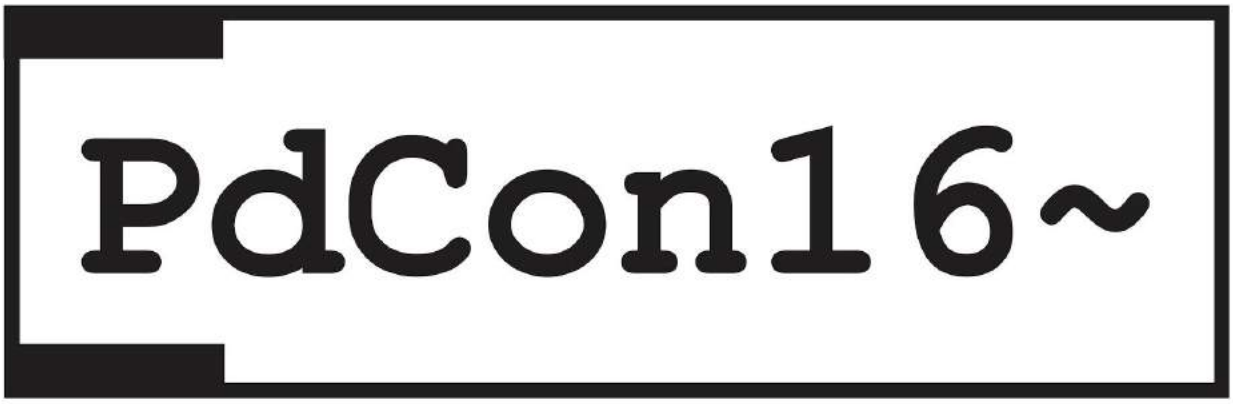


Proceedings

of the 5th International Pure Data Convention

New York

2016

Welcome

Dear PdCon16~ attendees,

Welcome the 5th international Pure Data Convention!

In the true spirit of Pd, this is an effort of the New York / New Jersey user base. Jaime Oliver, Richard Graham, and Sofy Yuditskaya collectively produced this convention with the help of their respective academic institutions and creative networks.

If we take Miller Puckette's 1996 publication* as the beginning of Pd, then 2016 marks its 20th anniversary. To celebrate this birthday with Miller and the Pd Community, New York University in collaboration with the Stevens Institute of Technology, will host the 5th International Pure Data Convention in New York City on November 16-20th, 2016.

The convention is made possible by the generous support of our host institutions, the Washington Square Contemporary Music Society, New Blankets, Waverly Project, and Harvestworks.

This is an exciting time for the Pd Community as we attempt to redefine the way in which we contribute to and extend Pd.

Thanks for coming over, we're happy to have you here, and hope you enjoy the Convention.

The PdCon16~ Team

* Puckette, Miller. "Pure Data: another integrated computer music environment." Proceedings of the Second Intercollege Computer Music Concerts (1996): 3741.

PdCon16~ Team and Supporters

PdCon16~ Conference Committee
Jaime Oliver La Rosa - Conference Chair
Richard Graham - Conference Chair
Sofy Yuditskaya - Performance & Installation Chair

Conference Team
Federico Camara Halac, Danielle McPhatter, Daniel Aleman, Friedrich Kern, Joel Rust.

PdCon16~ Advisory Committee:
Miller Puckette, Joe Deken, Alexandre Porres, Max Neupert, Katja Vetter, Alexandre Castonguay, IOhannes M Zmölnig.

Graphic and Web Design, and Video Documentation.
Federico Cámara Halac

Technical Team:
Adam Mirza, Yi-Wen Lai-Tremewan, Daniel Aleman, Federico Cámara Halac.

Volunteers:
Stevens Audio Engineering Club, Kuni Noubissi, Tara Annunziata, Tatiana Turin, Zijun Wang, James Barickman, Nicholas Metcalfe, Viola Yip, Joe Snape, Colten Jackson.

Paper Reviewers:
Miller Puckette, Marco Donnarumma, Spencer Topel, Julian Villegas, William Brent, Kerry Hagan, Giuseppe Silvi, Matt Barber, IOhannes Zmölnig, Jenn Kirby, Max Neupert, Ap Vague, Richard Snow, Seth Cluett, Andrea Franceschini, Jaime Munarriz, Ezra Teboul, Daniel Iglesia, Alessio Degani, Marco Matteo Markidis, Dan Wilcox, Alexandre Torres Porres, Eric Lyon, Gilberto Agostinho, Berenger Recoules, João Pais, James Dooley, Matt Barber.

Sponsors:
New York University, Stevens Institute of Technology, Washington Square Contemporary Music Society, New Blankets, Waverly Labs for Computing and Music at NYU, Harvestworks, Waverly Project.

Additional Thanks:
Lou Karchin and the Washington Square Contemporary Music Society. Lawren Young, Freeman Williams, and Music Faculty at the Music Department College of Arts and Sciences.

Music and Visual Arts Faculty, Kelland Thomas, Debra Pagan, Rosemary Damato, Andrew Stein, Kaitlin Donohue, Robin Hammermann, and Lina Kirby at Stevens Institute of Technology.

Index

array-abs: Array Abstraction Library for Pd

Matthew Barber
Rochester, NY USA
brbrofsvl@gmail.com

Abstract

array-abs is a new vanilla library of array processing abstractions employing Pd's `array` *objects and modeled on the list-abs collection. Four design principles that lend coherence to the library are discussed. Three examples from the collection illustrate the objects' utility and the rationale for these four design principles. Future work is proposed, with special focus on audio abstractions.*

Keywords

Array, Table, Abstraction, Library, Vanilla

1 Introduction

The vanilla Pd abstraction library list-abs (maintained by Frank Barknecht) [1] is one of the most successful Pd libraries due to its usefulness, its elegance, and its pedagogic value in teaching good patch design. Pd has long needed similar vanilla solutions for array manipulation and processing, but until the introduction of the `array` objects in Pd 0.45, Pd Vanilla had been missing the basic tools required for such a project. I present array-abs in this paper, a functional and feature-rich library of abstractions employing the `array` objects and modeled on the successes of list-abs.

2 Design

Overview

In order to make this a coherent library, I have adhered to a set of four design principles. First, and most important, every abstraction in the collection complies with the basic syntax and semantics implied in the vanilla `array` objects, and does so in a transparent way. I have extended these somewhat, e.g. by allowing negative indices to point backward from the end of an array. With very few exceptions, list inputs are distributed across the inlets as with Pd builtins, and outlets output from right to left.

Second, performance is important. To increase performance, some abstractions come in two versions, one with more features (e.g. bounds checking) and one streamlined and more efficient. In the same vein, `expr`, which can be inefficient, is not used unless absolutely necessary.

Third, it is important to reuse abstractions within other library objects when possible, as list-abs does. The obverse is equally important: if an algorithm requires a substantial subroutine, that subroutine should be included as an abstraction in the library in order to maximize encapsulation. For instance, `array-rfft` requires the generation of a bit reversal sequence to implement the fft butterflies, so it was added as `array-bitrevseq`.

Fourth and finally, every effort has been made to make the abstractions very readable and even pleasurable to study, both for ease of understanding and in order to encourage best patching practice, which is important if they are to be used as examples for teaching. In the rare case that these rules conflict, they are upheld in the order listed here; transparent syntax always holds, and performance wins out over reuse/encapsulation. There is almost never an instance where encapsulation interferes with readability.

As an exception to these rules, I have occasionally opted for a slightly less efficient implementation in order to illustrate a useful patching technique. For instance, `array-bitrevseq` requires bitwise XOR, and while Pd does not have a bitwise XOR object, `expr` does include the operator. Instead I decided on the two's complement identity $a \hat{} b = (a|b) - (a\&b)$ using Pd builtins, which runs about 85%-90% as fast as the equivalent `expr`.

2.1 Syntax and Semantics

The `array` builtins have a consistent design across each of the objects, which provides a guide for building abstractions around them. Integration with `table` and garrays complicates this somewhat, but in general I have hewn to the `array` object interface, with a few exceptions. I have decided not to support data structure arrays because the `array` objects require special flags and `pointer` handling in order to use them, and so

designing abstractions to use them becomes prohibitively complicated. Once a robust array-abs library is solidified for named arrays, perhaps a parallel library for data structure arrays could be built.

I take the following features to be essential to the `array` objects' interface. The rightmost inlet always expects an array name symbol. Most of the abstractions in array-abs read from or write to a single array, or perform an operation in place. For those which require a source array and a destination array, the two names are fed to the last two inlets on the right, source first and then destination. Unless the first inlet expects lists as a datatype, the `array` objects adhere to Pd standards and distribute members of an incoming list among the object's inlets, and "remember" the most recent input. The array-abs abstractions preserve this behavior.

The `array` internals are designed to operate on array ranges rather than entire arrays. Ranges are defined by an onset into the array and the number of elements in the range. Onsets less than zero are set to zero, while ranges with onsets greater than $n - 1$ are empty. Negative values for the range size specify the range from the onset to the end of the table. I have extended Pd's ranges to allow negative indices (as with Python lists), as well as negative direction. An index of -1 points to the last element of an array, -2 to the penultimate element, and so forth. For these deluxe ranges, the inlet order is always onset, size, direction. More information and examples may be found in the documentation patch for `array-range`.

The first creation argument to each the `array` internals is the array name. If the object deals with a range, the next two creation arguments specify the onset and size of the range, respectively, with defaults 0 and -1 (that is, the range defined by the whole array). This presents a challenge for abstraction design, because if a creation argument is not set, its `$n` value inside the abstraction defaults to zero, and zero is a legitimate range size. In order to get the abstraction to work with variable numbers of creation arguments (thus allowing nonzero defaults), one can employ a symbol-comparison trick invented by IOhannes Zmölnig. [2] See **Figure 1**. The `inlet` receives a bang when the parent patch requests the third creation argument. In a `symbol` object, the argument `$n-` only replaces the dollarsign value if it exists; if it does not, it treats the dollarsign as a symbol literal. If the argument exists, the patch replies with the value; if not, it sends a bang indicating the argument is empty, in which case a default value can be used instead. As a result of these complexities, some of the abstractions in the collection have a special init function to parse creation arguments, load defaults, and set initial conditions.

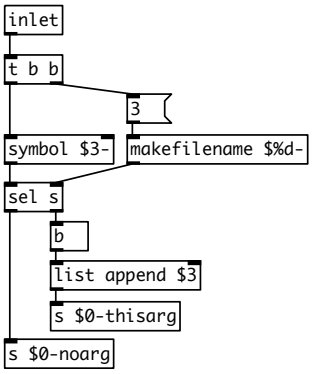


Figure 1. Detect empty argument.

2.2 Function

The file `array-abs-intro.pd` provides a list of all the abstractions organized into 13 categories, with a brief description of each one. The categories are:

1. **Pointwise Reading and Writing.** Includes interpolating array readers, an array writer, and swap functions.
2. **Ranges and Iterators.** Includes deluxe range objects with bounds checking, and index and element iterators over ranges.
3. **Copying and Moving Ranges.** Copy and paste elements of a range in bulk.
4. **Range Mutation.** Manipulate existing elements in ranges by changing their order. Includes abstractions to reverse, rotate, shuffle, partition, and sort.
5. **Searching Ranges.** Abstractions for finding values and comparing arrays.
6. **Set operations.** Set operations, such as intersection and union.
7. **Containers.** Container-like structures, such as FIFO and LIFO.
8. **Functional.** Algorithms common in functional programming.
9. **Statistics.** Statistical operations on ranges, such as various means.

10. **Range Filling.** Abstractions for filling ranges with values, including random values, additive synthesis functions, and others.

11. **Windowing.** Write or apply window functions to ranges.

12. **FFT Operations.** Real and complex FFT and IFFT operations.

13. **Signal Functions.** Miscellaneous signal abstractions, e.g. a partitioned convolution abstraction. This will eventually branch into subcategories.

In addition there are some helper abstractions. I have included as-yet unimplemented abstractions in `array-abs-intro.pd` to guide development and to encourage contributions from other developers.

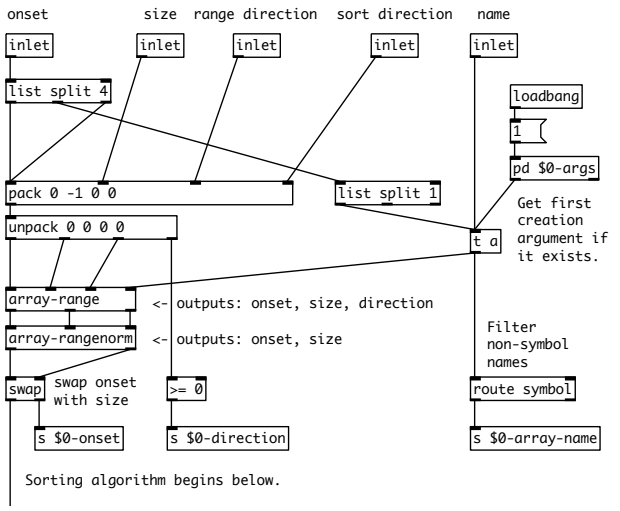


Figure 2. Distribute list input members.

2.3 Common Patching Idioms

The design principles outlined above constrain possibilities within the abstractions, so it has been useful to devise a few reusable patching idioms. The following are some examples of design problems and solutions. Because algorithms for array manipulation are well-documented, I am focusing on Pd-specific design problems here. Many tasks that seem inconsequential are necessary for maintaining consistent behavior.

2.3.1 Distributing List Input Members

There are several ways to properly distribute the members of a list input to the main inlet across the other inlets. **Figure 2** shows one idiom that appears often in array-abs (this is taken from one of the range sorting objects). The `pack 0 -1 0 0` stores values for the first four inlets listed at the top and sets defaults. In this case, the range "size" parameter is initially set to -1, which indicates the end of the table.[1]

At first blush it may seem wasteful to `pack` and `unpack` these values in short succession; since those values are going to be sent and stored elsewhere in the patch, why not just `unpack` straightaway? The answer is that for the inlets to work correctly, the values have to be stored and retrieved, sometimes in complicated ways, and default values can be hidden in the patch. Placing a `pack` near the top does the storage and retrieval in one step, and transparently "declares" parameter defaults. Some empirical tests have shown that it is also usually more efficient than the equivalent operation using several `value` objects, when all of the costs of storage, retrieval, and extra `trigger` ordering for the latter are factored in.

2.3.2 Array Names and Error Handling

The array name parameter of any `array` object requires special attention because there is no default and it is not usually set as often as the others; we also want objects with no array name to ignore input and those with a name of a nonexistent array to post errors upon input. **Figure 2** shows how the array abstractions get names from creation arguments and input.

On the right side, `pd $0-args` gets the first creation argument (if it exists), as in **Figure 1**. `list split 4` and `list split 1` route a fifth member of an incoming list to the same place as the argument subpatch and the rightmost `inlet`. On the lower right `route symbol` filters any non-symbol names that come through, and then the name is sent to where it is needed in the rest of the abstraction below. But why send the unfiltered name input to the `array-range` (and why filter non-symbols out in the first place)? This is because `array-range` has only one instance of an `array` builtin, and a non-symbol name will cause that instance to post an error; we filter it in **Figure 2** in order not to post more than one such error message. `array-range` already ignores input when no name is given, so we do not have to worry about the rest of the sorting algorithm executing with no array name. Finally, `array-range` posts an error on input when the array named does not exist.

This kind of error handling has to be managed explicitly in some of the abstractions. **Figure 3** il-

[1] `array-range` bounds-checks the input range, and `array-rangenorm` turns backwards ranges into normal ranges.

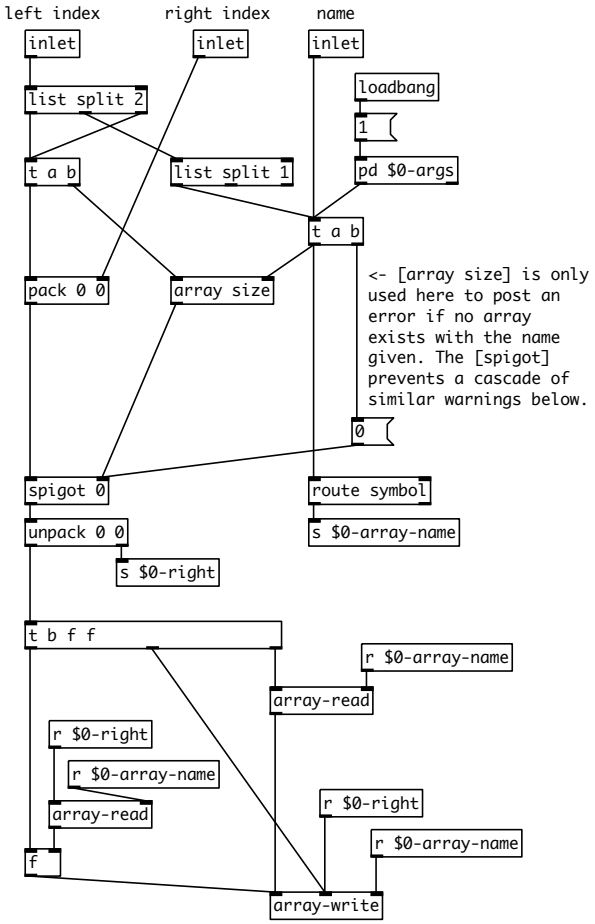


Figure 3. Explicit error handling in `array-swap`.

lustrates a typical technique from the `array-swap` abstraction, which swaps values at given indices in an array. A swapping algorithm requires two reads and two writes, and in Pd it is slightly more efficient to instantiate both reads as objects than to toggle the index values. The technically superfluous `array size` controls the `spigot`, in order not to let each `array` object report an error when something is wrong. This plus the negative index support incurs some overhead. For applications that require a fast swap (for instance, the various sorting algorithms), array-abs has a `tabswap` abstraction that has a `tabread`-like interface.

3 Examples

This sections contains a brief exploration of three members of array-abs. The first, `array-read4h`, is a table reader with cubic Hermite interpolation; it is one of many interpolating table readers in array-abs. The second, `array-bisort`, uses the binary-search abstraction `array-search` to implement an insertion sort algorithm. The third, `array-rfft` performs a real FFT on an array range and writes the real and imaginary components to a pair of arrays; it is as yet the most complex member of array-abs.

3.1 `array-read4h`

`array-read4h` is one of many interpolating table readers; it employs a cubic interpolator that is different from Pd's.

3.1.1 Interpolation Background

The cubic interpolation in Pd's `tabread4` and other classes is a piecewise "Lagrange" interpolator. To interpolate between two points, one imagines the four points surrounding the fractional index to be points on the unique curve defined by a cubic polynomial at $x = -1$, $x = 0$, $x = 1$, and $x = 2$ (where $0 \leq \text{index} \leq 1$). The output value is the y value of the curve at $x = \text{index}$.

Using this interpolator to oversample array elements results in a continuous piecewise curve with less RMS error than the curve resulting from interpolating linearly over the same data (as measured against an ideal sinc interpolator). The first derivative is discontinuous at breakpoints, however, leading to sharp corners in the waveform. This is easily apparent in the lefthand column of **Figure 4**, which shows the results of interpolating over a single impulse in the top graph and a cosine wave of period 4 in the bottom.[2]

It is possible to remove the first-derivative discontinuities at the expense of slightly greater RMS error by creating a cubic "Hermite" interpolator.

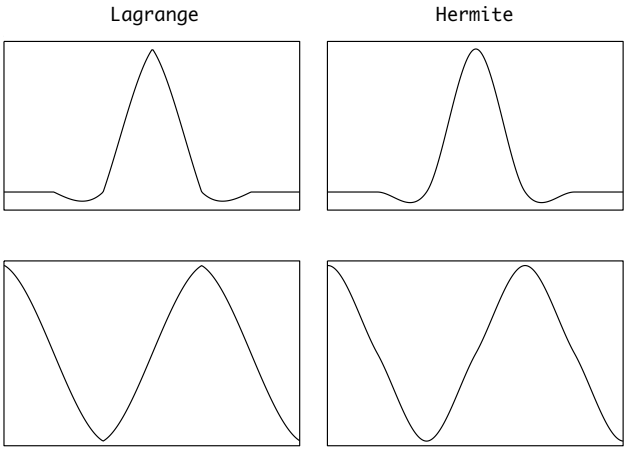


Figure 4. Top Row: Impulse Responses. Bottom Row: Cosine of Period 4.

[2]The data sets are $[(0), 0, 0, 0, 1, 0, 0, 0, (0)]$ and $[(0), 1, 0, -1, 0, 1, 0, -1, (0)]$, respectively, where values in parentheses are control points.

Instead of finding the cubic polynomial curve passing through all four points about the index, an Hermite interpolator finds the cubic polynomial curve that passes through the two points on either side of the index (i.e. $x = 0$ and $x = 1$) whose first derivatives at those points are approximated by the central difference. At $x = 0$, the central difference formula is $f'(0) \approx \frac{f(1)-f(-1)}{2}$, which is simply the slope of the line passing through the points $(1, f(1))$ and $(-1, f(-1))$.[3] This guarantees that the first derivative is continuous at breakpoints, as illustrated by the smooth curves in the righthand column of **Figure 4**.

While the second derivative is discontinuous at breakpoints for both interpolators, this discontinuity is removable for the Lagrange while it is not for the Hermite; note the sudden changes in concavity at zero crossings in the Hermite data in the bottom row of **Figure 4**.

3.1.1 4-point Hermite Interpolator

All of the table readers can read negative indices, and can interpolate between the end and the beginning of the array. When `array-read4h` receives an index, it calculates the fractional index F and retrieves the four surrounding points $a = x[-1]$, $b = x[0]$, $c = x[1]$, and $d = x[2]$ which are used to find the four coefficients in the function:

$$f(x) = C_0 + C_1 x + C_2 x^2 + C_3 x^3$$

We also need the first-derivative function:

$$f'(x) = 0 + C_1 + 2C_2 x + 3C_3 x^2$$

For Hermite interpolation, we have the following system of equations:[4]

$$\begin{aligned} f(0) &= b &&= C_0 + 0 + 0 + 0 \\ f(1) &= c &&= C_0 + C_1 + C_2 + C3 \\ f'(0) &= 0.5(c-a) &&= 0 + C_1 + 0 + 0 \\ f'(1) &= 0.5(d-b) &&= 0 + C_1 + 2C_2 + 3C_3 \end{aligned}$$

Solving for the four coefficients:

$$C_3 = 0.5(d-a) - 1.5(c-b)$$
$$C_2 = 0.5(c+a) - b - C_3$$
$$C_1 = 0.5(c-a)$$
$$C_0 = b$$

Together the coefficients require 4 multiplies and 7 adds to calculate. The polynomial as written above requires 5 multiplies and 3 adds. This can be reduced to 3 multiplies and 3 adds, making 7 and 10 total, by expressing it this way:

$$f(x) = C_0 + x \cdot (C_1 + x \cdot (C_2 + x \cdot (C_3)))$$

Figure 5 is the `array-read4h` implementation.

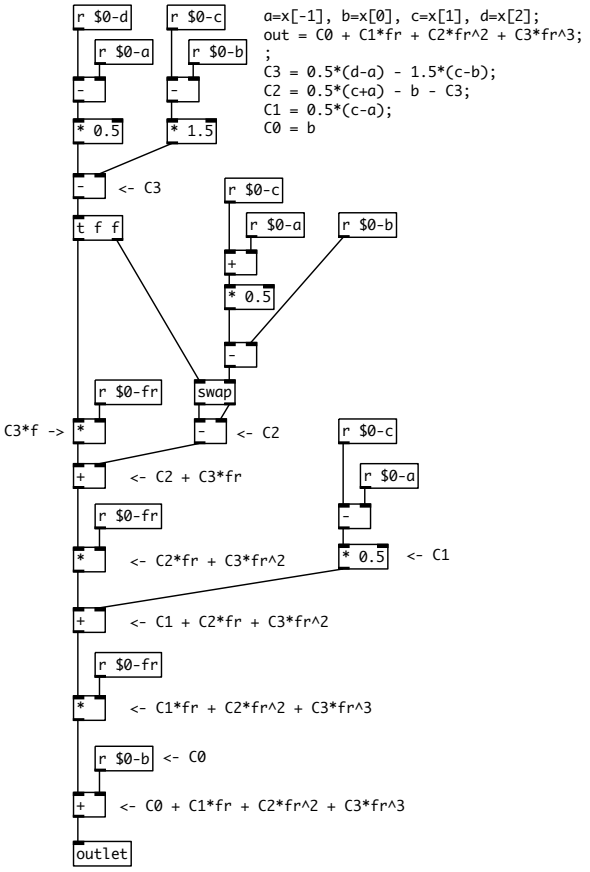


Figure 5. 4-point interpolation formula in `array-read4h`.

3.2 `array-bisort`

Sorting algorithms are interesting in Pd. Because Pd now supplies `array max` and `array min` internals, the fastest sorting algorithm for ranges of size less than about 1000 is a simple selection sort, which is among the slowest sorting algorithms all else being equal. For larger ranges, array-abs has several other sorting algorithms. One such is `array-bisort`, which is an insertion sort. An insertion sort works by searching through the elements that have already been sorted to locate the index of the next element, which is inserted at that position, shifting the sorted elements after that position one index higher.

The insertion sort is thus a combination of two sub-algorithms: a search algorithm and an insertion algorithm. Two search algorithms will work here: a linear search with low overhead which works in linear time, and a binary search which only works with sorted ranges and has higher overhead, but which runs in log time on average. `array-bisort` uses the binary search, imple-

[3] There are in fact other types of Hermite interpolation; this kind is known technically as a Catmull-Rom spline.

[4] These would usually be expressed in matrix form, but I have opted for equations to make it clearer to the uninitiated.

mented in array-abs as `array-bsearch`. For insertion, it uses the quick insert abstraction `tabinsert` rather than the more complex and higher-overhead object `array-insert`.

The `array-bisort` abstraction illustrates the advantages of reuse/encapsulation. In some ways it provides a better example of the potential uses of `array-bsearch` and `tabinsert` than a help file ever could, because the function of each is demonstrated in the context of a larger algorithm. Sort algorithms are generally excellent sources for encapsulated functions, e.g. swapping, merging, partitioning, and heap operations, to name a few.

3.3 `array-rfft`

The abstraction `array-rfft` performs a fast Fourier transform on real-valued input from a source array, and writes the real and imaginary results to a pair of destination arrays. It is the first of a planned suite of spectral abstractions, and is so far the most complex abstraction in array-abs. As such, it tests the limits of performance and readability. In some cases these two values are in direct opposition.

An exhaustive explanation of the workings of the FFT is beyond the scope of this paper, but here are some points of interest in `array-rfft`. It takes two creation arguments for the name of the source array and the base name of the destination arrays, which must be named <dest-name>-re and and <dest-name>-im. These names may be set by two rightmost inlets. The other four inlets are 1) source array onset; 2) number of elements to read from the source array; 3) FFT size – must be a power of two greater than or equal the value in the second inlet. If it is greater, the input is zero-padded. Zero (default) is the next power of two greater than or equal to the value in the second inlet; and 4) the onset into the destination arrays.

Figure 6 shows the order of execution after initial argument and input parsing. `pd $0-bounds-check` checks that the FFT size p is well-formed and that there is room in the destination arrays to write the $\frac{p}{2} + 1$ output points. `pd $0-one-or-two` manually writes output in case the FFT size is 1 or 2 points. `pd $0-resize-tables` resizes the internal arrays which hold the twiddle factors and the bit reversal sequence needed by larger FFTs, if the current sizes are too small.

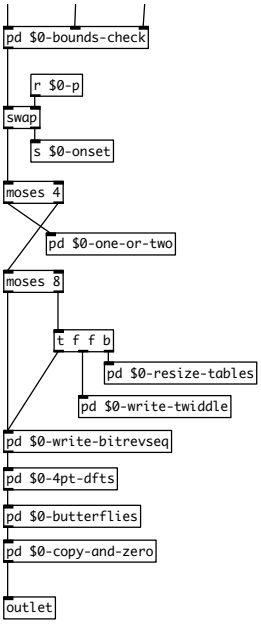


Figure 6. Order of Execution in `array-rfft`.

The subpatches `pd $0-write-twiddle` and `pd $0-write-bitrevseq` write the twiddle factors and bit reversal sequence; note that the twiddle factors are only needed for FFT of 8 or more points, as twiddle factors for a 4-point FFT are easily handled manually.[5] For all FFTs of 4 or more points, the first step is a series of 4-point DFTs. `pd $-4pt-dfts` writes the output of this 4-point DFT series directly to the destination arrays, and the rest can be done in place. `pd $0-butterflies` performs the FFT butterfly operations, and `pd $0-copy-and-zero` cleans up by moving the real Nyquist bin value from its temporary storage place in the zero-frequency imaginary bin to its proper place in the real destination array, and then zeroing out the imaginary zero-frequency and Nyquist bins.[6]

The real FFT uses symmetry to reduce the number of operations in comparison to the complex FFT, but the butterflies themselves are more complicated to patch correctly, and almost impossible to patch in a transparent, easy-to-read manner. See **Figure 7**; the `$0-A` and `$0-B` terms are the real and imaginary twiddle factors for the first butterfly, and due to symmetries in the cosine function they act as twiddle factors for the other butterflies with simple changes of sign. This is a case where performance demands a brute-force approach that is harder to read; the equivalent

[5]Since twiddle factors are read from a cosine table, there is no reason for a special `array-twiddle` abstraction.

[6]The temporary Nyquist storage is necessary for performing the FFT butterflies in place, and it saves several read/write operations on the arrays.

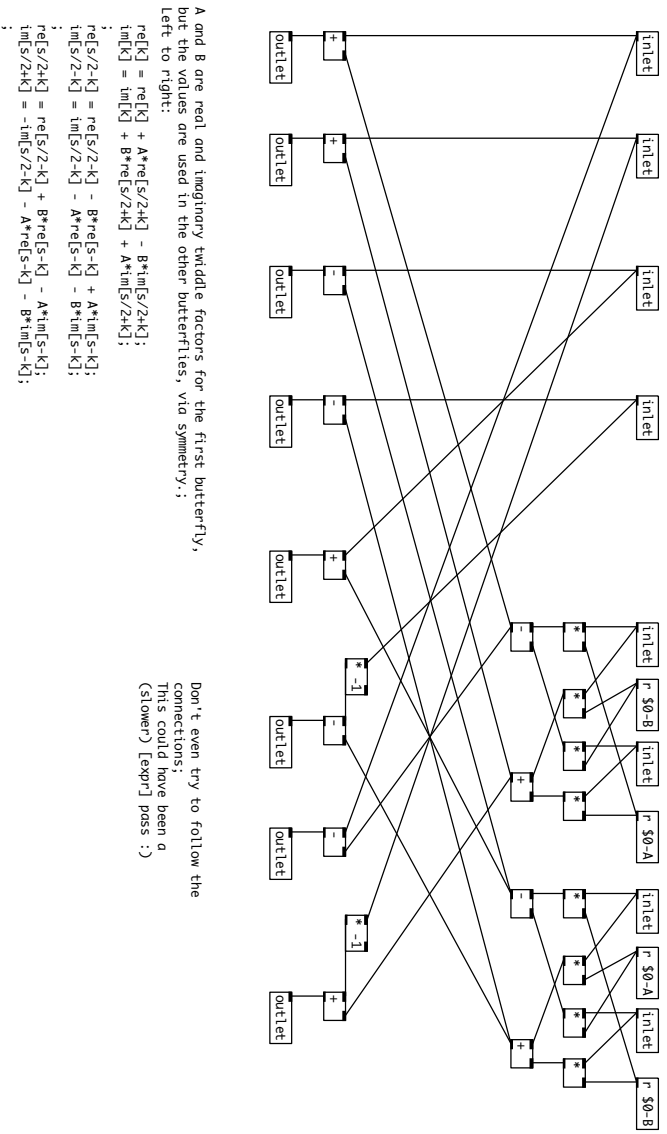


Figure 7. Butterfly Operations in `array-rfft`.

`expr` solution for FFT butterfly computation is far slower than the tangled web of patch connections, but it is far easier to read and comprehend. I have thoroughly commented each module of the algorithm to compensate for this opacity.

4 Future Work

While array-abs is already a useful contribution, it remains a work in progress. Aside from implementing the remaining abstractions in `array-abs-intro.pd`, I have outlined plans for future work below.

4.1 Documentation

Effective documentation is where most external libraries fail. So far I have only written basic documentation for the abstractions, and there is a lot of work yet to do. The first step will be to write a help patch conforming to pddp standards. For abstractions that form a family, such as the interpolating table readers, sort functions, and windowing abstractions, a special help abstraction should be included in each of the help files to compare and contrast the members of the family. Also, because of the commitment to reuse, the help file for each abstraction should contain a subpatch cross-referencing all abstractions it employs from the collection.

4.2 Signal Abstractions

So far I have worked on only one signal abstraction, `array-partconv~`, which is a partitioned real-time convolution implementation. There are obviously many opportunities for others. A suite of signal-domain interpolating array readers could be a starting point, but they may be too expensive for practical use, and the single-precision index would limit their utility. I welcome ideas, requests, and contributions from the Pd community.

4.3 Benchmarking

Vanilla implementations of these algorithms will likely never be as efficient as compiled externals. Once array-abs matures, I plan to compare the performance of these abstractions with similar objects, like William Brent's `tabletool` object, and Thomas Musil's iem_tab library.[7] Not only will this provide users with helpful information, but it will serve as a point of departure for improving the abstractions' efficiency.

5 Acknowledgements

Jonathan Wilkes offered a number of useful ideas. Thanks to Miller Puckette for the `array` objects.

References

[1] F. Barknecht, "list-abs: A collection of list-processing abstractions for Pd." `https://sourceforge.net/projects/pure-data/files/libraries/list-abs/`

[2] I. Zmölnig, "[PD] Testing for empty creation args?" `https://lists.puredata.info/pipermail/pd-list/2008-10/065465.html`

[7] `tabletool` is available here: `http://williambrent.conflations.com/pages/research.html#tabletool`
And iem_tab here: `https://git.iem.at/pd/iem_tab`

DRFX: A System for Multi-effect Processing in Pure Data

William Brent
American University
4400 Massachusetts Ave NW
Washington DC, USA, 20016
brent@american.edu

Abstract

This paper introduces [DRFX], an object for dynamic routing of audio between effect modules. [DRFX] allows routing connections to be defined on the fly with standard Pd messaging, or via a graphical routing matrix that is automatically generated from user-defined lists of audio inputs and effects. [DRFX] facilitates creative exploration of audio transformation by making inter-effect signal flow changes fast, easy to execute, visually explicit, and reproducible. This paper also describes two new audio effect objects for sinusoidal resynthesis and layered time stretching. All objects are vanilla Pd abstractions and have been made freely available.

Keywords

Routing, Mixing, Sound Design, Signal Processing, Dynamic Patching.

1 Introduction

Multi-effect processing is a thriving technique with broad creative applications. With even a modest set of audio effect modules, an abundance of distinct audio transformations can be produced. Pure Data (Pd) offers limitless real-time audio transformation options for use in sound design. User-defined signal processing (effect) modules can be patched together in parallel and series configurations to perform complex chains of processing that subtly or drastically alter audio signals for creative purposes. However, Pd lacks advanced native functionality for altering signal routing between effect modules with ease, speed, and precise control in live performance.

This paper introduces [DRFX]—a system for dynamic routing of audio between audio effect modules. It offers an approach to multi-effect processing in Pd that avoids multiple hard-coded processing chains with redundant effect modules, thereby helping to minimize processing load. Its automatically-generated graphical routing matrix encourages creative exploration of audio transformation by making inter-effect signal flow changes simple to execute, visually explicit, and immediately reproducible.

This paper also describes two new audio effect objects that can be used individually or within a [DRFX] chain. [martha~] is a companion to the [sigmund~] analysis object. It is specifically designed to accept output from [sigmund~]'s sinusoidal tracking function. In addition to housing an internal oscillator bank, [martha~] manages all the necessary voice bookkeeping required for resynthesis, and offers several sound transformation options tailored to sinusoidal decomposition/resynthesis. [streamStretch~] buffers multiple copies of incoming live audio and uses phase vocoding techniques to create overlapping streams of time-stretched and transposed output that trail the input. Individually, each of these objects can be tuned to produce an extensive range of unique results.

2 [DRFX]

[DRFX] is a vanilla Pd abstraction that uses dynamic patching to provide four primary features. First, automatic generation of a graphical routing matrix based on user-defined lists of audio inputs and effects. Second, the ability to route signal in any configuration between an arbitrary number of effect modules. Third, the ability to redefine routing configurations on the fly and immediately without audible distortion. Fourth, the ability to export the current routing configuration, output gain levels, and effect parameters as a text file in order to easily record and reproduce complex settings.

[DRFX] houses a dynamically patched signal routing system, a graphical routing matrix, and internal memory for storing preset data. Effect modules themselves are entirely separate from the [DRFX] abstraction.

2.1 Effect Module Conventions

When designing an effect module to be used with the [DRFX] system, a few specific conventions are required. Most importantly, effect modules must have a wireless [catch~] audio bus for input, a [send~] audio bus for

output to other effect modules, and a [throw~] audio bus for delivery to the main speaker output. For instance, an effect module named "echoDelay" would have wireless audio bus names as specified in Figure 1, where the input [catch~] name must be "echoDelay-in", the output [send~] name must be "echoDelay-out", and the [throw~] name for the tap output to the speakers can be freely chosen by the user (provided a corresponding [catch~] object exists somewhere in the patch). These simple audio bus naming conventions are required for [DRFX] to build an internal system of routing switches.

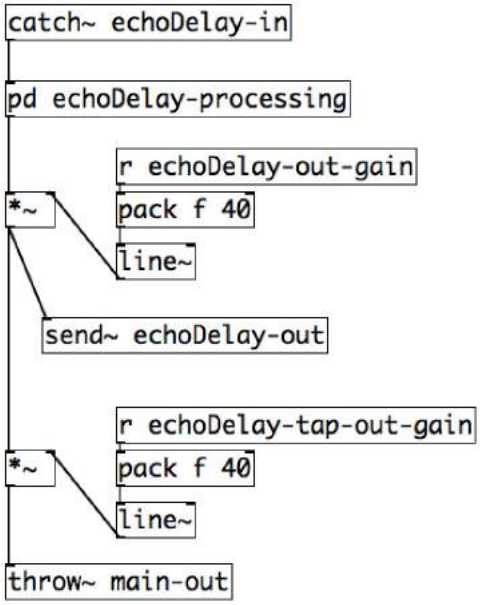


Figure 1: A basic effect module named "echoDelay".

Control data receive names for the output and tap output gains should end with "out-gain" and "tap-out-gain"[1] as shown in Figure 1. There is no restriction on send/receive names for effect parameters, and any number of parameters may be used for a given effect module. Although [DRFX] does not control mapping or routing of effect parameter data, it does store current parameter values for exporting preset files (described in Section 2.4).

Graphical controls are required for effect module output gain, tap output gain, and all effect parameters. These controls can be located at any level of a patch using [DRFX]; however, the send/receive names for these GUI controls must also follow specific conventions. For instance, a GUI control for the tap output gain of an effect module named "echoDelay" must have a send name of "echoDelay-tap-out-gain" and a receive name of "echoDelay-tap-out-gain-set". This is required for [DRFX]'s preset saving and loading feature.

Additional effect module features—such as output gain fade command logic, multichannel spatialization of the tap output, and switching off of DSP within unused modules—are neither required nor prohibited, and can be freely implemented as needed.

2.2 Primary [DRFX] Methods

[DRFX] takes no creation arguments, but must be supplied with 3 essential lists of information: audio bus names for all input signals, names for all effects, and names for all effect parameters. These are specified via the "input-list", "fx-list", and "fx-params" methods, respectively. Figure 2 shows the messages to [DRFX] required to create a routing system for one input signal transmitted via [send~ guitar], a second input signal transmitted via [send~ vocal], and 4 effect modules named "echoDelay", "pulser", "ringMod", and "softClip". Once these lists are supplied, internal routing objects, parameter memory, and a graphical routing matrix are generated by invoking the "create-matrix" method. Figure 3 illustrates the graphical routing matrix that would be generated in this scenario, which can be called up using the "matrix" method.

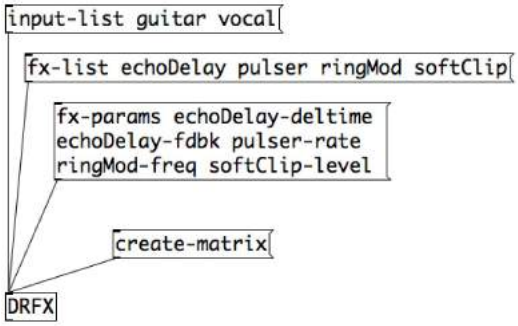


Figure 2: Methods for creating a [DRFX] routing system.

2.2 Graphical Routing Matrix

When using large numbers of effect modules, a graphical routing interface makes exploratory approaches to sound design intuitive and performative. [DRFX]'s routing matrix is comprised of GUI toggle objects with embedded send/receive names. A system with 2 inputs and 4 effects will produce 8 routing switches for inputs to effects, and 12 switches for inter-effect routing[2]. Horizontal and vertical labels are provided as comment objects, and

1 Separate gain stages for output and tap output are required for controlling the amount of signal sent to the speakers independently from the amount of signal sent from one effect module to another.

2 Currently, [DRFX] does not allow for the output of an effect module to be routed back to its own input. Therefore, a system with N effect modules will produce N^2-N inter-effect routing switches.

rows of switches alternate in color to facilitate navigation.

The send/receive naming convention for matrix toggles specifies the name of the source and target effects, followed by "-switch" for the send name and "-switch-set" for the receive name. Consistent send/receive naming for the toggles make the process of mapping to a physical control surface straightforward, and more importantly, enables routing preset messages like the following to be used in broadcast messages or cue list files:

```
echoDelay-ringMod-switch-set 1;
```

The message above would route the output of the "echoDelay" module to the input of the "ringMod" module, while sending a zero would disable the routing connection. Amplitude is ramped during the opening and closing of routing switches to avoid any distortion. The default ramp time is 40ms, but global and individual switch ramp times can be set to any duration using additional methods. Currently, all ramping is carried out using a quartic curve in order to approximate smooth changes in loudness, as described in [1].

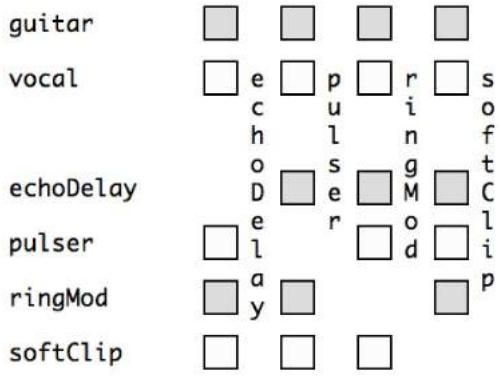


Figure 3: A [DRFX] routing matrix.

2.3 Routing Configurations

Apart from blocking individual effect modules from feeding back into themselves, there are no limitations on routing. Any possible combination of parallel and series connections can be defined via messaging commands to the graphical matrix, or by using the matrix GUI directly. As larger feedback loops may be desirable in some cases, [DRFX] does not prevent such connection schemes, and responsibility for managing feedback issues falls on the user.

2.4 Exporting Preset Files

When actively using [DRFX], settings for current routing connections, effect module output gains, tap output gains, and all parameters can be exported to a text file using the "export" method. The resulting text file contains the messages required to send preset data to all GUI controls associated with a [DRFX] session. Therefore, all send/receive names in preset files end with "-set", following the convention described in Section 2.1. When a preset file is loaded and executed using the [qlist] object, GUI control objects receive the data and relay it to their corresponding send names for use within [DRFX]'s internal routing system and external effect modules.

2.5 Other [DRFX] Methods

[DRFX] features many additional methods for handling tasks like clearing all routing, setting switch ramp time, muting all output or tap output gains, setting gains to unity, destroying an existing matrix, adjusting matrix label margins, etc. The accompanying help file provides detail on these functions and a basic multi-effect example.

3 [martha~]

[martha~] is a vanilla Pd abstraction that works in tandem with the [sigmund~] analysis object. It is designed to accept output from [sigmund~]'s sinusoidal tracking function. In addition to providing an internal oscillator bank and doing all the voice bookkeeping for routing data appropriately, it offers functionality for managing the attack and release times of partials, independent glissing and amplitude pulsing of individual partials, and forcing inharmonic spectra to harmonic arrangements. Combined, these features can create effects ranging from reverberation-like tails to melting disintegration of an input signal. Creation arguments are the number of tracks being followed by [sigmund~], and size of the internal oscillator bank.

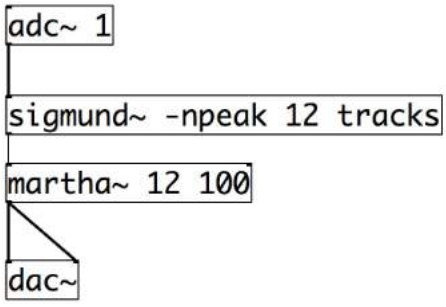


Figure 4: Basic signal flow for [martha~].

Although [sigmund~]'s "tracks" output specifies all the data needed for a basic resynthesis of analyzed audio, a great deal of further programming is required to gain control over several crucial synthesis details. For instance, control over the amplitude attack and release times of partials is especially important for achieving good resynthesis because analysis data often changes rapidly, causing clicks and

distortion when fed directly to oscillators. [martha~] works by storing [sigmund~]'s current and previous track flag data for every track being followed. At moments of discontinuity (e.g., when a previously existing track disappears), [martha~] ensures that the oscillator voice assigned to the dropped track fades out as desired before taking on any new track data. The same functionality is available for fading in of new tracks. With relatively short attack and release times, [martha~]'s output sounds like a good resynthesis of the input signal, while longer fade times result in an amorphous approximation that merely resembles the input. Fast attacks and long decays create an interesting reverberation effect.

Other control methods include a minimum amplitude threshold for spawning a sinusoidal track, pitch transposition, glissando of partials with individual trajectories (i.e., the glissando durations and depths can vary among partials), tremolo effects for individual partials, and forcing of partial frequencies to a specified set of pitches. The latter is especially useful in creating chord and pitch effects with broadband inharmonic signals like tam tams and gongs. Many other control methods exist, and are detailed in the accompanying help file.

4 [streamStretch~]

[streamStretch~] is a vanilla Pd abstraction that buffers multiple copies of incoming audio and creates overlapping streams of time-stretched and transposed output that trail the input. In terms of signal processing, it uses a phase vocoding technique described in [1], which allows for independent control over transposition and playback speed. [streamStretch~] manages two basic control processes: a record process and a playback stream process, each regularly triggering at its own independently specified rate.

Record streams triggered by the record process cannot write to a simple ring buffer, because the write index to the buffer could overtake a playback index in the case of an extremely slow playback stream, causing an audible discontinuity. To avoid this, a bank of audio buffers is used, and the record process assigns an unused buffer for each triggered record stream.

[streamStretch~]'s playback process triggers playback streams reading from the buffer which has been written to most recently. Because playback speed cannot exceed 100%, there is no risk of a playback index overtaking the write index of a given buffer. The rate of both record and playback stream triggering can be changed using simple messages to [streamStretch~] in order to control the content and density of output streams.

[streamStretch~] has many control methods detailed in its accompanying help file. These provide control over analysis window size, playback speed range (within which [streamStretch~] will randomly choose upon spawning playback streams), transposition range, amplitude range, and duration range. It is also possible to break up the steady triggering of playback streams (e.g., requesting that only 80% of playback triggers actually spawn playback streams). It is also possible to request a bundle of parallel streams that begin at once, enabling chords and harmonizer-like effects. In all, [streamStretch~]'s control methods can be used to create rich and varied output that ranges from simple chorusing to thick textural harmonized pastiche.

5 Conclusion

This paper introduced three new objects for audio effect processing in Pd. [DRFX] presents an efficient system for developing and executing multi-effect-based projects. It automatically generates all signal routing objects and graphical routing matrix objects with embedded send/receive names, provides control of the GUI matrix via Pd messaging, and offers methods for saving/loading routing and effect parameter settings. The two audio effect objects, [martha~] and [streamStretch~], both rely on spectrum analysis techniques, and put forward two distinct palettes for broad sound transformation results.

All three objects were specifically designed as vanilla Pd abstractions rather than externals with the intent of facilitating cross-platform maintenance and encouraging direct modification by users. [DRFX], [streamStretch~], and [martha~] are all freely available at http://www.williambrent.com.

References

[1] M. Puckette. *The Theory and Technique of Electronic Music*. World Scientific Press, Singapore, 2007.

Ableton Link integration for Pure Data

Peter Brinkmann
Google Inc
peter.brinkmann@gmail.com

Abstract

Ableton Link is a technology for synchronizing multiple musical apps on one or more devices on the same local network. We present an external, `[abl_link~]`, that makes the functionality of Link available in desktop Pd as well as libpd-based iOS apps. We discuss the considerations that went into mapping the C API of Link to an idiomatic representation in Pd, how to achieve accurate timing across large audio buffers on iOS, and how to generalize the external to other platforms.

Keywords

Ableton Link, Pd for iOS, externals, synchronization

1 Introduction

Ableton Link [1] is a general solution for synchronizing musical apps on one or more devices. It requires no configuration; Link-enabled apps on the same local network will automatically find and synchronize with one another. The emphasis is on jamming; apps can join or leave a session at any time, and each app maintains its own timeline, which Link will align with the timelines of other apps.

Link has been integrated into Ableton Live 9.7 as well as a growing number of other audio applications [2]. In this paper, we present our solution for integrating Link into Pd, for desktop as well as mobile platforms, concluding with a discussion of the open problem of accurate cross-platform latency compensation.

2 The Link API

Ableton has published two implementations of Link, the LinkKit SDK for iOS [3] and a cross-platform open-source library in C++11 [4]. Consequently, there are two versions of the `[abl_link~]` external, for libpd-based iOS apps and for desktop Pd [5]. The desktop version is also available via deken. We expect that the desktop version will work on Android, but this has not yet been tested at the time of writing.

The cross-platform API is similar to Version 2.0.0 of the iOS API. (There is also an earlier version of the iOS API that looks substantially different but remains compatible at the protocol level.) While the iOS implementation of `[abl_link~]` is separate from the desktop implementation, they are compatible at the patch level. In particular, patches developed with desktop Pd will work with libpd on iOS.

The external exposes the full functionality of Link, with one exception: The Link API includes a method for forcibly remapping timelines across all peers in a session. The documentation strongly advises against using this feature, and so we chose to omit it from the Pd external. If a compelling use case arises, we will revisit this decision.

3 Link Concepts

A Link session has a tempo that is shared by all participants. A new instance of Link takes a default tempo as a creation parameter, but this is overridden by the session tempo when joining an existing Link session. After joining, any participant can change the session tempo at any time.

For some applications, having a common tempo may be enough. In most cases, however, further synchronization will be required. To this end, Link introduces the notion of *beat time*. The core functionality of Link is to map wall time to beat time in such a way that integral beat times line up across all participants in a session.

On top of beat alignment, Link offers *phase synchronization* determined by a *quantum* (typically, the number number of beats in a measure or loop). Link will align the phase (i.e., beat time modulo quantum) of all apps in a session to the extent that their respective quanta are commensurable. For example, if all apps in a session choose the same quantum (e.g., four beats), then their downbeats will be synchronized.

4 Test Plan

Both versions of the Link API come with an extensive test plan [6], [7]. On iOS, a libpd-based app using `[abl_link~]` is virtually guaranteed to pass all tests. (It is theoretically possible to create a patch that fails tests TEMPO-2 or TEMPO-3 by changing the session tempo on load, but that seems like deliberately disruptive behavior that won't happen by mistake.)

On desktops, patches using the Link ex- ternal will pass all tests except possibly AUDIOENGINE-1, which requires the latency compensation to be accurate to within 3ms. While the current version easily achieves this on a recent MacBook Pro, results on other platforms may differ (see Latency Compensation for details).

5 The Pd External

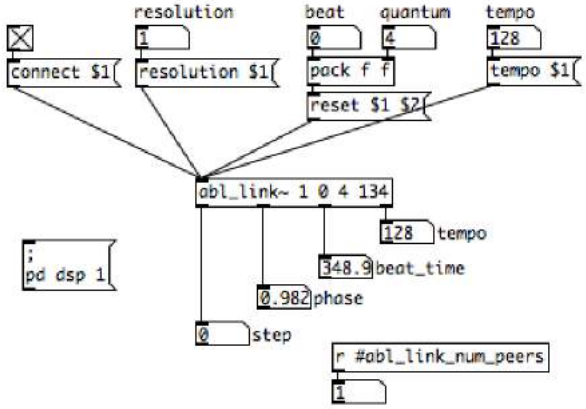


Figure 1: Methods and outputs of the external.

The Ableton Link external (Figure 1) has three outlets, corresponding to the fundamental Link concepts of tempo, beat time, and phase, plus an additional outlet that emits the index of the cur- rent step at the beginning of each step. The pur- pose of the step feature is to generate events in Pd at a given rate (measured in steps per beat). The tempo outlet fires on tempo changes; the beat and phase outlets fire after each DSP tick (i.e., every 64 frames). In particular, the external will be inert as long as DSP is disabled in Pd.

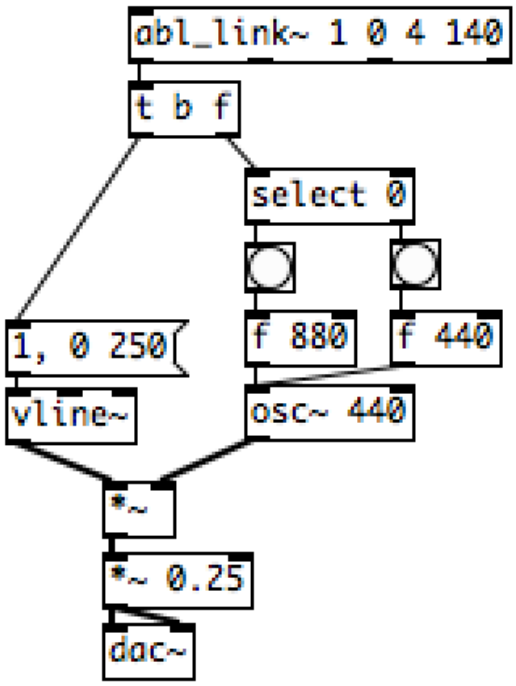


Figure 2: Metronome patch with Link.

We expect that many applications of `[abl_link~]` will just use the step outlet, much like Pd's built-in `[metro]` object (Figure 2).

Link's beat time may go backwards in some situations, e.g., when joining a session. When this happens, the step outlet will fall silent until the beat time moves forward again. The phase and beat outlets will output raw values from Link so that patches can implement their own handling of time going backwards.

The external responds to four methods, for connecting with other Link instances, for setting the resolution of the step outlet (in steps per beat) as well as the tempo (in beats per minute), and for resetting the beat time and quantum by having Link perform a quantized launch.

The creation arguments, all optional, specify the resolution (default 1) and the initial beat time (default 0), as well as the quantum (default 4) and tempo (default 120), which will be replaced by the session tempo if the Link instance is already connected. The Link external will perform a quantized launch on creation, i.e., the initial beat time may be smaller than requested.

The Link external emits events as soon as DSP is enabled in Pd. A newly created Link instance, however, will not automatically synchronize with other Link-enabled apps on the same network. In order to connect with other Link instances, you

need to send it a connect message. If there are multiple instances of `[abl_link~]`, they will share the same instance of Ableton Link internally. In particular, they will all be connected as soon as one of them is connected.

In addition to the values exposed by the outlets of the external, the desktop version of the API also reports the number of connected peers in a session. Since this value is not available from the iOS SDK and it has no immediate musical meaning, we chose to omit it from the outlets of the external. The desktop version will, however, send the number of peers to the symbol `#abl_link_num_peers` whenever a peer connects or disconnects.

6 Implementation

Audio apps interact with Link by obtaining a copy of their local timeline, which they can then query (e.g., to convert time stamps to beat time) and modify (e.g., by changing the tempo). Any changes remain confined to the local timeline until the timeline is returned to the Link instance to be reconciled with the overall Link session.

The Link API provides to pairs of methods for obtaining and committing a timeline, one to be called by the GUI and another one to be called by the DSP engine. It is important that each method be called by the appropriate thread only.

It would be dangerous to interact with Link timelines in response to Pd messages because messages may originate from a number of different threads, especially when running under libpd. For this reason, we decided to implement `[abl_link~]` as a *signal class*, which provides a method that Pd will invoke on each DSP tick. This method will always be called on the same thread, making it safe for Link's timelines.

By convention, time stamps passed to Link timelines are supposed to correspond to the time when the *end* of the current audio buffer reaches the speakers. This is a happy coincidence because the most straightforward implementation of `[abl_link~]`, mimicking Pd's built-in `[bang~]` object, does most of its work in a processing method that is invoked *after* the signal processing chain has run, i.e., there is an natural mapping from Link's beat time to Pd's sample time, with a granularity of 64 audio frames (about 1.5ms at a sample rate of 44.1kHz).

Another design challenge was the handling of multiple instances of `[abl_link~]`. Equipping each instances of the external with its own instance of Link would be hopelessly confusing because it would be possible for a patch to be out of sync with itself, if one instance is connected to a Link session and another one isn't.

The obvious solution is to have all instances of the external share an instance of Link, but it turned out that this isn't enough. An early implementation of a shared Link instance didn't work reliably, presumably because of the churn caused by multiple timeline commits within microseconds of one another. The solution was to introduce a wrapper object that manages the shared Link instances as well as a shared timeline. On each DSP tick, the first instance of `[abl_link~]` to be called creates a local timeline, and the last instance to be called commits the timeline.

7 Special Case: iOS

At the patch level, the desktop and iOS versions of `[abl_link~]` are virtually identical. When running under Pd for iOS [5], [8], [9], however, it is possible to improve the timing by integrating the external more deeply into the implementation of Pd for iOS.

Specifically, Apple's CoreAudio API provides accurate time stamps, including the time at which the current audio buffer will reach the speakers. This is a crucial piece of information that allows for accurate latency compensation, and our implementation makes it available to the external by modifying the audio unit used by Pd for iOS.

Moreover, CoreAudio sometimes increases its audio buffer size in order to conserve power. This is potentially problematic for timing sensitive applications like Link, but we were able to work around it by modifying the processing callback of Pd's audio unit so that it provides interpolated time stamps on each Pd tick.

Due to these two modifications at the CoreAudio level, the iOS version of `[abl_link~]` achieves excellent accuracy even in low-power situations, well below the 1ms tolerance that the Link test plan for iOS requires.

8 Latency Compensation

On other platforms, latency compensation is more difficult. First, at the level of abstraction at which Pd externals are implemented, there is no general way to query the current output latency. A relatively straightforward first step would be to add such a function to the internal API of Pd.

Second, Pd supports many different audio plat-

forms (PortAudio, ALSA, JACK, etc.), and so any API function for querying the output latency would have to be implemented many times over.

Finally, and most importantly, not all audio APIs report their output latency, and on many platforms the software sits on top of audio hardware that doesn't report its output latency. In this situation, the code samples that come with the Link API use the output buffer size as a proxy for the output latency.

In short, it may not be possible to consistently achieve accurate latency compensation on all platforms. At the time of writing the desktop version of `[abl_link~]` simply uses a hard-coded magic number that seems to work well on a recent MacBook Pro. Since the test plan of the desktop version only requires the timing to be accurate within 3ms, this value should also be good enough on other platforms with reasonably low output latency. We expect to refine this solution in response to feedback from the Pd community.

9 Acknowledgments

The design and implementation of `[abl_link~]` were shaped by many conversations with Alexander Randon, Christopher Rice, Dan Wilcox, Ed Kelly, Oliver Greschke, and others. Antoine Rousseau provided invaluable help and feedback. Special thanks to Matt Black for prompting the development of the desktop version.

References

[1] Ableton AG, Berlin, "Ableton Link," 2016. [Online]. Available: https://www.ableton.com/en/link/. [Accessed: 15-Oct-2016]

[2] Ableton AG, Berlin, "Ableton Link Applications," 2016. [Online]. Available: https://www.ableton.com/en/link/apps/. [Accessed: 15-Oct-2016]

[3] Ableton AG, Berlin, "LinkKit," 2016. [Online]. Available: http://ableton.github.io/linkkit/. [Accessed: 15-Oct-2016]

[4] Ableton AG, Berlin, "Ableton Link Codebase," 2016. [Online]. Available: https://github.com/Ableton/link. [Accessed: 15-Oct-2016]

[5] "Pd for iOS," 2016. [Online]. Available: https://github.com/libpd/pd-for-ios. [Accessed: 15-Oct-2016]

[6] Ableton AG, Berlin, "Test Plan," 2016. [Online]. Available: https://github.com/Ableton/link/blob/master/TEST-PLAN.md. [Accessed: 15-Oct-2016]

[7] Ableton AG, Berlin, "Test Plan," 2016. [Online]. Available: https://ableton.github.io/linkkit/#test-plan. [Accessed: 15-Oct-2016]

[8] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding Pure Data with libpd," in *Pure Data Convention Weimar 2011*, 2011.

[9] Peter Brinkmann, *Making Musical Apps: Real-time Audio Synthesis on Android and iOS*. O'Reilly Media, 2012.

Latest developments with Pd-L2Ork and its Development Branch Purr-Data

Ivica Ico Bukvic
Virginia Tech SOPA ICAT
DISIS L2Ork
Blacksburg, VA, USA 24061
ico@vt.edu

Jonathan Wilkes
jon.w.wilkes@gmail.com

Albert Gräf
Johannes
Gutenberg-University Mainz
IKM, Music-Informatics
Research Group
Mainz, Germany, 55122
aggraef@gmail.com

Abstract

In the following paper authors and co-maintainers will present Pd-L2Ork and Purr-Data forks of Pure-Data and Pd-extended, the motivation, and unique features. Introduced in 2010 Pd-L2Ork as a Linux-centric variant offers focus on improved usability, including SVG-enabled canvas, and a growing set of built-in objects and externals designed to lower the learning curve, facilitate rapid prototyping, and offer adaptable granularity. Purr-Data is a recent development branch of Pd-L2Ork focusing on a complete GUI rewrite and support for Windows and OSX. It leverages Node-Webkit and its unique affordances, enabling Pure-Data patches to utilize highly efficient canvas and CSS.

Keywords

Pd-L2Ork, Purr-Data, fork, usability, L2Ork

1 Introduction

Pure-Data, also known as Pd, [19] is arguably one of the most widespread audio and multimedia dataflow programming languages. Pd's history is deeply intertwined with that of its commercial counterpart, Cycling 74's Max [20]. A particular strength shared by the two platforms is in their modularized approach that empowers third party developers to extend software's functionality without having to deal with the underlying engine. Perhaps the most profound impact of Pd is in its completely free and open source model that has enabled it to thrive in a number of environments otherwise inaccessible to its commercial counterpart. Examples include custom in-house solutions for entertainment software (e.g. EaPd [16]), Unity3D [7] and smartphone integration via libPD [8], an embeddable library (e.g. RjDj [17], PdDroidParty [18], and Mobmuplat [15]), and other embedded platforms, such as Raspberry Pi [10].

Pd's author Miller Puckette has spearheaded a steady development pace with the primary motivation being iterative improvement while preserving the backwards compatibility and ability to run even oldest of patches in a rapidly growing library of community-generated creative code. Over the past two decades, Pd underwent several growth spurts. The earliest resulted in a rapid expansion of the program's functionality with a large number of supporting libraries, a number of which have delved in extending Pd's core operation. Its unprecedented popularity has challenged the evolving internal API with external library solutions leveraging calls and functionalities that were not in and of themselves finalized or deemed public. With Max's improved usability and a growing user base, the Pd community sought to complement Pure-Data's compelling core functionality with a level of polish that would lower the initial learning curve and improve user experience. In 2002 the community introduced the earliest builds of Pd-extended [4], the longest running Pd variant that was abandoned in 2013 due to lack of contributors to the project. Pd-extended is not the only Pd variant. There were other ambitious attempts, like pd-devel, Nova, and DesireData [5] that were born out of desire for a more nimble development, fundamental changes to the engine, and improved usability.

In 2009 the community led by Hans Christoph Steiner worked with Miller Puckette on refactoring the GUI code that had become overly convoluted and difficult to develop further. The rewrite was an opportunity for a convergence between Pd and Pd-extended. The effort was by and large successful and perhaps in part served as a catalyst for abandoning Pd-extended. Puckette's work on Pd continues to be instrumental in fostering creativ-

ity and curiosity across generations, and as the library of works relying on Pd grows, so does the importance of conservation and ensuring that Pd continues to support even the oldest of patches. This is where Puckette's ongoing stewardship and brilliance is indispensable. The inevitable side-effect of the increasingly conservationist focus of the core Pd is that any new addition has to be carefully thought out in order to account for all the idiosyncrasies of the past versions and ensure there is a minimal chance of a regression. This vastly limits the development pace. As a result, in recent years Pd has seen a resurgence in forks that aim to sidestep usability issues through alternative approaches, including embeddable solutions (e.g. libPd) and custom front ends.

2 Motivation

Introduced in 2009 by Bukvic, Pd-L2Ork [9] started as a Pd-extended variant that builds on version 0.42.5 with a growing number of patches submitted for upstream adoption. The focus was on nimble development designed to cater to the specific needs of the Linux Laptop Orchestra (L2Ork), even if that meant suboptimal initial implementations that would be ironed out over time as the understanding of the overall code base improved and the target purpose was better understood through practice. A part of L2Ork's mission was educational outreach. Consequently, a majority of early additions to Pd-extended focused on the usability improvements, including graphical user interface and editor functions. While some improvements were incorporated into the Pd proper, a growing number of rejected patches began to build an increasing divide between the two code bases. As a result in 2010 Bukvic introduced a separately maintained Pd-extended variant named Pd-L2Ork after L2Ork for which it was originally designed.

2.1 Philosophy

Pd-L2Ork's philosophy grew out of its initial goals and the early development efforts. It is defined by a nimble development with focus on both major and iterative code changes whose first and foremost goal is to streamline behavior and improve usability and stability as quickly as possible even if that means doing so at the expense of backwards compatibility. Despite an ostensibly lax outlook on backwards compatibility, to date Pd-L2Ork remains fully backwards compatible, and has over time included the "-legacy" flag for changes that may be too disruptive to existing users, such as the repositioning of all iemgui objects to reflect a consistent position in respect to their x and y locations on the patch canvas. Another important aspect of this philosophy is releasing improvements early and often even if the actual fix may not be optimal. Early on, such an approach offered opportunities for better understanding of Pd's internal code while having working iterations in the hands of dozens of students of varying educational backgrounds and experience ensured quick vetting of the ensuing solutions. In an attempt to minimize overhead in configuring the programming environment, including installing supplemental libraries, and in part address the potential for binary incompatibility with Pd, Pd-L2Ork provides a single turnkey monolithic solution with all the libraries included in one package.

Over time, as Pd-L2Ork grew in its scope and visibility, it attracted other long-term co-developers, co-maintainers, and community contributors. In 2014 Wilkes joined Bukvic as a co-developer. The same year Mathieu Bouchard briefly joined the team and assisted in code refactoring, with particular focus on streamlining the Tcl-C socket-based communication protocol. The team was further complemented in 2015 by co-maintainer and packager Gräf. The increasingly team-based development became yet another pillar of the Pd-L2Ork's philosophy and the team continues to openly invite others to contribute in whatever capacity may be the most appropriate.

3 Implementation

Pd-L2Ork's code base is increasingly divergent with that of Pd. It consists of over 1,700 bug-fixes, additions, improvements, and backports to the Pd-extended code base which can be split into engine, usability, documentation, new and improved objects and libraries, and scaffolded learning and rapid prototyping. The list below offers highlights to some of the most obvious additions to each of the said categories.

3.1 Engine

Internal engine contributions have largely focused on implementing features and bug-fixes requested by past and existing Pd users. Some of these include patches that have never made it to the core Pd, such as the cord inspector (a.k.a.

magic glass), improved data type handling logic, and support for outlier cases that may otherwise result in crashes and unexpected behavior (e.g. sending a signal which during execution changes the number of sends it is trying to reach before it has reached them all). Additional checks were implemented for the Jack Audio Connection Kit (JACK) [12] audio backend to avoid hangs in case JACK freezes (e.g. when an external soundcard is disconnected without closing JACK). Default sample rate settings are provided for situations where Pd-L2Ork may run headless (without the Graphical User Interface or GUI), so that objects that depend on sample rate can properly initialize, thus removing the need for potentially unwieldy headless startup procedures. Messages with the argument $0 now automatically resolve such value to the patch instance, while the $@ argument can be used to pass the entire argument set inside a subpatch or an abstraction. [trigger] logic has been expanded to allow for static allocation of values, which alleviates the need for creating bang triggers that are fed into a message with a static value and thereby considerably streamlines the patching process.

From a visual perspective, the Tk-based [22] graphical engine has been replaced with TkPath [6] which offers SVG-enabled antialiased canvas. While the ensuing GUI is less efficient than Tcl, in part due to uncertain state of the TkPath development, the internal engine changes offer accelerated displacement and redraw of objects. A lot of effort went into streamlining graph-on-parent (GOP), including proper bounding box calculation and detection, optimizing redraw, and resolving drawing issues with embedded (GOP) patches. Considerable effort went towards implementing consistent behavior. Improvements also focused on sidestepping the limitations of the socket-based communication between the GUI and the engine, such as keyboard autorepeat detection. As a result, the [key] object can be instantiated with an optional argument that enables autorepeat filtering, while leaving the default object behavior backwards compatible.

Another substantial core engine overhaul pertains to consistent ordering of objects in the glist (a.k.a. canvas) stack. Its implementation has helped ensure that objects always honor the visual stacking order on top of each other, even after undo and redo actions. More importantly, it has paved way towards more advanced functionality including advanced editing techniques (see usability below), and a system-wide preset engine. The preset engine consists of two new objects [preset_hub] and [preset_node]. Nodes can be connected to various objects, including arrays, and can broadcast current state to its designated hub for storing and retrieval. Multiple hubs can be used with varying contexts (defined using the optional symbol argument). The state saving references each object depending on its location in the multidimensional stack with predictable object positions. The ensuing system is universal, unaffected by editing actions, abstraction- and instance-agnostic (e.g. using multiple instances of the same abstraction is automatically supported), and efficient, allowing for anything from recording individual states to real-time automation of multiple parameters through periodic snapshots.

Path detection and retrieval for patches, externals, and abstractions has been expanded to improve Pd-L2Ork's ability to locate necessary files. The file path finding logic has been enhanced with universal prefixes, like the @pd_extra and @pd_help that autoresolve to extra and doc folders.

Data structures were also enhanced with the addition of sprites and new ways to manipulate said data. As a result, Pd-L2Ork ships with a game/tutorial designed by Wilkes where the user can navigate the patch with a virtual character who can interact with various objects and by doing so learn the basics of Pd programming language.

3.2 Usability

On the surface Pd-L2Ork builds on Pd-extended's appearance improvements. Under the hood, with the canvas being drawn as a collection of SVG shapes, the entire ecosystem lends itself to a number of new opportunities. The most obvious involve antialiased display, advanced shapes (e.g. Bézier curves that are also used for drawing patch cords), support for image formats with alpha channel (e.g. PNGs), and advanced data structure drawing and manipulation using SVG-centric enhancements.

A majority of usability improvements focus on the editor. The consistent stacking order implemented in the engine has served as a foundation for the infinite undo, as well as to-front and -back stacking options that are accessible via the right-click context menu. Iemgui objects' positions have been updated to ensure consistent be-

havior, namely accurate reflection of x and y values and consistent autopatching object placement. In addition, their size, display font, and label position were made editable through the GUI with the properties dialog values reflecting GUI-edited changes on-the-fly. Their redrawing was optimized and GOP visibility conditions updated to factor in labels. GOP window size and position were similarly retrofitted to support GUI-based adjustments and many of the supporting dialogs were streamlined and improved, including iemgui properties window and the color picker. Iemgui labels natively support spaces and comments provide graceful handling of line breaks that are also saved in backwards-compatible Pd files. Pd-L2Ork has integrated the old autotips patch and improved upon its design. The "tidy up" feature has been redesigned to offer a two-step realignment of objects. The first key press aligns the objects on a single axis, while the second respaces them, so that they are equidistant from each other. Intelligent patching was implemented to provide four variants of automatically generating multiple patch cords based on user's selection, and to provide additional ways of creating multiple connections (e.g. SHIFT + mouse click). The canvas scrolling logic has been overhauled to minimize the use of scrollbars, provide minimal visual footprint, and ensure most of the patch is always visible. Pd-L2Ork supports drag and drop, as well as preliminary support for pasting script code snippets directly onto the canvas. Its documentation browser has been enhanced to support xapian-enabled searching by keywords, as well as annotated navigation of folders and supporting libraries. To minimize confusion in running multiple concurrent instances, Pd-L2Ork introduces the "-unique" startup flag which is by default disabled and whose purpose is to explicitly specify that a new instance should be spawned. This is particularly useful when opening multiple patches from a file browser, thus preventing redundant spawning of multiple instances every time a new file is opened. The copy and paste engine has been overhauled to improve buffer sharing across multiple applications. The entire graphics engine has been made themeable and its settings are by default saved with the rest of the configuration files.

A part of the usability improvements also included the K12 mode (a.k.a. module) [11] accessible via the "-k12" startup flag. This education-centric mode designed specifically for beginners and children offers its own improvements to the user interface, including a sidebar with object buttons that is split into two tabs or categories (control and sound), and a simplified menu and reduced set of options. The ensuing patches can be transported seamlessly between the K12 and default (advanced) modes allowing for Pd-L2Ork to "grow" in its complexity in sync with user's proficiency. We will discuss the K12 mode in greater detail in the Scaffolded Learning section below.

3.3 Documentation

Pd-L2Ork continues to build on the Pd-extended documentation efforts. This includes over 200+ new and updated help files, including the cyclone library documentation. In addition, the K12 library offers a comprehensive documentation of its own, including a growing number of example patches. All of the new help files provide supporting meta info contained within the META subpatch that is also used by the autotips, thereby enabling easier prototyping of abstractions and documentation.

3.4 New and Improved Objects, Abstractions, and Libraries

Apart from the core Pd objects and improvements described in the Engine section above, including [trigger], [preset_hub], [preset_node], and [key], Pd-L2Ork offers a growing number of revamped objects while also pruning redundant and unnecessary objects. By doing so, its long term aspirational goal is to co-locate all the objects in a single folder and thereby abandon the subfolder structure for external libraries altogether.

Special attention was given to supporting the Raspberry Pi (RPi) platform with a custom set of objects designed specifically to harness the full potential of the RPi GPIO and I2C inter-faces, including [disis_gpio] and [disis_spi] [10]. The cyclone library has received new documenta-tion and a growing number of bugfixes and improvements. The [coll] object, for example, now offers threaded file reading and writing to prevent potential sample drops when used in complex patches on a low power hardware. Ggee library's [image] has received a significant overhaul and became the catchall solution for image manipulation, including accelerated displacement, support for file formats with alpha channel (e.g. PNGs), and size reporting. In addition to the standard Pd-extended libraries, Pd-

L2Ork has reintroduced the flext library with [disis_munger~] and an upgraded version of the [fluid~] soundfont synth external. Other libraries include fftease, lyonpotpourri, and RTcmix~. Pd-L2Ork bundles advanced networking externals [disis_netsend] and [disis_receive], convenience externals, like [patch_name], and abstractions (e.g. K12 and a growing number of L2Ork-specific abstractions designed to foster rapid prototyping). A few libraries have been removed due to lack of support and/or GUI object implementations that utilize hardwired Tcl-specific workarounds.

Most interpreted languages have mechanisms to do introspection. Pd-L2Ork features a collection of "info" or introspection classes for retrieving the state of the program on a number of levels, from the running Pd instance to individual objects within patches. Four classes provide the basic functionality:

- [pdinfo] reflects the state of the running Pd instance, including dsp state, available/connected audio and midi devices, platform, executable directory, etc.
- [canvasinfo] is a symbolic receiver for the canvas, abstraction arguments, patch filename, list of current objects, etc. The object takes a numeric argument to query the state of parent or ancestor canvases.
- [classinfo] offers information about the currently loaded classes in the running instance. This includes creator argument types, as well as the various methods.
- [objectinfo] returns bounding box, class type, and size for a particular object on the canvas.

While the introspection provided by these classes is relatively rudimentary, it alleviates the need for a large number of external libraries that add missing core functionality. For example, Pd-L2Ork ships with several compiled externals whose purpose is to fetch the list of abstraction arguments. These externals all have different interfaces and are spread across various libraries. Having an interface for fetching arguments that behaves similarly to other introspection interfaces improves the usability of the system. Furthermore, opening up rudimentary introspection to the user increases the composability of Pd. Functionality that previously only existed inside the C code can now be pushed to an abstraction, or to a collection of abstractions. Because abstractions don't require compilation and are written in Pd, they are more accessible to a wider number of users to test and improve them.

With L2Ork's focus on Wiimote devices as the primary ensemble controllers, Pd-L2Ork has furthered the development of the libcwiid that is provided as a custom fork with the most comprehensive support for Wii devices on Linux, including interleaved mode and a more recent variant of the Wii Motion Plus. Similarly, TkPath included with Pd-L2Ork is a fork of what appears to be an abandoned library that includes several bug fixes and improvements.

3.5 Scaffolded Learning and Rapid Prototyping

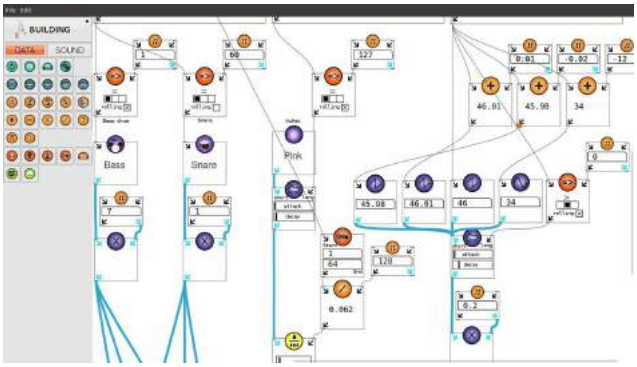


Figure 1: Pd-L2Ork K12 mode.

In 2011 Pd-L2Ork introduced a K12 learning module (Fig.) [11]. Embedded as a startup "-k12" option that modifies the user interface to address the needs of beginners and particularly young learners, the K12 module offers a collection of abstractions that encapsulate common functionality and thereby empowers users to quickly prototype an interactive sound-making instrument. Initially the external device support focused solely on the Wiimote family of controllers. Since its introduction it has been implemented in dozens of Maker-like experiences targeting various age groups [21]. As a result, its functionality was extended to support Arduino and eventually Raspberry Pi [10]. A further simplified K12 module variant was also used for a community-building 50 RPi installation titled Cloud [3] where community participants were given an opportunity to program the behavior of individual "cloudlets" in under an hour. In 2014, the K12 module was used for a gifted summer program where middle- and high-school students programmed and eventually performed in a Raspberry

Pi Orchestra. Most recently, we've begun experimenting with Pd-L2Ork and the K12 module as a rapid prototyping platform in robotics scenarios. This continues to be one of the primary thrust areas of both L2Ork and Pd-L2Ork.

3.6 Limitations

Apart from the ongoing need for further improving documentation and expanding its features, perhaps Pd-L2Ork's greatest limitation is its focus on the Linux platform. With its increased visibility, there was a growing community interest to support other platforms, namely Windows and OSX. The TkPath toolkit, however, lacks Windows support. As a result, there was a need for yet another GUI rewrite. This time its focus required a platform-agnostic solution.

4 Purr-Data Development Branch

The Tcl/Tk toolkit severely limits how usable Pd can be. Tk has no native notion of a hyperlink. It has no easy, deterministic way to separate autorepeat key presses from manual keypress. There is no easy way to use non-system fonts, no support for theming or native dialogs under GNU/Linux, and little default support for many of the common image formats.

When taken separately, each example above could be worked around given a few hours or days of clever playfulness. When taken as a whole, however, all the workarounds necessary to make Tk both responsive and usable across OSX, Windows, and GNU/Linux become too burdensome for a small or even medium-sized graphical free software community. Another, more modern toolkit is needed.

Tcl commands with Tk window strings are hard-coded into the C source files of Pd. This means that any port to a different toolkit must either replace those commands with an abstract interface, or write middleware that turns the hard-coded Tcl strings into abstract commands. Given the complexity of Tcl commands in both the core and external libraries, that middleware would essentially have to re-implement a large part of the Tcl interpreter. Consequently, Purr-Data opted for the former approach of directly implementing an abstract interface.

Adding to the porting difficulty is the fact that Pd has no formal specification, and its GUI interface follows no common design pattern for 2D graphics. For example, the GOP window appears at a glance as a viewport that clips to a specified bounding box. However, the bounding box itself behaves inconsistently–for built-in widgets like [hslider] or [bng] it clips (per widget, not per pixel), but for graphed arrays, data structure visualizations, and widget labels it does no clipping at all.

For this reason, Purr-Data uses a GUI toolkit called nw.js that allows the Pd canvas to be drawn and manipulated using the HTML5 API. Since the HTML5 API is a widely documented and used, there is an enormous pre-existing knowledge and developer base for it. Furthermore, the code to display Pd patches in HTML5 will work in any modern GUI toolkit that has a webview widget.

4.1 Pd Canvas as SVG

Purr-Data implements a Pd canvas as an SVG document. SVG was chosen because it is a mature, widely-used 2D API. Unlike HTML5 canvas, larger canvas sizes have little to no performance impact on the responsiveness of the graphics. Since Pd users sometimes employ unusually large logical canvas sizes, this responsiveness makes SVG a better choice for drawing a Pd canvas than the HTML5 canvas API.

4.2 Leveraging the SVG Spec to Improve Pd Data Structures

Purr Data includes a large number of improvements to data structure visualization. In order to achieve this, a small subset of the SVG specification was used.

Inheriting from a pre-existing standards-based 2D API has several advantages over an ad-hoc approach. First, if implemented consistently, the extant documentation can be used to test and teach the system. Second, it is not necessary to immediately understand all the design choices of the entire specification in order to implement parts of it. Since those parts have been used and tested in a variety of mature applications, it makes it easier to avoid design mistakes that might otherwise be likely for someone who isn't a 2D graphics expert. Finally, there is less risk of a standards-based API becoming abandoned than a more esoteric API.

To improve data structure visualizations, several [draw] commands were added which map to the basic shape/object types in SVG: there is [draw circle], [draw ellipse], [draw rect], [draw line], [draw polyline], [draw polygon], [draw path], [draw image], and [draw group]. [draw text] has not been implemented yet. Each has

a number of methods which map directly to SVG graphical attributes. Methods were also added for Document Object Model (DOM) events to trigger notifications to the outlet of each object. Unfortunately, Pd has no easy way to put key/value pairs in messages, so the outlet gives a message with positional arguments.

Figure 2: Purr-Data patch with a complex interactive SVG data structure.

The screenshot in Fig.1 shows the famous "SVG tiger" drawn from a few hundred paths found inside the [draw group] object. Even though the drawing is complex, Purr-Data caches the bounding box for the tiger object to prevent the hit-testing from causing dropouts. One can mouse over the tiger and trigger real-time audio synthesis.

5 Observed Impact

Pd-L2Ork has seen international adoption among other laptop ensembles, in curricula, and a growing number of outreach initiatives, including Maker camps, workshops, gifted programs. In 2010, L2Ork has formed a long-term partnership with a regional Boys & Girls Club of Southwestern Virginia where a number of initiatives of this kind have taken place. It has also helped start over half-dozen other similar initiatives across North and South Americas. The partnership with the regional Boys & Girls Club also served as the foundation for the initial exploratory study of the K12 module. Pd-L2Ork has been instrumental in ensuring L2Ork's recognition. In 2014 L2Ork was named as one of the top eight research projects at Virginia Tech [1] and in 2015 as one of the top six national transdisciplinary exemplars among research institutions [2]. With the prospect of cross-platform support, Pd-L2Ork is increasingly seen by some as Pd-extended's spiritual successor.
Since 2012, Pd-L2Ork is also being used extensively for teaching computer music courses at the Computer Music Research Group of the Johannes Gutenberg University (JGU) in Germany. The main reasons for switching from the vanilla and extended flavors of Pd initially were the GUI improvements (especially the ability to configure GUI objects and graph-on-parent patches in a graphical way) and the infinite undo capability, which make Pd-L2Ork much easier to use for beginners. On the other hand, one obstacle with Pd-L2Ork was that it required Linux, which hampered adoption by students (who, at least at the JGU, are often using Mac and Windows systems as their daily platform). But this is about to change now with the advent of Purr-Data.

5.1 Availability

Because of Pd-L2Ork's addons and its comprehensive set of bundled externals, the software has a lot of dependencies and a fairly complicated (and time-consuming) build process. So, while the software can be built straight from the source (which is best done using the included tar_em_up.sh build script located in the l2ork_addons subfolder), it is much easier to use one of the available binary packages:

- Virginia Tech's official Pd-L2Ork packages are available at `http://l2ork.music.vt.edu/main/make-your-own-l2ork/software/`, and
- Jonathan Wilkes' Purr-Data packages can be found at `https://git.purrdata.net/jwilkes/purr-data-binaries/tree/master`.

In addition, at the JGU we have created a Debian source package which can be used to build packages for all recent Ubuntu releases on both 32 and 64 bit architectures, along with a Makefile which fully automates the build process and also makes it easy to roll your own packages from current git sources. This build system is available in the debuild subfolder of the Pd-L2Ork and Purr-Data git repositories. Corresponding binary packages can be found on Launchpad. As of summer 2016,

these are also linked to from the Pd-L2Ork website as the default Linux packages. In a similar vein, we also support Arch Linux and its derivatives by means of corresponding package builds in the Arch User Repositories, as well as a binary package repository hosted on Bitbucket. More information about these, including pointers to the Ubuntu PPAs, binary Pacman repositories and bug trackers for reporting packaging issues, can be found at `http://l2orkubuntu.bitbucket.org/` and `http://l2orkaur.bitbucket.org/`, respectively. In the future, we also plan to improve support for recent OSX versions by creating a similar MacPorts package.

The above repositories also contain Pd-L2Ork versions of JGU's Faust and Pure externals that allow for running signal processors written in Grame's Faust and JGU's Pure programming languages [14] [13]. These enable you to program your own Pd externals in a high-level, functional program-ming language. Faust is tailored for creating au-dio effect and instrument plug-ins, while Pure is a general purpose language geared more towards advanced symbolic processing of control messages. Both are well-integrated in the Pd environment and support an interactive (live-coding) develop-ment style, since Faust and Pure modules can be reloaded dynamically while the patch keeps running. Please note that these extensions re-quire an installation of the Pure runtime environ-ment, which is readily available through a separate Ubuntu PPA and the Arch User Repositories; de-tailed instructions can be found under the links provided above.

6 Future Work

Following the stable release of the cross-platform version tentatively scheduled for the winter 2016 (Purr-Data is currently in beta), Pd-L2Ork's roadmap includes continued improvement of program's usability, consolidation of externals and libraries, and the pursuit of the remaining bugs. In addition, the team would like to work on designing libPd-L2Ork and streamlining the Pd-L2Ork to libPd-L2Ork pipeline, as well as continue to develop frameworks for the embedded scenarios and recently established thrust areas, such as RPi and robotics.

With the imminent expansion onto other platforms, Pd-L2Ork's key challenge is ensuring sustainable growth. This can be achieved through fostering greater community participation in its development and maintenance, as well as in securing necessary resources. If interested in contributing to the project regardless of the aspirational capacity please do not hesitate to contact us.

7 Acknowledgments

The authors would like to thank the original Pd author Miller Puckette, numerous community members who have complemented the Pd ecosystem with their own creativity and contributions, including Hans Christoph Steiner and Mathieu Bouchard. We would also like to thank the L2Ork sponsors and stakeholders without whose support Pd-L2Ork would have never been possible nor sustainable.

References

[1] 8 Awesome Research Projects At Virginia Tech, One Of The Top R&D Institutions In The U.S. http://dcist.com/2014/11/8_awesome_research_projects_at_virg.php.

[2] a2ru Selects Transdisciplinary Exemplars for the 2015 a2ru National Conference. http://a2ru.org/knowledgebase/a2ru-selects-transdisciplinary-exemplars-selected-for-the-2015-a2ru-national-conference/.

[3] Cloud | Ivica Ico Bukvic. http://ico.bukvic.net/main/cloud/.

[4] [PD-announce] MacOSX installers for pd 0.36 and pd 0.36 extended (CVS).

[5] pd forks WAS : Keyboard shortcuts for "nudge", "done editing". http://permalink.gmane.org/gmane.comp.multimedia.puredata.general/79646.

[6] TkPath. http://tclbitprint.sourceforge.net/.

[7] Unity - Game Engine. https://unity3d.com.

[8] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention*, volume 291. Citeseer, 2011.

[9] I. Bukvic, T. Martin, E. Standley, and M. Matthews. Introducing L2ork: Linux Laptop Orchestra. In *Interfaces*, pages 170–173, 2010.

[10] I. I. Bukvic. Pd-L2ork Raspberry Pi Toolkit as a Comprehensive Arduino Alternative in K-12 and Production Scenarios. In *NIME*, pages 163–166, 2014.

[11] I. I. Bukvic, L. Baum, B. Layman, and K. Woodard. Granular Learning Objects for Instrument Design and Collaborative Performance in K-12 Education. In *New Interfaces for Music Expression*, pages 344–346, Ann Arbor, Michigan, 2012.

[12] P. Davis and T. Hohn. Jack audio connection kit. In *Proc. Linux Audio Conference, LAC*, volume 3, pages 245–256, 2003.

[13] A. Gräf. Signal Processing in the Pure Programming Language. In *Proceedings of the 7th International Linux Audio Conference*, pages 137–144, Parma, 2009. Casa della Musica.

[14] A. Gräf. Pd-faust: An integrated environment for running Faust objects in Pd. In *Proceedings of the 10th International Linux Audio Conference*, pages 101–109, Stanford University, California, US, 2012. CCRMA.

[15] D. Iglesia. MobMuPlat (iOS application). *Iglesia Intermedia*, 2013.

[16] K. Jolly. Usage of pd in spore and darkspore. In *PureData Convention*, 2011.

[17] J. Kincaid. RjDj Generates An Awesome, Trippy Soundtrack For Your Life. http://social.techcrunch.com/2008/10/13/rjdj-generates-an-awesome-trippy-soundtrack-for-your-life/.

[18] C. McCormick, K. Muddu, and A. Rousseau. PdDroidParty-Pure Data patches on Android devices. *Retrieved January*, 21, 2014.

[19] M. Puckette. Pure data: another integrated computer music environment. *In Proceedings, International Computer Music Conference*, pages 37–41, 1996.

[20] M. Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

[21] B. Sawyer, J. Forsyth, T. O'Connor, B. Bortz, T. Finn, L. Baum, I. I. Bukvic, B. Knapp, and D. Webster. Form, function and performances in a musical instrument MAKErs camp. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 669–674, New York, NY, USA, 2013. ACM.

[22] B. B. Welch. *Practical programming in Tcl and Tk*, volume 3. Prentice Hall Upper Saddle River, 1995.

Granular Synthesis and Spatialisation in the Pure Data Environment

Oscar Pablo Di Liscia
Programa de Investigación "Sistemas Temporales y Síntesis Espacial en el Arte Sonoro", Escuela Universitaria de Artes, Universidad Nacional de Quilmes
Roque Saenz Peña 352
Bernal, Argentina, B1876BXD
odiliscia@unq.edu.ar

Abstract

This paper describes the design and some particular applications in granular synthesis and spatialisation of a *Pure Data* external developed by the author. It was developed with the collaboration of Damián Anache and Esteban Calcagno as a part of the research program "*Sistemas Temporales y Síntesis Espacial en el Arte Sonoro*" (UNQ, 2015-2019). The external offers all the capabilities of *GS* synthesis with full 3D control of the location of each grain through the *Ambisonic* technique. The general approach of its design is to use audio signals stored on a "pool" of tables as sources for both the grains and the amplitude envelopes, allowing several shuffling effects and synthesis/sequencing by concatenation.

Keywords

Sound Synthesis, Sound Spatialisation, Ambisonics, Granular Synthesis.

1 Introduction

Generally speaking, *Granular Synthesis* (from here on abbreviated as *GS*) is a technique based on the juxtaposition of small portions (called *grains*) of a source audio signal. In a typical *GS* application, the user is allowed to set (and eventually change over time) several parameters of a grain sequence, the most common of which are: duration, temporal gap, source audio signal, amplitude function, peak amplitude, and pitch shifting.

There are many excellent *GS* computer applications, either as standalone programs or as dedicated modules, some of the most important being the *GS* Opcodes for *Csound* (Specially *Granule* and *Partikkel*, [2], [7], [16]), the *GS* units for *SuperCollider* ([15], see also Del Campo 2011 [5]) and several external and abstractions for Pure Data ([11]). Of the latter, the most complete and powerful is *disis_munger~* ([3]), which is based on Dan Trueman's *munger~*. Of the many *GS* standalone programs available, one of the most powerful and versatile is *HourGlass* ([17]). There are also many *GS* software units in the form of *Plug-ins* (such as VST, JSFX, or any other format) to be used on Digital Audio Workstations.

GS can be refined in many different ways and allows the production of very complex and rich sonic events. Among the many interesting features of *GS*, it is worth mentioning its capability to produce smooth transformations of individual sonic event sequences into swarms of sounds (*sonic objects*) and textures or *vice versa*. The use of *GS* on electro acoustic and audio visual creation is very well known and widely documented (see, among others, Roads, 2004 [13], Batty et al, 2013 [1], Del Campo, 2011 [5]), as well as electronic works by many composers. The works by Barry Truax and Horacio Vaggione are referred to in Roads's *Microsound* ([13] pp.235) as those which explore extensively many of the possibilities of *GS* synthesis since approximately 1970. Many works by the composer Juan Pampin combine *GS* and sound concatenation using high resolution spectral analysis, audio descriptors, and sound spatialisation most effectively, to mention just one of the many recent applications of this technique in sonic creation (See, for example, *a line* (2015, [12])).

2 The *my_grainer~* external

The *my_grainer~* external[1] offers all the capabilities of *GS* synthesis with full 3D control of the location of each grain through the *Ambisonic* spatialisation technique (see below). It was developed by Pablo Di Liscia with the collaboration of Damián Anache and Esteban Calcagno as a part of the research program "*Sistemas Temporales y Síntesis Espacial en el Arte Sonoro*" (Universidad Nacional de Quilmes, 2015).

The general approach of *my_grainer~* is to use audio signals stored on a "pool" of tables (maximal, 24 of them, at present) as source for both the grains and their amplitude envelopes. The user is responsible for the creation of the tables, which can be done by either reading

1 http://puredata.info/Members/pdiliscia/grainer/grainer.zip/view (Last access: 09/2016)

pre-existent sound files, or using signals created "on-the-fly" by synthesis methods or obtained from real-time audio input. The strategy of using tables instead of a signal input as the audio grain sources was selected in order to allow shuffling effects using several different audio sources. The user is allowed to set the audio source and the amplitude envelope tables for the next grain to be synthesized by means of a message. A new message may either force the same table to be overwritten with a different audio content or to set a different table for the next audio grain to be synthesized. There is also the possibility of alternating different audio tables as audio sources and/or amplitude envelopes out of the set of different table names given to the external.

The choices for the grain duration, temporal gap, peak amplitude, audio starting and ending read point on the table, pitch transposing and/or warping, and spatial location (azimuth angle, elevation angle and distance) are handled by methods that allow the setting of a base value and a random deviation. There is also the possibility of setting the grain duration, temporal gap, source audio table, starting read point and pitch transposition by means of a random selection out of a set of discrete numeric values stored in lists. The probability distribution function of the random selection is uniform, but non-uniform biases may be achieved including several repeated values in the list, increasing the likelihood of their occurrence. If the grain duration exceeds the time span determined by the read and end points of the audio table, the user may choose to use three types of loops (no loop, forward loop or forward-backward loop).

The parameters for each grain (with the exception of pitch warping) are invariant over its duration, which means that any change will be taken into account for the next grains to be synthesized. Most parameters may be changed with a precise control for each one of the grains. As an example of this, the time gap between the grains can be set in five different ways:

(a) Simply setting a base value in seconds without any random deviation. This is usually referred to as *synchronous GS* (in *GS* terminology).

(b) Simply setting the range of a random deviation in seconds and setting the base value to zero. This is usually referred to as *asynchronous GS* (in *GS* terminology).

(c) A combination of the *b* and *c* possibilities will produce a more controllable *asynchronous GS*.

(d) Setting a list of discrete gap times which will be chosen at random for each grain, to produce *asynchronous GS* and to explore the potential of "time fields". In this case, the value previously set on *a* (if any) will be disregarded. As this list can be changed by successive messages, this can also prove useful in order to produce gradual transformations from sequences of individual sonic events with statistically controlled time gaps on textures or granulated sounds.

(e) A combination of the *b* and *d* possibilities is also possible, allowing random deviations of the discrete gap values.

A comprehensive Tutorial on *GS* using *my_grainer~* is being developed by Damian Anache[2] and a *Pure Data* patch called *GRAINER_DINAMICS* was developed by Esteban Calcagno[3] in order to allow the automation of the parameters of *my_grainer~* through editable functions stored in tables, so that the users may store and retrieve very complex simultaneous sequences of parameter changes.

Finally, it should be noted that some features of the external were also designed taking in account its possible use on *concatenative synthesis*. As noted by Diemo Schwarz (Schwarz, 2005 [14]): *GS can be seen as rudimentarily datadriven, but there is no analysis, the unit size is determined arbitrarily, and the selection is limited to choosing the position in one sound file. However, its concept of exploring a sound interactively could be combined with a pre-analysis of the data and thus enriched by a targeted selection and the resulting control over the output sound characteristics, i.e. where to pick the grains that satisfy the wanted sound characteristics.* As *my_grainer~* can also receive and use a list of pairs of starting times (in the audio table in use) and durations for the grains, it may be useful to achieve a sort of *concatenative synthesis* as well. Each start/duration pair is chosen at random out of the list, which could be made up of data coming from a pre-analysis and may even be updated by data coming from real time audio input being analyzed.

3 Granular synthesis and spatialisation using *my_grainer~*

my_grainer~ was specially developed to experiment on *GS* spatialisation. This poses some issues that should be taken into account. If a non-overlapping sequence of grains is produced by a *GS* synthesis unit, each of the grains could be individually processed by

2 http://puredata.info/Members/pdiliscia/grainer/my_grainer_Tutorial_by_Damian_Anache.zip/view (Last access: 09/2016)

3 http://www.mediafire.com/GRAINER_DINAMICS.zip (Last access: 09/2016)

spatialisation units that are independent of the *GS* unit (still, there is the problem of accurate synchronization of each grain with a time-dependent spatial location). However, if the grains overlaps, an individual spatial location of each one is clearly not feasible unless it is achieved from inside the *GS* unit. This has the drawback that the spatialisation technique used may not be freely chosen by the user. It is feasible to produce a *GS* unit capable of performing internally several individual spatialisation techniques (such as *Intensity Panning*, *VBAP*, *Ambisonics*, *WFS*, etc.), but some tradeoff between capabilities and efficiency must be taken in account. That is the reason for which *my_grainer~* was developed to produce mono, stereo (*intensity panning*), quad (*Ambisonic* B Format First Order), nine channel (*Ambisonic* B Format Second Order) or sixteen channel (*Ambisonic* B Format Third Order) outputs. *Ambisonics* is a technique that provides robust, versatile, and standarized 3D spatialisation. It is widely used and documented (See Gerzon [6] and Malham [10]). Despite its perceptual effectiveness (which may improved using high order encoding), perhaps the most valuable of its features is that it provides a way of keeping and transmitting both the audio signals and their spatial characteristics in such a way that the composer can think of spatial features independently of the number of channels that will be available on a particular performing situation. Another interesting feature of *Ambisonics* is the set of operations of field manipulation (which includes rotation, tilt, tumble, dominance and mirroring) that can be performed on its standard formats with relatively simple equations.

my_grainer~ may produce mono, stereo (*intensity panning*), quad, nine channel or sixteen channel outputs. The *quad*, *nine channel* and *sixteen channel* outputs consist respectively of the four (W, X, Y, Z) or the nine (W, X, Y, Z, R, S, T, U, V) or the sixteen (W, X, Y, Z, R, S, T, U, V, K, L, M, N, L, O, P, Q) encoded signals of an *Ambisonic B* first, second or third order format, which must be further decoded by the user for the available number of channels/loudspeakers of the system using appropriate externals or abstractions. For the *Pure Data* environment, an excellent choice would be the *HOA* Library (see Hum [8]). An extra (*rightmost*) signal outlet is always created to deliver a reverberation send signal which can be used in distance cues simulation (see below).

The user may spatialise a complete stream of grains by feeding the input of *ad hoc* externals or abstractions already available for that purpose using the mono output of *my_grainer~*. Some externals and/or abstractions that generate 3D trajectories based on equations of specially relevant curves (such as *Lissajous*, *Spirals*, *Hypothrochoids*, *Epithrochoids*, etc.) may be used to drive the spatial location of a stream of grains. The set of abstractions developed by Pablo Cetta for *Pure Data* may be an excellent choice for that purpose (see Cetta [4]).

If an individual spatial treatment of the grains is desired, the *Ambisonic* output of *my_grainer~* may be very effectively used, but it must be noted that both *Intensity Panning* and *Ambisonic* are angular location techniques. It is also important to stress the fact that, as *intensity panning* uses the *IID* cue (*Interaural Intensity Difference*), it may be used to produce gradual changes in the azimuth angle, but not in the elevation angle of the spatial position of a virtual sound source, while *Ambisonics* may be used for both, azimuth and elevation location.

The distance simulation of *my_grainer~* is achieved by scaling the amplitude of each grain by the inverse of a distance value that may also have a random deviation. Each grain may also be delayed by a duration computed taking into account the virtual source distance and a sound speed value provided by the user. Within closed environments, the ratio between the direct and reverberated signals and the time alignment between the direct and the reverberated signals are both simple, but perceptually effective distance cues. These may be easily obtained by feeding the input of a reverberation unit with an output signal of *my_grainer~* with neither of the amplitude/time cues aforementioned (which is always provided in the rightmost outlet).

Curtis Roads ([13], pp. 221) deals with the spatialisation of sound particles through two approaches: one consisting of the simulation of virtual spaces and sources, and the other consisting of several particle modulations. In what follows the latest will not be commented because it exceeds the scope of this paper. The former includes what Roads calls: “scattering particles in virtual spaces” and “per grain reverberation”. The capacities of *my_grainer~* already described show clearly that it allows both full 3D spatial movement of virtual sources, as well as morphing of the dimensions of sound clouds formed by small *quanta* of spatially spread sound.

From the wide number of relations that may be drawn between sound structure and sound spatial quality, another interesting possibility for which *my_grainer~* was initially designed is to work on the disruption of the spatial image of virtual or acoustic sources, as described by Gary Kendall (2009 [9]). Essentially, Kendall states that the spatial treatment of sound in

electro acoustic music makes possible an interplay with -among other features- the perceptual grouping of the sonic events. This may be achieved through the gradual modification of a *GS* stream in order to produce the sensation of listening either to a single sound source with a variable spatial width, a spatially granulated texture or a group of individual sound sources with or without coherent sound images. In summary, the use of the *GS* Techniques (time-domain decomposition and recomposition) combined with the spatialisation techniques may allow the production of sound images which have not only interesting timbral qualities but also surprising and even paradoxical spatial features.

4 Conclusions

my_grainer~ is an external that extends the possibilities of *GS* offering features that are not present in other existing abstractions/externals in the *Pure Data* programming environment, specially in the spatial treatment of sound. Since the external is in continuous development, its code may be further improved in order to speed up its performance. The external was tested in several performance situations showing robustness and efficiency and will allow composers to explore and experiment with spatial *GS* extensively. Some features of the external were also designed taking into account its possible use on *concatenative synthesis*, also combined with spatialisation, which may be one of the future lines of development.

5 Acknowledgements

Our thanks go to the Universidad Nacional de Quilmes, Buenos Aires, Argentina for supporting the Research Program in which this development is being accomplished.

References

[1] J. Batty, K. Horn and S. Greuter : Audiovisual granular synthesis: micro relationships between sound and image. *Proceedings of The 9th Australasian Conference on Interactive Entertainment: Matters of Life and Death*, (p. 8), ACM, Australia, 2013.

[2] Oeyvind Brandtsegg: *Applications using the partikkel opcode*. http://oeyvind.teks.no/results/applications/partikkelapplications.htm (Last access: 09/2016)

[3] Ivica Ico Bukvic et al: *disis_munger~ (a.k.a. munger1~)* source code, binaries and documentation. https://github.com/gilbertohasnofb/pd-abstractions-and-libraries/tree/master/disis_munger~ (Last access: 09/2016)

[4] Pablo Cetta: “Trayectorias espaciales a partir de curvas notables”. *Ideas Sónicas*, CMMAS (Centro Mexicano para la Música y las Artes Sonoras), Morelia, México. http://sonicideas.org/info.php?vol=8&num=16&secc=notes (Last access: 09/2016)

[5] Alberto Del Campo: “Microsound”, In Scott Wilson, David Cottle, and Nick Collins: *The SuperCollider Book*, The MIT Press, London, UK. pp. 463-504, 2011.

[6] M. A. Gerzon: “Periphony: With-height Sound Reproduction”, *Journal of the Audio Engineering* Society, vol. 21, N° 1, pp. 2-10, 1973.

[7] Joaquim Heintz et al: *Csound FLOSS Manual*. http://write.flossmanuals.net/csound/f-granular-synthesis/ (Last access: 09/2016)

[8] Dionys Hum: *HOA Library.* http://www.mshparisnord.fr/hoalibrary/en/downloads/puredata/ (Last access: 09/2016)

[9] Gary Kendall: "La interpretación de la espacialización electroacústica: atributos espaciales y esquemas auditivos”. Gustavo Basso, Oscar Pablo Di Liscia y Juan Pampin (Eds.). *Música y espacio: ciencia, tecnología y estética*. Bernal: Editorial de la Universidad Nacional de Quilmes, pp.241-259, 2009.

[10] Dave Malham: “El espacio acústico tridimensional y su simulación por medio de Ambisonics”. Gustavo Basso, Oscar Pablo Di Liscia y Juan Pampin (Eds.). *Música y espacio: ciencia, tecnología y estética*. Bernal: Editorial de la Universidad Nacional de Quilmes. pp 161-201, 2009.

[11] Nick Mariette: *Some Granular Synthesis Implementations In Pd.* https://puredata.info/Members/nmariette/granular-implementations/?searchterm=Syncgrain~%20Synchronous%20Granular%20Synthesis (Last access: 09/2016)

[12] Juan Pampin: *A line.* https://dxarts.washington.edu/creative-work/line (Last access: 09/2016)

[13] C. Roads: *Microsound*, The MIT Press, England, 2004.

[14] Diemo Schwarz: "Current research in concatenative sound synthesis". *Proceedings of the International Computer Music Conference (ICMC)*, Barcelona, Spain,

September 5-9, 2005.

[15] SuperCollider 3.8dev Documentation:
http://doc.sccode.org/Classes/GrainBuf.html
http://doc.sccode.org/Classes/GrainFM.html
http://doc.sccode.org/Classes/GrainIn.html
http://doc.sccode.org/Classes/GrainSin.html
(Last access: 09/2016)

[16] Barry Vercoe et al: *The Canonnical Csound Reference Manual*.
http://www.csounds.com/manual/html/SiggenGranular.html (Last access: 09/2016)

[17] Xenakios: *HourGlass granulator download page.*https://xenakios.wordpress.com/2012/04/29/hourglass-1-0-1-released/ (Last access: 09/2016)

Música Móvel Apps - Experimentation and open design for musical instruments in Android

Cristiano Figueiró
Universidade Federal
da Bahia
Rua Dom Basilio Mendes Ribeiro,
55 Salvador/BA, Brasil, 40170-120
figocris@gmail.com

Guilherme Soares
Universidade Federal do
Recôncavo da Bahia
Caixa Postal 7817
Salvador/BA, Brasil, 41900-970
organismo@gmail.com

Bruno Rohde
Curso Tecnologias Livres na
Criação Artística
Travessa Baependi, 08, ap 104
Salvador/BA, Brasil, 40170-090
brunorohde@gmail.com

Abstract

In this article we report the development of a suite of Android musical instruments "Música Móvel" based on combination of LibPD for the DSP and Processing language for the graphical interface. The article shows the interaction design decision details for the interface of the first eight applications developed to the present stage of work: Looper, D-Sonus, Arvoritmo, Horizons, PhotoSintese, B/I/T/S/L/C, Multigranular and Anartistas. We explain the electroacoustic techniques used in each of the applications, some related conceptual problems and its graphical interface operation serving here as a basis for decision-making pathways think of the project.

Keywords

Mobile Music, LibPD, Processing, Android.

1 Introduction

In this paper we report interaction design decision details for the interface of the first eight applications developed to the present stage of work: Looper, D-Sonus, Arvoritmo, Horizons, PhotoSintese, B/I/T/S/L/C, Multigranular and Anartistas. We explain below the electroacoustic techniques used in each of the applications, some related conceptual problems and its graphical interface operation serving here as a basis for decision-making pathways think of the project.

2 Looper - Microphone as sound source material

The audio from any recordable sound source quickly becomes an experimental material. The fact that each channel is automatically repeated creates a temporal extension of the present moment. Audio is captured, creating spontaneous interaction with a perceived landscape. Repetitions in different asynchronous layers (in 4 different channels) creates an interactive cluster of slices at different speeds creating an amalgam between expansion and contraction of interaction sound between recent memory and some of its debris.

A certain "quantization" of the recorded audio formatting operates a snapshot result of the interaction, but ensures a more "musical" experience by the untrained user in overlapping rhythms in different layers of audio. Nevertheless, the quantization does not affect the time position of the audio peaks, choosing a moderate rhythmic quantization, which results in most cases an average BPM for the strong pulses, but also enables a wide variety of rhythmic complex overlays to show up.

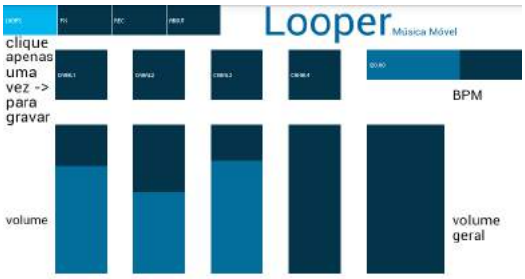


Fig. 1 - Looper main interface

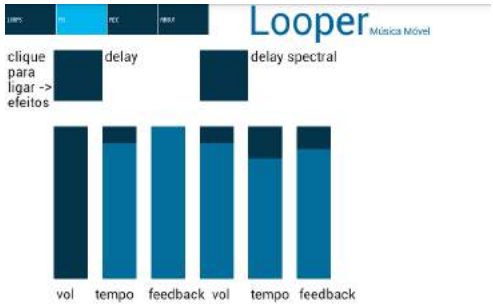


Fig. 2 - Looper effects interface

During the development we had to abandon the original feature of drawing in the recorded audio wavetable on each channel, on the style of multi-track audio editing programs. This was a matter of visual feedback by viewing the waveform and loop scroll bar as compositional decision. If in one hand, the audio gain synesthesia with the graphical interface on the other side it would lost processing power and consequently the code would run on a smaller number of devices. Greater synesthesia by visual feedback provides greater control of rhythmic parameters, but ended up creating more latency audio creating a lot of rhythmic instability.

The input audio processing causes a change of instantaneous sound reality, this experience approaches the concept of "digital drugs" that alters our perception of time and space. Typical uses and results: creating rhythmic textures sessions ranging from simple combinations to complex polyrhythms.

3 B/I/T/S/L/C and Multigranular - Sample processing

The issue of "slicing" and reassembling of a pre-recorded audio as an engine of an electroacoustic aesthetics goes back to the time of tape editions, with the concrete music questioning and shifting sounds reference to re-contextualize a reduced listening in their textural properties or acoustic, abstracting the original source.

Most recently we can find an aesthetic of saturated collage of cultural references to the point of being able to play with the question of individual authorship, an already iconic position in the age of downloads, as in John Oswald's music (and their recombinatory game called pastiches "plunderphonics") or conceptual experiments such as Johannes Kreidler (as when you compress all the works of the Beatles or Beethoven in a second).

B/I/T/S/L/C (beatslicer) is a software based in the possibility of constructing recombinated audio slices sequencing, that can be scaled, inverted, synchronized or have their transposed tone generating motivical variations that can be associated with great similarity, generating material with rhythmic diversity and easy contextual grouping.

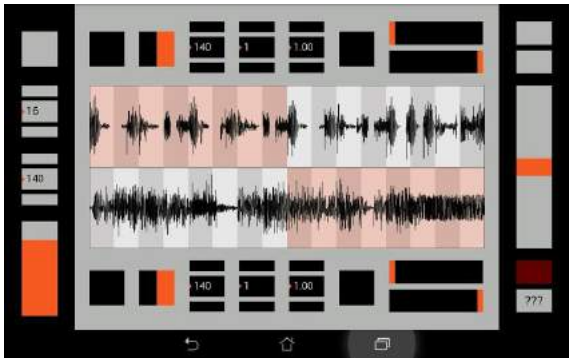


Fig. 3 – B/I/T/S/L/C main interface

Fig. 4 – B/I/T/S/L/C help view

Also the Multigranular app offers a way to "navigate" in the waveform taking the possibility of multi-touching to create a very close relationship of the musical instrument of manipulation to something tactile and instinctive. In addition to choosing the .wav file used as a sound source, you can change the size of each file read grain (generated by each touch), and has also added a simple effect of delay-feedback that amplifies the sound gestures in time. It can also be used to generate momentary or infinite loops (with feedback 100%).

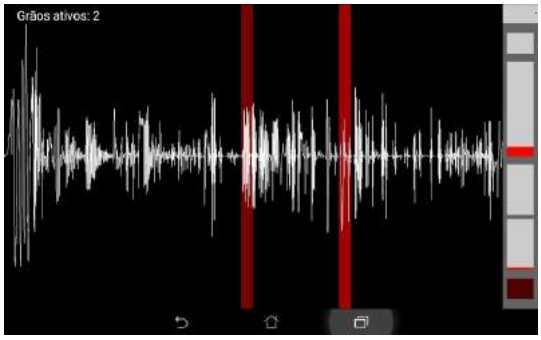


Fig. 5 - Multigranular interface

4 Arvoritmo and D-Sonus - Expansion of the concept of Sequencer

Sequencers traditionally act as readers left to right as a direct analogy to musical notation in pentagram or punched cards of pianolas and

organs, performed mechanically. The Arvoritmo and D-Sonus applications have interfaces that seek to break with this paradigm, offering more experimental experiences in graphical interaction.

During the development of Arvoritmo and D-Sonus we studied and analyzed various types of sequencers, such as piano roll, traditional notation, tracker, flowchart and other styles as some games style "guitar hero".

In Arvoritmo application, the solution was creating bifurcations branches for paths of rhythms closely related to the computational logical thinking of graph algorithms, with prioritization of competing or parallel decisions.

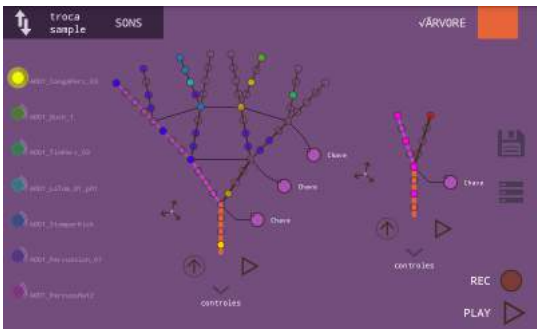


Fig. 6 - Arvoritmo main interface

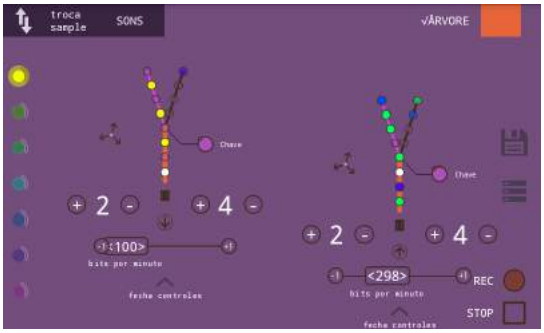


Fig. 7 – Arvoritmo BPM setting

D-Sonus emerges as an attempt to create a synesthesia between drawing and sound, extending the performer's gesture. The interface invites the musician to draw paths, there is a direct correspondence between the design and the resulting sound free trace.

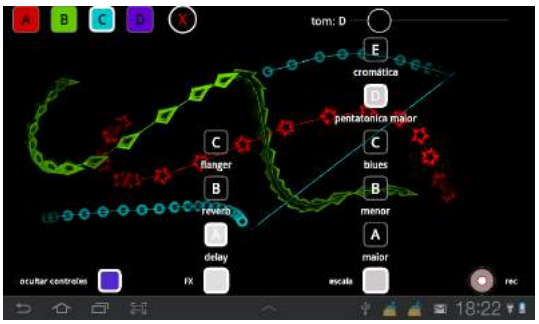


Fig. 8 – D-Sonus main interface

Every gesture is stored in a temporary memory that the application is repeating as a loop, and this loop is both visual and audible. The user can choose up to 4 instruments, colors or shapes brush and combine them by adding or removing elements. The other sound control possibilities are the choice of tonality of a melody, its melodic scale and sound effect applied, creating layers of textures.

5 Horizons and PhotoSintese - Computer vision

In two cases, the applications Horizons and PhotoSintese uses algorithmic procedures for detecting shape and color images provided in real time by the camera of the devices which are decisive for the composition of musical parameters. In these cases also considered a reflection on the issue of synesthesia in direct analogies between visual form and musical form, discussing this feature.

The Horizons application makes a camera accessed by image analysis and calculates visual contours present maps and boundaries to the melodic and rhythmic creation. The calculation of the contour is performed in the comparison of pixels from one frame to its previous.

The algorithm is shown below in a function in Processing:

```
void draw () {
    cam.loadPixels ();
    int x = 0;
    for (int i = 0; i <320; i = i + (int) (320 / (float) 64)) {
            int j = 0;
            boolean floor = false;
            while (j <240 &&!floor) {
                    if (brightness (cam.pixels [320 * j + i])> 100) {
                            j ++;
                    } Else {
                            floor = true;
                    }
            }
            contornoY [x] = j;
            x ++;
            }
    cam.updatePixels (); (…)}
```

Fig. 9 - Horizons main interface

The PhotoSintese application graphically represents the RGB values of the video captured with the tablet's camera or smartphone. This wave will be transformed into sound, with both still image and with the moving image.

The graph represents the amount RGB component darker (left) for the lighter (right).

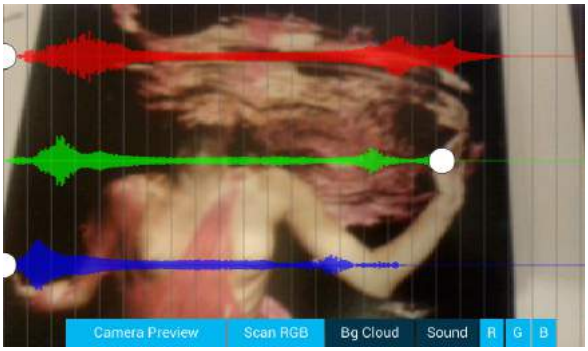


Fig. 10 - PhotoSynthese main interface

The result is the average amount of RGB colors in the camera area and is applied as three numerical vectors for modeling a sound synthesis, where each component RGB corresponds to a different synthesizer. The highest point of each RGB component is extracted to match the height of the modular synthesizer and spectrum parameters.

A problem to be taken into account in the direct conversion of image and sound is the fact that colors or contours of a two-dimensional or three-dimensional visual object can not guarantee a musical discourse consistency from a synesthesia inferred in immediate analogy to a vector of numerical parameters.

The fact, for example, that intuitively your perception infer "low frequencies" to "warm colors" or "the contour valleys and peaks in a pictorial horizon have a certain symmetry" to "balance of a sound" may in some cases generate a outstanding feeling of “figure-ground” (gestalt) of a soundscape; but it can not guarantee a subjective coherence of what is questioned by studies of music cognition as a fruition of a "compositional" gesture beyond a pre-musical and programmatic concept induced by semiotic references of these images and their related contexts.

6 Anartistas - Synthesis and multi-touch control over endless background

This application consists of a polyphonic synthesizer with its parameters controlled from a multi-touch interaction that draws all paths for a graph of multiple edges between the contact points. The parameters of the sound to be controlled are: frequency and volume, vibrato, low frequency oscillator (LFO), envelope, delay and distortion.

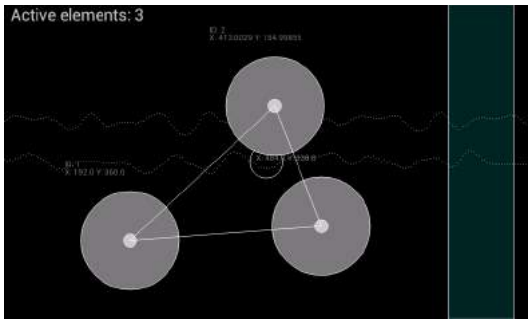


Fig. 11 - Anartistas main interface

The lack of buttons and explanations takes the user to a sound intuitive gaming experience which is ought to manipulate the interface for tactile discovery of parameter controls.

The synthesizer is tuned in a predefined range, which causes the application to enable results melodic consonants.

7 Conclusion

With the project Música Móvel was possible to see a full cycle of development of musical applications for mobile devices using a chain of well documented tools and accessible code.

The possibility of including in this procedure some programming languages already quite widespread in the arts - Puredata and Processing - facilitated the entry of developers with a more interdisciplinary and artistic profile, some of these still apprentices in programming and in search of a poetic and playful manner to deal with this knowledge.

The modular nature of libPD project also facilitates portability of the following to use with other programming languages, and it would be interesting in the near time trying to reproduce the chain of production used here

experiencing the python language as an intermediate layer and the ability to use hardware more modular and free as the Raspberry Pi and the like.

References

[1] Brinkmann, Peter et al: "Embedding pure data with libpd," *Proceedings of the Pure Data Convention,* 2011.

[2] Brinkmann, Peter: *Making Musical Apps*, O'Reilly Media, Inc., 2012.

[3] Farnell, Andy: *Designing Sound,* The MIT Press, 2010.

[4] Fry, Ben: *Visualizing Data: Exploring and Explaining Data with the Processing Environment,* O'Reilly Media, Inc., 2007.

[5] Glowinski, Donald; Mancini, Maurizio; Massari, Alberto: "Evaluation of the mobile orchestra explorer paradigm", *Intelligent Technologies for Interactive*

[6] *Entertainment,* Springer Berlin Heidelberg, 2012, p. 93-102.

[7] Silveira, Henrique Iwao: *Garden: Musical collage in experimental electronic music / Thesis (MS),* School of Communications and Arts - University of São Paulo, São Paulo, HIJ Silveira, 2012, 202 p.

[8] Thakur, Abesh: *Configuring Eclipse to work with Processing + libPd in Android,* Available at: <http:// twobigears.com/labs/configuring-eclipse-to-work-with-processinglibpd-in-android/ >

[9] Yuill, Simon: "All Problems of Notation Will be Solved by the Masses: Free Open Form Performance, Free / Libre Open Source Software, and Distributive Practice", *FLOSS + Art: Free Libre Open Source Art Poitiers*, France, GOTO 10, in Association with OpenMute, 2008.

The Context Modular Sequencer

Liam Goodacre
F1 802 Success Towers, Panchavati
Pashan, Pune 411008, India
liam.goodacre@gmail.com

Abstract

This paper presents Context, a powerful sequencer built in PD, and explores some new prospects for music composition made possible by the software. Through a series of case studies, it is shown how Context can represent musical structure and sequence many different sorts of patterns. By looking at Context's use of random number generation and internal communication, the paper explains how the composer can use Context to create generative music. The paper proposes the Context network—a collection of interconnected Contexts—as a medium of musical composition, and highlights the various possibilities and design choices that emerge during the development of a Context network.

Keywords

Context, sequencer, DAW, generative, composition

1 Introduction

Context is a modular sequencer built in PD. It aims to give the user more creative control over musical composition and to be versatile enough to fit into any PD patch, from beginner projects to sophisticated performances. Context is built with the familiar timeline paradigm at its core, but presents the timeline as a local object rather than a global environment. As Context is modular, it does not demand to be used in any particular way, inviting the user instead to create her own sequencing environment, according to her personal taste and style of composition.

I designed Context because I wanted a way of writing complex pieces of music in PD, as an alternative to Digital Audio Workstation sequencing. But building and testing the software has taken me in many directions that a conventional DAW could not, and has led me to question the very nature of musical composition. This paper presents a justification for the Context sequencer, along with some examples and musings on the kinds of music that it could create. It is not intended as a user manual: the software's functions are not listed in full and are described only insofar as they are necessary for the argument[1]. By the end of the paper, I hope that readers will appreciate what makes the Context sequencer special and have some ideas of their own about how they might use it.

[1] Full documentation is forthcoming at www.contextsequencer.wordpress.com

2 What is a sequencer?

A sequencer is a program or hardware unit that schedules events in time, for the purposes of making music. The events that it schedules are discrete in nature, consisting of simple messages (ones and zeroes, musical notes, etc), played back at the right time to create patterns and loops. Sequencers are arguably the greatest differentiating factor between electronic and acoustic music, because they relieve the musician from the obligation of playing each individual note. Computer music is *programmed* by the user/musician, not *played*. As such, sequencers are there to do what computers do best: to take care of routine operations, so that the user has time to focus on other, higher level decisions.

Sequencers are not the only attempt that has been made to program music. Musical scores are similar to sequencers in that they store music as information, to be replayed at a later date. As with sequencers, musical scores save the musician from the responsibility of memorizing or improvising. When she becomes literate, the musician has her repertoire expanded beyond the bounds of her memory, and the length and complexity of written music can be made correspondingly greater. In this way, sequencers can be seen not just as an invention of the digital age, but as the continuation of the classical attempt to formalize music. The difference is of course that a sequencer achieves a more complete degree of automation, containing both the score and the means to play it.

Sequencers occupy a small corner of electronic music theory, but the paradigm of scheduling events goes down to its very core. Waveforms are stored on computers as discrete data; their playback involves recalling the numbers at the right time and in the right order. Likewise, the timeline of a Digital Audio Workstation (DAW), populated with various recorded tracks and modulation envelopes, is also a way of organizing musical events in space so that they will occur at the right time. Oscillators, sequencers and DAW environments are all inherently concerned with determining what happens when, differing from each other only in scale, oscillators being the micro-, and DAW timelines the macro- approach to representing music as information.

2.1 Software sequencing environments

Most DAWs—indeed most pieces of software—provide an environment in which the user must function in a predefined way. Context does not take this approach. Instead, Context presents a small, simple building block which can be replicated and interconnected in many different ways. When more than one instances of Context interact with each other, Context networks are formed. It is these networks which define the musical composition.

Other software has adopted the network paradigm for representing composition. Nodal[2], one of the most successful generative sequencers, presents a circuit-diagram like grid where "nodes" represent musical events and random decisions. Koan and Noatikl[3] provide similarly unstructured environments where the user makes music by connecting various modular units. Many other programs have developed their own novel ways of representing composition spatially[4].

While such products provided some early inspiration, Context differs in that it is more completely modular. Context is not a suite; it is a tool. It is a PD abstraction which can be placed on any PD canvas, anywhere, and made to interact with other objects and existing patches. So rather than defining an environment to which the user must conform, Context lets users create their own sequencing networks within the PD environment. As such, a Context network stands closer to the paradigm of modular synthesizers—it even lets users "hack" an instance of Context beyond its original design[5]. But instead of shaping voltage or signal, Context determines events in time.

Context does resemble other sequencers built in PD, especially Xeq[6]. The main differences here seem to be the Graphical User Interface (GUI) and the fact that Context is a patch, not an external. While this comes at a cost to the CPU, it has the advantage that it is more versatile and accessible to the user.

3 Description of the Context sequencer

When Context is started (double clicked), the cursor moves across the canvas and plays the given pattern.

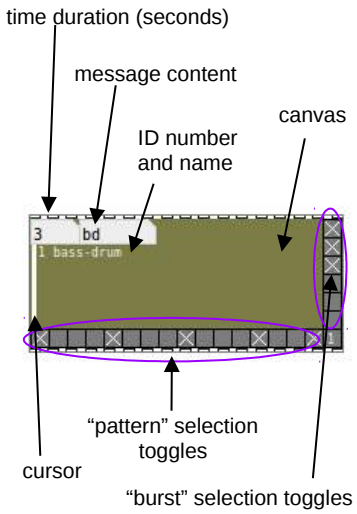


Figure 1: Anatomy of Context

- **Time duration** determines the length of time that Context takes to complete one sequence (i.e. how long the cursor takes to move across the canvas).
- **"Pattern" toggles** determine the pattern or rhythm that is played.
- **Message** determines the message(s) that are sent by the pattern.
- **"Burst" toggles** determine what happens after the pattern is finished.
- **Inlets** are all identical. When an inlet receives a message, it will start the Context cycle.
- **Outlets** are synched with the "pattern" toggles, so an outlet will send a message when the cursor moves over a selected toggle.

The distinction between the "pattern" and the "burst" is central to Context's operation. The pattern interprets its selection temporally (a *then* b *then* c), resulting in a fixed rhythm or melody played out through time. The burst interprets its selection logically (a *and/or* b *and/or* c), making a decision about what happens once the Context cycle is complete. As the pattern and the burst are always present, each Context contains a temporal and a logical component, allowing the user to define a musical phrase and consider its consequences at the same time.

Both axes are click-draggable, so the pattern and burst arrays can come in any length, and Context can occupy any amount of space on the canvas. Unfortunately, PD only allows for outlets on the bottom of an object and not on the side. To access outlets from the "burst" toggles, the user can flip the x- and y- axis; the pattern will then

---

2 CEMA, Monash Universtiy, 2007-2013: http://www.csse.monash.edu.au/~cema/nodal

3 Intermorphic, 2007-2016: https://intermorphic.com/noatikl

4 Notably Blokdust: Twyman, Philips & Silverton 2016: www.blokdust.com.

5 See "overlay hacking", Context documentation (forthcoming)

6 Czaja, "Time and Structure in Xeq", 2004: http://puredata.info/community/conventions/convention04/lectures/tk_czaja

play from top to bottom, and the outlets will fire simultaneously at the end of the Context cycle.[7]

4 Events in Context

It has been said that Context sequences events in time, but specifically what events? Context does not discriminate between different types of events it schedules. It sends PD messages[8] which can be interpreted in any way. This point sounds trivial, but is important for what follows. Some of the events that Context could schedule are:

1. **A musical note** (i.e. "C#) or rhythmic beat in another PD patch
2. **A note or beat in another program**, through MIDI or OSC
3. **A modulation event** (i.e. "filter to 1000 Hz")
4. **An arbitrary instruction** (i.e. "launch the rocket")
5. **An instruction to start another Context, or restart itself**
6. **An instruction to change another Context in any way** (i.e. "open toggle #1" ; "change the delay time to 2 seconds")

Numbers 5 and 6 are of special significance for the next sections of this paper.

4 Sequencing musical structure

I define musical structure as the pattern or alteration of musical or rhythmic phrases through time. So for instance a simple rhythmic structure might be:

```
        A A A B
where A = 1 x 1 x
and   B = 1 1 1 x
```

Figure 2: In this notation, 1 = one quarter note and x = one quarter note rest.

Structure is extremely important in music, but curiously under resourced in most DAW environments[9]. A DAW timeline gives a unique timecode to every event in a composition, down to the minutest details, but this omniscience comes at a cost: what if the composer wishes to leave some event undefined or implied? Structure cannot be depicted in such an environment: if the user decides to make a structural change to the composition, she must carry out every alteration by hand, analogous to calculating 2+2+2+2+2+2+2+2 instead of 2*8. This problem is eased somewhat by "storage banks" programmed into most step sequencers—the user can schedule the events A and B to signal the switching from one pattern to another. But these storage banks have their limitations. For instance, what if the composer wishes to design a second order structure, such as:

```
          C C C D
where     C = E F F F
          D = F E E F
          E = 1 x 1 x
and       F = 1 1 1 x
```

Figure 3

To capture this structure, you would need a storage bank of storage banks. Some sequencers offer such a thing[10], but then the question arises, what if you want a third order structure? and so on. Clearly the limit of a DAW will soon be reached.

Since Context does not discriminate between the types of events that it schedules, it can make light work of complex, multi-layered structures such as these. Musical structure can be determined by the step sequencer with the same ease as a rhythmic pattern. The following diagram makes this clear:

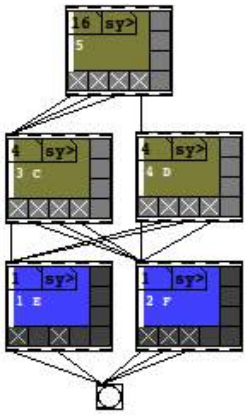


Figure 4: A context network representing the structure described in Figure 3. To see the structure, follow the connections: the first Context fires three times to C and once to D. C and D are

---

7 This comes across more readily on screen than on paper, but recognizing the difference between a normal and a flipped Context will help the reader interpret the diagrams in this paper. A flipped Context has its cursor on the top, as in figure 6, rather than on the left, as in figure 1. The timeline is then read top to bottom, rather than left to right.

8 Lists, symbols or floats

9 For example, the Recording and Sequencing suite in Propellerheads Reason: https://www.propellerheads.se/reason/recording

10 Ie. Reason's Matrix Pattern Sequencer: http://www.soundonsound.com/techniques/reasons-matrix-pattern-sequencer

similarly connected, to E and F, whose toggles represent quarter notes and quarter note rests.

Here we can see that the structure of the Context network reflects the structure of the music itself. The network is arranged in a hierarchy, with each level dictating what happens below. But hierarchy is not the only way of conceiving of musical structure. Consider the following structure:

```
          A B
where     A = 1 x 1 x
and       B = C or D or E
          C = 1 1 1 x
          D = 1 1 X 1
and       E = 1 X X X
```

Figure 5

What is required here is a way of depicting a set of parallel events. This is what Context's "burst" toggles are used for. The burst toggles all fire simultaneously at the end of the Context cycle, but specifically which toggles fire can vary. The following Context network satisfies the structure described in figure 5, with the "burst" toggles from A being used to select from a parallel series C,D,E.[11]

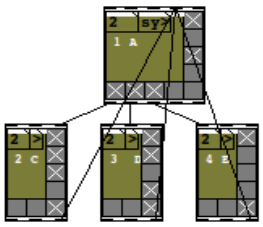


Figure 6: A Context network representing a parallel structure. Here, the "burst" toggles are on the x-axis and the "pattern" toggles are on the y-axis, so the timelines progress from top to bottom rather than left to right. The three open toggles in the "burst" section of Context 1 function as a selector, choosing first one then the next toggle once its cycle is complete.

With the techniques laid out in these two examples, Context can be used to create multi-layered musical structures with any degree of complexity. Hierarchies and parallel sets can be built layer upon layer and interconnected with each other in any way. Because Context can schedule structural events just as easily as musical events, the sequencer is capable of representing musical structure *per se*, where normal timeline environments leave it to be inferred.

[11] How the burst selects which toggles fire will be covered in the next section.

What is the point of representing music in structural terms? One practical reason is power and control. A tool that allows us to represent musical structure will save time and effort in the same way that a repeat sign or coda saves paper. This becomes most apparent when the composer decides to change something: a single alteration at a higher level can determine hundreds of other events, which would otherwise have had to be redefined individually[12]. Saving time is of course what computers do best, and so structural representation allows the user to focus on creative, rather than operational, decisions.

5 Randomness

Thinking about music in structural terms is enlightening, but it can also become overly deterministic. The brilliance of music often comes across more in spontaneity than in structure, and this is a feature which is difficult, if not impossible, to program. Lacking the judgment of the musician, a computer must rely on pseudo random number generation (PNG) to simulate spontaneity. It is very easy to program a computer to play random notes at random time intervals, but the result is no more musical than a Geiger counter. To generate non-deterministic music, it is necessary to find some way of containing the output of PRG's to specific decisions, times and ranges, and for the user to be able to override or modify a random decision made by the computer if it is not to her liking.

Context is designed with generative music in mind, and it is capable of using PRG's in many different ways. Because the composition is represented in the form of a Context network, random decisions are localized. So, for instance, the user can determine that the third note of a particular melody should be random, with the same ease that she would determine that it should be C#. Bearing in mind that Context can represent structural as well as musical events, it follows that the user can incorporate randomness at any level of the composition.

There are three different pseudo-random

[12] At its mathematical limit, structural representation has an exponential effect: in a structural hierarchy such as the one described in figure 4, an alteration at the nth level can affect x^n other events (assuming that all Contexts have a pattern size of x).

generators built into Context, each with its own attributes and characters:

1. Random delay time
2. Random messages
3. Random burst

Random delay time affects the time cycle for an individual Context, and hence the speed with which its step sequencer plays. It is defined by a range, so that the allowed intervals will be no bigger than e.g. 10 seconds, and by a resolution, so that the intervals will come in denominations of e.g. half or whole seconds. The resolution can also be defined exponentially, so that the allowed intervals are 2^n up until the range limit, where n is the resolution. This has the advantage of filtering out non musical time intervals, so that a Context cycle could last a whole, half, quarter or eighth note, but nothing in between.

Random messages allow one or more random terms to be added to the main send message, useful for determining musical notes and other parameters. Context messages can be "solved" with more or less the same ability as a scientific calculator, so for instance a message "n (2+4)/3" will simplify to "n 2", and "m (?10 + 100)" would simplify to "m 109" if the random term ? 10 yields its maximum 9. All simple arithmetic can be integrated into random note generation, expanding the user's options for harnessing chaotic number streams.

Random burst determines which of the burst's toggles are selected on the completion of a Context sequence. This is useful for decision making and structural randomness. The burst uses Gaussian distribution to select a toggle, so that the user can determine the likelihood of a particular toggle firing. The random selection is then summed with a deterministic progression, so that 1 (or any number) is added to the selected toggle number successively. The same set of parameters is available for determining how many toggles are selected, as well as which ones. The result is a diverse palate of options for random decision making, ranging from linear sequences (A then B then C), to high & low probability sets (probably A but maybe B), to completely random scenarios (one, some or all).

In summary, the prospects for generative music are very rich in Context. Random timings can be made in congruence with standard musical intervals, random notes can be manipulated with arithmetic, and random decisions can be taken according to a probability distribution. To recall the previous section, it is easy to see how randomness can be incorporated into musical structure. Events at any level of structure can be decided by the burst, and a Context network can randomly choose between an A and a B section just as easily as it could randomly choose to play the note C#.

But the lines between rhythm and structure are not always clear. Figure 7 depicts a simple pattern in Context 6, but is it a rhythm or a structure? Instead of being played directly, the beats are passed on to the blue Context which acts as a logic gate, choosing between three further options: an eighth note, two sixteenth notes, and a rest (notice the third open toggle at the end). The pattern depicted by the first Context then is a sort of a "meta-rhythm", an outline of a rhythm which is then interpreted and played by other Contexts in the network. This is the sort of non-deterministic music that becomes possible with Context's application of pseudo random generators.

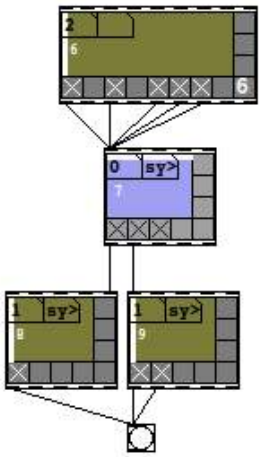


Figure 7: a structure involving random elements.

6 Internal Messaging

All Context parameters, including time, toggle selection, and dimension, can be set manually through the GUI, or automatically through Context's internal messaging system. For example, to open toggle number 3 on the x axis, the user can either click on it, or send a message to that Context " :x 3 ". Since Context does not discriminate between the types of events it schedules, it can also send such a message, giving Context the ability to alter its own settings. For instance, the message " :x 3 " could be sent from one Context to another, or even to itself, instructing it to open its third outlet. Opening and closing outlet toggles is the most obvious application, but every Context parameter can be accessed in the same way. I call this internal messaging.

The consequence of internal messaging is that a Context network can evolve over time. This is more potent than random decision making—consider the difference between a junction that offers you a set of alternative paths, versus a junction that offers you the tools to pave new road. The possibilities for self-evolving compositions include:

- A melody or rhythm which changes at a set point, i.e. C# becomes E.
- The modification of a structural event, i.e. “go to section A” becomes “go to section B”.
- The alteration of the Context cycle, i.e. the duration doubles.
- Conditional changes, such as “the first time that a C# is played, this outlet will open” or “every 5th time that this note plays, there is a 20% chance that it will be randomly changed to another note”

The way in which a network evolves is also open to definition. Internal messages can be sent by a dedicated Context timer which sends one ie. every 30 seconds, or they can be woven into the Context network together with notes and other events. Remembering that Context messages can also incorporate random numbers, we also have the possibility for Context networks to evolve in random ways. Randomness can be directed towards a specific parameter (i.e. “a random toggle on this Context will open”) or towards the Context ID numbers (i.e. “a random Context will have its third toggle opened”).

A Context network that evolves at random has some practical limitations: the law of Entropy dictates that over time the system will tend from an ordered to a disordered state. Intervention is necessary if the composer wants to achieve something purposeful and avoid musical degeneration. Since internal messages are interpreted the same as user defined events, the process of vetting can be carried out in the GUI in the same way as normal Context design. There is also a separate “undo” abstraction which keeps a log of all changes made to the network and reverses them on demand. So the user still has control over the Context network, even if it is moving in a chaotic way.

7 Recording

Context can record melodies and rhythms as well as playing them back. The record function saves notes or data to the message box, and a pattern to the pattern toggles, according to the timings in which they are received. Thus, what Context records is information, not sound.[13]

The recorded information can come from the object’s inlets, or from a send-receive channel. This allows one Context to record from others in the network, as defined by their ID tags. The duration of the recording is predefined by the Context cycle time. Because the Context has only a finite, usually small, number of outlet toggles, a kind of resolution is imposed, whereby the Context can record no more events than there are outlets.

[13] A future release of Context will feature sound recording as well.

There are two immediate implications for recording. First, a melody or rhythm can be played into a Context from an external source, i.e. a midi keyboard. This is a useful way of interfacing with Context and speeding up composition. The second is that Context can sample itself. A pattern or melody that is generated from elsewhere in the network, perhaps randomly, can be recorded and folded back into the composition. The record command can either be executed by the user, or automatically by the computer through internal messages.

This has further implications for the way that Context networks evolve. Through internal messaging, they are likely to evolve slowly and uniformly, but recording has the potential of making the process more discrete. One Context might house a melody or rhythm which functions as a base for a larger section of the network, as in Figure 7. The melody evolves somewhat but never deviates far from the base, until the command is sent to re-record it. Then a new base is formed, and the process repeats.

7 Using the Context Canvas

The Context canvas is the coloured area that lies between the message boxes and the pattern toggles. It is not wasted space: the canvas functions as an embeddable timeline for linear playback. “Content”, a modified PD array object, allows the user to place arrays on the timeline, turning Context into a sample player. The Context sampler has most of the features that one would expect from a basic sampler: the user can loop, slice, speed up and reverse the sample through simple GUI gestures (or internal messages). The sampler can also relate to the sequence messaging system: with a tilde (~) character, a Context message will take a snapshot of the sample, making Content ideal for custom modulations.

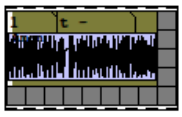


Figure 8: an embedded Content sample

It is also possible to embed one Context within the canvas of another; while the parent cursor hovers over it, the embedded Context is “on” and cycles continuously. Any number of objects can be embedded within one Context, and embedded objects can easily and accurately be moved around the canvas with a special drag-and-drop tool.

Thus, Context does not force the user to abandon the global-linear paradigm that has become so ubiquitous in music software, where each event occupies a point on a unified timeline. But rather than being conditions of the environment, globalism and linearity are choices that the user makes in designing a Context network. The user might reject altogether the localized and random approaches suggested in this paper and instead use one Context canvas as an environment to structure a whole composition, embedding and arranging samples and other PD instruments on the timeline. Such design choices are not restrictive: the user can easily create many such environments and have them interact as part of a larger network. In short, Context is capable of functioning as a universe as well as an atom. As an atom it can build more complicated structures, and as a universe it can host other objects for linear playback.

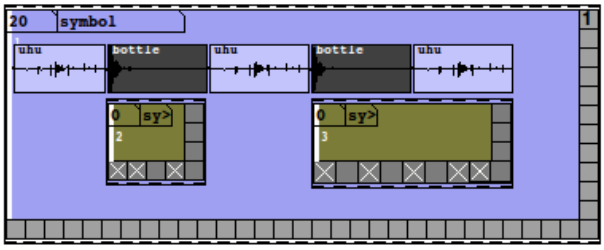


Figure 9: One Context acting as host to other embedded objects

8 Conclusion

What kind of music can be made on Context? As its designer, my main motivation is to see somebody else using Context to create music in a way that I hadn't even imagined. Context offers the user the opportunity of designing and shaping her own sequencing environment, rather than inheriting one in pre-packaged form. As such, Context can be whatever you want it to be, from simple to complex, deterministic to random, linear to non-linear. The limitations of the software remain to be found, as they surely will be, but for now the designing and exploring of Context networks promises to open many new horizons for generative and performance musicians.

A Context network can become a hive of activity, with events of all different sorts triggering each other and cascading through feedback loops like a giant, chaotic marble run. The resulting sounds might not resemble any conventional composition, and indeed it has been suggested that "composition" is not the right word for it at all. A Context network is like a musical score in that it has been designed by an artist to represent a piece of music, but this might guarantee nothing more than a starting point. The way in which the music progresses from there might be thought of as a sort of dialogue between performer and computer, with both parties free to make changes to the network. The computer takes some of these decisions at random, to which the user acts as a supervisor, accepting, canceling or modifying the computer's actions as she decides.

Context invites other forms of collaboration as well. The Context network is inherently de-centralized, with no part possessing absolute control over another. Because of this, it would be possible to design a Context network to be operated by two or more performers. This could be done in an organized way: one user having control over percussion while the other has control over melody. Or it could be done in a disorganized way, where both participants have access to arbitrary, undefined parts of the network, the resulting music being a surprise to them both. As Context networks can run themselves with a high degree of automation, the performer is also free to respond in other ways, for instance by playing a "real" instrument in conjunction with the Context composition.

To finish with a metaphor, I would hope that a Context network has something in common with a garden. The seeds originally sewn are not the same as the plants which grow to envelope their surroundings, and the gardener must constantly cultivate, prune and train her work, which is bursting with its own energy and will to return to nature. There is no final product, only the continual confrontation between an artist's intentions and forces that are beyond her control.

9 Acknowledgements

My thanks goes to the PDCon16~ organisation team, and to many people in the online PD community who have helped me with this project. I am also deeply indebted to Tristan Chambers for introducing me to the PD platform and inspiring me to make my own music software.

Context Version 2 is forthcoming at www.contextsequencer.wordpress.com, together with full documentation.

References

[1] CEMA, Monash Universtiy, 2007-2013, Nodal: http://www.csse.monash.edu.au/~cema/nodal

[2] Intermorphic, 2007-2016, Noatikl; https://intermorphic.com/noatikl

[3] Twyman, Philips & Silverton, 2016, Blokdust; www.blokdust.com

[4] Czaja, "Time and Structure in Xeq", 2004: http://puredata.info/community/conventions/convention0 4/lectures/tk_czaja

[5] Propellerheads, Reason, 1994-2016: https://www.propellerheads.se/reason

[6] Price, Sound on Sound, “Reason’s Matrix Pattern Sequencer”, 2007: http://www.soundonsound.com/techniques/reasons-matrix-pattern-sequencer

[7] Goodacre, 2016, Context: www.contextsequencer.wordpress.com.

Finding Balance: Introducing Art and Media Students to Programming with Puredata (Pd)

Mark Edward Grimm
Visual and Performing Arts
Syracuse University
Syracuse NY, USA
megrimm@gmail.com

Abstract

Students without prior knowledge in computer programming were exposed to coding for art and media based applications using the visual dataflow programing language Puredata (Pd). Single semester introductory teaching was conducted over the course of several years, in three separate higher-educational settings, and in four distinct departmental disciplines. This paper traces the development of a pedagogical approach to teaching beginner Studio/Media Art students Pd and is useful for those looking to introduce Pd to students, add Pd to the curriculum, or teach Pd in a Studio Art setting. Additionally, variations in teaching style, observed successes and failures, classroom settings, coding examples, and outcomes are also addressed.

Keywords

puredata, makey makey, studio art, pedagogy, art-based educational research

1 Introduction

Incoming freshmen with no prior experience or knowledge of coding, physical computing or interactive arts were typical of enrolled students. Students tended to fall into the following categories: (1) students wishing to pursue a career in design: fashion design, industrial design and graphic design (2) students wishing to pursue a career in media arts: film, photography, and computer animation (3) students enrolled in a general "new media" degree and (4) students working in the fine arts such as painting, drawing and sculpture.

Several unique factors contributed to curriculum development and experimentation with tools and course format. Teaching conducted at Cornell University, Syracuse University, and Tompkins Cortland Community College, variably between 2008 and present, allowed failures to be excluded and successes to be improved upon. Method refinement was consistent as data was collected and reacted upon leading to new teaching methods and new data, etc. Through this process a general educational practice emerged.

Prior to introducing Puredata (Pd) into the curriculum, no comparable departmental instruction for first and second year students was in practice. Courses generally allowed freedom to either develop curriculum from scratch or embed personal interests and expertise into the syllabus. Also, opportunities to modify established educational structures were presented. Choosing Pd as programming language and the Makey Makey physical computing boards [1] created a powerful entryway, as observed, for students to learn interface design, interactivity, and utilize real-world data to control custom Pd computer applications.

When I started teaching art in higher-education in the mid-2000's I found a lack of understanding on how and what type of computer based tools could and should be taught for studio art practitioners. Much studio instruction, at least locally in Upstate New York, relied on Adobe Suite based applications; continuing to this day in many cases. It was my contention that departmental or institutionally condoned software was hindering creativity rather that enhancing it. Students typically learned linear editing (Video Editing Software) and static image creation (Photoshop, etc). Programming environments for creating interactive applications for visuals and sound were not being utilized or introduced to students as an option for creative output. This could have been one of several reasons: (1) instructors did not know how to use these tools therefore felt uncomfortable teaching them, (2) support infrastructure was considerably lacking for instructors to learn these tools, (3) instructors own personal practice was not defined by interactive electronic art therefore they had no incentive to learn or instruct and (4) entry cost using similar proprietary software/hardware was too high to risk failure.

Recognizing the need for students to explore new ways for art creation I asked myself the following questions:

Given that no established pedagogy exists in the current educational environment to teach students how interactive artwork can be created, how can ways of creating be introduced into the curriculum?

Given that new tools were necessary to educate students in new ways of thinking and creating artwork, what tools should be used as an introduction point for new art students to create dynamic and interactive based artworks and how should these tools be used?

2 Teaching Methodology

There are many reasons to teach Pd to beginner art students in any specialization. Students of art and design can benefit independently and professionally by understanding how basic programming works and knowing how to build interactivity into their projects. Additionally, learning Pd helps students to grasp the 'guts' of what is 'under the hood' of their favorite design, image, audio and music making software. With this working knowledge students shed the mysticism of electronic media production having the ability to control and alter electronic work; they become media producers rather than consumers.

There are many ways that Pd can be taught but I have found through actively using, altering existing patches, programming from scratch, and becoming involved in the open-source community students form strong learning connections. Making things rather than looking at things or as Ryle [2] states, "knowing-how" rather than "knowing-that", builds foundational knowledge that leads to more inquiry.

There is a complex process to teaching and learning new things, technological things in particular. The learning process is often slow, patient, repetitive and reflective in its engagement. Learning is knowing when to get help and knowing when to help others. In addition, it is research and practice, shedding fear and becoming comfortable with experimentation. Learning is also idiosyncratically improvisatory across many educational research architectures [3] meaning instruction must give and take, push and pull, go with the flow, and be reactive to new data in novel and not immediately apparent ways.

Often students are told when asking trivial questions: "I don't know, try it! See what happens!". Students benefit when taught the skills of troubleshooting and creating through trial and error as understood through an Art-Based Educational Research (ABER) paradigm; which in essence is problem based learning [4]. They should acquire the understanding that there is no easy answer in creating interesting work and should be aware of the 'learning-to-teach-oneself' strategy. The student's goal is to imagine and create exciting new things that others have never seen prior. These are projects that will get people to stop and take notice rather than disregard. Due to general media saturation, students must work to stand out in novel, conceptual, and interesting ways with teaching having its own artistry as a model of inquiry [4].

Pd is a tool. it is a powerful tool to achieve an artistic goal, one preconceived in the imagination. While teaching Pd often an analogy is made to the class: there are many ways to drive a nail into a board, the tool does not really matter. The tool we just happen to use is Pd because (1) its free/open-source (2) artists are visual learners and respond well to it (3) it is portable and can be used in most computing situations and (4) it is minimalistic in its design and implementation.

2.1 Visual Programming

Enrolling in art-based disciplines, students often come prepared with enhanced visual thinking skills. Comparatively, they excel in constructing ideas in images, shapes, objects and color as opposed to the use of plain text. Prepared to strengthen visual literacy and artistic skills, students often exhibit anxiety when initial presentation of the course syllabus states that computer programming will be a requirement. As observed, the term 'programing' often contains the preconceived notion that coding is character-based 'text'. There is some relief when the visual-programing interface of Pd is presented through initial 'my-first-patch' creation. Students quickly realize visual cognitive skills have benefit [4] when Pd as a programming language coincides with the ability to learn.

Through the audio equivalent of 'hello-world' students often become empowered and excited to continue the learning process when potential for creating visual and sound oriented projects becomes clear. This sense of possibility is also amplified by the realization that programming a computer for interactive art is attainable, rather than futile, for the art practitioner.

2.2 Open-Source Philosophy

Students become aware early on that instruction leans heavily on open-source software. Often this comes as a happy surprise realizing they would not have to insure any addition cost with proprietary software because of monetary/budgetary concerns. having first time exposure to Pd as a programing environment means that they are unaware of commercial proprietary options.

Open-source is also a philosophy and is not just about being 'free'. It is a social development method. When using Pd, users can choose to become part of a community to make the software better, to contribute to its development and to evolve it in a forward direction. Open-source in this sense an abstract machine, evolving and changing over time as it passes through many ears and eyes.[7]

2.3 Software as Tool

Software, regarded as a 'means to an end', is a tool utilized in the realization of an idea. Students, therefore, are encouraged to conceive of concrete ideas for future development. Emboldened to 'think big' minus the additional weight of required technical proficiencies, student idealize an end goal that can, through the artistic process, be scaled down or altered as needed. Constructing ideas limitlessly imagined frees themselves from an impending box of constraints.

Translation from idea to project requires tools and effective skills in utilizing those tools. Lecture often dictates that tool mastery is not required; rather the tool is used to materialize what was not previously material. Often this process comes in the form of mapping solutions in sentence form through journaling. A student journal entry for example: if someone walks in front of the camera the camera recognizes that someone has walked in front of it, then sound is played.

If feasible and appropriate classroom conditions are provided [8], the student must translate this idea to program form, working diligently towards its construction. Instructing through analogy can make connections to the material world as vivid illustrations. For example: First one envisions the house to construct, then illustrates through architectural drawings, sketches, etc., then one translates to a physical construction though the use of tools and assembly process.

Often student proposals will be unrealistic for many reasons: (1) Time allotted to realize an idea is beyond the scope of the class, (2) They lack resources (monetary, material, etc), (3) They lack skills as individuals or in group project situations and (4) a combination of the three. Instruction, under no delusion, gives ample warning that proposal realization could potentially be problematic in preparation for student disappointment. Without compromising the integrity of their original "big" idea, proposals are encouraged to "scale down" and to meet intended goals utilizing maximum allotted resources. Encouraging future possible expansion or completion, final projects are often in the form of a project demonstration or prototype.

3 Practice

In practice, teaching Pd is dependent on a wide array of variables. Instructors must know their working environment and be prepared to address technical problems when they arise. Difficulties and problems can sometimes lead to solutions but can also lead to more problems. Building from a simple foundation has been successful and through repetition, slowly adding new concepts to the repertoire, students tend to retain knowledge.

3.1 First Attempts

Initial instruction began in 2008 in an experimental manner at Syracuse University teaching a course called "Time Arts" for incoming freshmen mainly interested in pursuing a career in design (fashion, industrial, graphic, etc). Outfit with (20) 2007 Apple eMacs, Pd was installed lab wide via Apple Remote Desktop. In addition, Roman Haefeli's original netpd (v1)[9] was installed on the desktop allowing users to create real-time collaborative music with each other. A custom server was run on the main teaching station and students were instructed to open the patch which connected to the server. Students were then given modest instruction and allowed to "play" music with each other. Students enjoyed the experience but quickly lost interest. Further instruction in Pd for the remainder of the course was abandoned.

Subsequent courses over the next few years continued with further one-day "workshop style" experimentation. In the beginning these classes were treated more like a 'fun' day for the students, occurring mid-semester as a break from our primary focus (audio/video creation and editing). Lacking official assignment status, students were initially intrigued but quickly became confused and disinterested in continuing. Independent learning was abandoned save from one or two students per class that often, not surprisingly, changed majors from design to fine art (painting, sculpture,etc).

For instruction simple patch examples were constructed on the screen as the students emulated or copied the examples which were devoid of variation from each other. Lacking a clear purpose or a goal to work towards contributed, in hindsight, to overall failure.

Through these classroom experimentations it emerged that simple patches worked best and if students could not get immediate sound from what they were making attention would be lost. Starting with a simple [osc~] -> [dac~] eventually became the starting point for beginning instruction with the understanding that if students can immediately be excited they will want to continue and if they become confused and overwhelmed their interest will quickly wain.

3.2 Evolving Style

Eventually by 2011 classroom experimentations with Pd became full assignments. Instruction methods were still rapidly changing from semester-to-semester and even course-to-course when multiple sections were being taught. Having a clear defined goal for students to work towards greatly improved their interest and sometimes their dislike for Pd simultaneously. Tutorials continued to improve, students were issued documentation (Pd cheat sheets, tutorials, etc), and assignments were being semi-competently completed.

Assignments given were simple and coincided with our unit on "sound". Students were instructed to create a sample library using field recorders. They would choose '4' subjects and record '10' variations on each subject of their choosing resulting in '40+' sounds for each student edited and exported from Audacity and organized into folders. This assignment had similarities to the previous assignment that dealt specifically with video so they were versed in the 'library' creation idea.

After renaming files sequentially (1.wav, 2.wav, etc) students programed a Pd patch using knowledge retained from classroom Pd tutorials. The goal of the patch was to randomly play their samples back thereby creating some kind of endless abstract soundscape based on the [readsf~] object and controlled by [key].

3.3 Learning Curves

The learning curve for beginner students can be quite steep. Students with limited general computing experience tend to struggle with even the most basic patching exercises. This includes saving files, knowing location of saved files, basic folder organization, basic OS navigation, etc. Compounding this problem are institutional barriers addressed in the next section.

Excluding these exterior issues, but also partially because of them, students often grapple with simple tasks within the Pd environment itself. Students first reaction to the "console" often is the inclination to maximize it as though that was the program they would be working. Instruction therefore dictate that this is the area that error messages will be displayed and other important information. They are advised to keep the window visible while patching to note any errors that might occur in 'red'. Students seldom keep the console visible often opting to minimize it whereby complicating troubleshooting during patching instruction.

During patching exercises other issues begin to emerge. Most importantly the beginner will have a difficult time understanding the difference between an object and a message. Object names are sometimes typed into messages and message names are sometimes typed into object boxes. To minimize errors and maximize success at understand differences between the two it has become imperative to go over the differences multiple times. Taking notes in the form of patch 'comments' has also become effective, again repetition is key throughout each newly created patch. Another effective solution to this problem, drawing on the 'visual learning' literacy rate for art students, is to highlight the visual differences (object backs are flat and message backs have a tail), again, repetitively.

Often analogies to analog equipment are made to varying effect from class to class. For classes that have interests in music, especially those that play electric instruments, the illustration of an 'instrument to cable to effects to amplifier' is fairly effective in describing the patching of one object to the next with patch cables and the idea of messages as knobs for effect or amplifier control. For those with stereo or DJ equipment experience, similar analogies can be made. For the simple [osc~] -> [dac~] example a Smartphone to headphones through a cable comparison is made. It is much easier and effective to make comparisons when student interests are known and much harder when they are not. Detailed student introductions in the beginning of a course can be very useful in gathering this kind of information but do not always yield a universal way to teach maximum student comprehension and retention.

Struggles also occur in the following other areas: (1) audio problems such as no audio or audio

fragmentation, (2) knowing the differences between 'edit mode' and 'performance mode', (3) losing windows, and (4) the inability to troubleshoot 'red-dotted boxes'.

After initial introductions and tests students are often left feeling overwhelmed and confused. Class teaching therefore focuses on repetition. Creating the same patch over and over again, slowing adding one more object per iteration. Saving each file in the same place using a simple versioning system (01-pd-tut.pd, 02-pd-tut.pd, etc) allows students to go over what they have made. Verbosely commenting on each object and message throughout each patch encourages retention. Consequently, often at the end of a four hour studio lesson, repetitive instruction of this magnitude yields a very sore throat!

3.4 What Works

Classroom teaching to beginner level students has become formulaic within the last several years being successfully applied to design, film and photography, new media and high-schoolers interested in media studies. This applies to multiple institutions as well as lab environments. The addition of the Makey Makey boards after initial tutorial instruction has greatly improved student interest in creating with Pd by offering them a simple way to quickly prototype ideas for control in the physical world. In addition, I have found using the GEM (Graphics Environment for Multimedia) library to control video samples and basic visuals is more concrete for students of the arts to understand. Many art students are disinterested in the abstract level of sound and sound creation and find creating visuals to be much more appealing.

3.4.1 First Steps

Class begins with opening Pd, describing the console, and then creating a new patch. The patches and all subsequent patches are saved using a numbering system to a 'tutorials' folder.

To elicit immediate excitement the first patch is a simple sound patch using [osc~] -> [dac~]. 'Objects' are described as simple, miniature, independent programs that perform a specific task. In this case the [osc~] produces a simple sound wave and is an 'audio object' defined by the tilde. The [dac~], or digital to analog converter, represents a virtual instance of headphones or computer speakers. A number box is attached to the left [osc~] inlet and is described as a message that can be controlled. The 'DSP' toggle on the console is then instructed to be checked.

At this point class instruction switches to the differences between 'edit mode' and 'performance mode' whereby students can begin to interact with the patch by dragging the number box attached to the oscillator. If no sound is heard troubleshooting begins.

This first patch also doubles as an audio test leading to 'Audio Settings' discussion, OS specific output instructions, 'Test Audio and Midi…' patch, etc. This testing phase can be problematic depending on the situations detailed in the preceding section. Students are often encouraged to assist each other to effectively decrease classroom troubleshooting time.

If all goes well a sound is heard and student excitement and interest are positively elevated.

3.4.2 Moving Forward

Subsequent objects are introduced in the following order:

[random] with attached [bng] to control oscillator. Additionally 'creation arguments' are discussed with [random 100] as well as the concept of 'control objects' as opposed to 'audio objects'.

[+] for scaling [random]'s numerical output; often with [+ 100]. Basic operators are addressed here as well as cutting patch cords for new object insertion into the chain or 'top-down' dataflow.

[metro] is introduced as [metro 500] controlled by [1(and [0(messages. Analogies are drawn to an on/off light switch.

Number boxes are introduced to right inlets for adjusting output.

At this point students are often left to "play" with copy/pasting, adding more [osc~], operator objects, and [random] objects

3.4.3 Audio Samplers

Teaching is not geared toward building complex synthesis so the simple [readsf~] is sufficient for reading the audio samples they have created through recording and editing.

After a 'message box' lecture on the use of a comma for playing a sound by clicking on one message instead of two, [key] is then introduced. This is coupled with Makey Makey instruction that focuses on building 'buttons' using a variety of techniques such as drawing buttons with graphite on paper, making buttons from metal/paper, and other materials that are electrically conducive such as water.

The [key] object is then used to get students to

play their samples with their self-made buttons and create some sort of DIY instrument either individually or collaboratively.

3.4.4 Video Samplers

Depending on the department/institution, tutorials switch to GEM with students building simple video players, again to control their previously created video samples with the Makeys Makeys often in conjunction with sound. Students are often more interested in creating visuals that audio and are excited to be using video interactively.

Simple GEM concepts such as the render chain and creating a [gemwin] are taught. [pix_film] is used to loop video and simple counters are created using [f]x[+ 1] to control video by frame. At this point 'pix-based' effects are introduced and students are instructed, using the help browser, to experiment by patching effects between [pix_film] and [pix_texture]. Instruction also adresses [rotateXYZ], [translateXYZ] and 2D primitives such as [square] and [rectangle].

3.5 Project Assessment

Tutorials evolve into final project proposals and then eventually final projects. Once students understand what the Pd/Makey Makey combo can do, this process is quick to materialize and eventually complete. Final projects are assessed at several different levels: (1) project proposal, (2) first demonstration, (3) final project demonstration, (4) exhibition and (5) final documentation usually in video form.

Success is determined through a balance of conceptual merit, design aesthetic and technical competency. Accomplished projects are often technically the simplest and conceptually the most interesting. Full project completion can be problematic but forgiven if prototype and/or demonstration are proficient. Determining work ethic for Pd programming compares originality in variation to tutorial content. Additionally, student initiative to gain additional help after/before class, during office hours, or through email is rewarded.

When troubleshooting, students are encouraged to search for solutions on the internet, join the Pd forum, mailing list or the Facebook group. Disregarding this advice often leads to troubled final projects and is reflected in grades. Instructing students on the benefits of independent research and having them proficiently conduct such research has been problematic and requires stronger methods.

Small 'build up' tutorials geared toward 'learning to program' are graded in the context of 'assignments'. Satisfactory completion leans toward 'skill' rather than project/creativity. Unsatisfactory completion results from missed homework and not following along in class. Ironically unaware that, even in design majors, they will eventually tackle Arduino boards, Processing, etc., noticeable disinterest in participation occurs when students feel they will "never have to do this again".

4 Challenges

When learning Pd, students are often left with uneven levels of understanding. Many factors contribute to this in addition to instructor deficiencies. Institutional challenges stem from diversity in classroom environments, difficult administration, student preparation and interest, and problems with Pd itself.

4.1 Institutional Diversity

Student equipment is dependent on many factors including university policy, personal preference and social/economic background. As of 2010, Syracuse University's School of Visual and Performing Arts has all but eradicated institutional computer maintained labs in favor of an incoming freshmen laptop requirement. This requirement, marketed "strongly recommended," states a minimum Apple 15" MacBook Pro and Adobe Creative Cloud for students majoring in Art photography, Art Film, Art Video and Computer Art [8]. This requirement has its benefits and weaknesses: (1) Students are responsible for their own computer maintenance which can be beneficial forcing students to understand their systems and more difficult in learning and maintaining them, (2) students can easily install software such as Pd without the need for asking for administrative privileges and (3) students have their systems wherever they go and can work in the environment of their choice.

This policy creates a Mac laptop majority, granted the one or two students per class with Windows systems. With the understanding Windows users will lack quality instructional support due to instructor unfamiliarity, students often opt to get a Mac during the semester as they envy their peers.

On the other hand university 'lab' computers are often Windows based systems such as at Cornell University and Tompkins Cortland Community College. From experience, IT administration has

quickly and quietly catered to instructor needs, but as those in academic situations can attest, this is not always the case and IT, administration, and other faculty members can sometimes be particularly difficult to communicate and interact with on a professional level, etc

Teaching Pd to aspiring artists in a higher-educational setting can be difficult as well as highly rewarding. Institutional support can be very poor in certain situations so it is up to the instructor to make the argument for the benefits of teaching Pd and to make the installation, maintenance, and support to be as easy as possible. I have also found that once Pd has been taught a few times and student work has been effectively demonstrated, the case for use is much clearer to understand.

4.2 Student Preparation

Generally, students come prepared with a large variation in skills. This can be difficult as well as challenging. It is within the first year of higher-education that skill equalization occurs. Student work ethic can quickly close gaps between those that do/do-not come prepared. Within the first year the knowledge divide is often filled.

Arts students often lack basic technical computing skills. The first weeks of foundation level class work is spent learning routine tasks. Simple organization, file/folding hierarchy, basic OS usage, troubleshooting, etc. regularly dominate the classroom. This is due, from my perspective, to deficiencies in computer education in primary and secondary school as well as home computer use practices. Operating system transition can be difficult for some transitioning from a family Windows based system to their own personal Mac laptop whereby they lack familiarity. Student command of basic skills greatly enhances the process of instructing new software. When Pd assignments are introduced and final project construction starts around mid semester, if basic computing skills are unsatisfactory, teaching and learning can be quite negatively hindered.

4.3 Problematic Pd

Problems arise because of support for operating systems. Because OSX releases a new OS each year, for example, Pd functionality can be broken without instructor awareness leading to technical issues. Instructors should be prepared to tackle these problems each semester as they occur and report technical issues to the pd mailing list, development group, log an issue on github, etc.

The GEM has often become problematic on OSX given how fast video capabilities, frameworks, etc. change. For example, The Quicktime framework is slowing being phased out in favor of AVFoundation framework leading to potential driver issues when using Pd/GEM combination compiled with the older framework.

Pd-extended was primarily used for several years because it came packaged with all libraries students would need in a single, easily installable package. Yet because it has become unmaintained things slowly have begun breaking thus becoming depreciated. Some Macbook Airs, for example, could not use their internal camera with Pd/GEM, GEM based patches would often crash, certain filetypes could not be read, etc. The transition from Pd-extended to deken based installable packages has been well received and is on the right path to easing installation of external libraries. Also the Purrdata project, presently in beta stage, has promise in filing the void left by the Pd-extended demise. Unfortunately, on OSX systems GEM is still quite deficient in its support with no recently released builds utilizing new Apple video technology namely the AVFoundation framework. Recent community attempts are on the right path but as of this writing are not in a usable state for instruction or student use on OSX systems.

5 Conclusions

The ideal students comes prepared with basic computing skills. Ample time is required for instructors to teach these skills prior to teaching Pd. Basic skill knowledge decreases frustration for instructors and students during introductory Pd lessons.

When introducing arts students to Pd, visual skills can be beneficial to Pd's visual programing platform. Encouraging use of visual analogy for forming connections between the material and virtual worlds enhances instruction and learning retention.

Software is considered a tool useful in meeting a project objective and is not instructed independently from that. Students learn to learn new tools, with instructor help or on their own, to realize their artistic objectives.

Assessment is determined through multi-step competency including a project proposal, subsequence draft revisions, final project submission and exhibition/demonstration. Work ethic, strong artistic concepts, technical skills and ability to independently learn and create lead to stronger projects and are noted when determining

final grade. Creativity and execution as opposed to Pd skill mastery constitutes the main determining assessment factor. More work is needed integrating successful teaching methods to get students to join the Pd community to get help, become developers etc. and for these communities to be more encouraging and assessable for beginner level questions.

Students must be aware that the skills they are learning will be useful in the future. Many artistic disciplines now use computer programming to create visual, designs, interactive projects, etc. Pd is just one programing language they may never use again but the skills/concepts they learn can be useful when learning other languages such as Processing and Arduino.

Faculty teaching Pd must exercise their skills, continuing to improve them. They must contribute to Pd's development through community contributions such as reporting bugs, development, education, attending conferences and sharing knowledge learned. Instruction must be based on situational dynamics: what might work and what might not work often through experimentation, trial and error. Faculty, administration and students must realize that with open-source software there may be bugs and they may not get all bells and whistles proprietary software does.

Teaching is quite an undertaking, it can be very demanding and physically and mentally exhausting. Instruction, especially part-time instruction, can sometimes be demeaning with the inability to make high-level decisions; directly being affected by incompetency from those with not a lot of support or knowledge about what faculty are doing in courses. Also, institutional support, in many ways, may be lacking. Through sharing student project documentation on video sharing websites such as Vimeo and social media platforms, faculty and administration can get an understanding of the type of work that is being done and hopefully become encouraging and gain an appreciation thus easing the instructors burden of explaining and defending the installation and use of Pd in the art classroom.

Teaching Pd requires a certain love: a love for open-source, a love for interface simplicity, and most of all a love for music and art creation.

References

[1] Makey Makey. http://www.makeymakey.com/

[2] G. Ryle: *The Concept of Mind*. 1949

[3] J. H. Rolling Jr.: *Arts-Based Research Primer,* Peter Lang 2013

[4] G. Sullivan: *Art Practice as Research,* Sage 2005

[5] S. Zepke: *Art as Abstract Machine: Ontology and Aesthetics in Deleuze and Guattari.* Routledge 2005

[6] I. Semetsky: Deleuze, Education, and Becoming, Sense Publishers 2006.

[7] R. Haefeli: *netpd,* http://www.netpd.org/

[8] VPA Computer Rec. http://vpa.syr.edu/current-students/undergraduate-students/new-first-year-students/resources

PD Digital Monochord Table - a tool for vibro-acoustic therapy

Anselmo Guerra de Almeida
Programa de Pós-graduação em Música - LPqS-Laboratório de Pesquisa Sonora
Universidade Federal de Goiás, Campus II
Goiânia GO, Brazil, 74110-060
guerra.anselmo@gmail.com

Abstract

Studies in the field of Music Therapy point out to positive experiences in the use of Monochord Table in the treatment of Parkinson's disease symptoms. The aim of this paper is the implementation of a Pure Data model that reproduces similar effects of this therapeutic instrument under electroacoustic reproduction. We conclude that the experience of creating this digital model can contribute to the understanding of the sound phenomenon, enabling comparative psychophysical tests in patients, hypothesizing that the Digital Monochord Table can be used as a complementary treatment therapy.

Keywords

Monochord Table, vibro-acoustic, PD and music therapy, physical modeling.

1 Introduction

The *object of study* is the therapeutic instrument Monochord Table, also called Mesa Lira in Portuguese, or Klangliege in German. It consists of a wooden table whose body is a sounding board, beneath which steel strings tuned at the same pitch attached in tuning pegs are arranged similar to the piano or harpsichord. The patient lies on the table while the therapist plucks the strings with certain gestures, which may be more intense or mild, transferring a sound massage to the individual in his skeletal, muscular and neural systems, who receives the vibration not only through the ears but also from the whole body (fig 1 and 2).

Figure 1: the monochord table

Figure 2: chords at the bottom.

Studies in the field of Music Therapy point out positive experiences in the use of Monochord Table in the treatment of Parkinson's disease symptoms, such as anxiety, stress, muscle tension, aches, fatigue and poor digestion. There are also reports of patients suffering from other disorders with akin symptoms who improved their quality of life.

This research is motivated by the lack of studies that demonstrate scientific evidence to justify the use and effectiveness of the Monochord Table. Researches are being conducted by a multi-disciplinary group in the areas of Music Therapy, Acoustics, Psychiatry, Neuropsychology, which this author is part of.

The *objective* of this paper is the acoustic study of acoustic phenomena present in the architecture of the Monochord Table, its therapeutic performance, and the realization of a computer music model that reproduces similar effect under the electroacoustic reproduction.

2 Methodology

The methodology initially consists on observation of acoustic phenomenon and computational implementation of a model using the programming language Pure-Data [1]. The implementation consists of the following modules:

(1) Gesture capture module: using MIDI controller (or mobile devices via OSC) to capture the movement that simulates the movement on the strings.

(2) Sound synthesis module, based on the Karplus-Strong theory for simulating plucked sounds.

(3) Spatialisation module: translates the gesture information to the stereophonic distribution.

(4) Resonance and reverberation module: performs digital signal processing on the simulation of the soundboard and the sound projection in the environment.

(5) MIDI sequencing module: allows the recording of a performance, recording a MIDI file and its subsequent execution.

(6) Audio recording module: allows recording the sound output to a file type WAVE e or AIFF.

For this implementation was used Pd-extended version 0.43.4.

3 Implementing a digital monochord table

The process of sound generation starts with the capture of the gesture through a MIDI controller [2] or mobile devices via OSC. If this controller is a keyboard, the midi-note is used not as pitch because this parameter is fixed. This is used to map the gestural movement, reflecting the dynamics and spatialisation. A function is added to introduce micro-detune-variations by pseudo-random function on each note, since this characteristic is present in the original instrument, as observed by this author. But the velocity is captured to record the intensities being used to compose the dynamics and envelope of each event (fig 3).

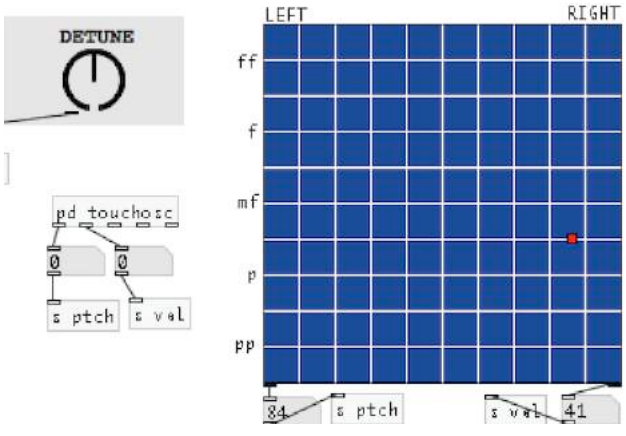


Figure 3: Gesture Capture - besides [notein] one can make midi notes by a 2D Grid. TouchOSC can send the same 2D Grid information. Each pitch sent can introduce micro-detune by a random function.

With this elements the Sound synthesis module is made based on the Karplus/Strong theory for simulating plucked sounds, as showed by Roads [3] and implemented in PD by Kreidler [4].

Subverting the function of the keyboard controller, the low/high relationship will represent the left/right sides of the strings, giving the sensation of movement of the sound pressure in different regions, since the movement of parallel chords in relation to the body position implies sound pressure variation in specific parts of the body and ears. This is the spatialisation module task.

We implemented the string-tuning option based on A440 or A432 pitch as an additional feature, since the A432 tuning is often used for alternative therapies. This is the reason we do not use the function [mtof].

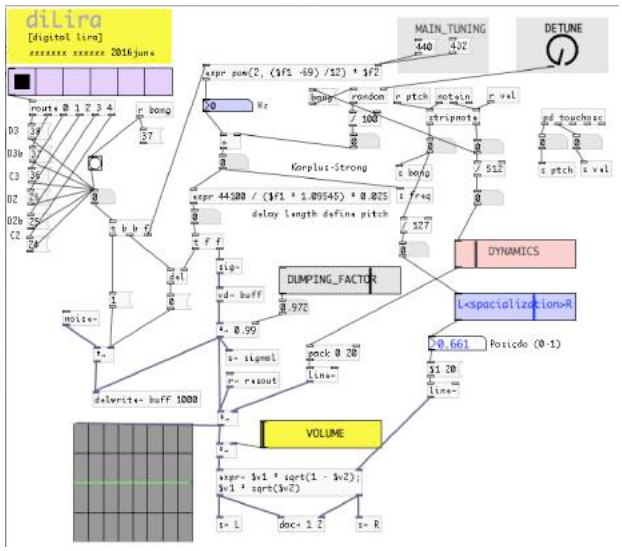


Figure 4: main part of the patch – Tuning, Detune, Karplus-Strong algorithmic, Dynamics, Spacialisation and general Volume.

We can see the main patch in figure 4 was the principal focus is the Karplus/Strong algorithm. It's a special use of looping algorithm, a process of physical modeling synthesis that attempts to simulate plucked strings:

> […] When it is plucked, a string first vibrates chaotically then adjusts itself to the length of the string. It also loses energy, i.e., the vibration dies away. This can be reconstructed mathematically by taking an excerpt of white noise and playing it back periodically again and again by writing it to and reading it from a buffer [4]: p. 132.

Then, the Karplus/Strong part receives the frequency information algorithm, which is

fixed. In the case of Monochord Table is usually tuned in D. In fact, there are several strings with the same pitch, and we have several attacks depending on the performer's gesture.

We observed that between strings there are micro-differences in pitch, producing an effect in the acoustic instrument that can be simulated in the digital instrument with small changes in frequency by [random] function introducing these variations through a pseudo-random function. These variations between the notes became significant because it results in a more intense sensory experience for the user, and cannot be cut due to the machine's accuracy

The resulting sound from Karplus/Strong algorithm is partly sent directly to the spatial distribution module, and partly sent to resonance and reverberation modules, so that they are added to the original sound and sent to the *Spatialisation module*.

As said by Porres [4], panoramic is a technique for playing an audio signal in different channels, such as distributing a mono channel stereo. More than just doubling the signal, a pan allows you to distribute a sound to a right or leftmost source. This is basically the same thing as making a crossfading, only instead of mixing two sound sources to the same channel; we are playing a source on two different channels. The type of panning used is an Equal Power panning or Constant Power. The idea is that the intensity of perception is the same – regardless of the pan position – attenuates both channels and around 3 dB in the central position. The Constant power/Attenuation function of 3 dB at the center works by square root.

We would like to highlight here that the Pan function is used to recreate the performer's gesture according to the position of the strings below the patient.

The resonance and reverberation module completes the sound projection as the Monochord Table has a resonance box that serves both as a natural amplifier and acts as a filter that emphasizes harmonics observed in the performances. The partials produce moves with great variety, like choir of harmonics, in contrast to the monotony of the fundamental pitch. To implement the reverberation the model of Schroeder/Moorer [6] was used, increased by an early reflections generator. The [freeverb~] was the appropriate function to be the basis of this module.

We added the resonance module to emphasize the intense presence of overtones that we observe in the original acoustic instrument. To represent the movement of these overtones we include an envelope (using [line~]) that scans a region of harmonics above the 8th harmonic applied to the Resonance Frequency parameter. The main function used in this module is [reson ~]. The resonance and reverberation module parameters we can see on figures 5 and 6:

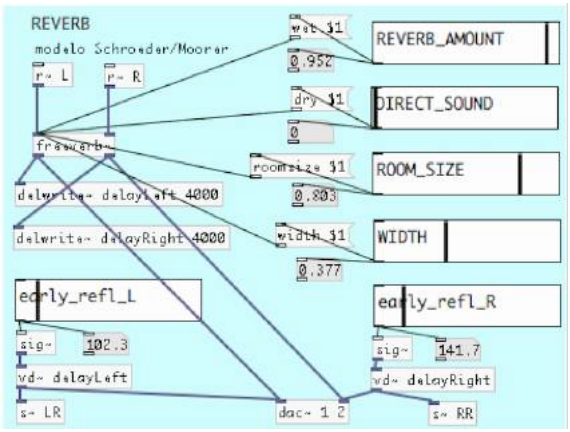


Figure 5: Reverb parameters.

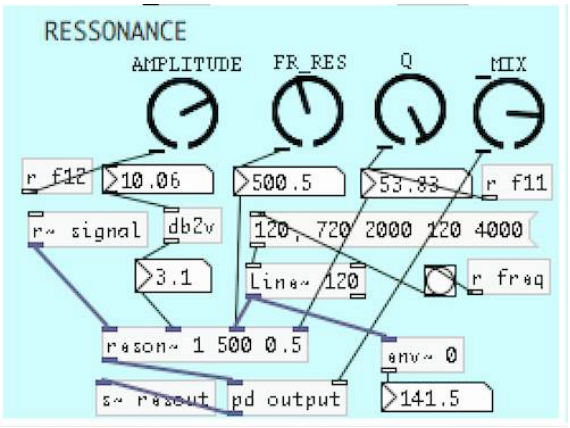


Figure 6: Resonance parameters.

The observation of the therapist's performance at the original instrument also showed difficulties. The main one was due to the friction of the steel strings causing lesion in the fingers. A similar gesture on a MIDI controller keyboard also causes friction if the performer is performing glissandos repeatedly. Having convenience and the possibility of advanced programming gestures in mind, MIDI sequencing module was created, which can be written once and run as many times as necessary.

The generated MIDI file can also be created or edited in an external program such as DAWs. Besides, in our digital model one can use the 2D panel that translates gesture in Midi messages, both in the computer interface and also via OSC if using a compatible mobile device. A simple interface was design to access the MIDI sequencer functions:

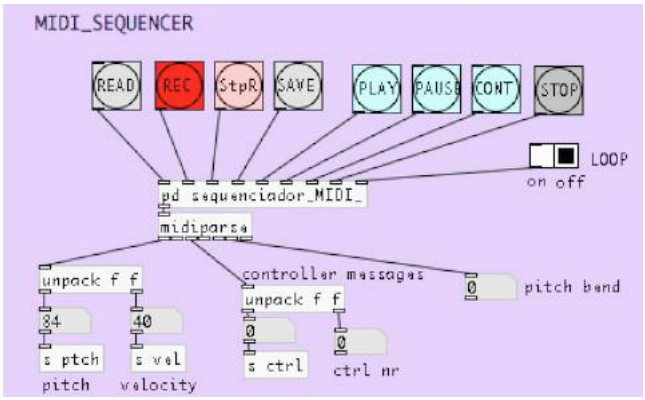


Figure 7: MIDI sequencer interface – read a file, record, stop record, save file, play, pause, continue, stop play and loop on/off.

Finally, the audio recording module allows portability to potential users, since it generates a WAV or AIFF format file that most computers can play with their players, or even smartphones.

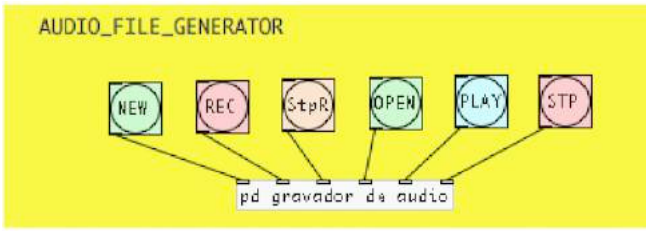


Figure 8: Audio recording module interface – create a new file, record, stop record, open file, play, and stop play.

4 Comparative performance

As stated above, the Monochord Table consists of a wooden table whose body is a sounding board, beneath which steel strings tuned at the same pitch are arranged.

The therapist's musical gesture is similar to the fingering of a harp, performing glissandos and arpeggios that may be faster or slower. The movements can be closer to the head or down towards the belly. The movement can be unidirectional or circular. All these motion variations translate into intensity variations and perception of sound in space.

The performance of the digital version can be performed in real time by 3 ways:

(A) using a keyboard controller that sends MIDI messages. The glissandos and arpeggios can be performed on the keys. Like the Monochord Table, all keys have the same tuning, with the region of the bass notes on the left side of the table, and, as a consequence, the high notes on the right. The *midi-velocity* message transmits the feeling of the top / base relative to the patient's body.

(B) using the [grid] interface where the X dimension maps the left-right relationship and the Y dimension maps the piano / forte relationship, ie, the distance ratio and closeness to the ears (see fig. 3).

(C) through the OSC protocol, sending MIDI notes through the same 2D grid system used in (B), with the advantage of performing the movement with the fingertips on touch screen of a smartphone or tablet.

In the original instrument, the strings should be tuned at the fundamental frequency of D3. However, practical measurements performed showed microtonal variations between the strings, coming from the material instability and its vulnerability to temperature, pressure and time of use.

The reason for using the pitch tuned by D3 is not found in literature sources. Our hypothesis is that this note is located in a central region where the resonance is stronger in the human chest (50 to 100 Hz), as seen on figure 5:

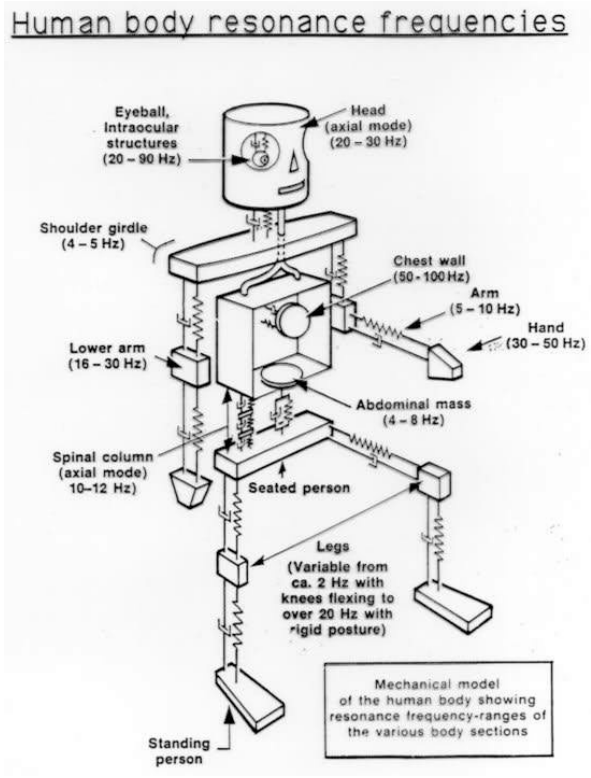


Figure 5: Mechanical model of resonance frequency ranges (picture credit to https://www.physicsforums.com/threads/human-body-resonant-frequencies.501607/)

Therefore, the pitch D (about 73 Hz) is located in a central position within the range of resonance sound in the human chest.

5 First results

We conducted some performances for a select group of students and researchers involved in music therapy practices that have had experience with Monochord Table. The reactions were unanimous in recognizing the sounds produced as highly similar to the acoustic instrument. This was just an informal audition test, but psychophysical experiments will take place in another step with a larger group of researchers.

We emphasize that the resulting sound from the digital instrument reflects the sound from the patient's point of view and not from the therapist or an observer in the room. This methodological option allows a user carrying headphones (preferably) has the acoustic feeling of being on the real Monochord Table.

6 Conclusion and next steps

The experience of creating a digital model that reproduces the sound of the Monochord Table can contribute both to the understanding of the sound phenomenon, allowing comparative psychophysical tests in patients are made, hypothesizing that the Digital Monochord Table can be used as a complementary treatment therapy.

A multidisciplinary team should conduct the next steps. This team will apply EEG tests on volunteers to study the variation patterns in neural responses when using the acoustic Monochord Table and then the Monochord Table Digital. From these results the patch interface can receive improvements that enhance the relationship with the users.

The actual stage of implementation presumes that earphones can induce the effects of the real Monochord Table. In the case of using the digital model as a complementary therapy, it is assumed that the patient will activate his body memories. On the other hand, if the tests results conducted by the multidisciplinary team indicate divergence due to the lack of tactile element, the current digital model can acquire hardware elements that apply physical vibrations and resonances at a table to transmit the vibro-acoustic sensation.

7 Acknowledgements

Our thanks go to the Universidade Federal de Goiás – UFG and FINEP & Fundo de Infra-Estrutura – CT-INFRA that supported the Laboratório de Pesquisas Sonoras – LPqS. Special thanks to Professor Dr. Tereza Raquel Alcântara-Silva (UFG) for introducing the Monochord Table.

References

[1] Miller Puckette: *The Theory and Technique of Electronic Music.* Hackensack, N.J.: World Scientific Publishing Co. 2007.

[2] Andy Farnell: *Designing Sound*, p. 31, Cambridge, Mass.: MIT Press, 2010. http://www.books24x7.com/marc.asp?bookid=47538. Accessed August 2016.

[3] Curtis Roads (editor): *The Computer Music Tutorial, p. 293,* Mass.: MIT Press 1996.

[4] Johannes Kreidler: *Loadbang: Programming Electronic Music in Pd*, p. 120, Hofheim: Wolke, 2009.

[5] PORRES, Alexandre. EL Locus Solus – Live Electronics Tutorials: chapter 9, patch 1, https://sites.google.com/site/porres/pd. Accessed August 2016.

[6] Curtis Roads (editor): *The Computer Music Tutorial, p. 481,* Mass.: MIT Press 1996.

The Mobility is the Message: the Development and Uses of MobMuPlat.

Daniel Iglesia
Iglesia Intermedia; Google, Inc.
California, USA
daniel.iglesia@gmail.com

Abstract

This paper documents the development of MobMuPlat, a software suite for running Pure Data on mobile devices. It chronicles initial ensemble-oriented goals, pre- and post-launch development, and a few common user issues. It also lists a range of projects which use MobMuPlat, highlighting how the reduction of technological barriers yields new and unanticipated results.

Keywords

Pure Data, libpd, mobile, ensemble.

1 Definition

MobMuPlat[1] is a platform, built on libpd[2,3], for running Pure Data (Pd) patches on mobile devices. No text coding is needed. In addition to running a patch, MobMuPlat handles the user interface, networking, interaction with mobile hardware and services (sensors, GPS), and communication with external devices such as MIDI and HID devices. The GUI can be the "native" Pd GUI, or a custom interface with additional features and widgets suited to mobile (e.g. swipable pages, multi-touch). With a goal of development and distribution on any OS, MobMuPlat is four pieces of software: an iOS app, and Android app, an OSX editor application, and a cross-platform (Java Swing) editor application. In addition, multiple networking protocols are implemented for robust group communication, reflecting MobMuPlat's original focus on ensemble performance.

2 Origins

MobMuPlat sprang from a specific mix of interests and frustrations. From 2010 to 2013, I was a co-leader of the Princeton Laptop Orchestra (PLOrk)[4,5], a group comprised of rotating classes of students. I was also a member of its sibling ensemble Sideband[6], comprised of a more stable group of graduate student, staff, and faculty. Due to the fertile and wide-ranging interests of both groups' composers, both groups aim to support a spectrum of hardware and software. The range of audio applications, drivers, and controller hardware would often lead to technical issues, consuming time in long rehearsals. Performers each had a multi-channel hemispherical speaker and subwoofer, combined with all the controllers and cabling required for various works. The sheer amount of physical equipment added logistical issues and long setup times.

My goal was an ensemble performance model to counteract these vexations. It would be self-contained, hand-held, mobile, battery-powered, with minimal software or hardware conflicts, minimal technical learning curve, and no cables to lay down ahead of time. An electronics ensemble should have the mobility of an acoustic ensemble: the ability to walk onstage with minimal setup and perform as self-contained individual sound sources. Rather than sitting, tethered to laptop screens, the hardware should get out of the way, allowing motion and "heads-up" ensemble interaction.

As someone personally invested in electronic ensemble repertoire, I also hoped for an ensemble model requiring a significantly smaller budget, so that the model could exist beyond well-heeled music departments. In both musical and educational contexts, providing each student with a laptop is often infeasible, and can be an unnecessary technical distraction from an musical or educational goal. This lower technical and financial barrier also meant acoustic ensembles could more easily perform pieces with electronics.

Composers for PLOrk and Sideband predominantly use audio software which are unavailable on mobile operating systems (e.g. Max, Supercollider, ChucK); no one wants learn to write native mobile applications for a single work. The fortunate appearance of libpd, however, meant that composers could still develop in a familiar graphical audio environment, and deploy their work to

mobile devices. Graphical environments also allow fast prototyping and tinkering in a way that lower-level text coding does not, making Pd & libpd (which run on nearly anything) a good intersection between composer-friendly and mobile-OS-friendly.

Unlike solo electronics performance, which usually demands complex control of multiple streams of audio, works for LOrks are often fairly technically and instrumentally simple, with a single synth or a sampler and a few parameters exposed to each player. Richness or complexity was not required from any single player, but arose from the combined behavior of the group. Mobile devices fit this model, in which complex GUIs, inter-app routing, DAWs, or other laptop-only qualities were not necessary. In fact, the homogeneity and limitations of mobile operating systems (in which applications have limited power to influence the overall state of the OS) is a benefit, cutting down on incompatibility and driver issues. It also helps that everyone already has one in their pocket.

The missing pieces were:

1. A graphical user interface to control the patch, particularly one that was oriented towards mobile interaction (multi-touch control, swiping through pages of content), specifiable and editable as text. This would be a set of widgets (sliders, knobs, toggles, XY sliders, etc) defined in JSON and rendered by native app code.

2. Leveraging the mobile device's sensors and hardware connectivity (tilt, compass, GPS, MIDI and USB controllers) as control input. These would be routed between the native app layer and the user's patch.

3. Network communication (OSC, and one-to-many protocols that improved reliability over pure multicast). Connectivity (addresses, ports) would be managed natively, and with messages routed to/from the the user's patch.

Once the above were complete, and after initial testing by colleagues, MobMuPlat was publicly released on iOS in January 2013.

3 Post-launch development

The development of MobMuPlat was influenced first my own interests, then by the interests of my personal circle of composers and performers. After launch, many other voices appeared, including early users versed in Pd, users of similar software, users in new performance configurations, and eventually longer-term users with deeper needs and questions. This section tallies MobMuPlat's continued development after initial launch.

Some missing features were apparent, and added shortly after launch. Composers working with dynamic graphical scores had little recourse until an "LCD" object was added, with simplified syntax inspired by Max's LCD object. Complex audio control signals were not expressible/drawable until the "table" object was added, with the ability to directly interact with Pd tables. For users looking to connect mobile to laptop, direct unicast to a user-specified IP and port was added. By popular request, I added Audiobus support, allowing MobMuPlat to be a node in a larger app-to-app audio graph.

Up to this point, MobMuPlat was only on iOS; the Android version was released in late 2014. Though it had the benefit of opening MobMuPlat to many, many more users, my primary motivation was far more specific. The GameTrak tether controller has a unique ability to move electronics performance into physical/gestural space; it had become a staple of electronic ensemble repertoire by this point[7]. HID devices remain publicly unsupported on iOS, while Android, a more open platform, supports them natively. As an initial goal of using mobile devices was to get performers away from a laptop screen, towards physical heads-up performance, then supporting tether input was a high priority. Prime pieces of PLOrk and Sideband repertoire (e.g. Jascha Narveson's *In Line*[8]) were now portable to mobile.

Narveson also designed and wrote LANdini[9], a networking protocol which added guaranteed message ordering and delivery on top of UDP networking. LANdini was an early feature of MobMuPlat, for the same reasons as tether support: to port existing LOrk repertoire to mobile. Its use on mobile devices, however, presented a limitation: power. LANdini was written for plugged-in laptops, and so heavy CPU and messaging load was not a problem. The same load on a mobile device would yield fast battery drain (along with a notable heat). Though LANdini remains supported on MobMuPlat, a leaner counterpart was needed. I wrote the Ping & Connect protocol in response,

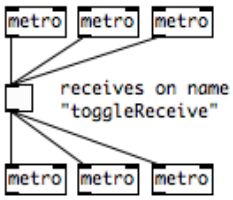
with a simple and unoriginal pattern. Like LANdini, each client pings out their existence (with an optional player index, the better to refer to one another programmatically) and everyone creates direct connections to the pings they receive. This model is a common one, and was baked into the earliest ChucK pieces for PLOrk (e.g. Clix [10]). Though message receipt is not guaranteed, a good router will drop few packets over direct socket connections[11], and the performance and battery usage is improved.

After MobMuPlat had matured this far, I revisited its overall learning curve. Creating a custom interface shouldn't always be necessary, and so I explored adding "native" Pd patch rendering. A user's patch ought to be openable directly, displaying its toggles, sliders, comments, etc. Fortunately, work by Chris McCormick (PdDroidParty for Android[12]) and Dan Wilcox (PdParty for iOS[13]), provided existing examples and reusable code. Their pattern was to have users notate desired Pd GUI widgets with send/receives, communicating their state to the rest of the patch; this works well with libpd's structure, in which the native app layer communicates via send/receive with the audio patch. I, however, endeavored to use patches without this modification, so that users could drop in their patches as-is. This turned out to be non-trivial. A brief overview of the strategy follows.

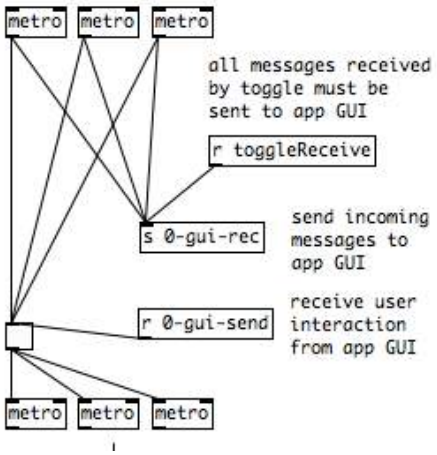
3.1 Native GUI processing

On selecting a patch to open, the patch (as text) is processed into two separate patches. The first is rendered as the app GUI, with auto-generated send/receive names for each widget. The second is the patch actually running within libpd, in which each widget is now surrounded with send/receive objects. All messages flowing to the patch widget are sent to the app GUI to render its visible counterpart; all messages flowing from the app GUI (in response to user interaction), are sent back into the patch widget. I attempt to illustrate this below.

Here's an initial patch state defined by the user:



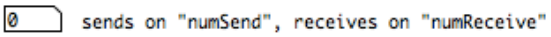
And here's what is generated internally for the patch. `"0-gui-rec"` and `"0-gui-send"` are the handles with which the app GUI widget communicates:



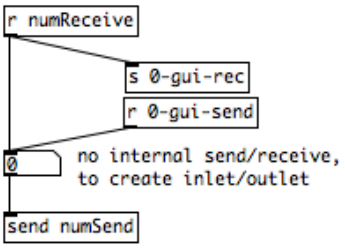
Note that there is no pass-through at the app GUI layer, e.g. something going to `[s 0-gui-rec]` will not trigger an output in `[r 0-gui-send]`.

There are still, however, some complications. Atom widgets (e.g. float number boxes) remove their inlet/outlet if the widget uses a send/receive name. This complicates the ability to connect to/from it. So in this case, the processing logic modifies the float box to not have internal send/receive names, restoring the inlet/outlet, and then manually generate send/receive objects to mimic the intended behavior.

Here's an initial state:



And the post-processing state:



4 Issues and future work

This section covers a few common questions asked by users, combined with some possibilities for future development.

The largest set of questions involves 1) the use of Pd-extended and external objects, and 2) the ability to generate a standalone app for an app store. To some, MobMuPlat's ease of use implies that related tasks are also easy; they are

not. Both these tasks are doable, but involve working with native mobile source code and IDEs, at which point most users understandably lose interest. (The whole point of MobMuPlat, after all, was to avoid needing to do that.) Regarding standalone apps, I believe that unless one wishes to monetize an app (which is difficult without a slick custom interface), or one is paranoid about people looking at one's patch, that installing MobMuPlat (or PdParty, PdDroidParty, etc) and sharing the patch is usually much simpler (and more open). The inability to create and monetize a standalone app also allows the community to avoid the ethical complaints, occasionally made on Pd forums, about open source and personal/commercial gain.

However, the questions regarding sharing patches coincides with a related set of questions regarding mass distribution. A composer/performer may wish to distribute a large-group work (e.g. a sound walk or audience-participation piece) and hope to do so in as few steps as possible. Right now, there is no faster way than installing MobMuPlat, and then downloading the patch and interface files onto the device, and then returning to MobMuPlat to select them. Future work may attempt a method of "pushing" patches to connected devices (though this has the additional hurdle of requiring the group to connect to a local router); this may help for small collaborative groups, but I doubt it will solve the large-group problem.

While I consider development of MobMuPlat to be approaching feature completion, there's still a few more open considerations for the future. Watch and other wearable support is of interest, since it allows even more heads-up musical interaction. MobMuPlat has Android Wear support (for specifying watch widgets that communicate with the mobile device) for some time; I personally use this to control a mobile device while using tether controllers. However I've not yet publicized this feature, nor investigated parity with Apple watches, nor implemented editor support. Additionally, my interest in networking protocols will likely continue, and there's constant discussion of new protocols; if any gain traction with a large community I may consider integration. (At the time of this article, there's some chatter about Ableton's new standard. No promises.)

5 Uses

MobMuPlat was originally conceived with specific uses in mind, but the primary personal benefit of MobMuPlat's adoption has been the surprising ways in which it has been used. Some projects continue common uses of Pd (e.g. solo performance, acoustic instrument augmentation, installation), while some highlight new use as ensemble instrument or embedded device, and many highlight uses completely specific to the nature of mobile devices (e.g. toys, running). This section contains a sample of the ways MobMuPlat has been used.

- **Mobility-based ensembles**: Though mobile-device-based ensembles have existed for some time[14,15], some new ensembles specifically use mobile devices in order to roam around public space. These include an outgrowth of PLOrk[16] (with hand-made controllers), and El Smartphone Ensemble in Colombia[17,18]. I discuss my own mobile ensemble in the following section.

- **Demonstration of scientific research**: MobMuPlat was used to demonstrate how the interaction of fireflies can be a model for synchronization within a group of devices[19,20].

- **Instrument design for solo performance systems**: Toby Hendrick (aka otem rellik) has built a suite of MobMuPlat-based instruments for his live rig[21].

- **Acoustic instrument extension**: For real-time acoustic processing, mobile phones are smaller and less distracting than laptops, and easier to deploy than embedded computers. Some examples include porting a processing chain for trombone performance (including AudioBus)[22] and an embedded processor for violin[23].

- **Concert works with audience interaction**: Celeste Oram recreated Vera Wyse Munro's 1940 experimental work *Skywave Symphony*, using audience's mobile devices in place of radios[24].

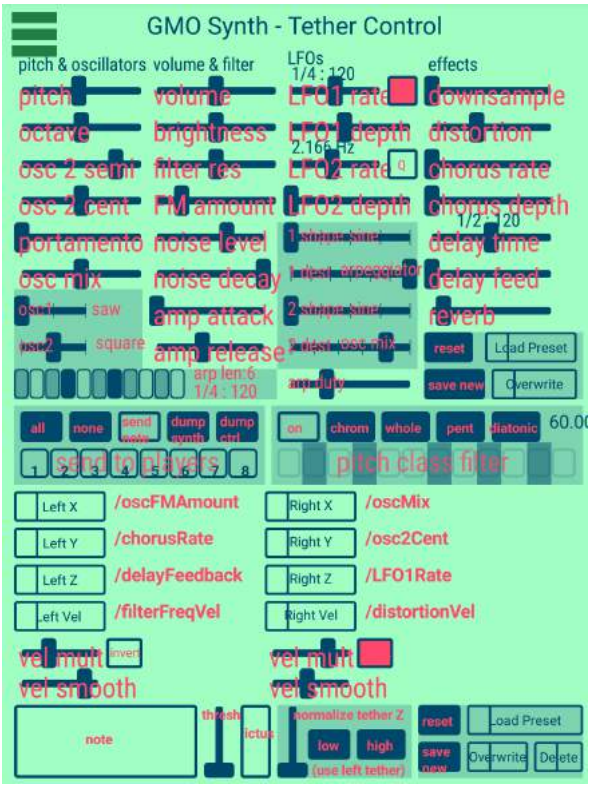
- **Sound mapping, sound walks, and personal surveillance**: Some projects focus on group experience in a specific location, including works for Seoul, Korea by Max Neupert[25] and Hailuoto, Finland by Antye Greie-Ripatti[26]. Others create conceptual works for personal, private experience, such as Oram's *Soft Sonic Surveillance*[27] and Beatrice Monastero's *Wandertroper*[28].

- **Toy development**: One of the most endearing and unexpected uses of MobMuPlat is Notori, a project to recycle mobile phones into toy components[29,30].

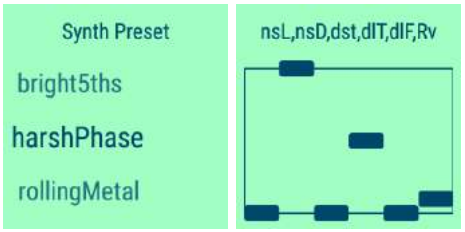
- **Research on running and music**: Multiple projects have used MobMuPlat to measure how music motivates running performance[31,32].

5.1 The Google Mobile Orchestra

The Google Mobile Orchestra (GMOrk), performs on tablets, with tether controllers, portable speakers, networking, and live video. It began in the LOrk model, with a repertoire of works, both adapted from the LOrk repertoire, and newly commissioned for mobile. The LOrk model, based on changing student membership with varying levels of musical experience, results in works often using fairly simple high-level control of a musical process, within a directed score. The members of GMOrk, however, felt that lower-level control, in a more self-directed improvisational context, would suit us better. While we retain the existing externally-composed repertoire, we have developed our own performance patches with many low-level parameters, custom real-time remapping to physical controllers, and heavy use of networking.



A page of a GMOrk interface, enabling control of, and tether mapping to, synth parameters.



Two pages of the accompanying watch GUI, for selecting presets and controlling a subset of synth parameters.

The basis of our networking allows any performer to change any parameter of any other performer. This can be useful to impart timbral or rhythmic unity when desired, and to inject challenging elements (e.g. "everyone send your commands to your left") which yield unanticipated results. This pattern is indebted to early network improvisers, particularly The Hub[33]. As every musical parameter is broadcast over the network, a central computer can then act as a recipient of this traffic, and re-renders the audio of all the players, using the audio to render real-time video for 3D glasses[34].

References

[1] `www.mobmuplat.com`

[2] `www.libpd.cc`

[3] Brinkmann, Peter, et al. "Embedding Pure Data with libpd." *Proceedings of the Pure Data Convention.* 2011. `link`

[4] `plork.princeton.edu`

[5] Trueman, Daniel, et al. "PLOrk: the Princeton laptop orchestra, year 1." *Proceedings of the International Computer Music Conference.* 2006. `link`

[6] `www.sidebandband.com`

[7] Freed, Adrian, et al. "Musical Applications and Design Techniques for the Gametrak Tethered Spatial Position Controller." *Proceedings of the 6th Sound and Music Computing Conference.* 2009. `link`

[8] `www.sidebandchronicles.com/jascha-narveson-in-line/`

[9] Narveson, Jascha, and Dan Trueman. "LANdini: a Networking Utility for Wireless LAN-based Laptop Ensembles." *Proceedings of the Sound and Music Computing Conference.* 2013. `link`

[10] Smallwood, Scott, et al. "Composing for Laptop Orchestra." *Computer Music Journal* 32.1, 2008. `link`

[11] Cerqueira, Mark. "Synchronization over Networks for Live Laptop Music Performance." Master's Thesis, Department of Computer Science, Princeton University. 2010. link

[12] http://droidparty.net/

[13] github.com/danomatika/PdParty

[14] Wang, Ge, Georg Essl, and Henri Penttinen. "Do Mobile Phones Dream of Electric Orchestras?" *Proceedings of the International Computer Music Conference.* 2008. link

[15] Oh, Jieun, et al. "Evolving The Mobile Phone Orchestra." *New Interfaces for Musical Expression.* 2010. link

[16] Snyder, Jeff, and Avneesh Sarwate. "The Mobile Device Marching Band." *Proceedings of the International Conference on New Interfaces for Musical Expression.* 2014. link

[17] sonologiacolombia.wordpress.com/lab/ensamble-de-smartphones/

[18] J. J. Arango and D. M. Giraldo, "The Smartphone Ensemble. Exploring Mobile Computer Mediation in Collaborative Musical Performance." *Proceedings of the International Conference on New Interfaces for Musical Expression.* 2016. link

[19] Nymoen, Kristian, Arjun Chandra, and Jim Tørresen. "Firefly with Me: Distributed Synchronization of Musical Agents." *Awareness Magazine.* 19 November 2013. link

[20] www.vimeo.com/67205605

[21] www.youtube.com/playlist?list=PLpP3G7pBTV-IiLO1oSdR7hI79sktGp3kb

[22] no-insects.blogspot.com/2014/10/meta-trombone-on-ios.html

[23] Overholt, Daniel, and Steven Gelineck. "Design & Evaluation of an Accessible Hybrid Violin Platform." *Proceedings of the International Conference on New Interfaces for Musical Expression.* 2014. link

[24] verawysemunro.nz/re-enacting-the-Skywave-Symphony

[25] choejeongeun.com/xD25-1-GPS-Audio-Walks

[26] medium.com/@poemproducer/when-you-listen-it-listens-back-4c4d99bea8f8

[27] workandthe.work/II-Soft-Sonic-Surveillance

[28] www.dawn.ul.ie/?/interactive-media/2014/BeatriceMonastero/

[29] www.yuichirock.com/notori/

[30] Katsumoto, Yuichiro, and Masa Inakage. "Notori: Reviving a Worn-Out Smartphone by Combining Traditional Wooden Toys with Mobile Apps." *SIGGRAPH Asia 2013 Emerging Technologies.* 2013.

[31] www.marcobasaglia.com/improving-running-motivation

[32] Forsberg, Joel. "A Mobile Application for Improving Running Performance Using Interactive Sonification." 2014.

[33] Gresham-Lancaster, Scot. "The Aesthetics and History of The Hub: The Effects of Changing Technology on Network Computer Music." *Leonardo Music Journal.* 1998. p39-44. link

[34] www.vimeo.com/178834011

Wavefolding: Modulation of Adjustable Symmetry in Sawtooth and Triangular Waveforms

Dr Edward Kelly
Synchroma Audio Engineering
11 Spenser Road
London, United Kingdom, SE24
0NS synchroma@gmail.com

Abstract

The Pulse-Width Modulation (PWM) technique has been used to generate varying timbres of odd-harmonic spectra from early on in voltage controlled analogue synthesis history. Methods for controlling the symmetry of a triangle-to-sawtooth wave have also been devised. This paper discusses a family of objects and techniques for piecewise waveform manipulation that may be modulated at audio rate, comparing the results with analogue equivalents, and looking specifically at the implications of modulator phase and subtle deviations from integer carrier-to-modulator ratios, and fine deviations from these, on adjustable-symmetry sawtooth waves.

Keywords

Synthesis, Timbre, Modulation.

1 Introduction

The sawtooth or ramp wave is a fundamental element in subtractive synthesis, since it contains both odd and even harmonics of the fundamental frequency. It's slightly dull cousin, the triangle wave, has weak overtones of odd harmonics and sounds much like a digital approximation of a sine wave. Both have their uses in synthesis, but it is possible in both analog and digital domains to generate waveforms that can be modulated between sawtooth and triangle. Some digital synthesis methods have used this principle particularly since the transformation from a sawtooth wave into a triangle wave creates a reduction in harmonic richness similar (but not the same as) subtractive filters. Historically, Casio's ill-fated VZ series of synthesizers in the 1980s used a method called IPD or Interactive Phase Distortion, based on the transformation of waveforms through progressively sharper sawtooth shapes. Software glitches with the interface along with bad commercial timing (the Korg M1 released at the same time, which also had a sequencer and drums) led to the withdrawal of Casio from the pro-audio market.

With computer synthesis it is a simple procedure to create an algorithm that generates adjustable symmetry sawtooth-to-triangle waves that may be modulated at audio frequencies. Empirical research into harmonic spectra of such modulations reveals a slightly more complex morphology of spectra than would be devised using subtractive methods, and the application of single frequency modulation (sine-wave modulation) of the waveform results in complex timbre transformations over time, highly dependent on phase ratios between carrier and modulator, and a temporal morphology that reflects the characteristic shape of the sawtooth wave itself.

2 The Wavefolder~ Object

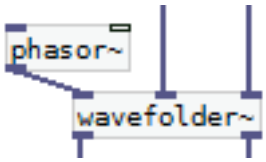


Figure 1. The wavefolder~ object generates variable asymmetry sawtooth/triangle waves from a phasor~ (ramp) input.

The implementation of an algorithm for converting a ramp wave into a triangle wave or inverse ramp is relatively simple. This was initially accomplished as a Pure Data[1] (Pd) patch using the sigpack~ library of objects[1]. More recently this has been created as an external for Pd, along with the wavestretcher~ and wavecutter~ objects. This has simplified the process of converting a ramp from a phasor~ object into an adjustable-symmetry waveform, and opened-up the possibility of audio frequency modulation of the waveform symmetry. The principle of wavefolder~ is simple. With a ramp waveform from 0 to 1, a threshold is set between 0 and 1. Sample-by-sample the output is given by:

$$O = IF(R > T; R(1/T); 1-((R-T)*(1/(1-T))$$

O = out sample, R = ramp input and T = threshold.

1 https://puredata.info/downloads/sigpack

Divide-by-zero errors are eliminated in a separate function that prevents R from arriving at precisely 0 or 1. This object can be found in the ekext library of Pd externals[2].

3 Spectral Characteristics of Static Waveforms

As the waveform is modulated between a setting of 0 (symmetric triangle waveform) and 1 (asymmetric ramp waveform), peaks and troughs in the harmonic spectrum are developed (see figures 1-4). This was empirically tested in order to establish the relationship between the symmetry of the waveform and the resultant harmonic spectrum, in order to establish how the functional description of a sawtooth or ramp wave is affected by this process. It makes sense to define this relationship in terms of deviation from the sawtooth or ramp waveform toward the triangle, as there is a reciprocal relationship between the troughs in the resultant spectra and the symmetry of the waveform. Furthermore, the reduction in the magnitude of even harmonics is not linear.

There is a modulation between the Fourier series of a sawtooth wave:

$$x_{saw} = \frac{1}{2} - \frac{1}{\pi} \sum_{k=1}^{\infty} \left(\frac{\sin(2\pi\, kft)}{k} \right)$$

and that of a triangle wave:

$$x_{tri} = \frac{8}{\pi^2} \sum_{k=0}^{\infty} \left(\frac{\sin(2\pi(2k+1)ft)}{(2k+1)^2} \right)$$

As can be seen from the figures below, the modulation of the magnitudes of harmonics closely resembles a cosine function of the magnitudes based on the harmonic number, starting at infinity for the ideal saw and starting at harmonic 2 for the triangle. Given that, in additive synthesis both waveforms' harmonics are alternately opposite in phase to the previous harmonic (1, -2, 3, -4 etc. and 1, -3, 5, -7 etc.) there are clues to how the combination of additive sine elements with different phase relationships may result in the spectra observed below. An exponential relationship between the linear asymmetry and the position of the first trough in the spectrum is observed, and the interval in harmonics until the next of these, such that a triangle wave has an absence of even harmonics (2, 4, 6, 8...interval=2) and figures for alternative symmetry settings as shown in the subsequent table and graphical figures.

Asymmetry	Interval
0 (triangle)	2
0.5	4
0.75	8
0.875	16
0.9325	32
1 (sawtooth)	Nyquist (SR/2)

Table 1. Asymmetry settings and their corresponding troughs in the harmonic spectrum.

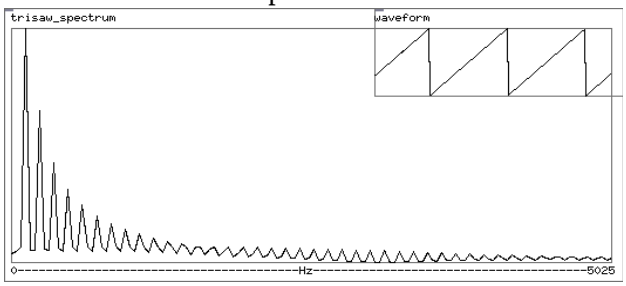


Figure 2. Spectrum and waveform at symmetry setting 1 (ramp waveform).

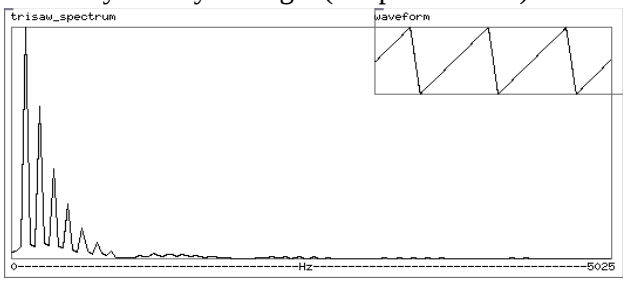


Figure 3. Symmetry setting 0.75.

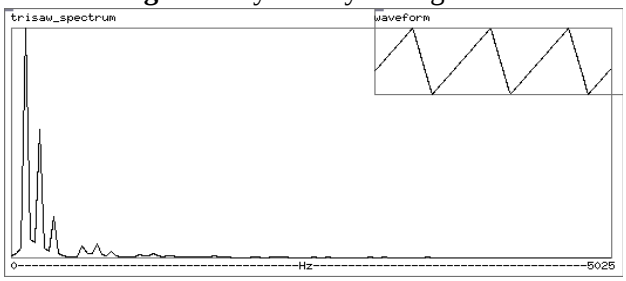


Figure 4. Spectrum and waveform at symmetry setting 0.5.

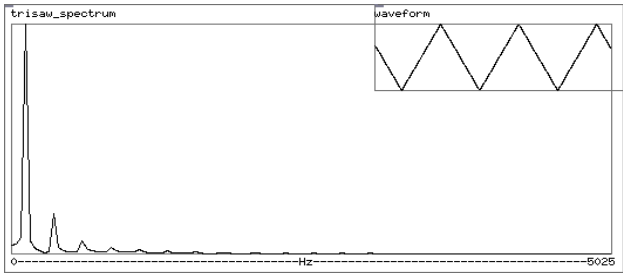


Figure 5. Spectrum and waveform at symmetry setting 0 (triangle waveform).

2 Latest versions can be downloaded from http://sharktracks.co.uk/html/software.html

4. Audio - Rate Modulation of the Waveshape

The shape of the wavefolder~ output is controllable at audio rate with limits of -1 (saw down) and 1 (saw up) with a setting of 0 representing the triangle waveform. The relationships between the phase of the modulation signal (in this case a simple sinusoidal waveform) and the phase of the asymmetry modulation are important to the resulting timbre. With a modulating sine function at the same frequency, at 270º there are more corners to the waveform, and more high-frequency harmonics are generated (fig. 7), whereas at 90º between trisaw and sine the waveform is more like a distended triangle wave and the harmonic spectrum is less bright (fig. 6).

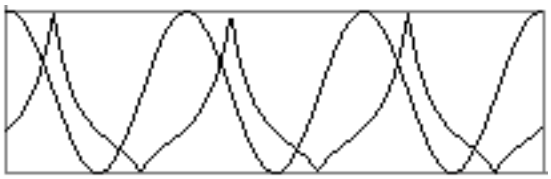


Figure 6. Superimposed waveforms of modulator and resultant waveform at modulator phase = 270º with respect to the tri/saw wave.

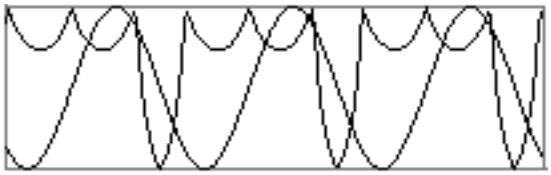


Figure 7. Superimposed waveforms of modulator and resultant waveform at a phase difference = 90º.

4.1 Spectral Morphology at Detuned Modulation Frequencies

Thus far this paper has considered static waveforms, and there is no single result from these that cannot be achieved by a wavetable method of synthesis. But this method begins to yield more interesting results as the modulation waveform is detuned from integer-multiples of the asymmetric modulated waveform. The phase relationship discussed above is continually changing, and this results in morphological transitions between very bright, harsh-sounding timbres and softer timbres.

At a positive detuning away from the frequency of the tri/saw wave, the sweep is from bright-to-dark timbre if the baseline shape is offset with a positive vale, repeating at a rate equivalent to the difference in frequency between the carrier (tri/saw waveform) and the offset from integer-ratio of the modulator (sine). The inverse is true at a negative detuning, that is the sweep in timbre is from dark-to-bright, and this relationship is reversed when the baseline shape if offset by a negative value. More complex timbres are achieved with simple non-integer ratios (1.5, 0.75 etc) giving inharmonic timbres but with a degree of tonality. Just as with frequency modulation synthesis, the more complex the integer ratio of the carrier to the modulator, the more inharmonic the timbre produced.

Furthermore, since the sweep in brightness is a rhythmic effect, this can be controlled mathematically to be consistent across all integer-ratio carrier-to-modulator values, and an object has been created to facilitate this, available on the author's website[3].

4.2 Pulse - Threshold Modulations of the Modulated Asymmetric Waveform

The wavefolder~ object has an extra inlet and outlet at audio rate allowing for the modulated waveform to have a process of pulse-threshold modulation (PTM) applied to it. Since the waveform shapes of a modulated asymmetric waveform are geometrically complex, a set of timbres are available from the object that are more varied than those of traditional PWM. When this is combined with the detuning of the modulator discussed above, the timbre evolution of the asymmetric waveform is transferred to the pulse waveform with the potential for modulations of the pulse-threshold to create further evolutions in timbre.

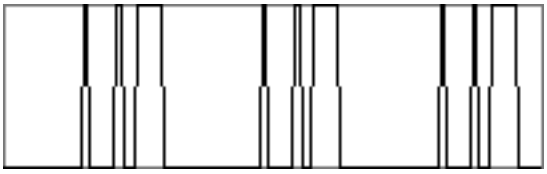


Figure 8. An example of the pulse-threshold modulated output from wavefolder~.

5 More Piece-Wise Waveshape Manipulations

5.1 Wavestretcher~

A second object uses a similar approach the the wavefolder~ by taking a breakpoint (threshold) and manipulating the geometric angle of the waveform differently depending on which side of the threshold it is. It is useful to think of this as a complementary function to the previous object. While the wavefolder~ modulates from a ramp input (from phasor~) towards a triangle waveform using a

3 http://sharktracks.co.uk/html/wavefolding.htm

breakpoint-based algorithm and through to a sawtooth waveform, wavestretcher modulates from the sawtooth (or any input waveform for that matter) towards pulse-train-style waveforms as shown below.

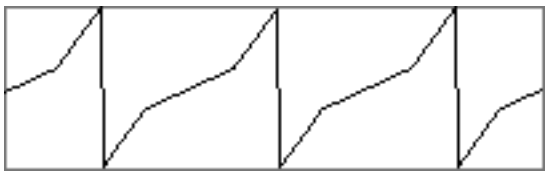


Figure 9. Stretched sawtooth waveform at breakpoint = 0 (middle of absolute value) and stretch factor at -0.5.

Figure 10. With the same sawtooth input, breakpoint = -0.75, stretch factor = -1.

Positive values of the stretch factor allow the modulation between triangular or sawtooth waveforms through trapezoidal waveforms until a square wave or clipped sawtooth waveform results.

Figure 11. With the same sawtooth input, breakpoint = -0.5, stretch factor = 0.7.

The use of both objects, with the output of wavefolder~ feeding into wavestretcher~ affords a situation where a large repertoire of complex timbres may be generate using a highly compact, efficient structure. It is possible to emulate timbres of subtractive synthesis without the use of filters, but with a greater degree of flexibility in terms of timbre control[4].

6 Analogue Realization

There are some implementations of this idea available on synth-DIY sites by Hoshuyama [2]. Tillmans [3] and Gratz [4]. All of these articles observed by this author come with the caveat that they are “untested” e.g.[2] although this seems unlikely given the knowledge and experience of those contributing this knowledge, since such features exist in Moog and MFB synthesizers (Moog Voyager, MFB Dominion) and it would be naive to assume that the potential of these systems was overlooked[5], especially given the ubiquity of PWM in commercially successful forms of popular electronic music and electronic dance music (EDM) over the past four decades. However only the Moog Voyager XL appears to offer a fully patchable (and hence audio rate) modulation of the waveform shape.

Of particular interest for its simplicity is the design and article by Don Tillmans [2], published in 2000 and revised in 2002, providing a simple circuit for analog waveshaping of a sawtooth wave using two operational transconductance amplifiers. A certain amount is left to the circuit-builder to figure out in this article. As the original circuit uses hard-to-find CA3280 chips[6] efforts are ongoing to adapt the circuit to use a readily available LM13700 dual operational transconductance amplifier (OTA) integrated circuit (IC), although a new solution is detailed below, based on the wavefolder~ algorithm.

The analog circuit designed by Don Tillmans (cited by Gratz[3]) uses an equivalent equation to that in the digital object wavefolder~ expressed in a form more mathematically elegant than the coding algorithm expressed in part 2 above, thus:

$$\frac{1}{1+e^{x}}+\frac{1}{1+e^{-x}}=1 \qquad (4)$$

where x is equivalent to the control voltage in the analog circuit, and the threshold/breakpoint value in the digital algorithm of wavefolder~. This accurately reflects the exponential relationship between the wavefolder~ threshold value and the harmonic modulations detailed in table 1, and figures 2-5.

Given that an exponential multiplication of a signal is the reciprocal of a division by a linear increase or decrease of the denominator, an alternative method can be devised for an analog circuit using the principles of the original wavefolder~ algorithm. An analog switch IC replaces the IF statements, and differential amplifiers are used to generate reciprocal control voltages for a voltage-controlled amplifier against a reference voltage. Both the input ramp wave and its inverted counterpart

4 It must be stated that there is no way to reproduce high-Q resonant peaks without the use of audio filters in this system, although a phase-distortion-type equivalent may involve added resonant circuits to one portion of the waveform. This is an area for further research.

5 These articles are all over a decade old at the time of publication.

6 These are now being manufactured by Rochester Electronics: http://www.rocelec.com

are switched alternately using a comparator, along with the control voltages to an exponential converter into an OTA. A single OTA may be used, and in this realization it is a CA3080 – the chip designed by RCA that was vital to the creation of early voltage-controlled synthesizers (the RCA Mk1 and Mk2) which, as the footnote below shows is now available again, albeit in lots of 100 ICs.

This circuit should perform in exactly the same way as the digital algorithm, with a threshold voltage determining the asymmetry of the resultant waveform. Effectively though, every analog implementation of this principle, from the low-frequency oscillator of the Korg MS20 to the switched OTA concept described above, uses the same principle of threshold-switching the separately amplified non-inverted and inverted portions of the ramp waveform on either side of the switching threshold.

7 Conclusion

This project was driven by curiosity into a way of generating complex timbres from simple means, and how pushing methods from analog experimentation by synthesis enthusiasts into the digital domain may open new approaches (asymmetry modulation) to timbre modulation of basic synthesis waveforms. The conceptual process of development for the analog circuit was realized by understanding that through some lateral transposition of the principles of the digital implementation, an analog realization could be created based on the same principles as the digital object. A conceptual loop can be observed where code-based digital methods and analog electronics can be created in parallel, and where understanding from one branch of electronic music can be adapted to function using the same principles in another.

The original approach to waveform geometry modulated at audio-rate produced some exciting timbre-generating possibilities, but the embodiment of these principles in an external object suggested that further manipulations of the geometry of cornered waves might result in further surprises.

A website with details of how to obtain the wavefolding externals, and the poster presentation formatted as a webpage is available on my personal website[5].

8 Acknowledgements

This project was devised and researched by the author, and I am deeply grateful to the University of the Arts London, and particularly Nick Gorse and Jonathan Kearney for releasing funding for its first presentation at ICMC 2016. A great deal of credit for understanding analog circuits in voltage-controlled synthesis should be given to Thomas Henry, Ray Wilson, Ian Fritz and Don Tillmans.

References

[1] Puckette, M, *Pure Data: another integrated computer music environment*, Proceedings, Second Intercollege Computer Music Concerts, Tachikawa, Japan, pp. 37-41, 1996

[2] Hoshuyama, O, *Wave-Shaper (Variable Slope-Ratio Triangular)*, http://www5b.biglobe.ne.jp/~houshu/synth/WvShp0306.gif, 2003.

[3] Tillmans, D, *Voltage Controlled Duty Cycle Sawtooth Circuit*, www.till.com, 1999, 2002.

[4] Gratz, A, *Triangle / Sawtooth VCO with voltage controlled continously variable symmetry*, http://synth.stromeko.net/diy/SawWM.pdf, 2006

[5] http://sharktracks.co.uk/html/wavefolding.htm

Casting a Line: A Flexible Approach to Soundfile Playback Using libPd in Ninja Jamm

Dr Edward Kelly
Synchroma Audio Engineering / Ninja Tune Records
11 Spenser Road
London, UK, SE24 0NS
synchroma@gmail.com

Abstract

This paper describes a strategy for implementation of both smooth playback and glitch-free scrambling of audio on multiple, independent tracks, using a single instance of an augmented version of the phasor~ object: phasorbars~. Each track has control of the playback point, so that functions to re-order slices of each looped sample are independent for each of the four tracks in the Ninja Jamm mobile app. Formulas used in calculating how to offset the single phase signal to achieve rhythmic flexibility within the bar are discussed, and a method for achieving on-the-fly envelopes on each re-ordered slice of audio are detailed.

Keywords

libPd, soundfile, synchronization, Ableton link.

1 Introduction

Ninja Jamm[1] is an app for iOS and Android that gives the user a set of tools with which to remix tracks by a large and growing number of artists on an iPhone, iPad or suitable Android device[2]. At its heart lies a highly complex Pd patch, with a suite of external objects created by the author. This runs all the audio and sequencing functions of the app. The GUI for the app is also a Pd patch using Pd Open Frameworks (POF) by Antoine Rousseau[3], who is also responsible for the GUI Pd Patch, and an Objective C wrapper and extra tweaks to the GUI infrastructure by Christopher Rice of Holderness Media[4], and of course Matt Black of Ninja Tune, who initiated the project and continues to manage it. This creates the environment in which Ninja Jamm operates on an iOS or Android device. In its original inception it was designed to run on an iPhone 4, and given the lack of processing power and memory of this device, a means had to be found whereby all four tracks of audio had to be run from a single phasor~ object. This is complicated by the jamming functions that beat-slice the audio, where each of the four tracks may be programmed to play from any point or slice within the bar on any rhythmic pulse.

The very first version of the app had poor sound quality, due to the fact a de-click mechanism was used on each slice, regardless of whether re-ordering of the material was taking place. In order for tracks to be played back cleanly when no re-ordering is taking place, and because due to low CPU power a simple tabread~ mechanism had to be implemented. Subsequently, memory usage was decreased by the creation of hextab~ and hexread4~ objects, using 16 bit memory rather than 32 bit table~ and tabread4~ objects, but preserving the 4-point interpolation algorithm devised by Miller Puckette and sacrificing load and save functions for a simple copy of data with translation from 32 bit to 16 bit arrays. From 152% CPU load on the iPhone 4, it was reduced to 71% before the 16 bit objects were introduced, and 56% after.

A multiple-bar spanning object was created called phasorbars~. Whereas in the original version the phasor~ output was used to read a single bar's worth of audio and metro objects provided the clock controls, phasorbars~ provides a single ramp signal that loops through multiple bars, and generates 1/8th, 1/16th and 1/24th note clocks from the ramp. These clocks are generated from the ramp/phasor~ signal, so that re-ordered slices will always trigger in time with tempo changes no matter how quickly these changes occur.

Beyond this point, it was discovered that there were discrepancies between control-rate and

1 http://ninjajamm.com

2 Although there is a significant latency with the Android operating system related to the round-trip audio path, but this is improving with newer iterations of Android.

3 https://github.com/Ant1r/ofxPof

4 http://www.holdernessmedia.com

audio rate block-boundaries that had to be reconciled at control-rate, while audio ramps kept-up with the error, so the wrap_overshoot~ object was developed in order to account for this difference in timing due to objects receiving control messages further down the audio chain and implementing them at the end of an audio block. This external deliberately overshoots a signal value of 1 when it is reached until the end of the current block of audio, so that the wrapped signal only returns to 0 when control-rate offsets take effect. Further developments in the app that have led to a more traditional phasor~ object in phasorbars~, but retaining the bar counters that drive the sequencing object in the app, most notably the use of the Ableton Link technique, whereby many instances of the same mobile app may be synchronized together, developed by Peter Brinkmann for Pd[1].

2 Slicing the Line

2.1 The Bar as a Single Phase-Cycle

The basic premise is that 1 cycle of phasor~ output = 1 bar. Within the app there is presently only the capacity for 4/4 time signatures, but the techniques detailed here can be adjusted to accommodate other rhythmic structures with relative ease. The frequency of the phasor~ is set to:

$$f = (60/BPM)/4$$

This assumes a bar length of 4 and a rhythm value of a quarter-note for the BPM (beats-per-minute), hence a 4/4 time signature. This formula that is familiar to many practitioners of electronic music is easily adjusted to account for different pulse and bar structures.

Figure 1 shows the connection structure between control elements of the patch (voice_control.pd) the phasorbars~ signal from the inlet~ and the audio generation objects (voice.pd).

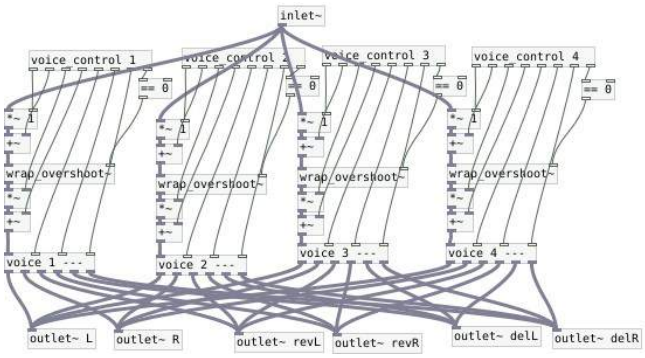


Figure 1. Control and Signal Paths for the Four Tracks or Voices.

The way in which this is structured is such that the length of each clip in bars is reciprocated to multiply the signal, and a float in fractional terms may be used to offset the incoming phasor~ if the loop is offset, that is if the first bar of the loop to play is not the first bar of the sample. The multiplier is unique to each track as is the offset, so that the output of wrap~ is always a vector from 0 to 1. After the wrap~ object, a separate *~ object multiplies the signal by the length of the clip in samples, and an offset in samples may be added for the beat-slicing to occur. This also requires a de-click mechanism when jumps are made from one portion of an audio array to another, since the nature of the app is that it is a platform for many existing works by many different artists – essentially a publishing platform for a record label to release music in an interactive state. The material is unpredictable at the sample level from within the Pd patch.

2.2 De-Click the J umps but Swim in the River

It is a requirement of the app that the user is able to switch between clips (i.e. audio arrays, with 8 per track) or invoke beat-slicing or arbitrary loops at any instant. In order that this process could happen as efficiently as possible, the first version of the app considered audio in 1/8th chunks, so one quaver pulse is equal to 0.125 of the output of the (originally used) phasor~ object. An event-based sequencer is used to change clips and set loop lengths within the song, with loops initiated either by the user or automatically when a jamming function or clip change is enacted by the user, with a de-click function used at every 1/8th note boundary, or when user interaction occurs.

There are clear disadvantages to this, in that there are very short dropouts in the audio that distort the material every 1/8th note even if it is playing normally, from the start of the audio array to the end. The phasorbars~ object was created to overcome this, but first, the de-click mechanism is discussed.

2.2.1 Block-Based Click Math

The default DSP block size in Pd is 64 samples. One of the great consistencies of Pd is its 32 bit architecture, so that if I calculate the length of the block in milliseconds it has the same precision as the audio architecture. A

single block of DSP measured in milliseconds is:

$$t = 64 / (SR / 1000)$$

At 44100Hz sample rate (SR), the time (t) is 1.45125ms, a fundamental unit of time within libPd, unless block~ or switch~ objects are used to change this value.

If a change is initiated in the playback point, or the clip is changed, or a mute etc. it happens at the next block-boundary as these events are outside of the audio (signal-driven) tileline, or more precisely a quantized timeline to 64 sample blocks. Given that we are dealing with recorded audio here, there may well be a discontinuity (a click) in the audio stream. Since we cannot go back in time to start the fade before the event occurred, it is necessary to delay the output of the sample reader by a few milliseconds. Then it is possible to fade out the play back for one audio block (1.45125ms). Experimental tests show that delaying the audio stream by 2 blocks (2.9025ms at 44100Hz SR) and fading up after 3 blocks (4.35375ms at 44100Hz SR) is enough to eliminate clicks.

It is a very slight trade-off in turns of sluggishness of the interactions that is barely noticeable. User-selected clips start 4.35375ms after they are triggered. So do sequenced beat-slicing chunks, but since the beat-slicing algorithms are all quantized and automatic any delay is more-or-less irrelevant to the user experience of those functions. Figure 2 shows the delay and vline~ objects in the Voice.pd patch, with the message sent to vline~ when a change occurs shown in figure 3.

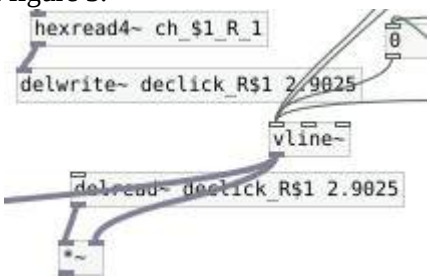


Figure 2. The vline~ object is used to attenuate delayed audio while changes happen to the source array or read point.

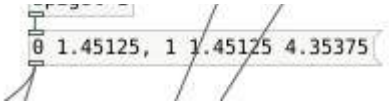


Figure 3. The de-click message sent to vline~, ensuring that discontinuities happen when the track is silenced.

2.2.2 The No-Click Playback Mode

In contrast to all this manipulation of the playback point or source array, it is also possible to play clips from start to finish, in an arrangement specified by a file within the tune pack. If de-click is being used when it is not needed (e.g. when playing an 8-bar loop from start-to-finish) then audio distortion is the result.

The answer is to have two different signal paths within each voice. One where manipulation takes place and an other where a smooth ramp played the audio back with no intervention. In order for this to be completely smooth however, the was a need for an object where the phasor~ signal spans 0 to 1 in 1 bar, but can transition to bar 2, 3, 4 etc. according to the maximum length, i.e. the length of the longest loop. So a transition from 1-bar-at-a-time plus offsets and e.g. 8 seamless bars of playback is made possible by the *~ and +~ objects on either side of the wrap~ object in the Voice.pd patch. Another object has been created to handle the phasor~ signal spanning multiple bars, and sending commands to the sequencing object as well as generating clocks for events. This is the phasorbars~ object discussed below.

3 Phasorbars~, A Single Synchronization Vector for Simple Rhythmic Values

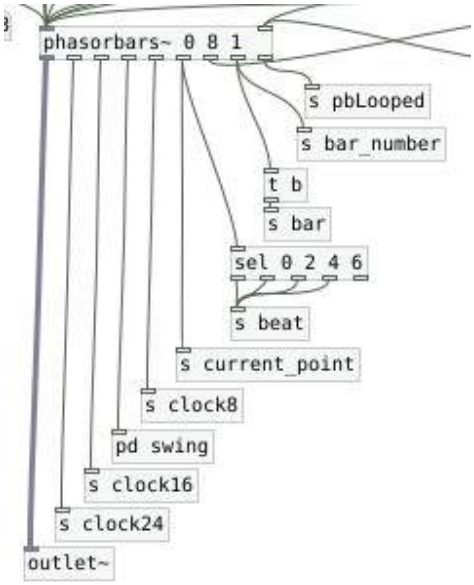


Figure 4. The phasorbars~ object.

Parameters within phasorbars~ are the phase (0-1 just as with the phasor~ object), loop length (how long is the sequence loop) whether the object is looping or not, and the cycle point (the current bar number within the loop). The bar number is incremented on each bar when

loop mode is not initialized, and this feeds into the sequencing object for playback mode. If the app is not looping, the bar number is incremented at the end of the bar offset + loop length, and phasorbars~ is slaved to the sequence which is organized with a resolution of 1 bar. If the object is in loop mode however, the sequence is slaved to the repetition of the loop, itself stored in the sequence object.

The point is that it goes from 0 to 1, 1 to 2, 2 to 3 etc on the signal outlet, while remaining aware of where it is in the sequence (the baroffset and length parameters).

Given that an individual object is used both to set the loop length in loop mode, and also send commands to change the mute, clip and clip offset in sequence mode - the sequence storage object also knows which bar each sample in each of the four tracks is on, fed from the bar number outlet of phasorbars~, so that beat-slicing is accurate when the signal is selected where a 0 to 1 ramp equals one bar. The sequencer sets the parameters for the sample-looping mechanism in phasorbars~, and the sample looping mechanism tells the sequencer when to move on, or to loop.

Its incremental bar counter interacts with the sequencer object to enable both smooth playback modes and the original (slice-based) mode. Its signal output is similar to phasor~ except that instead of a floating-point audio signal that changes between 0 and 1, it then continues the audio ramp through 2, 3 etc. until it reaches the loop length, when it starts again at zero. The only drawback to this approach is that there is a limit to how long an uninterrupted loop can continue, a limit of 4000000 discrete values, which matches with the default maximum length of a sample array. This evaluates to just over 52 bars at 140BPM, 4/4 time signature (or 2^22 rounded down to the nearest million). The point of this is that the chopped audio (with declick at every slice) is only used when some re-organization of the material is happening, and the rest of the time (i.e. when samples are playing from start-to-finish) the phasorbars ramp is playing bar 0, then 1, then 2, then 3 etc on a continuous sample-accurate ramp.

3.1 Voices.pd

Another feature of this object is that it has 1/8th, 1/16th and 1/24th rhythmic outputs that are tied to the progress of the internal phasor~ algorithm. Rapid changes in tempo have no impact on the accuracy of the clocks, since changes in tempo are directly translated into the speed of the phasor~ from which the clock thresholds are taken, and hence quantized changes may occur.

This leads back to the network of pre- and post- wrap~ *~ and +~ objects, and what they represent (see fig. 1). Given that the signal into wrap~ always emerges as a 0 to 1 vector, it can either be a bar with optional beat-slicing, or for example (if multiplied by 0.25) 4 bars with clean playback, depending on what happens on the other end of wrap~. Manipulating the numbers on either side of the wrap~ element allows for complete flexibility with regards to the playback strategy, from a single signal that is modified to read any of the audio arrays from any point.

Having stated this, it was recently discovered that some of the mechanisms in the pre- and post-wrap~ elements are not functioning as they originally were since the Pd patch has been modularized. A click was heard at the end of each bar, where bar lengths in samples were being incremented after the wrap~ object in the signal path, at control-rate.

3.2 wrap_overshoot~

The principle of the wrap_overshoot~ object is founded on the fact that control-rate calculations using +~ or *~ right inlets *after* a wrapped line~ or phasor~ signal do not take effect until the end of an audio block (64 samples). The wrap~ object is mathematically pure in that it allows a positive signal vector to inhabit strict boundaries between 0 and 1. What wrap_overshoot~ does is that when the signal hits the threshold of 1, it will wrap it only at the end of the current signal block. This means that control-rate messages sent into signal arithmetic objects further down the signal path are synchronized with the wrapping of the signal driving the sample readers, and so a slight overshoot (see figure 5) allows the playback of multiple-bar arrays to be click-free.

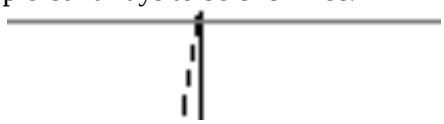


Figure 5. A highly magnified portion of the output from wrap_overshoot~ at a bar boundary.

Some may say this is a kludge, and there are grounds for this supposition. It is often the case with commercial projects that deadlines and external pressures lead to unusual solutions to

problems. There is an argument for a complete re-write of the voice_control.pd mechanism. A window of time is needed for this to happen.

3.3 The linkline patch

The development of Ableton Link[2] has meant that there is a way for multiple devices to be synchronized. Coldcut (Matt Black who manages this project, and Jonathan More) have been using this as a live performance tool for their music in recent months, with linked iPads as performance tools. The phasorbars~ external also has a slave mode, whereby the link phase determines the signal output. Peter Brinkmann's externals for link[5] have enabled the use of link as a synchronization device, and this is quite simple to apply to a sequencing patch or app. However, when playing recorded material there is a problem, since the phase information is sent at control rate.

A patch was devised in a similar way to wrap_overshoot~. Given the block length of 1.45125ms, each block is a ramp from the previous value to the next using the line~ object. When the phase messages from the abl_link external go from high to low values, the low value has 1 added to it as the destination of the line~, and the next block starts at the lower value and goes to the next. The mechanism is shown in figure 6.

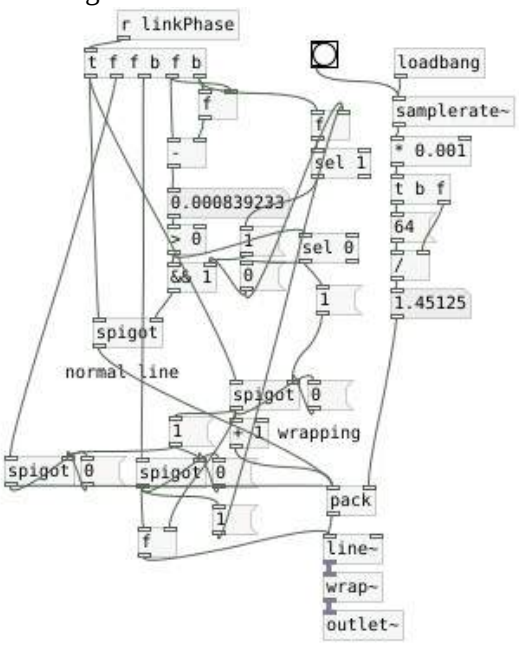


Figure 6. The linkLine.pd abstraction

Returning to the de-clicked beat slicing mechanism, it is possible to determine offsets and multipliers to any point in the bar from the single phasor~ signal if you know where we are.

How we work out the way to that point is another matter.

4 From Where to Here

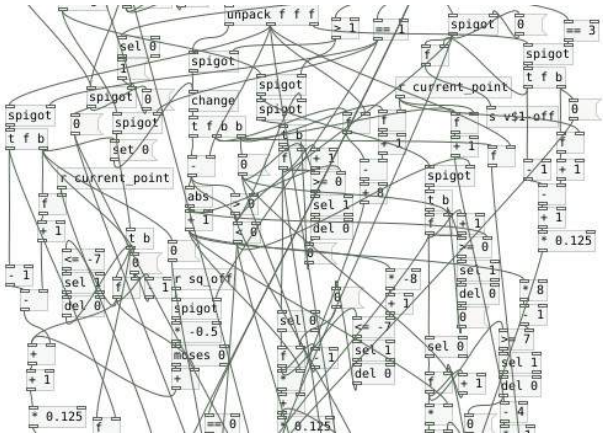


Figure 7. The drill and arbitrary loop mechanisms alone are fiendishly complicated, considering what they do.

There is a way of calculating at any instant the value of an offset to apply to a ramp, provided we know what value the ramp is at in a particular moment. The phasorbars~ object assists us with this task, since its event outlets observe the strict right-to-left ordering of Pd. In fact, with more efficient programming of the patch, perhaps one or more outlets could be eliminated. But the philosophy is that longer structural units happen before shorter ones ,with the sample-playback ramp being the finest resolution.

At any quantized moment, for a given resolution of 1/8th, to offset the phasor~ ramp W (where-we-are) to a point D (destination), an offset A is applied so that:

$$A = 1/8(D - W)$$

Where D and W are the number of 1/8th notes. Of course this value should be wrapped for the offsets to work properly, so that the complete version is:

$$A = 1/8(((D - W) + 8) \% 8)$$

Of course the modulus only applies to integers, and then we take 1/8 of the value, but this is easily ironed-out in patching.

This is deceptively simple, but the realization that this sort of momentary voodoo was necessary caused some angst-ridden moments, to get the CPU load down from 152% to 71% on an iPhone 4!

Its reverse equivalent:

5 https://github.com/nettoyeurny/pd_link_bridge

$$A = 1/8((D - 8) * -1 + 8 - W)$$

can be used when the input signal is running backwards, being multiplied by -1.

5 Missing Artefacts

There is a large amount of detail missing from this article, such as what the sequencer is (a linear database of possible states) or how the sequence object swaps roles with the phasorbars~ object (looping is triggered by user interactions and interventions, line modes are allocated depending on whether to slice). But the point of this explanation and the release of phasorbars~ is not to allow unlimited versions of Ninja Jamm. It is so that these methods can be utilised in ways unpredictable from here.

The future of Ninja Jamm is Jamm Pro, where this becomes a compositional tool for mixing and remixing user-generated material. There are many forthcoming additions to Ninja Jamm that will see the light of day with the version in development. Many aspects not discussed here, such as the inclusion of audio copy and paste functions, load-your-own-sample, elastic audio, snapshots and other customization options that are too far-ranging for a single paper. But the libPd platform has shown its resilience in this app in that the app has existed in a public release for over four years, and its development continues.

6 Conclusion

The techniques discussed here are in some ways obsolete. The rapid growth of mobile devices' speed, memory and overall power means that we can be less frugal with resources than we would otherwise be. However, as this author who grew up in the era of 8-bit computing and 16 colors knows, a little can go a long way.

By understanding the architecture of block sizes in Pd, and clever use of delays and vline~ it is possible to make interactions with recorded audio sound (almost) as smooth and clean as a hard-disk edit, but on-the-fly.

There is great simplicity in the structure of this app, in that it assumes a rigid hierarchy of electronic dance music as it's mode of transmission. However, each of the algorithms and formulas in this paper can be tweaked to suit different situations, such as bar lengths and time signatures, with suitable adaptation of the algorithms published here.

The phasorbars~ and wrap_overshoot~ externals, and the linkLine abstraction are available on my personal website[3].

7 Acknowledgements

Thanks go to Seeper[6] who developed the original interface, and of course to Matt Black and Jonathan More of Coldcut, and the Ninja Tune record label. Special thanks go to the current development team of Christopher Rice and Antoine Rousseau, Peter Brinkmann for the link externals and our pack assembler Aneek Thapar. Also great thanks to Dan Wilcox who develops ofxPd, the branch of libPd used in Ninja Jamm.

References

[1] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, H. C. Steiner: *Embedding Pure Data with libpd*, Weimar, 2011.

[2] https://www.ableton.com/en/link/

[3] http://sharktracks.co.uk/htmlninjaTools.html

6 http://seeper.com

VIS-for-Pd: A Computational Music Analysis Instrument

Reiner Krämer,
McGill University
Montréal, Québec, Canada
reiner@music.org

Abstract

I demonstrate how to use Pd as a computational music analysis tool. To this end, I integrate the VIS Framework (developed for polyphonic music analysis at McGill University), `music21` (developed for computational musicology at MIT), and `pandas` (Python data analysis library) into Pd via the *py/pyext* Python scripting object, resulting in the VIS-for-Pd library. VIS-for-Pd enables users to assemble powerful dataflow music analysis patches without having to write a single line of code. Analytical results are presented as text outputs, and can be saved as `.csv` files, audio examples, score examples, visualizations, and sonifications.

Keywords

Music Analysis, Music Theory, Computational Musicology, Symbolic Music, Big Data.

1 Introduction

In music, some students, scholars, and researchers may already have experience using dataflow programming environments, such as Max, VVVV, OpenMusic, PWGL, or Pd. Most of the time, users build music/art producing instruments. We will examine how to build such an instrument, but for music analysis.

The main goals of VIS-for-Pd[1] are: (1) to have an expandable toolkit with which to create music analysis projects (or instruments) freely available to other music scholars that anybody familiar with dataflow programming languages can expand; (2) to integrate the `pandas`,[2] `music21`,[3] and VIS Framework[4] Python modules that researchers familiar with the Python programming language can expand within the toolkit; and (3) to be able to use these three powerful Python modules for compositional projects.

2. Implementation

At the heart of VIS-for-Pd is Thomas Grill's `py/pyext` object library.[5] The library consists of two objects: (1) "*py*, which loads Python modules and allows to execute functions therein," and *pyext*, which "uses Python classes to represent full-featured message objects."[1] VIS-for-Pd mainly uses the *pyext* object. Binaries of the scripting objects can be downloaded from Grill's web site, but users can also compile the objects from source, currently available at GitHub.[6] In order for Pd to be able to use py/pyext, a user has to specify a path to the binaries in Pd. Grill provides thorough documentation of the object, however knowing how to use *py/pyext* is not necessary in order to use VIS-for-Pd.

Additionally, all Pd objects that have been implemented in Python, have also been wrapped into Pd abstractions, in order to create three different layers of usability, namely (1) the enduser level — a person simply using Pd abstractions to build an analysis patch, (2) a Pd developer — a per-

[1] VIS-for-Pd is available at GitHub (`https://github.com/ELVIS-Project/VIS-for-Pd`).

[2] `Pandas` is a high-performance, data structures and data analysis tool collection for Python (`http://pandas.pydata.org`).

[3] `Music21` is a Python computer-aided musicology toolkit developed by Michael Cuthbert (et al.) at MIT that was supported by grants from the Seaver Institute and the National Endowment for the Humanities (`http://web.mit.edu/music21/`).

[4] The VIS Framework is a Python library, developed at McGill University by Christopher Antila, Jamie Klassn and Alexander Morgan for the analysis of contrapuntal music (`https://elvisproject.ca`). The development of the library was enabled the by *Digging into Data Challenge* (`http://diggingintodata.org`) award, which was funded by SSHRC — Social Sciences and Humanities Research Council — in Canada and Jisc — Joint Information Systems Committee — in the United Kingdom. The VIS-Framework currently is maintained by Claire Arthur, M. Ryan Bannon, and Reiner Krämer at the SIMSSA (Single Interface for Music Score Searching and Analysis) project (`https://simssa.ca`). The VIS Framework can be installed with the `pip` Python package manager, and can be cloned from GitHub (`https://github.com/ELVIS-Project/vis-framework`).

[5] Python scripting objects for Pure Data and Max (`http://grrrr.org/research/software/py/`).

[6] `https://github.com/grrrr/py`.

son wishing to "tweak" abstractions to their own specific analytical use case, and (3) a Python/Pd developer — a person wishing to be able to build custom solutions to target specific analytical tasks. Let us examine why `music21` forms the basis of the VIS Framework.

2.1 `music21`

One of the main reasons why `music21` was used as a foundation for the VIS Framework, is its ability to import many different types of music data and symbolic score representations, like ABC, Capella, Humdrum, LilyPond, MEI, MIDI, MusicXML, MuseData, and others. [7] Another reason is `music21`'s pitch handling capabilities. `Music21` can convert pitches to and from many different naming schemes, e.g.: *cis4* to C#4 to pitch class 1 to MIDI pitch 61 to 277.18 Hz poses no prob-lems. The same is true for intervals, intervallic relationships, and their varying representative ex-pressions.

2.1.1 Parse Symbolic Music

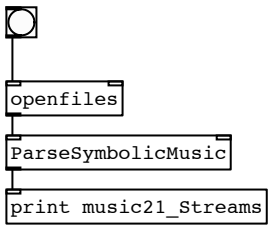


Figure 1: `ParseSymbolicMusic.pd`.

The `ParseSymbolicMusic.pd` abstraction wraps `music21`'s converter class into a Python object for Pd, meaning that symbolic music files are made available as `music21` streams (the intermediary storage format for symbolic music in `music21`) in the Pd workspace, or patch. It's companion object is the `OpenFiles.pd` object, which enables the user to open one or more files via a `bang` into its left inlet. The object also stores paths to the files as a list in case the symbolic files need to be passed to another object at another time, without having to re-open a file or folder. A `bang` to the right inlet recalls file path(s), and delivers them to the object's oulet. The `ParseSymbolicMusic.pd`, besides from converting symbolic music files to `music21` streams also caches the streams, so that they are available to other objects at a later time.[8] Figure 4 shows how the objects look when used in a Pd patch.

2.1.2 Count Pitches

When we connect the output of the `ParseSymbolicMusic` object (a symbolic file of the monophonic chant *Ave Maria ... Benedicta tu* has previously been selected with the `openfiles` object) to the second inlet of the `CountPitches` object, `music21` counts the pitches and creates a pitch profile histogram of the monophonic chant for us in the Pd window (as shown in Figure 4).[9]

```
Anonymous: Ave Maria ... Benedicta tu
-------------------------------------
F3: 1
G2: 2
A2: 8
B2: 4
C3: 10
D3: 12
E3: 7
```

Figure 2: Output of the `CountPitches` object.

The pitch label specifications can be changed in real time by passing the following messages to the third inlet: `frequency` (as in 123.470825314 Hz), `name` (as in 'F' without an octave specification), `nameWithOctave` (F3), `pitchClass` (6), `midi` (53), etc. The fourth inlet accepts `count` and `pitch` messages as arguments, which sort the output either by count, or by pitches in ascending order.

Currently, two other `music21` classes have been implemented in Pd namely the `Ambitus` object, which displays the range of pitches within a part (or voice) of a piece from the lowest to the high-est pitch, and the `CountMelodicIntervals` object, which provides a histogram of the different horizontal intervals devised from a part in a composition. Both of these objects are similarly structured with three to four inlets inlets and one outlet. Data lists as displayed in the Pd window (see Fig-ure 4) are directed toward the outlet, and are re-usable for compositional purposes and further data wrangling. There are hundreds of more `music21` classes that can be implemented as the ones shown

[7] "Symbolic" here simply means the use of a combination of alphanumeric characters to represent "traditional" music notation. For an exhaustive list of filetypes and formats visit: `http://web.mit.edu/music21/doc/moduleReference/moduleConverter.html#music21.converter.Converter.defaultSubconverters`.

[8] Currently the `music21` streams are being stored as pickled files on a file local directory, but a memoization scheme is in the works.

[9] Although only one file is shown here, we can actually choose many files at once.

above. However, the main focus is not the implementation of `music21` classes, but the integration of the VIS Framework within Pd.

2.2 VIS Framework

As previously stated the VIS Framework uses `music21` to import symbolic music scores. Once a set of scores has been imported they are then placed into a `pandas` DataFrame by use of the `NoteRestIndexer` class. `Pandas` has several data structures of which we will use the Series, a one-dimensional labeled array that can hold any data type, and the DataFrame, a two-dimensional labeled data structure with columns that can hold any data type as well. The columns are composed of Series.

The VIS Framework has two types of analyzer classes, namely the indexers, of which the `NoteRestIndexer` is one, and experimenters, which begin with a DataFrame generated from one of the indexers. Our focus for VIS-for-Pd is on the implementation of indexers.[10] Let us examine the `NoteRestIndexer` implementation in VIS-for-Pd.

2.2.1 The `NoteRestIndexer`

```
Anonymous: Magnificat anima mea
-------------------------------
Score   Part
Events  0

0.0     C3
1.0     D3
2.0     E3
3.0     E3
4.0     E3
```

Figure 3: A monophonic DataFrame generated by the `NoteRestIndexer` object.

As was the case with the `ParseSymbolicMusic` object, the `NoteRestIndexer` Python class has been implemented with *py/pyext* as a Pd object, and has then been wrapped into a Pd abstraction. The object receives the output of the `openfiles` object in its first inlet. The conversion of symbolic files to a `music21` stream has been built into the `NoteRestIndexer` class, so that we do not need to use the `ParseSymbolicMusic` object first. The DataFrame for a monophonic chant generated by the `NoteRestIndexer` looks like Figure 3. The DataFrame consists of one Series. It is abbrevi-ated to just the first five musical events that occur by default.

However, we can change the settings of the `NoteRestIndexer` results in real time. We can scroll through all of the events incrementally by changing the `Start` (fifth inlet), `End` (sixth inlet, first part of a tuple), and `Interval` (sixth inlet, second part of a tuple) settings, and we can spec-ify whether we would like to read the results from the `Beginning` or the `End` (fourth inlet) of a composition analyzed (Figure 4).

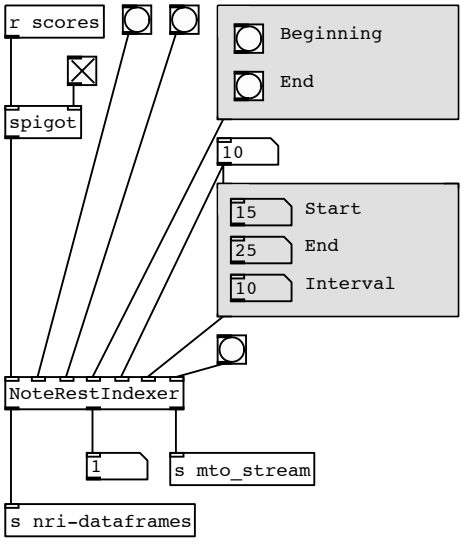


Figure 4: The `NoteRestIndexer` object.

The `NoteRestIndexer` object caches its resulting DataFrames, as to be able to access them any time during the analytical process, without having to re-build them.[11] The process saves time and computing power, which becomes relevant when handling larger amounts of data. Locations of the cached DataFrames are routed through the first outlet. The DataFrames can be re-routed by `bang`ing the second inlet of the `NoteRestIndexer` object. The second outlet shows how many scores have been processed (the number can be re-called anytime by supplying a `bang` to the third inlet), while the third outlet points to the cached `music21` stream that has been generated by the `NoteRestIndexer` object as well.

The `NoteRestIndexer` can index polyphonic pieces as well, in which case for each additional voice another Series is added to the DataFrame

[10] Experimenters vary from analysis project to analysis project, whereas indexers function at a bigger abstraction level. You can learn more about the design principles of the VIS Framework in its API documentation: `http://vis-framework.readthedocs.io/en/latest/about.html`.

[11] As was the case with caching `music21` streams, the DataFrames are currently cached in a local directory, but the caching process will be migrated to memoization

(Figure 5). The "Part 0" columns features several "^" symbols, indicating that a value carries over from the previous occurring value.

```
Jacob Obrecht: Monad   Aeolian
------------------------------
Score   Part    Part
Events  0       1

0.0     Rest    E3
4.0     Rest    F3
6.0     E3      G3
7.0     ^       E3
9.0     ^       C3
10.0    F3      D3
12.0    G3      C3
13.0    E3      ^
14.0    ^       Rest
15.0    C3      C4
```

Figure 5: A polyphonic DataFrame generated by the `NoteRestIndexer` object.

2.2.2 Other VIS Framework Classes

```
Jacob Obrecht: Monad   Aeolian
------------------------------
Score   Part    Part
Events  0       1

0.0     Rest    m2
4.0     Rest    M2
6.0     m2      -m3
7.0     ^       -M3
9.0     ^       M2
10.0    M2      -M2
12.0    -m3     Rest
13.0    -M3     ^
14.0    ^       Rest
15.0    M2      -m2
```

Figure 6: A horizontal interval DataFrame generated by the `HintIndexer` object.

Other objects that have been implemented in VIS-for-Pd are the `HintIndexer` (short for `HorizontalIntervalIndexer`), the `VintIndexer` (short for `VerticalIntervalIndexer`), the `OffsetIndexer`, and the `NGramIndexer`.[12] Both the `HintIndexer` and the `VintIndexer` take cached DataFrames generated by the `NoteRestIndexer`, and calculate various intervallic relationships. Alternatively, these indexer can also take output from the `openfiles` object directly.

The `HintIndexer` determines horizontally occurring intervals in a voice or part successively (Figure 6). The `VintIndexer` calculates vertical interval between voices or parts. The `VintIndexer` considers vertically occurring intervals amongst all voice combinations so that a composition with four voices will yield six vertical intervals, if the four voices are simultaneously sounding. The DataFrames generated by both interval indexers are cached as well. Both of the indexers are also structurally similar to the `NoteRestIndexer`.

The generated DataFrames of both horizon-tal and vertical interval indexers can be used to populate the `NGramIndexer`. Christopher Antila and Julie E. Cumming describe the output of the `NGramIndexer` in detail, and explain and how the `NGramIndexer` excels at finding *contrapuntal modules*, which are "repeated contrapuntal patterns made from a series of vertical (harmonic), and horizontal (melodic) intervals," in short, *n*-grams. [2][3][4][13]

The `OffsetIndexer` can calculate different offset intervals at which certain modules can occur, since harmonic and contrapuntal rhythms may move at much slower paces than other parts of the music. The `OffsetIndexer` can discover large scale contrapuntal modular patterns, can be used to filter out "discovered" modules, and can be used to identify false positives that happen at very short time intervals and may have important local musical roles, yet assume no significant roles as far as larger structural modules are concerned.

The VIS Framework features other indexers that have not yet been implemented in VIS-for-Pd, but will be implemented in future. Amongst them are: the active voices indexer, the cadence indexer, the contour indexer, the dissonance indexer, the fermata indexer, the LilyPond indexer, the meter indexer, the over bass indexer, the repeat indexer, and the window indexer, or "windexer," for short.

3 Working with Results

Analytical results can be processed through `music21`, for example by displaying score examples with `music21`'s `.show()` method, or by using

[12] The prefixes *Hint* and *Vint* are used in honor of David Huron's much beloved *humdrum* music analysis toolkit (`http://www.humdrum.org`).

[13] A Renaissance cadence pattern is such a module, combining elements of the *clausula vera*, and additional horizontal and vertical properties.

various visualization methods.[14] The VIS Framework offers its own score visualization through LilyPond, but results can also be re-mapped to MEI and displayed with a web-based tool such as Verovio.[15] Results from analyses can also be plotted directly through `pandas` in conjunction with the `matplotlib` Python module.[16] Web-based visualizations can be realized with the `D3.js` library.[17] Results can also be processed and fur-ther evaluated through the Python machine learn-ing module `scikit-learn`[18] and the deep learning tensor flow module developed by google.[19] And finally results can also be auralized, either by merely playing music examples of found modules, or by sonifying data in compositional processes.[20]

4 Future Considerations

Three versions of Pd are currently available to users: (1) Pd-L2Ork (Virginia Tech University),(2) Pd — "Vanilla", Miller Puckette's version, and (3) the now defunct Pd-extended.[21] A version of Pd, based on Pd-L2Ork, called *purr-data* has been ported to a HTML5 based GUI. The devel-opment opens up new possibilities for VIS-for-Pd. I was able to successfully build a version of *purr-data* that included Thomas Grill's *py/pyext*, which means that in the not too distant future VIS-for-Pd can become a dataflow music analysis programming environment for the web.

5 Conclusion

Why do we want to use Pd to build computational music analyses tools? Pd is a visual programming tool, in which complex programming grammar visually interconnects encapsulated objects into pictograms. This has the advantage the we can think about music analysis programming in a different way, enabling us to re-evaluate the VIS Framework and its modular structure. Pd has a large user base that includes many types of musicians and code enthusiasts. In effect, more programmers could get involved in the development of VIS-for-Pd. Pd enables users to quickly prototype new objects and test them in real time, especially with the `py/pyext` objects. Analytical results can be observed in real time, allowing the analyst to play with data. Everything developed in Pd is potentially portable to the web, as demonstrated by *Purr Data*. With Pd we can re-appropriate `pandas`, `music21`, and the VIS-Framework for compositional purposes. But most importantly using Pd is fun.

6 Acknowledgements

I would like to express my gratitude to Julie E. Cumming, Ichiro Fujinaga, the Schulich School of Music's *Single Interface for Music Score Searching and Analysis* (SIMSSA) project and the *Distributed Digital Music Archives and Libraries Lab* (DDMAL) at McGill University, the *Social Sciences and Humanities Research Council of Canada* (SSHRC), the *Centre for Interdisciplinary Research in Music Media and Technology* (CIRMMT), and the *Fond de recherche Société et culture Québec*.

References

[1] Thomas Grill: "py/pyext — Python scripting objects for Pure Data and Max," `http://grrrr.org/research/software/py/`.

[2] Christopher Antila and Julie Cumming: "The VIS Framework: Analyzing Counterpoint in Large Datasets," *15th International Society for Music Information Retrieval Conference, 2014*.

[3] J. A. Owens: *Composers at Work: The Craft of Musical Composition 1450–1600*, Oxford University Press, 1997.

[4] J. Cumming: From two-part framework to movable module. In Judith Peraino, ed., *Medieval music in practice: Studies in honor of Richard Crocker*, pp. 177-215. American Institute of Musicology, 2013.

[14] For a detailed list visit: `http://web.mit.edu/music21/doc/moduleReference/moduleStream.html?highlight=plot#music21.stream.Stream.plot`.

[15] `http://www.verovio.org/index.xhtml`.

[16] The `pandas` documentation outlines multiple procedures: `http://pandas.pydata.org/pandas-docs/version/0.18.1/visualization.html`.

[17] `https://d3js.org`.

[18] `http://scikit-learn.org/stable/`.

[19] `https://www.tensorflow.org`.

[20] Sonifications of search results generated by the VIS framework have already been implemented in SuperCollider: `https://elvisproject.ca/sonification/`.

[21] `https://git.purrdata.net/jwilkes/purr-data`.

FFTease and LyonPotpourri: History and Recent Developments

Eric Lyon
Institute for Creativity, Arts, and Technology
School of Performing Arts
Virginia Tech
ericlyon@vt.edu

Abstract

This paper curates two collections of externals originally created for both Max/MSP and Pure Data (Pd) at a time before the coding protocols of the two programs started to significantly and increasingly diverge. The current distributions of these two packages for Pd were created to finally separate the Max/MSP code from the Pd code. We will focus on some of the functionalities of this software that are not easily obtained by combining other Pd objects. Some of the more recent tools are especially suited for spatial composition, through arbitrary panning schemes, or by spectrally fractionating an incoming sound, for spectral diffusion to multiple loudspeakers. Several different spectral processors are also introduced.

Keywords

Spectral Spatialization, Spectral Processing, Computer Music, Pure Data externals.

1 The Original Externals Packages

Coding for both FFTease and LyonPotpourri commenced in 1999. FFTease was a collaboration project between Christopher Penrose and the author [1]. LyonPotpourri was written independently by the author. FFTease was first released in 1999 and LyonPotpourri was first released in 2006. Both packages were initially written exclusively for the Max/MSP platform. In 2003 I took over responsibility for maintenance and development of FFTease. At around the same time, at the invitation of Richard Boulanger, I began writing about the process of developing Max/MSP externals, a text that was originally intended to be part of *The Audio Programming Book* [2], but eventually became *Designing Audio Objects in Max/MSP and Pd* [3] once the size of the text could no longer comfortably fit within *The Audio Programming Book*. As I began working on my pedagogical text, I noticed that in many cases, the "perform" loop, which is the DSP callback routine executed on each signal vector, was identical across Max/MSP and Pd. For example, Figure 1 shows the perform routine for a LyonPotpourri external called *waveshape~*. This perform routine can work without modification in both a Pd external and a Max 4 external. This great similarity between Max/MSP and Pd "under the hood" was very suggestive, and it seemed to me at the time that a port of both FFTease and LyonPotpourri to Pd would be both relatively easy, and of some value in sharing the functionality of these objects with the Pd community.

```
t_int *waveshape_perform(t_int *w)
{
     float insamp; // , waveshape, ingain ;
     int windex ;
     t_waveshape *x = (t_waveshape *) (w[1]);
     t_float *in = (t_float *)(w[2]);
     t_float *out = (t_float *)(w[3]);
     t_int n = w[4];
     int flenm1 = x->flen - 1;
     float *wavetab = x->wavetab;

     if(x->mute){
          while(n--){
               *out++ = 0.0;
          }
          return w+5;
     }
     while (n--) {
          insamp = *in++;
          if(insamp > 1.0){
               insamp = 1.0;
          }
          else if(insamp < -1.0){
               insamp = -1.0;
          }
          windex = ((insamp + 1.0)/2.0) * (float)flenm1;
          *out++ = wavetab[windex];
     }
     return w+5;
}
```

Figure 1 Perform routine for a waveshaping algorithm that executes on both Max and Pd

1.1 Early Similarity of Max and Pd APIs

While the coding API for Max/MSP and Pd externals was similar in the first decade of the 21st century, it was not identical. For example, although the functionality of inlets and outlets was similar, the implementation was different. Figure 2 shows an initialization code routine in which differences between Max/MSP and Pd are managed through #ifdef statements. Before adopting the somewhat crude solution of using #ifdef statements to manage the differences between Max/MSP and Pd, Flext [4], was considered as potentially offering a somewhat more elegant solution to cross-platform development. However, in the end I decided not

to use Flext, since I did not want to introduce another layer of dependency to my code where it was not strictly necessary.

```
void *waveshape_new(t_symbol *s, int argc, t_atom *argv)
{
#if __MSP__
   t_waveshape *x = (t_waveshape
*)newobject(waveshape_class);
   dsp_setup((t_pxobject *)x,1);
   outlet_new((t_pxobject *)x, "signal");
#endif
#if __PD__
 t_waveshape *x = (t_waveshape *)pd_new(waveshape_class);
 outlet_new(&x->x_obj, gensym("signal"));
#endif
     x->flen = 1<<16 ;
     x->wavetab = (float *) t_getbytes(x->flen *
sizeof(float));
     x->tempeh = (float *) t_getbytes(x->flen *
sizeof(float));
     x->harms = (float *) t_getbytes(ws_MAXHARMS *
sizeof(float));
     x->hcount = 4;
     x->harms[0] = 0;
     x->harms[1] = .33;
     x->harms[2] = .33;
     x->harms[3] = .33;
     x->mute = 0;
     update_waveshape_function(x);
     return x;
}
```

Figure 2 An instantiation routine with code divergence between Max/MSP and Pd

2 Max and Pd get a Divorce

Certain additions to Max/MSP functionality in the Max 5 and Max 6 releases suggested the need to revaluate the use of a shared codebase for FFTease and LyonPotpourri. Max 5 introduced attributes, a mechanism for maintaining state within an object when a patch is saved and closed. Max 6 introduced 64-bit processing. The latter development resulted in a different interface for the perform routine that would require maintaining separate perform routines for Max/MSP and Pd going forward. At that point it was no longer trivially easy to keep the Max/MSP and Pd versions of FFTease and LyonPotpourri unified, so the next releases of FFTease 3.0 and LyonPotpourri 3.0 were separated into independent Max/MSP and Pd versions, with all of the #ifdefs stripped out.

3 Origins of FFTease

FFTease started as a collaboration project with Christopher Penrose in 1999 in order to facilitate experimentation with spectral processing on the Max/MSP platform. The development of the first FFTease externals preceded the introduction by Cycling '74 of the *pfft~* system, which was introduced to Max 4 in 2000. *Pfft~* is a system that facilitates exploration of spectral processing on Max/MSP. In 1999, it was still rather complicated to program FFT-based processors using existing Max objects to implement windowing, double buffering, overlap-add, phase unwrapping, and oscillator banks. But at the time, Penrose and I had extensive experience writing Unix-based C-code to implement all of those features, based largely on code for the phase vocoder introduced by F. Richard Moore in his book Elements of Computer Music [5]. And both of us had already released non-realtime FFT-based software to run on Unix, PVNation in Penrose's case [6] and POWERpv in mine [7]. Coming from this perspective, it seemed natural to write several monolithic externals to accomplish different FFT-based processing tasks that we found interesting, but felt would be too arduous to program in the visual data-flow programming language of Max/MSP.

3.1 Real-Time Architecture for FFTease

Transitioning from the non-real-time world of Unix software to the real-time environment of Max/MSP was easy at first, largely because we first went with a sub-optimal, but easy-to-code solution. Implementation of spectral processing generally requires an enveloped overlap-add scheme, which requires some form of double buffering. Other than the sampling rate, key parameters for FFT-based processing include the FFT size, and the overlap factor. For the FFT size, higher values yield better spectral resolution at the cost of higher CPU demand, and lower time resolution. The overlap factor determines how many samples to slide over the input signal for each FFT frame, known as the "hop size" and calculated as (N/o) where N is the FFT size and o is the overlap factor. Higher overlap factors reduce artifacts of windowing, at the cost of proportionally higher CPU usage.

In the original architecture of FFTease, we set the hop size equal to the signal vector size, which considerably simplified real-time calculations, since we could then calculate a new FFT on each perform routine callback. The overlap factor was simply the FFT size divided by the signal vector size. This architecture had the advantage of allowing us to quickly code up real-time FFT-based processing in Max/MSP. At the same time, an obvious downside to this architecture tied performance of the objects to the signal vector size. Changing the signal vector size would change the behavior of the object. Nonetheless, this design flaw remained

in place until 2005 when I revised FFTease to implement a double buffering scheme independent of the signal vector size. This was during the time period that I began to port both FFTease and LyonPotpourri to Pd, so the Pd version of FFTease was always based upon the later, improved Max version of FFTease.

3.2 Resynthesis options for FFTease

Most of the FFT operations performed on audio signals are so-called "real" FFTs. This is because audio signals are real rather than complex, so if represented as complex numbers, the complex component would always be zero. For FFT analysis of real signals, Pd provides the *rfft~* and *rifft~* objects. (*fft~* and *ifft~* are also provided for the analysis of complex signals.) Although the input signal to an fft~ object is real, the output is complex, representing the weights and phases of a harmonic series based on a fundamental frequency of analysis that is the sampling rate divided by the FFT size. This intermediary result can be useful for cross-synthesis of two analyzed signals, but more commonly another step is taken to convert the complex numbers to a polar representation of the amplitudes and phases for each harmonic. At this stage, any number of FIR filters may be applied by altering the amplitudes. Since the FFT analysis is being applied constantly, the filters being applied could be time varying. Once the amplitudes have been suitably altered, the resulting spectrum is converted from polar back to a complex format, and then converted back to a real audio signal with the efficient inverse FFT (IFFT), implemented in Pd by *~ifft*.

In addition to the IFFT, which allows for amplitude modifications of a spectrum, it is also possible to make estimates of the instantaneous frequencies in each bin, with a process called phase unwrapping. The process is described in Moore's book and is covered in greater detail in Mark Dolson's Phase Vocoder Tutorial. [8] Once frequency modifications have been made, it is generally necessary to implement resynthesis with an oscillator bank, rather than an IFFT. Most or all of this calculation could be done using existing Max/MSP or Pd objects. However the additional complexity would detract from the ease of use that was intended from the start for FFTease. In practice, the monolithic FFTease objects seem beneficial both for the interesting spectral processing that they provide, and the ease with which they can be integrated into Pd patches.

3.2 FFTease and pfft~

In 2000, Cycling '74 introduced the *pfft~* system. This system wrapped up an FFT/IFFT with windowed overlap-add. After absorbing the implications of *pfft~*, Penrose and I rethought FFTease in two ways. First, we thought that there was a good chance the entire project might be rendered obsolete, and be subsumed by further spectral innovations from Cycling '74. Second, we decided to jump on the bandwagon, and created a *pfft~*-centric version of FFTease called FFTease Lite. For FFTease Lite, we used the *pfft~* interface, and only wrote C code to process spectra in the amplitude/phase format. The advantage to this approach is that FFTease Lite processors could be stacked in succession without requiring conversion back to the time domain, as would be the case for the monolithic FFTease objects. However *pfft~* did not easily allow for oscillator bank resynthesis. And the convenience of monolithic objects remained compelling, so FFTease Lite was scrapped. But the *pfft~* idea remains valid. It may seem strange to discuss a Max-centric feature in a Pd-centric article, but the pfft~ option can be utilized in Pd as well. In the spectral chapter of "Designing Audio Objects for Max/MSP and Pd," [9] I showed how to implement the *pfft~* idea in Pd using a few additional utility externals. All of those externals have been added to LyonPotpourri, so a bit later in this paper, we will show how to use those externals to implement a pfft~-like processor in Pd.

4 Working with FFTease

FFTease is a portmanteau word, combining "FFT" with "ease," two words that did not ordinarily go together when Penrose and I started writing the collection. However, the monolithic structure of these externals makes them quite easy to use. The main limitation is that the user is limited to the algorithms we chose to implement, whereas in the pfft~ model, there is more flexibility to code up new spectral algorithms. At the time that the first FFTease objects were written, the monolithic approach was not the norm. Rather there was a focus on making smaller, more focused objects that could be combined to create more complex functionality. But a key design goal for FFTease was to provide powerful new functionality, while hiding most of the implementation details, at the expense of some end-user flexibility.

4.1 Pvoc~

Figure 3 shows a simple Pd patch employing the FFT object *pvoc~*. To simplify our examples, the input signal to the FFTease processor is a non-band-limited sawtooth wave. In more characteristic use cases, we would process a more complex signal such as live microphone input, or a sound file with rich spectral features. *Pvoc~* is largely based on the phase vocoder algorithm described by F. R. Moore in Elements of Computer Music. Since the object alters the instantaneous frequencies of the harmonics of the sound, an oscillator bank is required for resynthesis. The primary function of pvoc~ is to change the pitch of an incoming sound without altering its duration, as would be the case when playing back a sound file at a speed other than 1.0. This parameter is controlled by a number send into the second inlet of *pvoc~*. A value of 2.0 raises the pitch of the input signal by an octave. In order to function properly, *pvoc~* only requires two inputs: an input signal in its leftmost inlet, and a transposition value in its middle inlet, which could be either a float or a signal. It is difficult to imagine an object that would be easier to operate.

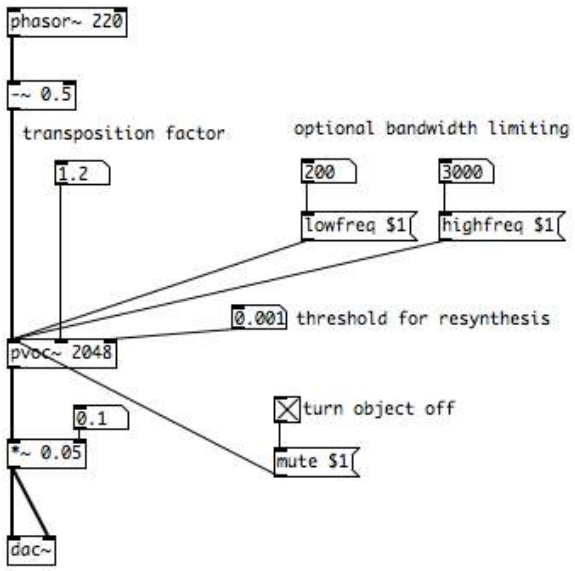


Figure 3 Using the FFTease object *pvoc~*

4.1.1 Refinements to pvoc~

Pvoc~ takes as an optional argument the FFT size, which must be a power of 2. If not provided, the default FFT size is 1024. The overlap size is fixed at 8. Moore's original phase vocoder algorithm provides a threshold for analysis. This is an optimization feature, to ameliorate the expense of an oscillator bank, which is usually considerably more CPU-intensive than an IFFT. By setting a threshold on a per-frame basis, any harmonic with a weighting that falls below the threshold is not resynthesized. We have provided a further refinement, so that the threshold is multiplied by the maximum amplitude of each frame to generate that frame's synthesis threshold, making the threshold adaptive. In Figure 3, the threshold is set to 0.001 or roughly -60dB below the maximum amplitude. Raising this threshold will significantly reduce the CPU-load of the object, but artefacts will quickly become audible as different parts of the spectrum rapidly cut in and out of the sound. These artefacts might be musically useful in some cases.

Another refinement to *pvoc~* allows the user to set both the low frequency and high frequency of resynthesis. Use of these parameters can dramatically reduce the CPU-load of the object. This also introduces the capability to apply a very sharp bandpass filter to the output, extending the range of sonic possibilities for the object. Finally, as a convenience for the user, the entire object can be muted, cutting off all FFT calculations. All FFTease objects respond to the "mute" message

4.2 Pvwarpb~

Pvoc~ implements a completely standard use of FFT processing. However the oscillator resynthesis model used in pvoc~ can easily be subverted to yield more interesting results than simple transposition. Figure 4 shows the use of FFTease external *pvwarpb~*. This external uses an oscillator bank for resynthesis, but unlike pvoc~, each individual oscillator can have a different transposition factor. These transposition factors are exposed to the user through a Pd array, into which the user can draw new values directly. The message "autofunc" generates a new line-based warp function with minimum and maximum values as argument. The warp function can be set back to all 1.0 with the Pd message "const." The warp function can be accessed by any method for addressing arrays that is available in Pd. In addition to the warp function, the output can be transposed, as accessed in the third inlet. The synthesis threshold generator may be entered in the fourth inlet. An offset to the warp function may be set in the second inlet. This object can powerfully bend spectra out of shape.

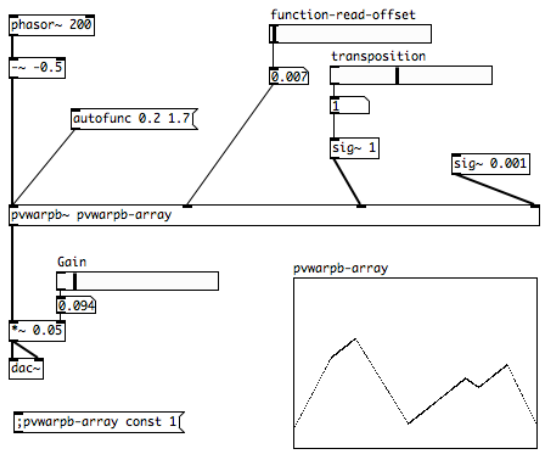


Figure 4 Using the FFTease object *pvwarpb~*

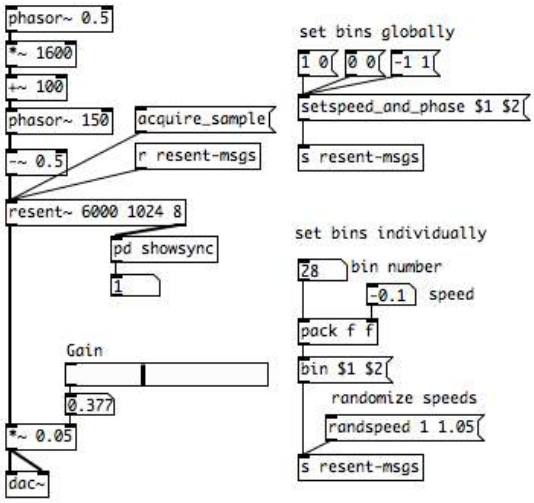


Figure 5 Using the FFTease object r*esent~*

4.3 Resent~

The last FFTease object that we will discuss in detail is called *resent~*. Just as *pvwarpb~* extends the functionality of *pvoc~*, *resent~* extends an FFTease object called *resident~* that implements arbitrary time scaling of a spectrally sampled sound. Spectrally sampling a sound consists of storing a series of short time FFT frames. Since each frame could theoretically be extended infinitely in time, a series of frames can be resynthesized in any speed and order. Moving linearly through a series of frames at half speed results in doubling the duration of a sound without altering its pitch. This is what *residency~* does. *Resent~* adds the capability for each individual bin of an FFT to move at a different speed, breaching the integrity of individual FFT frames, an effect that cannot be accomplished by *residency~*. For example, the high part of a sound can be moving forward while the lower part is moving backwards. Or each bin can move at a different, random speed. Or some parts of the spectrum could move at a glacially slow speed, while others move blindingly fast.

The use of *resent~* is shown in Figure 5. The required first argument is the buffer size in milliseconds. Following that are two optional arguments, the FFT size, which defaults to 1024, and the overlap factor, which defaults to 8. The message "acquire_sample" is used to record the input sound as a series of FFT frames. Once the recording is complete, resynthesis is determined according to the speeds of each individual bin. The speed and phase of playback can be set globally with the "setspeed_and_phase" message. More interestingly, the speed of each individual bin can be set with the "bin" message, or randomized within minimum and maximum speeds with the "randspeed" command.

4.4 Other FFTease Objects

A full exegesis of the FFTease package is beyond the scope of this paper. I will however call attention to a few of the externals I consider most interesting and unusual. *Pvtuner~* maps the frequency content onto a user-specified scale, which is completely arbitrary, and can be specified as a list of frequencies. A large number of pre-defined scales are also available. I used pvtuner~ in my 2001 composition Sacred Amnesia [10] to tune an excerpt from the first movement of Schoenberg's *Pierrot Lunaire*, a famous atonal composition, to A-major. *Dentist~* is an external that randomly designs filters with spikes at particular frequencies, and interpolates between these filters. *Reanimator~* performs a kind of audio texture mapping, where one sound file provides a bank of FFTs, and then a second sound file is reconstructed by real-time lookup of the FFTs from the first file. There are many other idiosyncratic FFTease externals to explore.

5 LyonPotpourri

LyonPotpourri was first released in 2006, though many of its externals were created for personal use considerably earlier. The functionalities of the externals are considerably more disparate than for FFTease, and as with FFTease, a full discussion of all of the LyonPotpourri externals is not possible within the scope of this paper. There are three major groups of externals within the collection. The first group is organized around the principle of sample-accurate timing. This group of externals

has been previously discussed in [11]. We introduce here, two recently added groups of externals focused around first, a reconstruction of the Max pfft system, and second, a group of externals intended for spatial composition for arbitrary numbers of loudspeakers.

5.1 The pfft~ System and Pd

In 2000, Cycling '74 introduced a system called *pfft~* to Max 4. This system greatly simplified the problem of implementing spectral processing, taking care of enveloping, overlap-add, and conversion in and out of the frequency domain. Had this system been introduced a year earlier, it is possible that Penrose and I would not have written FFTease. In its essential working, an abstraction is produced that incorporates pfft~ objects, and does further internal processing in the frequency domain. This abstraction is then introduced as an argument to the *pfft~* object, in which FFT size and overlap factor are specified as arguments. When writing "Designing Audio Objects for Max/MSP and Pd," I needed to reconstruct the functionality of *pfft~* for Pd. The externals designed for this need were introduced into LyonPotpourri 3.0 in 2016.

5.1.1 Max and pfft~

Figure 6 shows a typical pfft~ abstraction, implementing a high pass filter. The amplitudes for all frequencies below the selected bin are set to zero, resulting in a very sharp filter. The cutoff frequency for the selected bin is determined by the formula (b * r / N), where b is the bin number, r is the sampling rate, and N is the FFT size.

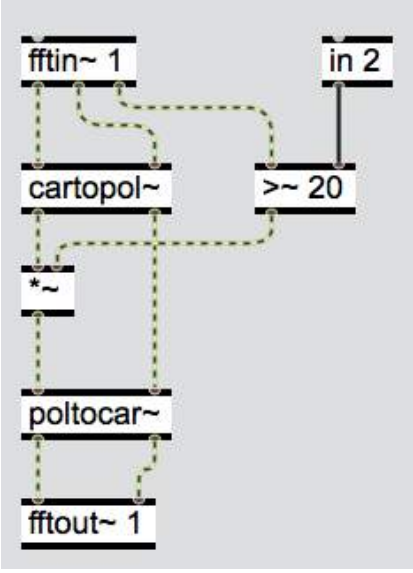


Figure 6 A pfft~ abstraction

The trick here is to compare the current bin number to the number sent in from the second inlet. The comparison is done with the >~ object. The third outlet from *fftin~* is a sample-accurate index that counts the current bin from zero. This index signal is an essential component of the *pfft~* system that allows for bin-specific operations. Figure 7 shows the deployment of this pfft~ abstraction in a Max Patch.

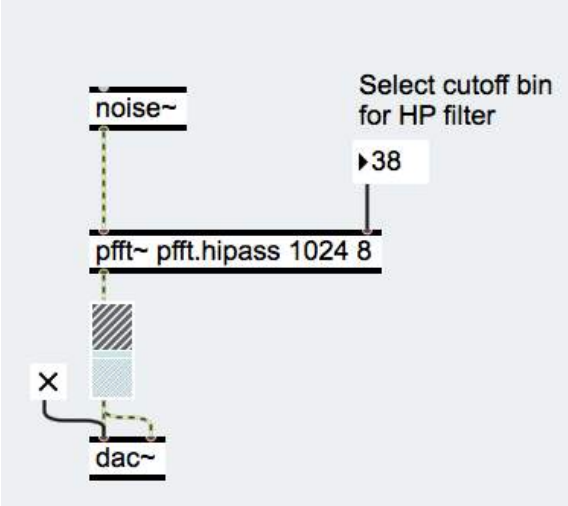


Figure 7 Using a pfft~ abstraction in Max

5.1.2 Replicating pfft~ Functionality in Pd

In order to replicate the basic functionality of *pfft~*, the following externals were introduced to LyonPotpourri: *windowvec~*, *cartopol~*, *poltocar~*, and *vecdex~*. These objects do not need to be deployed in an abstraction, but can be used directly in a Pd sub-patch. The FFT size and overlap are determined in the sub-patch with the use of the Pd object *block~*. The use of these objects to replicate the functionality of the Max pfft~ abstraction of Figure 6 is shown in Figure 8. A Hann window is applied at both input and output stages with the windowvec~ object. The index of the current bin is provided by the *vecdex~* object. An FFT size of 1024 and overlap factor of 8 are set by the *block~* object. A rescale factor is derived for resynthesis, since the output from the FFT/IFFT sequence is not normalized. Finally, since Pd does not provide signal comparison objects such as >~ the LyonPotpourri object *greater~* is employed. A comprehensive set of signal comparison objects would be a welcome addition to the core set of Pd externals.

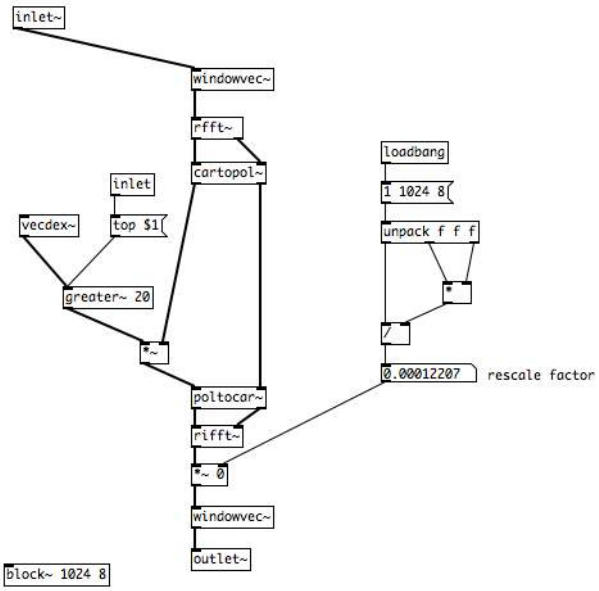


Figure 8 A Pd sub-patch employing a pseudo-pfft~ structure

The enclosing patch for the FFT-based hipass filter described above is shown in Figure 9.

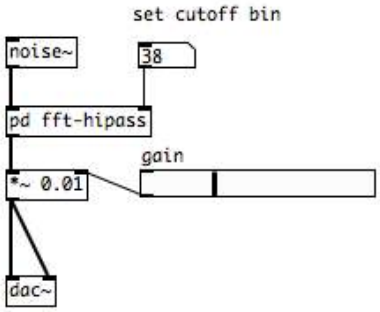


Figure 9 The parent Pd patch for the sub-patch shown in Figure 8

6 Generalized Panning Tools

Despite the existence of sophisticated algorithms for creating entire sound fields, such as Ambisonics, sometimes what is wanted is simple, equal-power panning across an arbitrary number of loudspeakers. This can be tedious to code in data-flow languages like Pd, and once coded, it is inconvenient to add or subtract channels. For this purpose, LyonPotpourri presents *npan~* which can pan a single input to an arbitrary number of output channels. The panning is controlled by a signal input that ranges from 0.0 to 1.0. An extension of *npan~* is provided with *rotapan~*, which can rotate an arbitrary number of input channels to its output.

Finally, the object *shoehorn~* provides a convenient way to collapse a larger number of channels to fewer channels, which can be convenient for listening to a multichannel piece in stereo, or mapping a dense loudspeaker array into a lower-order array.

6.1 Spectral Spatial Diffusion

LyonPotpourri 3.0 introduced two externals, *splitspec~* and *splitbank~*, for spectral spatial diffusion that I had first developed in 2003 during a residency at STEIM. They were based on the short-lived FFT Lite model, so they did not find a home in FFTease. And since there were relatively few multichannel systems that could deploy these externals at the time of their creation, I didn't get around to distributing them until very recently. The basic idea for these externals is to distribute an incoming spectrum to a fixed number of derived spectra, which in aggregate sum up to the source spectrum. In this model, a coherent input spectrum such as speech, which ordinarily would be perceived as coming from a fixed, single location, could be spectrally split, and then distributed to multiple loudspeakers, resulting in a characteristic form of auditory cognitive dissonance. A related technique, called Spatio Operation Spectral (SOS) synthesis was described in 2002 by David Topper [12].

An interesting extension of spectral spatial diffusion is to crossfade between two different spectral diffusion patterns. This technique is most effective on an input spectrum that is either slowly evolving, or static. The sub-patch for a typical use of s*plitspec~* is shown in Figure 10.

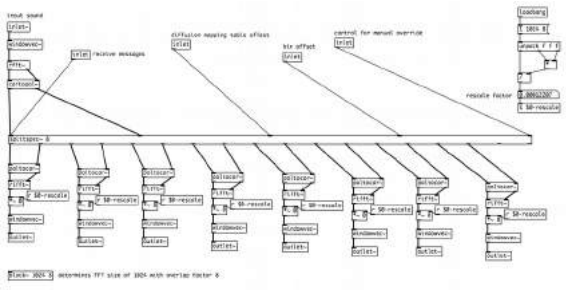


Figure 10 a sub-patch using splitspec~

The imported *pfft~* model is employed here. The argument to *splitspec~*, which must be a power of 2, determines the number of derived output spectra. A windowed IFFT must be performed on each output spectrum. The last

two outputs, unused here, send the current spectral diffusion mapping as a list, and the phase of interpolation between current diffusion mappings. The third and fourth inlets allow for real-time shifting of both the diffusion mapping table, and bin mappings. The main messages for *splitspec~* are shown in Figure 11.

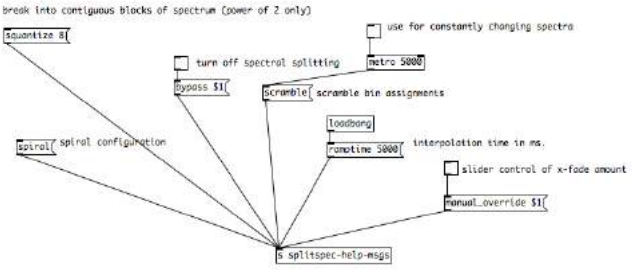


Figure 11 Message for splitspec~

The "scramble" message creates a new, random distribution of bins to output spectra. Each spectrum receives the same number of bin components, namely (N/2 * x) where N is the FFT size, and x is the number of derived output spectra. In the configuration shown in Figure 10, each derived spectrum will have 64 amplitude/phase pairs copied from the original spectrum. All other bin values will be set to zero. The "ramptime" message sends the interpolation time in milliseconds. When this is non-zero, each time a new distribution is generated, the object will cross-fade from the previous diffusion mapping to the new one. The "squantize" message quantizes the spectra according to its argument, which must be a power of 2. Each succeeding spectral frame receives a block of bin data from the original spectrum, the size of which is determined by the argument to "squantize." The "spiral" message distributes its bins sequentially to the output spectra, starting from zero. The first derived spectrum receives bin values from bin zero; the second derived spectrum receives bin values from bin 1, and so forth. The "manual_override" message allows the user to manually interpolate between the current diffusion mapping and the previous mapping, rather than an automated ramped transition, as is normally employed.

6.1.1 Tuned Spectral Spatial Diffusion

The external *splitbank~* is based on the *splitspec~* model, but implements oscillator bank resynthesis. This allows for the attractive possibility of tuning each derived spectrum independently. The deployment of this external in a sub-patch is shown in Figure 12.

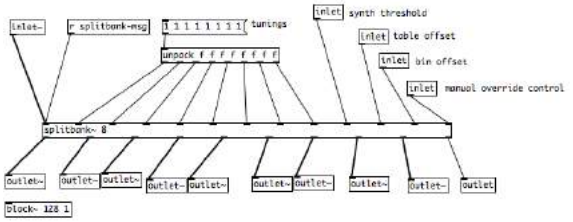


Figure 12 Splitbank~ used in a sub-patch

The required argument for splitbank~ determines the number of output spectra, and must be a power of 2. The first inlet receives audio signal to be spatially diffused. The next N inlets, where N is the number of output spectra, control independent tuning factors for each output spectrum. The next inlet allows for control of the synthesis threshold, which functions the same way as in *pvoc~*. The last three inputs are equivalent to the corresponding inputs for *splitspec~*. The first N outlets, where N is the number of output spectra, are the derived spectra as time-domain audio signal. *Splitbank~* is based on the earliest model of FFTease, where the signal vector size is also the hop size. The FFT size is then determined by (h * o), where h is the hopsize, and o is the overlap factor. With a preset overlap factor of 8, the FFT size for the example shown in Figure 12 is 1024. The *block~* object is used to set the local signal vector size inside of the sub-patch. The block~ overlap factor must be set to 1, since overlap-add and windowing are implemented within the *splitbank~* object. In addition to the global messages described for *splitspec~*, all of the tuning factors can be set simultaneously with a list message sent to the leftmost inlet containing N transposition factors, where N is the number of output spectra.

6.1.2 Extended Uses for Spectral Spatial Diffusion

While the externals *splitspect~* and *splitbank~* can produce some quite interesting effects through their basic functionality, they also lend themselves to extended forms of spectral spatial processing,, since their outputs are multiple audio signals that could be subjected to further signal processing prior to being routed to multiple loudspeakers. For example, the outputs could be run through *rotapan~*, allowing for the entire spectrally diffused sound field to be rotated around the audience. The outputs could be independently filtered, delayed, reverberated, or otherwise

processed prior to playback. The outputs could be further processed for binaural headphone listening, or independently processed for presentation in an Ambisonics-generated sound field.

7 Conclusion

We have highlighted certain aspects of FFTease and LyonPotpourri, two free, open-source collections of externals, in their most recent version for Pd. The source code for both collections can be downloaded from my Github page, https://github.com/ericlyon. Given the large number of externals in each collection, a comprehensive tutorial was beyond the scope of this paper. Instead, we have focused on core functionalities of FFTease, and spectral spatialization capabilities recently added to LyonPotpourri. We have also presented the implementation of a *pfft~*-like system for Pd in LyonPotpourri. It is hoped that this paper will increase the accessibility of FFTease and LyonPotpourri for Pd users, and will encourage increased experimentation with the externals found therein.

References

[1] E. Lyon and C. Penrose. “FFTease: A Collection of Spectral Signal Processors for Max/MSP.” In *Proceedings of the International Computer Music Conference,* pp. 496-498. ICMA, 2000.

[2] R. Boulanger, V. Lazzarini, and M. Mathews, eds.: *The Audio Programming Book*, MIT Press 2010.

[3] E. Lyon. “Designing Audio Objects for Max/MSP and Pd.” A-R Editions, Inc. 2012.

[4] T. Grill. "flext - C++ layer for Pure Data & Max/MSP externals." *The second Linux Audio Conference (LAC)*. 2004.

[5] F. R. Richard. *Elements of computer music*. Prentice-Hall, Inc., 1990.

[6] C. Penrose "Extending musical mixing: Adaptive composite signal processing." *Int. Computer Music Conf. Proc*. 1999.

[7] E. Lyon. "POWERpv: a suite of sound processors." *Proceedings of the 1996 International Computer Music Conference*. The International Computer Music Association, 1996.

[8] M. Dolson. "The phase vocoder: A tutorial." *Computer Music Journal* 10.4 (1986): 14-27.

[9] E. Lyon. “Designing Audio Objects for Max/MSP and Pd.” A-R Editions, Inc. 2012. pp. 211-243.

[10] E. Lyon. "Spectral Tuning." *Proceedings of the 2004 International Computer Music Conference*. 2004.

[11] E. Lyon. "A sample accurate triggering system for pd and max/msp." *Linux Audio Conference 2006 Proceedings*. 2006.

[12] D. Topper, M. Burtner, and S. Serafin. "Spatio-operational spectral (sos) synthesis." *Proceedings of the International Conference on Digital Audio Effects, Hamburg, Germany*. 2002.

Real-Time Sound Similarity Synthesis by an Ahead-of-Time Feature Extraction

Marco Matteo Markidis
nonoLab
V.le Vittoria 3
Parma, Italy, 43125
mm.markidis@gmail.com

José Miguel Fernández
IRCAM
Place Igor-Stravinsky 1
Paris, France, 75004
jose.miguel.fernandez@ircam.fr

Giuseppe Silvi
Centro Ricerche Musicali
Via Giacomo Peroni 452
Rome, Italy, 00131
me@giuseppesilvi.com

Abstract

This paper presents *path~*, a Pure Data external implementing a corpus-based, concatenative analysis-synthesis engine. Sound is acquired and synthesized by a similarity match between audio grains. A set of audio features are extracted offline for sound corpora and in real-time for live input.
The processing flow includes a non real-time part, where audio files are segmented, analyzed and ordered in a kD-tree structure, each grain being paired to a list of k-nearest neighbors. The system then performs a real-time feature extraction on live input, in order to find the best match in the tree. A pool of k grains is defined for the synthesis, exploiting the k-NN list coupled to this best match.

Keywords

Concatenative sound synthesis, real-time audio mosaicing, feature extraction.

1 Introduction

Sound synthesis techniques can produce countless timbres. Some of these create novel synthetic sounds; others aim to reconstruct defined spectra, often through a pre-analysis on existing sounds. Granular synthesis appears among the first applications in the field of sound production and reconstruction [1]. Unlike other sound synthesis techniques now crystallized in time, granular synthesis is still evolving in many musical applications, opening new perspectives and issues, concatenative sound synthesis (CSS) and audio mosaicing being an excellent example. The core idea of sound similarity synthesis is to extract descriptors from an audio file, to construct a timbral space and to introduce a metric such that the concept of distance becomes meaningful. The feature extraction highlights the perceptual similarities between sounds in a geometrical fashion. In this case, adjacent grains share similar sonic qualities in the morphological space. This space is multidimensional and its dimensionality depends on the number and the type of the descriptors used in the analysis. A non real-time feature extraction is performed for building a multidimensional database. In real-time, the incoming sound is analyzed using the same set of audio descriptors used in the offline analysis and the nearest neighbor search is carried out on this database. Lastly, a granular synthesizer plays back samples from a pool of sound units, constituted by the k-nearest neighbors (k-NN) of the best match. This part of the process is commonly referred to as synthesis.

The main motivation of this work is to develop a concatenative analysis-synthesis engine. The idea is to develop this system in Pure Data [1] environment giving the composer or the computer music designer the possibility of easily integrating this system into his programs dedicated to the processing of sound in real-time. The purpose of the project is also to create an open-source tool to perform contemporary music as in other commercial musical computing frameworks.
The main contribution of this work is to provide all the functionalities needed for the CSS in a single object, without the use of different tools for the analysis and the synthesis. This provides a more compact system that manages internally all the distinct steps. In addition, the internal algorithm is designed for the real-time application. Moreover other operations are performed offline, such as the database sorting and the calculation of k-nearest neighbors, that is needed in the synthesis part of the algorithm. Search on large database, i.e. long audio files, can be executed in a reasonable time.
The paper is organized as follows: the next section

[1] I. Xenakis' *Concret PH* is one of the first musical example.

describes the related works, focusing on projects developed in Max/MSP and Pd frameworks. Section 3 describes the methodology; in particular we focus on the proposed algorithm and on the specific implementation with some reference to technical decisions taken about the code. Results, both technical and artistic, are presented in Section 4. Finally we discuss the implementation and the future work.

2 Related Work

After the pioneering work of Iannis Xenakis, the attention of composers and researchers on granular synthesis and granulation of sampled sounds reached a point that, nowadays, granular synthesis is one of the most used techniques in electronic music composition.
However, several projects were developed using techniques involving concatenative synthesis from 2000s. In particular, corpus-based concatenative synthesis was explored with CataRT [3].

CataRT system is a set of patches for Max/MSP using several extensions, like FTM, Gabor and MnM. The idea is to load sound descriptors from SDIF [2] files, or to calculate descriptors on-the-fly, and to create a descriptors space using short grains as points. The user can choose two descriptors and using implemented displays, like graphic canvas, CataRT plots a 2-dimensional projection of grains in this descriptors space, where the user can navigate and explore the descriptors space through the mouse position. Within this graphical environment, the expressive nature of the grains can be emphasized. The CataRT architecture can be divided in five parts, as follows: analysis, data, selection, synthesis and interface. Apart from the interface facilities that we have previously discussed, this system uses other integrated Max/MSP tools like Gabor library for synthesis; in addition FTM data structures were used for data management. A piece for banjo and electronics by Sam Britton was composed within CataRT.

Another Max/MSP project was a piece of Marco Antonio Suárez Cifuentes: Caméléon Kaléidoscope [4], premiered at the Biennale Musiques en Scéne in Lyon in march 2010. A system combining FTM, Gabor, MnM and MuBu libraries was implemented for this production. MuBu is a multi-track container representing multiple temporally aligned homogeneous data streams similar to data streams represented by the SDIF standard [5].

In Pd world, several works were proposed on timbre analysis and sound classification. One of the largest projects is timbreID [2] library developed by William Brent. This library is a collection of externals for batch and real-time feature extraction. In particular, together with the presence of a set of low level features, i.e. a single-valued descriptors, some high level features, i.e. a multiple-valued descriptors, were implemented, such as mel-frequency cepstrum and bark frequency cepstrum. The Author shows that this set of high level features has an higher percentage of matching compared to the low level ones for audio event detection and classification [6]. Furthermore, the Author discusses the use of multi-frame analysis compared to single-frame analysis and the use of combined descriptors. In particular, high level ones plus low level features increase the percentage of matching at fixed analysis time.

This peculiarity is of great interest in the sound similarity reconstruction, because of the possibility of a best recognition and therefore a matching improvement between the incoming sound and the database elements. In addition the collection of single-descriptor external, a general object called timbreID is introduced, allowing for the management of the information database.
Finally, a recent audiovisual work, implementing a gestural controlled interface, has been presented [7]. This project is based on Pd, the computer graphics library GEM and timbreID library.

3 Methods

The *path~* analysis-synthesis architecture is presented in Figure 1.

Ahead-of-Time Analysis *path~* first performs an offline analysis. After loading samples, either from mono audio files in hard disk or from Pd's array, *path~* segments the audio corpora in slice, creating the available grains. Two types of segmentations are possible; the first one is frame-based, i.e. given an arbitrary frame, *path~* cuts out the samples at given frame and performs the analysis; all the grains have the same length.

[2] Sound Description Interchange Format.

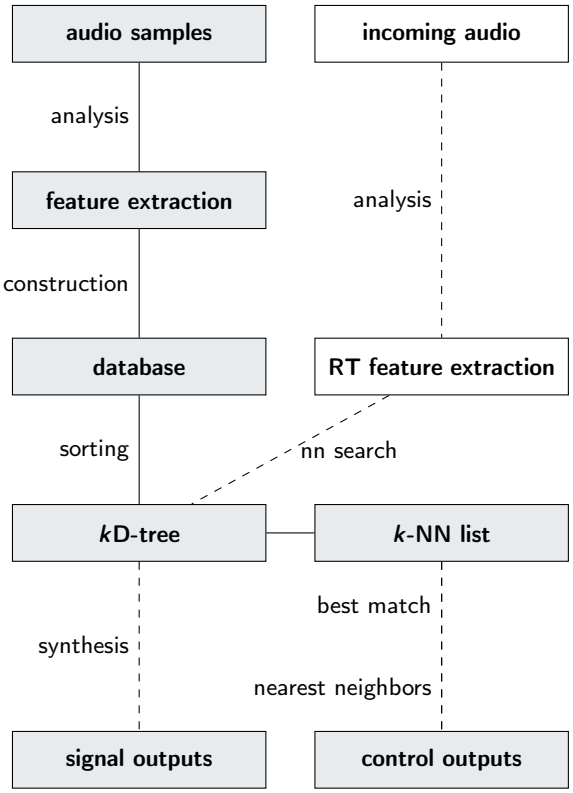


Figure 1: *path~* structure: ovals and dashed lines refer to real-time computations, rectangles and solid lines to offline analysis.

The second possible fragmentation is onset-based, i.e. an onset detection is made on the audio corpora and this fragmentation considers the gestural nature of audio files content. In this case, the grains have different lengths. Then features are extracted from all grains; these data form a database. We chose two multi-descriptors strategies. The default one is made by a high level descriptor plus a set of two low level features, while the second one excludes the high level descriptor leaving the others. The high level features is a mel-frequency cepstrum descriptor; while the low level features are a spectral centroid and an amplitude descriptor. The computation of Mel-Frequency Cepstral Coefficients requires a bank of overlapping triangular bandpass filters evenly spaced on the mel scale. A mel is the scale unit defined as:

$$Mel(f) = 2595 \times log_{10}(1 + \frac{f}{700}), \qquad (1)$$

where f is the frequency in Hz. Finally, in order to get the MFCCs we discrete cosine transform:

$$MFCC_i = \sum_{k=0}^{N-1} X_k cos[i(k + \frac{1}{2})\frac{\pi}{N}], \qquad (2)$$

with $i = 0, 1, \ldots, M-1$. M is the number of cepstral coefficients, N the number of filters and X_k is the log power output of the k^{th} filter. The user can choose the frequency of the mel spacing during the instantiation time. Varying this parameter, the number of resultant filters changes. This means that the dimension of our timbre space will change with respect to this parameter. The default value of mel spacing is 250 Hz given a 14-D timbre space [3], that is the number of MFCCs, plus 2, that is the dimensionality of the spectral centroid and of the amplitude descriptor: every grain will be represented as a set of 16 numbers, i.e the coordinates of a grain in the timbral space.
The spectral centroid (SC) is defined as the center of mass of magnitude spectrum.

$$SC = \frac{\sum_{k=0}^{N/2} f(k)|X(k)|}{\sum_{k=0}^{N/2} |X(k)|} \qquad (3)$$

with $f(k)$ the frequency of the k^{th} bin.
The amplitude descriptor is a root mean square amplitude by default; it can be changed in a peak amplitude.
path~ supports a multithreading strategy. In this way, the worker thread performs the offline analysis while the Pd thread continues its tasks. In this case, an offline analysis can be performed in an asynchronous manner without breaking the audio flow. However, our strategy partially breaks the Pd determinism. This is due to due to the fact the inter-thread communication is asynchronous and the analysis method is blocking only the worker thread function. The first attempt to alleviate this problem is to split the analysis method in two methods. The first one carries out the analysis while the second one performs the swap of the old data structure containing the needed data for the synthesis with the new one. The swap method lies in the Pd thread and it is synchronous with Pd. Moreover, we are dealing with this problem creating a format where user can store the data structure needed for the synthesis. This operation enables to simply load analysis that can be done in batch. This procedure reduces the time to get analysis complete. Finally, *path~* signals a bang message when the analysis is complete.
These are partially solutions to the problem and

[3] Tested mel spacings: 500 Hz = 6-D, 250 Hz = 14-D, 100 Hz = 38-D.

suggests as think to analysis as an ahead-of-time analysis. It can introduce several novel sound representations at the cost of relaxing Pd determinism.

Database The analysis forms a database, that is subsequently sorted in an indexed kD-tree. A kD-tree is a space-partitioning data structure for organizing points in a k-dimensional space. This data structure reduces the time complexity of nearest neighbor searches at the cost of tree construction. This sorting allows us to avoid linear searches giving the possibility of use large database. However, in high-dimensional spaces, the curse of dimensionality decreases the efficiency of the algorithm than in lower-dimensional spaces. The algorithm is only slightly better than a linear search of all of the points if the number of points is only slightly higher than the space dimensionality. For high-dimensional spaces, approximate nearest neighbor searches could be implemented.
Finally, *path~* constructs a k-NN list, that consists of available grains for the synthesis. This feature gives the user the possibility to have a pool of grains sharing sonic similarities that can be executed. Moreover, this list allows to have a fast and sequential access to the pool of grains without extra cost in real-time. The user can specify the maximum number of nearest neighbors to include in the pool. Due to the finite number of nearest neighbors usually requested, *path~* uses a quickselsort algorithm. Quickselsort is a partial sorting algorithm; it performs first a quickselect, a selection algorithm, then a quicksort, a sorting algorithm. This strategy allows us to decrease the cost of the k-NN list construction, that is executed for every grains.
Selection In real-time, *path~* performs a feature extraction on the live input. *path~* has the same analysis methods as for the source audio materials. However, there is no onset detection on live input; so even if the grains have different lengths the input analysis has a fixed length. The analysis is done according to the given user input and the windows analysis is independent on the blocksize. Then, *path~* searches in the kD-tree the element that minimizes the Euclidean distance. The selection is local, i.e. the best match is found individually. Accessing to the k-NN list, a pool of grains is available for the synthesis and *path~* outputs the control information. The control information is the matched index founded in the database, the list with the desired k-NN indices, the total number of grains and the number of grains actually played back.
Synthesis *path~* introduces a concatenation quality function. The term concatenation refers here to train length of the grain, i.e. the number of grains that are concatenated after the first one. In this way, a grains train can start creating sound textures. Accessing to the k-NN list, a certain amount of grains can be executed and concatenated. The user can access the list linearly or randomly obtaining different sonic results.
Preset Finally we implemented a script interpreter as preset system. In this way, the user can write a script within a given syntax and *path~*loads it and creates a preset that can be called with a dedicated method. Optionally, the user can specify an identificative name for every preset. Apart from the interpreter, a simple debugger has been added. Furthermore, the complete preset list containing the debugged list with the various preset names can be viewed in a GUI editor that can be called clicking on *path~*.

```
# path~ preset file example ;
# this is a comment ;

preset @ intro ;
threshold = 0.;
window = 512;
concatenate = 1;
amp = 1;
hopsize = 100;
weight = 1;
envelope = 0;
endpre

preset @ sec1 ;
window = 2048;
threshold = 0.25;
concatenate = 3;
amp = 0.2 0.4;
hopsize = 100 3000;
envelope = 2;
endpre
```

Preset file example.

4 Results

4.1 Latency

One of the main problems in real-time audio is the latency, defined as a time delay between the input arrival and the output response. So we study the time spent by *path~* between the user input and the grains response. Our estimate about real-time latency limit is 6 ms. On Authors' machine, i.e.

a Mac BookPro Early 2011 13" 2.7 GHz with Pd Vanilla 0.47.1, on a database size of the order of 30K obtained using a 10 minutes audio file long with a 23 ms analysis window, database lookups took less than 2 ms on average, while offline analysis took less than one minute on average.

4.2 Application in mixed contemporary composition

The use of real-time analysis systems for instruments' sound is a key field in the interaction between acoustic instruments and electronics. While over the last few years various methods of analysis of the sound signal, e.g. audio features extraction, have been implemented, these methods have not been explored in all them possibilities.
In the following, we discuss about three projects where *path~* is involved. In particular, we present the different strategies within which *path~* has been designed to fulfill the requests of composers and computer music designers.

On the fly

The first musical application was Marco Matteo Markidis' radical improvisation duo Adiabatic Invariants. The entire electronics of Cattedrali di Sabbia, a 40 minutes improvisation, is based on *path~*. One of the main problems was to use *path~* on the fly, due to the improvised nature of the duo. The multithreading idea was born in this way; moreover during the set several fragments of percussions are recorded on Pd's arrays and analyzed in the following. Moreover, in chaotic musical situation, an high number of small events can be very useful; we experimented several configurations with a lot of concatenated grains. Finally, *path~* was used for audio mosaicing. Audio mosaicing refers to the process of reconstructing the temporal evolution of a target sound from fragments taken from a source audio materials. In this case, *path~* was triggered with a metronome and in an asynchronous way by the use of an envelope follower.

Space

For *S4EF*, a work for augmented alto saxophone [4] and electronics by Giuseppe Silvi, *path~* is called in three instances at the same time: one for *Windback*, the electroacoustic device applied on saxophone; two for *S.T.ONE* tetrahedral loudspeakers.
The *Windback* augmentation consists in a feedback system between embouchure and bell of saxophone, with hardware and software control of both [9]. The Windback call of *path~* is monophonic with frame-based fragmentation to control fine morphology sounds for internal feedback. The two *S.T.ONE* calls of *path~* are polyphonic, each of four channels, one for each side of tetrahedral loudspeaker. The S.T.ONE system (*Spherical Tetrahedral ONE*) is a loudspeaker designed by Giuseppe Silvi, based on *Ambisonic* technology with the ambition to obtain electroacoustic diffusion of electronic music with the same space relationship and sound shape of acoustic instruments. The use of grain polyphony by this kind of technologies has been improved synthesizing contemporary grains on different loudspeakers. For S.T.ONE sound synthesis *path~* is used with onset detected fragments to obtain grains with emphasized musical content.
The Windback research team now is focused on full augmentation of saxophone quartet (SATB) where a multi-input version of *path~* will be an useful tool of sound synthesis for *D.R.Y.* augmented saxophone quartet by Giuseppe Silvi.

Live input

Currently, we are working with Lara Morciano, an Ircam composer, on her new piece for piano and electronics. One of the most interesting Morciano's requests is to use no sound as live input, but to investigate the timbre space directly with the use of motion caption. Our strategy is to access directly to the timbre space; it allows us to explore several configurations even with or without live sound. Technically, the incoming motion captured data are added to the features extracted ones; basically it is possible to change the coordinates of the live extracted point to be used in the database lookup. With no sound, these data act as an offset respect to space origin. This gestural controlled way to produce grains is particularly effective with onset detected grains in such a way that gestures are related to musical meaningful grains. For this piece, we are using *path~* without the high level descriptors; the captors communicate directly with the two low level descriptors. This strategy allows for a more clear control.

[4] The Windback system is made by Michelangelo Lupone; the *S4EF* work is produced by *CRM Centro Ricerche Musicali*, Rome.

5 Discussion and Conclusions

The primary aim of this work is the production of contemporary music pieces for acoustic instruments and electronics. To achieve this aim, several improvements are planned. In particular a multi-input version will be useful in an ensemble context. In addition, a greater control on spatialization and on single grain control localization can be desirable. Moreover, an editor improvement together with a more detailed preset configuration and setup will be designed. A growing interest will be dedicated to cluster analysis implementation and on pattern recognition, particularly trying to develop some specific strategy for extended contemporary techniques and for different grain lengths. Finally, a forthcoming research is a multi-agent approach to corpus-based CSS [8] allowing us to explore the morphological space.
path~ has been tested on OS X and Linux using Vanilla stable release 0.47.1. *path~* is an open source software and it is released under GPLv3. It is available via Deken, the externals wrangler for Pure Data.

6 Acknowledgements

We would like to thank Stefano Markidis for his suggestions; Michele Orrù and Ilaria Ciriello for the technical guidance and encouragement; Matt Barber and Jonathan Wilkes for the stimulating discussions; Miller Puckette, William Brent and the cyclone library team for the knowledge their codes gave us. We thank entire Pd community: great community, great software, great freedom.

References

[1] M. Puckette: *Pure Data: another integrated computer music environment*, Proceedings of the International Computer Music Conference (ICMC), 1996.

[2] W. Brent: *A timbre analysis and classification toolkit for Pure Data*, Proceedings of the International Computer Music Conference (ICMC), 2009.

[3] D. Schwarz and G. Beller and B. Verbrugghe and S. Britton: *Real-time corpus-based concatenative synthesis with CataRT*, Proceedings of the International Conference on Digital Audio Effects (DAFx), 2006.

[4] N. Schnell and Marco Antonio Suàrez Cifuentes and Jean-Philippe Lambert: *First steps in relaxed real-time typo-morphological audio analysis/synthesis*, Proceedings of the Sound and Music Computing Conference (SMC), 2010.

[5] N. Schnell and A. Röbel and Diemo Schwarz and Geoffroy Peeters and Riccardo Borghesi: *Mubu & Friends - Assembling tools for content based real-time interactive audio processing in Max/MSP*, Proceedings of an International Computer Music Conference (ICMC), 2009.

[6] W. Brent: *Cepstral analysis tools for percussive timbre identification*, Proceedings of International Pd Conference (PdCon09), 2009.

[7] J. Gossmann and M. Neupert: *Musical Interface to Audiovisual Corpora of Arbitrary Instruments*, Proceedings of New Interfaces for Musical Expression (NIME), 2014.

[8] D. Pozzi: *Composing exploration: a multi-agent approach to corpus-based concatenative synthesis*, Proceedings of Colloquium on Musical Informatics (CIM), 2016.

[9] M. Lupone, S. Lanzalone, L. Seno: *New advancements of the research on the augmented wind instruments: Wind-back and ResoFlute*, Electroacoustic Winds Conference, Aveiro. 2015.

SodaLib: a data sonification framework for creative coding environments

Agoston Nagy
Moholy Nagy University of Design and Arts
Zugligeti út 9
Budapest, Hungary, 1121
stc@binaura.net

Abstract

This paper introduces SodaLib, a sonification framework made for different programming languages [1]. It's aim is to let creative practitioners turn data into sound as easily as possible without interfering their regular workflow and without the need of advanced musical knowledge. SodaLib proposes generic and flexible ways of working with many types of data representations, including measurements by sensors, real-time data feeds, pre-recorded data, or even game and other interaction-driven temporal events. The sound engine is made with Pure Data using LibPd [2] and it can be embedded into applications running on many operating systems from desktop to mobile, including iOS, Android, Raspberry (ARM linux).

Keywords

Data Sonification, LibPd, Creative Applications, Extending Pd

1. Introduction

In the ever-growing area of data driven interfaces (embedded systems, social activities), it becomes more important to have effective methods to analyze complex data sets, observing them from different perspectives, understanding their features and dimensions, accessing, interpreting and mapping them in meaningful ways. With SodaLib, it is easy to map live data (sensors, web apis, etc), large, prerecorded datasets (tables, logs, excel files), or even unusual sources (images, 3D environments) to recognizable audible patterns through a set of sonification methods, including parameter mapping and event based sonification [3].

SodaLib is an open source, cross platform, multipurpose sonification tool for designers, programmers and creative practitioners.

2. Motivation

A personal motivation for creating this library is coming from the discussions of different workshops and courses on creative coding practice, where sound is used in novel ways for communication [4]. By discussing topics on sound, game engines, sonification, VR environments, data transmissions using different sensor systems, it turned out that there is a general need for a multipurpose tool which can be embedded with a few lines of code into the regular workflow of people coming from different backgrounds, letting them focus on application creation combined with creative sound production.

Another, more general need for a high level framework is the lack of well defined structures for representing data with sounds. While visualization has a great and extensive history of representing data through lines, shapes, colors based on cognitive and gestalt elements that are rooted in our visual perception [5], sonification lacks this common language structure and it is deeply overwhelmed with musical concepts and/or advanced synthesis systems that are hard to understand for people who primarily work with interaction and data instead of producing sounds. SodaLib provides an easy way to sonify data from one dimensional discrete events to multidimensional, continuous data streams with just a few lines of code.

The framework does not follow the tradition of most available sound libraries that are about to build sequencers, soft-synth like studio equipments and similar tools that are based on musical symbolic structures. Instead, the system is dealing with data as it is: flexible, neutral entities in multidimensional contexts. These elements can be sonified in similar ways among all dimensions, regardless of the underlying sound engine construction. The dimensions can be selected based on the relevant contexts of the data, they can be addressed using interchangeable values, such as shift in pitch, power in volume and position in pan (including binaural 3D positioning), depth as reverberation size, etc. The dimensions of the system can be

extended very easily if needed by adding custom generators and/or mapping functions. However, at the moment, these few parameters seem to fit the basic needs when dealing with human cognitive efforts and the regular perception conditions of the mind.

3. LibPd

SodaLib's core functions are written in Pure Data (vanilla), it is fully compatible with LibPd. LibPd is platform agnostic in terms that it does not rely on any audio API calls directly, instead it can be embedded into many environments with a few binding functions to access low level API calls. It can be easily adapted into many creative coding environments, including c++, java, python, targeting platforms from desktop to iOS, Android, Raspberry PI or server based applications.

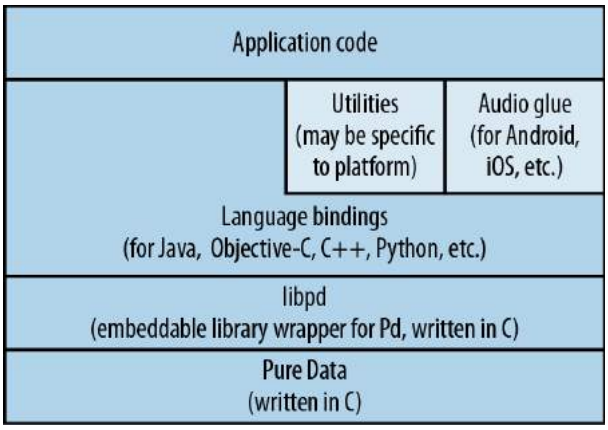


Fig. 1: Layers of LibPd. From the book Making Musical Apps by Peter Brinkmann. Chapter 4, fig. 4-1

When working with different creative coding frameworks within these environments, one can access and use the core Soda objects through a thin, streamlined language specific interface layer, without the need to know how to use Pure Data itself or things like synthesis and musical structures. Instead, users can stay with their preferred environment, keeping focus on application creation: representing data and designing interaction.

4. Implementation

4.1 Architecture of the sound engine

SodaLib consists of a sound engine that is written in Pure Data and a few utility functions that are available within the application code. The Pure Data part is a set of vanilla abstractions that are divided into some utility functions and building blocks. These blocks are dynamically generated sound creators. At the moment, polyphonic sampling, synthesis with basic, yet extendable waveforms and a noise filtering block have been implemented, since these can be applied well for different type of data representations. Please refer to the examples for more on which sound block is best for what type of data.

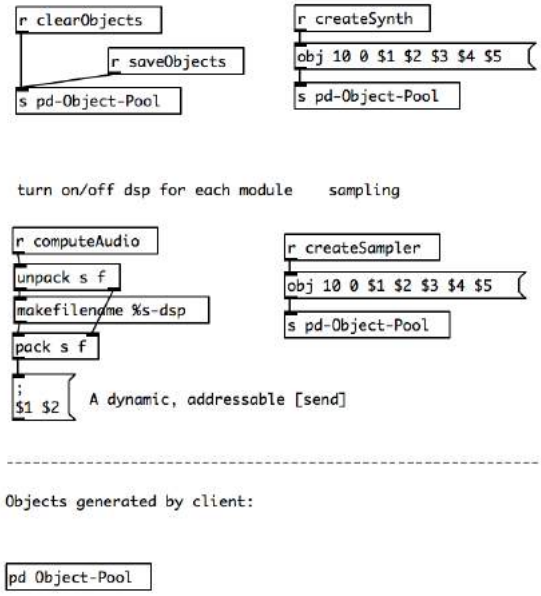


Fig. 2: Part of SodaLib's main.pd patch. Lists received from the client code are interpreted as object creation arguments to Pd. Each object can be addressed and manipulated during runtime, routing is based on it's name

The internal, flexible modularity is made possible by dynamic patching. This means, the building blocks are generated and saved on the fly, the interconnections between the different elements are made without chords, they are based on a simple internal addressing system, where each dynamically generated abstraction can be addressed via it's name that it has got during creation time.

The benefit of this concept is that the objects can be created and addressed directly from the application code, which is completely different from the sound generator code. This idea is based on the idea of libPd, where values can be sent to the sound engine via messages: if a function that sends data to libPd has a receiving pair, the data can be processed within LibPd.

SodaLib extends the concept of this system to a more abstract level: the readymade, complex sound rendering objects can be created via simplified message packages that are received in Pure Data.

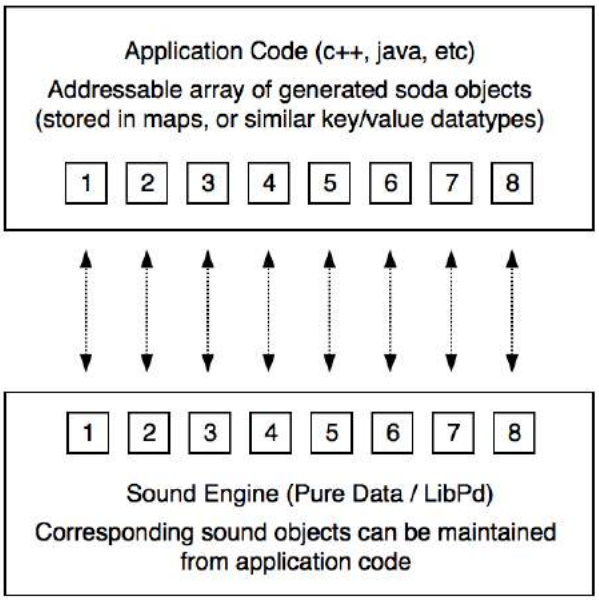


Fig. 3: Each Soda Object in the application side has a corresponding sound object in Pd. Modifying parameters of an object in the application code affects directly its sound generator in the Pd side.

4.2 Binding functions and object management in different languages

Since the messages addressed to the sound generators can be accessed via simple lists on the Pd side, the framework has some binding functions on the application side. All these functions are acting as a communication layer that can take some parameters, make a list from them and forward it to the sound engine. The code is taking care of the already existing Soda objects beyond these functions: when generating an instance, each object is put into an easily addressable data container (HashMap in Java, Map in C++) where objects can be addressed as key / value pairs. Keys are the names of the objects and the values are the referred objects themselves. This comes really handy when working with multiple objects at once. For example, if someone has a larger array of data, it is easy to generate a sound object for each data point in the array and address its corresponding sound object based on the index of the data in the array.

4.3 Example application code

Creating and manipulating custom sound objects can be made with just a few lines of code. While the following examples are taken from the syntax of C++ (OpenFrameworks [6]), similar functions and container datatypes can be used with other coding frameworks, such as Java (Processing), Python, Swift, C# (Unity), etc. In the first example, we create a custom polyphonic sampler object and change its pitch. First, initialize Sodalib. This function creates the sound engine and loads the main puredata patch to the system:

```
soda.init();
```

Then, create a sound generator, with the name *mySample*, loading the file *sound.wav*, with polyphony of 10. This last parameter is depending on the loaded patch type, it can have different meaning when making synths or other generator objects.

```
soda.createSampler("mySample","sound.wav",
10);
```

Later on, when the program runs, the sound object can be manipulated by the following code:

```
soda.set("mySample")->shift(0.5)->play();
```

Here, the *shift* value of the object is modified. 1 is original speed, 0.5 is half speed, etc.
A more interesting example would be to create several objects and control them based on incoming data values. Let's create a lot of soda objects now, this time some sonic texture generators

```
for(int i=0; i<100; i++) {
        soda.createTexture("soda-" +
        ofToString(i),60);

        soda.set("soda-" + ofToString(i))
        ->shift(i/100);
}
```

Now, we have 100 objects. These are generating noise with a bandpass filter (with a bandwidth of 60 Hz). Notice that we are using the *i* integer for naming and accessing our objects. This integer is parsed to a string using the *ofToString()* function. By changing their shift parameter after creation, each object will have different filter coefficients, that goes from 0 – 1. In SodaLib, ideally all function parameters should be between 0 – 1. This range is mapped to meaningful values in the sound objects. Volume goes between 0 (minimum) to 1 (maximum value), such as pan from 0 (left) to 1 (right), filter coefficient goes from 0 (0 Hz) to 1 (20 KHz) etc. This concept can be strange from a musical point of view at first glance, but this pattern is effective for keeping the parameter dimensions interchangeable and flexible.

We can access and manipulate any of the created objects by referring to their names. We can do it individually, say if we would like to access and manipulate only one object from our 100 instances:

```
soda.set("soda-42")->volume(1)->play();
```

Since the *set()* function returns a soda object, different parameters can be modified at the same time by making chains of the parameters, like

```
soda.set("soda-42")->volume(1)->pan(0.2)->shift(2)
->play();
```

Of course, we can set all of the objects together at once, based on existing values in an array or any data type. It's also possible to just loop through them and create different random pitches for them using another for loop:

```
for(int i=0; i< 100; i++) {
        soda.set("soda-" + ofToString(i))
        ->volume(ofRandom(1))->play();
}
```

There is another useful feature of the lib that can be applied for gaming and VR development. When a *pan()* function is called on an object, the type of the sound rendering is depending on the number of function arguments. If it has only one, the panning is done in a traditional, stereo left/right manner. However, by passing three parameters, the position of the sound can be controlled in three dimensions, by passing the azimuth, elevation and distance to the desired sound origin. This binaural sound positioning is using the *[earplug~]* external [7] for the math and modelling, and it can be really effective when developing games or 360° animation scenes.

5. Conclusion

SodaLib is a contribution for creative practitioners who are interested in making all type of realtime applications that involve sound generation without the need to know sound programming and/or advanced musical concepts. It can be used for very quick rapid prototyping, but since it is completely embeddable, it can also be used to target different professional digital publishing platforms [8]. Since data driven sound will be more common in the era of connected devices and embedded technology, the relevance for creating such frameworks is evident.

It can also help in teaching the concepts of interactive sound, visual sound instruments [9] and different techniques of sonification. On one hand, attention of newcomers from other creative disciplines and creative coding frameworks can be aroused to the directions of Pure Data and lower level DSP techniques, on the other hand, experienced pd-ers can learn how to build applications, publish their works in forms of different digital media, apart from the performance stage, installations or regular music production activities [10].

Last, but not least, a long-term, interesting topic is of course the emerging field of the theory and practice of sonification. While visualisation has its own literacy and well defined concepts with practical building blocks, sonification lacks most of these. With such tools, it would be easier to maintain discussions on the cognitive, psychoacoustical aspects of sounds that are representing data: which types of sonic structures are describing specific datasets more effectively?

6. Acknowledgements

Thank you for all the developers of Pure Data & LibPd, including Peter Brinkmann, Dan Wilcox and others, dynamic patching ideas of IOhannes Zmoelnig, the RJDJ crew and many others for their effort in the making, Gábor Papp, Bence Samu for helping in and discussing tons of interesting concepts, Réka Majsai, Adam Somlai-Fischer, Bill Manaris for having such inspiring discussions around and beyond the topic.

References

[1] https://github.com/stc/ofxSodaLib (implementation for OpenFrameworks)

[2] Brinkmann, Peter et al: Embedding Pure Data with LibPd. PdCon11: https://www.uni-weimar.de/medien/wiki/images/Embedding_Pure_Data_with_libpd.pdf

[3] Hermann, Thomas et al: Sonification Handbook. Logos Publishing Hous, Berlin, 2011 http://sonification.de/handbook/

[4] Sonic Instruments Workshop, ICAD, ISEA,

2015 http://www.binaura.net/stc/ws/isea/
[5] Fry, Ben: Visualizing Data. O'Reilly, 2007
[6] http://openframeworks.cc/
[7] Earplug External, a binaural filter based on KEMAR impulse measurement for Pd https://puredata.info/downloads/earplug
[8] Brinkmann, Peter: Making Musical Apps. O'Reilly, 2012
[9] Nagy, Agoston: Visual Sound Instruments (Thesis), MOME, 2014
[10] Steiner, Hans Christof: Build your own instrument with Pd. In: BangBook, pd-graz, 2006 https://puredata.info/groups/pd-graz/label/book/bangbook.pdf

Real-time Collection and Reassembly of Audiovisual Concatenative Synthesis Corpora

Max NEUPERT
Bauhaus-Universität Weimar
Weimar, Germany, 99423
max.neupert@uni-weimar.de

Abstract

The aim of this research is to give the musician or user a direct access to a sound corpus, without the original instrument or source. This creates a meta-instrument in which the the original instrument is not merely re-synthesized, but its sound qualities can be accessed in an entirely different way than through the original instrument. In the example of the cello, the meta-instrument allows for the creation of cello-like sounds from the corpus, which would be physically impossible to play on a cello. The visualization of the sound fragments is at the same time the interface through which they can be navigated and accessed. A display of the corresponding video frame(s) is provided too (Figure 2).

Keywords

Audio-visual, Real-Time, Cut-Up, Point-Cloud, Video.

1 Introduction

Based on experiences with previous research [1,2,3] in a fragment based feature instrument for the playback of audiovisual corpora, the current investigation tries to accomplish the continuous collection of grains for the growth of the concatenative synthesis corpus. The aim is to provide a better understanding of the point cloud's structure by allowing the user to build the corpus in real-time. Since TimbreID [4] expects the corpus to be loaded faster than real-time, this has been more difficult than anticipated.

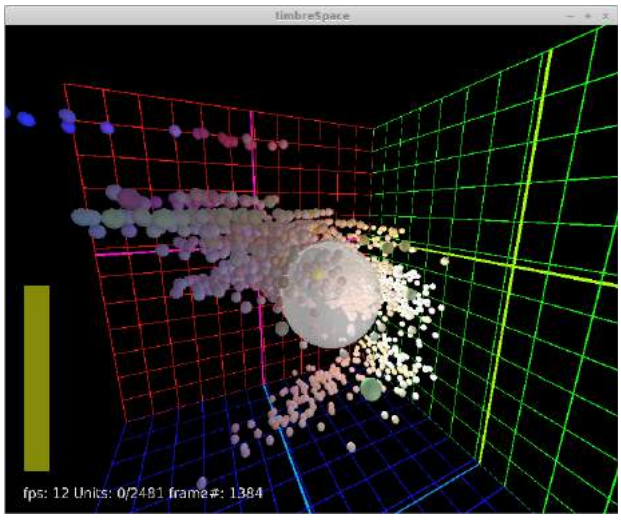


Figure 1: Point-cloud of the units

2. Why the structure matters

Concatenative synthesis (also called musaicing) [5] means that a "learned" sound is sliced in sub second fragments, further called units, which then are analyzed. In the playback they are made accessible through the results of this analysis. This gives the playback an understanding of what the grains represent and it distinguishes concatenative synthesis from granular synthesis, where the grain player is agnostic to the grains content.

2.1 Accessibility of the sound

An instrument like the chello provides its player with a straightforward relationship between the control parameters and their effect: Bow pressure + speed = volume, string length / tension = pitch. In the concatenative synthesis model those parameters may be amongst the vectors of the analysis and might be mapped in any way to a controller.

It is essential for the player of the instrument that a gesture is repeatable and predictable in its relation to the produced sound. Only if this condition is met, the playback can be a natural interface for an embodied interaction.

2.2 Using the hands in an empty space

A non-haptic, contact free interface is about the worst possible choice for a natural musical interface, but it's very economic space wise, light to travel with, low maintainance and convenient to build. By using the leap motion controller, this space is ready to be utilized. In a Theremin inspired interface model, the right hand navigates in the point cloud, while the left hand controls the volume and a reverb filter. Opening an closing the right hand also controls the envelope size of the units to play back. (Figure 3)

2.3 Navigation in a crowded space

It is often desired to play back one specific unit and revisit it later for a musical motif. However if this unit is to be found in a dense area (Figure 1), it can be difficult to find it later again. This problem could be solved in different ways: 1. by distributing the units evenly, but this would mean in the case of the pitch that the scale is lost. 2. give up the absolute tracking of the hand but implement one that accelerates with faster movements, just as we know it from mouse input. This would mean to loose the absolute positions of sounds. 3. give the user a way to zoom in temporarily to have a better control.

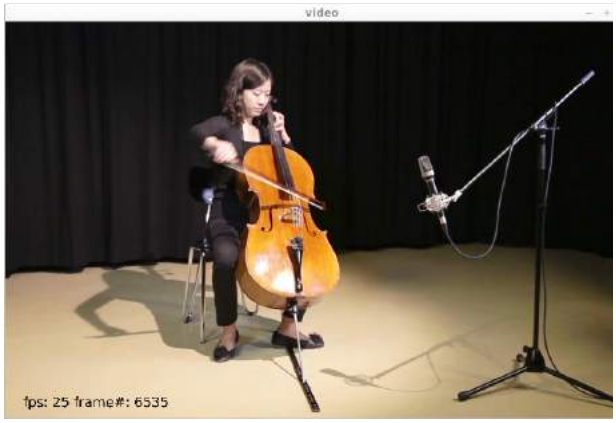


Figure 2: Video frame playback

2.4 Real-time input and its problems

The real-time analysis and collection of fragments helps the musician/user to understand where which sounds are placed in the coordinate system and can identify possible problems in the analysis. It also allows the player to fill gaps in the corpus. More importantly, it allows for a playful interaction with the meta-instrument and an acoustic instrument in a performance scenario.

However, since its maxima are unknown prior to their input, the coordinate system must either change its scale or an assumption must be made in advance.

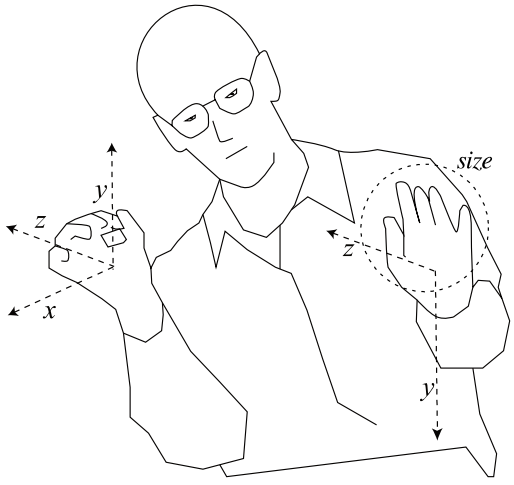


Figure 3: Interface metaphor

3 Acknowledgements

Special thanks to cellist Hyunji Cho. This work is building upon *Pure Data* by Miller Puckette and its external libraries *Gem* by Mark Danks/IOhannes zmölnig, *TimbreID* by William Brent and *shmem* by Cyrille Henry/Nicolas Montgermont. It makes use of the *leap-motion* external by Chikashi Miyama built with the *flext* environment by Thomas Grill. The grain player was developed with the help of Joachim Goßmann.

References

[1] M. Neupert and J. Goßmann: *Performing Audiovisual Corpora of Arbitrary Instruments* in Emille, the Journal of the Korean Electro-Acoustic Music Socienty, Vol 12, 2014

[2] J. Gossmann and M Neupert: *Musical Interface to Audiovisual Corpora of Arbitrary Instruments*, in *Proceedings of the international conference on new interfaces for musical expression*, London, United Kingdom, pages 151- 154. 2014.

[3] M. Neupert and J. Gossmann: *A remix instrument based on fragment feature-analysis.* In IEEE International Conference Multimedia and Expo Workshops (ICMEW), San Jose, pages 1–5, 2013.

[4] W. Brent: Cepstral analysis tools for percussive timbre identification. In: Proceedings of the 3rd International Pure Data Convention, São Paulo, 2009.

[5] D. Schwarz: Journal of New Music Research: Special Issue on Audio Mosaicing, 3 (35), 3–2, 2006

Cloning Max/MSP Objects: a Proposal for the Upgrade of Cyclone

Alexandre Torres Porres
EL Locus Solus
São Paulo-SP Brazil
porres@gmail.com

Derek Kwan
Fresno, CA USA
derek.x.kwan@gmail.com

Matthew Barber
Rochester, NY USA
brbrofsvl@gmail.com

Abstract

The Max/MSP compatibility library Cyclone has been updated, extending compatibility to Max versions 5, 6 and 7 over the current canonical Cyclone version's compatibility with Max version 4. Primary development goals have included updating existing objects to reflect features introduced in newer versions of Max, cloning Max objects missing from Cyclone, fixing bugs in existing code, and addressing errata found in Max objects and documentation. Secondary goals have involved an upgrade to documentation, streamlining the codebase, and improving the build system. Community discussion will resolve the project's future as either an update or fork of the current Cyclone.

Keywords

Cyclone, Max/MSP, Externals, Library, Update

1 Introduction

The Cyclone library is a set of externals that clones object classes from Cycling '74's Max/MSP software for Pure Data. Cyclone expands Pd with a large set of externals and provides some level of compatibility between Pd and Max, helping users of both systems in the development of equivalent or similar patches, and giving Max users a convenient gateway to Pd. Original development of Cyclone brought compatibility with Max version 4. We present a significant update to Cyclone which brings compatibility up to recent Max versions (7.3.0 at the time of this writing) by adding new functionalities introduced in newer Max versions to Cyclone's classes, fixing bugs, and writing new documentation.

2 A Brief History of Cyclone

Putting our update in its proper coding and social contexts requires a history of Cyclone.

2.1 Maintainers and Versions

The first iteration of Cyclone was begun in April 2002 by Krzysztof Czaja, when Max was at version 4.0. [1] The earliest experimental alpha release, cyclone-0.1-alpha1, was introduced to the Pd community in November 2002. [2] It had 60 control (Max) objects, and 12 signal (MSP) objects, collectively named "hammer" and "sickle," respectively. Cyclone grew to 166 cloned objects by 2005, but was still in alpha development (cyclone-0.1-alpha55).[1]

Cyclone could be loaded in several ways: in one, each object could be loaded from its own compiled binary. In another, `hammer`, a sub-library with all the control objects compiled into one file, and `sickle`, a similar sub-library with the signal objects could be loaded separately. Loading the `cyclone` sub-library would load `hammer` and `sickle`, as well as a few non-alphanumeric signal "bin-ops" like `>=~`, known in Cyclone as "nettles." Loading the sub-library `maxmode` would load `hammer`, `sickle`, and `cyclone`, along with a suite of "dummy" classes, which instantiated nonfunctional objects with the names of classes not yet cloned, so that – in principle – a Max patch could be opened from Pd. Finally, an executable binary `cyclist` was provided to convert binary Max patches into the equivalent text forms.

Hans-Christoph Steiner made Cyclone available in Pd-extended, and because Czaja ceased developing it in 2005, Steiner took over maintenance of Cyclone until 2013. The library was updated to cyclone-0.1-alpha56 for Pd-extended version 0.43.[2] All development on Pd-extended ceased in 2013, and Cyclone was left unmaintained as a result.

Fred Jan Kraan assumed maintenance of Cyclone in December 2014. He made cyclone-0.1-alpha57 available via the `deken` library manager plugin, and replaced it with the current canoni-

[1] Although Krzysztof Czaja's webpage no longer exists, a copy may be found at `http://fjkraan.home.xs4all.nl/digaud/puredata/cyclone/cyclone_site/cyclone.html`, which includes a downloadable source for cyclone-0.1-alpha54.

[2] Pd-extended releases may be found at `https://puredata.info/downloads/pd-extended/releases/` and at the sourceforge.net repository `https://sourceforge.net/projects/pure-data/files/pd-extended/`

cal version cyclone-0.2beta in December 2015. In February 2016, Kraan decided to abandon active development, but he still maintains the cyclone-0.2beta package. This version contains the original 166 cloned cyclone objects plus a `teeth~` object. The individual `hammer` and `sickle` libraries were no longer compiled by default. Kraan also fixed several bugs and revamped the documentation.

2.2 Cyclone Repositories

The original Cyclone library was a sub-library of the `miXed` library, which was developed by Cyclone's original author Krzysztof Czaja.[3] Kraan's forked git repository hosts versions cyclone-0.1-alpha57 and cyclone-0.2beta, and employs Katja Vetter's pd-lib-builder.[4]

2.2.1 Our Repository

In February 2016, as a response to Fred Jan Kraan's decision to cease development and provide maintenance only, Porres forked cyclone-0.2-beta1 to a new git repository to continue Cyclone development.[5] Our branch is in active development, independent of the original developer and subsequent maintainers.[6] Our intent is to continue work on Cyclone without diverging from its original goal and scope. Krzysztof Czaja could not be reached to discuss our plans for Cyclone development, but Steiner created ground rules for Cyclone development on the Pd-list:

> About maintaining cyclone, I think a reorg would be great, and further maintenance as well. If you want to do whatever you want with it, then just make a fork and work on it as a new name. If you want to stick to cyclone's central goal of Max/MSP compatibility, then keep working on it as cyclone. But please do not work on cyclone and break the Max/MSP compatibility. [3]

Because the Cyclone library has a very clearly defined scope – it consists only of cloned objects from Max/MSP – we have been able to remain faithful to its central goal. Thus our intent is to continue developing this library as Cyclone rather than as a fork with a new name. We do not wish to impose any personal view on or control over the project; on the contrary, we consider Cyclone to belong ultimately to the Pd community and not to any particular maintainer. We are therefore eager to collaborate with anyone who wishes to help with fixing bugs and coding new objects. Before releasing an official version of our work, we wish to discuss the future of Cyclone with the Pd community. With community agreement we plan to release our project as cyclone-0.3, with an alpha test version ready to demo at PdCon16.

3 Cyclone 0.3

Cyclone was originally developed in the Max 4 era, and Cycling '74 has since introduced new functionalities from Max versions 5 to 7. Cyclone was compliant only with Max 4 through version alpha56, but when Kraan took over he was open to including features from Max 5. The current canonical version of Cyclone (0.2beta) is still for the most part compliant only with Max 4, with a few exceptions (for instance, the `clear` method of `delay~` was new with Max 5 and is implemented in cyclone-0.2beta).

Our primary goals for Cyclone version 0.3 have been updating the existing codebase to reflect new features from Max versions 5-7 (up to 7.3.0 as of this writing), cloning important objects currently missing from Cyclone, fixing bugs in Cyclone code, and addressing bugs and errata in Max objects and documentation. In order to make the project more accessible for users and developers, we have completely rewritten the Cyclone documentation, restructured the codebase, and improved the build process.

3.1 Updating Cyclone to Max 7

Updating existing Cyclone objects to Max 7 compatibility and writing documentation to reflect this has received by far our most intense focus and hardest work. A careful analysis showed that 54 objects required an update to be compliant with

[3]There are two `miXed` repositories, a sourceforge.net site, `https://svn.code.sf.net/p/pure-data/svn/trunk/externals/miXed/`, and a migrated git repository `https://git.puredata.info/cgit/svn2git/libraries/miXed.git/`

[4]Kraan's git repository is `https://github.com/electrickery/pd-miXedSon` (note that although it is called pd-miXedSon, only the Cyclone library has been forked). Vetter's pd-lib-builder is available at `https://github.com/pure-data/pd-lib-builder`.

[5]Our branch is `https://github.com/porres/pd-cyclone`

[6]Kraan graciously links to our branch as the one in active development.

recent Max versions.[7] This work is nearly complete as of this writing.[8]

Cyclone's relationship to Max 4 is complicated. Max 4 was quite long lived, with updates from 2001 to 2007, which covered versions 4.0 to 4.6 (Max 5 was released in 2008). Since Cyclone was first released in 2002 with compatibility to Max 4.0, changes in Max after version 4.0 were not always reflected in the Cyclone code. For instance, the second argument to `coll`, which prevents the object from searching for a file with the symbol given in the first argument, was introduced in Max 4.0.8, but was never implemented in Cyclone.

Moreover, some features from Max 4.0 never made it into Cyclone objects (e.g. the missing "symout" argument to `sprintf`), and a number of bugs persist. We have treated the various missing features from Max 4.0 to 4.6 as bugs, since cyclone-0.1-alpha56 contained features from Max 4.6 and seemed to be incomplete on its own terms. Therefore we have not counted them among the 54 objects requiring update from Max 4 to Max 5+ compatibility. Part of this distinction is academic, since most of the objects missing features from Max 4 also needed updates to bring compatibility to Max 5+ (exceptions include `funbuff`, `mousestate`, `substitute`, and `slide~`).

To put this in perspective, we note that 54 objects is just under one-third of the 166 originally cloned objects, so the majority of them have not changed in Max from versions 4.6 to 7.3. Max/MSP objects have tended to reach a level of stability and maturity after time, but sometimes old bugs and kludges have also tended to be retained in mature objects (see **3.2.4 Inconsistencies** below).

Fortunately, nothing has been introduced in newer versions of Max that makes an update of Cyclone impractical. In fact, the major feature additions to Max have come in the form of large packages such as Jitter, Gen, and JavaScript capability, which can safely be left out of Cyclone. Most of the major changes to objects between Max 4.6 and 7.3 occurred in versions 5 and 6. `scope~` is the only object currently cloned in Cyclone to change in Max from versions 6 to 7 (this was merely a difference in default color scheme); only `midiparse` and `midiformat` received updates between Max 7.0 and Max 7.3.0, requiring updates in Cyclone.

3.1.1 Attributes

Some updates to Max cannot be as easily addressed as the addition of methods to existing objects, for example. Max "attributes" illustrate how our work brings some Max semantics into Cyclone and make them compatible with Pd. Attributes in Max function similarly to property settings and option flags in Pd. Although present in a few object classes from Max 4.5, attributes began to proliferate among Max/MSP's object classes in Max versions 5 and 6. [4] Property windows exist for GUI objects in Pd, and some classes (e.g. `declare` and `sigmund~`) take flags to control how the object operates. In current Max versions, many object settings can be controlled with attributes, using the special attribute syntax `[object @attribute-name <setting>]`.

In addition, every object has a set of attributes called "common box attributes," which set the look and feel of the object box either via `@attribute` flags to the object or a graphical menu in the inspector. Since our purpose is only to clone the function of Max classes and not their appearance, we have opted not to include support for common box attributes. While Pd's `-flag` options are syntactically identical to Max's `@attribute` options, we decided to use Max-style attributes rather than Pd-style flags for this kind of option setting in Cyclone. It is faithful to the Max compatibility goal and it provides Max users with a familiar environment.

In Max, once an object is instantiated, attributes can be set dynamically in three additional ways: first, via the inspector, second with a special attribute-editing object `attrui`, and third by sending the object a message with the attribute name and its argument(s). Pd does not have an inspector, but rather calls dedicated properties windows for GUI objects; therefore for code simplicity we plan to make properties windows only for GUI objects like we did with `scope~`, leaving the other objects with only `@attribute` flags. The `attrui` object inspects an object it is connected to for attributes and allows the user to set them from the patch. While such an object is possible in principle for Pd, we have decided to implement dynamic attribute setting via message passing only, saving any development of `attrui` for the future.

[7] An outlier is the `comment` object, which we are also updating. However, due to differences in GUI toolkits and platforms will likely never behave in exactly the same way as Max's, so it is not listed among these 54.

[8] Current progress, as well as a complete list of objects affected by our work, may be found at `https://github.com/porres/pd-cyclone/wiki/cyclone-0.3-changlelog`

```
while(argc > 0)
{
  if(argv -> a_type == A_FLOAT)
  {
    t_float argval = atom_getfloatarg(0, argc, argv);
    switch(argnum)
    {
      case 0:
        tripeak = argval;
        break;
      default:
        break;
    }
    argnum++;
    argc--;
    argv++;
  }
  else if (argv -> a_type == A_SYMBOL)
  {
    t_symbol *curarg=atom_getsymbolarg(0, argc, argv);
    if(strcmp(curarg->s_name, "@lo")==0)
    {
      if(argc >= 2)
      {
        trilo = atom_getfloatarg(1, argc, argv);
        argc-=2;
        argv+=2;
      }
      else goto errstate;
    }
    else if(strcmp(curarg->s_name, "@hi")==0)
    {
      if(argc >= 2)
      {
        trihi = atom_getfloatarg(1, argc, argv);
        argc-=2;
        argv+=2;
      }
      else goto errstate;
    }
    else goto errstate;
  }
  else goto errstate;
}
```

Figure 1. The attribute parsing loop from `triangle~`.

Introduction of attributes necessitates parsing an object's arguments as a list. If an object has more than one settable attribute, the `@attribute` flags may be called in any order, which requires a flexible constructor. As with Pd vanilla objects that allow flag options, elements of the argument list are parsed one-by-one, comparing any symbols to the symbols in the list of `@attribute` possibilities for that object. If the symbol does not match any attributes, then it may be considered a normal creation argument, such as the name of an array. If neither condition holds, or if the requisite number of arguments to the `@attribute` are not present, the constructor lands in an error state and the object does not instantiate. **Figure 1** is an example of an attribute parsing loop from `triangle~`, which has two possible attributes, `@lo` and `@hi`.

3.1.2 "Magic"

The difference between Max's patching syntax and Pd's is a common obstacle for Max users who are new to Pd. These differences make things difficult for a Max compatibility library. Some of them are intractable on the Pd side without making changes to the Pd core; for instance, Pd message fanouts require `trigger` to operate in proper order while Max controls order by spatial placement of objects. Others demand a hybrid approach; for instance Max does not share Pd's commitment to deterministic depth-first processing. For instance, `coll` can load large files in the background while other control and signal operations continue; in Cyclone, `coll` can be made to be deterministic if desired, or to use a threaded file-loading function that breaks determinism.[9]

There are also differences in signal inlet behavior between Max and Pd that can be addressed from within the external class. In Pd's tilde objects, secondary signal inlets contain a float field that can be set by an incoming float. If the inlet has an incoming signal connection it will ignore that float field. If it does not, Pd writes the value of that float field to its inlet's signal vector, thus promoting floats to signals. The object's behavior is the same in each case. In Max/MSP, floats are usually promoted to signals, but sometimes they are ignored or used to set specific parameters whether or not a signal is connected. Likewise some MSP objects have different internal behaviors depending on whether a signal is connected to one or more of its inlets, regardless of whether floats are promoted to signals.

This discrepancy is difficult to resolve because Pd and Max users have different expectations, and Cyclone has to fulfill commitments to both. Past Cyclone versions have favored matching the behavior of Max objects' inlets whenever the Pd API makes it possible, and so we have decided to continue this practice and extend it. Cyclone contains a collection of shared "unstable" modules which work by adapting Pd's API in novel and sometimes obscure ways.

The three primary modules involved are `forky`, which contains sections of conditionally compiled code that preserve functionality when Cyclone is compiled against different versions of Pd; `fragile`, which contains functions that depend on specific

[9] In principle the latter is not much different from employing a random `delay` in a message chain.

```
int forky_hasfeeders(t_object *x, t_glist *glist,
    int inno, t_symbol *outsym)
{
    t_linetraverser t;
    linetraverser_start(&t, glist);
    while (linetraverser_next(&t))
    {
        if (t.tr_ob2 == x && t.tr_inno == inno
#if FORKY_VERSION >= 36
        && (!outsym ||
            outsym == outlet_getsymbol(t.tr_outlet))
#endif
        )
        return (1);
    }
    return (0);
}

/*-----------------------------------------------------*/

static void scope_dsp(t_scope *x, t_signal **sp)
{
    x->x_ksr = sp[0]->s_sr * 0.001;
    int xfeeder, yfeeder;
    xfeeder = forky_hasfeeders((t_object *)x,
        x->x_glist, 0, &s_signal);
    yfeeder = forky_hasfeeders((t_object *)x,
        x->x_glist, 1, &s_signal);
    scope_setxymode(x, xfeeder + 2 * yfeeder);
    dsp_add(scope_perform, 4, x,
        sp[0]->s_n, sp[0]->s_vec, sp[1]->s_vec);
}
```

Figure 2. `forky_hasfeeders()`, **and its call in** `scope~`.

implementation details of Pd (e.g. from `m_imp.h`) and which are likely to break in new versions of Pd; and `fringe`, which contains functions deemed likely to be included in Pd's official API in the future.[10] In personal correspondence about the project, we have begun calling this collection of novel functions "magic tricks."

A good illustration of the magic tricks and how they are applied is the signal visualization class `scope~`. This object has two signal inlets and behaves differently depending on how the inlets are fed.[11] When only the left inlet has signal input, the signal is drawn with time plotted on the horizontal axis and amplitude on the vertical; this arrangement is reversed when only the right inlet has a signal, with time on the vertical axis and amplitude on the horizontal. When both are connected with a signal, the object goes into "X-Y mode," and plots the signals parametrically on a Cartesian plane. When neither inlet is connected, the object draws a flat horizontal zero-amplitude line. Furthermore, neither inlet promotes floats to signals: floats in the right inlet set the number of signal points displayed in the display buffer, and floats in the left inlet set the number of signal samples used to draw each point. The left inlet also accepts a number of method messages.

In order to alter the behavior of an object based on its signal input configuration, Cyclone classes use the function `forky_hasfeeders()` (see **Figure 2**). When called from within an object, this function traverses the object's canvas connections until it finds the object and the relevant inlet, and then checks whether it is connected, and if so whether that connection is a signal connection. It is called in the `dsp()` routine, and can be used to set options or load different `perform()` functions based on the return value.

Keeping the two signal inlets from promoting floats to signals is more difficult, and the procedures involved are newly introduced into Cyclone classes. These procedures differ for left and right inlets. Left inlets in Pd have an underlying infrastructure that automatically allows them to receive both signals and messages, which is usually accessed through the `CLASS_MAINSIGNALIN()` macro. In the usual case, incoming floats to the left inlet sets a float member of the class's struct, which are internally promoted to signals whenever no signal connection is attached. This can be overridden by giving the class both a signal and a float method, as in this example from `scope~`:

```
class_addmethod(scope_class,nullfn,gensym("signal"),0);
class_addfloat(scope_class,(t_method)scope_float);
```

Incoming floats now call `scope_float()` instead of the default float method.

Doing the same with secondary inlets is more difficult because float inputs do not automatically call a class method, but rather set a float field in the inlet's struct. Direct access to this field is usually hidden, but the `obj_findsignalscalar()` routine from Pd's `m_obj.c` returns that field's address.[12] The class's `perform()` routine(s) can then poll that field for changes every tick and call a method if a change is detected. Here is how it works in `scope~`:

[10] This organization is both pragmatic and problematic, and is undergoing restructuring. See **3.2.1 Code Restructuring** below for more details.

[11] The following only applies when DSP is on.

[12] Cyclone's `fragile` module contains a similar routine, `fragile_inlet_signalscalar()`. We might phase this out because it requires maintaining a copy of the `inletunion` and `_inlet` declarations in `m_obj.c`, whereas using `obj_findsignalscalar()` only requires declaring it as an `EXTERN` with the proper arguments.

```
int bufsize = (int)*x->x_signalscalar;
if (bufsize != x->x_bufsize)
   scope_bufsize(x, bufsize);
```

3.1.3 Other Significant Revisions

We have made a number of especially significant revisions to some classes in the process of moving to Cyclone version 0.3. The following is a brief catalog of salient examples.

- Previous versions of `comb~` and `allpass~` all had an error in the difference equation that required a complete rewrite of their `perform()` routines, including the addition of secondary delay buffers. We added a new `teeth~` object based on `comb~` to replace a former abstraction.
- `cycle~` now has an internal 16384-point, fully symmetric cosine table so that, unlike with `osc~` and previous iterations of `cycle~`, frequency modulation is stable over time.
- We have implemented the extra interpolators that were added to `wave~` in Max 6. Since Pd's cubic Lagrange interpolator is not among them, but was included in previous versions of Cyclone, we have retained it as an option.[13]
- `delay~` now accepts signals to control samples of delay, with cubic interpolation.
- The bitwise signal operators `bitand~`, `bitor~`, and `bitxor~` have been revised to fix their second inlets (which set the objects' bitmasks) using the procedures described in the previous section. All of the type punning required by these objects has been rewritten for stability, replacing pointer casts with unions. Denormal output values are set to zero as they are in Max.
- `train~` now allows pulses of width 0 (resulting in single-sample widths) and width 1 (resulting in widths of one sample less than an entire cycle). The phase behavior has also been altered to respond correctly when changed.
- `fromsymbol` now allows any single- or multi-character string to act as delimiter.
- `funnel` has been almost completely rewritten so that lists are stored properly at each inlet and output correctly upon receiving a `bang` (previously only the first element of a list was stored and output on `bang`).
- `sustain` has been rewritten to accommodate two additional modes for handling repeated note-on messages, which were introduced after Max 4.
- `scope~` has undergone major surgery. It now supports the "Y-only" mode and the "alternate drawstyle" options. It has new methods allowing `rgb` values to be set according to a 0.0 to 1.0 float range. The "X-Y" mode has been simplified and its performance improved. Several bugs were fixed, including a crasher bug on x86_64. Finally, it has a new Pd-style properties window.

3.2 Beyond Max Compatibility

Our primary goal has been bringing existing Cyclone code up to full compatibility with Max 5+, but we have made various other improvements along the way. Besides updating the documentation, we have restructured the codebase, created a number of new cloned objects, and fixed bugs, many of them longstanding. We also addressed some bugs in the Max objects themselves.

3.2.1 Code Restructuring

Over the course of the update it became clear that the Cyclone codebase was in need of restructuring and streamlining. Fred Jan Kraan initiated this process in alpha57 and 0.2beta by importing pd-lib-builder and adjusting the build targets. As of 0.2beta (the current version), the sub-libraries `hammer`, `sickle`, and `maxmode` are not compiled by default. The `cyclone` sub-library has been renamed `nettles`, and it only loads the non-alphanumeric bin-ops (and not all of the control and signal objects).

We have taken the restructuring process several steps further. Given the increasingly wider divergence of Max and Pd since the Max 4 era, any effort to emulate Max as deeply as earlier Cyclone attempts seems forlorn. We therefore no longer keep `cyclist` or the legacy sub-libraries as build targets, but keep their code in a maintenance directory. We did, however, restore the `cyclone` sub-library, which loads the bin-ops and serves as a space for future implementation of some of Max's syntactic sugar (e.g. the `zl.mode` syntax). Since the sub-libraries are no longer in operation, all of the source files for the object classes have been moved from "hammer" and "sickle" directories to "control" and "signal" directories. This also helps newcomers by making the purpose of the directory structure clear.

The examples in previous sections illustrate the extent to which Cyclone relies heavily on code modules shared among the object classes. Many of the functions in these modules serve as thin

[13] The MSP `wave~`'s new interpolators are identical with the ones found in a 1999 web article by Paul Bourke, and they are included in the same order they appear in the paper. The Max documentation contains no attribution; Bourke's article is here: `http://paulbourke.net/miscellaneous/interpolation/`.

wrappers over the standard Pd API. The two most important are the `sic` and `arsic` (i.e. "sickle" and "array sickle") modules, which create `t_sic` and `t_arsic` types inheriting from Pd's `t_object`. These modules also implement custom creation methods such as `arsic_new()` and `sic_inlet()`, and provide routines for use in all signal object classes (`sic`) and all object classes that access Pd arrays (`arsic`). We have begun editing object classes to remove this cruft. By standardizing Cyclone to comport with the Pd API, we have rendered the codebase more transparent and made our new contributions much easier to code, especially for attributes that affect inlet instantiation and the "magic" code.

Eliminating dependence on the `sic` module has been relatively easy, amounting to replacing the wrapper functions with those from the standard Pd API. Disentangling from the `arsic` module has been much more difficult, because `arsic` emulates MSP's multichannel `buffer~` by allowing signal classes to read from and write to multiple Pd arrays (which are collected according to a naming convention). `arsic` also depends on the "vector of floats" submodule `vefl`, which has custom functions for gaining access to Pd array contents.

The necessary functions from `vefl` and `arsic` have been collected into a new "Cyclone buffer" module `cybuf`. This module introduces a new type, `t_cybuf`, which is similar to `t_arsic` except it has no `t_object` member. An object class that depends on `cybuf` can now keep its own `t_object` member, which used to be replaced with `t_arsic`. Now that the array functions in `cybuf` are separated from the standard Pd object creation methods, they can be readily replaced or supplemented if a proper `buffer~` class is developed, or if multichannel arrays are ever introduced in vanilla Pd.

As mentioned above, the code in the `forky`, `fragile`, and `fringe` modules is not collected according to similar function, but rather similar maintenance status vis-à-vis Pd's API. From a pragmatic standpoint, this does make maintenance somewhat easier because there is less disruption when there is a change in Pd. On the other hand, it makes development more difficult, especially for new contributors. One example that proves this point is inlet and outlet handling. In **3.1.2 "Magic"** we discussed the `forky_hasfeeders()` function, which returns the connection status of an inlet. There is a corresponding function that does the same for outlets, but it is found in the `fragile` module instead – `fragile_outlet_connections()`. There are many similar examples. We plan to reorganize these modules according to function, and in the process eliminate code that provides compatibility with very old versions of Pd. We hope that these and other improvements to the code structure will encourage participation from other contributors, who may not have wanted to learn an unnecessary and complicated wrapper API.

3.2.2 New Object Classes

Cyclone 0.3 introduces 45 new object classes that were not present in previous versions.

13 control classes: `acosh asinh atanh atodb dbtoa join loadmess pak pong rdiv rminus round scale`

32 signal classes: `atodb~ biquad~ bitsafe~ cascade~ cross~ dbtoa~ degrade~ downsamp~ equals~ filtercoeff~ freqshift~ gate~ greaterthan~ greaterthaneq~ hilbert~ lessthan~ lessthaneq~ modulo~ notequals~ number~ phaseshift~ plusequals~ rdiv~ rect~ rminus~ round~ saw~ scale~ selector~ thresh~ tri~ trunc~`

The bandlimited oscillators `rect~`, `saw~`, and `tri~` as well as the signal number box `number~` currently exist in Cyclone 0.3 as abstractions, but we are planning to implement them as externals in future versions.

3.2.3 Documentation

Czaja originally released Cyclone with no documentation. Many of the changes to Cyclone over the years were attempts to create good documentation. After careful scrutiny we found that much of the existing documentation had mistakes. Because we were already planning on a major documentation update for the new and upgraded objects, we made the decision to rewrite all of the documentation from scratch. This had the dual purpose of systematically testing every Cyclone object to ensure it complied with the specifications from the Max/MSP documentation.

In the process we found that the Max/MSP objects and documentation also had a number of bugs and inconsistencies, which made reverse engineering that much more difficult. Many objects had undocumented behaviors, so we had to make several decisions about whether a given undocumented behavior was a feature or a bug. We have

tried not to duplicate obvious bugs, and to thoroughly document the behaviors we have retained.

In some cases we have even rewritten the stated purpose of an object, where the Max documentation was either misleading or incomplete. For instance, the documentation for `wave~` bills it as a sample player, where in fact it is an all-purpose buffer reader that can be used for oscillating waveforms, waveshaping, etc. This major overhaul is still far from complete, however, and is likely to be what delays a transition from alpha to beta.

3.2.4 Inconsistencies

As discussed above, the many differences between Pd and Max have made cloning some features difficult. There are still several inconsistencies between some Max 7 objects and their Cyclone 0.3 counterparts. As of version 0.47.1, Pd lacks a multichannel array object, implemented in MSP as `buffer~`. The MSP objects `index~`, `peek~`, `buffir~`, `poke~`, `wave~`, `record~`, and `play~` all read from and/or write to `buffer~` objects. These objects in Cyclone 0.3 have been written to use the `cybuf` module. As mentioned above, the functions in `cybuf` can be replaced or supplemented by the introduction of multichannel arrays in Pd vanilla or a new `buffer~` clone. Backwards compatibility would probably indicate a hybrid approach here, with support for any and all of these options.

Another major inconsistency is that Pd lacks anything like Max's `transport` object, which provides a globally accessible scheduling clock and optional secondary clocks bound to symbol-defined names. The `transport` object is used to allow certain objects (e.g. `record~` and `delay~`) to define time intervals relative to a tempo rather than absolute terms. All of the classes which utilize `transport` in Max/MSP have been written without this functionality.

Due to differences in toolkit font rendering, the GUI object `comment` may never be fully compliant with Max. We invite developers who are proficient in tcl/tk to help make `comment` behave more consistently across platforms.

4 Roadmap

Here we discuss our vision for the future of Cyclone. We want to emphasize that because Cyclone belongs to the Pd community, all of the following is subject to revision according to the needs of the community.

4.1 Plans for New Code

In the near future, once we have reached a stage in development for a stable release, we plan to make Cyclone 0.3 available for all platforms via Pd's externals manager and in upcoming versions of Pd-l2ork and Purr Data. The source code will be available on GitHub as usual, and we hope also on `puredata.info`. As stated earlier, we welcome contributions from the Pd community in any form.

Despite our progress, Cyclone is far from complete; there are many Max/MSP object classes we plan on including in future releases. New GUI object classes and Max's more complicated control and signal object classes present the most exciting opportunities for future development.

Cyclone currently only has two GUI object classes (`comment` and `scope~`). Users making the transition from Max to Pd lament the dearth of GUI objects, so we hope to introduce more in future releases. Such classes include `radiogroup`, a one-dimensional grid of toggles; `matrixctrl`, a two-dimensional grid of toggles or knobs for use with `matrix~`; `rslider`, a slider which allows the user to select a number range; `kslider`, a visual piano keyboard representation; `multislider`, an object class similar to Pd's garrays, and `spectroscope~`, which displays a signal's spectrogram or sonogram. Any new development of GUI object classes should make as efficient use of tcl/tk as possible. A secondary consideration is porting to the nwjs toolkit used in Purr Data.

We have already mentioned `transport` and `buffer~` as potential new control and signal classes. The hash-table dictionary object `dict` would be an important contribution given the problematic accretion of features and bugs in Max's `coll`, which has similar functionality. We hope also to clone the signal object classes `groove~`, a user-friendly sample player, and `gizmo~`, which detects and shifts frequency peaks of an FFT analysis for pitch transposition. In any event, the Pd community's priorities ought to help shape future development.

4.2 Community Logistics

To conclude this paper it is necessary to discuss logistics of Pd community involvement going forward. First, some words about the legal ramifications of reverse engineering software are in order. Cyclone has always existed in a legal gray area. Although it employs names, syntax, and semantics from Max/MSP, the algorithms and pro-

cedures used in the code are the original work of the developers. It may also be problematic that it specifically purports to offer Max/MSP compatibility. In the unlikely event of legal trouble, we ought to have a contingency plan in place to salvage the library, and community input is welcome.

Second, there have already been a number of disagreements about the status and direction of this project, and a good-faith effort to resolve them are in the best interests of all involved. The primary disagreement has been about whether this project is best released as a new (forked) library with a new name, or as a new version of Cyclone taking over as the canonical development branch.

There are well-reasoned arguments for both positions. Here are some of the best reasons for releasing this update under a new name.[14] A supplemental library of new objects could be released under a new, similar name such as "Typhoon," and then users could install both libraries to get all the objects. If they do overlap, two non-identical libraries with the same name is sure to be confusing to users, while two libraries with similar goals but with different names would give users finer control over what they install and use. Forking projects to add new features under a different name is a common – even normative – development strategy in open-source software; after all, Pd has already had several forks of its own and users have managed to navigate this terrain. Finally, Cyclone's infrastructure is overly complicated, and a library of cloned objects might do better not to use it.

We respectfully disagree with much of this reasoning. It should be clear from the foregoing that our primary goal has been to bring the existing object classes from Cyclone up to date, not merely to supplement it with new classes. It would make some sense to release a separate library of new objects with a new name if it did not overlap in content with the old Cyclone, but name clashes resulting from the existence of different classes with the same name from libraries with different names (e.g. `counter` in Cyclone and Gem) is already a source of confusion for Pd users.

Suppose we were to release our work under a different name ("Recyclone," say), as a drop-in replacement for Cyclone. Recyclone would be identical to Cyclone in some respects, and its differences would be non-arbitrary because of the straightforward goal of Max/MSP compatibility. One obvious downside to this is that users who wanted to switch from Cyclone to Recyclone would need to change namespace settings in their existing patches. The larger point is that since the purpose of each branch differs only in that they target different versions of Max, they seem as much like predecessor and successor versions of the same software as the different versions of Max do. All of the predecessor's functionality is included in the successor's, with only minor changes.

The fact that two versions of the same library would be available via Pd's externals manager does not weaken our position. Indeed, multiple versions of many libraries are available, and it is up to the user to install whichever suits their needs. We believe that the disagreement is not about the simultaneous availability of two versions of the same library, but about the existence of two source branches with different maintainers/developers. This is more a social problem than a technical one. Provided that all parties are satisfied, we see no reason why cyclone-0.2 and cyclone-0.3 cannot exist side by side in the Pd ecosystem as a maintenance branch and development branch, respectively. While forking may be the normative strategy, forks make the most sense when each branch has ongoing diverging development.

Finally, we agree that Cyclone's system of modules is complicated; it is the primary motivation for reorganizing and streamlining the code. However, there is a great deal of useful code that need not be redesigned from scratch. We hope our efforts in increasing code transparency will encourage more participation from other developers.

Because many of the new Pd users in our community come from a Max/MSP background and tend to be daunted by Pd's steeper learning curve and relatively economical object collection, we believe that the project presented here will be an important first step toward preserving Cyclone's vital role in helping these users make that transition gracefully. However, we also believe that Cyclone is useful to Pd users in its own right. While we acknowledge that our development might overlap territory occupied by other libraries, we emphasize that Cyclone is a large library with many purposes outside of Max/MSP compatibility: it can replace many objects from unmaintained libraries, reduce the number of libraries a patch needs to load, or simply serve as a supplement to Pd vanilla. All of these purposes are significant justification for continual active development.

[14] These were all expressed in a discussion on The Pd-list from February 2016. See `https://lists.puredata.info/pipermail/pd-list/2016-02/113377.html`

5 Acknowledgements

There are many people we wish to thank. Krzysztof Czaja's initial effort was a large and worthy undertaking, and one of the most successful in the Pd world. Thanks to Hans-Christoph Steiner and Fred Jan Kraan for their stewardship; without them the library would likely be dead.

Katja Vetter's pd-lib-builder is an extraordinary tool for Pd developers, and has proved invaluable for understanding the structure of the Cyclone code and identifying opportunities for improvement.

We thank Ivica Ico Bukvic for the threaded version of `coll`, his ethic of active development, and his constant encouragement. Jonathan Wilkes's work on Purr Data will be very important to the future of Cyclone.

Thanks to Marco Matteo Markidis, Joel Matthys, and anyone else who has contributed code. Esteban Viveros and Flávio Schiavoni were instrumental in getting the process started on sure footing.

IOhannes m zmölnig merits gratitude for his leadership and advice. We thank David Zicarelli for his candid correspondence.

And of course we must thank Miller Puckette for inventing both Max and Pd.

References

[1] K. Czaja, "[PD] pd-max compatibility," April, 2002. `https://lists.puredata.info/pipermail/pd-list/2002-04/005949.html`

[2] K. Czaja, "[PD-announce] cyclone-0.1-alpha1," November, 2002. `https://lists.puredata.info/pipermail/pd-announce/2002-11/000139.html`

[3] H-C. Steiner, "[PD] Update cyclone maintenance," June, 2015. `https://lists.puredata.info/pipermail/pd-list/2015-06/110620.html`

[4] D. Zicarelli, "Max 5 and Attributes," October, 2007. `https://cycling74.com/2007/10/31/max-5-and-attributes/`

BiA: a digital library for music and acoustics

Felipe de Almeida Ribeiro
Universidade Estadual do Paraná
Rua Francisco Torres, 253
Curitiba, Brazil, 80060-130
felipe.ribeiro@unespar.edu.br

Clayton Rosa Mamedes
Universidade Federal do Paraná
Rua Coronel Dulcídio, 638
Curitiba, Brazil, 80420-170
claytonmamedes@gmail.com

Pedro Samsel Geraldo
Universidade Estadual do Paraná
Rua Francisco Torres, 253
Curitiba, Brazil, 80060-130
ps.samsel@gmail.com

Abstract

This paper introduces BiA, an ongoing library of patches for musical acoustics. This research intends to establish an experimental approach that deals with musical acoustics theory. Pd presents as an affordable solution for students, especially compared to similar professional platforms. BiA is developed having in mind a specific audience: students who are unfamiliar with computer as a potential creative tool. Although the whole project focuses on musical acoustics, it is also expected that users start to manipulate Pd in a spontaneous way, as we feel that it presents a powerful tool for diffusion of experimental electroacoustic music in general.

Keywords

Pure Data, Computer Music, Musical Acoustics.

1 Introduction

Musical Acoustics is a course offered in almost every music undergraduate program. It is a field that includes some hard mathematical and physical concepts, which sometimes may result in resistance from students that are not acquainted with these skills in their everyday activities. In our experience, we have noticed some students have difficulties envisioning the topic conceptually. In this context, we have developed BiA[1] (*biblioteca de acústica* in Portuguese, which stands for acoustics library), a library of patches to support teaching activities.

The main objective of this research is to allow students to experience contents introduced in class. In that way, topics on acoustics and music theory could be audibly verified, making the process of comprehension easier. As stated by Andrey Savitsky, "(...) working with Pd becomes an exciting game: a game where the rules are created and changed on the spot by the player; a game where the process of playing may be much more thrilling than the outcome, while the outcome may be unpredictable and quite surprising." [11]. We believe that this strategy can enhance the learning process.

As a strategy to assist students in Brazil, BiA has been developed mainly in Portuguese[2] and applied to the context of experiences we have had at the *Universidade Estadual do Paraná* (Brazil). It is intended as supporting material to our classes, helping to establish a connection between theory and practice. There are other libraries developed in Pure Data with similar intentions to the one we are presenting. The most known initiative – and by far the most complete and advanced in content – is developed by Miller Puckette for his own acoustic course, taught at University of California, San Diego (UCSD) and freely available online[3]. The tools we present in this paper focus on complementary aspects of Puckette's work, especially on psychophysical and perceptual aspects of musical acoustics. Another initiative is the work of Alexandre Porres [8] towards a creative approach to the topic on *Dissonance Psychoacoustic Models*. Finally, the Pd tutorials themselves cover some particular topics in acoustics, such as Shepard Tones or Fourier analysis and resynthesis. BiA, on the other hand, focuses on a different target group and pedagogical content. It is mainly developed to intersect with the literature by [3], [5], [1], [13], and [9] – all in Portuguese. It also keeps in mind users that have never had contact with Pure Data. This is part of a dual strategy of teaching. On the one hand, the simplicity of control one might observe in these tools is a consequence of the contextual aspect of its application. Concerning this underlying aspect, our main objective is to allow students to intuitively operate these patches[4]. On the other hand, this is a strategy to familiarize users with Pure Data, allowing them to participate in advanced computer music courses. Lastly, BiA complements our classes in the sense of providing

1 BiA can be downloaded at: http://www.nucleomusicanova.com/BiA_pdcon16.zip

2 For the Pdcon16~, patches were translated to English.

3 http://msp.ucsd.edu/syllabi/170.13f/index.htm

4 Considering that our main public has never used Pd, we avoided using external MIDI controllers in order to decrease the complexity of patching and hardware configuration. Additionally, this strategy enables students to work on their assignments with their own computers, without requiring external gear.

a practical experience for the students based on the selected textbooks for the course.

2 Development and implementation

At the present development stage, BiA explores topics on microtonality, room acoustics, and psychoacoustics. Since the library is freely available on our research group's website, this section is dedicated to describe the elementary concepts involved in each case. For researchers interested in a 'how it works' perspective, models and equations can be found inside each patch. We recommend users to wear headphones, especially for patches related to psychoacoustics. The next tables present a brief description of BiA:

Microtonal Tools	**Objective**	**How it works**
Cents Analysis	Output interval in cents based on any given frequencies.	Students can choose any pair of frequencies and the output will be given in cents format.
Diatonic scales	Allow the perception between two diatonic melodic scales: Pythagoras and Zarlino.	Students can use the laptop/desktop keyboard as a controller in order to trigger each scale degree.
Temperament	Allow the perception between two melodic temperaments: Werckmeister and Rameau.	Students can use the laptop/desktop keyboard as a controller in order to trigger each scale degree.
Chord builder	Build microtonal chords.	Students can use the laptop/desktop keyboard as a controller in order to trigger each chord degree.

Table 1: Description of Microtonal tools.

Room Acoustics Tools	**Objective**	**How it works**
Stationary Wave Generator	Create sound stimulus to verify and reveal the different standing waves modes of a room.	Students can calculate and listen to different standing wave modes by providing room dimensions and temperature.
Schroeder diffuser	Display a visual model of a Schroeder Diffuser.	Users can calculate Schroeder diffuser dimensions and get a visual model of its construction according to three different prime numbers.

Table 2: Description of Room Acoustics tools.

Psychoacoustics Tools	**Objective**	**How it works**
Just Noticeable Difference	Verify the perception of changes in sound level based on the difference of amplitude between L-R channels.	Students can select waveform, frequency and panning. JND levels are displayed in decibels.
Virtual Pitch	Induce the perception of a fundamental pitch by playing upper partials.	Students can select a fundamental frequency and control the amplitude of each partial on a bank of sinusoidal oscillators. Presets of timbres are available.
Aural Harmonics	Verify the perception of aural	Students can select a fixed fundamental

	harmonics resulting from the difference between two frequencies.	frequency. A slider allows frequency changes on a second oscillator. The first three aural harmonics can be visualized in arrays and number boxes.
Stevens Effect	Verify the perception of changes in pitch based on different levels of sound amplitude.	Students can select a frequency and control its amplitude through a slider.
Pitch Perception	Verify the perception of pitch according to a sound grain's frequency and duration.	Through sliders, students can change frequency, duration, and amplitude. Different waveforms and envelopes are available.
Loudness	Verify the perception of changes in amplitude changes according to Fletcher & Munson's theory. Implemented accordingly to ISO 226:2003 specifications.	Students can select two ways to control amplitude: fixed or changeable according to perception curves. In changeable mode, sound amplitude is continuously adjusted to match perception curves.
Critical Bands	Verify the perception of dissonance between two different frequencies.	Students can control two sliders related to two superimposed frequencies. Differences of frequencies are displayed both in Hertz and percentage. Presets of different waveforms are available.
Auditory Masking	Verify the perception of a sound being hidden by another of different frequency and amplitude.	Students can control frequency and amplitude of two superimposed oscillators. Presets of different waveforms are available.

Table 3: Description of Psychoacoustics tools.

For pedagogical reasons, we have adopted a common graphic layout for all elements of the library. Controls were implemented using sliders, radio selectors and buttons. Results are plotted into number boxes or graphic arrays, depending on the context. Regarding the different operational systems and Pd distributions, a decision was made to only use the Pd Vanilla distribution, without external libraries. This choice facilitates the distribution of BiA among users non-familiarized with computers and specific installation procedures. The following paragraphs will discuss the library's content.

2.1 Microtonal tools

BiA has a section dedicated to the exploration of microtones, including historical and acoustic perspectives. The first patch *cents_analysis* presents to the user the concept of frequency intervals based on the microtonal resolution: based on any given two frequencies, the user can calculate microtonal intervals. The patch is based on the equation: *cents = (1200/log2)*log(f2/f1)*, whereas *f2* and *f1* are two given frequencies [10].

The next patch presents melodic differences between German and French Baroque temperaments, more specifically Werckmeister's fifth temperament and Rameau's second temperament [4]. The aim is to introduce the world of temperament, especially to performance students, in order to encourage a discussion on performance practices and authenticity, topic discussed on graduate programs of music performance.

The third patch presents two melodic *diatonic scales* based on ancient tuning systems: Pythagoras and Zarlino [4]. The main goal is to engage with questions on how music from the Medieval times and Renaissance eras sounded like. There is also a pedagogical aim to link the content with well known counterpoint treatises, such as Johann Joseph Fux's *Gradus ad parnassum* published in 1725 and Knud Jeppesen's *The polyphonic vocal style of the Sixteenth century* from 1931.

Finally, the fourth patch *chord_builder* simulates microtonal polyphony, i.e. the user can construct and play any chord based on MIDI notes (middle C = 60) and with cents shifting. The idea is to work with a *tetrachord* in any tuning system available. This patch can also be used for psychoacoustics experiments.

2.2 Room acoustics tools

This section is composed of two patches for room acoustics measurements. The first one is a *Standing Wave* generator that allows students to simulate, hear, and experience the effects of standing waves in any "shoebox" room (with parallel walls, ground and roof), also taking temperature in consideration. Waves are generated in nine different frequencies and in three different room modes: *Axial*, *Tangential* and *Oblique*, giving a total of 63 different standing waves for each room.

The second patch attempts to solve the problem detected on the first one: to improve the room frequency response. By given parameters, it generates data to create a *Schroeder's Diffuser*. The main goal is to improve internal room acoustics. In addition, users can review how to build a sound diffuser.

2.3 Psychoacoustic tools

In this section we present eight patches related to psychoacoustic properties. The first one explores the concept of *Just Noticeable Difference* [15], [3], [9] and allows students to explore differences of sound level between left and right channels. The intention is to experiment with the perception of amplitude changes. It is possible to define a waveform as sine, triangle, saw tooth or square. They can also test different frequencies in the scale between 100Hz and 1kHz. The patch displays the difference in decibels between left and right channels and amplitude.

Virtual Pitch, the psychoacoustic effect of perceiving a fundamental frequency either if it is not present [6], [9], [3], was implemented through a patch that allows students to control the volume of each partial component of timbre. It was implemented through a bank of oscillators generating a set of predefined timbres accessible through a radio selector. Students can change the fundamental's frequency and explore different configurations of amplitude for each partial, decreasing or increasing its presence.

The effect of *Aural Harmonics* [14], [3], [9] generates a set of harmonics that are not present in the original signal, which results from the difference between two sinusoidal frequencies. The patch allows students to select a fixed frequency and continuously increase a second one through a slider. According to the combination of frequencies, number boxes and graphic arrays plot the relative position of the three salient harmonics.

The patch that presents the *Stevens Effect* describes the perceived variation of pitch to a fixed frequency as its amplitude increases. Students can explore this effect by selecting frequencies mentioned in [12] and [3] to audibly perceive the frequency deviation.

The patch that explores the *Perception of Pitch* [7], [3], [9] generates constantly timed grains as sound stimulus. These grains can be associated to different waveforms and envelope curves. It allows students to combine frequency, amplitude and duration of the sound stimulus. Students can try different combinations in order to audibly verify that higher frequencies are perceived in smaller time grains, as well as waveforms that result in more complex spectral distributions are also easier to perceive.

The patch that explores the concept of *Loudness* [3], [9] allows students to audibly verify two different approaches to sound level perception. The first one presents fixed amplitude, independently of frequency changes. The second one applies Fletcher and Munson's [2] curves on sound level perception, compensating the reduced ability of human beings to perceive frequencies near to the threshold of hearing. Curves were implemented based on ISO 226:2003[5] specifications that updated Fletcher and Munson's studies. Concerning those ISO 226 specifications only considers frequencies in the range of 20Hz to 12.5kHz, the patch features a slider to control frequencies restricted to this range. To better illustrate this phenomenon, each time students select a new amplitude level a graphic array displays the corresponding ISO 226 values. This array is also the buffer we use to control amplitude in order to match loudness curves. Students can change parameters 'on the fly', allowing them to compare differences between the amplitude and their variation through frequencies.

The psychoacoustic effect of *Critical Bands* or the perceived dissonance between two close frequencies [15], [9], [3] was implemented through a patch that allows users to change two sliders, each one associated to an individual frequency. The patch reveals that critical bands are better described as percentage than as

5 Available in: *http://www.iso.org/iso/catalogue_detail.htm?csnumber=34222*. Last access in August 13th, 2016.

absolute difference in Hertz.

Auditory Masking is the inability to perceive a first sound due to the presence of a second one, both distinct in frequency and amplitude values [14], [3], [9]. The implemented patch allows students to set up through radio selectors a first pair of frequency and amplitude. Afterwards, they can control a second pair of frequency and amplitude through sliders, having as goal to mask the first one, making it unperceived. Other waveforms with more complex spectral configurations are available for experimentation.

3 Final considerations

As a flexible and free software, Pure Data allows us to implement and easily distribute this library. This project goes in direction to decrease the gap between socioeconomic situation and learning possibilities. This is an important aspect to our research since most of our users are low-income students. Within this context, development of BiA's main intention to support pedagogical activities. With this in mind, we developed a library of patches to support our course in musical acoustics. The general community is welcome to download our material and to join the creation of new tools with our research groups: *Grupo de Pesquisa Núcleo Música Nova* (CNPq/UNESPAR) and *Grupo de Pesquisa em Criação Musical, Gesto e Processos de Interação Artística* (UFPR). For future development, we intend to: 1) increase the number of patches within the mentioned topics; 2) develop tools for the acoustics and design of musical instruments; 3) develop tools for analysis and acousmatic composition; 4) expand the application of this library to support high-school teaching activities as a tool to illustrate aspects of physics.

6 Acknowledgements

We would like to extend our heartfelt thanks for the financial support of the following institutions: CAPES (Process 1593232/2016) and UNESPAR (Process 03/2015 *Pró-Reitoria de Pesquisa e Pós-Graduação*).

References

[1] Cysne, L. F. O. *A Bíblia do Som,* Cysne Science Publishing Co., USA 2006.

[2] Fletcher, H.; Munson, W. A.. "Loudness, its definition, measurement and calculation." *Bell System Technical Journal*, vol. 12, no. 4, pp. 377–430, 1933.

[3] Henrique, L. *Acústica Musical*. Ed. Fundação Calouste Gulbenkian, Lisboa, 2002. ISBN 972-31-0987-5.

[4] Jedrzejewski, F. *Mathématiques des systèmes acoustiques: Tempéraments et modeles contemporains*. L'Harmattan, Paris 2002.

[5] Menezes, F. *A acústica musical em palavras e sons*. Ateliê Editorial, Cotia 2004.

[6] Plomp. R. "Pitch of Complex Tones." *Journal of the Acoustical Society of America*, vol. 41, pp. 1526-1533, 1967.

[7] _____ *Aspects of Tone Sensations*. Academic Press, New York 1976.

[8] Porres, A. T. "Dissonance Model Toolbox in Pure Data." *Proc. of the 4th Pure Data Convention*, Weimar, 2011.

[9] Roederer, J. G. *Introdução à Física e Psicofísica da Música*. Ed. Edusp, São Paulo 1998. ISBN 85-314-0457-6.

[10] Rossing, T.D.; Moore, F. R.; Wheeler, P.A. *The science of sound*. Addison-Wesley, Reading 2002.

[11] Savitsky, A. "An Exciting Journey of Research and Experimentation." *Bang: Pure Data*. Fränk Zimmer (Org.). Wolke Verlag, Hofheim 2006.

[12] Stevens, S. S. "The Relation of Pitch to Intensity." *Journal of the Acoustical Society of America*, vol. 6, no. 3, pp. 150-154, 1935.

[13] Vale, S. *Manual prático de acústica*. Ed. Música & Tecnologia, Rio de Janeiro 2009. ISBN 978-85-89402-14-2

[14] Wegel, R.L.; Lane, C. E. "The Auditory Masking of One Sound by Another and its Probable Relation to Dynamics of the Inner Ear." *Physical Review,* vol. 23, pp. 266-285, 1924.

[15] Zwicker, E.; Flottorp, G.; Stevens, S.S. "Critical bandwidth in loudness summation." *Journal of the Acoustical Society of America*, vol. 29, no. 5, pp.548-557, 1957.

Pure-data-based transaural filter with range control

Julián Villegas
Computer Arts Lab., University of Aizu
Tsuruga, ikki-machi
Aizu-Wakamatsu, Japan, 965-8580
julian@u-aizu.ac.jp

Takaya Ninagawa
Computer Arts Lab., University of Aizu
Tsuruga, ikki-machi
Aizu-Wakamatsu, Japan, 965-8580
s1210015@u-aizu.ac.jp

Abstract

We present an extension to Pure-data by which users can truly spatialize sound via a pair of loudspeakers, i.e., spatialize monaural sound sources at an arbitrary azimuth, elevation, and distance. Although transaural techniques have been long explored, our system takes advantage of a recently collected Head-Related Impulse Response (HRIR) dataset measured in the near field ($20 - 160$ cm from the center of a mannequin's head) to allow a more accurate distance control, a missing feature in other implementations.

Keywords

Transaural, spatial sound, HRIR, near field, virtual environments.

1 Introduction

Situations arise where the use of headphones for spatial sound reproduction is less convenient than loudspeaker reproduction: fatigue, heat, unawareness of the surroundings, etc. are some of the disadvantages associated with wearing headphones; in the same vein, users wishing to share a colocated experience find it more difficult when sounds are displayed via headphones. The tradeoffs of using loudspeakers include a narrower and more fragile sweet-spot, cumbersome installation, etc.

Several techniques such as Vector-Base Amplitude Panning (VBAP) [16], Ambisonics [13], and transaural audio [8] have been discussed in the literature for spatializing sounds over loudspeakers [2]. In this research, we focus on the latter, a simple technique that requires fewer loudspeakers than the mentioned alternatives. This feature makes transaural audio suitable for applications such as gaming, video watching, car navigation systems, etc., i.e., applications where users do not move (rotate or displace) their heads to a great extents.

Modern transaural audio relies upon the ubiquitous pair of stereo loudspeakers for reproduction and digital signal processing for canceling the deleterious effect from a given loudspeaker signal arriving to the contralateral ear (i.e., cross-talk cancelation). This idea was first put forward by Schroeder and Atal [18] and further advanced by Cooper and Bauck [3]. Several real-time spatializers based on transaural audio have been developed as summarized by Gardner [8], for example.

Transaural audio inherits the benefits and shortcomings of the Head-Related Impulse Response (HRIR) dataset it is built upon: generic datasets (e.g., captured with a mannequin) yield less accurate aural images than those obtained when personalized datasets are used instead. Moreover, many of the existing datasets have beed captured at a fixed distance, usually ≥ 1 m because at closer distances the loudspeaker cannot be considered a point source anymore. Thus, to convey distance in HRIR-based systems, attenuation rules based on the inverse distance law are usually used (i.e., in ideal conditions, the sound pressure is halved as distance from the listener is doubled). This approximation is not valid for sound sources in the near field [6].

To achieve transaural audio with distance control in Pure-data, we propose filtering monophonic signals using `hrir~` [20], a Pure-data extension that convolves an incoming audio stream with a pair of impulse responses corresponding to an arbitrary azimuth, elevation, and distance. The impulse responses are a modified version of those obtained by Qu et al. [17], who capture them at distances as close as 20 cm from a mannequin's head. Subsequently, the binaural signal is treated to eliminate the influence of the contralateral loudspeaker with a filter in a shuffler topology [8] that we called `trans~`.

In what follows, we describe the underlying techniques used in our work, the implementation of a transaural audio library in Pure-data along

with its limitations, and further discuss future development of the implemented library.

2 Background

2.1 Head-Related Transfer Functions

Sounds are modified (i.e., reflected, delayed, filtered, etc.) by a listener's head and upper body. These modifications are different for each ear and depend on the relative location of a sound source and a listener, the attitude of the listener (i.e., rotations around three perpendicular axes), etc. These transformations, known as Head-Related Transfer Functions—HRTFs, can be obtained by recording Impulse Responses (HRIRs) from each ear at different angles and distances and can be used in virtual environments to locate sounds.

Several implementations of HRTF filters exist in the Pure-data environment, e.g., `earplug~` [23], `iem_bin_ambi~` and similar libraries [15], and `CW_binaural~` [5]; we decided to use `hrir~` [20] since this library is unique in allowing the projection of aural images at different distances in the near field. This truly spatialization is achieved by using an HRTF database based on that published by [17] which comprises measurements at different azimuths, elevations and ranges ($20 - 160\,\text{cm}$ from the center of a KEMAR Head And Torso Simulator—HATS).

2.2 Cross-talk

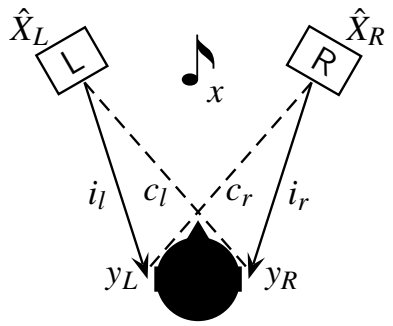


Figure 1: A monophonic signal x is spatialized (X_L and X_R—not shown) and displayed through a pair of stereo loudspeakers which driving signals are indicated as $\hat{X}_L$ and $\hat{X}_R$. Ipsilateral signals i_l, i_r are mixed in the air with the contralateral ones c_l, c_r producing a deleterious effect on the virtual image.

HRTF filters are effective when the processed signal is presented via headphones since they ensure that each ear receives its corresponding channel, however when the same signal X is presented over a pair of stereo loudspeakers, the loudspeakers imprint their own impulse responses on the desired signal, but more importantly, contralateral loudspeaker signals (c_l, c_r) get mixed with the ipsilateral ones (i_l, i_r), effectively destroying the intended audio image, as illustrated in Figure 1.

The cross-talk between the ipsilateral and contralateral signals can be mitigated, prevented, or canceled in many ways, for example, by placing auditory barriers in the mid-sagittal plane in front of the listener [4], which is rather cumbersome for many applications, placing the transducers very close to a listener's ears [21], using high directional loudspeakers or ultrasonic beams [12], or by means of digital signal processing [8], as explained in the following section.

2.3 Transaural

Transaural techniques refer to the use of filters which cancel (or mitigate to a great extent) the spatial effects of the loudspeaker locations on the desired signal. These techniques were first devised by Schroeder and Atal [18], but several alternatives have been proposed since then (see [24] and [9] for an extensive review). In general, these binaural techniques attempt to preserve the virtual image by conceiving a pair of virtual headphones, i.e., eliminating not only the cross-talk between loudspeakers but their ipsilateral effect as well. Using matrix notation, the system can be described as

$$\mathbf{y} = \mathbf{H}\hat{X}, \tag{1}$$

where

$$\mathbf{y} = \begin{bmatrix} y_L \\ y_R \end{bmatrix}, \mathbf{H} = \begin{bmatrix} H_{Li} & H_{Rc} \\ H_{Lc} & H_{Ri} \end{bmatrix}, \hat{X} = \begin{bmatrix} \hat{X}_L \\ \hat{X}_R \end{bmatrix},$$

are the signals present at the listener's ears, the HRTFs imposed by the stereo loudspeakers, and the input signal at the loudspeakers, respectively. Letting $\mathbf{y} = X$ (the spatialized signal), and introducing a cancelling matrix $\mathbf{C}$,

$$\mathbf{y} = \mathbf{HC}X, \tag{2}$$

it can be easily shown that $C = H^{-1}$, or

$$\mathbf{C} = \frac{1}{H_{LL}H_{RR} - H_{LR}H_{RL}} \begin{bmatrix} H_{RR} & -H_{RL} \\ -H_{LR} & H_{LL} \end{bmatrix}. \tag{3}$$

In our implementation, we assumed lateral symmetry of a listener's head and the stereo loudspeaker setup. These assumptions simplify the

computation of the transaural filter in a "shuffler" configuration [3], as shown in Figure 2. The depicted gains in the figure are $2^{-0.5}$, but in practice, these values were adjusted so no clipping was heard.

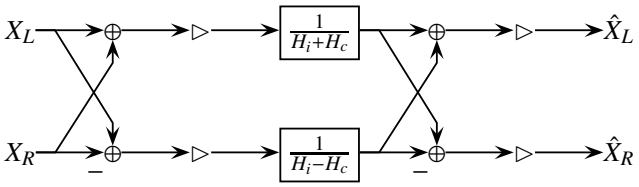


Figure 2: Shuffler implementation. Assuming perfect lateral symmetry of the listener's head and the loudspeaker stereo setup, the transaural filter can be simplified by computing the addition and subtraction of the ipsilateral and contralateral channels.

3 Implementation

The implemented transaural libraries compiled for Mac OS (v. 10.11.6), Windows 7, source code, and HRIR database are available under a GNU license from `https://bitbucket.org/julovi/trans`.

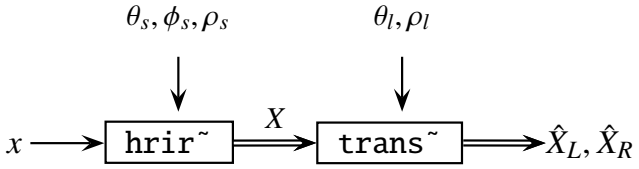


Figure 3: A monaural signal x is first spatialized by means of an HRTF filter (`hrir~` block), depending on its relative azimuth θ_s, elevation ϕ_s, and distance ρ_s. A cross-talk mitigation is carried out subsequently in the `trans~` block. This mitigation depends on the azimuth θ_l and distance ρ_l of the loudspeakers. Loudspeaker elevation is assumed to be 0° in this research.

The overall process for presenting binaural material via a pair of stereo loudspeakers is illustrated in Figure 3. A monophonic source is first filtered with an `hrir~` object according to its current location (i.e., azimuth, elevation, and distance). The resulting binaural signal is treated with `trans~`, a newly developed object, to eliminate the loudspeaker spatial effects. In our implementation, loudspeakers are assumed to be at the same height of the listener's ears. `Trans~` receives as arguments, the absolute value of the loudspeaker azimuths (measured from the front mid-sagittal plane), and their distance (a single value each, since symmetry is assumed).

3.1 Offline computations

The original HRIR compiled by Qu et al. [17] was first diffuse-field equalized [9] to make it suitable for common rooms. The database was resampled from its original $sr_o = 65.536\,\text{kHz}$ to the ubiquitous $sr = 48\,\text{kHz}$. Qu's database is not regularly sampled (i.e., the number of measurements depends on the elevation angle, and the distances were not equally spaced). A regular measurement grid (5° in azimuth, 10° in elevation, and 10 cm in distance) was built to ease the real-time implementation. Missing HRIRs linearly interpolated between the two closest ones in the desired dimension, as suggested by Wenzel and Foster [22]. These modifications were done in Matlab [14].

The new database was stored in a SQLite3 database (v. 3.8.10.1) as binary data (512 simple precision floats per channel starting from the left one).

3.2 Online computations

The binaural signals output by `hrir~` and the loudspeaker HRIRs retrieved from the database were first zero-padded according to the processing block size and then Fourier transformed. In the construction of the filtering blocks of Figure 3, limitations on the magnitude response were imposed when the sum or difference of the ipsilateral and contralateral HRTF was near zero. This step is by no means trivial, and the simplistic approach taken here (essentially, inverting the HRTFs and limiting their magnitude response) could be replaced by a more accurate method such as minimum phase inversion [10]. Finally, the inverse Fourier transformation of the filtered signals were computed and sent to the respective loudspeakers.

The real-time operations were performed with the FFTW library (v. 3.3.4) [7]. In every audio block, `trans~` verify changes in the location of the loudspeakers; in the case that no changes were found, the previous HRTFs are used instead; in the rare case when the location of the loudspeakers is changed, the routine looks for the closest measurement points in the database and perform a n-simplex interpolation [11] of the closest HRTFs, if necessary. Finally, the output of the previous block is interpolated with the output of the current block to avoid discontinuity problems in the resulting audio.

The apparent drawbacks of using Fourier transformations are outweighed by the benefits of

this alternative: it eases the HRTF inversion presented in Equation 3 and for most block sizes, the multiplication in frequency domain is faster than the convolution in time domain, effectively easing the real-time constraints.

3.3 Limitations

Transaural techniques rely on a known position (location plus orientation) of a listeners head, which can be inferred in moving picture reproduction and some gaming applications, but not so for music reproduction (especially, non-diegetic music), virtual reality applications, etc. For such cases, additional head-tracking systems are necessary. Although it is possible to track several heads simultaneously, reproducing a sound field coherent with each listener position is a very difficult task.

Additionally, although presenting audio via headphones and loudspeakers modify the intended signals with the impulse response of the transducers, transaural techniques further modify the signals by introducing the sometimes undesired impulse response of the current room.

4 Future work

We have collected anecdotal evidence indicating that `trans~` produces accurate results. However, we are interested in validating such behavior with subjective tests comparing the accuracy of `trans~` spatializations with those obtained via headphones.

To speed up the development of the transaural object, some assumptions and shortcuts were taken. For example, the symmetrical location of the loudspeakers from the mid-sagittal plane; the use of a generic HRIR database; and the restriction of the loudspeakers elevation to be 0°. We are currently working on eliminating some of these restrictions. Specifically, we distinguish at least two scenarios for which we would like to extend our development: for static loudspeaker setups (like those commonly found in vehicles), personalized HRTFs from the loudspeaker locations could be used instead of the generic HRTFs to increase the accuracy of the displayed aural image (by incidentally eliminating the loudspeaker's impulse response, as well). We also envision the possibility of using transaural techniques in dynamic setups where a swarm of loudspeakers (possibly transported by unmanned aerial vehicles) reorganize around a listener for aesthetic purposes, for example.

The aforementioned capture of personalized HRTFs from loudspeaker locations can be performed with small microphones inserted in the ear canal of the listeners and existing tools such as those described in [19] and [1]. However, the resulting impulse responses obtained with these methods need to be seasoned to be used by `trans~`.

5 Conclusions

We presented an implementation of transaural techniques based on the Pure-data architecture. With the newly implemented prototype, users can present binaural material over a pair of stereo loudspeakers adequately, provided that the loudspeakers are at the same height of the listener's ears and located symmetrically around the mid-sagittal plane. The spatialization devised here allows users to locate sound sources at arbitrary azimuths and elevations, as well as distances within the near-field. As far as the authors know, the latter feature is currently unique to our system.

6 Acknowledgements

This work was supported by JSPS Kakenhi Grant Number 16K00277. We also thank Miller Puckette and the Pure-data community for the Pure-data platform on which our work is built upon.

References

[1] Edgar J Berdahl and J O Smith. Transfer function measurement toolbox, 2007. Available from `https://ccrma.stanford.edu/realsimple/imp_meas/` (October 15, 2016).

[2] Jens Blauert and Rudolf Rabenstein. Providing surround sound with loudspeakers: A synopsis of current methods. *Archives of Acoustics*, 37(1):5–18, 2012.

[3] Duane H. Cooper and Jerald L. Bauck. Prospects for transaural recording. *J. Audio Eng*, 37(1/2):3–19, 1989.

[4] Timothy M.; Keele D. B. Bock. The effects of interaural crosstalk on stereo reproduction and minimizing interaural crosstalk in nearfield monitoring by the use of a physical

barrier: Part 1. In *Proc. 81 Audio Eng. Soc. Conv.*, 1986.

[5] David Doukhan and Anne Sédès. `CW_binaural~`: A Binaural Synthesis External for Pure Data. In *Proc. of 3rd Pure-data Int. Conv.*, 2009.

[6] Richard O Duda and William L Martens. Range dependence of the response of a spherical head model. *J. Acoust. Soc. Am*, 104(5): 3048–3058, 1998.

[7] Matteo Frigo and Steven G. Johnson. FFTW*: The Fastest Fourier Transform in the West*. Massachusetts Institute of Technology, 2013.

[8] William G. Gardner. Transaural 3-d audio. Technical report, MIT Media Laboratory, Perceptual Computing Section, 1995.

[9] William G Gardner. *3-D audio using loudspeakers*. Springer Science & Business Media, 1998.

[10] Malcom Omar Hawksford. Digital signal processing tools for loudspeaker evaluation and discrete-time crossover design. *J. Audio Eng. Soc.*, 45(1/2):37–62, January/February 1997.

[11] Peter Hemingway. n-Simplex Interpolation. Technical report, Hewlett-Packard Laboratories Bristol, 2002.

[12] Reuben Johannes and Woon-Seng Gan. 3D sound effects with transaural audio beam projection. In *Proc. 10 Western Pacific Acoustics Conf.*, Beijing, Sep. 2009.

[13] D. G. Malham and A. Myatt. 3-D sound spatialization using ambisonic techniques. *Computer Music J.*, 19(4):58–70, 1995.

[14] Mathworks. MATLAB. Software, 2016. Available from `www.mathworks.com` (October 15, 2016).

[15] Thomas Musil, Markus Noisternig, and Robert Höldrich. A library for realtime 3D binaural sound reproduction in Pure Data (pd). In *Proc. Int. Conf. on Digital Audio Effects*, 2005.

[16] Ville Pulkki. Virtual Sound Source Positioning Using Vector Base Amplitude Panning. *J. Audio Eng. Soc.*, 45(6):456–466, 1997.

[17] Tianshu Qu, Zheng Xiao, Mei Gong, Ying Huang, Xiaodong Li, and Xihong Wu. Distance-Dependent Head-Related Transfer Functions Measured With High Spatial Resolution Using a Spark Gap. IEEE *Trans. on Audio, Speech & Language Processing*, 17(6): 1124–1132, 2009.

[18] M. R. Schroeder and B. S. Atal. Computer simulation of sound transmission in rooms. In IEEE *Conv. Record*, 1963.

[19] Katja Vetter and Serafino di Rosario. ExpoChirpToolbox: a Pure Data implementation of ESS impulse response measurement. In *Pure Data Convention*, 2011.

[20] Julián Villegas. Locating virtual sound sources at arbitrary distances in real-time binaural reproduction. *Virtual Reality*, 19(3): 201–212, Oct 2015.

[21] Julián Villegas and Michael Cohen. "Gabriel": Geo-Aware Broadcasting for In-vehicle Entertainment and Larger Safety. In *Proc. 135 Audio Eng. Soc. Int. Conv.*, October 2010.

[22] Elizabeth M Wenzel and Scott H Foster. Perceptual consequences of interpolating head-related transfer functions during spatial synthesis. In *Proc. IEEE Wkshp. on Applications of Signal Processing to Audio and Acoustics*, pages 102–105, 1993.

[23] Pei Xiang, David Camargo, and Miller Puckette. Experiments on spatial gestures in binaural sound display. In *Proc. 11 Int. Conf. on Auditory Display*, 2005.

[24] Udo Zölzer, editor. *DAFX – Digital Audio Effects*. John Wiley & Sons, New York, NY, USA, 2nd edition, 2011.

PdParty: An iOS Computer Music Platform using libpd

Dan Wilcox
University of Denver
danomatika.com
danomatika@gmail.com

Abstract

This paper presents PdParty, an open-source iOS application for running Pure Data patches on Apple mobile devices using libpd. Directly inspired by Chris McCormick's DroidParty for Android and the original RjDj by Reality Jockey, PdParty takes a step further by supporting OSC (Open Sound Control), MIDI, & MiFi game controller input as well as implementing the native Pd GUI objects for a WYSIWYG patch to mobile device experience. Various scene types are supported including compatibility modes for PdDroidParty & RjDj and both patches and abstraction libraries can be managed via a built-in web server. Unlike the rise of the single-purpose audio application, PdParty is meant to provide a platform for general purpose digital signal processing via Pure Data patches.

Keywords

mobile, libpd, wearable computing

1 Introduction

Projects such as PD-Anywhere by Günter Geiger [1] have pioneered using Pure Data as a lingua franca for DSP between desktop & mobile platforms. The 2008 release of Reality Jockey's commercial RjDj application for iOS [2] built upon this work through the concept of user-friendly scenes with bundled content and included access to built-in smart phone sensor events such as multitouch, accelerometers, and GPS. RjDj "songs" are, in fact, Pure Data patches running live on the users device, allowing for interactive & generative audio beyond simple playback. This approach allowed for both distribution of artist scenes to listeners as well as a platform for computer musicians fluent with Pure Data itself. In 2010, Peter Brinkmann, with help from the RjDj team, released the first version of libpd [3], an embeddable DSP library using the core of Pure Data itself, much to the benefit of the community.

2 Background

Figure 1: *robotcowboy* @ New Media Meeting in Norkörping SE 2009

robotcowboy is the author's ongoing human-computer wearable performance project. Focusing on the embodiment of computational sound, *robotcowboy* was originally built in 2006-2007 as an MS thesis project using an industrial wearable computer[1] running GNU/Linux & Pure Data, external stereo USB sound & MIDI interfaces, and various input devices including HID gamepads. Influenced by roadworthy analog gear, chief system requirements are mobility, plug-in-play, reliability, & low cost. Compositional approaches must include live input/generation and room for failure as opposed to overly sequenced output. [4]

The original *robotcowboy* system hardware was gigged often, went on a 2 month tour of the United States in 2008, and lasted until the 2011 Pd Convention in Weimar. Around this time, Apple released the iPad 2 which featured a dual core processor and, most importantly, supported USB audio & MIDI interfaces. Seeking an option for new system hardware, the author began on and off development of an iOS application that could perform all of the tasks required for a live *robotcowboy* performance: run patches, full duplex stereo audio, MIDI, HID game controller support, & Open Sound Control communication.

[1] Xybernaut MA-V 500 Mhz P3 256 MB RAM, $350 second hand on Ebay.com in 2006

Figure 2: *robotcowboy* hardware 2007: Roland UA-25 audio interface, Xybernaut MA-V wearable computer, USB hub

3 PdParty

The development of what became PdParty began in 2011 as the "robotcowboy app" using the OpenFrameworks C++ creative coding toolkit. Over the course of a year, a set of OF add-on libraries were also developed and/or updated to provide required capbilities: ofxPd to wrap libpd [5], ofxLua as a scripting engine for visual interfaces, and ofxMidi to provide MIDI support. By the fall of 2012, the alpha prototype was working[2] but the author felt the scope of the project had far outgrown the core needs of *robotcowboy* and a slimmed-down approach was needed.

Around this time, Chris McCormick released DroidParty for Android which pioneered the concept of emulating native Pure Data GUIs in a mobile device app [6]. In early 2013, PdParty began as a native Objective-C port of DroidParty focused on usability as a general purpose platform for running Pure Data patches.

3.1 Focus

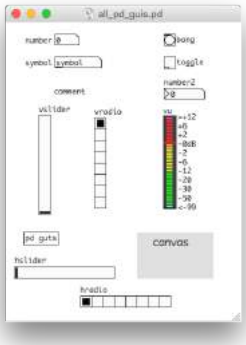


Figure 3: Demo patch in Pure Data on macOS

PdParty is focused on the easy deployment & playback of Pure Data patches on iOS devices. To that effort, great care has been taken to accurately emulate *all* aspects of the built-in GUI objects: number, symbol, comment, number2, bang, toggle, sliders, radios, vumeter, and canvas. This enables a WYSIWYG patch UI experience between desktop & mobile usage as opposed to requiring custom tools and/or programming.

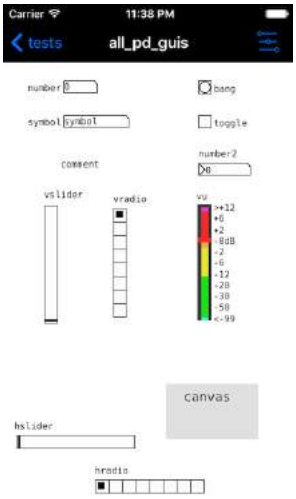


Figure 4: Demo patch in PdParty on iPhone

Like Pure Data itself, PdParty is meant as a general purpose platform for new expression and attempts to stick with Pd idioms as much as possible. Usage should be straight forward and "plug and play."

3.2 Features

The main features of PdParty include a libpd core, native GUI object emulation, scene types, onscreen controls, sensor events, game controller support, MIDI, OSC network communication, and a built-in web server. PdParty is a universal app which runs on both iPhone and iPad with appropriate interfaces and is released as open-source on GitHub.

3.2.1 libpd

PdParty is built around libpd, a wrapper library for the Pure Data vanilla DSP core with an included Objective-C AudioUnit. Patches created in vanilla will work directly in libpd, including those using the `extra` externals including `[expr~]`, `[sigmund~]`, etc. Additionally, the following externals are also included to provide access to directory information & midi files: ggee `[getdir]`, `[stripdir]`, and mrpeach `[midifile]`. As iOS does not allow dynamic library loading, all externals are compiled into the application itself.

[2] "robotcowboy app" alpha demo video: https://vimeo.com/52557228

3.2.2 GUI Emulation

PdParty emulates the Pure Data built-in GUI objects via CoreGraphics drawing routines in native Objective-C. When loading a patch or scene, the main patch is parsed, supported objects are identified by object name, and those with send/receive names are created and added to the main patch view. When interacting with the patch, control messages are intercepted using each object's send and/or receive names via libpd. GUI objects without send/receive names are ignored. Screen orientation is interpreted based on the aspect ratio of the patch canvas itself and emulated GUI object placement is scaled to approximate the original patch on desktop.

Patching a UI for PdParty follows the Model-View-Controller design pattern with GUI objects acting as both view & controller elements which communicate with the core logic of the patch via sends/receives. This approach was adapted from DroidParty whose custom GUI objects are also emulated: display, knob, loadsave, menubang, numberbox, ribbon, taplist, touch, and wordbutton.

3.2.3 Scene Types

Beyond plain patches, PdParty supports running "scenes" which are folders with a specific layout that are treated as a single entity for encapsulation and have certain attributes. RjDj scene folders end with ".rj", contain a _main.pd patch, and an optional thumbnail, background image, and Info.plist metadata file. DroidParty scene folders contain a droidparty_main.pd patch and optional an background image and .ttf font file. Native PdParty scene folders contain a _main.pd patch and an optional thumbnail and info.json metadata file.

The scene type specifies supported attributes such as required sensors and preferred samplerate. RjDj scenes are locked to portrait on iPhone, touch events are normalized to 0-320, additional sensors are accessed via rj sensor abstractions, and a 22050 Hz samplerate is used. DroidParty scenes are locked to landscape, do not require touch or accelrerometer events, and additional sensors are accessed via the `[droidsystem]` object. PdParty scenes infer orientation from patch aspect ratio, normalize touch events to 0-1, and support all sensor types.

3.2.4 Onscreen Controls

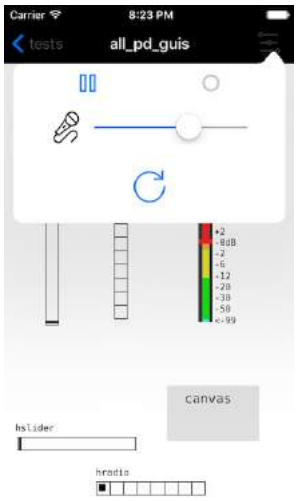


Figure 5: Onscreen control popover on iPhone

Inspired by the original RjDj app, onscreen controls appear either on the scene view itself for RjDj scenes or via a popover view controller for plain patches and all other scene types. Controls are DSP play/pause, record, microphone input level, scene restart, and an optional button to open a console view for debugging. As with RjDj, patches should use the rjlib `[soundinput]`/`[soundoutput]` abstraction wrappers for `[adc]`/`[dac]` which are required to enable the microphone input level and live recording controls.

3.2.5 Sensor and Control Events

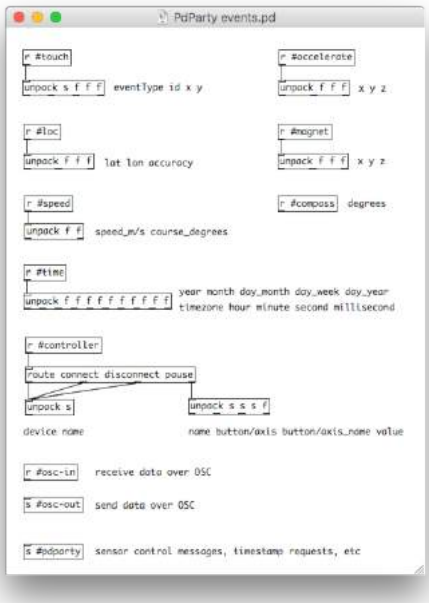


Figure 6: PdParty event receivers

PdParty provides access to the touch screen, accelerometer, gryoscope, magnetometer, GPS antenna, and built-in compass on an iOS device via events to special receive names starting with a '#'. The `#touch`, `#accelerate`, and `#loc` events match those used by RjDj. Scene types that require specific sensors and events will enable them by default, RjDj scenes for example always receive `#touch` and `#accelerometer` events.

Since some sensors use additional resources when enabled, they must be turned on by the patch or scene that uses them by sending a control message to the `#pdparty` send name, ie:

```
#pdparty loc 1 ; enable gps loc events
#pdparty loc accuracy 10m ; accuracy
```

Additionally, PdParty provides access to timestamp generation sent to the `#timestamp` receiver, manual record cueing, and opening a local or online URL through messages sent to #pdparty.

Furthermore, `[key]` events work with an external bluetooth or USB keyboard[^key]. `[keyup]` and `[keyname]`, however, are not supported as there is currently no official way to intercept raw key events in iOS.

3.2.6 Game Controllers

Compatible iOS MiFi game controllers can be read in PdParty and are hot-pluggable. Controller events are sent to the `#controller` receive name and iOS supports up to 4 simultaneous controllers.

3.2.7 MIDI

MIDI is supported on iOS via either USB or over Wifi using Network MIDI with a computer running macOS. PdParty detects hot-plugged devices and automatically enables sending and receiving MIDI messages. All Pure Data MIDI objects are supported (`[notein]`, `[ctlout]`, etc).

3.2.8 OSC

PdParty sends and receives OSC (Open Sound Control) messages internally between the libpd instance and a built-in OSC server using liblo, an open-source C library for the OSC protocol. Messages can be received in Pd patches using the `#osc-in` receive name and sent to the `#osc-out` send name. Message parsing and formatting are provided through the `[oscparse]` and `[oscformat]` objects which are part of Pure Data vanilla versions 0.46+.

If enabled, all PdParty events can be streamed over OSC including Pd prints, eg. `#touch` events are sent to the `/pdparty/touch` address. This allows development and debugging of patches and scenes in desktop Pure Data using events streamed from PdParty running on a mobile device.

3.2.9 Browser

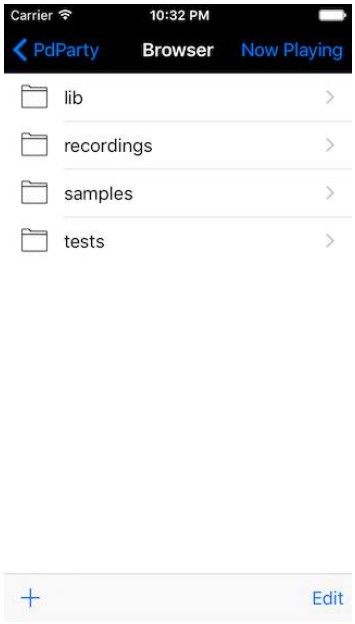


Figure 7: PdParty browser on iPhone

Patches and scenes are managed in PdParty via a standard "drill-down" file browser which displays files and folders. Additionally, common editing controls are supported including delete, rename, move, and copy. Selecting a patch or scene will open it in a patch view. Also .pd patch files and .zip archives are affiliated with PdParty and can be copied and opened from other applications including Mail and DropBox.

3.2.10 Web Server

Patches and scenes can be loaded onto PdParty either using iTunes File Sharing through iTunes or over a local network using the built-in WebDAV web server. The server allows for full access to the PdParty Documents folder and can be enabled from the start screen which also displays the server IP and .local address.

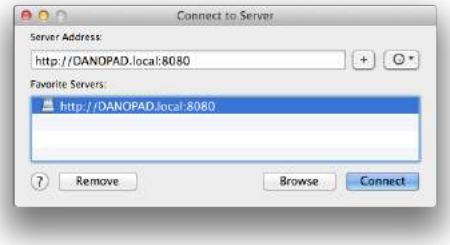


Figure 8: Connecting to the PdParty WebDAV server in macOS Finder

A connection can be made using a file transfer program such as FileZilla or Cyberduck as well as from operating system file managers that support WebDAV including macOS Finder and Gnome Nautilus. Working in this manner allows for live, direct access to the patch files on the device from desktop Pure Data.

3.2.11 Lib Folder

PdParty ships with a default "lib" folder which contains PdParty's required abstractions, allowing for the bundled patches to be upgraded or replaced by the user. Similarly, any subfolders are automatically added to the libpd search path when opening a patch or scene, so it can be used as a central place for abstraction libraries. If the folder or any required abstractions are missing, PdParty falls back to its own internal copy.

3.2.12 App Settings

Important PdParty application settings control behavior, OSC event forwarding, audio latency, and default folder copying. PdParty can be allowed to run in the background and configured to disable the lock screen from appearing. OSC event types can be individually enabled for automatic event forwarding if the OSC server is running. The audio latency can be chosen automatically or set manually by buffer size (64-2048). Last, the contents of the libs, samples, and tests folders can be recursively overwritten by their default files.

3.3 User Guide & Composer Pack

Usage and patching information is detailed in the online PdParty User Guide which includes notes on all event send & receive formats as well as OSC addresses. A composer pack is also available via .zip file which includes notes, scene type templates, and OSC communication patches for desktop Pure Data.

3.4 Development Timeline

The first major alpha version, 0.3.0, was finished in March 2013 and featured native emulation of all built-in Pure Data GUI objects, a patch browser, MIDI support, OSC communication, and a web server for on-device patch management. In September 2013, 0.4.0 alpha was released to testers using the TestFlight framework and included a settings interface, on-screen controls, RjDj & DroidParty scene type support, and DroidParty custom UI emulation. At the time of writing, PdParty is at version 0.5.6-beta and includes a user guide, composer pack, UI icons, demo scenes, full iOS sensor event support, game controller support, and various bug fixes & improvements.

4 robotcowboy with PdParty

With PdParty, the author now has a stable low latency mobile/wearable platform with a touchscreen, accelerometer, WiFi networking, and USB MIDI/audio. Here is a belt-based wearable setup using an iPhone, Camera Connection Kit, powered USB hub, Roland Edirol UA-25 USB audio interface, and a Behringer direct box (the latter two are built in the case on the left):

Figure 9: Prototype *robotcowboy* belt with iPhone 2016

5 Future

Although PdParty is largely feature complete, new developments are always possible since "software is never finished."

5.1 Multiple Patch Views

Currently, PdParty's patch view only displays GUI objects loaded from the main scene patch. It may be useful to be able to display multiple GUI patches in either separate tabs or from within a temporary modal patch view. One possible use case could be to open a mixer view from a running patch.

5.2 Link

Ableton Link[3] is a cross-device protocol for tempo synchronization which was released as open-source for iOS and desktop computers in 2016. Although not a new concept, Ableton's clout as a music software company will most probably push Link's adoption on many music environments and platforms in the future. PdParty could integrate Peter Brinkmann's abl_link~ external to send and receive Link messages within patches. [7]

5.3 AudioBus

AudioBus[4] is an iOS library for routing audio between multiple apps running on the same device. As PdParty is a general purpose Pure Data DSP platform, it is a natural fit as a node within the overall ecosystem of iOS audio applications. AudioBus support was not one of PdParty's main requirements but could be added in the future.

5.4 libpdparty

DroidParty can both run scenes as well as be used for creating new Android applications. The core of PdParty (libpd, GUI emulation, event handling, etc) could be similarly spun off as a separate Objective-C library for use when creating custom PdParty applications. Notedly, libpd-based MobMuPlat (Mobile Music Platform) by Daniel Iglesia uses the PdParty GUI emulation classes on iOS [8].

5.5 Patch Editing

PdParty is focused on the running of Pure Data patches and scenes but does not have the capability to edit them. If libpd adds a standard API for communication with the Pure Data GUI, an editing UI could be added for mobile patch creation. Some degree of interaction design and research, however, will be required for adapting a desktop UI idiom to mobile devices.

6 Conclusion

After years of on and off development, the author is happy to finally release PdParty to the Pure Data community. It is hoped PdParty will be a useful tool for musicians seeking alternate performance paradigms on an embedded device they already own. With a growing libpd-based mobile ecosystem, the future of computer music is in your pocket.

Links

https://github.com/danomatika/PdParty

7. Acknowledgments

The development of ofxPd was supported by the CMU Frank-Ratchye Studio for Creative Inquiry and director Golan Levin. PdParty is directly influenced by Reality Jockey's RjDj and Chris McCormick's DroidParty. Frank Barknecht & Joe White provided insight into the RjDJ scene format. Thanks to Miller Puckette and the Pure Data community for Pd itself.

References

[1] G. Geiger, "PDa: Real Time Signal Processing and Sound Generation on Handheld Devices," in *International computer music conference*, 2003.

[2] P. Kirn, Create Digital Music, Oct-2008 [Online]. Available: http://cdm.link/2008/10/rjdj-responsive-interactive-music-on-iphone-now-available-free-3

[3] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding Pure Data with libpd," in *Pure Data Convention Weimar 2011*, 2011.

[4] D. Wilcox, "robotcowboy: A One Man Band Musical Cyborg," Master's thesis, Chalmers University of Technology, 2007.

[5] D. Wilcox, "ofxPd: a Pure Data addon for OpenFrameworks using libpd." [Online]. Available: https://github.com/danomatika/ofxPd

[6] C. McCormick, "DroidParty." [Online]. Available: http://www.droidparty.net

[7] "Ableton Link integration for Pd." [Online]. Available: https://github.com/libpd/pd-for-ios/tree/master/abl_link

[8] D. Iglesia, "MobMuPlat." [Online]. Available: http://danieliglesia.com/mobmuplat

[3] https://www.ableton.com/en/link
[4] https://audiob.us

libpd: Past, Present, and Future of Embedding Pure Data

Peter Brinkmann
Google Inc
peter.brinkmann@gmail.com

Dan Wilcox
University of Denver
danomatika@gmail.com

Tal Kirshboim
tal.kirshboim@gmail.com

Richard Eakin
rich.eakin@gmail.com

Ryan Alexander
Okaynokay
scloopy@gmail.com

Abstract

libpd is a C library that turns Pure Data into an embeddable audio synthesis library, with wrappers for a range of languages and support for mobile platforms such as iOS and Android. We present an update on developments that have happened since PdCon11, including a discussion of libpd in creative environments and the emergence of apps that provide platforms for deploying Pd patches to mobile devices with little to no code. Looking forward, we discuss some challenges for future development, including questions of real-time safety and support for concurrency, as well as support for multiple instances. Finally, we examine the relationship of libpd to Pd itself.

Keywords

library, dsp, embedding Pure Data

1 Introduction

libpd began in July 2010, as a learning project whose only goal was to run Pd patches on Android devices. The core library emerged as a byproduct of a refactoring of the original solution. Within a few months of its publication, the team at RjDj decided to replace their in-house solution for embedding Pure Data with libpd. This led to the creation of Pd for iOS, whose open-source release, along with the creation of the Pd Anywhere forum at Create Digital Noise, marked the beginning of an exciting period of growth and creativity as libpd was adopted by hundreds of researchers and developers. Six years later, it is still going strong.

We presented the design of libpd and its language wrappers for Java and iOS at PdCon11 [1]. Much like Pd itself, the core library has remained rather stable, but thanks to the efforts of several dedicated contributors, the original, haphazard packaging has been replaced by proper release engineering. The integration of libpd into creative coding environments such as OpenFrameworks and Cinder as well as the emergence of general-purpose apps based on libpd have further supported the growth and adoption of libpd.

The purpose of this paper is to give an account of important technical developments around libpd that have occurred since PdCon11 and to identify problems that we will need to solve if libpd is to remain relevant, such as concurrency and support for multiple instances.

2 The Current State

Simply put, libpd *is* Pure Data vanilla without the graphical user interface or audio backends. It is not a fork of Pd. The Pure Data source code is included in the libpd git repository as a git submodule and directly tracks stable upstream releases. Further, any libpd-related bug fixes for the Pure Data core are submitted upstream to ensure the libpd codebase does not diverge. So far, this practice has propagated a number of fixes for 64bit related issues on iOS back into Pure Data with the help of Miller Puckette and community members.

2.1 Building

Building libpd is handled by a Makefile that compiles the core libpd C library as well as C++, Java, and C# wrappers. Make options also control the conditional compilation of libpd utilities and the bundled externals in the Pure Data `extra` directory. The Python wrapper is built using a traditional `setup.py` script that first builds the C library using the main Makefile. Obj-C support for iOS and macOS is provided via the included libpd Xcode project which is also utilized by the sample projects in the Pd for iOS repository.

2.2 Workflow and Release Cycle

The core libpd C API has been stable for a while. Most current work revolves around creation and maintenance of the various language wrappers. As of 2014, libpd uses semantic versioning and keeps a changelog between versions. Major changes are undertaken and tested in git branches, simple updates are committed to the master branch, which can be considered generally stable, and stable release versions are git tagged. Tagging allows for support by library management systems including CocoaPods via libpd's included libpd.podspec file and NuGet.

2.3 Documentation

High-level documentation comes in the form of the libpd website at http://libpd.cc, the libpd book "Making Musical Apps" [2], and sample projects such as those included with the Pd for iOS and Pd for Android repositories on GitHub. Low-level development documentation is available on the libpd GitHub wiki[1] which details build settings, per-language APIs, and working with libpd in various software environments. Supplementary information is largely community-based on the Pure Data mailing lists, forums, and via third parties through text and video tutorials.

2.4 C++ Wrapper

The development of the libpd C++ wrapper began in early 2011 as part of ofxPd, a libpd add-on library for the OpenFrameworks creative coding toolkit, and was integrated into libpd itself a year later. Influenced by the Java wrapper, the C++ interface is built around a class called PdBase which wraps the main libpd C API, base receiver classes for messages and MIDI data, and a set of convenience classes for the variable list data type and unique patch identifier. All classes are declared within the "pd" namespace. Additionally, PdBase provides a C++ style stream interface for message building and sending:

```
pd << StartMessage() << 1.23
   << "sent from a streamed list"
   << FinishList("fromCPP");
```

A minimal libpd C++ use case involves the following: implementing a subclass of PdReceiver and/or PdMidiReceiver for message handling; creating and initializing an instance of PdBase; creating a receiver subclass instance and setting PdBase to use it; subscribing to any required message sources; opening a patch; starting audio; and calling one of the PdBase process functions in an audio callback. The C++ wrapper does not contain an audio interface but can easily be dropped into a project using a cross-platform audio library such as PortAudio or RtAudio.

Optionally, PdBase can be built with a C++11 std::mutex to provide thread safety if the `LIBPD_USE_STD_MUTEX` compiler macro is defined, and it can be configured to pass messages through a lock-free ringbuffer in the C layer. Also, the C++ wrapper is forward-designed as PdBase currently utilizes a protected PdContext singleton class as a placeholder for possible upcoming multi-instance support.

2.5 ofxPd

ofxPd is a C++ add-on library for the OpenFrameworks creative coding toolkit that runs an instance of Pure Data within an OpenFrameworks application [3]. Audio, messages, and MIDI events can be passed to and from Pure Data patches with the OF run loop and the library is thread safe. The main ofxPd class is a convenience wrapper around the libpd C++ wrapper that adds support for multiple message and MIDI receivers, routing specific message sources to specific receiver class instances, and small changes such as updating MIDI channel ranges from 0-15 to 1-16 to match the numbers used in Pure Data itself.

Figure 1: NodeBeat scene on iPad

OpenFrameworks projects using ofxPd include NodeBeat, NinjaJamm, and Scrapple. Nodebeat is a node-based visual music app for iOS, Android, macOS, and Windows by Seth Sandler, Justin

[1] https://github.com/libpd/libpd/wiki

Windle, and Laurence Muller [4]. Record label Ninja Tune's NinjaJamm is a looper for Android and iOS with live effects and high quality, artist curated sample packs [5]. Scrapple is an interactive installation by media artist Golan Levin that uses computer vision to interpret the shapes of physical objects on a projected surface as a live visual score [6]. The audio engine for Scrapple was rebuilt in 2012 using ofxPd and features a bank of 128 FM synthesis voices.

2.6 Cinder and libpd

libpd can be used with the Cinder C++ creative coding library with an add-on called Cinder-PureDataNode [7]. Cinder has its own cross-platform, modular audio processing graph, in which the entire Pure Data context can be utilized as a node. The ability to process audio from both Cinder and Pd can be very powerful, allowing for the use of OpenGL for rich visuals, platform-specific file decoders and encoders, and low level DSP tools such as sample rate conversion.

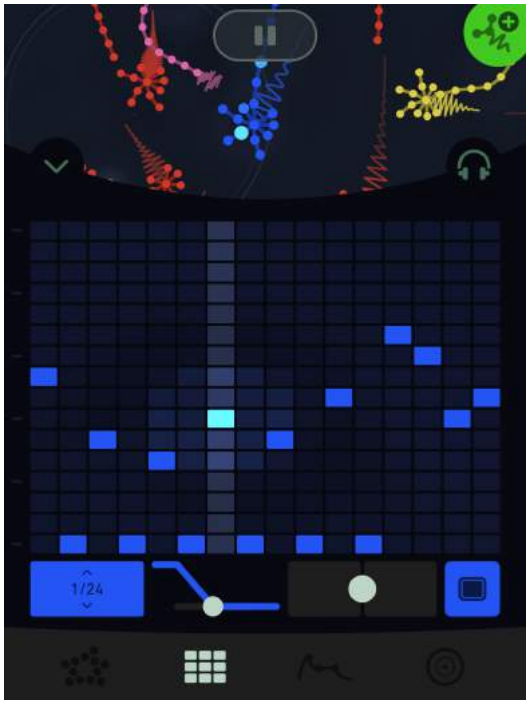


Figure 2: Seaquence on iPad

Cinder-PureDataNode was used extensively in the upcoming iOS app Seaquence [8], which features a stereo spacialized 10 voice, 50 note polyphonic subtractive synthesis engine built entirely with Pd. The ability to test and work on the synth outside the iOS app was a major win for Seaquence developers in terms of iteration time. Also, the ability to collaborate with knowledgeable members of the synthesis community was invaluable.

2.7 C# and NuGet

LibPDBinding, the C# wrapper for libpd was written by Tebjan Halm in 2012 for Windows .NET and acts as a thin layer around the C API with support for thread synchronization. In 2016, Thomas Mayer updated the build system to support 64bit on Windows and the MONO platform on Linux and macOS. Additionally, builds of libpd for C# are now hosted on NuGet [9], an open source package management system for Microsoft development, which greatly simplifies adding libpd to a C# project in Visual Studio or a supported MONO environment.

2.8 Pd for Android

Developing for the Android platform has changed substantially since the initial libpd release. Android Studio was released in late 2014 together with a new gradle build tool plugin for Android. These have replaced the Eclipse IDE and the ant build tool that were used for building Android applications until that time. The Pd for Android project was migrated to use this new toolchain in early 2015.

Another change that followed the migration to Android Studio was releasing Pd for Android as a maven artifact on JCenter [10]. Using JCenter to resolve Android dependencies is a common practice in Android development, and it significantly reduced the amount of effort involved in integrating Pd for Android in new Android apps. Prior to the release on JCenter, developers had to clone the project repository and its submodules and build the Java as well as the C code in the project, the latter using the Android NDK. With the JCenter dependency, all that is required in order to include Pd for Android in an app is to add a single line in the application's build file. When developers wish to use Pd externals which are not part of libpd in an Android app, they will still have to resort to the original and more complex way of integrating Pd for Android.

As part of the process of releasing Pd for Android on Jcenter, the btmidi[2] submodule that allows sending and receiving MIDI on Android devices was released on JCenter as well. While Android already includes MIDI support from version

[2]https://github.com/nettoyeurny/btmidi

6.0 Marshmallow on, the btmidi module allows devices with older Android versions to use MIDI.

Aside from the project-related structural changes mentioned here, Pd for Android proves itself to be very stable, even on newer Android versions. Problems that are reported with the library are often not specific to Pd for Android, but rather general issues with audio performance.

2.9 CocoaPods

As of 2013, libpd includes a podspec file for CocoaPods[3], a library dependency manager for iOS and macOS projects. This allows for the easy addition of libpd to an Xcode project without the need to manually include files and/or fiddle with build settings, thereby increasing deployment and accessibility. CocoaPods support was contributed to libpd by members of the CocoaPods community.

2.10 General Purpose Mobile Apps

Inspired by Reality Jockey's original RjDj app, a number of general purpose mobile applications for hosting libpd-based projects have been developed: DroidParty, MobMuPlat, and PdParty. Each of these apps can run Pure Data patches, access touch and sensor events, and support encapsulated scene directories including metadata, abstraction libraries, and sound files.

DroidParty by Chris McCormick pioneered the concept of recreating GUI objects (bang, tog- gle, hslider, etc) on Android [11]; Daniel Iglesia's MobMuPlat (Mobile Music Platform) for iOS and Android provides a custom GUI designer and supports OSC, MIDI, and networking features [12]; iOS app PdParty by Dan Wilcox faithfully replicates all Pure Data GUI objects, incorporates a web server and patch browser, and supports similar communication features to MobMuPlat [13].

3 Future Development

While the stability of libpd is generally a sign of success, we will need to make sure it remains relevant and maintainable as technology and engineering practices progress.

3.1 Visual Studio Support

Currently, the libpd C sources do not compile in Microsoft Visual Studio, mostly due to historical issues related to supported C versions that require changes to the Pure Data core. Consequently, builds of the library on Windows thus far have relied on MinGW. Newer versions of Visual Studio include C99, and it will be worthwhile to revisit any needed source updates to support building for platforms including Windows and Xbox, as well as the Windows app store.

3.2 Autotools

libpd currently relies on a single Makefile for most of its build and install options. As support for various languages, operating systems, and environments has grown, so has the complexity of the Makefile. A more sustainable long term approach for building, maintenance, and distribution would be to convert the project to use autoconf and automake which provide standard mechanisms for OS and compiler specific configuration, compile time options, and distribution tarball creation.

3.3 Alternatives to Locking

When processing audio in real time, it is generally advisable to avoid locks because of the risk of priority inversions. We do not agree with the frequently heard admonition to never use locks; since none of the popular operating systems are hard real-time systems, glitches are always possible, and so the question of whether to use locks is just another trade-off.

When working with a code base that predates ubiquitous multiprocessing, it is often hard or impossible to avoid locks. Pd falls into this category. libpd addresses this problem by deliberately discarding Pd's built-in global mutex lock, leaving the core library lock-free. That does not mean, however, that it is thread safe. Rather, the question of how to synchronize libpd calls is left to higher-level components, usually wrappers for languages like Java and C++.

The distribution of libpd includes a lock-free queue that is meant to reduce the need for locks. In practice, the trade-off we chose for most use cases is to pass outgoing messages (i.e. messages from Pd to the ambient app) through the lock-free queue, while passing incoming messages individually when holding a lock. The main reason for this is Pd's symbol table, which sits at the center of most of Pd's operations and is not thread safe.

While this approach has held up well in most cases, there is the possibility of audio dropouts

[3]https://cocoapods.org

when sending too many messages to Pd. Solving this in a general, performant way will probably require replacing Pd's symbol table with an implementation that is both thread and real-time safe.

3.4 Multiple Instances

The Pure Data core was originally written to be used within a single application and utilizes little data encapsulation, which makes it difficult to use within multi-context, multi-threaded environments such as reusable audio plugins. Preliminary work on multiple in- stance support by Miller Puckette in 2014[4] allows for context creation and switching with the `pdinstance_new()` and `pd_setinstance()` functions using a `t_pdinstance` type:

```
t_pdinstance *pd1 = pdinstance_new();
t_pdinstance *pd2 = pdinstance_new();
...
pd_setinstance(pd1); // 1st instance
libpd_openfile(argv[1], argv[2]);
pd_setinstance(pd2); // 2nd instance
libpd_openfile(argv[1], argv[2]);
```

This approach has been built upon by Kjetil Matheussen's Radium graphical DAW [14] and Pierre Guillot's Camomile VST plugin [15], both of which utilize custom wrappers of the Pure Data core with improved support for multiple instances and multithreading. Hopefully, this community work can be reincorporated into libpd and Pure Data.

3.5 libpd as a Core for Alternate GUIs

libpd's API is a standardized interface for message communication and sample I/O with the Pure Data DSP core. By abstracting much of the detail, omitting unused backends, and providing dummy interfaces, the changes made to the Pure Data source code allow for a simple embeddable C library suitable for use within other languages and environments beyond the desktop TCL/TK GUI.

Looking forward, this approach can be used to standardize communication between the core and the GUI, allowing for the creation of alternate editing GUIs using libpd. Jonathan Wilkes's GUI messaging specification for Purr Data provides a preliminary roadmap[5]. If this work can be integrated into the main Pure Data source and elements of the GUI code such as Undo/Redo be factored out into overridable C files with standard APIs, the various flavors of Pure Data (vanilla, Pd-L2ork, Purr Data) could all share the same upstream development source while maintaining their own customizations and improvements. Additionally, a future version of libpd with GUI messaging would facilitate the creation of even more esoteric editing environments such as an ncurses-based ASCII GUI or patch creation on mobile devices.

4 Conclusion

The first six years of libpd have seen the adoption and steady growth of a Pure Data-based ecosystem. Support for numerous programming languages and development environments has been added, leading to a new variety of audio visual apps, digital instruments, and sonic experiences. At this point, libpd development is largely stable but, moving forward, there are a number of improvements that can be made for libpd to work within audio plugins and as a core for new GUIs. The future is bright for Pure Data patches as the basis for continued computer music expression.

Acknowledgments

We thank everyone who contributed to libpd over the past six years, as well as everyone who made it a success by building amazing apps with it. It is an honor and a privilege to be a part of this community.

Peter Kirn maintains the libpd website: libpd.cc.

The development of ofxPd was supported by the CMU Frank-Ratchye Studio for Creative Inquiry and director Golan Levin.

References

[1] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding Pure Data with libpd," in *Pure Data Convention Weimar 2011*, 2011.

[2] P. Brinkmann, *Making Musical Apps: Real-time Audio Synthesis on Android and iOS*. O'Reilly Media, 2012.

[3] D. Wilcox, "ofxPd: a Pure Data addon for OpenFrameworks using libpd." [Online]. Avail-

[4] https://lists.puredata.info/pipermail/pd-dev/2014-05/019832.html

[5] https://git.purrdata.net/jwilkes/purr-data#gui-messaging-specification

able: https://github.com/danomatika/ofxPd

[4] "NodeBeat." [Online]. Available: http://nodebeat.com

[5] "NinjaJamm." [Online]. Available: http://www.ninjajamm.com

[6] G. Levin, "Scrapple." [Online]. Available: http://flong.com/projects/scrapple

[7] R. Eakin and R. Alexander, "Cinder-PureDataNode." [Online]. Available: https://github.com/notlion/Cinder-PureDataNode

[8] R. Alexander and G. Dunne, "Seaquence." [Online]. Available: http://okaynokay.xyz/presskit/seaquence

[9] "LibPDBinding on NuGet." [Online]. Available: https://www.nuget.org/packages/LibPdBinding

[10] "pd-for-android on JCenter." [Online]. Available: https://bintray.com/pd-for-android/maven/pd-for-android

[11] C. McCormick, "DroidParty." [Online]. Available: http://www.droidparty.net

[12] D. Iglesia, "MobMuPlat." [Online]. Available: http://danieliglesia.com/mobmuplat

[13] D. Wilcox, "PdParty." [Online]. Available: https://github.com/danomatika/PdParty

[14] K. Matheussen, Available: http://users.notam02.no/~kjetism/radium/

[15] P. Guillot, "Camomile." [Online]. Available: https://github.com/pierreguillot/Camomile/wiki

TOWARDS FULLY AUTOMATED `object`-VERIFICATION

IOhannes m zmölnig
Institute of Electronic Music and Acoustics
University of Music and Dramatic Arts
Graz, Austria
zmoelnig@iem.at

ABSTRACT

While other sound synthesis systems have made the switch to 64-bit floating point values for signal processing a while ago, Pure Data (Pd) seems to be stuck with single precision numbers. An initial port to 64-bit precision has been presented in 2011, but 5 years later little progress has been made. One of the alleged showstoppers for switching to 64-bit precision is the plethora of available 3rd party externals, many of which might start malfunctioning in subtle ways. In this work we try to solve this problem by means of automatic `object` verification.

Keywords. unit tests, fuzz testing, automatic testing, double precision Pd, Pure data

1. INTRODUCTION

In 2006, Csound5 was released and featured a `Csound64` flavor, which uses `double` precision signal processing throughout [6]. It took until 2011, when Vetter created an initial 64-bit precision build of Pd [7], but 5 years later we are practically still using only single precision builds, with the `double` precision amendments that made it into the Pd-core silently bit rotting away.

There are a number of reasons why we see so little adoption. One of the alleged showstoppers for switching to 64-bit precision is the plethora of available 3rd party externals, many of which might appear to work fine (e.g. it is possible to compile these external with setting the Pd-data types to `double` precision). But when those objects are actually used, they might turn out to be fundamentally broken [8].

The two obvious ways to deal with this problem are to either optimistically introduce the change and hope that people will report any problems so they can promptly be fixed (at the expense that the users will have a broken system until all those issues are fixed), or to proactively inspect the source code of all those externals for potential problems with the proposed change. Neither of the two ways is especially appealing.

In software industries the „standard" way to automatically verify whether a piece of code is working as expected (e.g. after doing a major refactoring of the source code), is by running it through a number of *unit tests*, that check whether a „program" produces the expected result when fed with known input data.

Unit tests for Pd are nothing new: a few libraries come with their own ad-hoc unit test suites (e.g. `zexy`, including a bash-script for running about 100 unit tests since late 2005 [11] (although about 70% of the tests check for trivial parameters like existence), or `PuREST JSON`, featuring a python-based system since 2014 with 20 functional tests).

There have also been several proposals for formalized unit test frameworks, starting with `PureUnity` [1] up to `testtools` [3].

Incidentally, the latter was designed (among other things) to provide a means to automatically verify whether an object is fit for `double` precision Pd.

One big disadvantage of unit testing is that somebody has to write the actual unit tests, which is time consuming and unrewarding, resulting in poor adoption: of more than 160 libraries found on the Pure Data SVN on Sourceforge, only six (6!) have a test suite than can be run automatically.

However, poor adaption of unit testing can also be observed in the software industry in general [4]. This has led to the development of automatic testing systems, like `American Fuzzy Lop` [9], which automatically creates a corpus of test data that will test as many lines of code as possible. These systems are normally used to find security flaws and crasher bugs, by feeding unexpected input data to a program. Notably, such unsupervised testing is not targeted at verifying whether a program produces the *correct* output given a known input. However, a synthesized test corpus can be used to verify, whether the software being tested behaves the same in different *environments*.

The underlying assumption is, that if we have a test corpus that - when fed into a software unit running in an environment A - triggers execution of all possible code paths in that unit, then the unit's behavior is fully determined. If we then use this very same test corpus on that unit in a different environment B and the unit's response is the „same" (within limits), then the object can be considered to work the **same** in both environments. Finally, if the unit is already known to work **correctly** in environment A, we can conclude that it is then also working **correctly**

in environment *B*.

For Pd, those *units* could be single objects, while *environments* could be different supported platforms (Linux, OSX, W32,…), but they could also be different flavors of Pd, such as the traditional single precision Pd or the `double` precision Pd. Furthermore, such *environments* could be the different forks of the Pd-engine, such as Pd-devel, Desire Data, Pd-extended, Pd-l2ork or Purr Data. Given that many Pd objects have been heavily tested over the last few years on Pd-vanilla (and Pd-extended), we assume that they are working **correctly** (though not necessarily bug-free) in this environment.

2. GOAL

This project's goal is the automatic creation of test corpora for any Pd object. We are targeting primarily at externals (compiled objects, written in C), since the *environments* we want to investigate (single resp. `double` precision Pd) are mainly differing on the low, binary level. If all objects that interact with the low-level data representation are working correctly, we don't expect any environment-specific problems with objects that only operate on the high level.

However, the methods we are using can equally be applied to abstractions.

3. DESIGN

For the task at hand we created the `PeDAnT` framework (an acronym for *Pure Data Automatic Testing* framework).

The purpose of this testing framework is to generate a corpus of test data, and to provide a way to run each test case through the object: e.g. by sending messages or signals to its `inlet`s, or letting time pass. It also needs to provide a way to record all output of the object (whatever message or signal was sent to its `outlet`s at which time), so test runs in different environments can be compared.

3.1. Corpus Synthesis

The most complicated part in this endeavor is certainly the creation of a test corpus, that covers all aspects of our object. For this task, we propose to utilize fuzz testing tools. In *fuzz testing*, software is tested by feeding it generated random data as input, and watching the software's reaction. It is mostly used to detect vulnerabilities, such as memory leaks or even crashes that depend on the input data. However, naïvely generating random data is unlikely to find many of these „interesting" edge cases in finite time. A promising technique to solve this problem is to use program-flow analysis of the tested binary as a fitness indicator for a genetic algorithm, which is used to (per)mutate a small starting corpus.

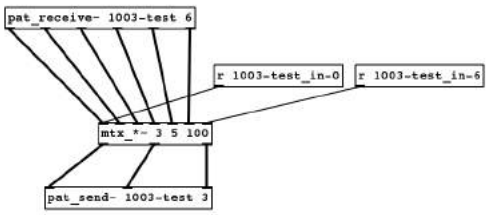


Figure 1. Automatically connecting all iolets of `mtx_*~ 3 5 100` under test

One popular (and fast) tool that utilizes this approach is Michał Zalewski's *American Fuzzy Lop* (`afl-fuzz`), which uses code instrumentation to generate a test corpus that triggers the execution of a maximum number of source code lines in the tested program. `afl-fuzz` also features a „fork server" to rapidly start numerous test runs (e.g. we found that it could start up Pd up to 1000 times per second - with Pd opening a test patch, loading the test data, feeding data into the test object and the quitting).

However, despite the astonishing performance of the test runs and a corpus generation that is sped up by code coverage analysis, fuzz testing is still a very laborious undertaking: testing a single program can easily take days!

Since `afl-fuzz` will keep generating data forever, a termination condition is needed. Using `gcov` (the test coverage program that comes with the *GNU Compiler Collection*), it is easy to create a (human readable) report on how many lines of code are already covered by the test corpus. Once a maximum of code is covered, the test corpus can be considered complete. In practice this maximum will often be below 100%, mostly because some branches in the code are simply unreachable (e.g. a debugging function that is compiled but not actually used).

3.2. `PeDAnT` Framework

A small support library has been written that instantiates an object, fully connects its iolets (Fig.1) and then feeds test data to the object and records any response it gets.

This library is used in an abstraction `pedant-run.pd` that executes a single test run and then quits Pd. It is to be used in `-batch` mode for speedy execution (this is important as the tests also include (randomized) time, so running a test in *real time* might take years).

The rest consists of a set of Python scripts, mainly wrapping a correct invocation of `afl-fuzz` (and `gcov`), so it uses the Pd-patch to instantiate the tested object and feed it the synthesized test corpus.

There are also scripts to feed a (completed) corpus to a tested object and record it's responses, and to

compare the responses of two test runs (e.g. within different environments).

Most scripts take a configuration file (Listing 1) that can be shared among the various tasks.

```
[pedant]
subject=limiter~
[pd]
#binary=/usr/bin/puredata
args=-path /home/pedant/src/zexy/src/
[afl-fuzz]
tests=workbench/limiter~/seedtest/
#dict=workbench/limiter~/dict/
out=workbench/limiter~
[afl-cov]
binary=/home/pedant/bin/afl-cov
codedir=/home/pedant/src/zexy/src/
args=--coverage-at-exit
```

Listing 1: `PeDAnT` shared configuration

3.3. Test Input Data

So far we found three different types of input data that determine the behavior of an object. Any concrete object may not necessarily require all three types to be fully determined.

3.3.1. Signals:

Signals consist of a series of numbers (signal blocks). For every DSP-tick all `inlet~`s need to be filled) with full signal blocks (each of equal length).

3.3.2. Messages:

Messages consist only of serializable data (numbers and symbols), that are send immediately („now") to the specified `inlet` of the object.

3.3.3. Time:

The *time* data type allows to schedule *messages* (and *signals*) by advancing the logical time („now"). Some objects exhibit their specific behavior only over time (e.g. `delay`), and this data type provides a way to make (logical) time pass.

3.4. File Format

`afl-fuzz` is capable of synthesizing test data of almost arbitrary complexity [10]. However, the synthesizing algorithm mostly alters existing test cases by randomly flipping bits, or by cutting up test cases and splicing them together anew.

A file format with the following characteristics should greatly improve the speed of test data generation:

- expressive: randomly flipping bits should create (very) *different* test-cases.
- robust: randomly splicing test-data should create *valid* test-cases again.
- precise: it should be easy to express any number and any string.
- extensible: it should be easy to add new data types.

These requirements led to the design of a binary format as described in Listing 2.

To allow for an arbitrary number of data points (e.g. messages or signals) each holding arbitrary data (including typical `EOL` characters like `CRLF` or `NUL`), each data point is SLIP-encoded [5]. The data representation of the actual values are modeled closely after the values found in current (32-bit) Pd-vanilla: all *numbers* (including samples) are 32-bit floating point values. Timestamps are stored as `double` precision (64-bit) floating point values, and are actually time increments (in [ms]): in order to guarantee a monotonic clock those values must therefore always be positive.

Signals are stored as an arbitrary length list of numbers. The actually used signal samples are then calculated by repeating the sequence as needed, until all `inlet~`s are filled. In order to fully determine the signal input to an object, an additional parameter *signalblocksize* is therefore required. To keep the signal block size in a reasonable range, the actual value stored is an exponent in the range 0..16 (which gives possible block sizes 1..65536).

Data generated by an object during a test run can be stored for later comparison by prefixing each data frame with a `*`. When reading a data file *as input*, any lines prefixed with `*` are simply ignored. (In general, unknown prefixes are ignored).

4. WORKFLOW

4.1. Generating Input Data

The central task of the framework is the synthesis of a meaningful test corpus. This is done with the help of the fuzzer tool `afl-fuzz`.

4.1.1. Instrumenting the Tested Objects

`afl-fuzz` works best if it can instrument the binary under test, by adding *marker points* that allow a quick evaluation whether a given code path has been executed or not. Instrumenting the binary also allows `afl-fuzz` to apply its *fork server* during the tests, which greatly speeds up testing as it doesn't need to spawn a new process for each test run.

```
; data points are SLIP-encoded before being stored in the file
datapoint = direction (message / timeincrement / signal / signalblocksize)
direction = ["*"] ; the optional '*' prefix indicates result data (not used as input)
message = %s"m" iolet *(string / float)
timeincrement = %s"t" float64 ; positive delta time in [ms]
signal = %s"s" *float32
signalblocksize = %s"b" bsize ; the actual bufsize is 2^bsize
string = %s"s" *CHAR *1%x00
float = %s"f" float32
float32 = 4OCTET ; 4byte float (big-endian)
float64 = 8OCTET ; 8byte float (big-endian)
iolet = OCTET ; unsigned char
bsize = OCTET ; unsigned char
```

Listing 2: ABNF of the test file format

Instrumenting is done by by building the object with a special compiler `afl-gcc` [1] , an enhanced variant of `gcc`. Any reasonable build system allows to override the compiler via the `CC` (or `CXX`, in case of C++-projects) variable.

4.1.2. Seeding the Test Data

In order to quickly create „interesting" test data, the genetic algorithm needs to be seeded with some initial „meaningful" test cases. For this purpose we pursued the following strategies, all of which can at least be semi-automated:

1. Harvesting existing (help-)patches for messages that make sense to the object: For this we simply extracted *any* message (as found in `message boxes`) from patches that use the object, and created one test case per found message. We relied on the splicing capabilities of the fuzzer to create meaningful message sequences from these solitary messages.

2. Keyword dictionaries: `afl-fuzz` can use dictionaries of keywords to build additional tests. By extracting strings/symbols from the source code and binary files of the object, we built a small set of keywords that are hopefully useful when testing the object.

3. Existing tests: Many Pd objects have similar interfaces. Therefore the test corpus of one object can serve as a good starting point for another object. However, this can quickly give a too large seed corpus, which will slow down the genetic algorithm, especially if many tests do **not** fit the object's interface (e.g. a message for which the object has no method). `afl` provides a tool for *corpus minimization*, which will remove superfluous tests from a corpus that do not trigger any new execution paths.

[1] or `afl-clang` if that is preferred, though we haven't used that in our tests so far.

4.1.3. Synthesizing Data

Once a suitable seed corpus has been created, the fuzzer can start generating a full test corpus.

```
pedant-fuzz \
  --config mytest.conf \
  --fuzz-out <inputcorpus>
```

The fuzzer will then start to generate thousands of test cases, and run them through the tested object. Whenever it detects that new execution paths have been triggered by a given test, it will use that test as the new seed for generating even more tests.

Targeting the fuzzer at the `zexy`'s `limiter~`, it took about 10 days (with 4 synchronized fuzzer instances) to generate a total of 11643 test cases (see Fig.2).

4.1.4. Minimizing Input Data

The generated corpus is usually rather large and covers a lot of redundant tests. Many items in the corpus will contain garbage data (due to the random nature of the test generation), which can easily be eliminated by parsing the test file, discarding invalid data points and saving the remaining ones. Any duplicates created during this process can be safely eliminated.

Also, `afl` provides a corpus minimization tool, `afl-cmin`, which can further reduce the corpus to a minimum set of files that covers all code paths.

```
pedant-cmin \
  --config mytest.conf \
  <inputcorpus> <reducedcorpus>
```

This was able to reduce the 11643 test cases for `limiter~` to 696. (while at the same time reducing the memory footprint from 90MB to 3.4MB).

4.1.5. Creating Tests

Once a sufficiently large corpus of test data has been generated, one more test run is needed to generate

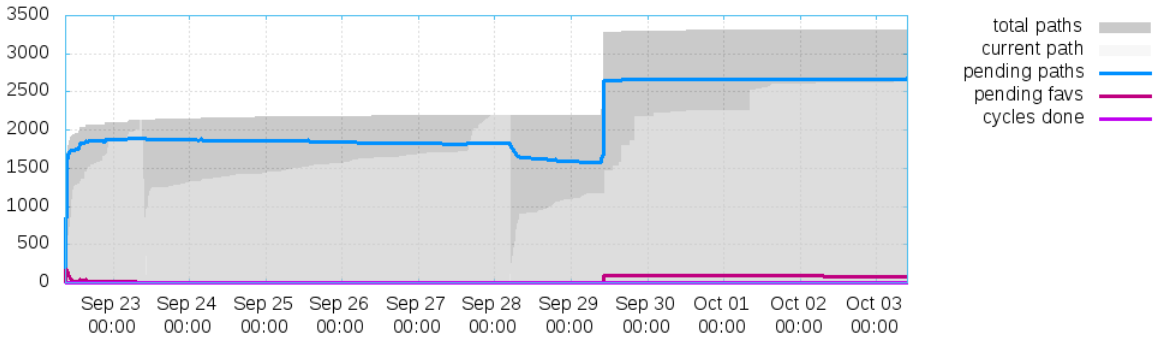


Figure 2. code paths discovered by one of four parallel fuzzer instances over the period of 10 days, targeting zexy's `limiter~`. After one week, a fundamentally new code path was discovered, creating a new set of test-cases to be explored.

a set of *comparable* tests. This time, the output of the tested `object` is stored alongside its input, thus recording the entire interaction of the object (modulo side effects). Output data has the very same format and available types as input data (see 3.3) (though this time the `outlet` number is stored along with a message), but is marked with an asterisk (`*`) prefix within the file.

```
pedant-run \
  --config mytest.conf \
  <inputcorpus> <resultsA>
```

4.2. Running Tests

The test corpus (consisting of input and output data) can then be used to verify the object in a different environment, simply by using it as input (discarding any *output* data present in the corpus) and storing the results in a different directory:

```
pedant-run \
  --config mytest.conf \
  <resultsA> <resultsB>
```

4.3. Comparing Results

Verifying the `object` is done by comparing `<resultsA>` and `<resultsB>`, data item by data item: if any of the *output* data differs the test has failed (if the *input* data differs, the test run itself was faulty). Symbols and integer values in the test data, are compared for absolute equality, but floating point data needs special consideration [2] and is compared with both an *absolute* and a *relative* (in relation to the reference (input) data) ϵ. One can also specify a value for ∞ (any data that has a bigger absolute value, will be treated as inf). This is useful, since an intermediate inf value can poison the final result, and when doing `double` precision calculations this poisoning is less likely to happen.

```
pedant-compare \
  --absolute-tolerance 3e-5 \
  --relative-tolerance 1e-5 \
  --infinity 1e30 \
  <resultsA> <resultsB>
```

5. DISCUSSION

5.1. Interpreting Failed Tests

Even when fuzzily comparing test results (as discussed in 4.3), verifying a `double` precision build of `limiter~` against a `single` precision reference still yields a few false positives. These happen with very large input values that make the `single` precision binary unstable. There are a number of false positives like this, where a failing test is actually desired behavior, as it hints at the advantages of the tested environment: e.g. that it is possible to do things with a `double` precision environment that are impossible to do in a `single` precision environment.

It turns out that interpreting the result of a failed test (the differences between the output data sets generated by multiple test runs) is surprisingly hard, as it lacks meaningful context data. Is the failed test a false positive? If it hints at a real problem, where to start looking for in the source code?

5.2. Performance

So far, we have only conducted experimental test runs on a very small scale. The hardware used to conduct the tests was an Intel®Core™i7-870 CPU, running at 2.93GHz, where 4 cores where used to run four fuzzer instances in parallel.

The corpus synthesis for a slightly complex object like `limiter~` already took quite a lot of time - in the order of days to weeks, raising the question on how useful the proposed technique would be in testing the hundreds of objects commonly used.

On the other hand, simple objects like `sgn~` take considerably less time: starting from a single generic

seed test case, it took less than five minutes to create 200 test cases (before minimization) that cover 97.2% of the entire code.

5.3. Automation

As of now, the automation of the framework consists of a few scripts that still require some human intervention. Most notably the corpus generation requires supervision to determine the halting condition.

5.4. Limitations of Testable Objects

Currently the only kind of objects that can be tested without user intervention are *functional* objects. That is, only objects that generate their output solely from their input data (and their internal state, as defined by creation arguments). This excludes any object whose acquire input data from external sources (such as files, user-input, hardware,…). Also, the tests depend on the object sending any output through its `outlet`s and currently do not check for any side-effects (such as created files, or controlled hardware).

Objects that depend on other objects (e.g. `r foo` or `tabread bar`) *can* be tested, but require a (human) agent to create a functional *abstraction* that wraps the interaction of the object with it's peers and presents an interface where only serializable data is sent via `inlet`/`outlet`s.

5.5. The Bad News: Code Coverage as a Terminating Condition

Unfortunately, it was easy to proof that the primary assumption of the framework (that by executing a maximum of code-paths we can reliably trigger all bugs that produce different results on single resp. `double` precision builds) can be falsified.

E.g. running aggressive minimization on the corpus for `limiter~`, reduced the number of test cases to 260, which cover **98.1%** of the `limiter~.c` source code (a local maximum: the remaining 7 lines of code were never executed because they were generally unreachable or because the creation arguments to `limiter~` were kept fixed). Running those tests in `single` resp. `double` precision environments revealed 11 test cases that would produce different results, thus failing the tests.

Removing these interesting test cases from the set (thus producing a non-failing test corpus) and re-evaluating the covered code-paths resulted in a coverage of **98.1%**, which is exactly the same coverage as when those test cases were included.

Therefore, maximizing the code-coverage is not a sufficient strategy to catch all errors introduced by `double` precision builds (since the generated test corpus could have well not-included the interesting test cases, and still would have been considered „complete"). This also suggests that aggressive corpus minimization based on coverage analysis, might eventually drop interesting test cases.

5.6. The Good News

While the initial goal of verifying objects across environments cannot be achieved reliably, the proposed methodology is still able to generate interesting test corpora that can help improve the implementation of the tested object.

For instance, running the fuzzer on a couple of `zexy` objects quickly revealed a systematic programming error that would trigger a crash of Pd. Consequently, these have now been fixed.

Despite not having a reliable metric for the corpus completeness regarding different precisions, we found that in practice the large number of generated tests makes it still *probable* that at least one of the tests will expose issues in the problem domain.

Also, for different problem domains (e.g. comparing different implementations of the Pd-engine, like Pd-vanilla vs Pd-l2ork) the initial assumption might still hold true.

6. FUTURE WORKS

As shown in 5.5, code coverage is not a sufficient metric to determine whether a test corpus will expose certain problems. More research is required to find an indicator that can reliably guide the test synthesis algorithm to produce test cases that expose problems with 64-bit precision builds.

6.1. Harvesting for Seed Data

The current approach to generate an initial seed corpus for the fuzzer uses human-guided brute force. It includes all messages the happen to occur in some help patch, regardless of whether these messages are actually sent to the object under test. Better data might be harvested by actually running such a patch, triggering all user-interaction (e.g. by randomly clicking any message-boxes, toggles and similar) and recording all messages as they are received by the tested object (e.g. by replacing the test-subject with a dummy object with identical interfaces).

Also the compiled object itself could be harvested more efficiently than just extracting all the strings (which returns a lot of garbage). A good start would be to check which message selectors are accepted on which inlet (and what argument they take).

6.2. Testing Creation Arguments

So far the creation arguments of an object are fixed, and test synthesis does not vary them during the test runs. Since some objects have parameters that can

only be set via creation arguments (e.g. the number of iolets), this possibly leaves a substantial part of the code untested.

We are currently investigating the most stable way to allow the fuzzer to also control the way an object is instantiated. One choice is to interpret a message at the very beginning of the test data as creation arguments.

6.3. Intelligent Corpus Minimization

The current strategy for corpus minimization (code coverage analysis) has been shown to be problematic. A simple strategy to remove garbage from the test data is partly applied, by reading a data file, discarding any unknown messages and writing it back to disk. This can further be improved, by discarding any message that is not understood by the tested object (The `no method for 'bong'` case).

This will also create more meaningful seed corpora for new objects from existing test sets.

6.4. Interpretation Help

In most cases, one has to understand *why* a test fails in order to fix the code. However, the current representation of failed test results (basically the differing values are displayed in a human readable format, value by value) is not very helpful. A representation with diff-highlighting that also provides a bit of context depending on the data type, will most likely make the interpretation of the test results easier. For signal data, a graphical representation might make even more sense.

6.5. A Central Repository for Test Corpora

On the long run it would be nice to have a central repository of test corpora for „all" objects. This would make it possible to quickly validate new environments. Having a largish set of tests available, can also speed up the synthesis of a new corpus for a yet-untested object.

7. CONCLUSIONS

We have presented the `afl`-based fuzz-test framework `PeDAnT` for semi-automated testing of Pd-objects. The main strength of the framework is that little human interaction is required for the chores of *writing tests*: `PeDAnT` is able to synthesize test corpora with a maximum code coverage of the tested object on its own, trying to find as many different issues as possible.

While our original objective - fully verifying an object's functionality for `double` precision builds of Pd - has not (yet) been met, the framework can give useful hints when problems are likely. Even without comparing test results from different environments, the test framework has already proven useful for finding a number of crasher bugs.

`https://git.iem.at/pd/pedant`

REFERENCES

[1] M. Bouchard. PureUnity [online]. 2005. URL: `https://git.puredata.info/cgit/svn2git/libraries/pureunity.git/` [accessed 2016-10-05].

[2] B. Dawson. Comparing Floating Point Numbers, 2012 Edition [online]. 2012. URL: `https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/` [accessed 2016-10-05].

[3] F. J. Kraan and K. Vetter. testtools [online]. 2011. URL: `https://puredata.info/downloads/testtools` [accessed 2016-10-05].

[4] X. Qu and B. Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 117–126. IEEE, 2011.

[5] J. L. Romkey. A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP. RFC 1055, RFC Editor, June 1988. URL: `http://www.rfc-editor.org/rfc/rfc1055.txt`.

[6] B. Vercoe, J. ffitch, J. Piché, P. Nix, R. Boulanger, R. Ekman, D. Boothe, K. Conder, S. Yi, M. Gogins, and A. Cabrera. *The canonical Csound reference manual*. MIT Media Lab, 2007.

[7] K. Vetter. Double precision Pd [online]. 2011. URL: `http://www.katjaas.nl/doubleprecision/doubleprecision.html` [accessed 2016-10-05].

[8] J. Wilkes. What's the deal with [utime] object? [online]. 2016. URL: `https://lists.puredata.info/pipermail/pd-list/2016-02/113637.html` [accessed 2016-10-05].

[9] M. Zalewski. American Fuzz Lop [online]. 2013. URL: `http://lcamtuf.coredump.cx/afl/` [accessed 2016-10-05].

[10] M. Zalewski. Finding Bugs in SQLite, the Easy Way [online]. 2015. URL: `http://lcamtuf.blogspot.co.at/2015/04/finding-bugs-in-sqlite-easy-way.html` [accessed 2016-10-05].

[11] IO. m. zmölnig. zexy [online]. 2005. URL: `https://git.iem.at/pd/zexy` [accessed 2016-10-05].