

# FrameLib: Audio DSP using Frames of Arbitrary Length and Timing

Alex Harker

Creative Coding Lab: CeReNeM, University of Huddersfield  
ajharker@gmail.com

## ABSTRACT

*Block-based processing of audio streams commonly employed in realtime audio environments such as Max, Pd and SuperCollider is ill-suited to digital signal processing which functions on discrete chunks, or frames, of audio. Such environments currently lack comprehensive support for complex multi-rate processing. Consequently, well-documented frame-based processing techniques requiring sophisticated multi-rate processing DSP graphs are currently under-exploited in the creative coding community.*

*FrameLib provides an extensible open-source library for realtime frame-based audio processing, and is currently available as a set of Max externals and a C++ codebase. It enables rapid prototyping and creation of DSP networks involving dynamically sized frames processed at arbitrary rates. Unlike prior solutions, FrameLib provides novel systems for scheduling and memory management, reducing complexity for the user. Details of these novel approaches are explored in detail, and an overview of FrameLib in its current state is given.*

## 1. INTRODUCTION

### 1.1 Motivations

Realtime audio environments such as *Max*<sup>1</sup>, *Pd*<sup>2</sup>, and *SuperCollider*<sup>3</sup> are built on a model of audio processing in which a continuous *stream* of samples is grouped into small consecutive blocks of equal size for the purposes of calculation.<sup>4</sup> Whilst this offers an appropriate model to represent and process a continuous analog signal, such a model is less well suited to digital signal processing which functions on discrete chunks, or *frames*, of audio in which either the position of a sample within the frame is meaningful (e.g. spectral representations where position in a frame represents frequency), or in which the frame might better considered as a whole (e.g. in the context of granular synthesis).

There are a range of frame-based techniques that are well-documented in the DSP literature but are not accessible natively in realtime audio environments. These include use

<sup>1</sup> <https://cycling74.com/products/max/>.

<sup>2</sup> <https://puredada.info>.

<sup>3</sup> <http://supercollider.github.io>.

<sup>4</sup> This increases efficiency by reducing the number of function calls, when chaining operations, and the fact that some operations can be calculated only once per block.

of the wavelet transform [1] or other frequency transforms such as the MDCT [2], multitaper spectral analysis [3], audio descriptor analysis [4], finely timed synthesis and processing such as FOF and FOG [5], or waveset manipulation [6]. Spectral processing more generally also remains cumbersome in these environments, despite solutions such as *pfft~* for *Max* or *PV UGens* in *SuperCollider*. Advanced multi-rate processing, based on DSP graphs which operate on frames of different rates and sizes simultaneously, is not currently natively supported by any of the common environments. Low-level coding in compiled languages allows realtime exploration of such techniques, but where rapid prototyping and creative concerns are of paramount importance this approach brings significant barriers in terms of access and the requirement to deal with low-level technical details.

### 1.2 Scope

*FrameLib* seeks to address the gap between the wealth of frame-based techniques outlined in the DSP literature, and their adoption by the creative coding community, by providing an extensible open-source library specifically aimed at realtime frame-based audio processing. The library is current available as a set of *Max* externals, but can also be bound to other realtime audio environments, or accessed as a C++ codebase.<sup>5</sup>

*FrameLib* aims to minimise low-level concerns for users whilst offering a high level of flexibility in terms of scheduling and frame size:

- *FrameLib* supports frame-based processing using dynamically allocated *arbitrary length* frames which may vary freely in length over time or between DSP network nodes.
- *FrameLib* supports *arbitrarily scheduling* of frames with sub-sample accuracy across a network and can combine synchronous and asynchronously updating inputs to a single node of a DSP graph.
- Input and output to and from a standard *block-based* audio stream is possible, and output frames can *overlap* in which case they may be accumulated to the output using an overlap-add model.

### 1.3 Prior Work

Beyond the limited multi-rate processing native support discussed above, there are two notable related prior so-

<sup>5</sup> See <https://github.com/AlexHarker/FrameLib> for the current version of the project. Distribution is under the BSD 3-clause license.

lutions: IRCAM's *Gabor* [7] and *PIPO*<sup>6</sup> (Plugin Interface for Processing Objects) projects. The first of these builds on the *FTM* objects for *Max* [8] to enable multi-rate frame-based realtime audio processing. The second is a lightweight C++ library for designing frame-based DSP graphs. A notable difference here is that, unlike previous approaches, *FrameLib* tackles issues of scheduling and realtime memory management directly, simplifying usage, and allowing the full functionality of the library to be ported to different audio environments. *Gabor*'s scheduling is built on top of the *Max* scheduler and processing takes place in the *Max* message thread, rather than in an audio thread. Memory management is based on the *FTM* approach of in-place operation, leaving the user to consider low-level concerns, which become complex in advanced multi-rate scenarios. *PIPO* provides no scheduling or memory management facilities and is limited to only unary operations on data. The novel approaches to scheduling and memory management are discussed in detail below.

## 1.4 Aims and Design Considerations

*FrameLib* aims to provide a *fast, lightweight* and *extensible* library specifically for realtime frame-based audio processing. The underlying C++ codebase is *environment agnostic*, and provision for a specific environment is made via an appropriate set of bindings to an environment's API. The library targets two overlapping audiences, the first of whom will use it only within a specific realtime audio environment (referred to as *users*), and the second of whom will also extend its capabilities by adding *objects* to a core set of tools, provide bindings to a new environment or use the C++ codebase directly (referred to as *developers*).

### 1.4.1 Design Considerations Concerning Users

Users are *not required to deal with developer-level concerns*. More specifically, this means that all mechanisms of dynamically scheduling processing and memory management are handled by the library. The basic data representation, designed to be *flexible*, is simply a vector of sample values. These might represent a time series, frequency series, or other collection of data. Throughout the library the preference, where possible, is to assume an abstract data model, rather than to assume that the content of a frame is tied to a particular representation.

### 1.4.2 Design Considerations Concerning Developers

The aim is to enable *ease of extensibility*. Wherever possible, tasks are delegated to library code, including the scheduling of processing and details of memory management<sup>7</sup>, so that developers need provide only a minimal amount of code to create simple objects. Within reason, *safety* is also enforced by this delegation (although speed is preferred where this would also be a concern). Effort has been made to minimise requirements for handling environment bindings, or direct C++ usage.

<sup>6</sup><http://ismm.ircam.fr/pipo/>.

<sup>7</sup> Developers are responsible for allocating appropriate output frame sizes, and handling temporary memory allocations in full, but not freeing the memory used for output frames.

## 2. KEY IMPLEMENTATION ISSUES

The core implementation issues within *FrameLib* arise from the requirement for dynamic scheduling of processing and the related dynamic memory requirements for storing inputs/outputs to each object. In a synchronous audio processing model, in which all objects in a DSP graph are executed in each call, the processing order for items on the graph can be serialised, and thus memory can be pre-allocated and utilised in a consistent manner. This is the method currently employed in *Pd*, for instance. Here, before any processing occurs, the network is "walked" (or "compiled") to determine the order of dependencies, and then memory is allocated to account for the temporary storage required. Usually, the same blocks memory will be reused where possible in order to reduce memory requirements, and also to increase cache efficiency which can have a noticeable effect on speed.

Given the constraints of arbitrary frame length and arbitrary scheduling, such an approach is not possible, as neither the size of memory requirements, nor the order of dependencies can be known ahead of time.

## 2.1 Scheduling and Timing

### 2.1.1 Scheduling Model

The key concern for scheduling processing within a *FrameLib* network is the progression of time, which must be kept in line with the host environment, and resolved correctly between connected objects. The current time of objects and data in the network must be updated in a controlled manner, with correct synchronisation applied to local parts of the network (for instance, two inputs arriving at a node in the graph at the same time must be available simultaneously when the object calculates). Objects should also calculate or update only once in the case that two or more inputs arrive simultaneously. To assist this process, each frame in the network is accompanied by two timestamps: one that indicates when the frame was generated, and a second one that indicates when it is valid until (this second timestamp allows time to update in the network without the generation of new frames in the case that the time of the next frame is not known beforehand).

In *FrameLib*, objects which trigger frames directly are called *schedulers* and these provide a C++ *schedule* method which calculates the time interval until the point at which they next need to update the timestamps of their output. This method also returns information about whether or not the scheduler has generated an output when it updates, or if it is due to do so on the next update (the latter assists memory management as outlined below). The ability to update time without triggering an output allows schedulers to be dependent on future input (including dependency on audio from the host environment), in which case they can continue to update without triggering new output until they receive relevant input.

All other objects are known as *processors* and these objects are scheduled automatically by the library. Here the approach is based on a combination of dependency counting, examination of the timestamp of each input and the ability of each input to trigger an output (which is allowed to vary between an object's inputs, or over time for a single input, to allow for flexibility of operation). Each object

is updated only if all of its dependencies have progressed suitably. An object's dependencies include all its inputs, any objects that rely on its output and, where relevant, the progression of time in the host environment's audio thread. The object's dependency count is used to keep track of any objects which have not yet progressed, and is decremented when the dependencies update. When this count drops to zero, the object is added to a queue which is serviced until empty to update the objects in the network.

When an object is called to update, library code examines whether a new frame has been triggered and, if so, calls the object's C++ *process* method to perform its calculation. Regardless of whether the object has produced new output or not, the next scheduling stage is to determine the time at which the object will next update. This is calculated based on the timestamps of the object's inputs, and is equal to the earliest time at which any one input will no longer be valid.

After the next update time has been determined, the object's dependency count is increased by the number of its inputs due to become invalid at that time (the inputs it is waiting for to next update), as well as the number of objects dependent on the output (to ensure it does not update before them). Finally, the dependency count of objects that rely on the output of this object, or that provide input to the object which is no longer valid (and are thus due to update) are decremented so that other objects in the network are able to progress. This also occurs automatically for scheduler-type objects. At this point any objects with a dependency count of zero will be added to the queue, and so the process repeats until all objects have updated.

Synchronisation to the host environment is maintained by all schedulers and any processors which have audio input connections from the host environment.<sup>8</sup> These objects have an additional dependency on the audio processing routine called by the host. This routine is used to update an object-specific timestamp calculated by accumulating the size of each audio block (thus reflecting the progression of time in the host environment). The dependency is updated (decremented with a matching increment) once per block, with a virtual C++ method provided for performing audio processing before this synchronisation point. This ensures that any audio input has arrived from the host before frames are processed. Schedulers are required to update until their output timestamp matches or exceeds their object timestamp, ensuring that the *FrameLib* network advances correctly in relation to the host environment. Objects with audio outputs also maintain a timestamp in relation to the host, used for correctly synchronising audio into the output stream, but do not have a dependency on the host audio thread, as frames will arrive before audio processing takes place. Updating of timestamps is handled in all cases by the library code, so developers have no possibility of altering timestamps to be out-of-order.

For clarity, Listing 1 shows a pseudocode summary of the key scheduling-related class methods for processor-type objects. This omits memory management handling, and the calculation of output frame times.

```
// Object Variables

bool handlesAudio;

int dependencyCount;
int numOutputDependencies;

TimeFormat hostTime;
TimeFormat validTime;

// Called when a dependency updates

void notify()
{
    if (--dependencyCount == 0)
        addToProcessQueueAndService();
}

// Called by the host's audio thread

void blockProcess(int blockSize)
{
    hostTime += blockSize;

    if (object is scheduler or has audio input)
        notify();

    processBlockAudio();
}

// Called by the processing queue

void dependenciesReady()
{
    // Check for calculation

    bool calculate = false;

    for (each input)
        if (input->triggers && input->time == now)
            calculate = true;

    // Calculate next update time

    TimeFormat updateTime = infinite;

    for (each input)
        if (input->validTime < updateTime)
            updateTime = input->validTime;

    // Process if required

    if (calculate == true)
        process();

    // Reset the audio dependency if necessary

    if (handlesAudio && validTime >= hostTime)
        dependencyCount++;

    dependencyCount += numOutputDependencies;

    // Notify Dependencies

    for (each input)
        if (input->validTime == now)
            input->object->notify();

    for (each outputDependency)
        outputDependency->notify();
}
```

**Listing 1.** Processor Scheduling Methods Summary

<sup>8</sup> Processors may have either audio input or output connections to the host, but not both, due to conflicting synchronisation requirements in order to ensure correctness in complex networks. For this reason, schedulers can take audio input only.

### 2.1.2 Characteristics of the Scheduling Model

It is hopefully clear that the key mechanism in *FrameLib* for scheduling is the control of time across the network, which is constrained to update only in a forward direction for each object, although the rate of progression can vary across the DSP graph. The method of scheduling via dependency counting acts as a two-way lock to prevent time from becoming ‘out-of-order’ in the network (earlier objects in the graph can progress further in time than later ones, but not vice versa), and also results in a push-pull model, in which objects may calculate either as a result of their inputs updating, or as a result of their output no longer being held by an object dependent on it. This has potential benefits for the exploitation of parallelism which are discussed below. Finally, it should also be noted that, as data for the processing queue is stored within the object structures within the network, the underlying processing graph is entirely distributed across the graph, removing the requirement for any additional realtime memory management to deal with scheduling state.

### 2.1.3 Multithreading Opportunities

Although the current version of *FrameLib* allows only single-threaded audio processing, its scheduling algorithms are designed with multi-threading in mind. As dependency counting can be carried out with atomic variables, and the resulting scheduling algorithm is threadsafe with no other alterations, it is a trivial matter to make this part of the library ready for multithreading. Furthermore, due to the push-pull nature of the scheduling model, and as only increment/decrement operations are required, the resulting algorithm is also non-blocking and wait-free.<sup>9</sup>

The difficulty lies in constructing a suitable multithreaded queuing algorithm (which would ideally also be non-blocking and wait-free in the situation where there is enough work for each thread). When it is possible to provide such a queue, *FrameLib* will be able to exploit multiple cores in two distinct situations. The first, and most obvious of these exploits natural parallelism within a DSP graph, in which case independent branches of the graph may proceed concurrently. The second, less obvious situation, is a scenario in which multiple frames are scheduled within a single block from the host. In this scenario, earlier objects in a DSP graph may begin processing later frames concurrently with objects later in the graph processing earlier ones, even when in a serial configuration, as objects are allowed to progress in time independently of the later nodes in the network. This may be thought of as akin to pipelining, although as each object holds its inputs until it has calculated (preventing it from updating) it is impossible for each object in a serial chain to update simultaneously and there will at least one object waiting between each of those that is processing. Nonetheless, both of these scenarios might result in better use of computing resources.

## 2.2 Realtime Memory Management

### 2.2.1 Fast Memory Allocation

The realtime constraint requires speed and consistency of allocation and deallocation time, as far as possible. Ideally

a realtime allocator would be used. Whereas generalised OS memory management is subject to unknown complexity and completion times, realtime memory allocators aim to allocate and deallocate using algorithms with a complexity of  $O(1)$ . Several such algorithms exist, offering fast memory management (e.g. [9] and [10]). On embedded systems these may be used as the core method of memory management. However, for general purpose computing this is not possible or practical. The only way to use exclusively a realtime allocator in the context of *FrameLib* would be to use one to manage a large fixed-size buffer preallocated from the OS. This approach has significant flaws, as the memory allocated might be insufficient, such that memory cannot be allocated as needed. Or, if the overall allocation is made unnecessarily large, the OS and host environment will be needlessly starved of memory. Thus, in order to ‘play nicely’ with other processes and the host environment, the approach taken here is to use infrequent calls to OS-level memory management routines in order to maintain a dynamic pool of memory, which is then managed using faster methods optimised for realtime usage.

### 2.2.2 Memory Management Model

The current approach taken in *FrameLib* relies on three levels of memory management:

The first of these employs thread-local free pools which hold a fixed number of memory blocks each.<sup>10</sup> Memory held in a thread-local free pool will only be allocated if a good match for the size required can be found (blocks cannot be split or combined). This code executes quickly, and ensures that memory reuse (and thus cache efficiency) will be high in networks with frames of fixed, or roughly similar length, as is expected in many scenarios, at least for localised parts of the network (e.g. a chain of objects processing FFT frames of fixed length). This also minimises thread synchronisation, which is required for the other levels of memory management.

Should the thread-local memory management fail, the allocator falls back on a second level of memory management. This is a two-level segregated fit allocator [9] of  $O(1)$  complexity, taken from <https://github.com/mattconte/tlsf>. This code has been modified to allow OS-managed memory pools (the final level of memory management) to be dynamically added and removed. This happens asynchronously in a low-priority thread based on monitoring of memory levels in the audio thread. Pools are added when the available memory falls below a given threshold and released when they have been unused for a significant period of time. The heuristics employed favour growing the amount of available memory (which occurs as fast as required) over shrinking it (which happens at a rate of one pool every ten seconds), in order that calls to the OS are minimised when overall memory usage is fluctuating.

Should the realtime allocator fail, the third level of memory allocation (the OS) is called directly to add a new pool to the available memory. This is a worst case scenario, as, ideally, memory will have been allocated ahead of time asynchronously, but in most cases this cost should be acceptable as direct calls to the OS should be infrequent.

The allocation process is summarised in Figure 1.

<sup>9</sup> All threads are guaranteed to progress.

<sup>10</sup> Although checking these blocks involves looping, as the number of items is fixed the time taken to allocate can be considered tightly bounded.

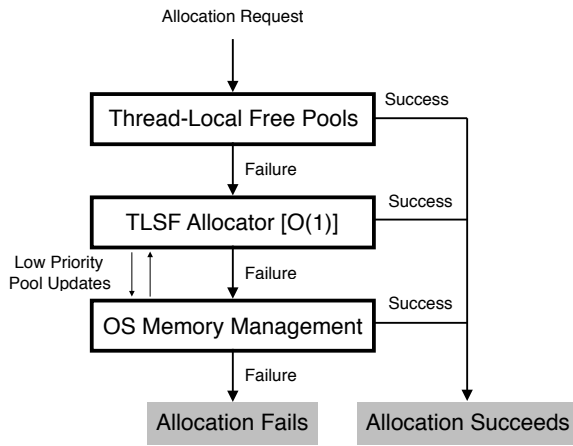


Figure 1. Memory Allocation Process.

### 2.2.3 Scheduling Deallocation of Outputs

As the length of frames in *FrameLib* is variable, all processing is performed out-of-place. This is at odds with common DSP network practice, where input and output memory is commonly aliased for a single unit generator to gain cache efficiency.<sup>11</sup> This design approach also simplifies coding objects that need to calculate all samples of their output with reference to the whole input.

As the output of each object in *FrameLib* is the input to one or more other objects, it is necessary to retain the memory allocated until it is no longer required for further calculation. Calculations can depend on inputs arriving at different times and it may be necessary to retain output memory for a considerable period of time, which cannot always be predicted in advance under the constraint of arbitrary realtime scheduling. However, it is desirable to release memory as soon as possible to gain efficient memory usage. A second method of dependency counting is employed here, which keeps track of the number of other objects dependent on each object's output. When this count falls to zero, memory is automatically released, and this count is updated based on information about when outputs are due to recalculate (in the case that this is known ahead of time) and the current time. An added improvement to this approach involves objects which are dependent on the output of only a single other object. These will decrease the count of the output on which they are dependent directly after calculating, as it is known that they will only ever update synchronously with their dependency. Thus, serial processing chains behave with a high degree of memory efficiency.

Although the aim is to release memory as early as possible, in a dynamic system, determining this point with complete accuracy is not always practical, and thus situations with parallelism in which it is unknown when objects are expected to recalculate ahead of time may exhibit some memory inefficiency. It is anticipated that this will affect a minimal number of cases, although further analysis of this situation should be carried out in the future.

<sup>11</sup> In other words input and output pointers refer to the same memory. Here the constraint that inputs and outputs have a single fixed size in a *block-based* model allows this approach, whereas the output size is not known in *FrameLib* until the inputs are processed, so this is impractical.

## 3. FRAMELIB OVERVIEW

*FrameLib* consists of three main elements outlined below:

### 3.1 C++ Classes

The underlying C++ classes provide basic functionality and core base classes for creating scheduler- and processor-type objects.

Key features include:

- Global classes that share data between objects
- A custom high-precision fixed-point time format<sup>12</sup>
- Real-time memory management
- Base classes for schedulers and processors
- Multichannel expansion support (*details below*)
- Parameter handling (*details below*)
- Common algorithms (RNG, sorting, etc.)

#### 3.1.1 Multichannel Expansion

This optional functionality allows multiple channels to be carried via a single connection between objects, and works by wrapping underlying single channel objects with a multichannel expander template class. Classes are also provided to concatenate channels into a single connection and split multichannel connections into individual channels. The expansion class creates multiple instances of the underlying object in order to deal with each channel, resulting in parallel copies of a DSP graph per channel. Mismatches between channel counts are handled by using modulo indexing according to the maximum channel count on each connection. An example usage might be a panner in which a single input audio channel can be connected via a multiplier to a multichannel set of coefficients. The resulting output will be the one channel for each of the coefficients multiplied against the mono input audio.

Whilst the most obvious usage might be multichannel work, the primary motivation for this feature was to allow easy creation of multiple resolution frequency-domain processing, in which a copy of the graph is required for each resolution of processing.

#### 3.1.2 Parameter Handling

Although numerical vectors are the primary data format in *FrameLib*, some scenarios call for parameters that can only take one value per frame or are set once at instantiation of an object. An example of this would be a mode setting (which must be applied over the whole vector), or a length parameter for an object that generates vectors of random numbers. *FrameLib* uses a second data type to handle these values such that multiple named parameters can be serialised and sent to an object simultaneously, either at the time of object creation, or in realtime via a single connection. Parameters may take a number of different data types (booleans, integers, doubles, enums, strings), and helper classes are provided for serialising and concatenating parameter data for the purpose of output.

<sup>12</sup> This 128-bit format provides 64 bits for sample-level accuracy and 64 bits for subsample divisions.

### 3.2 Objects

The library also provides a set of C++ classes for performing specific operations on vectors of samples. At the time of writing, this numbers around 90 classes. C++ class templates are provided for quick creation of unary and binary operations from unary or binary scalar functions, so that simple operators are trivial to construct. For the binary template, various strategies for operating on mismatched vector lengths are provided, including truncation to the shorter length, padding of either inputs or outputs, and the default wrap mode, that reads the shorter vector using a modulo index so that it is “wrapped” against the larger vector. Thus, control values in *FrameLib* can be efficiently calculated as scalar values (stored as frames with only a single item) and then applied across a vector.

A full list of available objects is beyond the scope of this paper, but functionality falls into the following categories:

- Basic mathematical operators
  - wrappers for `math.h`
- Vector operations
  - e.g. min, max, percentiles, average, sum, sort
- Scaling and control signal creation
  - including ramp and random number generation
- Spectral processing
  - e.g. FFT transforms, windowing, convolution
- Buffer reading
  - currently specific to the *Max* environment
- Time-domain filters
  - e.g. state-variable, one-pole, resonant
- Audio IO
  - gathering input frames incur associated latency
  - output incurs no latency

### 3.3 Environment-Specific Wrappers

Currently a single wrapper exists, targeting the *Max* environment. Adding additional wrappers is more complex than writing a simple object, but still a relatively contained task.<sup>13</sup> *FrameLib* places a minimal number of requirements on any host environment, key amongst which is that the host processing model must be *block-based*, with a single block size used across the network per DSP update (this may, however, vary over time). It is possible to wrap the objects either with or without multichannel expansion capabilities as required.

Environment-specific wrappers must support:

- Management of global objects
- Object creation and deletion
- Managing object connections
- Calling *block-based* audio methods where relevant

- Ensuring DSP dependency order within the host<sup>14</sup>
- Handling fixed value inputs

The *Max* wrapper equates each multichannel expanded *FrameLib* object directly with a max object, and uses messages sent through patch cords to manage connections, thus allowing the user to patch as normal. In this environment *FrameLib* can be considered another paradigm, in the same sense that messages, MSP signals and jitter matrices are all distinct data connection types that can be made within a single patch.

## 4. EXAMPLE APPLICATIONS

To demonstrate practical usages of *FrameLib*, two examples of realworld applications in *Max* are outlined below.

### 4.1 Spectral Dynamics Processing

Spectral dynamics processing is often applied within high-level environments as an STFT process with parallel per bin processing. However, for practical purposes, any signal that has been windowed will spread energy across all output bins, even if the input is a single sine wave. With a per bin approach, sidebands and spectral leakage are treated separately from the components to which they belong, resulting in audible artefacts. As far back as 1999, Jean Laroche discusses the treatment of what he terms “regions of influence” in which bins representing a single peak in the amplitude spectrum are grouped and processed together to avoid this issue [11]. This approach is also present as an option in Soundhack’s *Spectral Shapers*<sup>15</sup> plugins, but it is impractical using native *Max* objects.

A secondary, related problem is that the amplitude of sine waves on a bin frequency will appear higher than that of sine waves between two bins (where the energy is split between bins). Thus, even within bins close to a spectral peak, per bin values are a poor representation of the amplitude of spectral components. This can be solved by interpolating the peak amplitude of spectral peaks (see [12]) and using this as the value for the detector circuit.

Using *FrameLib* objects built to perform peak detection and interpolation, a scheme was constructed in which amplitude changes are applied to each “region of influence” based on the peak amplitude value alone. The peak detection object outputs a list of the peak amplitudes of all detected peaks (which vary in number from one spectral frame to the next), along with a list of which peak (by index) each bin belongs to. It is thus possible to map back and forth from one to the other with an appropriate object for performing lookup of one vector against another. The detection algorithm then operates on vectors only as large as the number of peaks in each frame, before the results are applied to a full spectral frame, resulting in more efficient processing, due to the reduced frame size for the detection circuit.

A third and final issue is the high variance of spectral power estimates, which leads to variations in the detected amplitudes in the input. This can be reduced by using a

<sup>13</sup> The *Max* wrapper totals around 900 lines of code.

<sup>14</sup> i.e. audio inputs to *FrameLib* objects must process before connected outputs back into the audio host environment.

<sup>15</sup> <http://www.soundhack.com/spectral-shapers/>.



multitaper analysis as the detector input to a spectral dynamics process (see [3] for info on the technique, or [13] for use in audio applications) leading to more stable behaviour for stochastic (noise-based) inputs. The multitaper analysis is performed directly on the time domain input to produce a power spectrum estimate, which is then used to modulate the frequency-domain amplitudes of a conventionally windowed version of the signal. Multitaper analysis is usually performed using multiple FFTs (one per taper) but it can also be applied for the cosine tapers using a single double-length FFT and zero-padding, and then applying the tapers in the frequency domain. In this case, a custom object was built to simplify the calculation of the multitaper power spectrum for the detection circuit, although this scheme could also be built using simpler *FrameLib* objects. Such a DSP graph that combines time domain signals with FFT processing of different lengths interchangeably and with correct time synchronisation is impossible using only native objects in *Max*. The main detector patch code is shown in Figure 2, including per threshold thresholds and attack/decay smoothing.

The combination of these improvements on a naive per bin approach is a higher-quality sounding result, with a substantially lower level of audible artefacts.

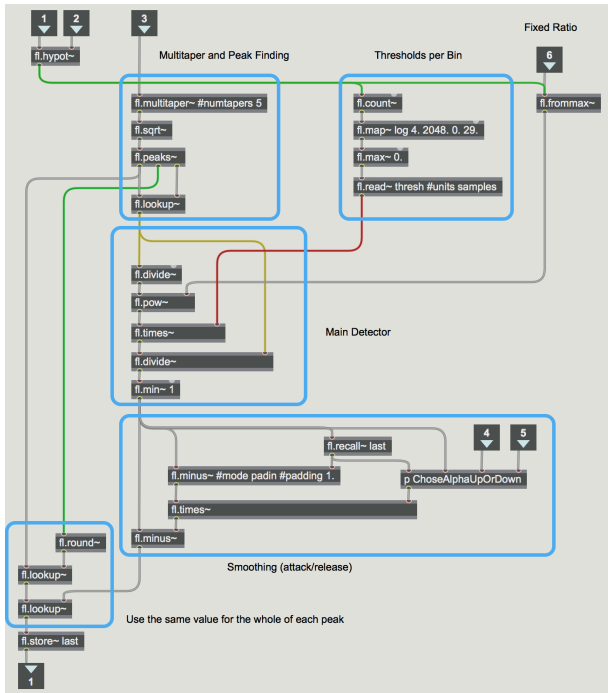


Figure 2. Spectral Dynamics Detector.

Multi-resolution spectral dynamic processing has also been prototyped using *FrameLib*. The process is simply to slice the input audio into a number of different lengths and concatenate these into a multichannel signal prior to input to the object network. The network is automatically copied by multichannel expansion to deal with each resolution separately. At the output, the channels are separated and appropriately delayed/filtered for recombination.

## 4.2 Granular Synthesis with Descriptor-driven Spatialisation

Granular synthesis algorithms can be created with relatively simple code in *FrameLib*, as there is no need for voice management. Each grain is simply processed in a single frame.

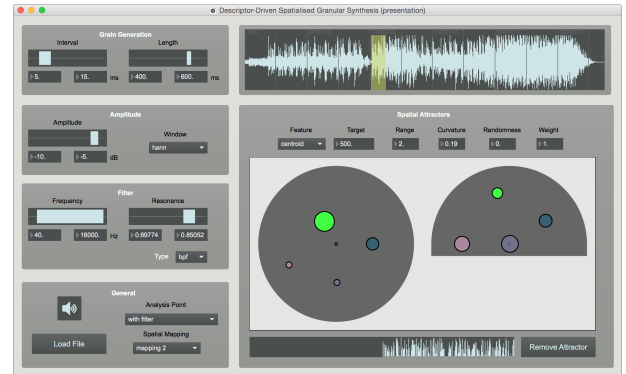


Figure 3. Descriptor-driven Spatialisation Interface.

This example concerns a more advanced application, in which the fact that each grain can be processed as a unit within *FrameLib* is used to enable analysis of the grain content, in order to drive a spatialisation algorithm. The interface to this patch is shown in Figure 3. A small number of audio descriptors were calculated, including the spectral centroid, spectral flatness measure and RMS value, all of which were possible simply using abstract vector based operations (mean, sum, etc.) alongside basic mathematical operators and an FFT transform with windowing. Thus, unlike the *zsa.descriptors* project [14] or the *descriptors~* object<sup>16</sup>, such low-level tasks are accessible from building blocks not specific to descriptor analysis within *Max*.

In this scenario, once a grain has been generated by the granular engine, the time-domain representation is sent to analysis, and the results used to determine the spatialisation of the grain. Spatialisation is applied in the time-domain, and the signal only enters the frequency domain for analysis, the result being a small set of descriptor values per grain produced synchronously with the time-domain audio frame. Positioning of each grain in a 3D space is calculated in relation to a set of attractors placed in a virtual representation of the physical space, each of which are assigned target values for a particular descriptor. Thus, when the value of the descriptor analysis is close to that of the attractor (for instance, a grain with a spectral centroid of 1160Hz in relation to an attractor with a spectral centroid target value of 1200Hz) the grain is spatialised closer to the attractor. If the value is further away from that held by the attractor (e.g. 200Hz in comparison to the same attractor as before) the grain is placed further away in space. Panning was achieved using an object that outputs multiplication coefficients using an algorithm based on DBAP [15] based on an input vector of 3 values representing x, y and z positions.

From an artistic viewpoint the overall result is to allow complex and perceptually coherent immersive spatial images from a relatively simple set of controls. However, in

<sup>16</sup> <http://alexanderjharker.co.uk/Software.html>.

the context of this discussion, this example serves to highlight the range of tasks that can be generalised by a frame-based approach, the level of sophistication possible with the *FrameLib* system in its present state, and the possibility of mixing different types of frame representation within a single DSP network.

## 5. FUTURE WORK

### 5.1 Object Set Expansion

Further objects are required to extend the basic object set and provide access to the fundamentals of advanced frequency domain processing (e.g. different frequency domain transforms).

### 5.2 Efficiency

One planned set of improvements to basic processing capabilities is the use of SIMD operations for basic mathematical operator objects (e.g. SSE or AVX instructions) to increase throughput. Another planned improvement is to complete the provision of multithreading support discussed in 2.1.3

### 5.3 Documentation

At the time of writing *FrameLib* is only documented via source-code comments. User and developer guides and tutorials are needed, along with an inbuilt help mechanism for binding to host environments' help systems.

## 6. CONCLUSIONS

*FrameLib*, an extensible library of C++ classes for realtime frame-based processing, has been presented, along with a wrapper to bind it to the *Max* environment. This library supports rapid prototyping and creative work with frames of arbitrary size, and arbitrary scheduling and utilises a novel scheduling technique based on dependency counting and controlling the flow of time through the DSP network. This core scheduling algorithm can be trivially made non-blocking and wait-free for multithreading support. *FrameLib* also provides dynamic memory management suitable for realtime use.

Example applications have been discussed, particularly in the context of enabling access to advanced spectral and granular processing techniques. The system has been shown to be capable of sophisticated results, without requiring complex code constructions from a user perspective. This is facilitated by the handling of low-level concerns of scheduling and memory management by the library, leaving users free to concern themselves with more creative matters. It is hoped that this will allow more widespread investigation of the frame-based processing from the DSP literature by the creative coding community.

## 7. REFERENCES

- [1] I. Daubechies, "The wavelet transform, time-frequency localization and signal analysis," *IEEE transactions on information theory*, vol. 36, no. 5, pp. 961–1005, 1990.
- [2] J. Princen, A. Johnson, and A. Bradley, "Subband/transform coding using filter bank designs based on time domain aliasing cancellation," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 12. IEEE, 1987, pp. 2161–2164.
- [3] D. J. Thomson, "Multitaper analysis of nonstationary and nonlinear time series data," *Nonlinear and nonstationary signal processing*, pp. 317–394, 2000.
- [4] G. Peeters, "A large set of audio features for sound description (similarity and classification) in the CUIDADO project," IRCAM, Tech. Rep., 2004.
- [5] M. Clarke, "Composing at the intersection of time and frequency," *Organised Sound*, vol. 1, no. 02, pp. 107–117, 1996.
- [6] T. Wishart, *Audible design: a plain and easy introduction to practical sound composition*. Orpheus the Pantomime, 1994.
- [7] N. Schnell and D. Schwarz, "Gabor, multi-representation real-time analysis/synthesis," in *Proceedings of the International Conference on Digital Audio Effects*, 2005, pp. 122–126.
- [8] N. Schnell, R. Borghesi, D. Schwarz, F. Bevilacqua, and R. Müller, "FTM – complex data structures for Max," in *Proceedings of the International Computer Music Conference*. ICMA, 2005, pp. 9–12.
- [9] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *Proceedings of the Euromicro Conference on Real-Time Systems*. IEEE, 2004, pp. 79–88.
- [10] T. Ogasawara, "An algorithm with constant execution time for dynamic storage allocation," in *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications*. IEEE, 1995, pp. 21–25.
- [11] J. Laroche and M. Dolson, "New phase-vocoder techniques for pitch-shifting, harmonizing and other exotic effects," in *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. IEEE, 1999, pp. 91–94.
- [12] J. O. Smith and X. Serra, *PARSHL: An analysis/synthesis program for non-harmonic sounds based on a sinusoidal representation*. CCRMA, Department of Music, Stanford University, 1987.
- [13] Y. Hu, "Subspace and multitaper methods for speech enhancement," Ph.D. dissertation, The University of Texas at Dallas, 2003.
- [14] M. Malt and E. Jourdan, "Zsa.Descriptors: a library for real-time descriptors analysis," *Proceedings of the Sound and Music Computing Conference*, pp. 134–137, 2008.
- [15] T. Lossius, P. Baltazar, and T. de la Hogue, "DBAP – distance-based amplitude panning," in *Proceedings of the International Computer Music Conference*. ICMA, 2009, pp. 489–492.