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.

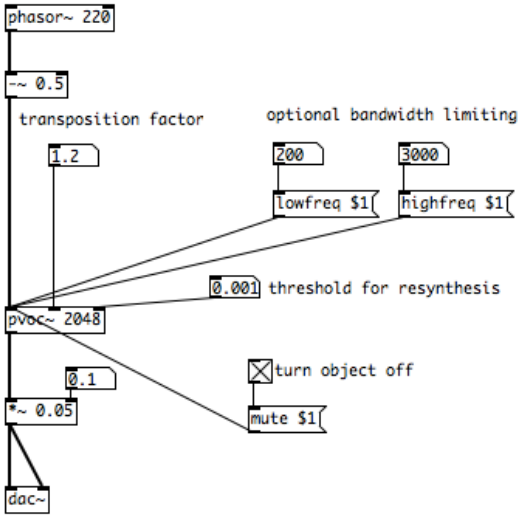

Figure 3 Using the FFTease object *pvoc~*

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.

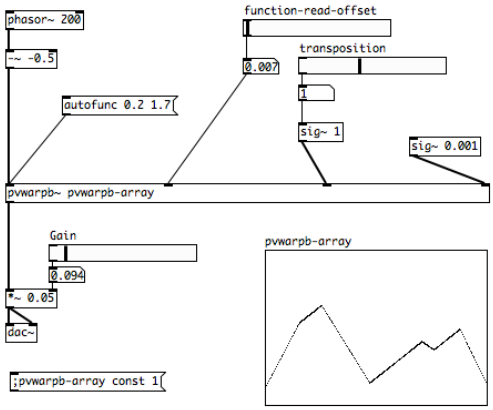

Figure 4 Using the FFTease object *pvwarpb~*

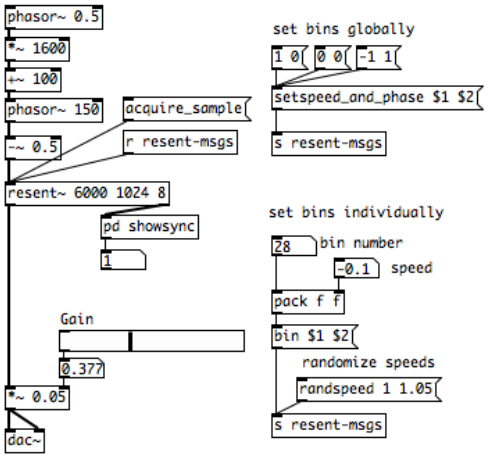

Figure 5 Using the FFTease object r*esent~*

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.

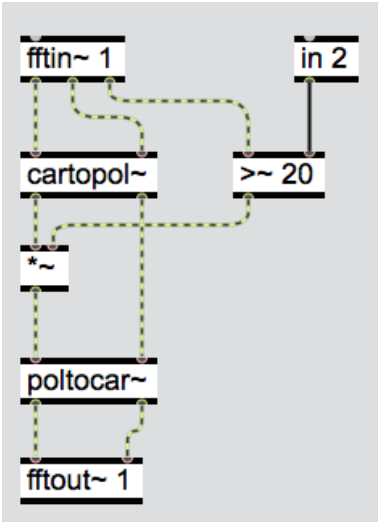

Figure 6 A pfft~ abstraction

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.

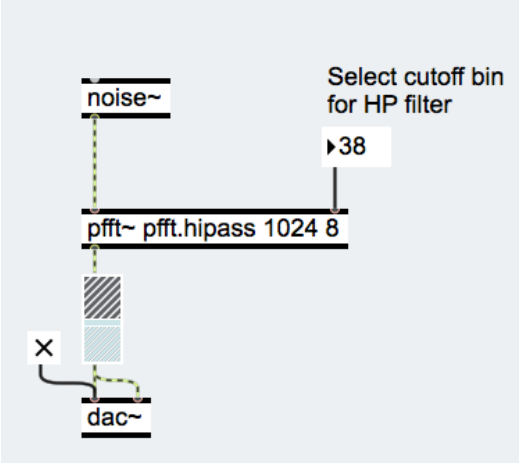

Figure 7 Using a pfft~ abstraction in Max

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.

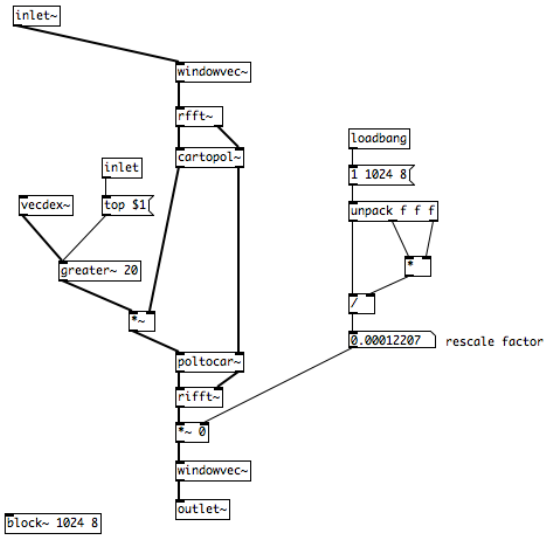

Figure 8 A Pd sub-patch employing a pseudo-pfft~ structure

The enclosing patch for the FFT-based hipass filter described above is shown in Figure 9.

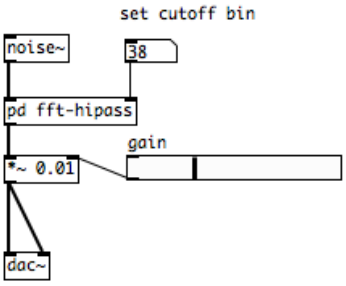

Figure 9 The parent Pd patch for the sub-patch shown in Figure 8

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.

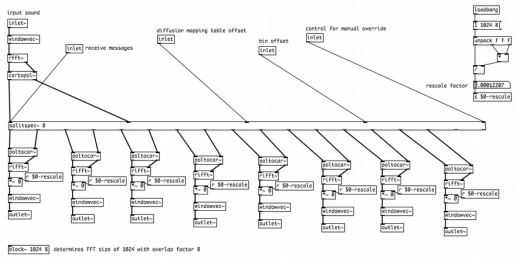

Figure 10 a sub-patch using splitspec~

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.

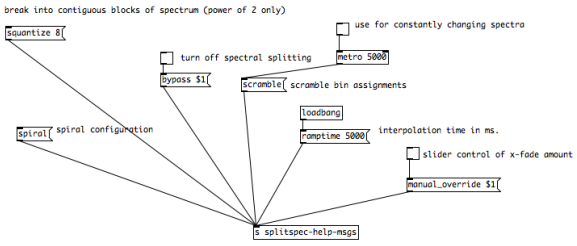

Figure 11 Message for splitspec~

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.

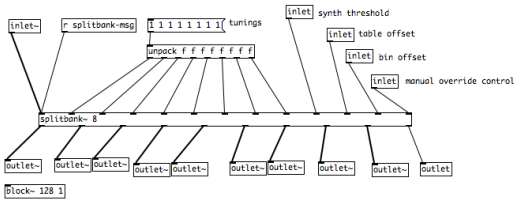

Figure 12 Splitbank~ used in a sub-patch

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.