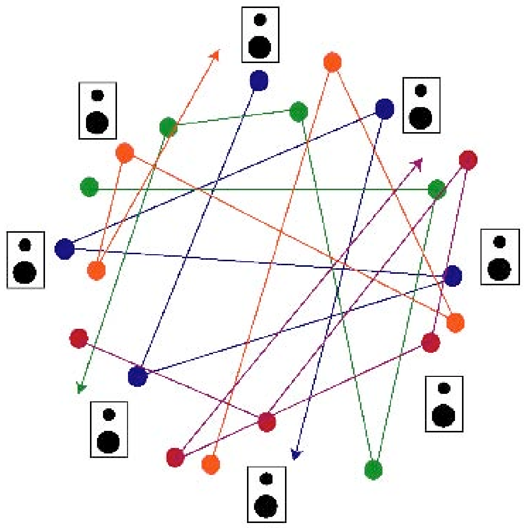


Figure 7: *Anugi Unipkaaq* SOS spatial motion “blizzard” algorithm

Accompanying the spatialized four winds are four percussionists. The piece is scored for a solo percussionist who plays a battery of toms and drums. The other three players are gathered around a single large bass drum, playing it simultaneously. At the end of the piece, as the rhythmic structure concentrates into a single common rhythm, the solo percussion joins the other players at the large bass drum and they end together. The four players focussed around a single point on the stage create a kind of focus for the four winds thrashing around the hall.

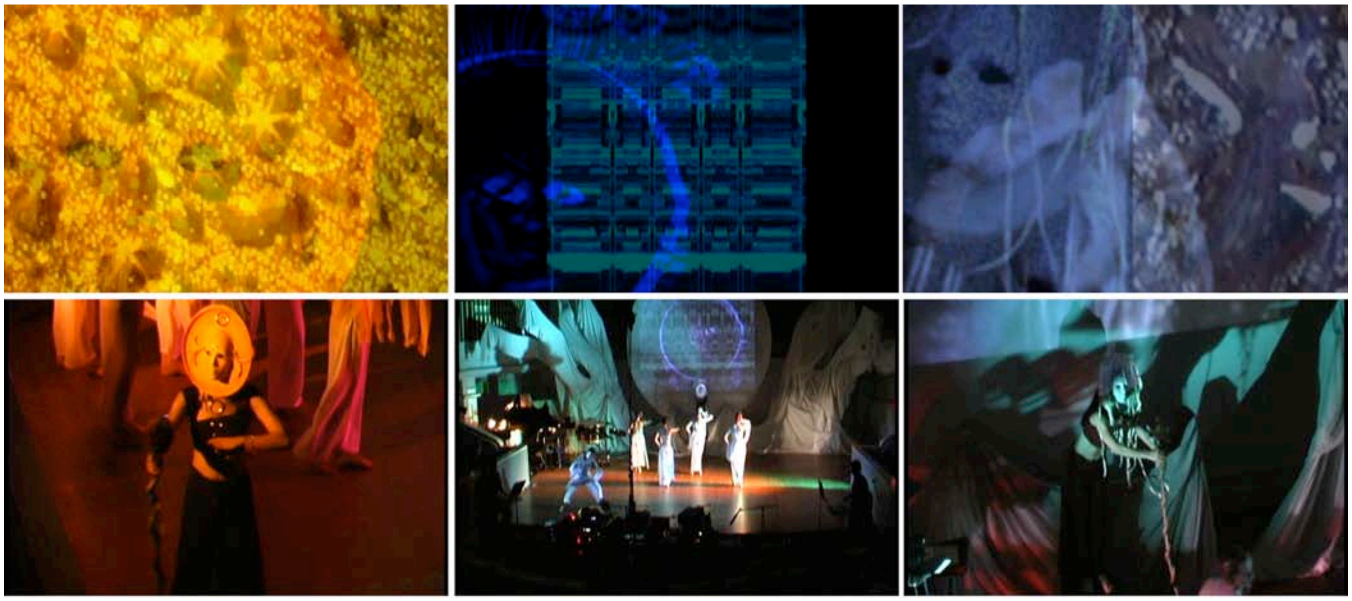


Figure 6. Each column above shows the processed video (above) and mask dancer (below). The rows from left to right show: *Siknik Unipkaaq (the story of sun)* *Siku Unipkaaq (the story of ice)* *Anugi Unipkaaq (the story of wind)*

6. FUTURE DIRECTIONS

Current SOS research has been done primarily in a two dimensional environment. Exploring a three dimensional environment will increase the effect of spatialization algorithms and offer a greater means of separation for various models (ie, 3D waveguides).

So far, only the authors who agreed on the results have performed listening tests. Future work consists of testing more subjects, in order to see if the segregation of the synthesis algorithms is performed in the same way by human listeners.

Much of the psychoacoustic research that inspired SOS also looks at the related phenomenon of audio streaming, in sequential segregation. In addition to exploring SOS based on "spectral" separation, it would be interesting to explore sequential stream separation and granular synthesis.

With respect to the creative applications of SOS, the work described here has relied on macro-level procedures and more work on micro-level structures (eg particle-based synthesis) is anticipated. In addition, stronger and more concrete sonification algorithms will help articulate the ecoacoustic compositional strategies. Further integration of the video aspects of the works with SOS would also be advantageous.

7. REFERENCES

[1] A. S. Bregman. *Auditory Scene Analysis: the perceptual organization of sound.* MIT Press, Cambridge, MA, 1999.

[2] M. Burtner, *Ukiuq Tulugaq (Winter Raven).* Doctoral of Musical Arts Thesis. Stanford University, Stanford, California. 2001.

[3] M. Burtner, D. Topper, S. Serafin. *S.O.S. (Spatio-Operational Spectral) Synthesi).* Proceedings of the Digital Audio Effects (DAFX) Conference. Hamburg, Germany, 2002.

[4] D. Keller. *Social and perceptual dynamics in ecologically-based composition*. Proceedings of the VII Brazilian Symposium of Computer Music, Curitiba, PN: SBC. 2000.

[5] D. Keller, (1999). *touch'n'go: Ecological Models in Composition.* Master of Fine Arts Thesis. Burnaby, BC: Simon Fraser University. 1999.

[6] M. Kubovy, D. V. Valkenburg. "Auditory and Visual Object*s," Cognition*. 80, p97-126. 2001.

[7] S. McAdams, and A. Bregman. "Hearing Musical Streams." *Computer Music Journal.* vol. 3 num. 4. CA., 1979.

[8] B. Garton, and D. Topper. "RTcmix -- Using CMIX in Real Time," Proc. of *International Computer Music Conference* (ICMC), Thesalonika, Greece*,* 1997.

[9] D. Topper. "PAWN and SPAWN (Portable and Semi Portable Audio Workstation)." Proc. of *International Computer Music Conference* (ICMC), Berlin, Germany., 2001.

[10] B. Truax. ed. “Handbook for Acoustic Ecology.” Arc Publications, Cambridge Street Publishing, CD-ROM Edition, Version 1.1. 1978/1999.

[11] B. Truax. "Discovering Inner Complexity: Time-Shifting and Transposition with a Real-time Granulation Technique," *Computer Music Journal,* 18(2), 1994, 38-48 (sound sheet examples in 18(1)).

“Once again text & parenthesis – sound synthesis with Foo”

Gerhard Eckel
Ramón González-Arroyo
Martin Rumori

April 30, 2004

Foo is a sound synthesis tool based on the Scheme language, a clean and powerful Lisp dialect. **Foo** is used for high-quality non-realtime sound synthesis and -processing. By scripting **Foo** like a shell it is also a neat tool for implementing common tasks like soundfile conversion, resampling, multichannel extraction etc.

Note:
According to the talk at the Linux Audio Conference, this text will mainly cover the **Foo** kernel layer. This is because the main author of this text, Martin Rumori, is mostly involved with porting and developing the **Foo** kernel. Quotation from [5]:

> Whereas the **Foo** kernel layer implements the generic sound synthesis and processing modules as well as a patch description and execution language, the **Foo** control layer offers a symbolic interface to the kernel and implements musically salient control abstractions.

Find out more about the **Foo** control layer in [4] and [5] and the **Foo** control layer's source code at [1].

1 Introduction

When the **Foo**-project evolved at ZKM Karlsruhe in 1993, nobody knew how to call it. Just to be able to talk about, it got the working title “foo” according to RFC 3092. When it came to the first publicly available version, its “nickname” had got deep into the slang of the authors, and considering that “foo” may stand for the two main programming paradigms used in it (functional and object oriented) it was decided, not without some irony, to leave it as its name. Due to that, the installation of **Foo** at IRCAM's computers was refused by their administrator first…

Since the *SourceForge* team has been granted the takeover of the sample project “foo” when the program was ten years old in late 2003, the existence of `/usr/bin/foo` is legalized now. To avoid confusion, we discourage from using the term “foo” as a general sample name in the future :-)

2 History of Foo

Foo was developed by Gerhard Eckel and Ramón González-Arroyo at ZKM Karlsruhe in 1993. Its development was continued by the authors in the context of an institutional collaboration between IRCAM and ZKM until 1996. At that time, machines by *NeXT* running NeXTStep were the computers of choice for those tasks. Since NeXTStep is based on Objective-C, the kernel part of **Foo** was written in that language as well.

The original motivation was the lack of a high quality tool providing techniques known from the analogue audio tape, such as varispeed playback. In the digital domain, the most crucial point with those techniques is the *resampling algorithm*. Unlike other sound synthesis programs (e. g. *Csound* with the *oscil**- or *phasor/table**-opcodes), **Foo** allowed for scalable, high quality resampling using the Sinc-Interpolator [6] from the very beginning [1].

After an infrastructure for accessing these key features *musically meaningful* had been designed and implemented, additional functionality was integrated with **Foo**, such as oscillators and filters. Thanks to its open and extensible design, **Foo** got a standalone, general purpose sound synthesis system.

3 Key concepts of Foo

3.1 Patch generation

Foo was also inspired by patch based sound synthesis systems such as *Max/MSP*. Similar to an analogue synthesizer, different basic modules are connected in order to form more complex signal processing entities.

Unlike *Max*, **Foo** does not copy the wiring process of an analogue synthesizer one-to-one to the screen. In fact, **Foo** is meant as a *patch generation language*. Currently, this language is *Scheme* [7], a clean and powerful *LISP* dialect. *Scheme* allows for patch generation in several abstract ways, such as recursion and high order functions. It is very easy to build patch templates which are instantiated several times with different parameters, which is quite hard with graphical languages like *Max*.

In **Foo**, everything is a signal. There is no distinction between audio rate and control rate, since that inherently holds the risk of aliasing artefacts. There are *Scheme* bindings for the constructors of each available *module* (unit generator), which evaluate to the signal produced by that modules. This value in turn may be used as an input for another module constructor.

3.2 Context

Dealing with patches in **Foo** is done via so called *contexts*. A *context* is kind of a container for a patch, which allows for treating a patch as an entity. From outside a *context*, the resulting signal of a complete patch is accessible via the *output* modules of that patch only.

A *context* is also a means for "executing" the associated patch. Using a *task*, one can render a *context* into a soundfile. Therefore a *context* represents exactly the sound it can produce, it is somehow a compressed *description* of the sound.

With **Foo**, it is possible to save such *contexts* in a binarily serialized form and load them again into the runtime system. This is useful especially when working with *incremental mixing* (see 3.5), since you don't have to keep all the interim versions as probably large, space-wasting soundfiles.

3.3 Time

Each **Foo** patch is associated with a *context*. This is visible for the *output* modules as well as for *temporal* relations.

A **Foo** *context* has a local *time origin*, which is zero. Any patch structures inside the *context* are temporally related to this time origin by specifying a time shift. This shift can be positive or negative; the *context's* time axis reaches from negative to positive infinity.

Time shifts can be nested, so that every shift refers to the outer time frame (with the *context's* time origin as outmost frame).

[1]As of 2002, *Csound* allows for sinc interpolation by means of the *tablexkt* opcode

3.4 Task

A **Foo** *context* containing a complex patch is just a description for creating e. g. a sound file. A *context* itself knows nothing about the concept of sampling rate, sample format, soundfile headers etc. Thus a *context* is an abstract description which could be used in several different environments after it has been constructed.

A **Foo** *task* is an execution controller for a *context*. All the above mentioned parameters are set by the *task* object when binding the *context* to an output medium (currently a sound file, which provides those settings like sample rate and -format).

A *task* also provides a means for handling the *context's* time model (even **Foo** is not really able to create sound files with an infinite duration…). This is done by two temporal related parameters of the *task* constructor: the *reference* and the *offset* values. The *reference* determines where in the associated output sound file the time origin of the *context* should be anchored, while the *offset* parameter specifies at which position in the *context's* time axis the rendering process should start. Together with the *duration* parameter of the *task's* rendering process, one can specify exactly which *part* of the *context* is being rendered.

3.5 Incremental mixing

The clean semantics of *task*, *context* and *time* in **Foo** allows for another neat feature: the *incremental mixing*. Consider *contexts* different layers of a composition, you might want to be able to *incrementally* construct the final composition out of these layers and perhaps do later corrections to one of the layers.

With **Foo**, a *task* is not just able to render a *context* into a new soundfile, but can also *add* the resulting sound material into an existing file at a specified time (via the *reference* parameter). When archiving each of the involved *contexts* along with the resulting file, it is later possible to render one of the layers again into a temporary file and to subtract it from the resulting file. This way, it's possible to do later corrections in the layer layout of a composition without having to keep all the intermediate versions and single layers as sound data.

3.6 Scripting Foo

Another one of the charming features of **Foo** is its scriptibility. Unlike with other sound synthesis systems, one does not necessarily has to enter the **Foo** environment (in other words, the **Foo** command line prompt) for doing sound synthesis tasks. Instead, it's possible to write "**Foo** scripts" like shell scripts, which could then be used as standalone signal processing applications. This is extremely useful for recurring basic tasks, like resampling, or for batch processing in general.

Similar to a shell script, a directive like `#!/usr/local/bin/foo` sets **Foo** as the interpreter for a script. The argument vector issued when calling the script is accessible via the `(command-line-args)` function from inside the script. That way it is possible to create complex scripts which are seemlessly integrated with the usual shell environment.

4 Future plans

The last substantial changes to **Foo** were made in 1996. Now, with having been ported to *Linux* and *Mac OS X*, **Foo** gets faced with a completely different world compared to that of the middle nineties…

4.1 Dynamically loadable modules

One major goal is to create a more flexible *module* interface for **Foo**. By now, the signal processing modules are compiled into the **Foo** kernel. A solution with dynamically loadable modules would make it easier for developers to add new modules to **Foo**.

This would also allow for interfacing with other DSP systems, such as an interface to *LADSPA* [8]. We are also thinking of using *Faust* [9] as a means for building modules for **Foo**.

4.2 Typed signals

To improve the flexibility of **Foo**, we think of introducing signal types other than audio signals, so that control signals, triggers etc. could be used more efficiently.

4.3 Further modularization of Foo

Currently, the **Foo** kernel consists of a single library, which is an extension to the *elk* [10] Scheme interpreter. To allow for more flexible use of **Foo**, it will see some restructuring.

The **Foo** kernel will be a library written in Objective-C, which could be used for other applications, too. The interface to the *elk* interpreter will be done via another lightweight library as an extension. This way, **Foo** could be easily interfaced to other Scheme interpreters as well as other languages in general.

4.4 Jack interface

After having ported **Foo** to *Linux*, the preliminary direct play support (via the `(play~)` module) was disabled.

To ease the process of composing with **Foo**, we think of creating an interface to the *jack* audio server [11]. This would mean just a sound file player which is better integrated with **Foo** than other players; it will *not* mean realtime rendering capabilities for **Foo**. In conjunction with an interface to the *jack* transport API, rendering and playing could be triggered by jack events, which makes using sound material created with **Foo** in other applications more comfortable.

References

[1] **Martin Rumori**: *Foo Website*,
`http://foo.sourceforge.net`

[2] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Kernel Concepts*, ZKM, Karlsruhe 1996

[3] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Kernel Reference Manual*, ZKM, Karlsruhe 1994

[4] **Gerhard Eckel, Ramón González-Arroyo**: *Foo Control Reference Manual*, ZKM, Karlsruhe 1993

[5] **Gerhard Eckel, Ramón González-Arroyo**: *Musically Salient Control Abstractions for Sound Synthesis*, Proceedings of the 1994 International Computer Music Conference, Aarhus, 1994

[6] **Julius O. Smith**: *The Digital Audio Resampling Home Page*,
`http://www-ccrma.stanford.edu/~jos/resample/`

[7] **'(schemers . org)**: *An improper list of Scheme resources*,
`http://www.schemers.org`

[8] **LADSPA**: *Linux Audio Developer's Simple Plugin API*,
`http://www.ladspa.org`

[9] **E. Gaudrain, Y. Orlarey**: *A FAUST Tutorial*,
`ftp://ftp.grame.fr/pub/Documents/faust_tutorial.pdf`

[10] **Oliver Laumann et al.**: *Elk: The Extension Language Kit*,
`http://sam.zoy.org/projects/elk/`

[11] **Paul Davis et al.**: *Jack: The Jack Audio Connection Kit*,
`http://jackit.sourceforge.net`

Developing spectral processing applications

Victor Lazzarini,
MTL, NUI Maynooth, Ireland
1. May 2004

Abstract

Spectral processing techniques deal with frequency-domain representations of signals. This text will explore different methods and approaches of frequency-domain processing from basic principles. The discussion will be mostly non-mathematical, focusing on the practical aspects of each technique. However, wherever necessary, we will demonstrate the mathematical concepts and formulations that underline the process. This article is completed with an overview of the spectral processing classes in the Sound Object Library. Finally, A simple example is given to provide some insight into programming using the library.

1. The Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is an analysis tool that is used to convert a time-domain digital signal into its frequency-domain representation. A complementary tool, the IDFT, does the inverse operation. In the process of transforming the spectrum, we start with a real-valued signal, composed of the waveform samples and we obtain a complex-valued signal, composed of the spectrum samples. Each pair of values (that make up a complex number) generated by the transform is representing a particular *frequency* point in the spectrum. Similarly, each single (real) number that composes the input signal represents a particular *time* point. The DFT is said to represent a signal at a particular time, as if it was a 'snapshot' of its frequency components.

One way of understanding how the DFT works its magic is by looking at its formula and trying to work out what it does:

$$DFT(x(n),k) = \frac{1}{N}\sum_{n=0}^{n-1} x(n) \times e^{-j2\pi kn/N} \quad k = 0,1,2,...,N-1 \tag{1}$$

The whole process is one of multiplying an input signal by complex exponentials and adding up the results to obtain a series of complex numbers that make up the spectral signal. The complex exponentials are nothing more than a series of complex sinusoids, made up of cosine and sine parts:

$$e^{-j2\pi kn/N} = \cos(2\pi kn/N) - j\sin(2\pi kn/N) \tag{2}$$

The exponent $j2\pi kn/N$ determines the phase angle of the sinusoids, which in turn is related to its frequency. When $k=1,$ we have a sinusoid with its phase angle varying as $2\pi n/N$. This will of course complete a whole cycle in N samples, so we can say its frequency is $1/N$ (to obtain a value in Hz, we just have to multiply it by the sampling rate). All other sinusoids are going to be whole-number multiples of that frequency, for $1 < k < N-1$. The number N is the number of points in the analysis, or the number of spectral samples (each one a complex number), also known as the *transform size*. Now we can see what is happening: for each particular frequency point k, we multiply the input signal by a sinusoid and then we sum all the values obtained (and scale the result by $1/N$).

Consider the simple case where the signal $x(n)$ is a sine wave with a frequency $1/N$, defined by the expression $\sin(2\pi n/N)$. The result of the DFT operation for the frequency point 1 is shown on fig.1. The complex sinusoid has detected a signal at that frequency and the DFT has output a complex value [0, -0.5] for that spectral sample (the meaning of –0.5 will be explored later). This complex value is also called the *spectral coefficient* for frequency $1/N$. The real part of this number corresponds to the detected cosine phase component and its imaginary part relates to the sine phase component. If we slide the sinusoid to the next frequency point ($k=2$) we will obtain the spectral sample [0, 0], which means that the DFT has not detected a sinusoid signal at the frequency ($2n/N$).

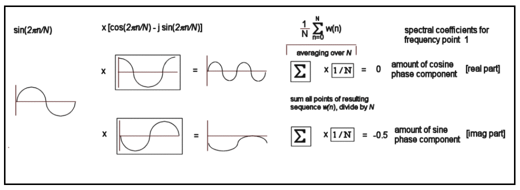

sin(2πn/N)
x [cos(2πn/N) - j sin(2πn/N)]
$\frac{1}{N}\sum_{n=0}^{N} w(n)$
spectral coefficients for frequency point 1
averaging over N
x
=
Σ x 1/N = 0
amount of cosine phase component
[real part]
sum all points of resulting sequence w(n), divide by N
Σ x 1/N = -0.5
amount of sine phase component
[imag part]

Figure 1. The DFT operation on frequency point 1, showing how a complex sinusoid is used to detect the sine and cosine phase components of a signal.

This shows that the DFT uses the 'sliding' complex sinusoid as a detector of spectral components. When a frequency component in the signal matches the frequency of the sinusoid, we obtain a non-zero output. This is, in a nutshell, how the DFT. Nevertheless, this example shows only the simplest analysis case. In any case, the frequency *1/N* is a special one, known as the *fundamental frequency of analysis*. As mentioned above, the DFT will analyse a signal as composed of sinusoids at multiples of this frequency.

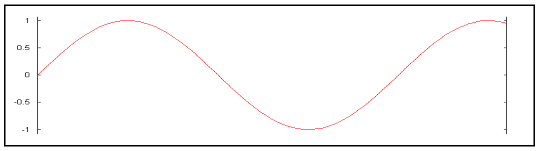


Figure 2. Plot of sin($2\pi 1.3n/N$)

Consider now a signal that does not contain components at any of these multiple frequencies. In this case, the DFT will simply analyse it in terms of the components it has at hand, namely the multiples of the fundamental frequency of analysis. For instance, take the case of a sine wave at *1.3/N*, sin($2\pi 1.3n/N$) (fig.2). We can check the result of the DFT on table 1. The transform was performed using the C++ code above with *N=16*.

point (k)	**real part** (re[X(k)])	**imaginary part** (im[X(k)])
0	0.127	0.000
1	0.359	0.221
2	-0.151	0.127
3	-0.071	0.056
4	-0.053	0.034
5	-0.046	0.022
6	-0.042	0.013
7	-0.041	0.006
8	-0.040	0.000
9	-0.041	-0.006
10	-0.042	-0.013
11	-0.046	-0.022
12	-0.053	-0.034
13	-0.071	-0.056
14	-0.151	-0.127
15	0.359	0.221

Table 1. Spectral coefficients for a 16-point DFT of sin($2\pi 1.3n/N$)

Although confusing at first, this result is what we would expect, since we have tried to analyse a sine wave, which is 1.3 cycles long. We can, however, observe that one of the two largest pairs of absolute values is found on points 1. From what we saw in the first example, we might guess that the spectral peak is close to the frequency *1/N*, as in fact it is (*1.3/N*). Nevertheless, the result shows a large amount of spectral spread, contaminating all frequency points (see also fig.3). This has to do with the discontinuity between the last and first points of the waveform, something clearly seen on fig.2.

1.1. Reconstructing the time-domain signal

The result in the table above can be used to reconstruct the original waveform, by applying the inverse operation to the DFT, the Inverse Discrete Fourier Transform, defined as:

$$IDFT(X(k),n) = \sum_{n=0}^{n-1} X(k) \times e^{j2\pi kn/N} \quad n = 0,1,2,...,N-1 \tag{3}$$

In other words, the values of *X(k)* are [complex] coefficients, which are used to multiply a set of complex sinusoids. These will be added together, point by point, to reconstruct the signal. This is, basically, a form of additive synthesis that uses complex signals. The coefficients are the amplitudes of the sinusoids (cosine and sine) and their frequencies are just multiples of the fundamental frequency of analysis. If we use the coefficients in the table above as input, we will obtain the original 1.3-cycle sine wave.

We saw above that point 1 refers to the frequency *1/N*, and point 2 to *2/N* and so on. As mentioned before, the fundamental frequency of analysis in Hz will depend on how many samples are representing our signal in a second, namely, the sampling rate (SR). So our frequency points will be referring to *k*SR/N Hz, with k=0,1,2,..., N-1.. So we will be able to quickly determine the frequencies for points 0 to N/2, ranging from the 0 Hz to SR/2, the Nyquist frequency, which is the highest possible frequency for a digital signal.

We can see in table 1 that points 9 to 15 basically have the same complex values as 7 to 1 (except for the sign of the imaginary part). It is reasonable to assume that they refer to the same frequencies. The sign of the imaginary parts indicates that they might refer to negative frequencies. This is because a negative frequency sine wave is the same as positive one with negative amplitude (or out-of-phase): sin(-*x*) = -sin(*x*). In addition, cos(-*x*) = cos(*x*), so the real parts are the same for negative and positive frequencies.

The conclusion is simple, the second half of the points refer to negative frequencies, from *–SR/2* to *–SR/N*. It is essential to point out that the point *N/2* refers to both *SR/2* and *–SR/2* (these two frequencies are indistinguishable). Also, it is important to note that the coefficients for 0 Hz and the Nyquist are always purely real (no imaginary part). We can see then that the output of the DFT then, splits the spectrum of a digital waveform in equally-spaced frequency points, or bands. The negative and positive spectral coefficients only differ in their imaginary part. For real signals, we can see that the negative side of the spectrum can always be inferred from the positive side, so it is, in a way, redundant.

1.2. Rectangular and polar formats

In order to understand further the information provided by the DFT, we can convert the representation of the complex coefficients, from real/imaginary pairs to one that is more useful to us. The amplitude of a component is given by the magnitude of each complex spectral coefficient. The magnitude (or modulus) of a complex number *z* is:

$$|z| = \sqrt{re[z]^2 + im[z]^2} \quad (4)$$

As a real signal is always split into positive and negative frequencies, the amplitude of a point will be ½ the 'true' value. The values obtained by the magnitude conversion are know as the *amplitude spectrum* of a signal. The amplitude spectrum of a real signal is always mirrored at 0 Hz.

The other conversion that complements the magnitude provides the phase angle (or offset) of the coefficient, in relation to a cosine wave. This yields the phase offset of a particular component, and it is obtained by the following relationship:

$$\theta(z) = \arctan\frac{im[z]}{re[z]} \quad (5)$$

The result of converting the DFT result in this way is called the phase spectrum. For real signals, it is always anti-symmetrical around 0 Hz. The process of obtaining the magnitude and phase spectrum of the DFT is called *cartesian-to-polar* conversion.

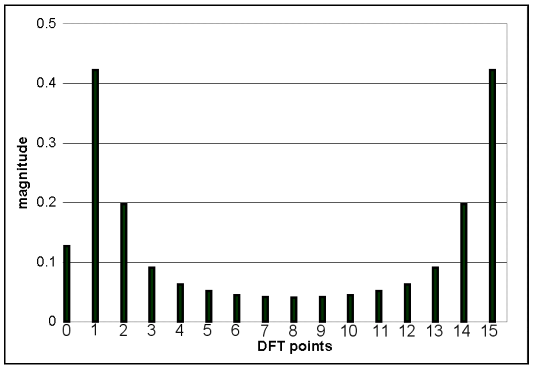


Figure 3. Magnitude spectrum from a 16-point DFT of sin(*2π1.3n/N*).

We can see from fig.3 that the DFT results are not always clear. In fact unless the signal has all its components at multiples of the fundamental frequency of analysis, there will be a spectral spread over all frequency points. In addition to these problems, the DFT in the present form, as a one-shot, *single-frame* transform, will not be able to track spectral changes. This is because it takes a single 'picture' of the spectrum of a waveform at a certain time. For a more thorough view of the DFT theory, please refer to (Jaffe, 1987a) and (Oppenheimer and Schafer, 1975).

2. Applications of the DFT: Convolution

The single-frame DFT analysis as explored above has one important application, the convolution of time-domain signals through spectral multiplication. Before we proceed to explore this technique, it is important to note that the DFT is very seldom implemented in the direct form shown above. More usually, we will find optimised algorithms that will calculate the DFT much more efficiently. These are called the Fast Fourier Transform (FFT). Their result is in all aspects, equivalent to the DFT as described above. The only difference is in the way the calculation is performed. Also, because the FFT is based on specialised algorithms, they will only work with a certain number of points (*N*, the transform size). For instance, the standard FFT algorithm uses only power-of-two (2,4,..., 512, 1024...) sizes. From now on, when we refer to the DFT, we will imply the use of a fast algorithm for its computation.

Convolution is an operation with signals, just like multiplication or addition, defined as:

$$w(n) = y(n) * x(n) = \sum_{m=0}^{n} y(m)x(n-m) \quad (6)$$

One important aspect of time-domain convolution is that it is equivalent to the multiplication of spectra (and vice-versa). In other words, if *y(n)* and *h(n)* are two waveforms whose fourier transforms are *Y(k)* and *H(k)*, then:

$$\begin{aligned} DFT[y(n) * h(n)] &= Y(k)H(k) \\ DFT[y(n)h(n)] &= Y(k) * H(k) \end{aligned} \quad (7)$$

This means that if the DFT is used to transform two signals into their spectral domain and the two spectra can be multiplied together, the result can be transformed back to the time-domain as the convolution of the two inputs. In this type of operation, we generally have an arbitrary sound that is convoluted with a shorter signal, called the *impulse response*. The latter can be thought of as a mix of scaled and delayed unit sample functions and also as the list of the gain values in a tapped delay-line. The convolution operation will impose the spectral characteristics of this impulse signal into the other input signal. There are three basic applications for this technique:

(1) Early reverberation: the impulse response is a train of pulses, which can be obtained by recording room reflections in reaction to a short sound.
(2) Filtering: the impulse response is a series of FIR filter coefficients. Its amplitude spectrum determines the shape of the filter.
(3) Cross-synthesis: the impulse response is an arbitrary sound, whose spectrum will be multiplied with the other sound. Their common features will be emphasized and the overall effect will be one of cross-synthesis.

Depending on the application, we might use a time-domain impulse response, whose transform is then used in the process. On other situations, we might start with a particular spectrum, which is directly used in the process. The advantage of this is that we can define the frequency-domain characteristics that we want to impose on the other sound.

2.1. A DFT-based convolution application

We can now look at the nuts and bolts of the application of the DFT in convolution. The first thing to consider is that, since we are using real signals, there is no reason to use a DFT that outputs both the positive and negative sides of the spectrum. We know that the negative side can be extracted from the positive, so we can use FFT algorithms that are optimised for the real signals. The discussion of specific aspects of these algorithms is beyond the scope of this text, but whenever we refer to the DFT, we will imply the use of a real input transform.

Basically, the central point of the implementation of convolution with the DFT is the use of the *overlap-add* method after the inverse transform. Since our impulse response will be of a certain length, this will determine the transform size (we will capture the whole signal in one DFT). The other input signal can be of arbitrary length, all we will need to do is to keep taking time slices of it that are the size of the impulse response. Now, because we know that the resulting length of the convolution of two signals is the sum of their lengths minus one, this will determine the minimum size of the transform (because the IDFT output signal, as a result of the convolution, will have to be of that length).

The need for an *overlap-add* arises because, as the length of the convolution is larger than the original time-slice, we will need to make sure the tail part of it is mixed with the next output block. This will align the start of each output block with the original start of each time-slice in the input signal. So, if the impulse response size is *S*, we will slice the input signal in blocks of *S* samples. The convolution output size will be *2S – 1,* and the size of the transform will be the first power-of-two not less than that value.

The inputs to the transform will be padded to the required size. After the multiplication and the IDFT operation, we will have a block of 2S – 1 samples containing the signal output (the zero padding will be discarded). All we need to do is to time-align it with the original signal, by overlapping the first S – 1 samples of this block with the last S –1 samples of the previous output. The overlapping samples then are mixed together to form the final output. Fig.4 shows the input signal block sizes and the overlap-add operation.

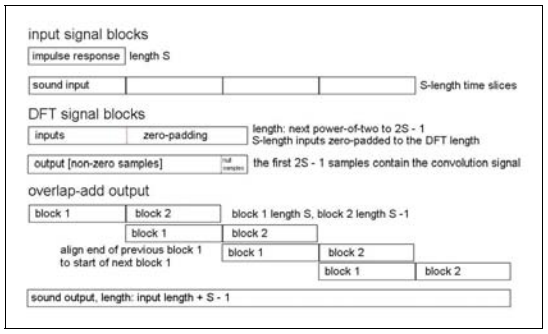

input signal blocks
impulse response
length S
sound input
S-length time slices
DFT signal blocks
inputs
zero-padding
length: next power-of-two to 2S - 1
S-length inputs zero-padded to the DFT length
output [non-zero samples]
the first 2S - 1 samples contain the convolution signal
overlap-add output
block 1
block 2
block 1 length S, block 2 length S -1
align end of previous block 1
to start of next block 1
sound output, length: input length + S - 1

Figure 4. Convolution input and output block sizes and the overlap-add operation.

In terms of realtime applications, it is important to remark that there is an implicit delay in the DFT-based convolution. This is determined by the length of the impulse response. So with longer responses, a possible realtime use is somewhat compromised.

3. The Short-Time Fourier Transform

So far we have been using what we described as a single-frame DFT, one 'snapshot' of the spectrum at a specific time point. In order to track spectral changes, we will have to find a method of taking a sequence of transforms, at different points in time. In a way, this is similar to the process we used in the convolution application: we will be looking at extracting blocks of samples from a time-domain signal and transform them with the DFT. This is known as the Short-Time Fourier Transform. The process of extracting the samples from a portion of the waveform is called *windowing*. In other words, we are applying a time window to the signal, outside which all samples are ignored.

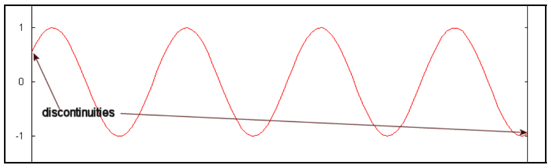


Figure 5. Rectangular window and discontinuities in the signal.

Time windows can have all sorts of shapes. The one we used in the convolution example is equivalent to a *rectangular* window, where all window contents are multiplied by one. This shape is not very useful in STFT analysis, because it can create discontinuities at the edges of the window (fig.5). This is the case when the analysed signal contains components that are not integer multiples of the fundamental frequency of analysis. These discontinuities are responsible for analysis artifacts, such as the ones observed in the above discussion of the DFT, which limit the its usefulness.

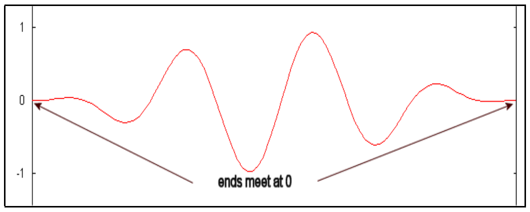


Figure 6. Windowed signal, where the ends tend towards 0.

Other shapes that tend towards 0 at the edges will be preferred. As the ends of the analysed segment meet, they eliminate any discontinuity (fig.6).

The effect of a window shape can be explained by remembering that, as seen before in (7), when we multiply two time-domain functions, the resulting spectrum will be the convolution of the two spectra. This is of course, the converse case of the convolution of two time-domain functions as seen in the section above. The effect of convolution in the amplitude spectrum can be better understood graphically. It is the shifting and scaling of the samples of one function by every sample of the other. When we use a window function in the DFT, we are multiplying the series of complex sinusoids that compose it by that function. Since this process results in spectral convolution, the resulting amplitude spectrum of the DFT after windowing will be imposition of the spectral shape of a window function on every frequency point of the DFT of the original signal (fig.7). A similar effect will also be introduced in the phase spectrum.

When we choose a window that has an amplitude spectrum that have a peak at 0 Hz and dies away quickly to zero as the frequency rises, we will have a reasonable analysis result. This is case of the ideal shape in fig.7, where each analysis point will capture components around it and ignore spurious ones away form it.

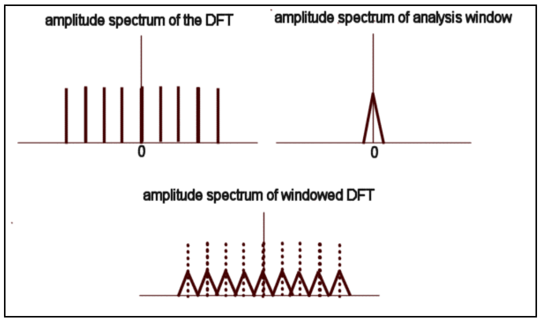

amplitude spectrum of the DFT
amplitude spectrum of analysis window
0
0
amplitude spectrum of windowed DFT

Figure 7. A simplified view of the amplitude spectrum of a windowed DFT as a convolution of the DFT sinusoids and the window function. The dotted lines mark the position of each DFT frequency point.

In practice, several windows with such low-pass characteristics exist. The simplest and more widely-used are the ones based on raised inverted cosine shapes, the Hamming and Hanning windows defined as:

$$w(n) = \alpha - (1-\alpha)\cos\left(2\pi \frac{n}{N-1}\right) \quad 0 \le n < N$$

where $\alpha = 0.5$ for Hanning and $\alpha = 0.54$ for Hamming

(8)

In order to perform the STFT, we will apply a time-dependent window to the signal and take the DFT of the result. This will mean also that we are making the whole operation a function of time, as well as frequency. Here is a simple definition for the discrete STFT of an arbitrary-length waveform *x(n)* at a time point *t*:

$$STFT(x(n),k,t) = \frac{1}{N}\sum_{n=-\infty}^{\infty} w(n-t)x(n)e^{-j2\pi kn/N}$$

$$k = 0,1,2,\ldots,N-1$$

(9)

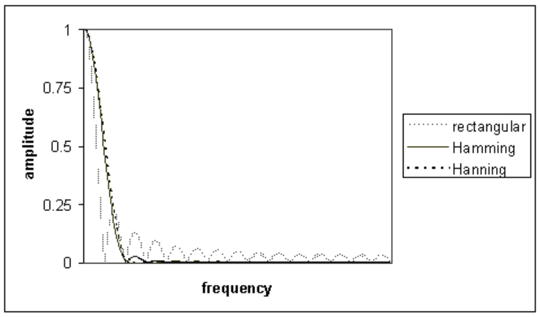


Figure 8. Spectral plots for rectangular, Hamming and Hanning windows (positive frequencies only).

The STFT provides a full spectral frame for each time point.. In general, it is possible to take as little as four overlapped transforms (hopsize = *N/4*) to have a good resolution. Effectively, when using the STFT output for signal processing, the smallest hopsize will be determined by the type of window used (*N/4* is the actual value for Hamming and Hanning windows).

The overlapped spectral frames can be transformed back into the time-domain by performing an inverse DFT on each signal frame. In order to smooth any possible discontinuities, we will also apply a window to each transformed signal block. The waveform is reconstituted by applying an overlap-add method that is very similar to the one employed in the convolution example.

4. Spectral Transformations: Manipulating STFT data

Each spectral frame can be considered as a collection of complex pairs relative to the information found on equally-spaced frequency bands at a particular time. They will contain information on the amplitude and frequency contents detected at that band. The rectangular, or cartesian, format that is the output of the transform packs these two aspects of the spectrum in the real and imaginary parts of each complex coefficient. In order to separate them, all we need to do is to convert the output into a polar representation. The magnitudes will give us the amplitude of each bin and the phases will be indicative of the detected frequencies. A single STFT frame can give us the amplitudes for each band, but we will not be able to obtain proper frequency values for them. This would imply extracting the *instantaneous frequency*, which is not really possible with one STFT measurement. Instead, the phases will contain the frequency information in a different form.

4.1. Cross-synthesis of frequencies and amplitudes

The first basic transformation that can be achieved in this way is *cross-synthesis*. There are different ways of crossing aspects of two spectra. The spectral multiplication made in the convolution example is one. By splitting amplitude and frequency aspects of spectra, we can also make that type of operation separately on each aspect. Another typical cross-synthesis technique is to combine the amplitudes of one sound with the frequencies of another. This is a spectral version of the time-domain *channel vocoder*. Once we have the STFT spectra of two sounds, there could not be an easier process to implement:

1. Convert the rectangular spectral samples into magnitudes and phases.
2. Take the magnitudes of one input and the phases of the other and make a new spectral frame, on a frame-by-frame basis.
3. Convert the magnitudes and phases to rectangular format. This is done with the following relationships:

$$\begin{aligned} re[z] &= Mag_z \times \cos(Pha_z) \text{ and} \\ im[z] &= Mag_z \times \sin(Pha_z) \end{aligned} \qquad (10)$$

The resulting spectral frames in rectangular format can then be transformed back to the time-domain using the ISTFT and the overlap-add method.

4.2. Spectral-domain filtering

If we manipulate the magnitudes separately, we might also be able to create some filtering effects by applying a certain contour to the spectrum. For instance, to generate a simple low-pass filter we can use the ¼ of the shape of the cosine wave and apply it to all points from 0 to N/2. The function used for shaping the magnitude will look like this:

$$mag[k] = \cos\left(\frac{\pi k}{2N}\right) \qquad 0 \le k \le N/2 \qquad (11)$$

A high-pass filter could be designed by using a sine function instead of cosine in the example above. In fact, we can define any filter in spectral terms and use it by multiplying its spectrum with the STFT of any input sound. This leads us back into the convolution territory. Consider the typical 2-pole resonator filter design, whose transfer function is:

$$H(z) = \frac{A_0}{1 - 2R\cos\theta z^{-1} + R^2 z^{-2}} \qquad (12)$$

Here, θ is the pole angle and R its radius (or magnitude), parameters that are related to the filter centre frequency and bandwidth, respectively. The scaling constant A_0 is used to scale the filter output so that it does not run wildly out-of-range. Now if we evaluate this function for evenly-spaced frequency points $z = e^{j2\pi k/N}$, we will reveal the discrete spectrum of that filter. All we need to do is to do a complex multiplication of the result with the STFT of an input sound.

The mathematical steps used to obtain the spectrum of the filter are based on Euler's relationship, which splits the complex sinusoidal $e^{j\omega}$ into its real and imaginary parts, cos(ω) and $j$sin(ω). Once we obtained the spectral points in the rectangular form $A_0(\mathrm{a} + \mathrm{ib})^{-1}$, all we need is to multiply them with the STFT points of the original signal. This will in reality turn out to be a complex division:

$$Y[k] =$$
$$= (re[X[k]] + im[X[k]]) \times A_0(re[F[k]] + im[F[k]])^{-1} =$$
$$= A_0\left(\frac{re[X[k]] + im[X[k]]}{re[F[k]] + im[F[k]]}\right)$$

where $X[k]$, $A_0F[k]^{-1}$ and $Y[k]$ are the spectra of the input signal, filter and output, respectively

(13)

There are many more processes that can be devised for transforming the output of the STFT. The examples given here are only the start. They represent some classic approaches, but several other, more radical techniques can be explored.

5. Tracking the Frequency: the Phase Vocoder.

As we observed, although we can manipulate the frequency content of spectra, through their phases, the STFT does not have enough resolution to tell us what frequencies are present in a sound. We will have to find a way of tracking the instantaneous frequencies in each spectral band. A well-known technique known as the Phase Vocoder (PV) (Flanagan and Golden, 1966) can be employed to do just that.

The STFT followed by a polar conversion can also be seen as a bank of parallel filters. Its output is composed of the values for the magnitudes and phases at every time-point or hop period for each bin. The first step in transforming the STFT into a Phase Vocoder is to generate values that are proportional to the frequencies present in a sound. This is done ideally by the taking the time derivative of the phase, but we can approximate it by computing the difference between the phase value of consecutive frames, for each spectral band. This simple operation, although not yielding the *right* value for the frequency at a spectral band, will output one that is proportional to it.

By keeping track of the phase differences, we can time-stretch or compress a sound, without altering its frequency content (in other words, its pitch). We can perform this by repeating or skipping spectral blocks, to stretch or compress the data. Because we are keeping the phase differences between the frames, when we accumulate them before resynthesis, we will reconstruct the signal back with the correct original phase values. We are keeping the same hop period between frames, but because we use the phase difference to calculate the next phase value, the phases will be kept intact, regardless of the frame readout speed.

One small programming point needs to be made in relation to the phase calculation. The inverse tangent function outputs the phase in the range of –π to π. When the phase differences are calculated, they might exceed this range. In this case, we have to bring them down to the expected interval (known as *principal values*). This process is sometimes called phase unwrapping.

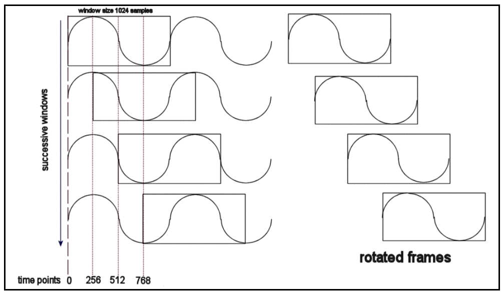


Figure 9. Signal frame rotation, according to input time point

5.1. *Frequency estimation*

So far we have been working with values that are proportional to the frequencies at each analysis band. In order to obtain the proper values in Hz, we will have to first modify the input to the STFT slightly. We will rotate the windowed samples inside the analysis frame, relative to the time point n (in samples and taken modulus N) of the input window. If our window has 1024 samples and we are hopping it every 256 samples, the moduli of the successive time-points n will be 0, 256, 512, 768, 0, 256.... The rotation will imply that for time point 256, we will move samples from positions 0 – 767 into positions 256 to 1023. The last 256 samples will be moved to the first locations of the block. A similar process is applied to the other time points (fig. 9).

The mathematical reasons for this input rotation are somewhat complex, but the graphic representation shown on fig. 9 goes some way on helping us understand the process intuitively. As we can see the rotation process has the effect of aligning the phase of the signal in successive frames. This will help us obtain the right frequency values, but we will better understand it after seeing the rest of the process. In fact, the input rotation renders the STFT formulation mathematically correct, as we have been using a non-rigorous and simpler approach (which so far has worked for us).

After the rotation, we can take the DFT of the frame as usual and convert the result into polar form. The phase differences for each band are then calculated. This now tells us how much each detected frequency deviates from the centre frequency of its analysis band. The centre frequencies are basically the DFT analysis frequencies $2\pi k/N$, in radians. So, to obtain the proper detected frequency value, we only need add the phase differences to the centre frequency for each analysis band, scaled by the hopsize. The values in Hz can be obtained by multiplying the result, which is given in *radians per hopsize samples,* by SR/[$2\pi x$ *hopsize*] (SR is, of course, the sampling rate in samples/sec).

Here is a summary of the steps involved in phase vocoder analysis:

1. Extract N samples from a signal and apply an analysis window. Rotate the samples in the signal frame according to input time n mod N.
2. Take the DFT of the signal.
3. Convert rectangular coefficients to polar format.
4. Compute the phase difference and bring the value to the $-\pi$ to $+\pi$ range.
5. Add the difference values to $2\pi kD/N$, and multiply the result by $SR/2\pi D$, where D is the hopsize in samples. For each spectral band, this result yields its frequency in Hz, and the magnitude value, its peak amplitude.

5.2. *Phase vocoder resynthesis*

Phase Vocoder data can be resynthesised using a variety of methods. Since we have blocks of amplitude and frequency data, we can use some sort of additive synthesis to playback the spectral frames. However, a more efficient way of converting to time-domain data for arbitrary sounds with many components is to use an overlap-add method similar to the one in the ISTFT. All we need to do is retrace the steps taken in the forward transformation:

1. Convert the frequencies back to phase differences in radians per I samples by subtracting them from the centre frequencies of each channel, in Hz, kSR/N, and multiplying the result by $2\pi I/SR$, where I is the synthesis hopsize.
2. Accumulate them to compute the current phase values.
3. Perform a polar to rectangular conversion.
4. Take the IDFT of the signal frame.
5. Unrotate the samples and apply a window to the resulting sample block.
6. Overlap-add consecutive frames.

As a word of caution, it is important to point out that all DFT-based algorithms will have some limits in terms of partial tracking. The analysis will be able to resolve a maximum of one sinusoidal component per frequency band. If two or more partials fall within one band, the phase vocoder will fail to output the right values for the amplitudes and frequencies of each of them. Instead, we will have an amplitude-modulated composite output, in many ways similar to beat frequencies. In addition, because the DFT splits the spectrum in equal-sized bands, this problem will mostly affect lower frequencies, where bands are perceptually larger. However, we can say that in general, the phase vocoder is a powerful tool for transformation of arbitrary signals.

5.3. *Spectral morphing*

A typical transformation of PV data is spectral interpolation, or morphing. It is a more general version of the spectral cross-synthesis example discussed before. Here, we interpolate between the frequencies and amplitudes of two spectra, on a frame-by-frame basis.

Spectral morphing can produce very interesting results. However, its effectiveness depends very much on the spectral qualities of the two input sounds. When the spectral data does not overlap much, interpolating will sound more or less like cross-fading, which can be achieved in the time-domain for much less trouble. There are many more transformations that can be devised for modifying PV data. In fact, any number manipulation procedure that generates a spectral frame in the right format can be seen as a valid spectral process. Whether it will produce a musically useful output is another question. Understanding how the spectral data is generated in analysis is the first step in designing transformations that work.

For a more detailed look into the theory of the Phase Vocoder, please refer to the James Flanagan's original article on the technique. Other descriptions of the technique are also found in (Dolson, 1986) and (Moore, 1990).

6. The Instantaneous Frequency Distribution

An alternative method of frequency estimation is given by the instantaneous frequency distribution (IFD) algorithm proposed by Toshihiko Abe (Abe et al, 1997). It uses some of the principles already seen in the phase vocoder, but its mathematical formulation is more complex. The basic idea, which is also present in the PV algorithm, is that the frequency, or more precisely, the instantaneous frequency detected at a certain band is the time derivative of the phase. Using Euler's relationship, we can define the output of the STFT in polar form. This is shown below, using $\omega = 2\pi k/N$:

$$STFT(x(n), k, t) = R(\omega, t) \times e^{j\theta(\omega,t)} \tag{14}$$

The phase detected by band k at time-point t is $\theta(2\pi kn/N,\ t)$ and the magnitude is $R(2\pi kn/N,\ t)$. The Instantaneous Frequency Distribution of $x(n)$ at time t is then the time derivative of the STFT phase output:

$$IFD\,(x(n), k, t) = \frac{\partial}{\partial t}\theta(\omega, t) \tag{15}$$

This can be intuitively understood as the measurement of the *rate of rotation* of the phase of a sinusoidal signal. In the phase vocoder, we estimated it by crudely taking the difference between phase values in successive frames. The IFD actually calculates the time derivative of the phase directly, from data corresponding to a single time-point:

$$IFD(x(n), k, t) = \omega + imag\left\{\frac{DFT(x'_t\,(m), k)}{DFT(x_t\,(m), k)}\right\} \tag{16}$$

The mathematical steps involved in the derivation of the IFD are quite involved. However, if we want to implement the IFD, all we need is to employ its definition in terms of DFTs, as given above. We can use the straight DFT of a windowed frame to obtain the amplitudes and use the IFD to estimate the frequencies. Also, because we are using the straight transform of a windowed signal, there is no need to rotate the input, as in the phase vocoder. If we look back at the DFT as defined in (19), we see that it, in fact, does not include the multiplication by a complex exponential (as does the STFT). Finally, the derivative of the analysis window is generated by computing the differences between its consecutive samples..

7. Tracking spectral components: Sinusoidal Modelling

Sinusoidal modelling techniques are based on the principles that we have held all along as the background to what we have done so far: that time-domain signals are composed of the sum of sinusoidal waves of different amplitudes, frequencies and phases. The number of sinusoidal components present in the spectrum will vary from sound to sound, and also can vary dynamically during the evolution of a single sound. As we have pointed out before, since we are still using STFT-based spectral analysis, at any point in time, the maximum resolution of components will depend on the size of each analysis band. Partial tracking will, therefore, not be suitable for spectrally dense sounds. However, there will be many musical signals that can be manipulated through this method.

7.1. Sinusoidal Analysis

The principle behind sinusoidal analysis is very simple, although its implementation is somewhat involved. Using the magnitudes from STFT analysis, we will identify the spectral peaks at the integral frequency points (STFT bins or bands). The identified peaks will have to be above a certain threshold, which will help separate the detected sinusoid components from transient spectral features. The exact peak position and amplitude can be estimated by using an interpolation procedure based on the magnitudes of the bins around the peaks. With the interpolated bin positions we can then find the exact values for the frequencies and phases obtained originally from the IFD/STFT input, again through interpolation. These will then, together with the amplitude, form a 'track', linked to each detected peak (fig. 11). However the track will only exist as such if there is some consistency in consecutive frames, ie. if there is some matching between peaks found at each time-point. When peaks are short-lived, they will not make a track.

Conversely, when a peak disappears, we will have to wait a few frames to declare the track as finished. So most of the process becomes one of track management, which accounts for the more involved aspects of the algorithm.

As seen in fig.10, the sinusoidal analysis will output tracks made up of frequencies, amplitudes and phases. This in turn can be used for additive re-synthesis of the signal or of portions of its spectrum. One typical method involves the use of the phase and frequency parameters to calculate the varying phase used to drive an oscillator. This uses the two parameters in a cubic interpolation, which is not only mathematically involved, but also computationally intensive. A simpler version can be formulated that would employ only the frequency and amplitudes interpolated linearly. This is simpler, more efficient and for many applications sufficiently precise. Also, since we do not require the phase parameter, we can simplify the analysis algorithm to calculate only frequencies and amplitudes for each track. This version could employ either the IFD (as shown in fig.12) or the Phase Vocoder, as discussed in the previous sections, to provide the magnitude and frequency inputs.

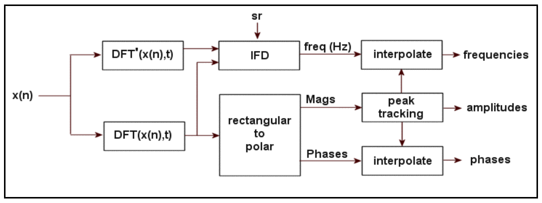


Figure 10. Sinusoidal analysis and track generation from a time-domain signal *x(n)*.

7.2. Additive resynthesis

The additive resynthesis procedure will take the track frames and use a frame-by-frame interpolation of the amplitudes and frequencies of each track to drive a bank of sinewave oscillators (fig.12). We will only need to be careful about using the track IDs to perform the interpolation between the frames. Also, when a track is created/destroyed, we will create an amplitude onset/decay so that we do not have discontinuities in the output signal. We will be using interpolating lookup oscillators, with a table size of 1024 points to generate the signal for each track. Each sine wave component will be then mixed into the output.

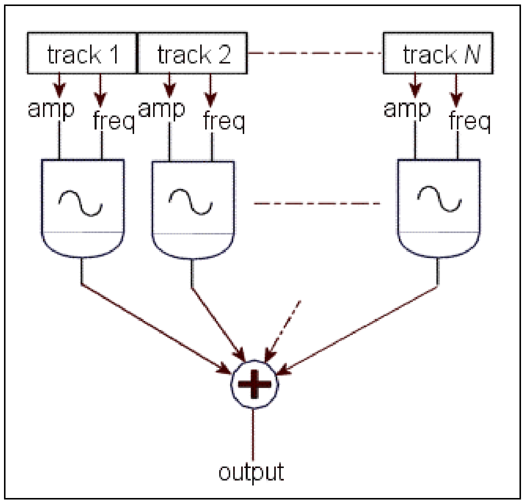


Figure 11. The additive synthesis process.

The main point about this type of analysis is that it is designed to track the sinusoidal components of a signal. Some sounds will be more suitable for this analysis than others. Distributed spectra will not be tracked very effectively, since its complexity will not suit the process. However, for certain sounds with both sinusoidal content and some more noise-like/transient elements, we could in theory obtain these 'residual' aspects of the sound by subtracting the resynthesised sound from the original. That way, we would be able to separate these two elements and perhaps process them individually. A more precise method of resynthesis, using the original phases is required for the residual extraction to work. This idea is developed in the Spectral Modelling Synthesis (SMS) technique (Serra, 1997).

8. Spectral processing with the Sound Object Library

The Sound Object (SndObj) library (Lazzarini, 2000) is a multi-platform music and audio processing C++ class library. Its 100-plus classes feature support for the most important time and frequency-domain processing algorithms, as well as basic soundfile, audio and MIDI IO services. The library is available for Linux, Irix, OS X and Windows; its core classes are fully portable to any system with ANSI C/C++ compilers.

The spectral processing suite of classes of the SndObj version 2.5.1 include the following:

(a) Analysis/Resynthesis:

FFT	STFT analysis
IFFT	ISTFT resynthesis
PVA	Phase Vocoder Analysis
PVS	Phase Vocoder Synthesis
IFGram	IFD + Amps (and phases) analysis

SinAnal	Sinusoidal track analysis
SinSyn	Sinusoidal synthesis (cubic interpolation)
AdSyn	Sinusoidal synthesis (linear interpolation)
Convol	FFT-based convolution

(b) Spectral Modifications

PVMorph	PV data interpolation
SpecMult	spectral product
SpecCart	polar-rectangular conversion
SpecSplit/SpecCombine	split/combine amps & phases
SpecInterp	spectral interpolation
SpecPolar	rectangular-polar conversion
SpecThresh	thresholding
SpecVoc	cross-synthesis

(c) Input/Output

PVRead	variable-rate PVOCEX file readout
SpecIn	spectral input
SndPVOCEX	PVOCEX file IO
SndSinI0	sinusoidal analysis file IO

In addition, the library provides a development framework for the addition of further processes. The processing capabilities can be fully extended by user-defined classes.

8.1. A programming example

The following example shows the use of the library for the development of a PD class for spectral morphing (Fig.12). Here we see how SndObj objects are set-up in the PD class constructor code:

```
void *morph_tilde_new(t_symbol *s, int
                      argc, t_atom *argv)
{
(...)
x->window = new HammingTable(1024, 0.5);
x->inobj1 = new SndObj(0, DEF_VECSIZE,
                       sr);
x->inobj2 = new SndObj(0, DEF_VECSIZE,
                       sr);
x->spec1  = new PVA(x->window, x->inobj1,
                    1.f, DEF_FFTSIZE,
                    DEF_VECSIZE, sr);
x->spec2  = new PVA(x->window, x->inobj2,
                    1.f, DEF_FFTSIZE,
                    DEF_VECSIZE, sr);
x->morph  = new PVMorph(morphfr, morpha,
                      x->spec1, x->spec2,
                       0,0,DEF_FFTSIZE,
                       sr);
x->synth  = new PVS(x->window, x->morph,
                    DEF_FFTSIZE,
                    DEF_VECSIZE, sr);
(...)
}
```

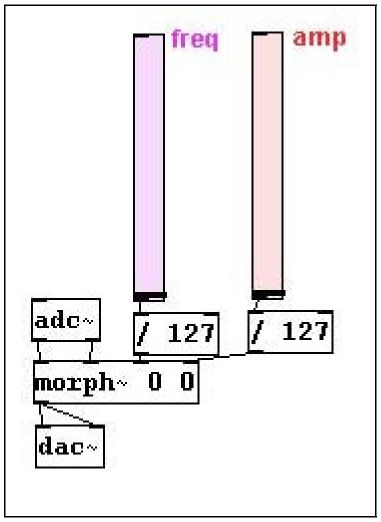


Figure 12. The morph~ object in a PD patch.

The class perform method will then contain the calls to SndObj::DoProcess() methods of each processing object. The methods SndObj::PushIn() and SndObj::PopOut() are used to send the signal into the SndObj chain and to get the processed output, respectively:

```
t_int  *morph_tilde_perform(int *w){
     t_sample *in1 = (t_sample*) w[1];
     t_sample *in2 = (t_sample*) w[2];
     t_sample *out = (t_sample*) w[3];
     t_int  size =   (t_int)     w[4];
     t_morph_tilde *x =
                 (t_morph_tilde*)w[5];

     int pos =
       x->inobj1->PushIn(in1, size);
     x->inobj2->PushIn(in2, size);
     x->synth->PopOut(out, size);

     if(pos == DEF_VECSIZE){
             x->spec1->DoProcess();
             x->spec2->DoProcess();
             x->morph->DoProcess();
             x->synth->DoProcess();
      }
      return (w+6);
 }
```

9. Conclusion

The techniques of spectral processing are very powerful. We have seen that they in fact have a multitude of applications, of which we saw the classic and most important ones. The standard DFT is generally a very practical spectral analysis tool, mostly because of its simplicity and elegance, as well as the existence of fast computation algorithms for it. There are adaptations and variations of it, which try to overcome some of its shortcomings, with important applications in signal analysis.

Nevertheless their use in sound processing and transformation is still somewhat limited. In addition to Fourier-based processes, which have so far been the most practical and useful ones to implement, there are other methods of spectral analysis. The most important of these is the Wavelet Transform (Meyer, 1991), which so far has had limited use in audio signal processing.

10. Bibliography

Abe T, et al (1997). "The IF spectrogram: a new spectral representation," *Proc. ASVA 97*: 423-430.
Dolson, M (1986). "The Phase Vocoder Tutorial". Computer Music Journal, 10(4): 14-27. MIT Press, Cambridge, Mass.
Flanagan, JL, Golden RM (1966). "Phase Vocoder". Bell System Technical Journal 45: 1493-1509.
Jaffe, D (1987a). "Spectrum Analysis Tutorial, Part 1: The Discrete Fourier Transform". Computer Music Journal, 11(2): 9-24. MIT Press, Cambridge, Mass.
Jaffe, D (1987b). "Spectrum Analysis Tutorial, Part 2: Properties and Applications of the Discrete Fourier Transform". Computer Music Journal, 11(2): 17-35. MIT Press, Cambridge, Mass.
Lazzarini, V (2000). "The Sound Object Library". Organised Sound 5 (1). Cambridge University Press, Cambridge.
McCaulay, RJ, Quatieri, TF (1986). "Speech Analysis/Synthesis Based on a Sinusoidal Representation". IEEE Trans. On Acoustics, Speech, and Signal Processing, ASSP-34 (4).
Meyer, Y (ed.)(1991). *Wavelets and applications*. Springer-Verlag, Berlin,1991
Moore, FR (1990). *Elements of Computer Music*. Prentice-Hall, Englewood Cliffs, N.J.
Openheim, AV, Schafer, RW (1975). *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, N.J.
Serra, X. (1997). "Musical Sound Modelling with Sinusoids plus Noise". in: G.D. Poli et al (eds.), *Musical Signal Processing*, Swets & Zeitlinger Publishers, Amsterdam
Steiglitz, K (1995). *A Signal Processing Primer*. Addison-Wesley Publ., Menlo Park, Ca.

AlsaModularSynth - An instrument for the electrified virtuoso

Matthias Nagorni,
SUSE Linux AG, Nürnberg, Germany

30. April 2004

Abstract

AlsaModularSynth is designed as a modular synthesizer for realtime performance. When used together with the VCO and filter plugins developed by Fons Adriaensen, it features a faithful emulation of vintage analogue modular systems. This article is a summary of the talk presented at the 2. Linux Audio Conference. It gives an overview over AlsaModularSynth with emphasis on parameter interfaces and MIDI I/O.

1. Introduction

An instrument designed for expressive live performance should offer a limited number of controls that the musician can modify in a deterministic and intuitive way. Ideally these controls allow a wide variation of volume and timbre.

In electronic music, either analogue circuits or virtual oscillators and filters are used to create the desired sound spectra. Patches for all three „classical“ synthesis techniques, namely additive, subtractive and frequency modulation (FM) synthesis can be realized with AlsaModularSynth. As will be shown in this article, subtractive synthesis offers the limited parameter set required for live performance and is therefore still widely used for this purpose.

2. Expressive historic electronic instruments

Surprisingly, it was already in the early days of electronic music that its most expressive exponent was invented: Around 1918 Lev S. Termen invented the instrument that bears his name. The Theremin is the only instrument that is played without touching it.

Fig. 1: The author demostrating how to play a Theremin.

However the Theremin is even more difficult to play than a string instrument. Therefore let's go back to the world of keyboard instruments. Also in this area some of the earlier designs allow a much more expressive playing than current commercial keyboards. Probably the most advanced example is the Sackbut synthesizer built in 1948 by Hugh Le Caine [1].

Its keyboard did not not only feature velocity sensitivity but also pitch control. When a key was moved horizontally the respective pitch would change. While the player was assumed to play the notes with his right hand, the left hand was supposed to modify the timbre. For this there were pressure-sensitive controls for each finger (numbering as in piano scores):

1:	Main Formant, Auxiliary Formant
2:	Basic Waveform Control
3-5:	Controls which produce departure from periodicity

The webpage [1] lists also other instruments by Hugh Le Caine and has some useful links.

3. AlsaModularSynth and MIDI

Controls of such complexity are not available commercially. Anyway it's even more fun to design alternate controls oneself. The Linux operating system with its open source approach is ideally suited for building interfaces to alternate input devices. Frank Neumann has e.g. written a MIDI driver for a graphic tablet offering 5 independent parameters: Pos-X, Pos-Y, Pressure, Tilt-X, Tilt-Y. The modular synthesizer AlsaModularSynth is designed to offer a flexible interface to MIDI controllers. Any parameter of a patch can be bound to either MIDI controller or note events.

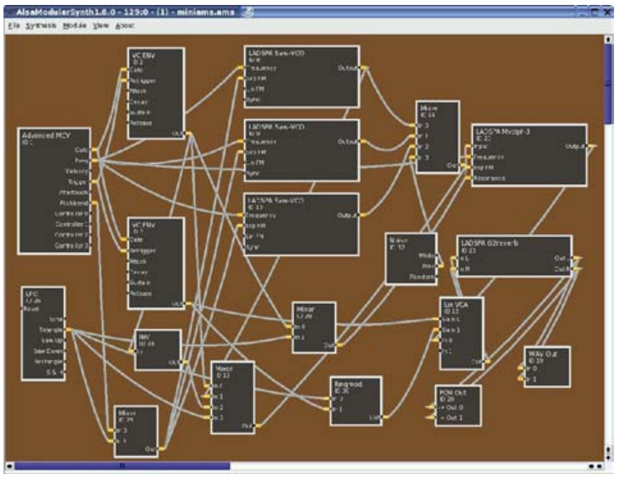
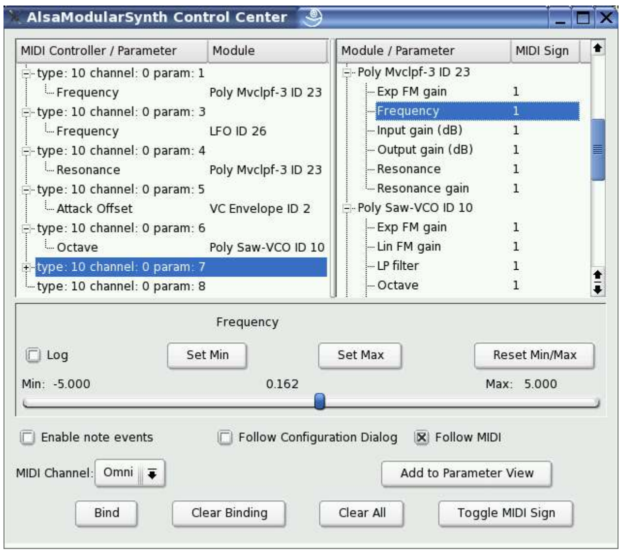


Fig. 2: The main window of AlsaModularSynth with a three-oscillator patch typical for subtractive synthesis.

All MIDI bindings are defined in the „AlsaModularSynth Control Center“. If the window is visible and receives MIDI Controller events, the respective controllers are listed in the left list. They can then be bound to the parameters listed below their respective module in the right list. In many cases only a limited range of a continuous parameter is used. Therefore the range to which the MIDI range of 0..127 is mapped can be freely chosen. For each continuous parameter a logarithmic mapping can be activated.

Note that AlsaModularSynth will follow a MIDI controller only after it has passed the current parameter value. This „snap-in“ feature is useful to avoid parameter jumps.
It is also possible to bind note events to parameters. If „Enable note events“ is activated, the „Control Center“ will also list note events. The velocity of the note event defines the parameter value. Due to the „snap-in“ feature of MIDI controlled parameters, it might be necessary to move the parameter to its leftmost position before it will follow the note events.
If „Follow MIDI“ is activated, AlsaModularSynth will automatically highlight the MIDI controller that currently sends input data. If there is a MIDI binding, the respective parameter will be highlighted as well and a GUI for this parameter will be created in the lower part of the „Control Center“.


AlsaModularSynth Control Center

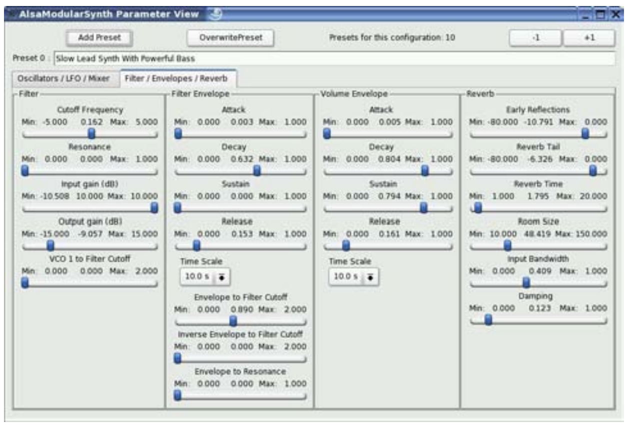
MIDI Controller / Parameter	Module
type: 10 channel: 0 param: 1	
Frequency	Poly Mvclpf-3 ID 23
type: 10 channel: 0 param: 3	
Frequency	LFO ID 26
type: 10 channel: 0 param: 4	
Resonance	Poly Mvclpf-3 ID 23
type: 10 channel: 0 param: 5	
Attack Offset	VC Envelope ID 2
type: 10 channel: 0 param: 6	
Octave	Poly Saw-VCO ID 10
type: 10 channel: 0 param: 7	
type: 10 channel: 0 param: 8	

Module / Parameter	MIDI Sign
Poly Mvclpf-3 ID 23	
Exp FM gain	1
Frequency	1
Input gain (dB)	1
Output gain (dB)	1
Resonance	1
Resonance gain	1
Poly Saw-VCO ID 10	
Exp FM gain	1
Lin FM gain	1
LP filter	1
Octave	1

Frequency
Log
Min: -5.000 0.162 Max: 5.000
Enable note events
Follow Configuration Dialog
[✓ Follow MIDI]
MIDI Channel: Omni

Fig. 3: The „Control Center“ of AlsaModularSynth lists the parameters of a patch and MIDI controllers.

In addition to the complete parameter list in the „Control Center“, each module has its own configuration dialog. It is opened by right-clicking on the module name. Usually only a subset of these parameters is needed to control the sound charateristics of a given patch. Parameters can therefore be arranged into a „Parameter View“ dialog.


AlsaModularSynth Parameter View

Presets for this configuration: 10
Preset 0 : Slow Lead Synth With Powerful Bass

Filter
Cutoff Frequency Min: -5.000 0.162 Max: 5.000
Resonance Min: 0.000 0.000 Max: 1.000
Input gain (dB) Min: -10.508 10.000 Max: 10.000
Output gain (dB) Min: -15.000 -9.057 Max: 15.000
VCO 1 to Filter Cutoff Min: 0.000 0.000 Max: 2.000

Filter Envelope
Attack Min: 0.000 0.003 Max: 1.000
Decay Min: 0.000 0.632 Max: 1.000
Sustain Min: 0.000 0.000 Max: 1.000
Release Min: 0.000 0.153 Max: 1.000
Time Scale 10.0 s
Envelope to Filter Cutoff Min: 0.000 0.890 Max: 2.000
Inverse Envelope to Filter Cutoff Min: 0.000 0.000 Max: 2.000
Envelope to Resonance Min: 0.000 0.000 Max: 1.000

Volume Envelope
Attack Min: 0.000 0.005 Max: 1.000
Decay Min: 0.000 0.804 Max: 1.000
Sustain Min: 0.000 0.794 Max: 1.000
Release Min: 0.000 0.161 Max: 1.000
Time Scale 10.0 s

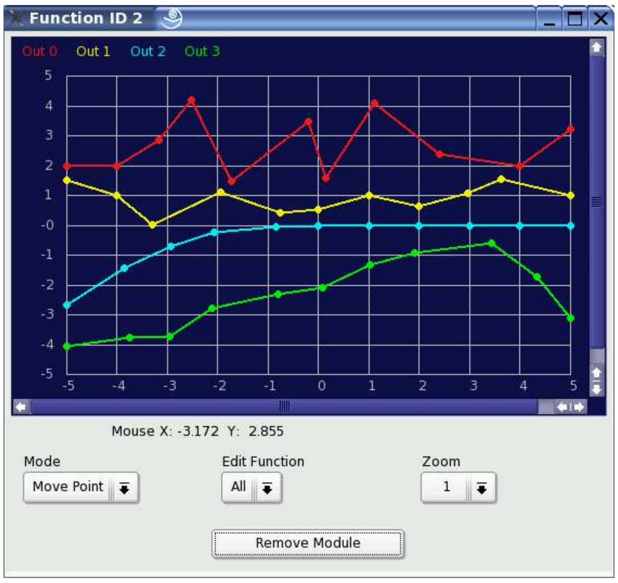
Reverb
Early Reflections Min: -80.000 -10.791 Max: 0.000
Reverb Tail Min: -80.000 -6.326 Max: 0.000
Reverb Time Min: 1.000 1.795 Max: 20.000
Room Size Min: 10.000 48.419 Max: 150.000
Input Bandwidth Min: 0.000 0.409 Max: 1.000
Damping Min: 0.000 0.123 Max: 1.000

Fig. 4: The „Parameter View“ dialog is the interface for live performance.

Moving a parameter into the „Parameter View“ dialog is done in the „Control Center“ by first selecting it and then pressing „Add to Parameter View“. Parameters can be grouped into tabs and named frames. It is possible to change the parameter name according to its function in a certain patch.
The parameter settings of the „Parameter View“ can be saved as named presets. These presets can be accessed via MIDI program change.

4. The function module and expressive velocity patches

To build an expressive instrument, it is interesting to let the MIDI velocity determine the timbre of a sound. A convenient way of mapping the velocity control voltage (CV) to up to 4 independent CV's is offered by the function module.



Mouse X: -3.172 Y: 2.855
Mode: Move Point
Edit Function: All
Zoom: 1

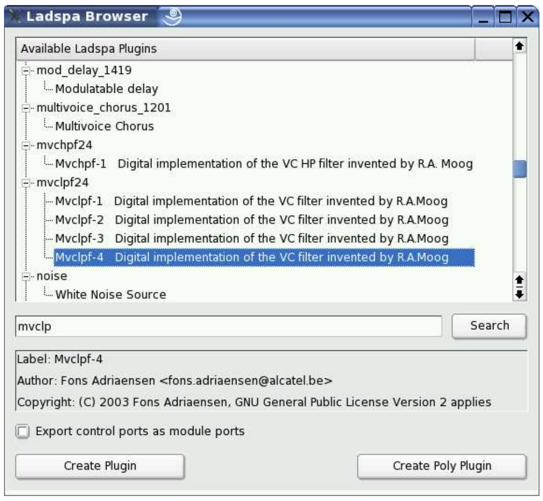
Fig: 5: Arbitrary mapping of a single input CV to 4 output CV's in the function module.

The typical subtractive synthesis patch has a filter envelope to control the cutoff frequency of a lowpass filter over time. A simple but effective way to add velocity dependend timbre control to such a patch is to let the velocity CV control the amount of the filter envelope signal to be sent to the filter cutoff.
In a setup with several VCO's it is also interesting to quantize the velocity CV with a „Quantizer 2“ module and let it control the octave of the upper VCO. Playing such a patch in a controlled way is a good excercise for pianists.

5. Additive vs. subractive synthesis

The most complex module of AlsaModularSynth is the „Dynamic Waves“ module. It features additive synthesis of up to 8 oscillators. Each oscillator is controlled by an eight-step envelope. Due to its large number of parameters, it is obvious that this kind of synthesis is not suited for realtime sound modifications during live playing.
Subtractive synthesis is much better suited for realtime sound screwing because the amount of parameters that

define a sound is much less than for additive synthesis. Fons Adriaensen has written high quality oscillator and filter plugins [3] that can be loaded into AlsaModularSynth via the „Ladspa Browser“.



mod_delay_1419
  Modulatable delay
multivoice_chorus_1201
  Multivoice Chorus
mvchpf24
  Mvchpf-1 Digital implementation of the VC HP filter invented by R.A. Moog
mvclpf24
  Mvclpf-1 Digital implementation of the VC filter invented by R.A.Moog
  Mvclpf-2 Digital implementation of the VC filter invented by R.A.Moog
  Mvclpf-3 Digital implementation of the VC filter invented by R.A.Moog
  Mvclpf-4 Digital implementation of the VC filter invented by R.A.Moog
noise
  White Noise Source

mvclp

Label: Mvclpf-4
Author: Fons Adriaensen <fons.adriaensen@alcatel.be>
Copyright: (C) 2003 Fons Adriaensen, GNU General Public License Version 2 applies

☐ Export control ports as module ports

Fig. 6: The „Ladspa Browser“ lists all available LADSPA plugins.

To enable polyphonic playing, they have to be created with „Create Poly Plugin“. The button „Create Plugin“ will create only one single instance of the plugin. All voices will be downmixed for such a plugin. This mode should be used for effects and reverb.

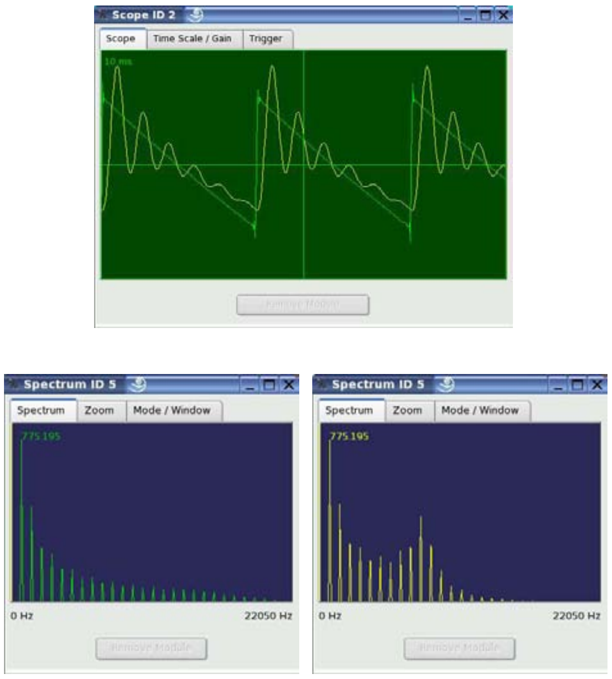


Fig. 7: Scope module (top) with saw wave and filtered saw, Spectrum module with saw spectrum (bottom, left) and filtered saw (bottom, right). The peak caused by the resonance is clearly visible.

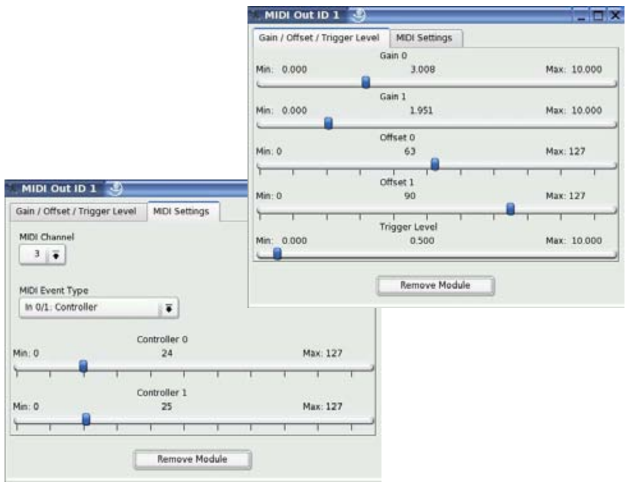
AlsaModularSynth has „Scope“ and „Spectrum“ modules that can be used to show how subtractive synthesis works: The rich spectrum of a e.g. a sawtooth wave is shaped by a filter. The most famous of such filters is the lowpass invented by R.A. Moog around 1964. Its characteristics are mainly defined by only two parameters: cutoff frequency and resonance.

In a typical subtractive synthesis patch we find therefore only a small number of parameters that need to be modified to obtain large variations of the sound. Among these are the attack and decay of the volume and filter envelopes, the amount of the latter that is sent to the filter, the VCO octave settings, filter cutoff and resonance. All of these can be either controlled by the note velocity and aftertouch values or via MIDI controllers.

6. The MIDI Out module

AlsaModularSynth can not only receive MIDI data but also send it. The „MIDI Out“ module can convert the internal control voltages of AlsaModularSynth into MIDI controller, pitchbend or note events. This way you can e.g. control your MIDI expander with a LFO of AlsaModularSynth.

If the „Trigger“ port is connected, MIDI events are send whenever the trigger threshold is exceeded. Otherwise they are sent when a new MIDI value is reached.



Gain 0 — Min: 0.000 — 3.008 — Max: 10.000
Gain 1 — Min: 0.000 — 1.951 — Max: 10.000
Offset 0 — Min: 0 — 63 — Max: 127
Offset 1 — Min: 0 — 90 — Max: 127
Trigger Level — Min: 0.000 — 0.500 — Max: 10.000

MIDI Channel: 3
MIDI Event Type: In 0/1: Controller
Controller 0 — Min: 0 — 24 — Max: 127
Controller 1 — Min: 0 — 25 — Max: 127

Fig. 8: The „MIDI Out“ module offers a flexible conversion of virtual CV's into MIDI controller, pitchbend and note events.

Acknowledgements

Fons Adriaensen has provided many ideas (VC Double Decay envelope, Function module, speed optimization, etc.) and code to the AlsaModularSynth project. The high quality of his filter and oscillator plugins form the basis of the warm analogue sound of AlsaModularSynth.

The author is grateful to the SUSE Linux AG for support of the AlsaModularSynth project.

Links

[1] www.hughlecaine.com
[2] www.obsolete.com/120_years
[3] users.skynet.be/solaris/linuxaudio
[4] www.moogarchives.com
[5] www.till.com/articles/moog
[6] www.doepfer.de/a100e.htm